STFT Analysis Driven Sonographic Sound Processing in Real-Time using Max/MSP and Jitter

being a project submitted in partial fulfillment of the requirements for the Degree of Bachelor of Science

in the University of Hull

by

Tadej Droljc

October 2011

STFT Analysis Driven Sonographic Sound Processing in Real-Time using Max/MSP and Jitter

Tadej Droljc

October 2011

Preface

This dissertation was submitted in October 2011, that is just before Max 6 saw the light of the day. With Max 6 and accompanying Gen add-on some new options of spectral processing appeared. Some of these options, together with other interesting findings that I came across in the meantime, are presented in this updated paper in form of notes at the end of the sections 5.1.3, 5.2.3 and 5.5. There are also 4 new patches that were not the part of submission. These patches can be found in folder zzz_New_Patches.

At the moment I am working on a patch where all the accompanying patches and some new features are joined together in one large application. The spectrogram, that is rendered in 3D, can be read with 3 different reading positions, where each reading position is placed inside an independent moving processing area (see chapter 5.1.1 for the explanation of moving processing area). The project is unfortunately on hold, since I need a stronger computer and especially stronger graphics card to continue. Despite that I managed to carry out one performance, that can be seen at http://www.youtube.com/watch?v=LzBPr3Hc6CA&feature=youtu.be . Video and especially sound quality of the clip are extremely poor, since both were recorded with a camera.

Tadej Droljc, March, 2012

Abstract

This project explores the possibilities of analysis driven sonographic sound processing in real-time in graphical programming environment Max/MSP and Jitter. The central tool of the research is a phase vocoder whose FFT data is stored in a matrix. The Matrix has the form of a spectrogram that serves as a human–digital instrument interface on one hand and as a window into the world of digital signal processing on the other.

Sonographic processing is all about controllable and accurate data scaling and repositioning, hence these sorts of techniques are presented in the paper. This research also deals with the transfer of spectral processing from CPU to GPU and general methods for localization of sonographic manipulations.

Table of Contents

Chapter 1

Introduction

When in 1997 Christian Fennesz, Jim O'Rourke and Peter Rehberg formed probably the first EAI (electro-acoustic improvisation) laptop band (Warburton, 2009), stereo was a luxury. "During the first concerts due to limitations of the technology used many of samples had to be in mono", said Rehberg in a recent interview (2010). Today we are facing similar problems when performing real-time analysis driven sonographic spectral processing (ADSSP). Spectral processing of large matrixes can be extremely expensive for CPU. When using high quality analysis parameters even short samples are represented with huge amount of data and certain processing techniques demand either data reduction or a powerful computer.

ADSSP can be imagined as an alternative way of sampling. In samplers the sound is loaded in a memory and processed along the processing chain. The initial sample or the sound source remains intact during the processing. ADSSP on the other hand analyses the sound source and replaces its original time-domain representation with amplitude and frequency data needed for sound resynthesis. The copy of analysis data is later transformed by various combinations of

data scaling and repositioning and at the end presents a processed frequency-domain sound sample that can be played like an ordinary sample in classic or granular sampler. Since the analysis data can be visualized in the form of a spectrogram, the whole DSP procedure when using real-time ADSSP is visible and therefore gives an extra synesthetic dimension to signal processing.

The chosen tool used for analysis, data manipulation and resynthesis during the whole research is a phase vocoder with integrated spectrogram as a human–digital instrument interface. Apart from interactive role the spectrogram also serves as a window into the world of signal processing.

1.1 Summary of problem

Max/MSP and Jitter offer a great potential for real time analysis driven sonographic sound processing that is, considering the available documentation, only partially explored.

1.2 Aim of project

To study various existing and non-existing (undocumented) ADSSP techniques in Max/MSP and Jitter and categorize them on the basis of their underlying graphical processing methods and their basic functionality to establish a database of guidelines for methodological approach to ADSSP and further research.

1.3 Project scope

This project will focus on the design and development of a real time analysis driven sonographic processing patch in software Max/MSP and Jitter by means of integrating the spectrogram into the phase vocoder where the spectrogram will serve as an interface to observe and interact with the audio data. Various sonographic processing techniques will be explored through graphical manipulations of spectral data. The project will focus only on spectral processing of sampled existing sounds.

1.4 Methods

1. Understanding the phase vocoder through FFT study.
2. Examining existing ADSSP techniques.
3. Designing and implementing an ADSSP patch (phase vocoder with integrated spectrogram) in Max/MSP and Jitter that will serve as a basis for further experiments.
4. Testing various existing sonographic and non-sonographic analysis driven spectral processing techniques in real-time environment and analysing the potential benefits of real-time sonographic modulations.
5. Trying various non-existing (undocumented) ADSSP techniques.

6. Reducing examined ADSSP techniques to the level of graphical scaling, repositioning and masking and categorizing them under these conditions.
7. Additionally, categorizing examined ADDSP techniques according to their functionality and type of processing unit usage (CPU or GPU).
8. Conducting a database of ADSSP techniques in Max/MSP and Jitter that could serve as an introduction to ADSSP and also as set of guidelines for further studies.

1.5 Chapter overview

Chapter 2 begins by placing the real-time ADSSP in narrower context of sound synthesis and wider context of acoustic quanta sound model. It continues with the history of Fourier analysis and its implementations. The last sections of chapter 2 outline the history of ADSSP followed by the reasoning behind choosing the graphical programming environment Max/MSP and Jitter.

Chapter 3 explains the conceptual and mathematical background of phase vocoder though Fourier analysis. Later sections of chapter 3 include peculiarities of Short term Fourier transform (STFT), phase vocoder and spectrogram scaling when using available objects inside Max/MSP and Jitter.

Chapter 4 discusses the actual implementation of phase vocoder with sonogram as a spectral data storage and user interface in Max/MSP and Jitter.

Chapter 5 presents concepts and implementations of various real-time ADSSP techniques and categorizes them according to their functionality or underlying graphical processing methods.

The main sections of Chapter 5 are Optimization, Global transformations, Local transformations, Spectral interactions and FFT data processing on GPU. The more general section on local transformations also suggests specific Max or Jitter objects that were found as the most optimal for accurate and flexible localization of sonographic effects.

Chapter 6 concludes the research by a critical evaluation of the presented ADSSP techniques, the used programming environment and the real-time processing benefits in terms of live performance and studio use.

Chapter 2

Background

2.1 Graphical Modifications of Visually Presented Audio Data in the Context of Sound Synthesis

Sound synthesis based on graphical modifications of visually presented audio data spans through the whole universe of sound synthesis. With special methods that will be described in detail in later chapters one could recycle practically any concept of sound synthesis or processing that exist in time-domain – additive, subtractive, modulation, distortion or granular synthesis, convolution, delay, reverb or frequency shifting effects etc. By recycle we mean that the results obtained during processing in frequency domain can never sound the same as the results obtained from conventional sound synthesis methods. Therefore old concepts spring new results through different approaches. Beside that, many bizarre techniques impossible to implement in time domain become realizable with spectral modifications in frequency domain. Despite that, as we will see in the following chapters, the sound synthesis in question is not some kind of supreme technique for generating computer based sounds although it is extremely powerful. Everything has its price.

Following the terminology of Curtis Roads we will distinguish between graphic and sonographic sound synthesis. While both approaches are based on interpretation of graphical images as sounds, graphic sound synthesis is defined as drawing or transforming time-domain waveforms and sonographic sound synthesis is defined as drawing or transforming frequency-domain sonograms (2001).

Further distinction in sonographic sound synthesis can be drawn between spectral analysis, transformation and resynthesis of existing sounds on one hand and synthesis of drawn frequency-domain pictures on the other hand. Both techniques can be combined by adding some drawings to analysis based pictures.

This research will focus exclusively on spectral analysis, transformation and resynthesis of existing sounds. By transformation we mean scaling and dislocation of analysis data by various mathematical operations. On the other hand, spectral drawing is considered as a process of replacing the analysis data with non-analysis data or a process off adding the non-analysis data to analysis data. Following that definition we can say that this paper is only about analysis driven sonographic sound processing.

Additional categorization of ADSSP may arise from the difference between real-time and non real-time processing of spectral data. In non-real-time ADSSP the sound is a product of reading a static picture while in real-time ADSSP the sound is a derivate of reading a moving picture. Hence in real-time ADSSP the modulations of graphic parameters become the modulations of audio parameters. This enables the user an old fashion intuitive “tweaking” approach to sonographic sound synthesis like in majority of conventional sound synthesizers where each parameter change can be instantaneously heard. On the other hand all graphical (or sound)

interventions in non-real-time applications have to be rendered “off-line” and are therefore not a good choice for live performances. Also non-real-time ADSSP is not an appropriate tool for extremely slowly evolving sounds since that would demand extremely long sonograms. Real-time solution to this problem would involve looping and gradual modulations. As we will see later there are also other benefits of real-time ADSSP.

Two of the most common criticisms of improvised computer music performances are that there is no physical performance or presence on the stage and that the instrument or the interaction with an instrument is not visible. “Nobody knows what a sitting person behind the computer is doing”. Real-time ADSSP offers one of the possible solutions to the second problem. Sonogram or as we will see later, phase vocoder with sonogram as an interactive surface, can be played in real-time just like a conventional instrument and at the same time projected on the screen. Since sonogram can be quite easily intuitively understood it could offer a solution for deeper involvement of audience by seeing an instrument in live performances of non-pre-recorded computer music. From reasonable audio-visual mappings synaesthetic effects may occur in the audience (Jones & Nevile, 2005, p. 56).

2.2 Placing the Analysis Driven Sonographic Sound Processing in Wider Context That Made It Possible

> *The wave theory of sound dominated the science of acoustics until 1907, when Albert Einstein predicted that ultrasonic vibration could occur on the quantum level of atomic structure, leading to the concept of acoustical quanta or phonons… the visionary composer Edgard Varèse recognized the significance of this discovery …[but] The scientific development of acoustical quantum theory in the domain of audible sounds was left to the physicist Dennis Gabor.*
>
> (Roads, 2001, p. 54)

Until the middle of 20th century, time based descriptions of signals operated "with sharply defined instants of time" (Gabor, 1946) while on the other hand, frequency based descriptions (Fourier theorem) operated with "infinite wave-trains of rigorously defined frequencies" (Gabor, 1946). Even though these methods were mathematically undisputable they did not correspond with people's everyday experience of sound. As Gabor pointed out:

> *The orthodox [analysis] method starts with the assumption that the signal s is a function s(t) of the time t. This is a very misleading start. If we take it literally, it means that we have a rule of constructing an exact value s(t) to any instant of time t. Actually we are never in a position to do this…if there is a waveband W at our disposal…we cannot physically talk of time elements smaller than 1/W.*
>
> (Gabor, 1952)

> *If the term frequency is used in the strict mathematical sense which applies only to infinite wave-trains, a changing frequency becomes a contradiction in terms, as it is a statement involving both time and frequency.*
>
> (Gabor, 1946)

Gabor joined two opposite views into complementary time-frequency definition of audio signals. He presented to the world "time and frequency units…called quanta…[that are nowadays] called grains, wavelets, or windows, depending on the analysis system being used" (Roads, 1996, p.546). The concept of acoustic quanta is fundamental to the analysis of time-varying signals such as sound. All window based analysis are analogous to Gabor's groundbreaking method. His discoveries were ahead of the time and had a delayed impact on signal analysis.

Nowadays the world of sound analysis and spectrograms is dominated by STFT or Short-term Fourier transform that is a windowed analysis. A sonogram image is analogous to visual representation of the Gabor time-frequency matrix that will be discussed later in this paper. And "to manipulate the sonogram image is to manipulate the sound" (Roads, 2001, p. 269).

The story of STFT started at the beginning of the nineteenth century when a French mathematician and engineer Jean Baptiste Joseph, Baron de Fourier, discovered that "it was possible to represent an arbitrary mathematical function by a sum of (possibly infinite) simpler functions. The simpler functions which he chose for this representation were the elementary sine and cosine functions" (Wishart, 1996, p. 48). In 1807 Fourier presented his idea of Fourier transform (FT) to the French Academy of Sciences and was "severely attacked by established

scientists who insisted that such ideas were mathematically impossible" (Roads, 1996, p. 1075). His theory was published fifteen years later in 1822 in his work The Analytic Theory of heat (Fourier, 1822). The delay of publication reflects the controversy of the subject (Kahane & Lemarie-Rieusset, 1995, p. 14) although the same controversy lead also to the fact that the credit for the theory in question goes to Fourier. Recent discoveries in history of mathematics namely reveal that the Swiss mathematician Leonard Euler discovered an analogous method to FT in 1770 and that the German mathematician Karl Friedrich Gauss invented a similar algorithm to *Fast* Fourier transform (FFT) in 1805 – as Roads pointed out, FFT was re-invented in mid-1960s (1996, p. 1076). The reason why Gaussian algorithm remained unknown for so long lies in the fact that he never published it (Heideman, Johnson and Burrus, 1984, p. 14).

In the nineteenth century Fourier analysis was manually calculated. That was an extremely time-consuming task that demanded constant focus and a lot of patience. In 1878 Lord Kelvin (also known as Sir William Thomson) invented a first known mechanical harmonic analyzer (Thomson, 1878, p. 371) that was based on Fourier coefficients and an array of ball-and-disk integrators invention by his brother James Thomson (Reif, 2004, p. 13). Kelvin's gear-and-pulley device "calculated the amplitude and phase of the Fourier harmonics of solar and lunar tidal movements" (Reif, 2004, p. 13). According to Roads (1996) the most elaborated device of this kind was Michelson-Stratton harmonic analyzer from year 1898 that "was designed around a spiral spring mechanism … [and] could resolve up to 80 harmonics" (Roads, 1996, p. 1076).

In the twentieth century spectrum analysis or rather rough spectral estimation was carried out in a subtractive manner by a bank of analogue filters (an array of equally spaced or fixed-frequency band-pass filters distributed over frequency spectrum). Subtractive synthesis was also the way of resynthesizing the analyzed sounds and the original contraption was called the vocoder ("voice

coder") invented by Homer Dudley in 1928 at AT&T Bell Laboratories (Smith, 2010). First public demonstration took place at the 1936 World's Fair in New York City by a speaking robot (Roads, 1996, p. 197).

> *Synthesis-from-analysis systems were originally inspired by the desire to reduce the amount of data needed to represent a sound. Most of the initial work was done under the auspices of telecommunications companies with the aim of improving the efficiency of transmission.*
>
> (Dodge & Jerse, 1997, p. 220).

According to Dodge and Jerse, linear predictive coding and formant tracking were two methods that emerged with the rise of computers and dominated the world of speech analysis. Fourier analysis on the other hand produced too many unnecessary data for successful speech analysis-synthesis.

First digital implementation of FT took place in 1940s with the appearance of stored-program computers. At that time the price for high computational costs was colossal amount of computer time – "a scarce commodity in that era" (Roads, 1996, p. 1076). In following decades computers radically changed and gained some processing power. At the same time mathematicians elaborated Fourier's theory that lead to the discovery of Fast Fourier Transform (FFT) by James Cooley at Princeton University and John Tukey at Bell Telephone Laboratories (Cooley and Tukey, 1965). Enormous computations required for FT were significantly reduced by "clever reordering of the operations" (Dodge & Jerse, 1997, p. 244).

Despite the FFT, Fourier analysis remained “a numerically intensive operation, it was until the mid-1980s confined to scientific laboratories equipped with exotic hardware” (Roads, 1996, p. 1075). Nowadays, FFT or STFT is easily performed on “any” personal computer or laptop with negligible consumption of processing power. As we will see later, STFT is a windowed FFT and this is where the concepts of Fourier and Gabor come together.

There are many other techniques and theories for sound analysis that could be divided in Fourier-related or Non-Fourier related but will not be presented in this research paper.

2.3 History of Analysis Driven Sonographic Sound Processing and Overview of Its Most Important Implementations

A pioneering work in the field of spectral transformations was made by Gerhard Eckel at IRCAM in 1989 (Eckel, 1990). Eckel’s prototype implementation of frequency domain sound editor was called SpecDraw and the experiences obtained during the development served as a basis for the infamous IRCAM’S AudioSculpt. AudioSculpt is a graphical user interface for SPV (Super Vocodeur de Phase) that was developed by IRCAM’s Analysis/Synthesis team under the supervision of Gerhard Eckel (Battier, 1996, p. 1). In 2005, as the real-time processing of audio became even more feasible due to continuing increase of processing power of personal computers, AudioSculpt gained an option of real-time spectral processing. But because real time parameters in AudioSculpt can not be altered “on the fly” (Charles, 2008), “AudioSculpt’s real-time mode is meant for pre-flight experimentation rather than as an interactive performance tool”

(Bogaards & Röbel, 2005, p. 7). Considering the "advanced phase vocoder algorithms…frequency-bin independent dynamics processing…[customization of] advanced parameters for the FFT analysis… …calculation of spectral envelopes" (Charles, 2008), accurate pencil tool, fundamental frequency estimation, chord sequence analysis, partial tracking, cross synthesis, high quality time stretching with transient preservation (Bogaards & Röbel, 2005) etc., AudioSculpt is one of the most powerful spectral processing tools available nowadays.

Another extremely powerful analysis driven spectral sound processor and synthesizer (it also enables to importing and exporting pictures in standard graphical format) is MetaSynth from U&I Software company (Wenger & Spiegel, 2006). The advantage of MetaSynth is that it offers a vast palette of drawing tools and graphical effects that can be even extended by exporting sonographic images to software like Adobe's Photoshop. The weakness of Metasynth is that is does not offer a control over phase spectrum and that it can not be used as a performance tool due to its inability to process sound in real-time.

Another interesting piece of software comes from Michael Klingbeil (2005). "Rather than represent the analysis data as a sorted list of time frames, SPEAR uses a list of partials which are represented by breakpoint functions of time versus amplitude, frequency, and phase" (Klingbeil, 2005).

Widely used real-time environments such as Pure Data, Max/MSP/Jitter or Reaktor offer a huge potential for real-time sonographic spectral processing. We will focus on Max/MSP and Jitter since it is a platform of choice for this research. Two main contributions to Max/MSP community were made by Luke Dubois and Jean-Francois Charles. Luke Dubois's groundbreaking patch jitter_pvoc_2D.maxpat set the fundaments for implementation of phase

vocoder with spectrogram as an interface. On the other hand Jean-Francois Charles took Dubois's work further by developing some new and advanced matrix-based spectral processing techniques (Charles, 2008).

2.4 The Reasons Behind Choosing Max/MSP and Jitter

Phase vocoder is en extension of STFT that produces two-dimensional data. In Max/MSP, two-dimensional data can be stored only in one-dimensional array called *buffer~.* That can be imagined as columns of a 2-D matrix lying horizontally in succession in 1-D buffer.

On one hand it is very counterintuitive to work with 2-D data in 1-D space and on the other hand video, pictures and of course spectrograms do not exist in only one dimension. Despite that, most of the phase vocoders implemented in Max/MSP store their spectral data in stereo buffers (one buffer for amplitude and second for phase information).

> *Jitter, a software package first made available in 2002 by Cycling '74, enables the manipulation of multidimensional data in the context of the Max programming environment. An image can be conveniently represented by a multidimensional data matrix…the general nature of the Jitter architecture is well-suited to specifying interrelationships among different types of media data including audio.*
>
> (Jones & Nevile, 2005, p. 55).

Max/MSP also provides very handy and easy to use objects based on Fourier analysis. Especially convenient is object *pfft~* that draws a clear distinction between time and frequency domain by placing frequency domain in a special *subpatcher*. Using STFT or building a phase vocoder in Max/MSP/Jitter is also comprehensively covered by official Cycling ’74 Help and Documentation and various other academic papers and implementations (patches).

The main problem when working with phase vocoder in any programming environment is the complex theory behind the phase vocoder itself. In order to understand the fundaments of phase vocoder one needs to deeply understand STFT. While it is possible to experimentally discover a new ADSSP technique it is impossible to scientifically evaluate it and develop it further.

Chapter 3

From Fourier Transform to Phase Vocoder and Sonogram

3.1 Introduction to the Concepts of Fourier Analysis and Its Offspring

Fourier analysis is based on a model that any periodic signal can be represented as a sum of harmonically related sinusoids, each at a different amplitude, frequency and phase. Behind this relatively simple concept lies a very complex mathematical theory that will be covered to some extend in this paper. We will try to include only as much mathematics as needed to explain and understand the values and techniques we will be working with. At the same time Fourier analysis is not the subject of our study but merely an unavoidable and extensive starting point.
Calculations of various layers of mathematical abstractions of Fourier analysis-resynthesis are in Max/MSP encapsulated in objects such as *fft~, ifft~, fftin~, fftout~* and *pfft~* will be explained. Hence knowing all the mathematics behind Fourier analysis is not required when performing spectral processing in Max/MSP.

According to Roads (1996, p. 1086), Fourier theory says that any signal of infinite length can be represented with a Fourier transform (FT) spectrum that spans from 0Hz to + and – infinity, as can be seen on figures 3.3 and 3.4. When periodic signal is presented in digital form one can use a formula known as Discrete Fourier Transformation or DFT. Ideally DFT is used on short time slices that should be of exactly the same length as a period of the analyzed signal. In other words, one cycle of the input waveform should fit exactly in the analysis time slice (see figure 3.1). When using DFT on real sounds that are usually non-periodic and of unknown frequency, DFT should be used on successive time slices (Dobrian, 2001, MSP Tutorial 25). When visualizing this process the result are constantly changing spectral snapshots (dynamic spectral analyzer) that gives us a sense of varying frequency content of the signal. In case that analyzed time slices consist of 2^n samples, faster version of DFT should be used - FFT or Fast Fourier Transformation. But unfortunately FFT introduces the same problem as DFT – it works decently only when analyzing exactly one period of the waveform or its integer multiple. When these conditions are not provided various artifacts may appear and in order to reduce them the signal should be windowed with overlapping window functions (figure 3.2). "This kind of processing using overlapped, windowed time slices is known as Short Term Fourier Transform (STFT), and is the basis for frequency-domain audio processing" (Dobrian, 2001, MSP Tutorial 25). STFT also presents the core of Phase Vocoder that is the central tool of this research.

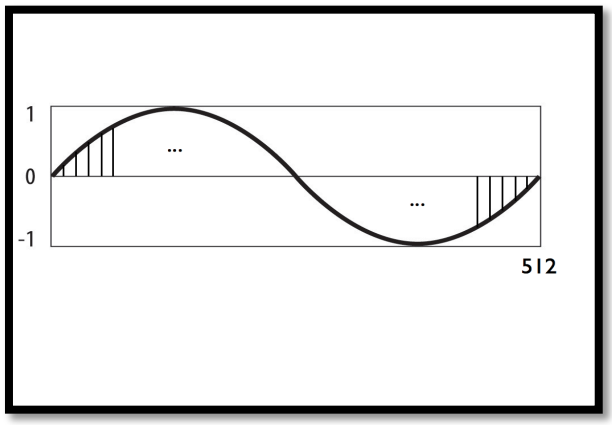

Figure 3.1 512 samples long time slice of 88.13Hz sine wave. Period of 88.13Hz sine wave equals 11.61ms or 512 samples at the sampling rate 44.1KHz (original).

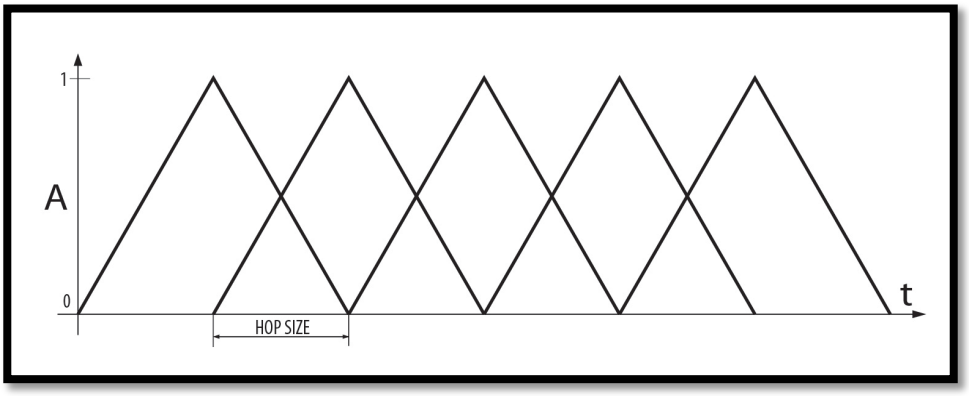

Figure 3.2 The process of windowing with overlap factor 2 (original).

3.2 Explaining the Fourier Analysis

3.2.1 Fourier Series

Periodic function in a form of Fourier series can be written as:

$$x(t) = C_0 + \sum_{n=1}^{\infty} C_n \cos(n\omega_0 t + \theta_n) \qquad (eq.3.1) \text{ (Roads, 1996, p. 1085)}$$

As mentioned earlier, periodic functions or signals can be expressed as a sum of harmonically related sinusoids, with magnitude C_n (magnitude is an absolute value of amplitude), phase θ_n frequency $\omega_n = n\omega_0 = 2\pi/T$, where T is a period of function $x(t)$. "The first sinusoidal component [$n = 1$] is the *fundamental;* it has the same period as T" (Roads, 1996, p. 1085). Since n is an integer, the sinusoidal functions $\text{Cos}(n\omega_0 t + \theta_n)$ inside the infinite sum occur at frequencies that are integer multiples of the fundamental. That means they are in harmonic relationship. Or as Smith pointed out (2008, Periodic Signals Chapter), all harmonic sounds are periodic in time domain. That given, most tonal instruments produce periodic signals either in sustain phase of their temporal sound development or at least in short time segments

As we will see later, fundamental frequency of STFT is determined by the size of the analysis window that can be interpreted as an imposed imaginary period on non-periodic signal. Hence the STFT spectrum is a *harmonic* spectrum of that fundamental frequency.

Fourier series summation formula does not tell us how to set the coefficients C_n (magnitude) and θ_n (phase) for an arbitrary and possibly stochastic function. According to Roads (1996, p. 1085), the problem can be solved with Fourier Transform.

3.2.2 Fourier Transform

At this point we are still in the world of analogue signals which means that we are dealing with continuous functions or continuous-time signals. "Fourier's theory says that $x(t)$ [figure 3.3] can be accurately reconstructed with an infinite number of pure sinusoidal waves of different amplitudes, frequencies, and initial phases. These waves make up the signal's Fourier transform spectrum [figure 3.4]" (Roads, 1996, p. 1086).

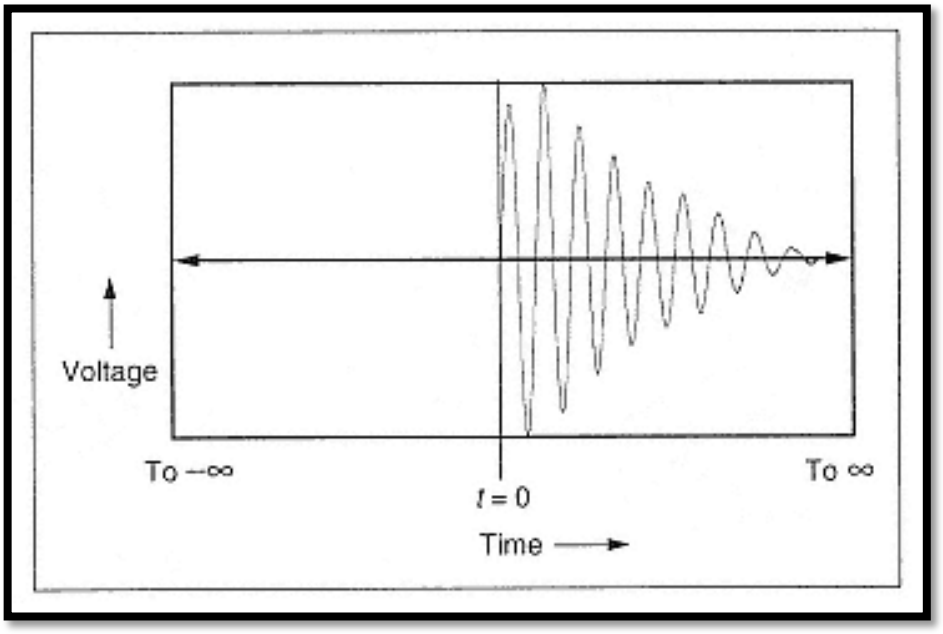

Figure 3.3 Continuous-time signal $x(t)$ of infinite length (Rhoads, 1996, p. 1085).

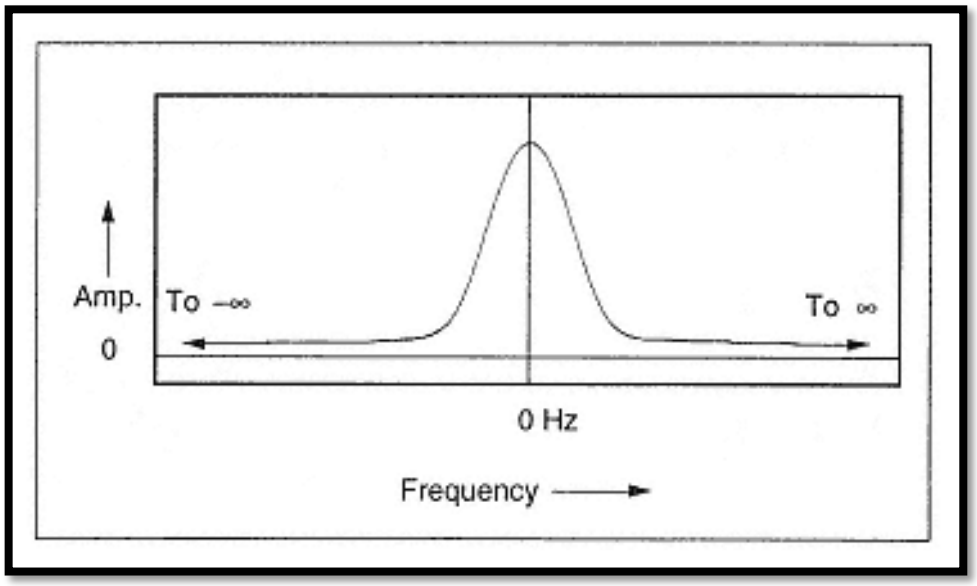

Figure 3.4 Magnitude spectrum after Fourier transform of the input signal $x(t)$ in figure 3.3 (Rhoads, 1996, p. 1086).

Fourier transform or Fourier integral is defined as

$$X(f) = \int_{-\infty}^{\infty} x(t)e^{-j\omega t}dt. \qquad (eq.3.2). \quad \text{(Roads, 1996, p. 1087)}$$

Before we continue we should mention that we are using an engineering nomenclature for $\sqrt{-1}$ that is *j* as opposed to *i* that is used in mathematics. Symbol *i* in engineering is used for electrical current. Also we will be ignoring the negative Fourier spectrum through the rest of the paper, because it is not crucial for the understanding of Phase Vocoder, it has no physical significance and is automatically trimmed off by Max/MSP object *pfft~* that is being used in our implementation. Negative or mirrored Fourier spectrum can be seen on figure 3.4.

Fourier integral as every other integral, represents the area between the function and the abscise. In our case, that area is filled with infinite amount of pure sine tones. In order to understand the equation 3.2 on a deeper level we first need to take a look at some alternative ways of denoting the circular functions.

3.2.3 The Link Between Complex Numbers, Circular Functions and Exponential Functions

"Complex numbers are central to signal processing because they act as a bridge to the circular functions" (Roads, 1996, p. 1077). Complex numbers are like circular functions also presented on a (2D) plane since they have a real and imaginary part. In next section of this paper we will show various ways of denoting the sinusoidal function and prove that Fourier integral consist of multiplication of input signal by (infinite amount of) sine waves. Following that we will try to explain the logic and purpose of such doing.

As we can see from figure 3.5 a complex number in 2D space can be written in Cartesian coordinate system (x coordinate, y coordinate) or in polar coordinate system (distance from center point, angle). Complex number Z can also be perceived as a vector and written as a sum of real and imaginary part:

$$Z = A + jB \qquad (eq.3.3).$$

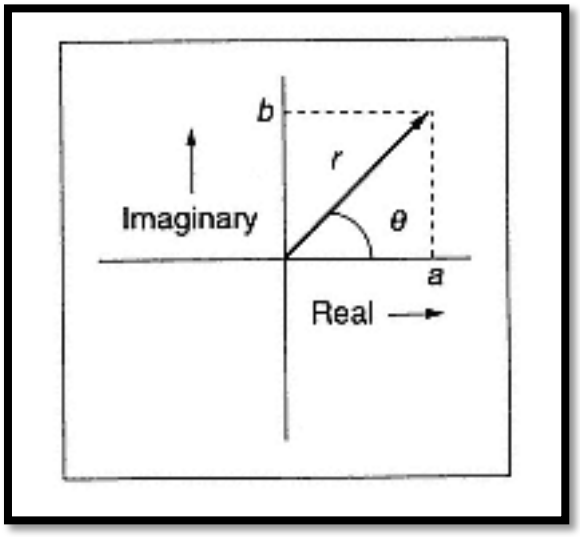

Figure 3.5 Complex number presented with Cartesian coordinates as point (a,b) and with polar coordinates as point (r, θ) where r is a distance from center and θ is the angle between vector and abscise (Roads, 1996, p. 1077).

From figure 3.5 can also be derived:

$$sin\ \theta = \frac{B}{r};\quad cos\ \theta = \frac{A}{r} \qquad (eq.3.4)$$

But since audio signals exist only in time we need to add a third dimension (time) to complex numbers in order to become useful in musical signal processing (fig. 3.6).

Now our angle θ becomes ωt where ω is angular speed ($\omega = 2\pi f$) and t stands for time. Equation $\theta = \omega t$ can be also seen as a polar version of more familiar equation $s = vt$ where s is distance and v is speed. In polar world distance becomes angle and speed becomes angular speed.

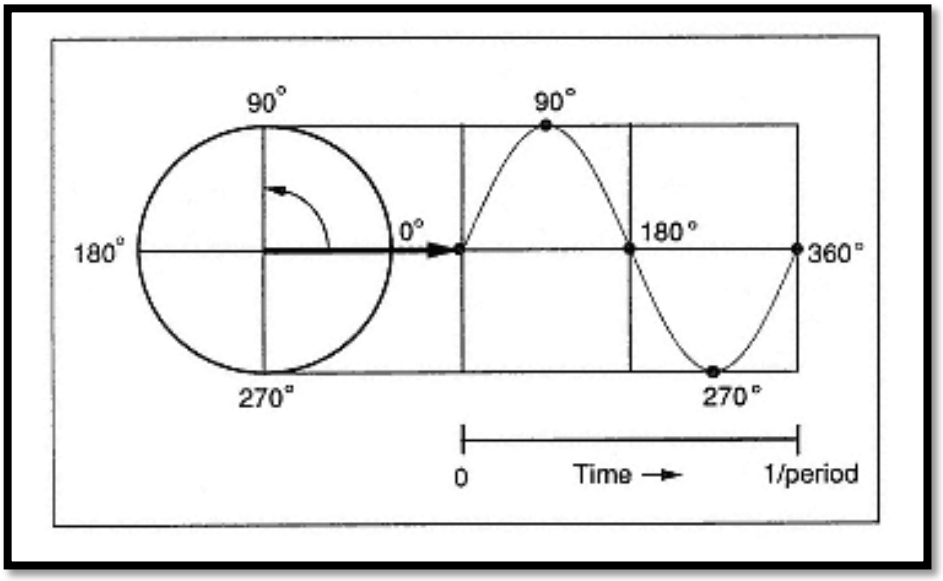

Figure 3.6 The projection of a rotating vector on a vertical axis is travelling vertically up and down between values 1 and -1. This projection is a function $sin\theta$. When time dimension is added θ becomes ωt and $sin\theta$ becomes $sin\omega t$. (Roads, 1996, p. 1079).

Another thing that changes in time driven 3D world is parameter B. From eq. 3.4 we can derive:

$$B \rightarrow signal(t) = y(t) = rsin(\omega t) \qquad (eq. 3.5)$$

Symbol r now represents magnitude of a sinusoidal function. Considering the initial phase ϕ of a sine wave:

$$y(t) = rsin(\omega t + \phi) \qquad (eq. 3.6)$$

"The Fourier transform uses one of the most common tricks of engineering mathematics: the representation of a sinusoidal function as a sum of a sine and a cosine at the same frequency but with possibly different amplitudes" (Roads, 1996, p. 1081).

$$y(t) = rsin(\omega t + \phi)$$

$$y(t) = r(sin\omega t \cdot cos\phi + sin\phi \cdot cos\omega t) \quad * sin(\alpha + \beta) = sin\alpha \cdot cos\beta + sin\beta \cdot cos\alpha$$

$$y(t) = r\left(\frac{A}{r} sin\omega t + \frac{B}{r} \cdot cos\omega t\right) \quad * from\ eq.3.4$$

$$y(t) = Asin\omega t + Bcos\omega t \qquad (eq.3.7)$$

(Mathcentre, 2004)

Sinusoid from equation 3.7 has a magnitude $\sqrt{A^2 + B^2}$ and phase shift $\phi = arctan\frac{B}{A}$ as explained on Mathcentre web page (2004).

But there is jet another way of representing the sine function. The most common way of denoting sine wave in engineering has actually a form of complex exponential function. According to Roads (1996, p. 1082), this representation makes the algebra of signal manipulation much easier. The relationship between circular functions, complex numbers and exponential functions were discovered/defined by the Swiss mathematician Leonard Euler.

From previously mentioned equations

$$Z = A + jB\ ;\ \ sin\,\theta = \frac{B}{r};\ \ \ cos\,\theta = \frac{A}{r}\ \ ;\ \ \theta = \omega t$$

and considering $r = 1$ in unit circle (see fig. 3.7), it can be concluded:

$$z = cos\theta + jsin\theta \qquad (eq.3.8)$$

The derivate of a function represents a change in the function with respect to its variable. If we derive z:

$$\frac{dz}{d\theta} = -sin\theta + jcos\theta$$

$$\frac{dz}{d\theta} = j^2 sin\theta + jcos\theta$$

$$\frac{dz}{d\theta} = j(cos\theta + jsin\theta)$$

$$\frac{dz}{d\theta} = jz$$

$$\frac{dz}{zd\theta} = j$$

$$\ln(z) = j\theta + C$$

Looking back at eq. 3.8 we notice when $\theta = 0,\ z = 1$ which means that $C = 0$. Hence

$$\ln(z) = j\theta$$

$$z = e^{j\theta}$$

$$e^{j\theta} = z = cos\theta + jsin\theta$$

$$e^{j\omega t} = cos\omega t + jsin\omega t \qquad (eq. 3.9)$$

(Ward, 1994)

$$e^{-j\omega t} = cos\omega t - jsin\omega t$$

$e^{-j\omega t}$ from Fourier's integral (eq. 3.2) is therefore a sinusoidal waveform.

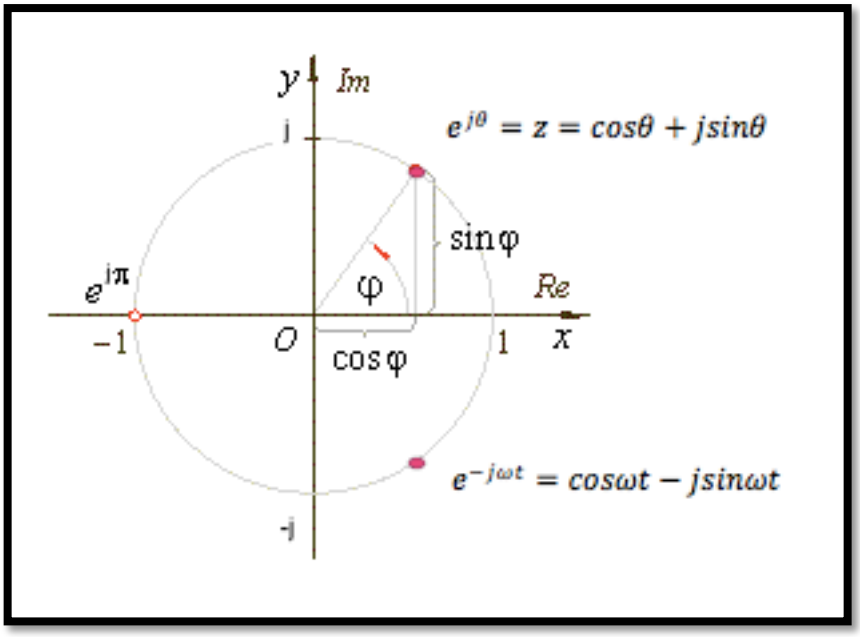

Figure 3.7 Complex exponential function (original)

Now that we know what the symbols in Fourier integral mean we can further investigate why they are in the integral in the first place. Why would multiplication of sinusoidal function with the input signal tell us anything about the spectrum of incoming signal? As we will see later, sinusoidal function acts as a detector for specific harmonic sinusoidal component in the input signal and gives us also the information about component's amplitude and phase.

3.2.4 Magnitude Detection in Fourier Transform

> *The FFT is itself a periodic, harmonic-rich entity, and it detects which of its own many virtual sinewaves are also present, strongly or weakly, in the input signal by comparing its own virtual signal with the input signal. It does this form of 'pattern matching' by a process of multiplication.*
>
> (Dobson, 1993).

Multiplication of (two) signals is called ring modulation and that is what is continuously going on in FFT. FFT basically exploits the laws of ring modulation when one of the signals is a sinusoidal function, like $e^{-j\omega t}$. Following the explanation and terminology of Nigel Redmon (2002) we will continue with one very useful property of sine waves. Redmon's explanation is only conceptual and does not include any mathematics.

3.2.5 Magnitude Detection in Special Cases

If all the harmonics in the input signal would have the same phase as is the phase of the sine part of $e^{-j\omega t}$, the cosine part of $e^{-j\omega t}$ would not be needed at all. There would be no need for complex numbers in Fourier analysis and the Fourier integral could be transformed like this:

$$X(f) = \int_{-\infty}^{\infty} x(t)e^{-j\omega t}dt \quad \rightarrow \quad X(f) = \int_{-\infty}^{\infty} x(t)sin\omega t\, dt \quad \text{(original)}$$

Therefore let us for now imagine that we have only $sin\omega t$ in Fourier integral instead of $cos\omega t - jsin\omega t$ and that the incoming signal is also $sin\omega t$ (no phase difference). Hence we have a multiplication of two equal sine waves in the integral. Now let us also forget the integral for a while and concentrate only on ring modulation of those two sine waves (fig. 3.8).

DC offset or average (mean) of any sine wave is always zero. In other words, if we imagine the area between sine wave and the abscise, the area above the abscise is exactly the same as the area below the abscise. The mean is therefore zero. Now if we multiply two sine waves together with the same phases and frequencies (amplitudes could be different) the mean of the result is proportional to the amplitudes of the multiplied sine waves (fig. 3.8). On the other hand the mean equals zero if the frequencies of the sine waves are not identical (fig. 3.9). This was experimentally tested in Max/MSP (fig. 3.8 and fig. 3.9).

If the wave that we are testing is called *target* and the wave that we use to make a test is called *probe* then the mean of the multiplication (result = probe * target) reflects the dependency to the target's amplitude (when freq. of probe and target is the same). In example presented on fig. 3.8 the amplitude of probe was 1.0, the amplitude of target was also 1.0 and the mean or DC offset of the result was exactly 0.5. If we would do another test with target's amplitude 0.5 the DC offset of the result would be 0.25. As the amplitude of the target is lowered DC offset of the result is also lowered by the same percentage. That reveals the linear relationship between the amplitude of the target and the mean of the result. In Fourier integral, amplitude of all probes is always 1.

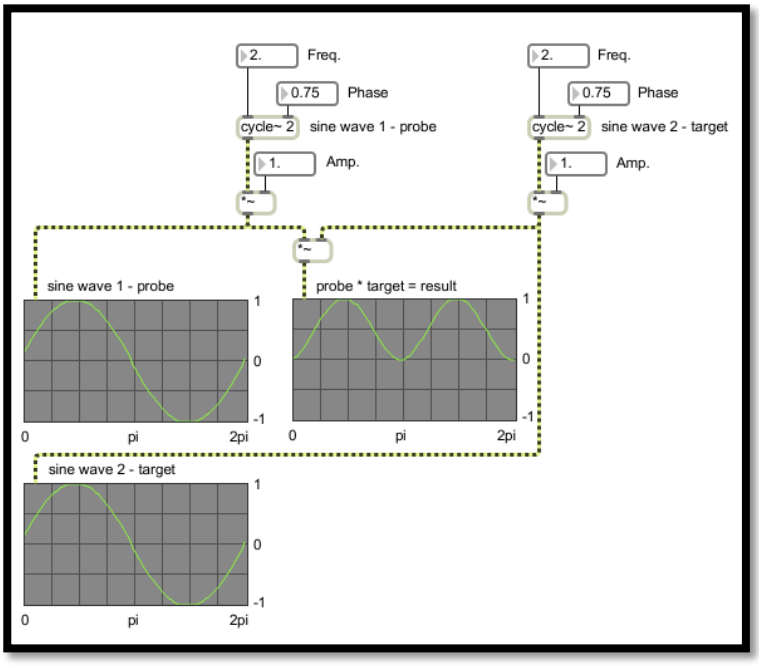

Figure 3.8 Multiplication of two sine waves (probe and target) at the same frequency - the mean or DC offset of the result is always bigger that 0 (original).

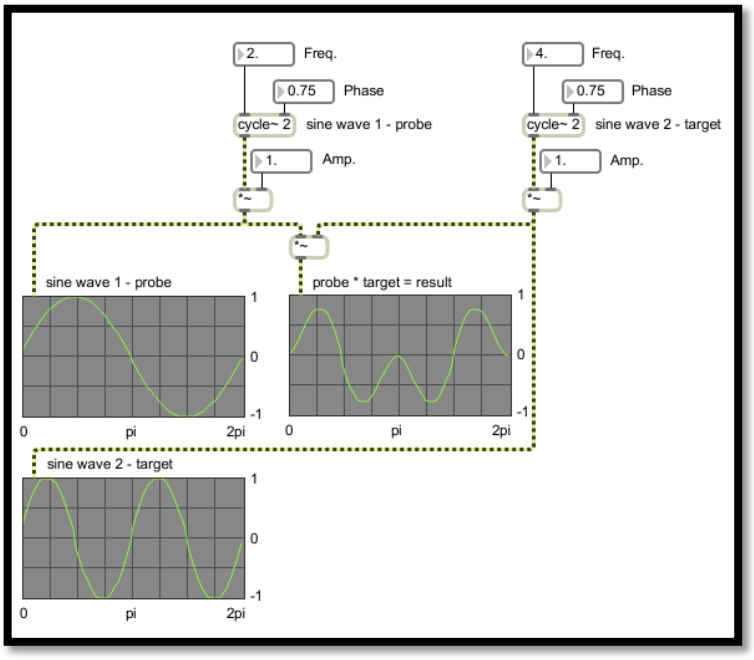

Figure 3.9 Multiplication of two sine waves (probe and target) at different frequencies - the mean or DC offset of the result is always 0 (original).

The resulting function $sin^2\omega t$ if using Euler's relations equals $0.5 - 0.5cos2\omega t$. A proof for this can be found on Simon Bramble's web page (Branble). As we can see the equation confirms what is obvious from fig. 3.8 – the resulting frequency is doubled ($2\omega t$) and the amplitude and DC offset is half the target's amplitude. This is true for any amplitude:

$$sin\omega t \cdot sin\omega t = 0.5 - 0.5cos2\omega t$$

$$\frac{1}{x} sin\omega t \cdot sin\omega t = \frac{0.5}{x} - \frac{0.5}{x} cos2\omega t$$

Scaling the target's amplitude with factor x scales DC offset of the result and amplitude by the same amount.

3.2.6 Maximum Magnitude Values

One of the most confusing things in FFT is understanding the magnitude values after the analysis. Maximum magnitudes that can be obtained from the analysis are always half the window size if using square windows and maximum magnitude can be obtained from a sine wave as an input signal where all the energy is concentrated at exactly one frequency. The concept of window will be explained later when we will come to the discrete digital world. For now we should think of wave's period instead of the window that is 2π in our example. See fig. 3.8.

For now let us still stay in the world where probe and target has the same phase since this does not change the maximum possible magnitude values. But to explain the magnitude values we will need to bring in the integral.

Magnitude values are the result of integration since Fourier transfer is an integral. Integral represents the area between the function and the abscise. This area is in the case of sinusoidal function the same as DC offset.

On fig. 3.8 it seems that the area between the resulting function $sin^2\omega t$ and the abscise represents a half of the rectangular area above the abscise. Since the area of the rectangle above the abscise is 2π * 1 = 2π we could intuitively conclude that the definite integral of $0.5 - 0.5cos2\omega t$ between 0 and 2π equals π (2π / 2 = π) that is half the wave's period. And mathematical truth is not far from that. Let's calculate first the indefinite integral:

$$\int (0.5 - 0.5cos2\omega t)\, dt = 0.5t - \frac{0.25sin\omega t}{\omega} \quad (eq.3.10)$$

The second part of the result will be always near 0 because $sin\omega t$ can never be greater than 1. That means that our assumption was correct: $\int (0.5 - 0.5cos2\omega t)\, dt \cong 0.5t$. Time t or borders of definite integral in our case go from 0 to 2pi which results in 0.5 * 2π – 0.5 * 0 = π.

If we measure time in digital samples as it is common in musical signal processing then our definite integral (for a very short time slice) would be still half the wave's period. For example if the period is 512 samples long:

$$\int_0^{511} (0.5 - 0.5cos2\omega t)dt \cong 256 \quad (eq.3.11).$$

Now we see that when analyzing time slices that fit exactly to the wave's period the resulting magnitude is always half the time slice. Later in the DFT chapter we will see that target's period and time slice has to be always the same size.

Upper integral is not perfectly correct from a purist mathematical perspective because we are using samples as continuous time units - samples are discrete time units and should be denoted with n instead of t. As we will see later x(t) will become x[n] and $\int$ will become $\sum$ in the discrete time world of DFT. Also the result (magnitude) will be exactly and not approximately 256. But to avoid the confusion at this point symbol t and round brackets were used. In STFT chapter this example will help us understand the maximum magnitude values that can be obtained from Fourier analysis using different window shapes and sizes.

Hence if the phase and frequency of probe and target are the same the magnitude does not have to be calculated by formula $\sqrt{Re^2 + Im^2}$ since there is no cosine part and therefore no need for complex numbers. The magnitude is simply the result of our simplified integral.

The next thing is to prove that our simplified integral can really be used when probe and target have the same phases and frequencies.

3.2.7 Fourier Integral of the Incoming Sine Wave

If we take a look again at the Fourier integral we see that $x(t)$ which is $sin\omega t$ in our case is not multiplied only by $sin\omega t$ but $e^{-j\omega t}$ that is $(cos\omega t - jsin\omega t)$. The result of Fourier integral is

therefore always a complex number that consists of real and imaginary part. But if we write $X(f) = \int_{-\infty}^{\infty} x(t)e^{-j\omega t}dt$ like this

$$X(f) = \int_{-\infty}^{\infty} x(t) \cdot (cos\omega t - jsin\omega t)dt$$

$$X(f) = \int_{-\infty}^{\infty} x(t) \cdot (cos\omega t)dt \; - \int_{-\infty}^{\infty} x(t) \cdot (jsin\omega t)dt$$

and insert $sin\omega t$ as an input signal $x(t)$ and at the same time consider that DFT (next chapter) is calculated for time slices long exactly the integer multiple of a period of target's sine wave, we see that the real part equals 0:

$$X(f) = \int_{0}^{2k\pi} sin\omega t \cdot (cos\omega t)dt \; - \int_{0}^{2k\pi} sin\omega t \cdot (jsin\omega t)dt \qquad k = integer$$

$$X(f) = \int_{0}^{2k\pi} sin\omega t \cdot (cos\omega t)dt \; - j\int_{0}^{2k\pi} sin\omega t \cdot (sin\omega t)dt$$

Real part of $X(f)$:

$$sin\omega t \cdot cos\omega t = \frac{1}{2} sin2\omega t$$

$$\int_{0}^{2k\pi} (\frac{1}{2} sin2\omega t)dt = 0 \qquad for\ all\ k$$

Therefore Fourier integral for incoming signal $sin\omega t$ in the time span of $2k\pi$ equals:

$$X(f) = -j\int_0^{2\pi} sin^2\omega t\, dt = -j\int_0^{2\pi} (0.5 - 0.5cos2\omega t)\, dt$$

The only difference between this result and the result obtained from eq. 3.10 or eq. 3.11 is $-j$. We used integral 3.11 for the calculation of magnitude and since magnitude is calculated as $\sqrt{Re^2 + Im^2}$, $-j$ can be avoided. For our case $\sqrt{Re^2 + Im^2} = \sqrt{0^2 + Im^2} = Im$. In other words the magnitude is a direct product of integral as shown in chapter 3.2.6. Hence the cosine or the real part of $e^{-j\omega t}$ does not have any role at magnitude detection and calculation when the phase of the target is the same as the phase of the probe.

3.2.8 Magnitude Detection in Harmonic Signals

So far we have seen the criteria and mechanism for detecting the frequency and amplitude if a target is a sine wave. In case that DC offset of ring modulation equals 0 that means no match - target frequency and probe frequency are different. But "the best part is that the target need not be a sine wave. If the probe matches a sine component in the target, the result's average will [also] be non-zero" (Redmon, 2002). This statement was experimentally tested in Max/MSP. The harmonics of a square wave were found using the described technique (fig. 3.10)

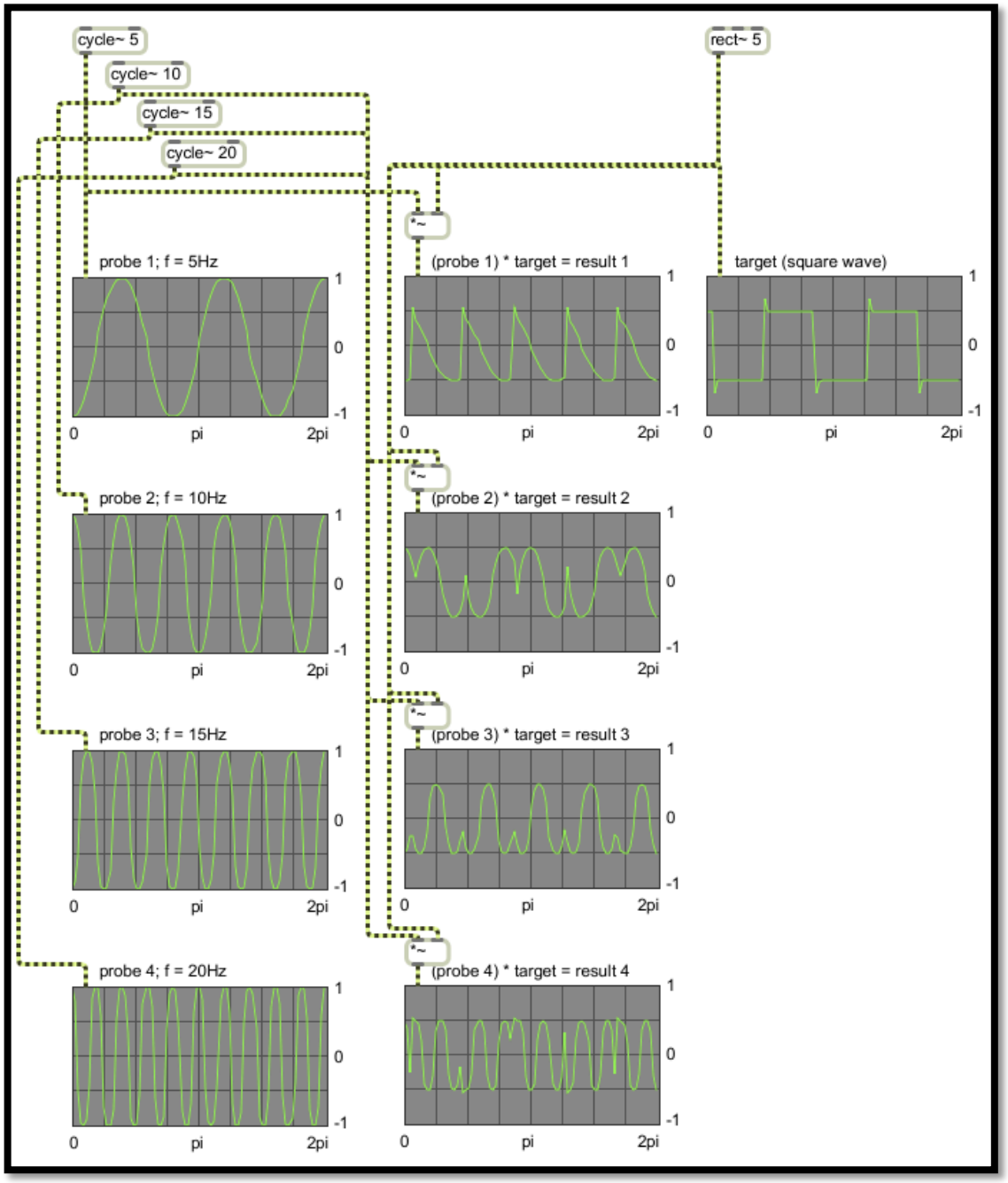

Figure 3.10 Rectangular waveform (target) at 5 Hz multiplied by 4 sine probes at harmonic frequencies 5Hz, 10Hz, 15Hz and 20Hz (original).

On fig. 3.10 we see a rectangular waveform or square wave at 5Hz multiplied by 4 different sine probes at 4 different frequencies (5Hz, 10Hz, 15Hz and 20Hz). It is known that square wave has only odd harmonics. Our 5 Hz square wave has therefore first harmonic at 5 Hz, second at 15Hz, third at 25 Hz, etc.

We see that test results coincide with the theory. Mean values of results 1 and 3 are non zero while results 2 and 4 equals zero (this should be observed in the time span of one cycle of 5Hz sine wave). This means that square wave has its first and second fundamental at 5Hz and 15Hz which is correct.

The mechanism behind detecting sine components in complex signals is the same. The reason why last few sections are dedicated to harmonic sounds only lies in two things. First is that example with harmonic target is much more illustrative and the second is that we have not talked about phase differences jet.

3.2.9 Phases

Now that we know how Fourier integral detects harmonics and measure their amplitudes when probe and target are synchronized we need to explain what does it happen when the phases of probe and target are not the same. As we will see finding the phase of the target is necessary for correct amplitude detection when phase is arbitrary.

In next experiment a sine wave was multiplied with 4 another sine waves. Amplitudes and frequencies were identical only phases were different each time.

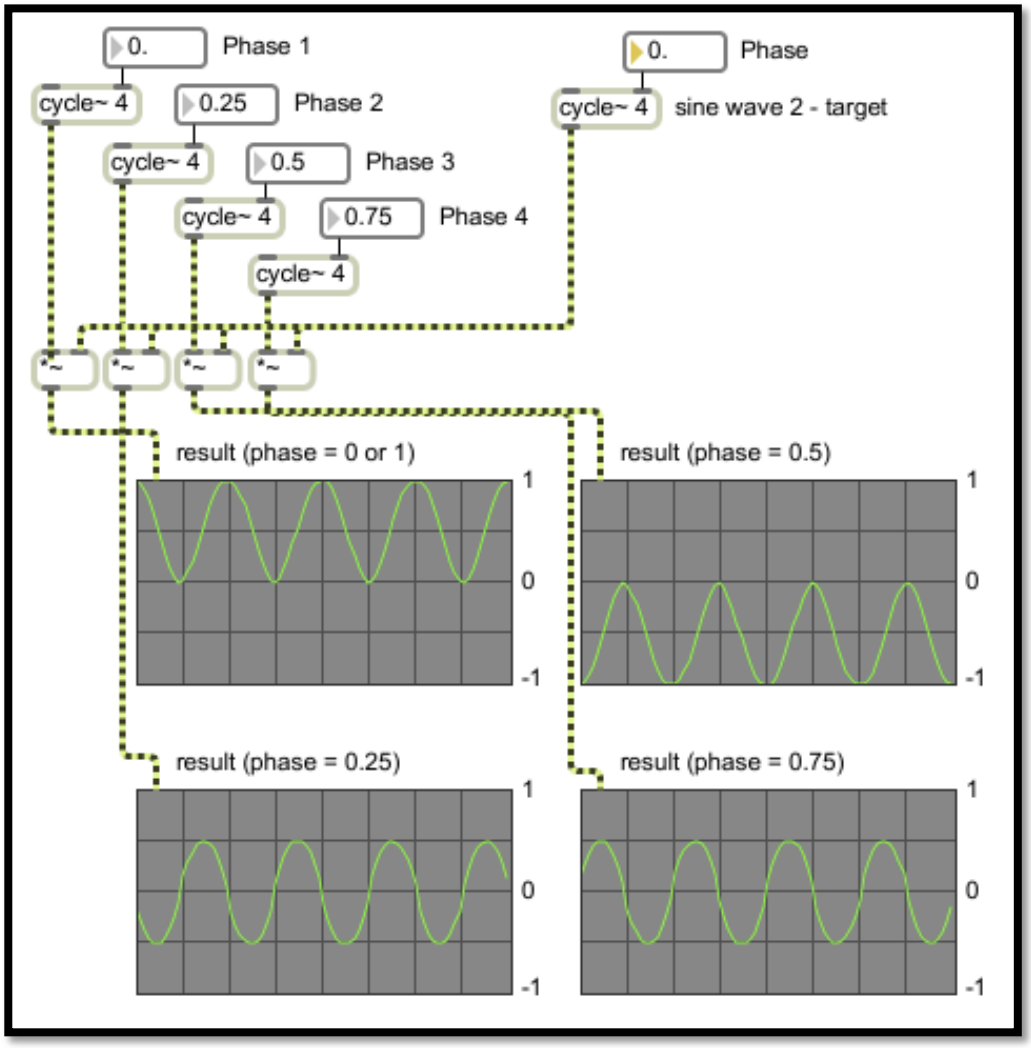

Figure 3.11 A sine wave (f = 4Hz, Amp = 1) multiplied with 4 equal sine waves (f = 4Hz, Amp = 1) but with different phases. Phase 0.25 means $\pi/2$ radians, 0.5 = π, 0.75 = $3\pi/2$, 1 = 2π (original).

From fig. 3.11 is evident that when there is no phase difference ($\phi = 0\ or\ 2\pi$) the DC offset is 0.5. This is basically the same example as presented on fig. 3.8. When phase difference is $\phi = \pi/2$ or $3\pi/2$ (0.25 or 0.75) DC offset equals 0 and when phase difference is π (0.5) DC offset is – 0.5.

From these results it can be assumed that DC offset varies accoring to the $\frac{cos\phi}{2}$ or half of the phase's (phase difference between target and probe) cosine ($\cos 0\ and\ \cos 2\pi = 1, \cos \pi/2\ and \cos 3\pi/2 = 0, \cos \pi = -1$). Mathematical proof based on Euler's relationships can be found on Branble's web site and confirms our assumption:

$$\sin \omega t \cdot \sin(\omega t + \phi) = \frac{\cos \phi}{2} - \frac{\cos(2\omega t + \phi)}{2}$$

DC offset indeed equals $\frac{\cos \phi}{2}$. On the other hand frequency and amplitude remained the same as in example without phase difference (chapter 3.2.5). Frequency is again doubled $(2\omega t)$ and amplitude is halved $\left(\frac{1}{2}\right)$.

If we would be still using our simpleified version of Fourier integral with only sine part of $e^{-j\omega t}$ we could find the target's phase by changing the phase of a probe until the DC offset of ring modulation would be maximal (0.5 in our case). This could be implemented by changing the probe's phase gradualy »thousand« times and choosing the highest result. On one hand that would work but on the other hand that would result in »ESFT—the Extremely Slow Fourier Transform" (Redmon, 2002).

If we take a look again at fig. 3.11 we see that sine with phase (difference) 0.25 ($\phi = \pi/2$) is actually a cosine. In that example the incoming sine or target had phase 0. Multiplication with equal probe (sine wave, phase also 0) gave us the best result – perfect match (DC offset = 0.5). On the other hand multiplication with probe cosine (also phase 0 because this sine wave with a phase shift 0.25 is now considered as cosine) gave us the worst result (DC offset = 0). If the incoming signal (target) would be cosine the story would be exactly opposite. We could take

only the cosine part of $e^{-j\omega t}$ and repeat all the ring modulation experiments and we would figure out that sine part of $e^{-j\omega t}$ is not needed when target is cosine. And if the incoming signal is something in between sine and cosine, both multiplications would give us a partial match. Since $sin^2\phi + cos^2\phi = 1$ for any ϕ we can calculate the phase and amplitude easily using only 2 probes – sine and cosine. "This is also the basis for the DFT" (Redmon, 2002).

> *[As shown before with the square wave example,] this process can clearly be extended in principle to arbitrary input signals by sweeping the frequency of the [probe] sinewave continuously though the audible range and recording the fluctuating amplitude of the output - any non-zero value signifies the presence in the input of a harmonic at that frequency and amplitude. Unfortunately, such an ideally perfect variable [analogue] ring modulator is extremely difficult to design - apart from any other considerations it would probably be impossibly expensive.*
>
> (Dobson, 1993).

3.2.10 DFT and FFT

The Discrete Fourier Transform (DFT) is a "Fourier transform for digital world" that works with discrete signals (time) and discrete spectrum (frequency). On the other hand the Discrete-Time Fourier Transform (DTFT) deals only with discrete signals (and continuous spectrum) and will not be presented in this paper.

In previous sub-section we were probing only fundamental frequency of the target and also the first over tone in the case of square wave. All our conclusions were based on the observation of sine waves in the time span of one cycle or period (t = 2pi). From now on our observation interval will be defined with an arbitrary time slice that will be measured in samples. We wil also discuss only discrete signals and discrete spectra.

Since we usually do not know the fundamental frequency of the target we set the fundamental of the probe randomly. We do that by choosing the length of the analysis time slice (in samples). But if we choose a time slice that consists of a 2^n samples a faster version of DFT can be used that is called FFT or Fast Fourier Transformation. For instance 2^9 or 512 samples long window at sampling rate 44.1kHz would give us a fundamental frequency of the probe at 86.13Hz (44100Hz/512=86.13Hz). This frequency is called fundamental FFT frequency (see fig. 3.1).

When using 512 samples long time slice 86.13Hz is the lowest frequency that can be analyzed in the incoming signal. Lower frequencies have periods that are larger than 512 samples and therefore do not fit in the time slice. As shown in the previous section this is not something that Fourier analysis is after. “FFT assumes that the samples being analyzed comprise one cycle of a periodic wave” (Dobrian, 2001, MSP Tutorial 25).

After probing with fundamental frequency of FFT, FFT continues “with the harmonic series (2x, 3x, 4x…) through half the sample rate...We also probe with 0x, which is just the average of the target and gives us the DC offset” (Redmon, 2002).

Since FFT is only an efficient algorithm for calculating DFT as Smith concludes (2007), we will continue with mathematical definition of DFT:

$$X[\omega_k] = \sum_{n=0}^{N-1} \mathrm{x}[t_n] e^{-j\omega_k t_n} \; ; \qquad k = 0, 1, 2, \ldots, N-1 \qquad (eq. 3.12).$$

where

$$\sum_{n=0}^{N-1} f(n) = f(0) + f(1) + \cdots + f(N-1)$$

$\mathrm{x}[t_n] =$ input signal $amplitude$ (real or complex) at time t_n (sec)

$t_n = nT =$ nth $sampling\ instant$ (sec) or nth $sample\ of\ time$, n an integer ≥ 0

$T = sampling\ interval$ (sec)

$X[\omega_k] = spectrum$ of x (complex valued), at frequency ω_k

$\omega_k = k\Omega =$ kth $frequency$ sample (radians per second)

$$\Omega = \frac{2\pi}{NT} = \text{radian} - \text{frequency } sampling\ interval \text{ (rad/sec)}$$

$$f_s = \frac{1}{T} = sampling\ rate \text{ (samples/sec, Hertz (Hz))}$$

$N = number\ of\ time\ samples = no. frequency\ samples$ (integer).

(Smith, 2007)

If we compare this DFT definition with the definition of Fourier Transform, $X(f) = \int_{-\infty}^{\infty} x(t)e^{-j\omega t}dt$, we see that they are very similar. First of all $X(\omega)$ is the same as $X(f)$ since $\omega = 2\pi f$ (ω is f multiplied by a constant). On the other hand infinite integral or in other words infinite sum of infinitesimal small part is replaced by finite sum of discrete parts. Also the summation boarders go from 0 to N-1 (N is the size of time slice in samples) instead of going from – infinity to + infinity.

Another common notation of DFT uses X[n, k] instead of $X[\omega_k]$ telling us explicitly that X depends on *n* and *k*. An important thing to remind at this point is that separating *n* and *k* is just a mathematical formalism that can be confusing. Values of *n* and *k* are all the time the same and they both go from *0* to *N-1*. The consequence of that is the number of time samples being analyzed is always the same as number of frequency bands we get from the analysis. In other words, if we choose to analyze a 512 samples big time slice we get the magnitudes and phases of 512 harmonically related frequency bands after the DFT.

Since frequency bands in Fourier analysis are equally spaced or harmonically related their frequencies can be easily calculated. All the frequency bands are integer multiples of the fundamental FFT frequency.

All the frequencies being analyzed that exceed the Nyquist rate (half the sampling rate) are being folded back. The mirror effect (the fold back) is the same as in the case of aliasing although the reason for fold back here lies elsewhere – in negative angles. Therefore we are interested only in the first half of the DFT results.

[In Max/MSP, object fftin~ performs] what is known as a real *FFT, which is faster than the traditional complex FFT (...) This is possible because the time-domain signals we transform have no imaginary part (or at least they have an imaginary part which is equal to zero). A real FFT is a clever mathematical trick which re-arranges the real-only time-domain input to the FFT as real and imaginary parts of a complex FFT that is half the size of our real FFT. The result of this FFT is then re-arranged into a complex spectrum representing half (from 0Hz to half the sampling rate) of our original real-only signal. The smaller FFT size means it is more efficient for our computer's processor.*

(Dobrian, 2001, MSP Tutorial 25).

Let us see what is going on in Max/MSP when doing DFT on the incoming signal (861.3281Hz sine wave) at N = 512 and $f_s = 44.1$kHz:

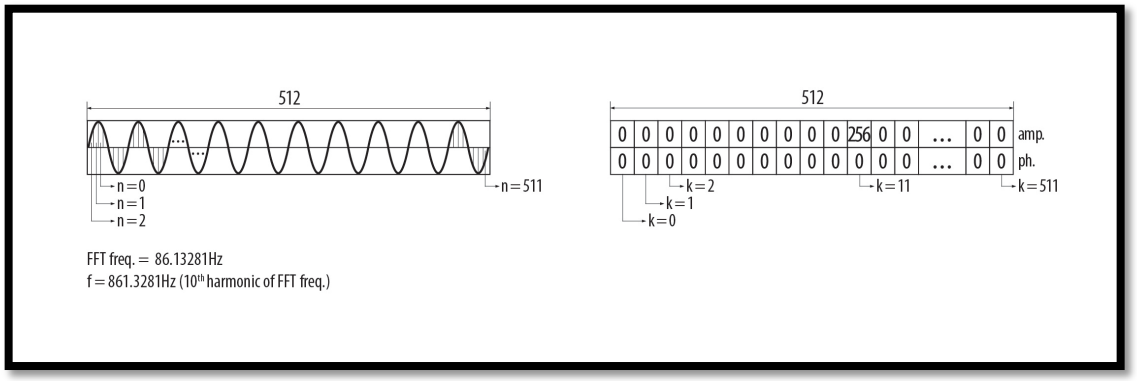

Figure 3.12 On the left side there is a 512 samples long discrete signal. On the right side we see the discrete spectrum after the *real* FFT analysis or one FFT frame (original).

In this example the incoming sine wave has exactly 10 times bigger frequency than the FFT fundamental. Analyzed sine wave therefore fits perfectly in the analysis window resulting in magnitude 256 and no phase shift in the 11^{th} frequency bin (10^{th} harmonic). Magnitudes in all other frequency bins are 0 and that is perfectly correct.

If we slightly change the frequency of the incoming sine wave those 10 periods will not fit into the analysis time slice any more. Since FFT interprets all the analysis data as periodic signals our sine wave is not a sine wave any more for Fourier analysis. Consequence is that we detect energy (magnitude values) and phase shifts in all 256 frequency bins. Energy is still greatest in 11^{th} bin but there is a lot of so called spectral leakage. In other words, 256 sine waves with different amplitudes and frequencies would be needed to reconstruct our "periodic" signal ("periodic with a step"). And this is what happens if inverse Fourier transform is applied in the process. We get almost perfectly reconstructed input signal from all that spectral leakage at the output. The reason for almost perfect and not perfect lies only in calculation errors and numerical rounding. This is not surprising according to the fact that inverse FFT is really just an inverse function of FFT. Therefore when doing only FFT and inverse FFT different analysis parameters that affect spectral leakage are not important (so far we have came across only one parameter that is the size of time slice or FFT fundamental). Calculation errors in the process of FFT and inverse FFT are not audible - all the time and frequency information is therefore preserved. "The fact that it may be difficult to interpret a single FFT output of a spectrally complex input does not mean that the information has somehow got lost or damaged in some way - it simply means that it is difficult to interpret!" (Dobson, 1993).

On the figures 3.13.a, 3.13.b and 3.13.c we have plotted the output of FFT into the sonogram. On the sonograms below there is a visual presentation of a sequence of 320 FFT outputs or FFT

frames for the previous example. 3 different sine waves were probed with FFT fundamental 86,13Hz (512 long time slices at sample rate 44.1KHz).

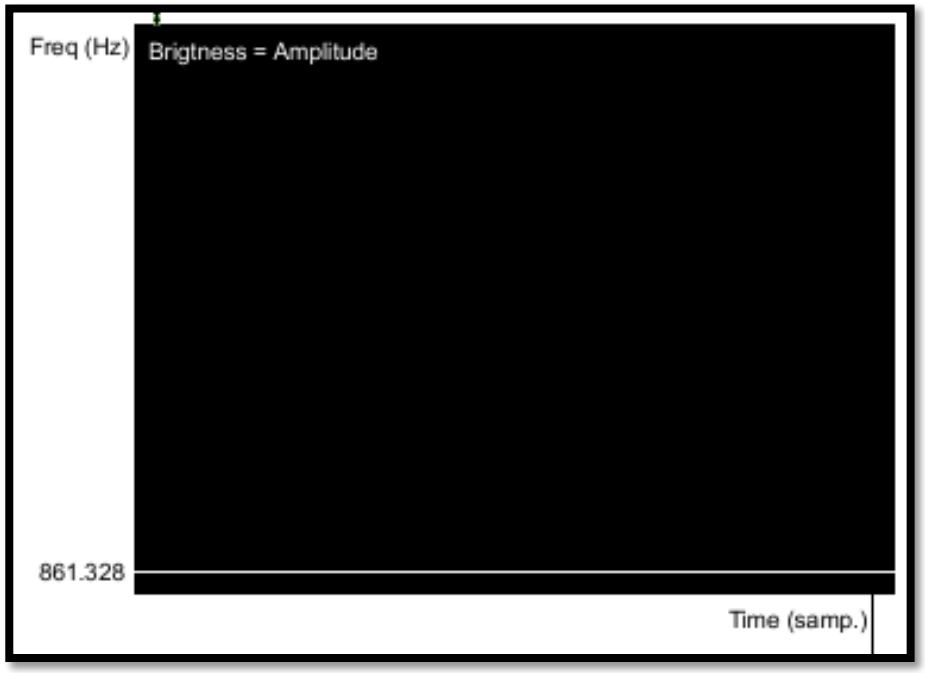

Figure 3.13.a Spectral analysis of a 861.328 Hz sine wave with a 512 samples long FFT time slice or square window (original).

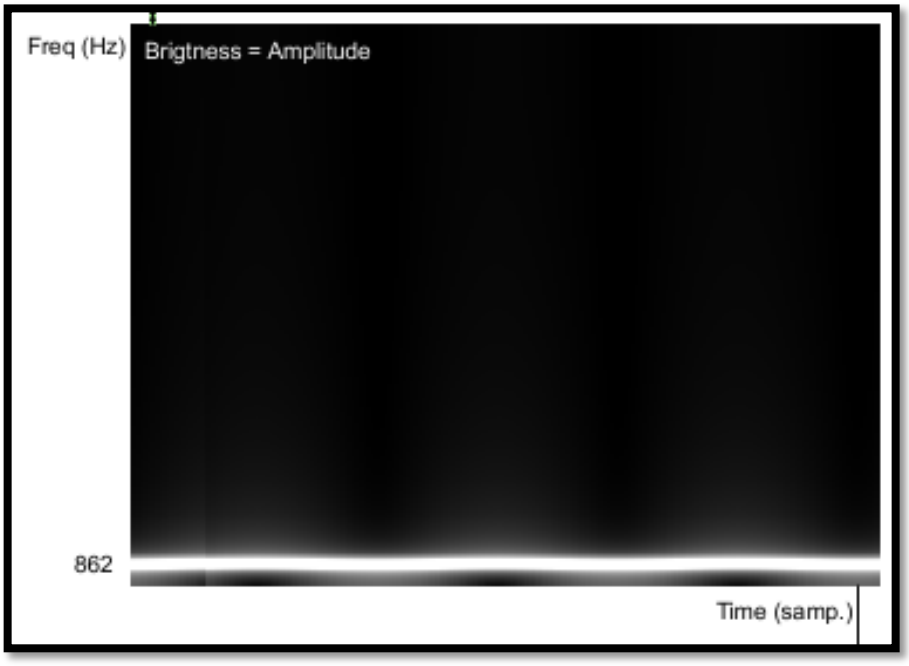

Figure 3.13.b Spectral analysis of a 862 Hz sine wave with a 512 samples long FFT time slice or square window (original).

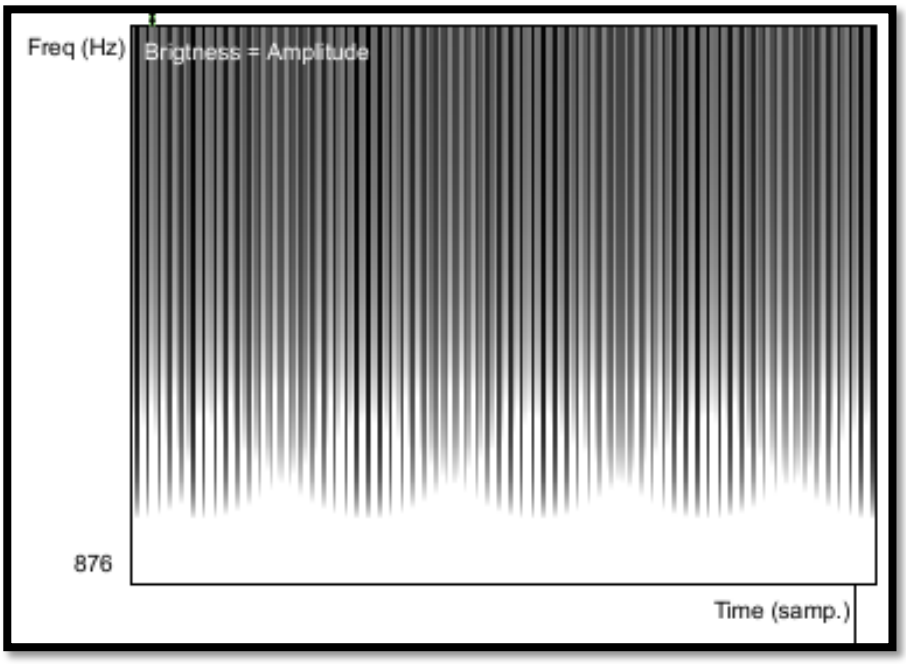

Figure 3.13.c Spectral analysis of a 876 Hz sine wave with a 512 samples long FFT time slice or square window (original).

When looking at figures 3.13.a, 3.13.b and 3.13.c it is important to remember that from mathematical perspective all three results of the analysis are equally good and correct. They all hold all the temporal and frequency information needed for reconstruction of the original incoming sine wave. The only reason why the analysis data from 3.13.a is the only practically useful is because it is easy to interpret.

Since the only sense of performing FFT (and potentially its inverse) is in visualization, processing or further analysis of the FFT data, we need that data in the most clear, manageable and unobstructed form (like in case presented on fig 3.13.a where period of a target was exactly the same as analysis time slice). Therefore we need to modify the incoming signal in a way to prepare it for Fourier analysis. We need to force the incoming signal to become periodic in a way to exactly fit in the chosen 2^n samples long time slice and make as little change to it as possible. This can be done using overlapping windowing technique that is the basis of the Short-Time Fourier Transform or STFT – "practical cousin of the DFT" (Roads, 1996, p. 1094).

3.2.11 STFT

From figures 3.13.a, 3.13.b and 3.13.c we could conclude that DFT is practically useful for harmonic sounds with known fundamental. For everything else and even for harmonic sounds if we want to use faster algorithm FFT a method called Short-Time Fourier Transform or STFT is needed (although FFT could work fine in exceptional cases).

STFT applies a window function on a time slice (see fig. 3.14). Window function is a volume envelope that scales the ends of the sliced signal and has the same length as the time slice.

Volume envelope is applied to time slice by multiplication and multiplication of signal with unipolar envelope is called amplitude modulation (AM). AM distorts the signal by producing sidebands. “If the frequency of the variation [*amplitude modulation] is raised to the audible band (i.e. higher than approximately 18Hz) then additional partials (or *sidebands*) will be added to the spectrum of the signal” (Miranda, 1998, p. 20). Therefore the result of the analysis is “convolution of the spectra of the input and the window signals” (Roads, 1996, p. 551). Convolution of spectra means multiplication in time domain and vice versa.

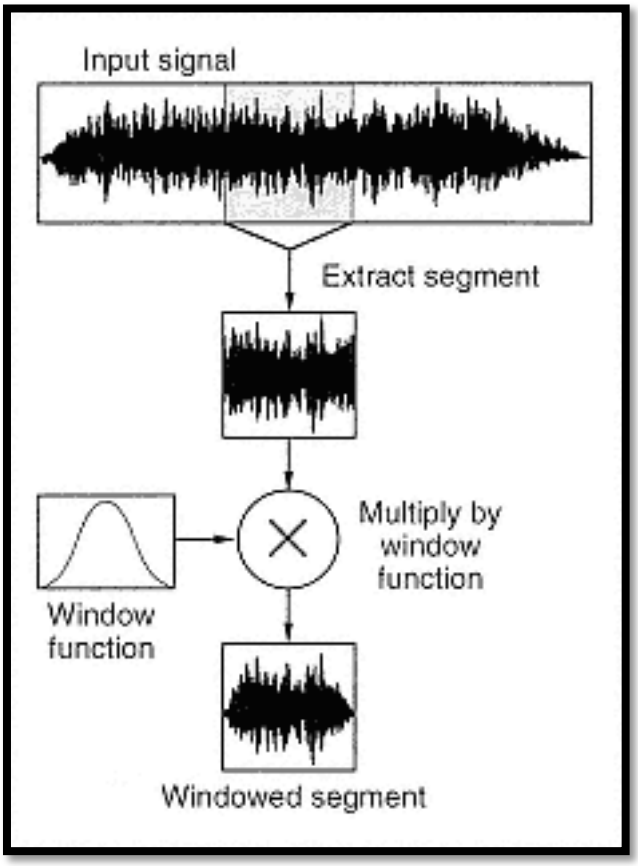

Figure 3.14 Time Slice multiplied by window function (Roads, 1996, p.550).

Side effect of window function can be reduced to great extend by overlapping windows. AM in general can be practically eliminated with overlapping triangle or Hanning modulation. For overlap 2 for instance, the sum of Hanning or triangle functions adds to a constant 1, producing no AM. As stated by Julius O. Smith, if the window $\mathrm{w}[t_n]$ has the *Constant OverLap-Add (COLA) property* at hop size R then the sum of the successive FFTs over time equals the FFT of the whole signal $X[\omega_k]$ (2008, Mathematical Definition of STFT chapter).

It is important to emphasize that actual sum of successive FFTs does not happen in analysis stage (in our implementation) but rather the sum of inverse FFTs in overlap-add (OA) resynthesis stage.

Smith's mathematical definition of STFT adopted for discrete frequency world (by author of this text) is:

$$X[\omega_k] = \sum_{n=0}^{N-1} \mathrm{x}[t_n]\mathrm{w}[t_n + \mathrm{mR}]e^{-j\omega_k t_n} \;; \qquad k = 0, 1, 2, \dots, N-1$$

where

$$\mathrm{w}[t_n] = window\ function$$

$$\mathrm{m} = frame\ index$$

$$\mathrm{R} = hop\ size$$

$$\mathrm{w}[t_n + \mathrm{mR}] = time\ shifted\ window\ function$$

As we can see the definition is very similar to the definition of DFT. The only new thing in equation is time shifted window function $\mathrm{w}[t_n + \mathrm{mR}]$. For the *Constant OverLap-Add (COLA)* windowing we can rewrite the STFT definition:

$$X[\omega_k] = \sum_{n=0}^{N-1} \mathrm{x}[t_n] e^{-j\omega_k t_n} \sum_{n=0}^{N-1} \mathrm{w}[t_n + \mathrm{mR}] = \sum_{n=0}^{N-1} \mathrm{x}[t_n] e^{-j\omega_k t_n} \cdot C$$

Where

$$\mathrm{C} = constant \text{ that can be 1 for certain window functions and hop sizes.}$$

When sum of overlapping windows does not add up to a constant then some form of modulation will be heard at the frequency of a hop size, as noted by Allen and Rabiner (1977). This can be caused by various transformations of spectral data. Hop size in overlap-add (OA) resynthesis stage can also be different as hop size in analysis stage due to potential time stretching of phase vocoder. Comb filtering effects and other artifacts may arise when perfect summation criterion is broken.

Since perfect summation criterion can be achieved with appropriate window functions and overlap 2 (R=N/2), why the need for higher overlapping?

> *Increasing the overlap factor is equivalent to oversampling the spectrum, and this protects against the aliasing artifacts that can occur in transformations such as time-stretching and cross-synthesis. An overlap factor of eight or more is recommended when the goal is transforming the input signal.*
>
> (Roads, 1996, p. 555)

The output of STFT (or FFT) can be imagined as successive frames of a film: each successive FFT output is analogues to frame of a film that is basically a snapshot frozen in time. These FFT snapshots are called FFT frames containing amplitude and phase spectral data (fig. 3.12 and 3.15). When played fast we get the sense of movement but when played extremely slow, we can see the reality behind the illusion – the structure based on static frames. So more frames that we have (bigger overlap) better will be the time resolution of resynthesized sound (this matters only when FFT data is processed which happens by default in phase vocoder as we will see later).

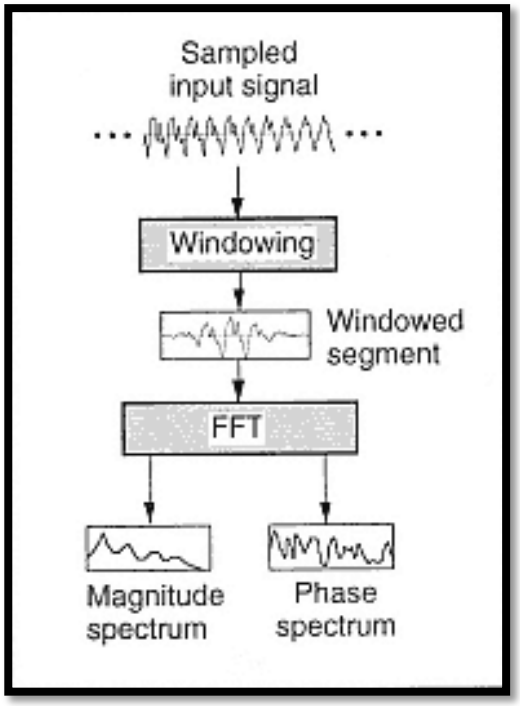

Figure 3.15 Magnitude and Phase spectrum after the STFT (Roads, 1996, p.551).

Time resolution also depends on window size that is usually the same as FFT frame size. As already mentioned, Max/MSP object *fftint~* inside *pfft~* performs real FFT that truncates the

unwanted mirrored spectrum producing FFT frames that are half the window size. Beside that there is no zero-padding going on inside *pfft~* as mentioned by Charles (2008). Zero-padding means adding an array of zeros to the end of the time slice before the FFT from various reasons. According to Weeks (2007, p. 202), by the definition of frequency resolution, zero-padding also increases frequency resolution although it does not add any information. Frequency domain is just smoother looking.

The problem with FFT is that frequency resolution also depends on window size but in opposite relationship as time resolution. Larger windows mean higher frequency resolution (more frequency bins) but lower time resolution and vice versa. "Since the FFT computes the average spectrum content within a frame, the onset time of any spectrum changes within a frame is lost when the spectrum is plotted or transformed. If the signal is simply resynthesized, the temporal information is restored" (Roads, 1996, p. 567). There is always a tradeoff between these two parameters and that major problem of FFT is known as Heisenberg's uncertainty principle of time-frequency resolution. Named after one of the pioneers of quantum mechanics that discovered the tradeoff between position and velocity when measuring quantum particles. So we could conclude that also in STFT measurement or analysis method determines the result.

"Artifacts of windowed analysis arise from the fact that the samples analyzed do not always contain an integer number of periods of the frequencies they contain" (Roads, 1996, p. 1099). If we translate this to frequency domain world it means that samples analyzed does not contain only harmonics - we can have two or more partials in the frequency range covered by only one bin (fig. 3.16). Our target frequency can be therefore assembled from many partials and with phase vocoder, as we will see later, we are "forced" to consider them as single frequencies. That is why the term energy is commonly used instead of magnitude to more explicitly express that

FFT does not give us the exact information about what is being analyzed. How many partials from input signal are actually present in specific frequency bin is not known. Ideally there should be only one.

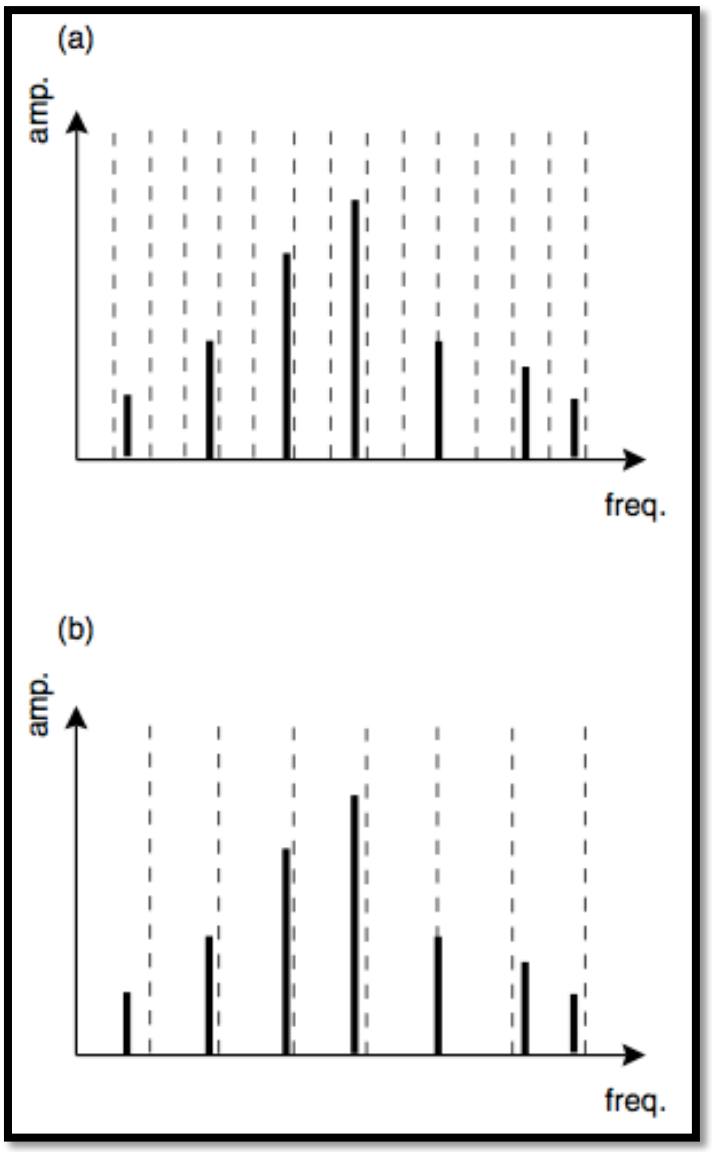

Figure 3.16 a) STFT distinguishes between the two uppermost partials (frequency resolution is high enough). b) STFT does not distinguish between the two uppermost partials (frequency resolution is to low) (Miranda, 2002, p.61).

For this research various window types are not of any great importance. Different window functions would at the end result in subtle sound changes that comes from phase vocoder but this is not something this research is focused upon. In the implementation of our phase vocoder COLA windowing and Hanning function will be used. “Maximum magnitude that can be obtained from using Hanning window is “FFTsize * 0.25” (Dobrian, 2001, MSP Tutorial 26).

In chapter 3.2.10 (DFT and FFT) we showed that resulting magnitude is always FFTsize * 0.5 when using only time slicing or in other words square windows. Since the mean value of symmetric Hanning function is exactly 0.5, multiplying the result obtained from time slice (square window) by Hanning gives us the maximum magnitude FFTsize * 0.5 * 0.5 = FFTsize * 0.25.

3.2.12 Phase Vocoder and Spectrogram in Max/MSP

> *The phase vocoder has its origins in a long line of voice coding techniques aimed at minimizing the amount of data that must be transmitted for intelligible electronic speech communication. (...) The phase vocoder is so named to distinguish it from the earlier channel vocoder (...) [It] was first described in an article by Flanagan and Golden (1966), but it is only in the past ten years that this technique has become popular and well understood.*
>
> (Dolson, 1986)

In the aim of upgrading the vocoder technique, Flanagan and Golden at Bell Telephone Laboratories proposed “another technique for encoding speech to achieve comparable bandsaving and acceptable voice quality” (Flanagan and Golden, 1966). Their program was

called phase vocoder and was unfortunately causing a data explosion instead of data compression (Roads, 1996, p. 549). More than 10 years later, Michael Portnoff from MIT developed an efficient phase vocoder, based on FFT algorithm discovered by Schafer and Rabiner. Beside audio data reduction in communications, Portnoff presented his phase vocoder also as a "technique for manipulating the basic speech parameters" (Portnoff, 1976). This lead to a paper called *The Use of the Phase Vocoder in Computer Music Applications* by James Moorer (1978) that according to Dodge and Jerse, introduced phase vocoder to computer music (1997, p. 253).

Phase vocoder (PV) can be considered as an upgrade to STFT and is consequentially a very popular analysis tool. The added benefit is that it can measure a frequency deviation from its center bin frequency as said by Dodge and Jerse (1997, p. 251). For example if the STFT with fundamental frequency 100Hz is analyzing a 980Hz sine tone, the algorithm would show the biggest energy in 11^{th} bin (index 10) - in other words at the frequency 1000Hz. PV is on the other hand able to determine that the greatest energy was concentrated -20Hz below the 1000Hz bin giving us the correct result 980Hz.

Since phase is a relative measure it is the difference that is important. "The difference between the phase values of successive FFT frames for a given bin determines the exact frequency of the energy centered in that bin. This is often known as the phase difference" (Dudas & Lippe, 2006). The link between phase and frequency deviation can be seen on fig. 3.17.

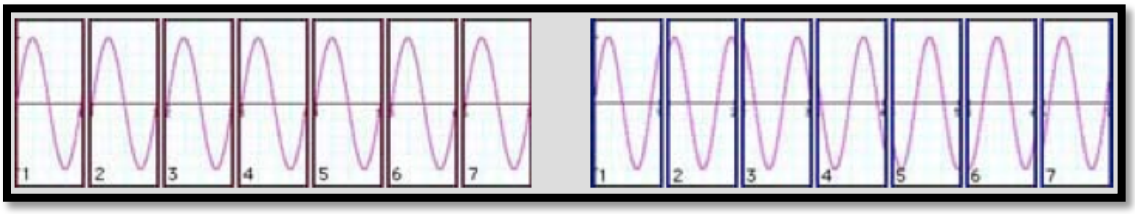

Figure 3.17 The content of 7 successive time slices prepared for the analysis. Left signal perfectly fits into the analysis window all the time (frequency of the signal equals FFT frequency) while on the other hand right signal appears to have a phase shift in each window (frequency of the signal is higher than FFT frequency) (Sprenger, 1999).

From fig. 3.17 it seems that phase and frequency difference are in linear relationship. If we imagine that frequency of the right signal in fig. 3.17 would be 50% higher than FFT frequency, starting with phase offset 0, then we would have phase offset 0 in all odd indexed windows and phase offset *half window size or* π in all even indexed windows. The constant phase difference between successive frames would be therefore π or 50% - the same as frequency deviation from specific FFT frequency.

In order to know if frequency deviation from center bin frequency is positive or negative, phase difference should be wrapped within the range of $[-\pi, \pi]$. The equation for phase difference is:

$$\Delta\phi_{m,k} = \phi_{m,k} - \phi_{m-1,k} \quad \text{(original)}$$

where m is frame index and k is a bin index.

If we take a look again at fig. 3.17 we see that the example was made with overlap 0. In order to get the phase difference between the two successive *non overlapping* FFT frames for a given bin when using overlap, we propose the solution of summing N (=overlap factor) successive phase

differences together. The resulting phase difference is the one we can use for calculating the frequency deviation Δf:

$$\Delta f_{m,k} = \sum_{m=m}^{m+N} \Delta\phi_{m,k} \cdot \frac{f_{FFT}}{2\pi} \quad eq.3.13 \qquad (original)$$

There should be also a way to calculate the frequency deviation Δf from only one phase difference instead of N (overlap factor), using the time of the hop size in the equation instead of the whole window, but any sufficient equation that would work with data obtained after the STFT analysis in Max/MSP could not be found. The problem is for instance that when analyzing a sine wave with frequency twice the FFT size and using overlap 4, the phase differences between third bins of successive frames are $\pi, -\pi, \pi, -\pi, \ldots$ As we see the phase differences for a constant sine wave are not constant so it might be even impossible to calculate the true bin frequency using only the phase differences between two successive frames. On the other hand when summing all 4 phase differences together the result is constantly 0 that gives us also the correct and constant 0 frequency deviation.

The critique of proposed method is that when summing N (overlap) phase differences together, they might not belong to the same partial any more. That could also be the case when using only one phase difference for true bin frequency calculation, but the chances are bigger in the first case.

The correct frequency (assuming only one frequency is present in each bin) in bin k and frame m is therefore:

$$f_{m,k} = f_{Ck} + \Delta f_{m,k} \qquad eq.14$$

where center frequency (for all frames) $f_{Ck} = k \cdot f_{FFT}$.

Phase hence contains a structural information or information about temporal development of a sound. ”The phase relationships between the different bins will reconstruct time-limited events when the time domain representation is resynthesized” (Sprenger, 1999). Bin’s true frequency, as Sprenger concludes, therefore enables a reconstruction of time domain signal on a different time basis (1999). In other words, phase difference and consequently a *running phase* is what enables a smooth time stretching or time compression in phase vocoder.

Since inverse FFT demands phase for signal reconstruction instead of phase difference, phase difference has to be summed back together and this is called running phase. If there is no time manipulation present (reading speed of PV = 1), running phase for each frequency bin equals phase values obtained straight after the analysis. But for any different reading speed running phase is different from initial phases and responsible for smooth signal reconstruction. And taking phase into consideration when resynthesizing time stretched audio signals is the main difference between phase vocoder and synchronous granular synthesis (SGS).

Running phase for bin k and frame M

$$\phi_R = \sum_{m=0}^{m=M} \Delta\phi_{m,k} \qquad \text{(original)}$$

Running phase can be imagined as a clock that accumulates seconds. Due to very small imprecisions in the mechanism time gets gradually more and more incorrect and therefore needs readjustment. Since running phase is calculated for each bin separately this results in even bigger problem - inconsistencies in phase relationships. According to Laroche and Dolson (1997), horizontal phase-coherence (among frames) and vertical phase-coherence (among bins) has to be

guaranteed for high quality time-scaling modifications with phase vocoder. As a solution to this problem, Puckette (1995) suggested phase-locking while Laroche and Dolson (1999), upgrading the Puckette's theory, suggested peak phase-locking. These are the two groundbreaking contributions to phase related problems in PV and are unfortunately not used in our implementation. Phasiness problems are namely a very complex and comprehensive subject and improving the sound quality of PV is also not the purpose of this research.

As already mentioned, phase vocoder assumes that the energy presented in a single frequency bin belongs to a single partial. That is why large analysis windows (with big overlap) are recommended for spectral analysis with phase vocoder. High frequency resolution means lots of frequency bins that significantly increases the probability that only one partial could emerge per one bin (fig. 3.16). On the other hand, when analyzing sounds with important transients (percussive sounds for instance) smaller windows are preferable.

Therefore in Phase vocoder we can process amplitude and phase difference spectral data. Each analysis window corresponds to 1 frame of spectral data containing amplitude and phase difference information (phase difference in phase vocoder should not be mixed with phase difference in chapter 3.2.9). In our implementation the frame in question is split into two, so we have control over both spectrums separately. Following the ideas of Dennis Gabor, both spectrums are written in a matrix and plotted as sonograms. Each frame containing frequency bins is written vertically (row number = bin number) and each successive frame is written in successive column (column number = frame number). The original Gabor matrix can be seen on fig. 3.18 where ν means frequency, t time and values in matrix cells presents amplitude.

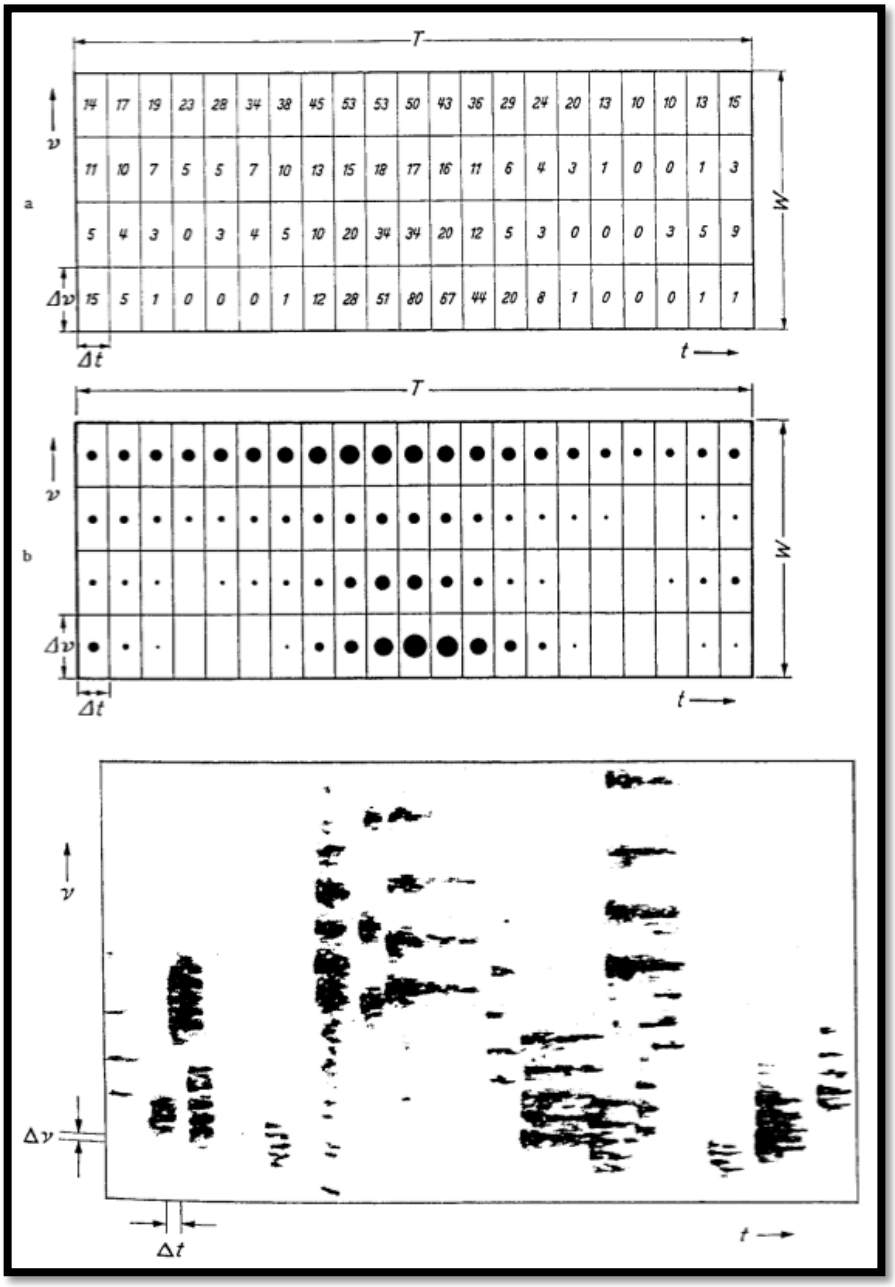

Figure 3.18 "The Gabor matrix. The top image indicates the energy levels numerically. The middle image indicates the energy levels graphically. The lower image shows how the cells of the Gabor matrix (bounded by $\Delta\nu$, where ν is frequency, and Δt, where t is time) that can be mapped into a sonogram" (Roads, 2001, p.60).

3.2.13 Interacting With Sonogram

In Jitter picture is presented as 2 dimensional (x, y) 4 plane (A, R, G, B) matrix. Each pixel corresponds to 1 matrix cell (x, y) that spans through all 4 planes (A, R, G, B). First plane (index 0) is the alpha plane and this is the only plane needed for presenting black and white pictures. Since our amplitude and phase difference matrixes both consist of only one plane it is appropriate to think of them as two separate grayscale pictures.

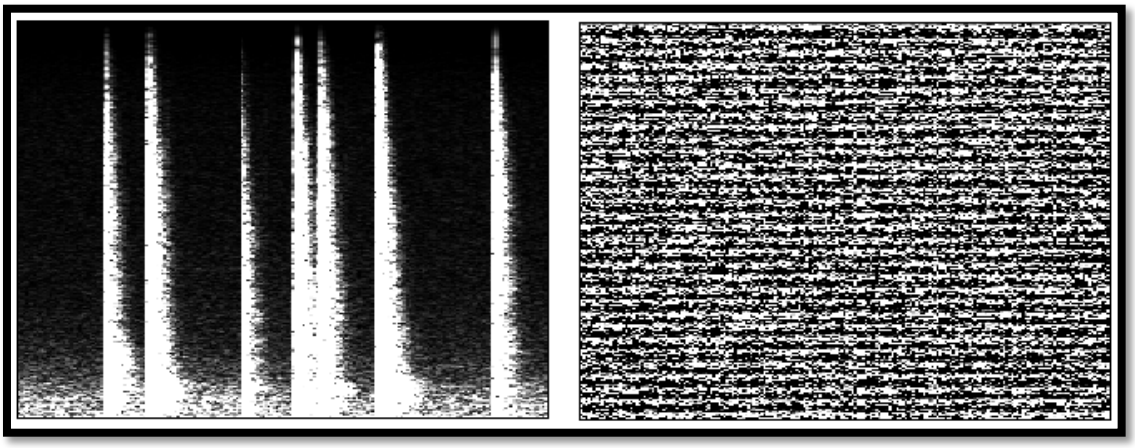

Figure 3.19 Amplitude and phase plane: Linear sonographical presentation of the sound of clapping with low frequency background noise. Left picture presents visualization of amplitude matrix and right picture presents visualization of phase difference matrix. Both matrixes consist of 200 columns (200 FFT frames) and 2048 rows (2048 FFT frequency bins). STFT parameters used: Hanning window long 4096 samples with overlap 4 (original).

To interact with spectrogram we need to extract spectral data (magnitude and correct frequency at specific time) from those two matrixes. Since phase plane is usually not shown in any

spectrogram, we will interact with phase difference plane through magnitude plane. Due to the same size of matrixes, x and y or column and row coordinates are the same for both.

The correct frequency of the clicked matrix cell *m, k* can be calculated using the already mentioned equations 3.13 and 3.14. Where *m* is a matrix column and *k* is a matrix row. According to Charles (2008), time in samples on the other hand equals *clicked column* or FFT frame multiplied by hop size. To get the time in milliseconds the result has to be divided by sampling rate.

$$t = \frac{mR}{sr}$$

where hop size R equals

$$R = \frac{windowSize}{overlapFactor}$$

One of the weaknesses of FFT when analyzing complex polyphonic sounds is that it may be too accurate in high frequency range and not accurate enough in low frequency range. This is due to its linear nature while the "octave series relates to pitch in a logarithmic manner" (Dobson, 1993). The frequency span of an octave or a semitone increase exponentially so logarithmic placement of FFT bins would be more appropriate for analyzing sound with complex spectra. But this is unfortunately impossible with Fourier analysis. Despite that it is sensible to remap the linear spectral matrix into logarithmic due to better visual presentation and easier interaction within frequency range that defines sound the most. Technically that means that the lower rows of the matrix will be "stretched" (multiplicated) while upper rows will be "shrunk" (some rows

will be left out). In Jitter that can be done using the *jit.repos* object and exponential Y spatial map (spatial map or a plane has to be made separately). To avoid the pixilation that emerges from multiplication of lower rows interpolation is recommended. If using 8 bit interpolation inside *jit.repos,* both spatial maps (X and Y) has to be multiplied with 256 or 2^8. See fig. 3.20a and 3.20b.

Since the FFT data is still stored in "linear" matrix, frequency is still calculated with equations eq. 13 and eq. 14. The only new thing in case of using logarithmic spectrogram (that is stored in separate matrix) is getting the index of a clicked row. That should be done using exactly the same scaling that was used for the generation of Y spatial map fed into *jit.repos*. Intuitively we would expect using the inverse function but when using *jit.repos,* the logarithmically scaled picture (spectrogram in our case) is a result of the exponential spatial map and not the logarithmic values of rows (or columns).

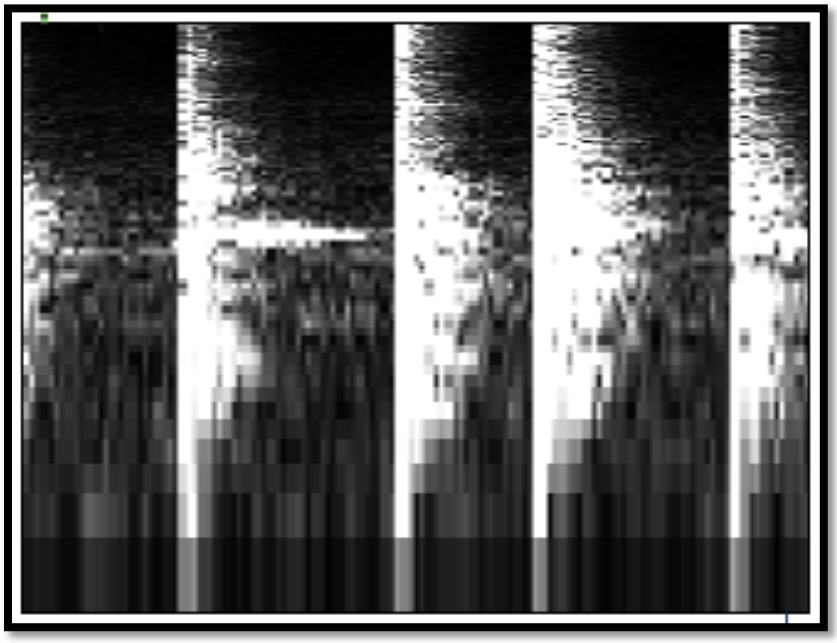

Figure 3.20a Logarithmic spectrogram as a consequence of exponential spatial mapping using object *jit.repos* at interpbits 0 (original).

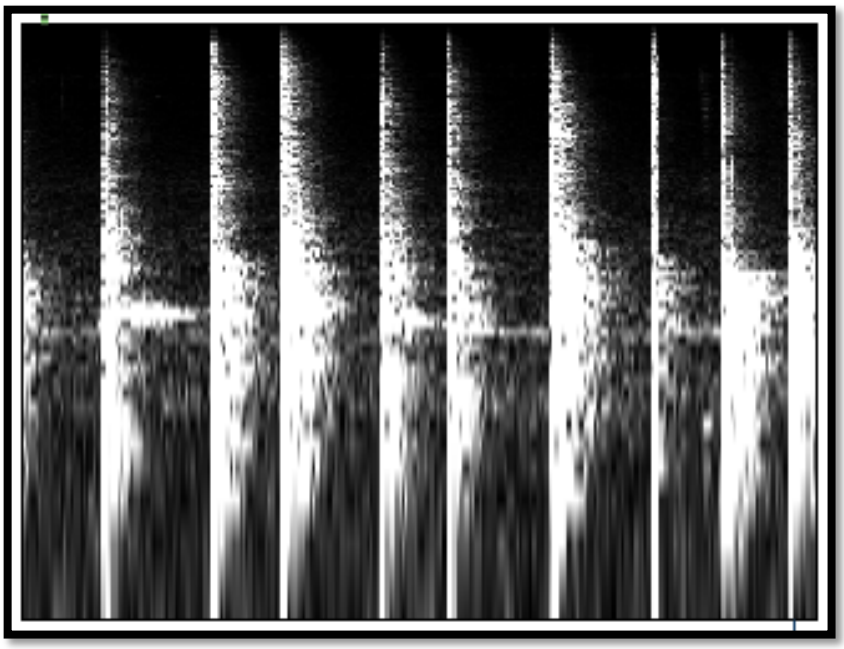

Figure 3.20b Logarithmic spectrogram as a consequence of exponential spatial mapping using object *jit.repos* at interpbits 8 and stretched X and Y spatial maps with factor 256 (original).

In conventional phase vocoder, FFT data is written in buffer instead of a matrix. If we play the input signal from “input buffer” and store FFT data in “FFT buffer”, then the result after the analysis are 2 FFT buffers (amplitude and phase difference), both overlap times bigger than the input buffer. The reason for “overlap times bigger” is that overlap in FFT buffer is gone – each FFT frame is placed separately and consecutive in the buffer. Since the buffer has to be read with audio signal the sampling rate inside *pfft~* object/subpatcher has to be at least overlap times bigger than sampling rate in the mother patch (where the input buffer is stored) in order to preserve a real time FFT. We have the same situation when FFT data is written in a matrix. The only difference is the way we write and read the matrix. The writing and reading progresses through consecutive columns, always going from first row with index 0 to the last row with index *half the FFT size* (fig. 3.21). Also the matrix has to be read with audio signal in order to process data at audio rate. Since sample accuracy has to be preserved during the process it is the best option to have all the crucial parts of Phase vocoder done in MSP (audio) domain. As Eric Lyon had written “Max/MSP event scheduler is prone to permanently drift from a sample accurate measurement of timing” (2006). Although in the case of PV, a solution from Christopher Dobrian, originally made for *synchronous granular synthesis* controlled with Max information should also be suitable:

> *MSP calculates a* vector *of several milliseconds worth of samples at one time, and control information from Max--such as unique starting point and transposition values for each grain--can only be supplied at the beginning of each vector calculation. For this reason, I have chosen to express the offset between grains as an integer multiple of the signal vector size, and the length of each grain (each triangular window) is twice that. By looking for the end of the window (testing for the maximum sample value coming from a* ***count~*** *object), the* ***edge~*** *object sends out a bang which can be used to trigger new control values at the beginning of the next vector (with the* Scheduler in Audio Interrupt *option checked to ensure that the control information is always synchronized with the beginning of a new vector).*
>
> (Dobrian, 1999)

Sample accuracy is crucial inside *pfft~* since phase difference and phase difference accumulation are calculated with objects *framedelta~* and *frameaccum~*. Both objects perform mathematical operations *in each position of its incoming signal vectors*. And signal vector inside *pfft~* is *half the FFT size* as noted in MSP tutorial 26 (Dobrian, 2001). In other words, signal vector inside *pfft~* has the same size as one column of our spectral matrix and therefore *framedelta~* and *frameaccum~* operate "*in each row of its incoming columns"*.

When using *pfft~* object inverse STFT is done by *fftout~*. Inverse STFT is basically inverse FFT that "takes each magnitude and phase component and generates a corresponding time-domain signal with the same envelope as the analysis window" (Roads, 1996, p. 553). *fftout~* also handles the overlap-add synthesis of output signal windows.

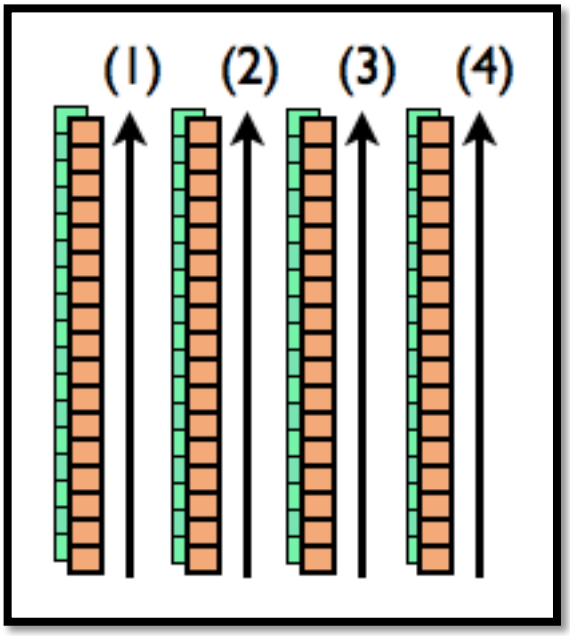

Figure 3.21 Reading the spectral matrix (original).

Inverse FFT (iFFT) will not be covered in details in this paper. On one hand mathematic is very similar as with FFT (only inverse) and on the other hand the concept of iFFT is very straightforward – it can be easily explained with the model of additive synthesis.

The schematic representation of PV implementation in Max/MSP is shown on fig. 3.22. Since FFT always calculates a spectrum of previously received windows there is always some delay between input and output of FFT. Usually the delay equals the window size. *pfft~* object however introduces a delay of *window size – hop size*. This delay can be seen on fig. 3.22.

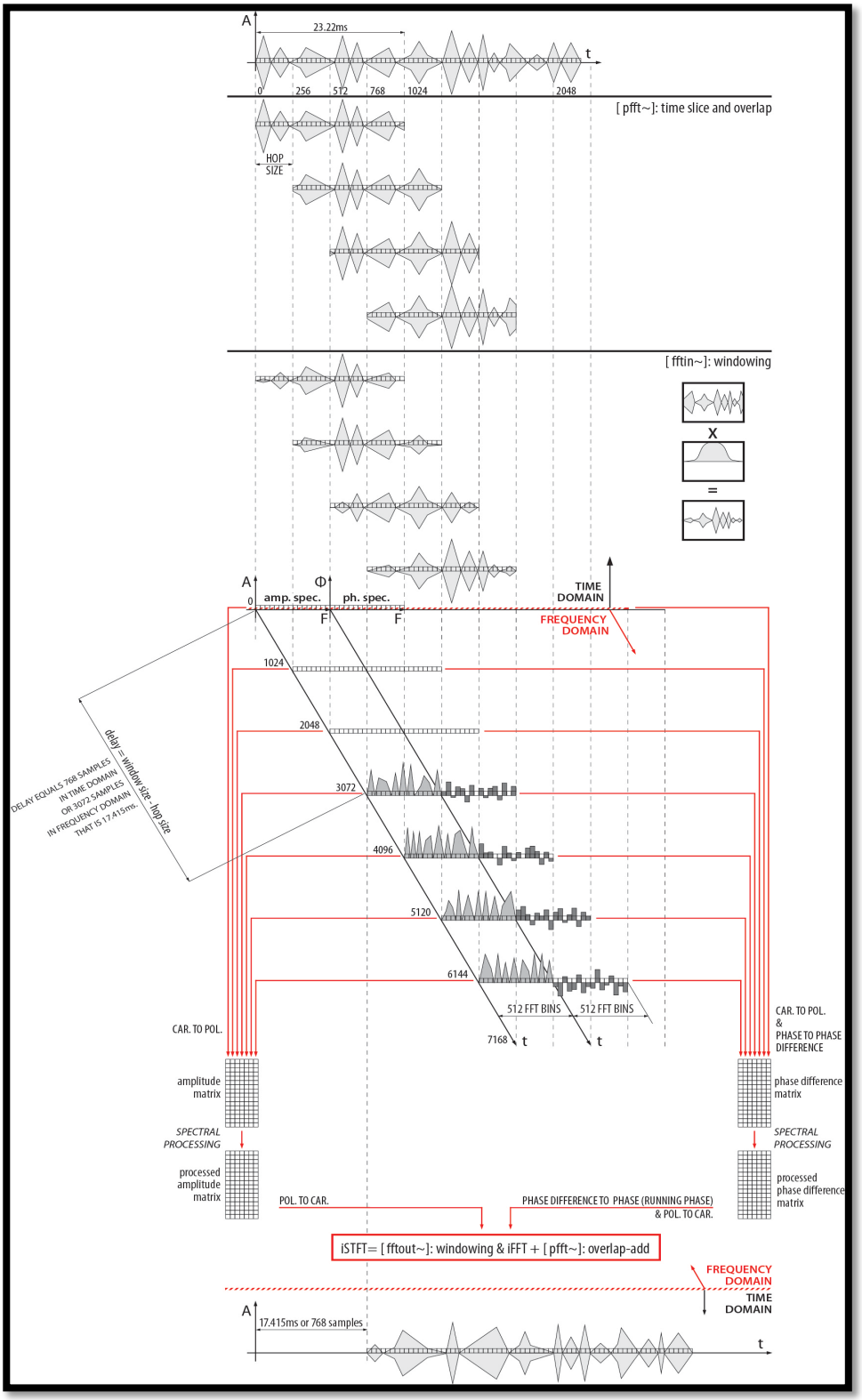

Figure 3.22 The process of sound analysis and resynthesis in phase vocoder using sonogram as a spectral data storage and user interface. The process is shown from the moment of turning on the audio in Max/MSP (original).

Chapter 4

Max/MSP/Jitter Patch of PV With Spectrogram as a Spectral Data Storage and User Interface

Our implementation of phase vocoder with spectrogram as an interface in Max/MSP is based on Luke Dubois groundbreaking patch jitter_pvoc_2D.maxpat and Jean-Francois Charles patches that accompanies his article *A Tutorial on Spectral Sound Processing Using Max/MSP and Jitter* that was published in Computer Music Journal (2008).

Programming in MSP domain is different as programming in Max. Programming in Max is conceptually similar to classical programming - all the events, or controls as it is popular to say within Max community, have to be triggered by some function or action. On the other hand events in MSP are occurring non-stop at the sample rate in all signal paths. The only way to stop these continuous events is to turn off the audio engine of Max/MSP. Even if signal is set to 0, it means that value 0 is “banging” the attached object at sample rate. Therefore some unconventional methods have to be used frequently when programming in MSP.

4.1 Main Patch – Time Domain

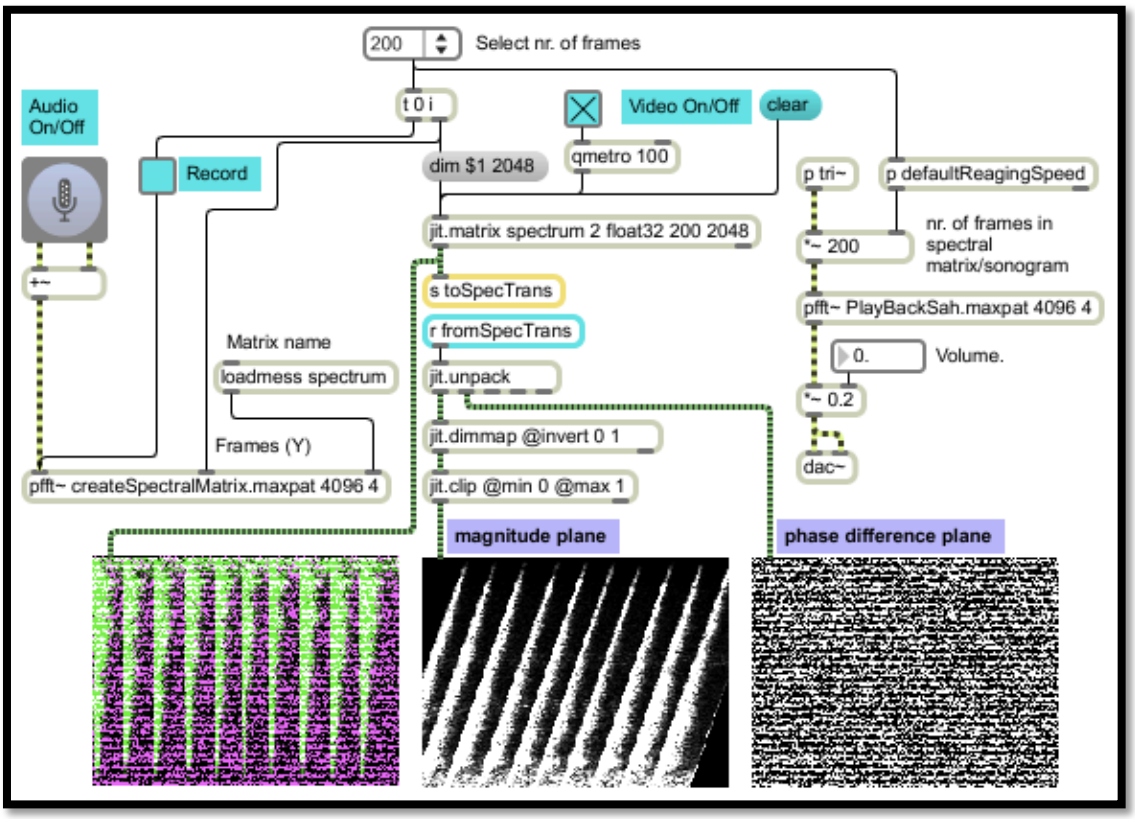

Figure 4.1 The main patch, time domain (original).

This is the main patch of phase vocoder implemented in Max/MSP and Jitter. All the audio (MSP) events in this patch exist in time domain and are limited to the speed of sampling rate. On the other hand Max and Jitter perform tasks as fast as possible and are limited only by the speed of computer. All the frequency domain events that are tied to the sample accuracy of audio signals are performed inside the two *pfft~* objects with audio signals (MSP), while the processing of FFT data that is done by Jitter happens outside.

FFT data is stored as 32-bit floating-point numbers in Jitter matrix. This is the same data type as it is, according to Jones and Nevile (2005), used by MSP for audio representation. Named jitter matrix is basically an object through witch we store, retrieve and modify the data that is actually stored at specific place in computer's memory. Hence any *jit.matrix* with the same name have access to the same place in computer's memory. We could say that equally named matrixes hold the same data and hence patch chords between them are not required. This is one reason why storage and processing of spectral data can be placed outside the *pfft~* object. The second reason is that the speed of processing and the way we process FFT data is completely arbitrary. On the other hand writing and reading the (processed) FFT data has to be sample accurate and is hence done inside *pfft~* subpatcher.

Frequency domain is hidden inside the two *pfft~* objects/subpatchers named *createSpectralMatrix* and *PlayBackSah*. First subpatcher performs STFT and writes spectral data into matrix called *spectrum* while the other performs iSTFT and reads the data from the matrix named *spectrumTransformations* (this matrix is not visible in figure 4.1).

STFT parameters are defined in both *pfft~* objects and should be the same for both. These are window size, overlap factor and window function (the default is Hanning). The number of frames that we intend to plot in our spectrogram can be chosen from object *umenu*. Number of frames defines the number of columns in both matrixes (*spectrum* and *spectrumTransformations)* and are crucial for calculating the speed of writing and reading.

Audio path begins with *ezads~* object and goes directly to object *pfft~* named *createSpectralMatrix*. FFT frames produced in that spectral subpatch are written into matrix called *spectrum*. The content of the *spectrum* matrix can be seen at the leftmost display window

in fig. 4.1 and is passed on at the rate defined by *qmetro* object. Green colour represents amplitude plane and pink colour represents phase difference plane (*The reason for green and pink lie in a peculiar way that *jit.pwindow* object or display, interprets a two plane matrix. *jit.pwindow* expects 1 (grayscale) or 4 (colour) plane matrixes. When 2 plane matrix is send to that object first plane represents first and third plane (A+G=G) while second plane represents second and fourth plane (R+B=pink).). Matrix *spectrum* is then send to the spectral processing part of the patch that is not visible on fig. 4.1. The processed FFT data is stored in matrix called *spectrumTransformations* and sent back to main patch where the two matrix planes get separated using the object *jit.unpack*. Amplitude plane is vertically inverted before the display so that low frequencies are at the bottom as it is practice with sonograms. Slightly rotated amplitude plane shows that some spectral processing took place.

4.2 Creating the Spectral Matrix – Inside the object *pfft~* – Frequency Domain

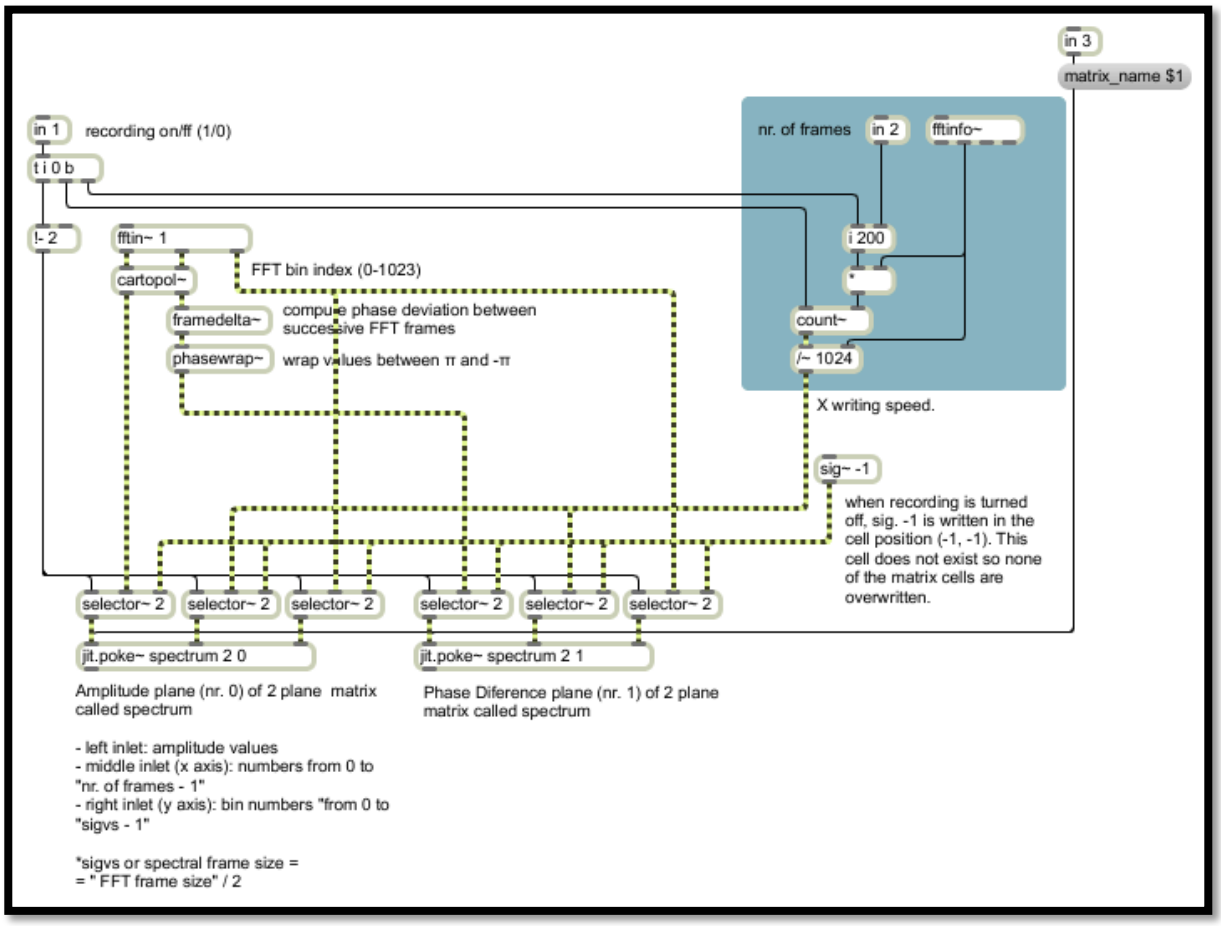

Figure 4.2 Inside the pfft~ subpatcher where STFT and writing into spectral matrix is performed (original).

fftin~ object serves as in input for time domain signals and the output of spectral data. It outputs the data in the form of complex numbers (Cartesian coordinates) for the bin response for each incoming sample. The *cartopol~* object translates Cartesian coordinates to polar coordinates. Polar coordinates consist of angle and distance or in other words phase and magnitude. Instead of

using *cartopol~* the conversion can be done using clever mathematical trick based on basic operations such as addition, subtraction and multiplication, that is cheaper for the CPU. Such example can be found in the PV implementation by Dudas & Lippe (2006). *framedelta~* object "computes a running phase deviation by subtracting values in each position of its previously received signal vector from the current signal vector" (Dobrian, 2001, *framedelta~* help). Phase wrapping is done by object *phasewrap~* and wrapped phase difference is passed to object *jit.poke~*. *jit.poke~* "object writes the value specified by the leftmost signal input into one plane of a matrix cell at the position specified by"(Dobrian, 2001, *jit.poke~* help) middle (x) and rightmost (y) signal inputs. Magnitude from the leftmost output of object *cartopol~* goes directly into the remaining *jit.poke~* object. Both *jit.poke~* objects write in two plane matrix called *spectrum* that is placed outside the spectral subpatch. The number of columns is optional while the number of rows should be the same as spectral frame size (half the FFT frame size). *jit.poke~* that receives magnitude values writes in plane 0 of matrix *spectrum* and *jit.poke~* that receives phase difference values writes in plane 1. The two signals that define writing position or matrix cell are the same for both *jit.poke~* objects. The signal that defines y writing position (row) comes from the rightmost output of *fftin~* that outputs FFT bin index and therefore changes at the speed of *overlap factor * sampling rate* (sampling rate inside *pfft~*). The signal that defines x writing position (column) has to increase n-times slower than the sampling rate inside *pfft~,* where n = *spectral frame size*. In other words, when all the rows of column 0 are filled with values, x writing position should increase by 1 going to column 1. In classical programming environment this could be solved with *if* statement like the following: "if y equals *spectral frame size – 1*, add 1 to x". In Max we could simply use *count* object to do the task but in MSP the implementation is a bit trickier. If we set the *count~* object to count from 0 to *the number of all*

cells in spectral matrix (**count*~ never reaches the limit value therefore no -*1*) and divide the output by the *number of rows* and comply with only integer values, we get the desired speed of x increment. The last part, ignoring the float values, is automatically done by *jit.poke*~ since it accepts only integers.

The only function of *selector*~ object is to enable or disable recording into spectral matrix. When recording is enabled *selector*~ passes through all the signals that are headed towards *jit.poke*~. When disabled, *selector*~ passes through constant signal -1. When *jit.poke*~ receives signal -1, it stops writing.

4.3 Reading the Transformed Spectral Matrix – Inside the Object *pfft*~ – Frequency Domain

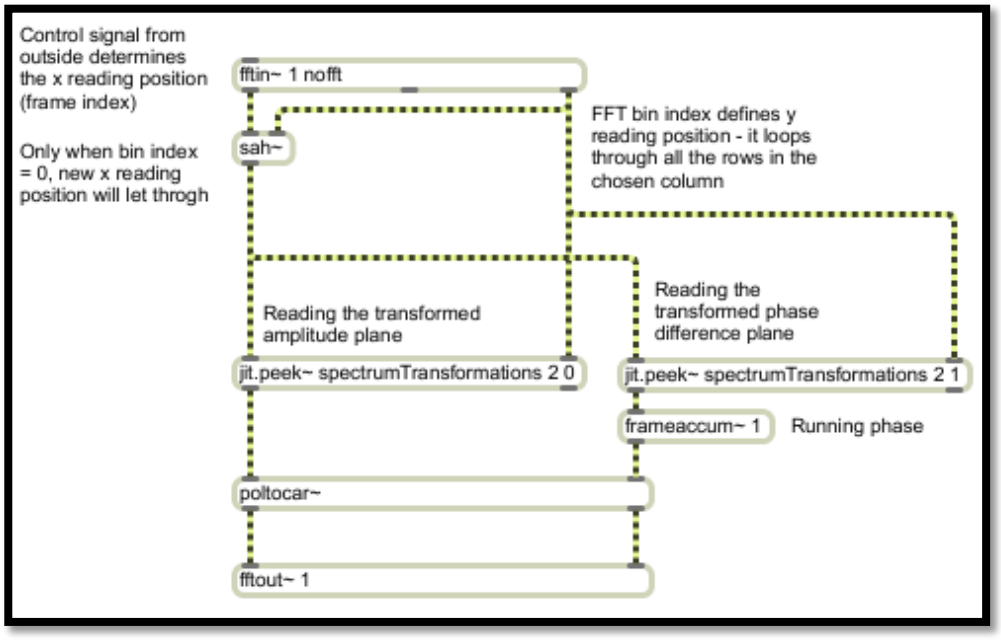

Figure 4.3 Inside the *pfft*~ subpatcher where inverse STFT and reading of transformed spectral matrix is performed (original).

In order to get the audio signal in *pfft~* object without time to frequency domain transformation, an argument *nofft* in object *fftin~* can be used. Incoming audio signal is outputted through leftmost outlet of *fftin~* while the rightmost output is the same as when actually using FFT – it loops through all the FFT bin indexes or in our case, rows of spectral matrix. Although *nofft* argument is used, FFT size still has to be defined as an argument to *pfft~* (fig. 4.1).

The audio signal from the leftmost output of *fftin~* defines the horizontal reading position of matrix (fig. 4.3). Therefore it spans between 0 and the number of columns in spectral matrix. The role of *sah~* object is to let through the horizontal reading position only when the vertical reading position is 0. Given that, the horizontal reading progresses in time steps long FFT frame size (samples to milliseconds conversion should be calculated with sampling rate inside *pfft~*).

jit.peek~ is designed to read the matrix with audio signals. Both *jit.peek~* objects read matrix with the name *spectrumTransformations*, which is the matrix that stores the processed data from matrix *spectrum*. Left *jit.peek~* (fig. 4.3) reads the (transformed) amplitude plane and right *jit.peek~* reads (transformed) phase difference plane. The values from phase difference plane goes to *frameaccum~* object where running phase is calculated. Since *fftout~* expects FFT data in Cartesian format, object *poltocar~* is used before inverse FFT.

4.4 Mapping the Display Window to Spectral Matrix

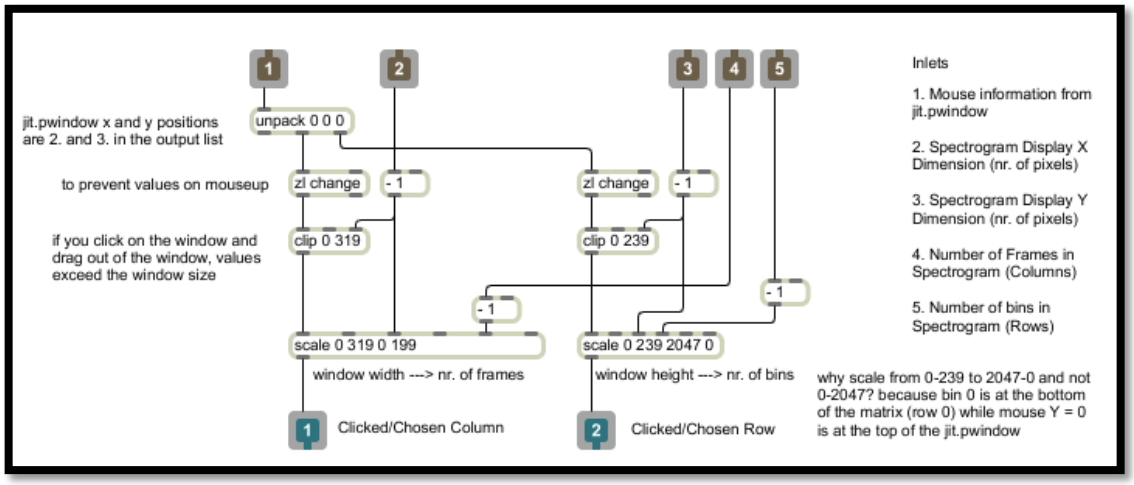

Figure 4.4 Mapping the Display Window to Spectral Matrix (original).

For the sake of spectral processing that depends on interaction with sonogram (clicking on the display), the display window has to be correctly mapped to the spectral matrix. On fig. 4.4 we see a subpatcher where mapping takes place. Mapping is basically all about scaling *x* and *y* of the display to *x* and *y* of spectral matrix. That is one by object *scale.*

In the process of scaling, there are only two variables. These are *x* and *y* of the clicked position. All the rest are constants and come into the *subpatcher* through inlets 2-5. The constants are *X* and *Y* dimension of display and *number* of columns and rows in spectral matrix. *x* and *y* of the clicked position come from *jit.pwindow* object (display) though inlet 1.

jit.pwindow outputs a list of information when the display is clicked. Second and third item on the list are *x* and *y* positions that are separately forwarded to two *scale* objects. *zl change* object

filters out any list repetitions and in that case blocks values from *jit.pwindow* on *mouseup*. *clip* object prevents values that are outside the range of *jit.pwindow* area. These values may occur if one clicks on *jit.pwindow* and drag the mouse out of display area.

For an option of logarithmic spectrogram see figures 4.5 and 4.6.

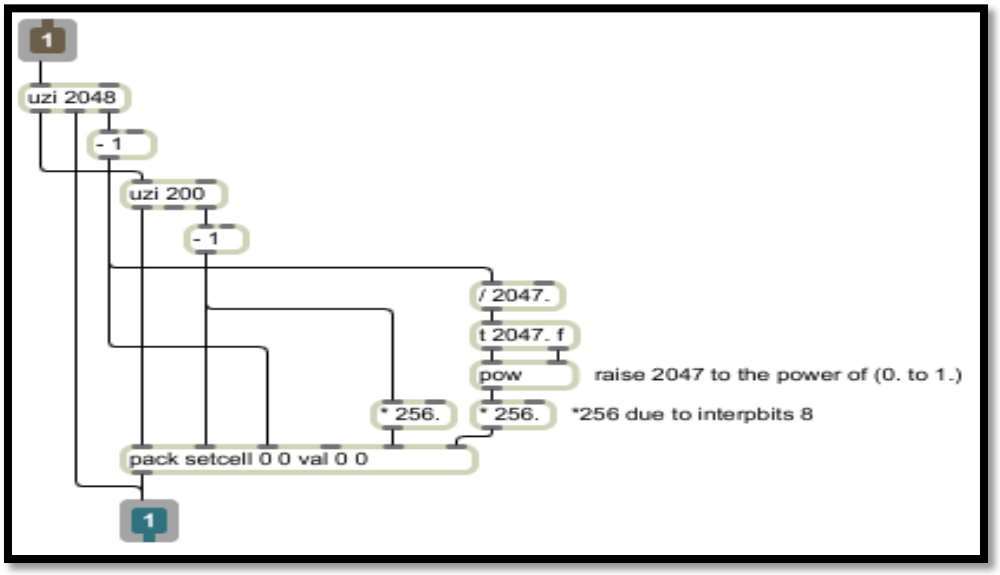

Figure 4.5 Creating x and y spatial maps for *jit.repos* in absolute mode at interpbits 8 (original).

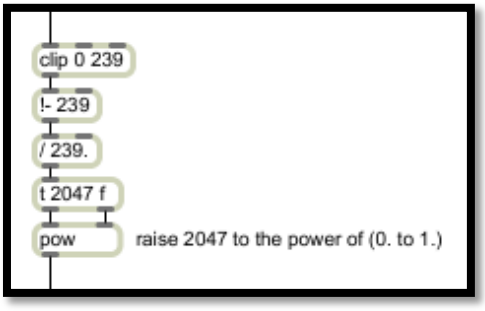

Figure 4.6 The rightmost *scale* object from fig. 4.4 should be replaced by the lower four objects (original).

We will return to logarithmic spectrogram in section 5.3 (Local transformations).

4.5 Default Horizontal Speed of Reading the Spectrogram

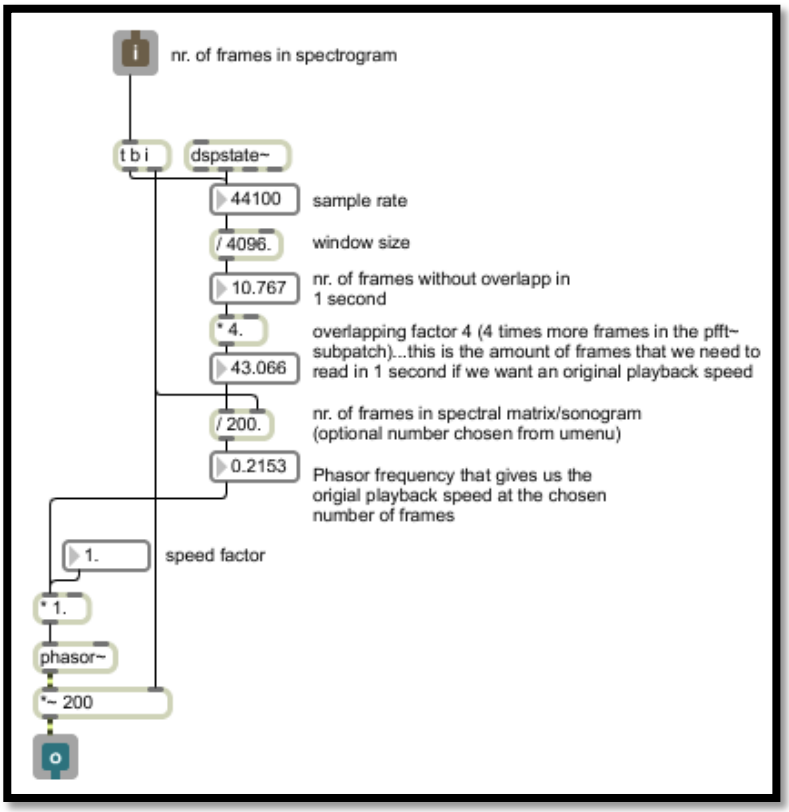

Figure 4.7 Frame-dependent reading speed of spectrogram; number of frames (x size) in spectrogram equals 200 (original).

The *phasor~* object generates signal that goes linearly from 0 to 1 (saw waveform). Hence it is generally used as an object to read samples in samplers when output ramp is scaled to fit the desired range. The concept of reading the spectrogram in horizontal direction is the same as reading in conventional samplers.

When reading samples with *phasor~,* there are two variables that defines the speed of reading. One is frequency of *phasor~* and the other is *range* (defined by scaling the output of *phasor~*). These two variables are both proportional to the speed of reading. Therefore each chosen *x* size (nr. of columns) of spectral matrix demands its own reading frequency if we want a playback at original speed.

The idea behind this patch is that the speed of reading can be controlled with one parameter – *speed factor*. If *speed factor* is multiplied by the *original playback frequency* then *speed of reading* is proportional to *speed factor*. In other words, the *speed of reading* is determined by multiplication of *speed factor* by *original playback frequency,* resulting in original playback speed for factor 1, double speed for factor 2, half speed for factor 0.5, etc.

Number of FFT frames that are produced in *one* second equals *M* = (*sample rate*/*window size*)**overlap factor.* If our spectrogram would consist of *M* columns, *phasor~* with frequency 1Hz (and with output scaled from *0 to M*), would give us the original speed. Given that, in order to calculate the original playback frequency we divide *nr. of frames* with *M*.

Chapter 5

Implementation and Critical Evaluation of Existing and Unexisting Real-time ADSSP Techniques in Max/MSP

We will start with the proposed solution to CPU limitation problem and continue with modifications of existing ADSSP techniques, adaption of existing spectral processing techniques to matrix-based sonographic world, exploration of possibly new ADSSP methods and integration of existing approaches into new ones.

ADSSP techniques will be divided into five categories: optimization, global transformations, local transformation, spectral interaction and FFT data processing on GPU.

5.1 Optimization

The five techniques presented in this section could be used along all further experiments because they concern the way we read the spectrogram and the way we process spectral data in general.

5.1.1 Moving Area of ADSSP According to Reading Position

The reasoning behind this idea is that processing of small matrixes does not demand huge processing power. Therefore if we process only a small area around reading position we could use high quality analysis-resynthesis parameters on large samples (matrices) without the need of extremely powerful CPU.

The size of the area depends on effects, parameters used and of course personal preferences. For instance if we use the effect that smears the picture in x direction and set the parameters to smear one cell across next 20 cells (pixels), then it might be sensible to chose a 40 pixels wide processing area (fig. 5.1). In such case we could hear the gradual introduction of the effect.

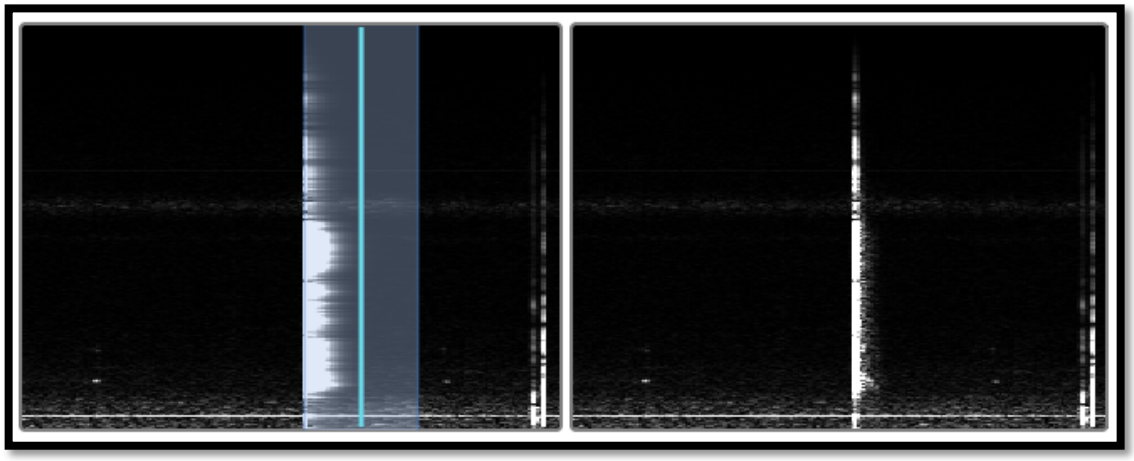

Figure 5.1 Picture of processed (left) and unprocessed (right) spectral data. Blue interval indicates a 40 pixels wide processing area. The graphical effect was set to smear each pixel in right direction for 20 pixels. If the reading direction is left, the gradual introduction of smearing effect will be heard. The audible result would be a reverse spectral smearing effect (original).

The implementation is based on equally named matrices (*composit*) and submatrices (fig. 5.2). The spectrum of the whole sample is stored in matrix called *completespectrum*. That matrix gets divided into two separate matrixes. One becomes for a moment a copy of *completespectrum* and is called *composit.* The other is a subsection of *completespectrum* at the size of chosen processing area. The content of this subsection or submatrix depends on reading position and size of processing area. Submatrix is then separately processed and written into the already mentioned matrix called *composit.* In other words, submatrix overwrites the content of the matrix *composit.* Which part of matrix *composit* is overwritten is defined by the reading position.

On fig. 5.2 only amplitude plane of FFT spectrum is shown although the described process applies to both planes. Since the visual representation of phase plane is practically impossible to interpret only amplitude plane will be shown in the following sections of this paper.

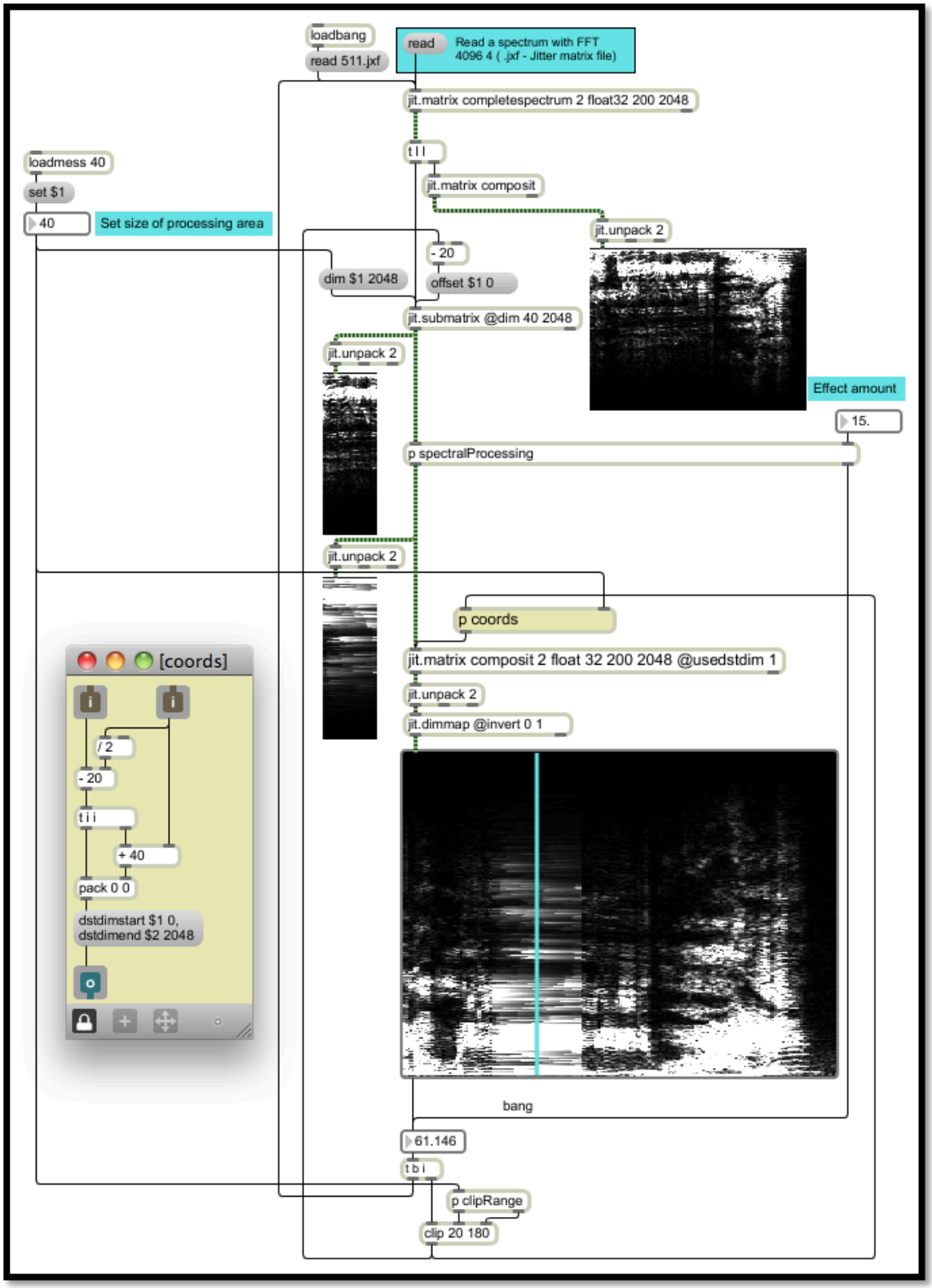

Figure 5.2 Moving area of ADSSP according to reading position.

5.1.2 Horizontal Blur

Horizontal blur is a technique presented by J. F. Charles (2008). He named it blur or stochastic synthesis. The reason we call it horizontal blur is because we will present a different approach to this graphical effect with extended options of blurring in later sections (5.2.1 and 5.5).

Horizontal blur is a technique for removing a frame effect of phase vocoder. It is in a way an adaptation of Nobuyasu Sakodna's technique (used in his MSP Granular Synthesis patch v2.5) for PV (Sakodna, 2000). Sakodna's audio driven synchronous granular sampler used *noise~* object as a random generator at audio rate. Scaled random values from *noise~* object served for a stohastic grain offset. The result was blurry, reverberant or frozen sound effect that increased the quality of playback since it replaced the static mechanical looping at slow playing speeds. Charles's idea in case of PV is very similar – using *noise~* object for random *bin* selection between neighboring FFT frames during vertical reading (fig. 5.3).

The implementation is very simple. Scaled noise is added to horizontal reading position and *sah~* object, that was assuring accurate and synchronized vertical reading, is removed from *pfft~* object (fig. 5.4).

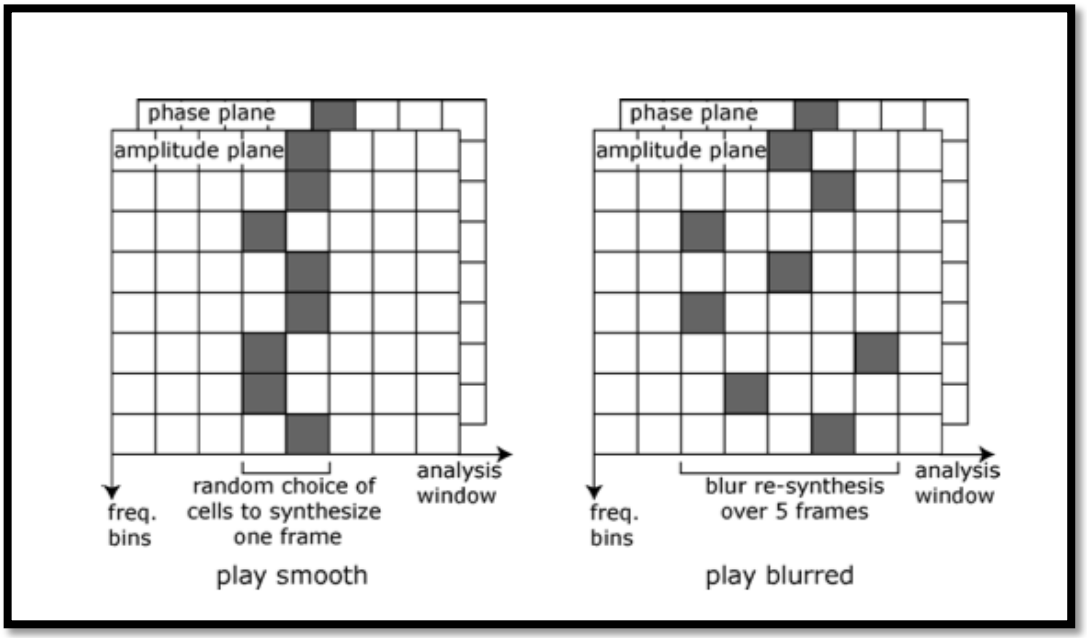

Figure 5.3 Schematic representation of horizontal blur (Charles, 2008).

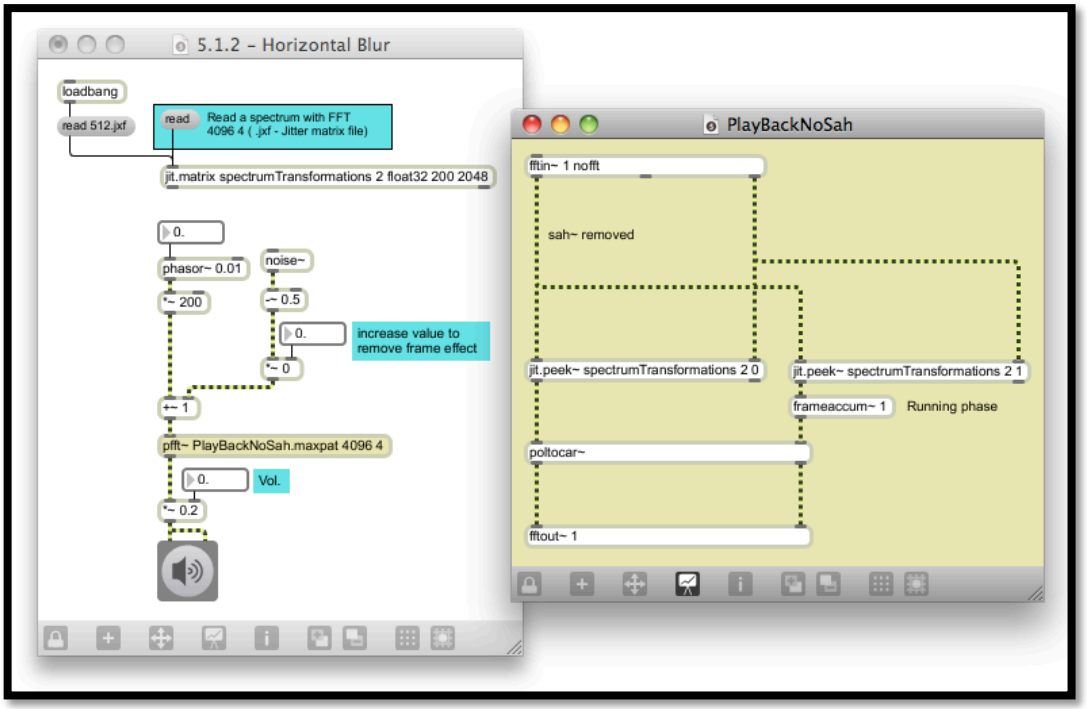

Figure 5.4 Horizontal blur (original).

The benefit of horizontal blur based on MSP is that it works at audio rate and presents only negligible burden for CPU. Also the "CPU use is constant regardless of the parameters, including playback rate" (Charles, 2008). The weakness of this technique lies in introduction of vertical phase incoherence.

Horizontal blur has actually a double function. Beside removing the frame effect while reading the spectrogram very slowly it also removes the effect of mechanical looping when reading only one frame (horizontal reading speed equals 0). That is jet another benefit of this technique.

5.1.3 Frame Interpolation

Frame interpolation is another technique from J. F. Charles for removing the frame effect of PV (2008). The idea is that we are constantly reading a matrix that consists of only one column. That column is an interpolated frame between two successive frames. To do that we need two single columned submatrixes and one single column matrix. One submatrix is copy of a current frame, second submatrix is a copy of a next frame and remaining matrix is an interpolation between both frames (*if using object *jit.submatrix*, a submatrix is not an actual copy but only a reference). As we move from frame 0 to frame 1, interpolated frame is constantly calculated for every position. For instance when we are at reading position 0.1, the interpolated frame consists of frame 0 (90%) and frame 1 (10%). There are 4294967295 possible interpolations between two frames when using 32-bit floating point for horizontal reading.

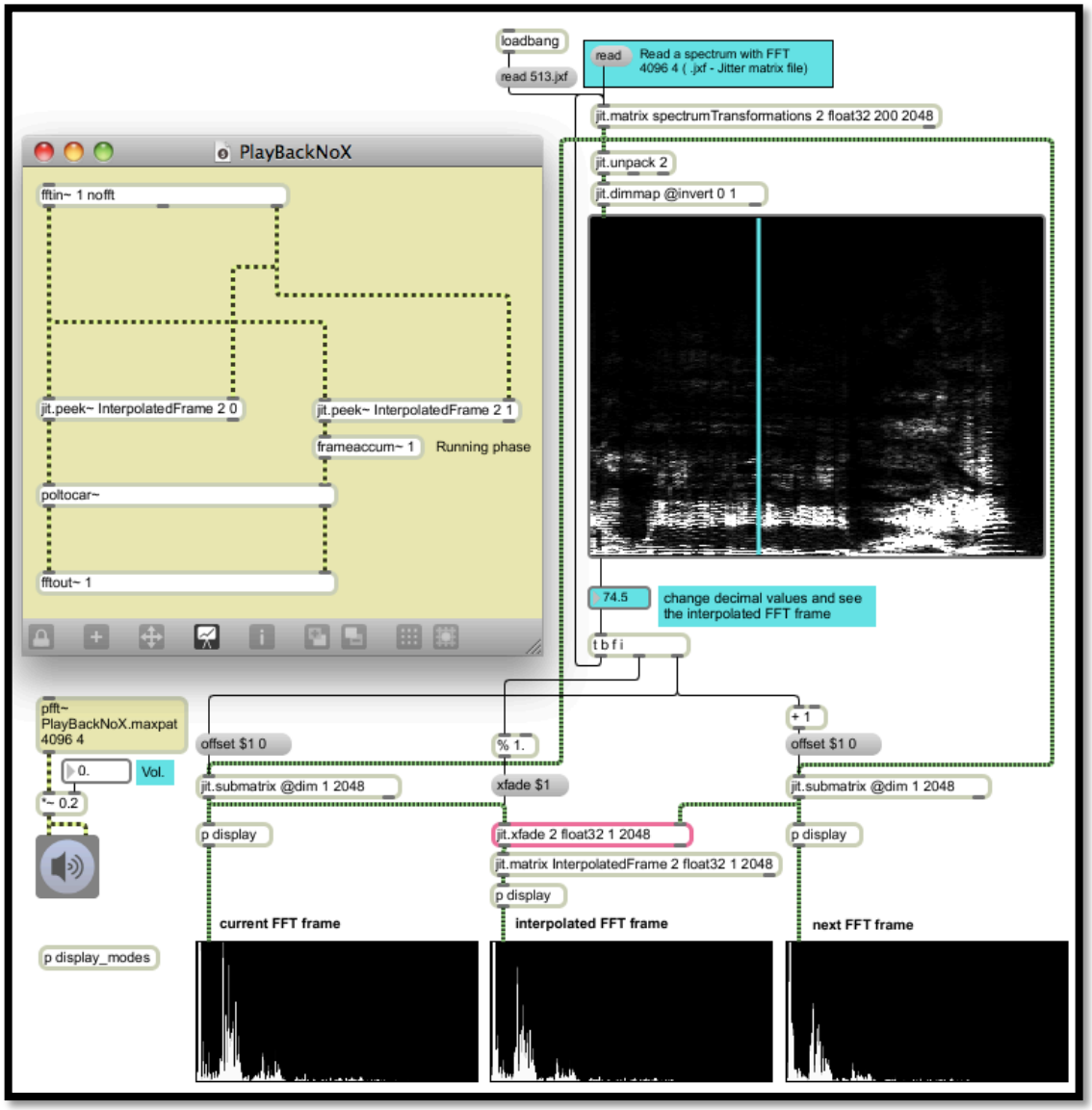

Figure 5.5 FFT frame interpolation (original).

In implementation (fig. 5.5) we read spectrum in x direction with float values. By trimming of the decimal part of reading position, the integer reference or sub matrix offset is provided. Hence when reading the spectrum at position n (float), the two submatrices consist of columns N (integer) and N+1 (integer) from original matrix called *spectrumTransfrmations*. The decimal

part is obtained with object *modulo* that outputs the reminder of a division. All the incoming float values are divided by 1, resulting in output range between 0.00 and 0.99. At figure 5.5, the reading position is 74.5. That means that first submatrix represents column 74, second matrix represents column 75 and *modulo* outputs value 0.5. Output from *modulo* is passed to *jit.xfade* where the interpolation between submatrixes is performed.

pfft~ object named *playBackNoX* has no signal input for horizontal reading position. That means that MSP patch chords going to left inlets of *jit.peek~* objects receive value 0 at audio rate, meaning that horizontal reading position is contantly 0.

The main advantage of that frame effect removing technique is that it preserves vertical coherence. That results in more naturalistic sound that is also very smooth. But if we "freeze" the sound (playback speed equals 0), we can hear the mechanical (vertical) looping because we are constantly reading one and the same (interpolated) frame. The horizontal blur technique from previous chapter 5.1.2 was a great solution for such problem but unfortunately it can not work on a single frame. J. F. Charles continued his experiments with horizontal blur technique (2008) but we will hold on to frame interpolation technique during our further explorations. Actually we are suggesting an extension to frame interpolation that enables the integration of horizontal blur and is presented in the following chapter (5.1.4).

* Frame interpolation can be also performed on GPU with slab td.rota.jxs (see patch 01_Frame_interpolation_using_td.rota.jxs.maxpat). In Max 6, there are also options to use jit.gen instead of jit.xfade and similarly jit.gl.pix when working with slabs on GPU. One of the potential benefits of using jit.gen or jit.gl.pix is that we can implement hermite interpolated crossfade instead of linear, that gives us slightly different sonic effect.

The frame interpolation technique is efficient when using small reading speeds. When used at higher reading speeds, its beneficial effect is on one hand not noticeable and on the other hand an average computer is not capable to process all the data fast enough – hence the audio quality can drop significantly.

5.1.4 Interpolate – Concatenate – Blur (ICB)

ICB brings together best of both world – the refined, smooth and defined sound of frame interpolation technique when reading the spectrogram extremely slowly and blurry, reverberant or frozen sound of horizontal blur technique, when horizontal reading speed equals 0. There are many ways how these two techniques could be combined so ICB presents the most optimal solution found in the scope of this research.

In frame interpolation technique we are reading a fixed one column matrix named *InterpolatedFrame* that holds the data of interpolated frame. The idea behind ICB is that *InterpolatedFrame* matrix becomes flexible and that the blurring factor defines the number of its columns. So when blur is set to 1 (blur between the current and consecutive frame), one column (consecutive frame) is added to matrix *InterpolatedFrame.*. As we increase the blur factor (by multiplying scaled audio noise with integers) the *InterpolatedFrame* matrix is growing by the same integer amount and vice versa. The added columns are of course not interpolated and the impact on CPU is negligible (not noticeable in DSP status of Max/MSP).

In order to avoid audio interruptions when changing the size of flexible *InterpolatedFrame* matrix, the noise that defines the blur effect has to be scaled after the size of a matrix has increased. When decreasing the matrix size the order of events has to be opposite (noise first, matrix last). Otherwise we could be reading for a moment a non-existing frame or remove an existing one in the middle of reading.

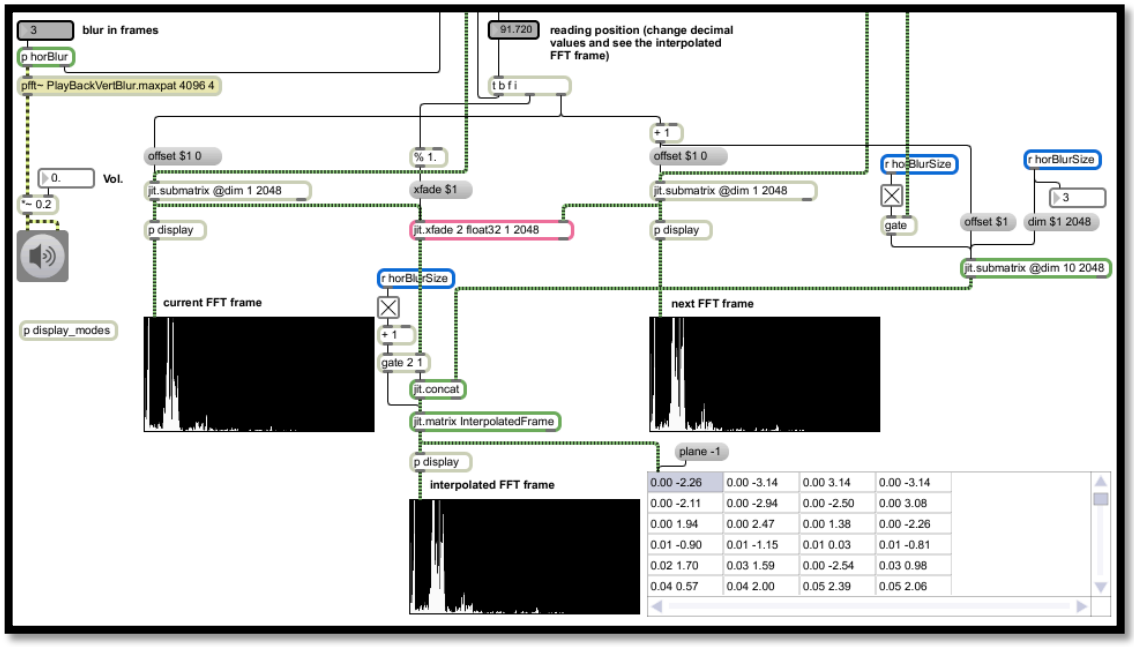

Figure 5.6 Interpolate – Concatenate – Blur (IPB). Comparing to figure 5.5, there are four main new objects (with green border). These are subpatcher horBlur (fig. 5.7), *jit.submatrix* (that receives the x size message from *horBlur*), *jit.concat* that glues the columns together and *jit.matrix* named *InterpolatedFrame* that has no defined dimensions (therefore it adopts the sizes of incoming matrixes). In the *jit.cellblock* table the content of *InterpolatedFrame* matrix can be seen. Blur factor is set to 3, x reading position is 91.72, so the four columns, from left to right are: interpolated frame 91.72, frame 92, 93 and 94 (original).

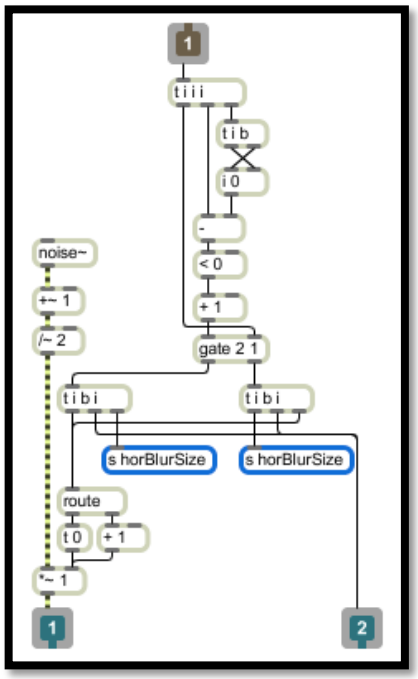

Figure 5.7 Subpatcher *horBlur* from fig. 5.6. From *inlet* to < object spans the algorithm that checks if blur factor is increasing or decreasing. According to that, an appropriate succession of actions is triggered (original).

In practical implementation of an ADSSP instrument, horizontal blur could be automatically turned on when horizontal reading speed reduces to 0.

5.1.5 Poking Holes in Interpolated Spectrum

One of the possible solutions to remove the mechanical looping effect when reading a single interpolated frame is also to multiply the amplitude plane of frame in question with a constantly changing column, filled with values 0 and 1 at random positions. That is basically stochastic filtering of the spectrum.

The benefit of this technique is that it is easy to implement but on the other hand the sound becomes a bit more thin when “frozen”. Which is not necessary a bad thing – it is just an opposite approach, in the context of removing the mechanical looping effect, as offered by horizontal blur (horizontal blur makes the sound denser).

In practical use, the poking holes technique function could be automatically turned on when the horizontal reading speed reaches 0.

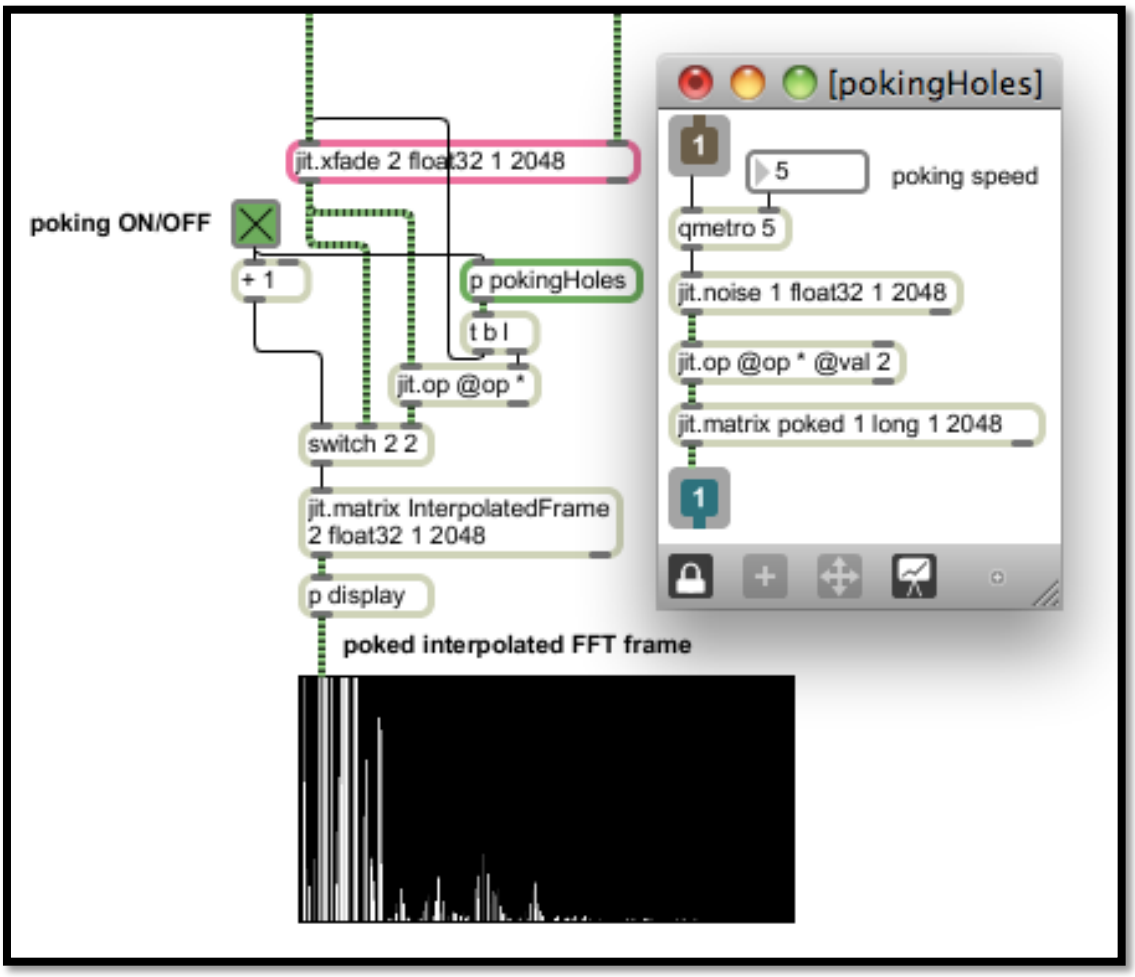

Figure 5.8 Poking holes in the Interpolated Spectrum (original).

5.2 Global Transformations

By global transformations we mean spectral processing that is applied to the whole processing area regardless of any spatial preferences of the user. The distribution of graphical effect therefore can not be applied to any specific place of moving ADSSP area.

5.2.1 Blur/Water Effect

Blur/water effect is a Jitter extension of horizontal blur presented in sub-section 5.1.2 (for Gaussian blur see section 5.5). In Jitter it is possible to produce extremely flexible blurring in x and y direction that can extend to "water" effect (picture seems like being under the wavy water). Since there is no object in Jitter that produces such an effect it has to be made with object *jit.repos* and specifically created spatial maps for that object. *jit.repos* is at the same time an object that was found as the most useful and powerful object for performing spectral processing. It has enormous musical and synesthetic audio-visual potential.

jit.repos has two modes of repositioning matrix cells, absolute and relative. For blur/water effect we used relative mode and *jit.noise* as a 2 plane relative spatial map (fig. 5.9). The output matrix from *jit.noise* is upsampled to the size of spatial map (if smaller) and also interpolated (upsampling and interpolation are vital for achieving the water effect). The 2 resulting planes represent x and y cell to cell relative repositioning map that is used for amplitude and phase difference dislocation.

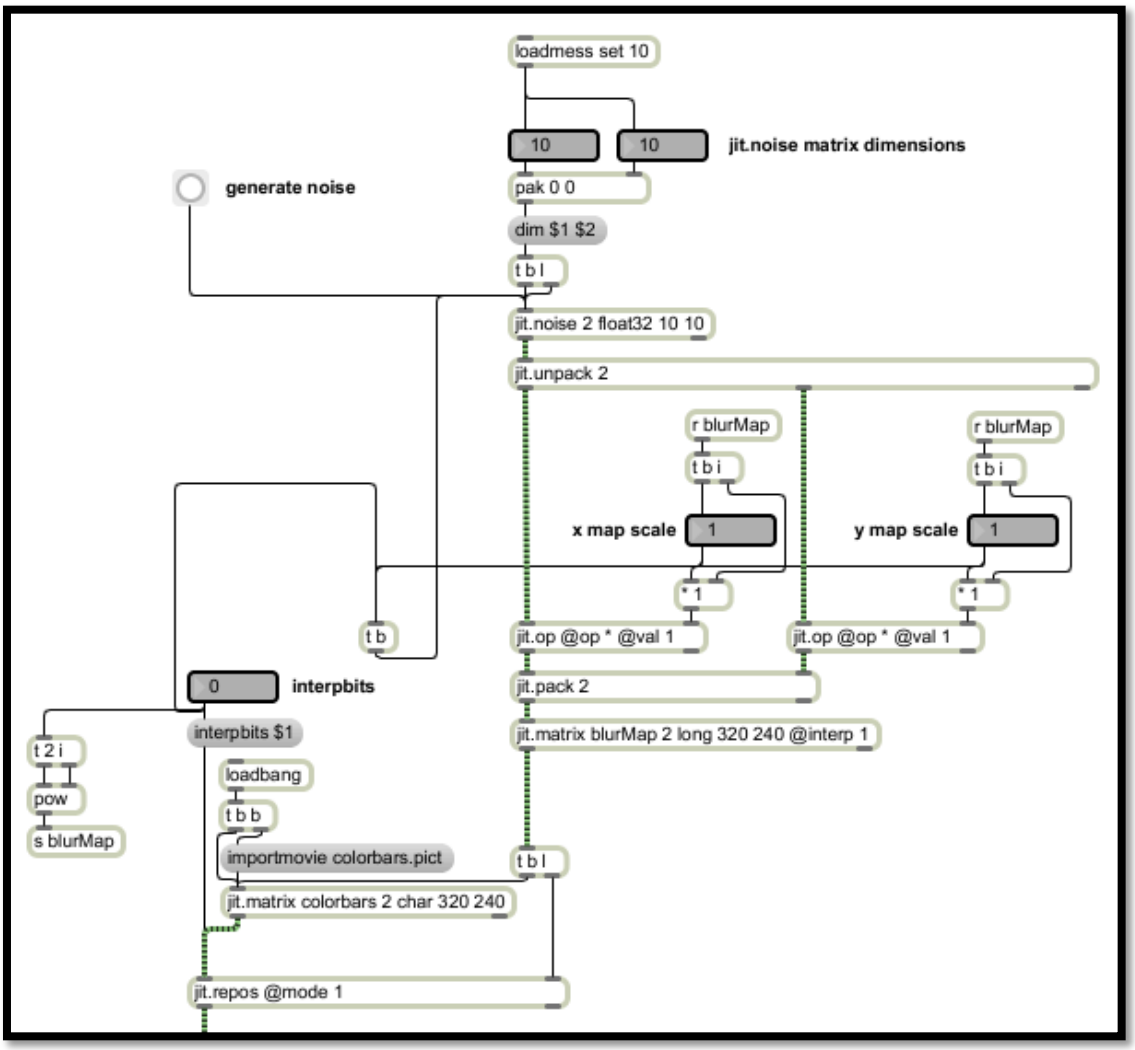

Figure 5.9 Implementation of blur/water effect with upsampled and interpolated *jit.noise* matrix as a 2 plane relative spatial map for *jit.repos* (original).

As it is evident from figure 5.9, there are 5 parameters that control blur/watter effect – x and y dimensions of *jit.noise* output matrix, scalar scaling of both *jit.noise* matrix planes and the number of *interpbits,* that is an attribute of *jit.repos*. Scaling factors are both automatically multiplied by 2^n, where n is a number of interpbits.

The result of various parameter combinations can be seen on figures 5.10 to 5.16. Picture *colrbars.pict* that comes with Max/MSP was used for these examples as the visual effect is better seen.

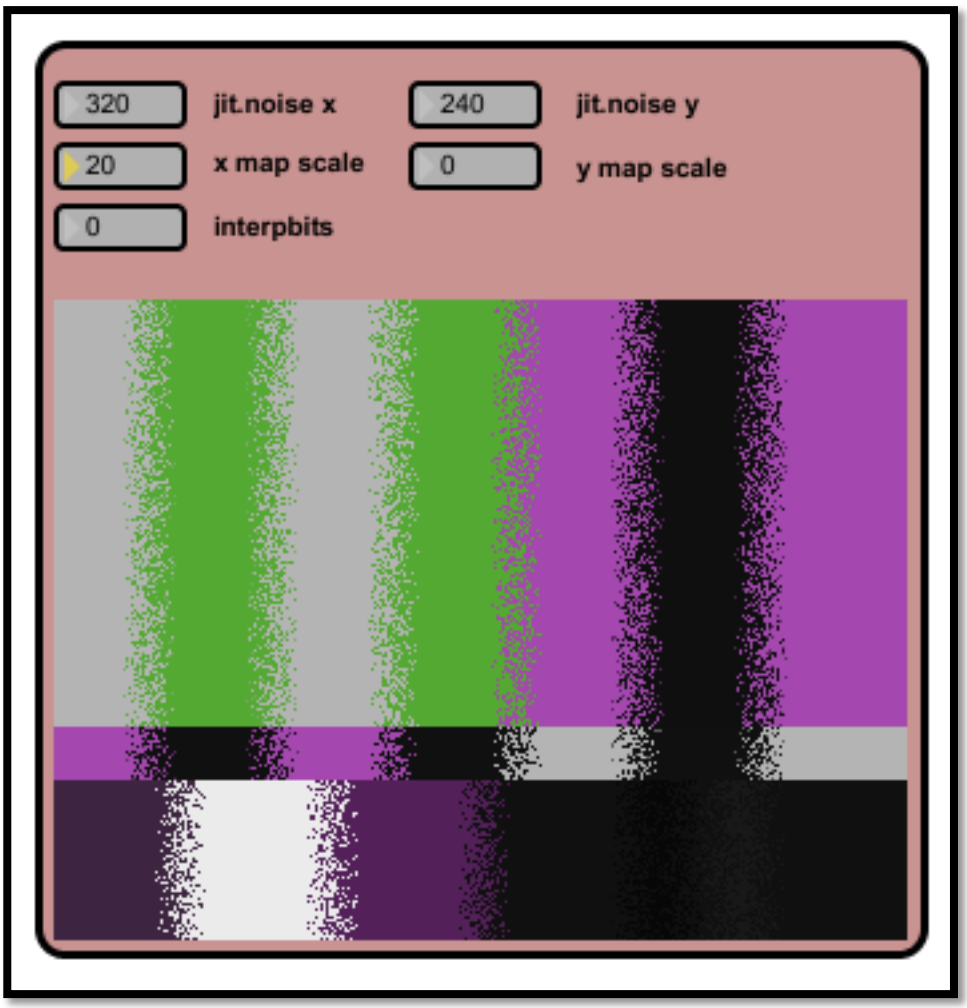

Figure 5.10 Replication of horizontal blur effect presented in chapter 5.1.2 with Jitter. The speed of generating noise matrices can be controlled by object *qmetro.* For this example, the size of a jit.noise matrix is equal to the size of spatial map and also the input matrix (colorbars picture). Hence the interpbits attribute is set to 0. To achieve blur only in horizontal direction, y map is scaled to 0. Horizontal lines remained intact (original).

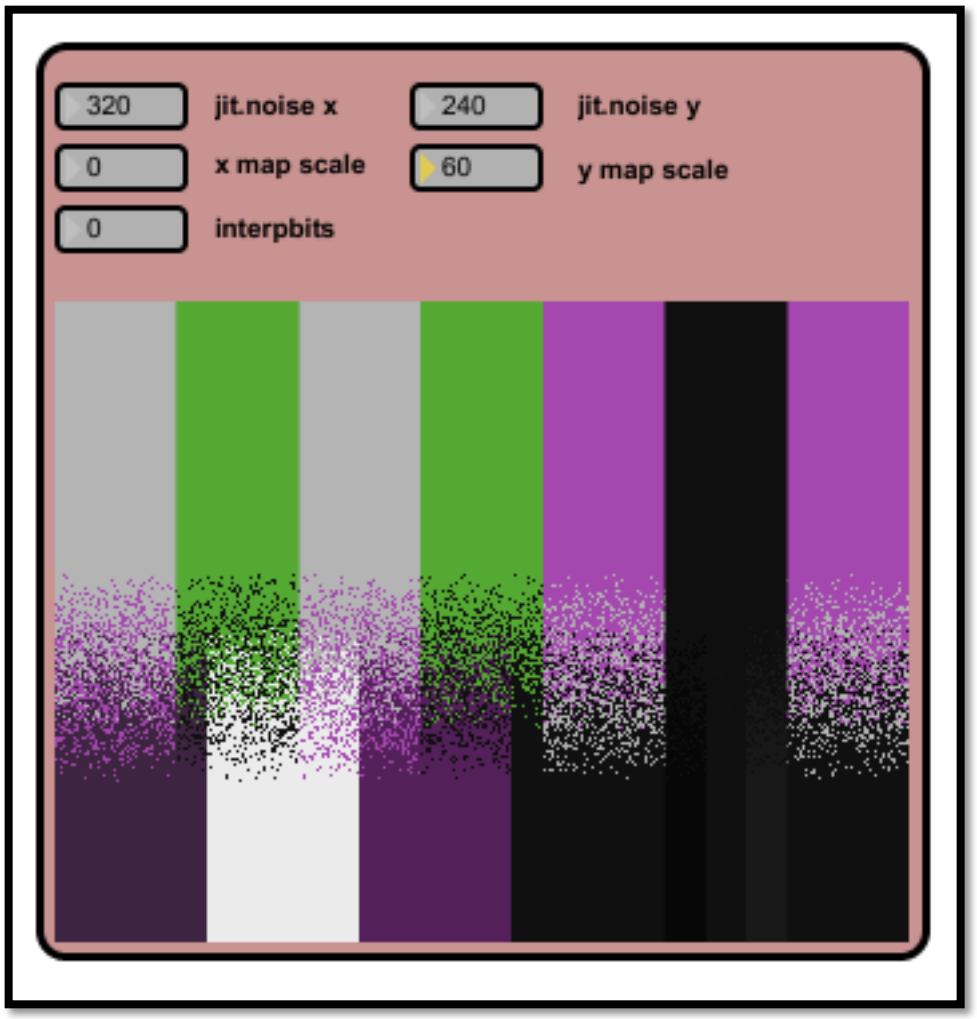

Figure 5.11 Vertical blur. The same example as presented in fig. 5.7 except x map is scaled to 0 this time. Vertical lines remained intact. If visual effect is not preferred, vertical blur should be used on interpolated frame only to save the CPU costs. The main difference between horizontal and vertical blur is that the later one performs a stochastic frequency shifting. That is also a reason why this technique is not an option for the mechanical looping effect removal. Although in practice, if FFT frequency resolution is high and vertical blur very small, the frequency shifting effect is hardly noticeable (original).

Figure 5.12 Upsampling of very small noise matrix gives us few rectangles (5x6 in this example) filled with equal values (5x6 different values or 5x6 differently coloured rectangles). Interpolation smoothens the differences in values between rectangles that visually results in smooth fading between differently coloured rectangles. When such a matrix is used for repositioning the straight lines become askew. This gives us a slight water effect (original).

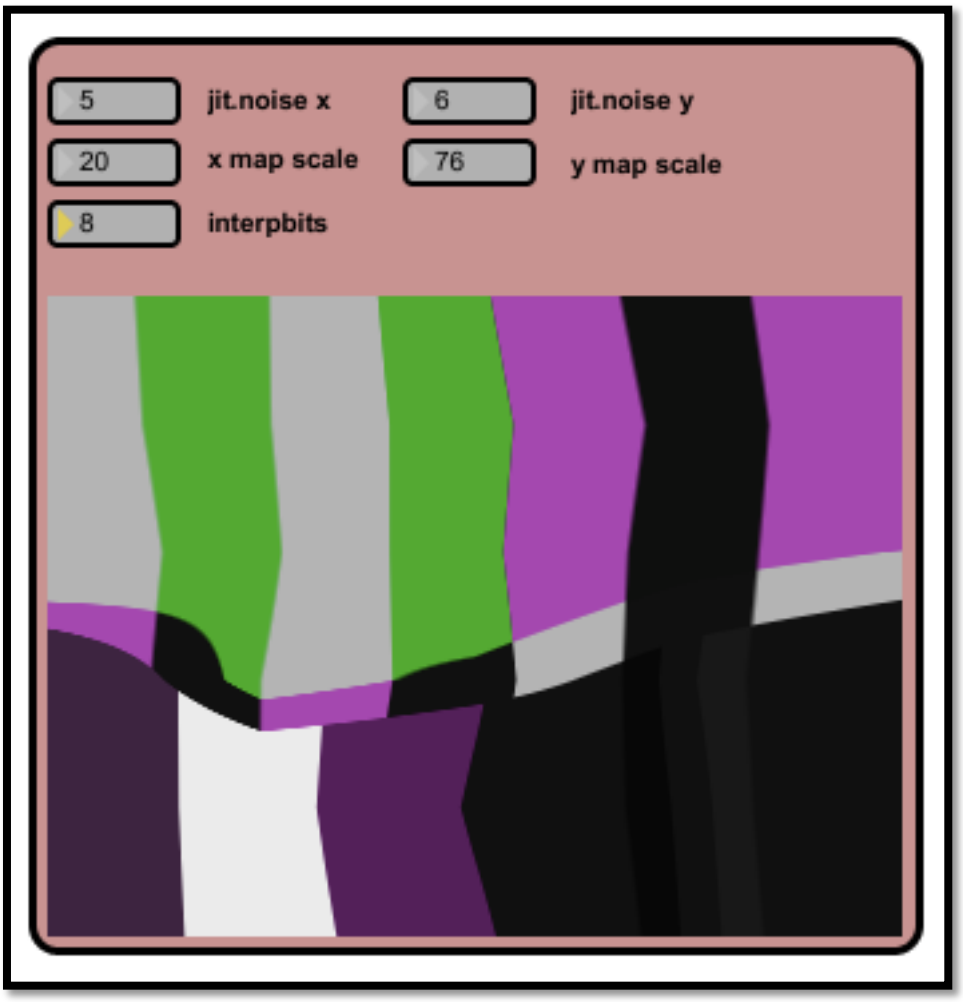

Figure 5.13 The same as figure 5.12 with exception of interpbits being set to 8. Jagged lines are smoothed out with fractional repositioning (original).

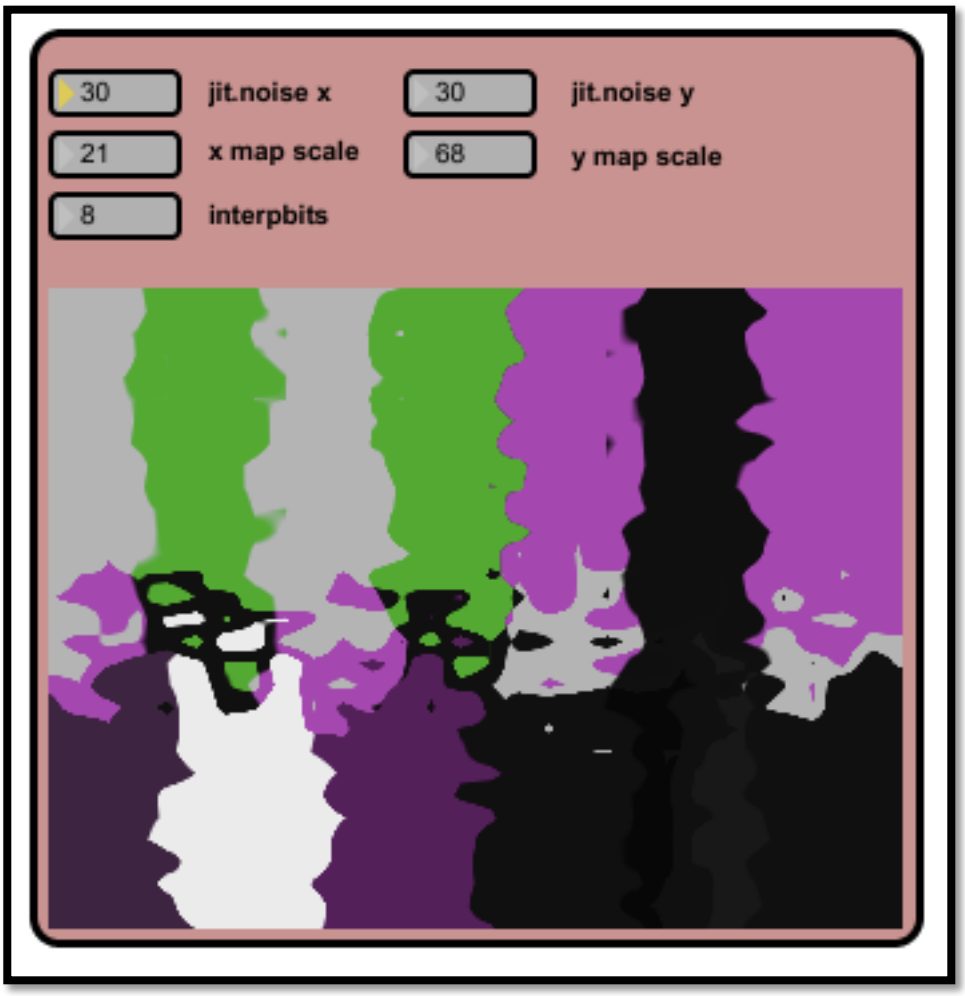

Figure 5.14 The same as figure 5.13 with exception of larger noise matrix. Water effect becomes stronger (original).

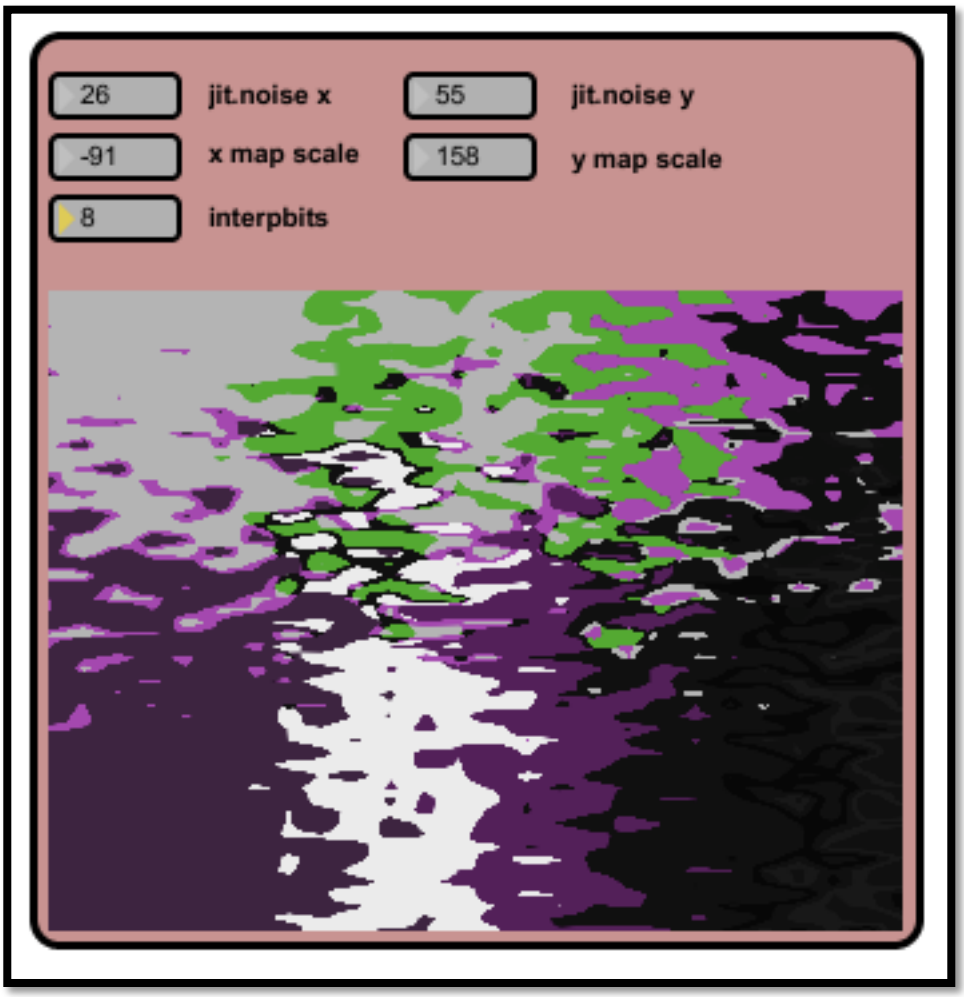

Figure 5.15 Large scaling of noise matrix values results in even stronger water effect. In combination with upsampling and interpolation the picture becomes like an archipelago. The distances between "islands" increase with scaling and the size of "islands" increases in opposite relation to the size of noise matrix. (original).

The power of water effect is that it repositions time and frequency events in a "continuous" way. Resulting sound is therefore morphing like a surface of a water.

When this effect is used on moving processing area, the visual (and audible) effect changes as we move along the spectrogram, without driving *jit.noise* with a metro. On the other hand the

jit.noise should start outputting new matrixes in order to preserve the morphing visual and audio effect when x reading speed equals 0.

Another way to create similar blur effects is to generate spatial maps for *jit.repos* with *jit.bfg*. Using the fractal categories of basis function for instance gives us an interesting kind of fractal blur.

5.2.2 Spectral Smear

Spectral smear uses object *jit.scanslide* to smear the amplitude spectrum in horizontal direction. This technique is called spectral delay by Sedes, Courribet and Thiebaut, since the "STFT bins [are] delayed graphically" (2004). Our version of spectral delay will be presented in sub-sub-sub-section 5.3.2.2.2.

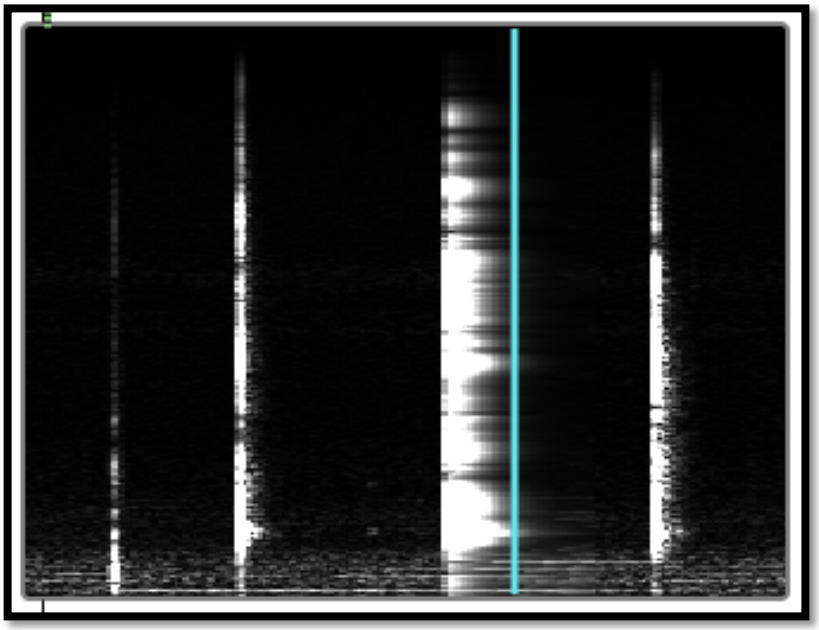

Figure 5.16 Spectral smear applied around the reading area (original).

5.2.3 Lock to Grid

Lock to grid multiplies phase difference plane by scalar in range from 0 to 1. When using interpolated frame technique for reading the spectrogram it is recommended to process only interpolated frame. The maximum effect of force to grid is achieved when phase difference plane is multiplied by 0. No phase difference means constant frequencies and no frequency deviation from the center of FFT bins. Frequencies are therefore locked to harmonic FFT grid.

* For GPU version see 02_lock_to_grid_on_GPU.maxpat.

5.2.4 Spectrum Shift, Scale and Rotation with jit.rota

Frequency or spectrum shift can be achieved by many jitter objects simply by sending the message *offset_y $1*. For classical spectrum shift *boundmode* 1 (=clear) should be used, which leaves no traces after moving the content of the matrix. Using *boundmode* 2 (=wrap) the frequencies that should disappear from the spectrogram reappear at the opposite site that gives us kind of circular frequency shift.

Spectral scaling can be done by vertical zooming. It is recommended that y anchor point is set to 0 so the spectrogram is fixed at the bottom. *boundmode* should also be set to 1. In wrap mode shrinking the spectrum will generate new spectrum clones that will get smaller in size as they grow in number (fig.5.17).

Spectrum rotation is another straightforward technique with jit.rota. The effect of spectrum rotation can be seen on fig. 5.18.

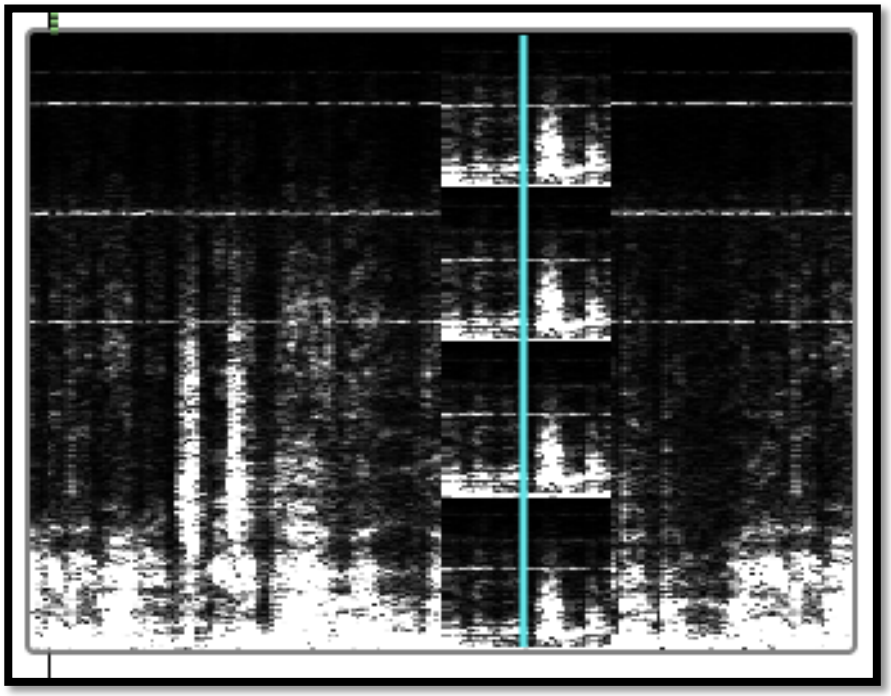

Figure 5.17 Vertical zooming or spectrum scaling with *jit.rota* in *wrap boundmode*. Zoom 0.25 generates 4 scaled spectrums (original).

Horizontal zoom and offset generate the same results as changing the speed of reading. The effects that can be achieved with *jit.rota,* from pure technical perspective, present a combination of Spectral delay (5.3.2.2.2) and Frequency warping (5.3.2.2.1).

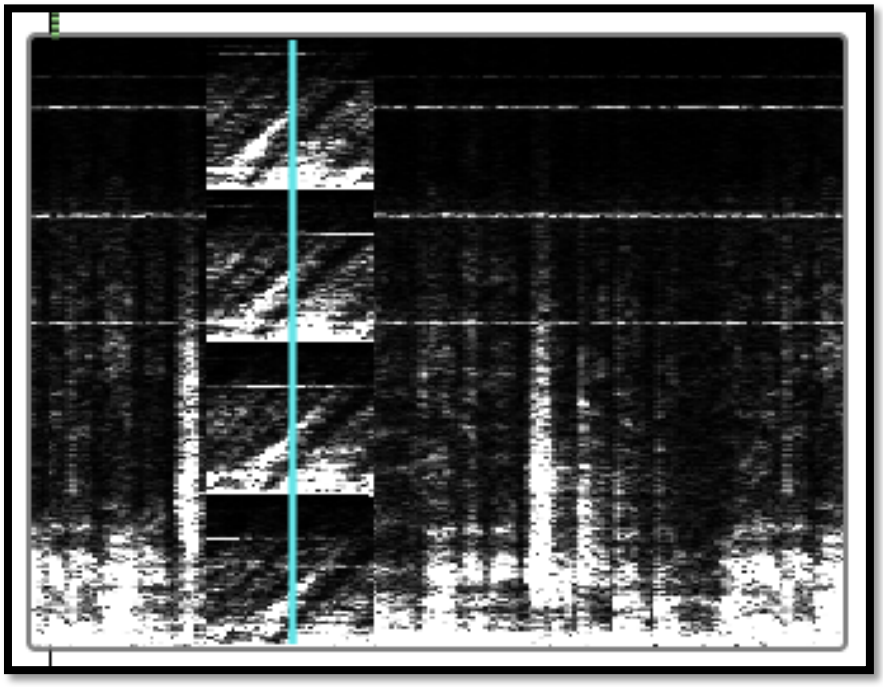

Figure 5.18 Spectrum rotation of example presented in fig. 5.17. x and y anchor points were set to 0 and half the processing area (reading position) (original).

5.2.5 Time Scramble

Time scramble is a random brother of frequency warp effect and operates in time instead of frequency. Implementation is presented on fig. 5.19. For details on mapping please see the sub-sub-sub-section Frequency warp (5.3.2.2.1).

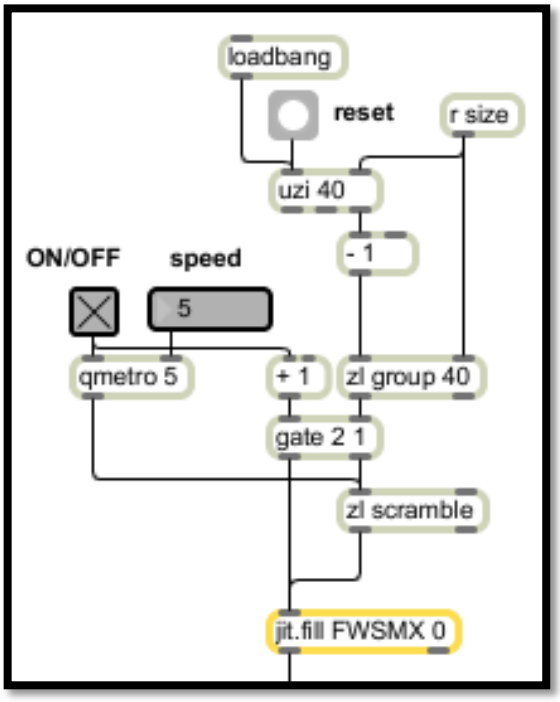

Figure 5.19 Implementation of x spatial map for *jit.repos* (original).

Time scramble randomly relocates columns or FFT frames. Linear map created by object *uzi* is stored in a list and written in matrix with object *jit.fill.* The order of values in a list that produce linear horizontal mapping (0, 1, 2, …, size of processing area) is randomly changed by object *zl* in *scramble* mode.

The concept and audio result of time scramble resemble asynchronous granular synthesis. Time scramble can be also done with MSP – in the same manner as horizontal blur. To transform horizontal blur into time scramble we would only need to put back the *sah~* object (into the *pfft~* subpatcher) that was removed especially for horizontal blur. So the change of x reading position would not change every single sample (effecting each and every FFT bin) but every new FFT frame.

5.2.6 Slice Fade

Slice fade is a technique that works only with moving processing area (5.1.1). Slice fade should be ideally used only when horizontal reading speed is 0 or near zero. The user should only click on different time positions in the spectrogram so that reading position is jumping.

The basis of slice fade is ordinary crossfade. So the specialty of slice fade is to crossfade between various time slices of a single sample. For instance as we jump from one reading position to another one, the spectrum from initial processing area is copy-pasted into new processing area and crossfaded into the original spectrum that was originally there. Therefore this technique should be used on samples with varied spectrum and not on spectrally static sounds.

Slice fade has a pleasant synesthetic effect and big musical potential for live performance – especially if crossfading is combined with other effects. For a smooth crossfade, both spectral planes should be copied and crossfaded. For sonically more unusual effect, with slightly percussive onset, only amplitude plane can be used.

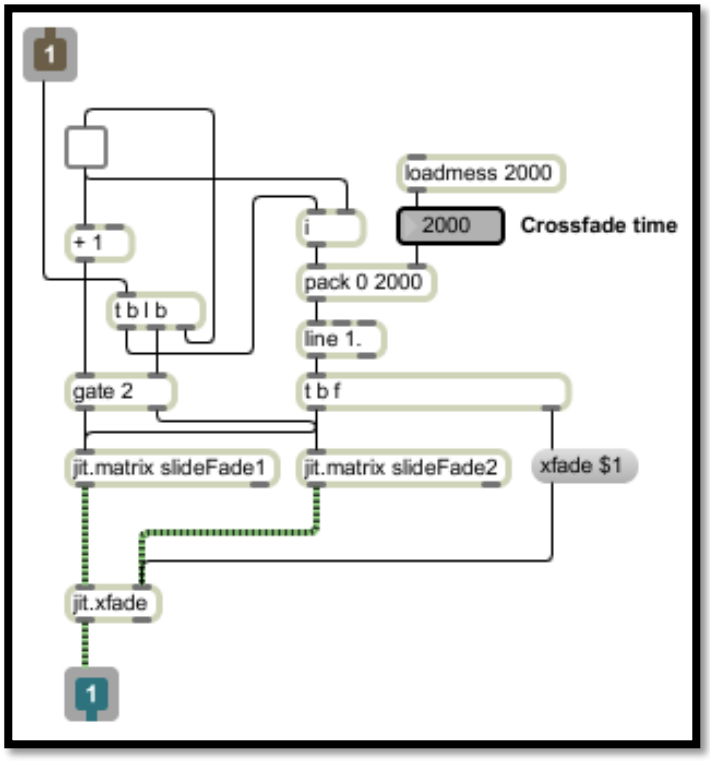

Figure 5.20 Slice fade acting on both spectral planes with adjustable crossfade time (original).

5.2.7 Sound Rider

Performing dynamic processing on a sound with high dynamic range is usually problematic. For instance when applying a compressor on vocals with high dynamic range, the louder parts can be squashed while quitter parts would not be processed at all. Hence sound engineers usually manually perform and record the volume automation before compressing. There are also plug-ins that do such tasks automatically. One of them is Vocal Rider from Waves and that is where the name sound rider originates.

Sound rider is based on moving processing area (5.1.1) and width of the area is the only parameter that technique uses. Spectrum in processing area is normalized and as we read the spectrogram, the volume is constantly adjusted to maximum. Since maximum level depends on the loudest sound fragment inside the processing area, smaller areas result in more responsive volume correction. For instance if very quiet part of the sound is normalized on its own, it becomes loud. On the other hand if we normalize it together with louder sound (wider processing area), the volume would not be increased that radically.

Sound rider uses object *jit.normalize*. For best effect, sub frequencies should be attenuated before applying sound rider since they usually dominate in loudness.

5.2.8 Saturation / Limiter

Primitive version of limiter and at the same time saturator can be implemented by multiplication of amplitude plane with scalar (amplification) and the *maximum* attribute of *jit.clip* object (ceiling). Normalization of amplitude spectrum before saturation is optional although it is very handy to determine the point of saturation. If we normal amplitude spectrum with *jit.normalize* and use its *amplify* attribute, we know exactly the threshold point, where limiting/saturation takes place. In example below (fig. 5.21), the *amplify* attribute is set to 1024. As the ceiling is reduced below that value, the limiter/saturator starts working. It also works in opposite direction -fixed ceiling and changing amplification.

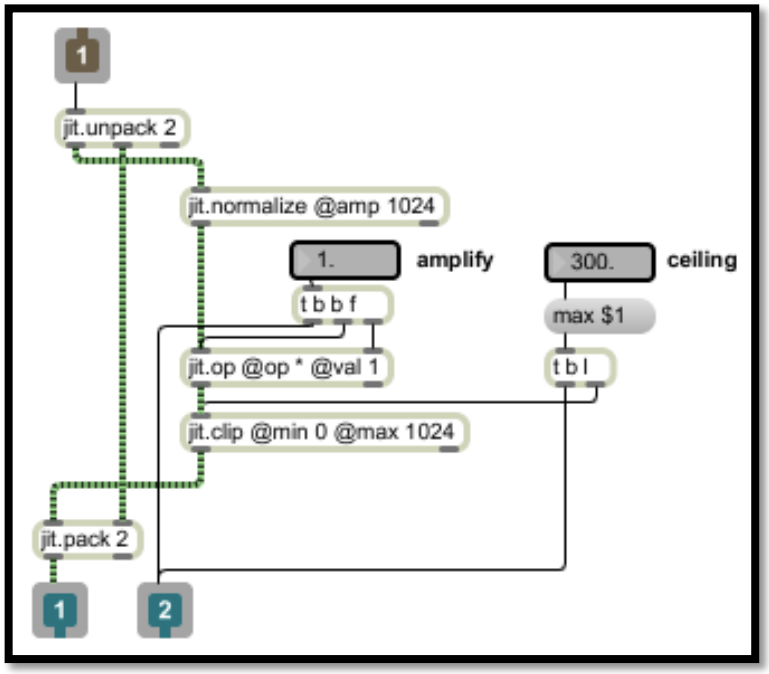

Figure 5.21 Saturator / limiter. When amplify parameter is set to 1, ceiling below 1024 causes saturation (original).

5.2.9 Denoiser

Implementation of denoiser consists of a single Jitter object *jit.op* at attribute >*p,* that is controlled by floating number. *jit.op* at attribute >*p* passes through all the (amplitude) values that are larger than specified value. All values that fail to meet the criteria are zeroed.

Real-time denoiser implemented in PV has great musical value beside its obvious purpose. When reading speed is set to 0 and denoising threshold extremely high (so the sound practically disappears) we can slowly reveal the sound partial by partial with lowering the threshold.

5.2.10 Compression

Compressor can be done by creating a longitudinal section of amplitude plane into “bottom” and “top” matrix, and scaling only the upper one. The “height” of longitudinal section represents a threshold while scaling the “top” matrix represents the compression ratio. Threshold value should be subtracted from the “top” matrix before scaling it and adding it back to the “bottom” matrix (fig. 5.22). For example if threshold is set to 30 and compression ratio to 0.5, the amplitude cell with value 36 (“top” matrix) would first become 6 (subtraction), then it would be scaled to 3 and added back to value 30 from the “lower” matrix resulting in 33.

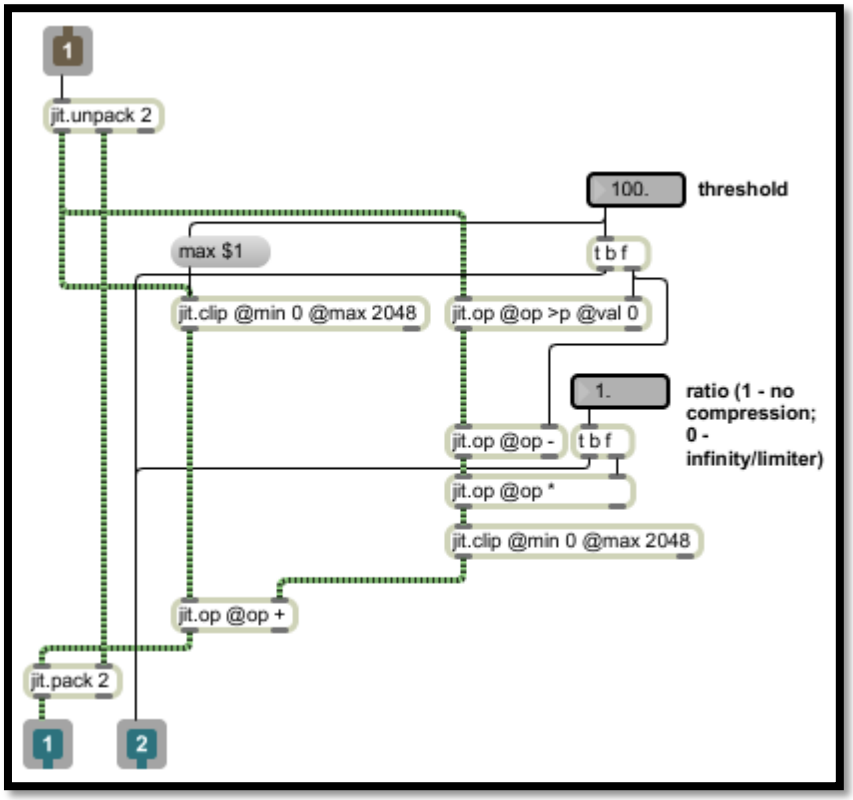

Figure 5.22 Compressor - splitting the amplitude spectrum into “bottom” and “top” matrices (original).

5.3 Local Transformations

By local transformations we mean spectral processing that is applied to specific place of the spectrogram or processing ADSSP area. This can be achieved by multiplication of spectrogram by transfer matrices that defines the place and amount of desired graphical effect, as proposed by Sedes, Courribet and Thiebaut (2004). These kind of matrices will be called masks and the process of applying will be called masking. Another way of using transfer matrices is to use them as accurately controllable spatial maps for repositioning spectral data. These matrices will be called spatial maps and the applying process will be called repositioning.

The best thing about using transfer matrixes in Jitter is that we can use any Jitter object to process them since it is probably the most reasonable to store them as *char* (character) data. Storing transfer matrixes as char data is CPU and computer memory friendly and gives us the approximate accuracy of 0.004 (1/255), when used to scale the FFT data that is stored as 32-bit floating point numbers.

5.3.1 Masks

In this chapter we will take a look at creating three types of masks. Filter masks, rectangular masks and arbitrary masks. In general, filter masks can also be rectangular or arbitrary, but due to their certain specifications, a separate chapter will be dedicated to spectral filtering. We will continue working with large FFT windowing since that impose additional problems when

drawing masks. Spectrums with high frequency resolution demand logarithmic scaling when high accuracy is needed in the most important part of the spectrum.

5.3.1.1 Filter Masks

"In an ideally sharp filter, the cutoff frequency is a kind of brick wall: anything outside it is maximally attenuated, dividing the frequency response neatly into a passband and a stopband" (Rhoads, 1996, p. 188). While this is impossible to achieve with time-domain filtering, it is easily achieved with frequency-domain filtering.

From a pure technical perspective, spectral filtering is not really a filtering – it is just a sound resynthesis, where we exclude and multiply (boost or attenuate) specific FFT data. The reduction of sound quality and generally inferior frequency resolution is an unavoidable price for such an "ideal filter". Despite the fact that FFT filters are practically not in use among sound engineers, they offer huge potential in sonic arts. The ability that every amplitude bin of FFT analysis can be separately set without any cutoff transition bands represents an unique tool for performing spectral subtractive synthesis.

The shape of primary filter masks can be arbitrary. We recommend using *multislider* and *jit.fill* objects for that task. When drawing the shape of band-pass filters, we also recommend 1:1 slider to pixel relationship, hence without the need of upsampling. In other words, the number of sliders in *mutislider* object should correspond with the number of rows in produced matrix. The

maximum amount of sliders in *multislider* object is 256, which should be enough for any band-pass filtering needs at the level of bin accuracy (fig. 5.3.1).

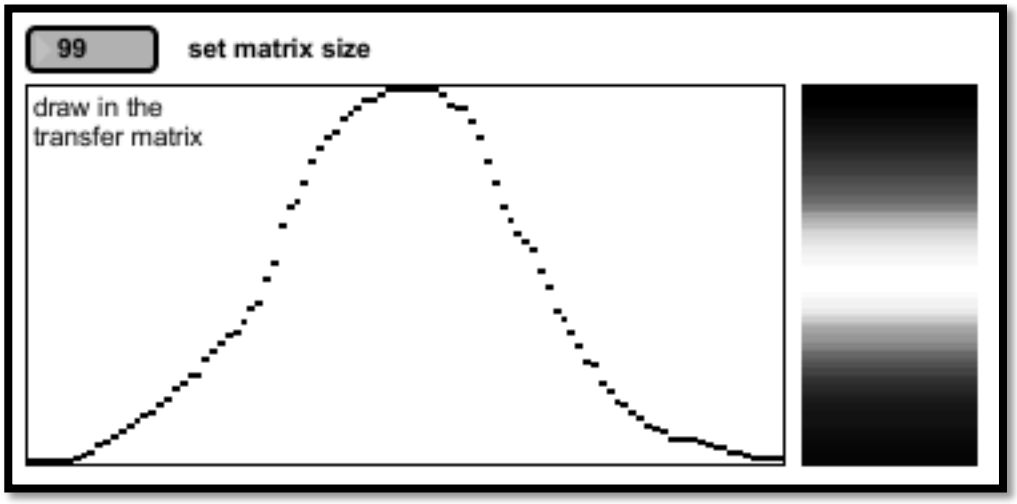

Figure 5.23 Hand drawn mask for band-pass filter using *multislider* and *jit.fill* objects (original).

Instead of hand drawn masks we can also use mathematical expressions to position the sliders of *multislider* object.

The real power of band-pass (BP) filtering when used for sound design or sonic art purposes is when a set of band-pass filters is distributed linearly or exponentially over the spectrum. Another useful feature is the ability to turn on and off even numbered filters that could be called harmonics or partials when using filters with very narrow bands. Having a hand control over each separate filter is of course an extra bonus. All that can be implemented in Max/MSP using *lists* (arrays), *multislider* object and *dstdimstart/dstdimend* attributes of *jit.matrix*.

The idea is that all mathematical calculations concerning BP filters distribution, are stored and handled by lists, and objects that process lists. The information contained in lists are the instructions on how to copy/paste the primary filter mask into a new matrix that has a size of the whole spectrum (2048 rows in our case). Before the information from the "list world" is passed to the "matrix world", it is visualized by *multislider* object that accepts and outputs lists (*the size of a incoming list automatically sets the number of sliders). So we can use the *multislider* as an additional controlling interface (fig. 5.24a – 5.24d).

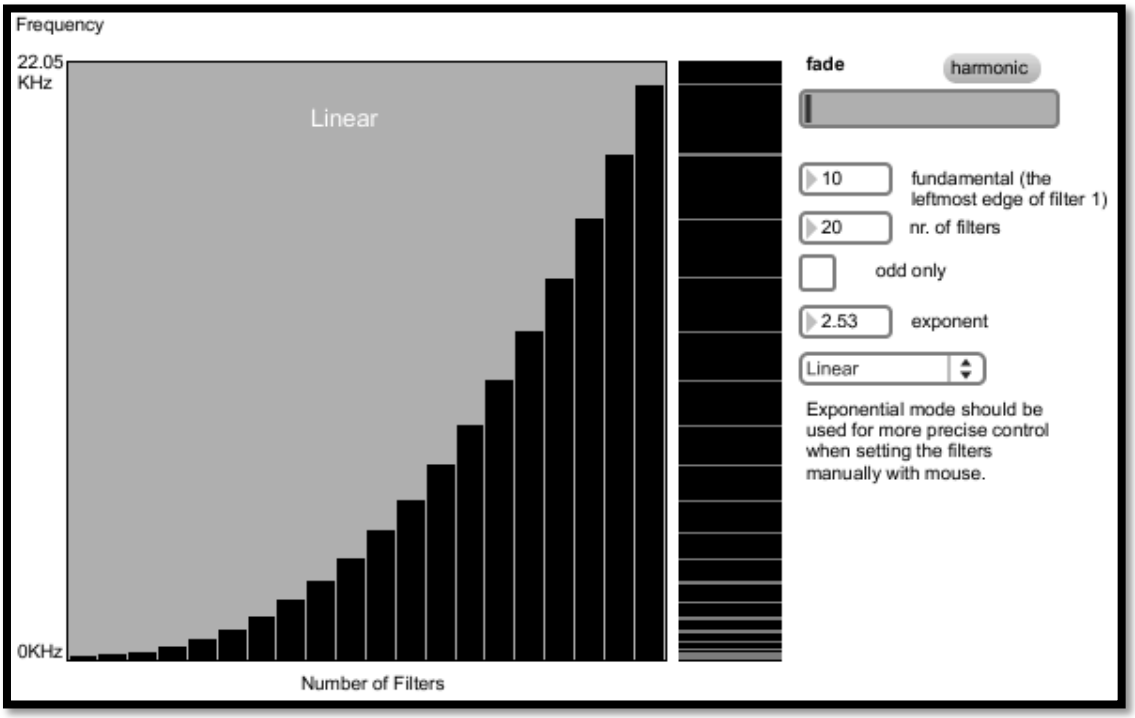

Figure 5.24a Exponential distribution of primary filter mask can be seen in secondary filter mask (at the right hand side of *multislider* object). For this example, rectangular primary mask of width 5 pixels (5 rows) was used. The inconsistency of secondary mask is a consequence of displaying 2048 row matrix in 320 pixels height display (therefore the problem is only visual). The frequency scale is linear (original).

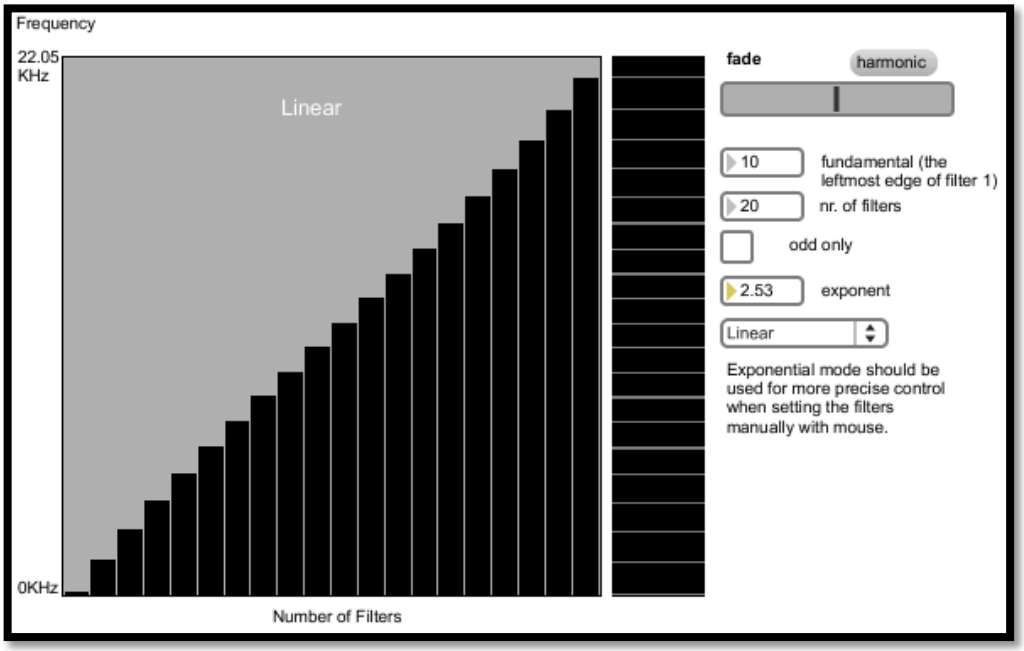

Figure 5.24b Crossfading between the two opposite exponential distributions result in linear distribution when crossfade factor is at 0.5 (half value). For additional notes pleas see the text below figure 5.24a (original).

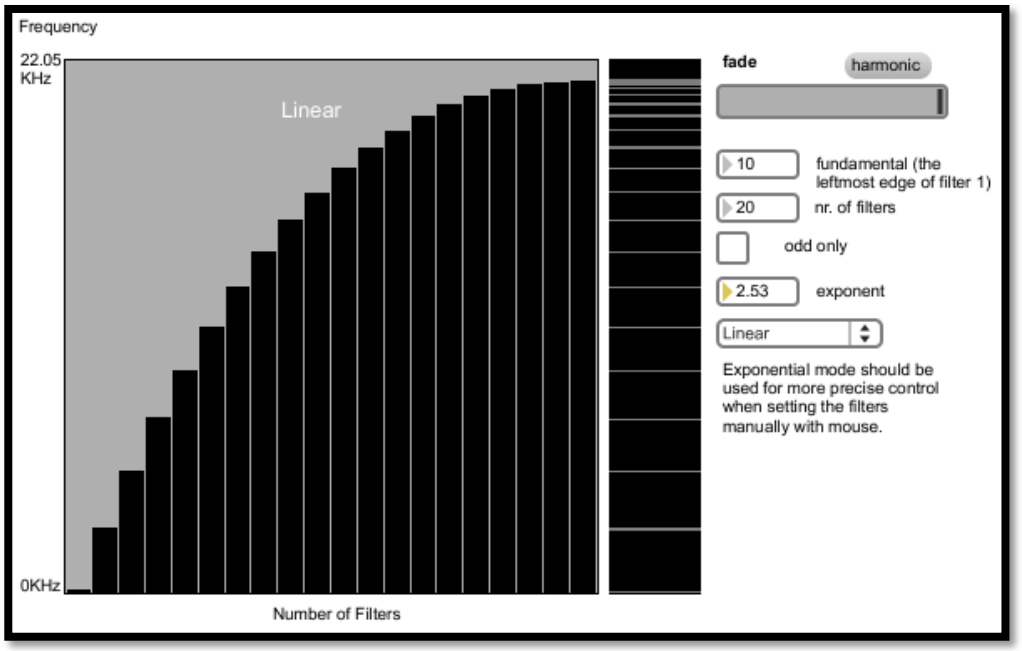

Figure 5.24c The opposite exponential distribution of primary filter mask as a consequence of crossfade factor set to 1. For additional notes pleas see the text below figure 5.24a (original).

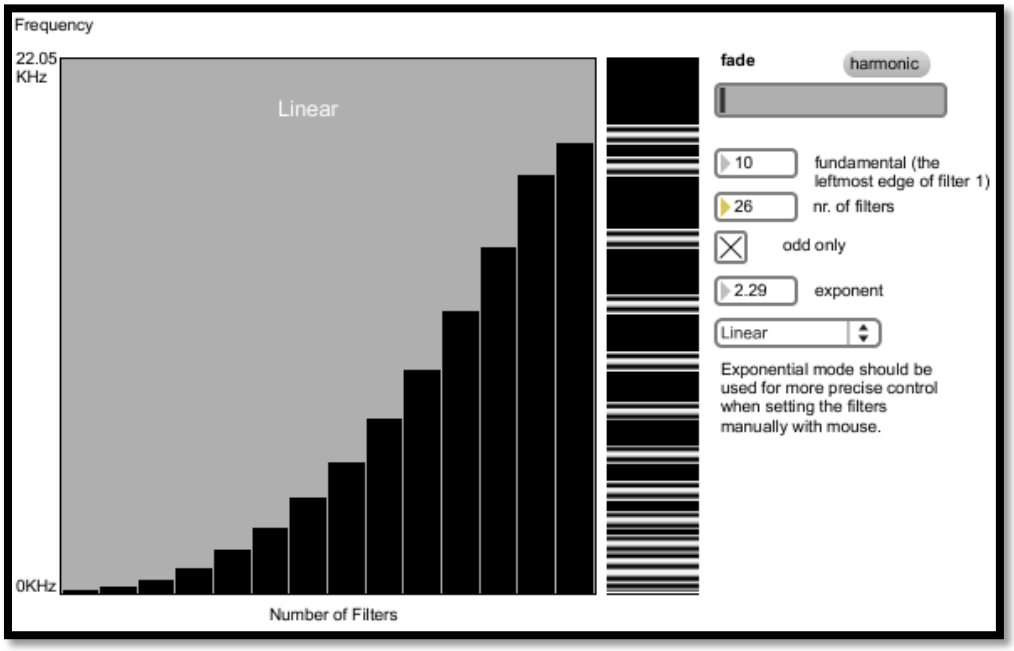

Figure 5.24d Using hand drawn "cosine" function as a primary filter mask. Number of BP filters (primary masks) is set to 26 but only odd numbered ones are in use. Exponent is also different as in previous three examples and the upper most BP filter was manually lowered by using the *multislider* (original).

The secondary filter mask can be additionally scaled by another transfer mask that functions as parametric EQ (fig. 5.25). We implemented parametric EQ also with *multislider* object. Since we are using 128 sliders (max. is 256) to control the content of 2040 row matrix, the matrix generated with list from *multislider* has to be upsampled and interpolated. Parametric EQ can be also used as a low, high or band pass filter.

All transfer matrices used in this section are single columned matrixes. Matrix with only one column or row is an array and that is the reason why all the calculations can be performed with lists and objects that process lists (these are mainly the *zl* object family and the *vexpr* object). The only reason why displayed matrices have a width is because we are stretching a single column matrices with *jit.pwindow* (display object). The distribution and crossfading can be also

implemented with the *poly~* object but that method demand more CPU power and more computer memory.

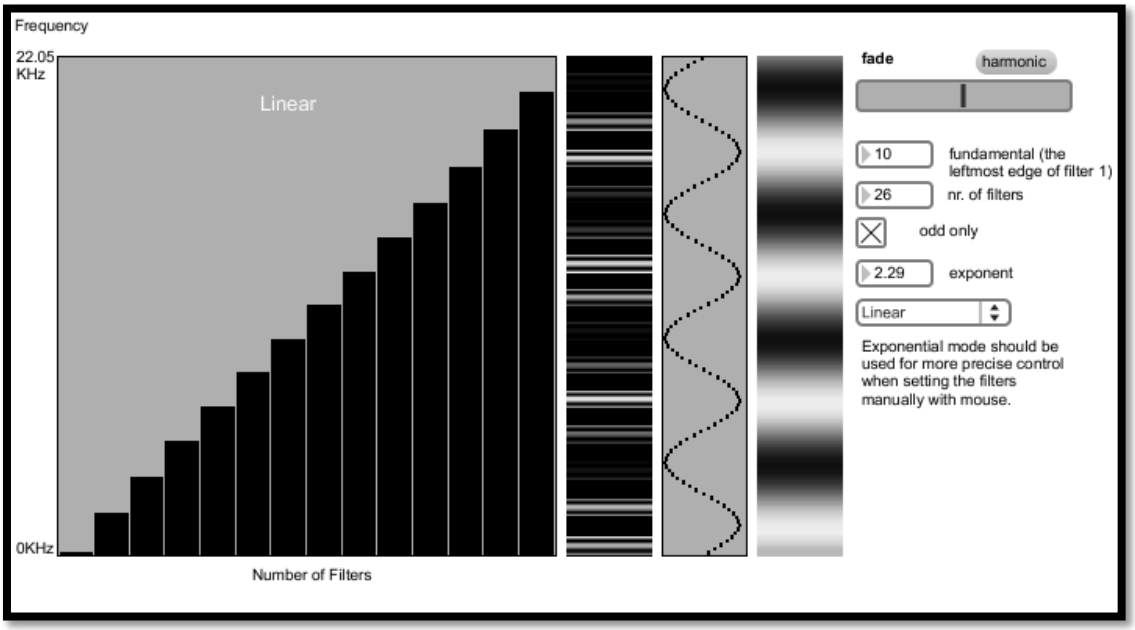

Figure 5.25 Using mathematically calculated sine wave to position the sliders of object *multislider*. That generates an additional transfer matrix that functions as parametric EQ and gives the secondary filter mask a final shape. Instead of sine wave we could use any mathematical function or hand drawn curves (original).

When copying and pasting the primary mask into secondary one using the *dstdimstart* and *dstdimend* matrix attributes, it is important to be aware of the possibility that BP filters (primary masks) can be overwritten by neighboring BP filters if placed too close together. In other words, BP filters that are placed too close can cancel each other out. That is completely opposite as when placing close together BP filters in time domain. The cancellation effect can be avoided by adding the primary matrices into secondary matrix, instead of just overwriting the existing data. For that approach an additional empty matrix at the size of secondary matrix would be needed.

Each primary matrix would have to be first placed into an empty matrix (using the *dstdimstart* and *dstdimend* attributes) and then *added* to a final secondary matrix. Before any new instance of primary matrix would be copy/pasted (at different position) into the "empty" matrix, the "empty" matrix should be literally emptied by clearing its content.

The final step or the practical use of filter masks is a matrix multiplication of secondary filter mask and amplitude spectrum. The secondary filter matrix should be stretched to the width of ADSSP area or the whole spectrogram if preferred for the sake of visual effect. Filter modulations can be achieved by real time modulation of parameters that define the secondary matrix (all parameters basically). If secondary filter matrix is horizontally stretched across the whole spectrogram, we could also use the blur/water effect (chapter 5.2.1) on secondary filter mask to graphically modulate certain parameters (straight parallel lines would become stochastically curvy). In case we would like to keep the parallel relationship and change straight lines into accurately controlled curves, we could use the technique presented in sub-sub-sub-section Spectral delay (5.3.2.2.2). The only difference it would be that we would apply the method in vertical direction instead of horizontal. The result could be further rotated using the object *jit.rota*.

"If a high-Q filter is excited by as signal near its center frequency, the filter *rings* at the resonant frequency, that is, it goes into oscillation, for some time after the signal has passed" (Rhoads, 1996, p. 189). This is of course impossible to achieve with spectral filtering because it is not based on feedback or feedforward delay lines. Although the effect of self-oscillation can be simulated by adding the spectral smear effect presented in chapter 5.2 to the spectral filter. Spectral smear introduces a very short delay effect with adjustable feedback.

The possibilities of spectral filtering in Jitter are enormous. There are many objects appropriate for creative filtering. When placing for instance a black picture with single white horizontal stripe in object *jit.mxform2D* or *jit.rota*, one can quickly and intuitively create useful and interesting transfer matrixes - especially when using mentioned objects in *wrap boundmode*. Although for accurate, flexible and especially methodological approach to filtering one has to create its own graphical effect.

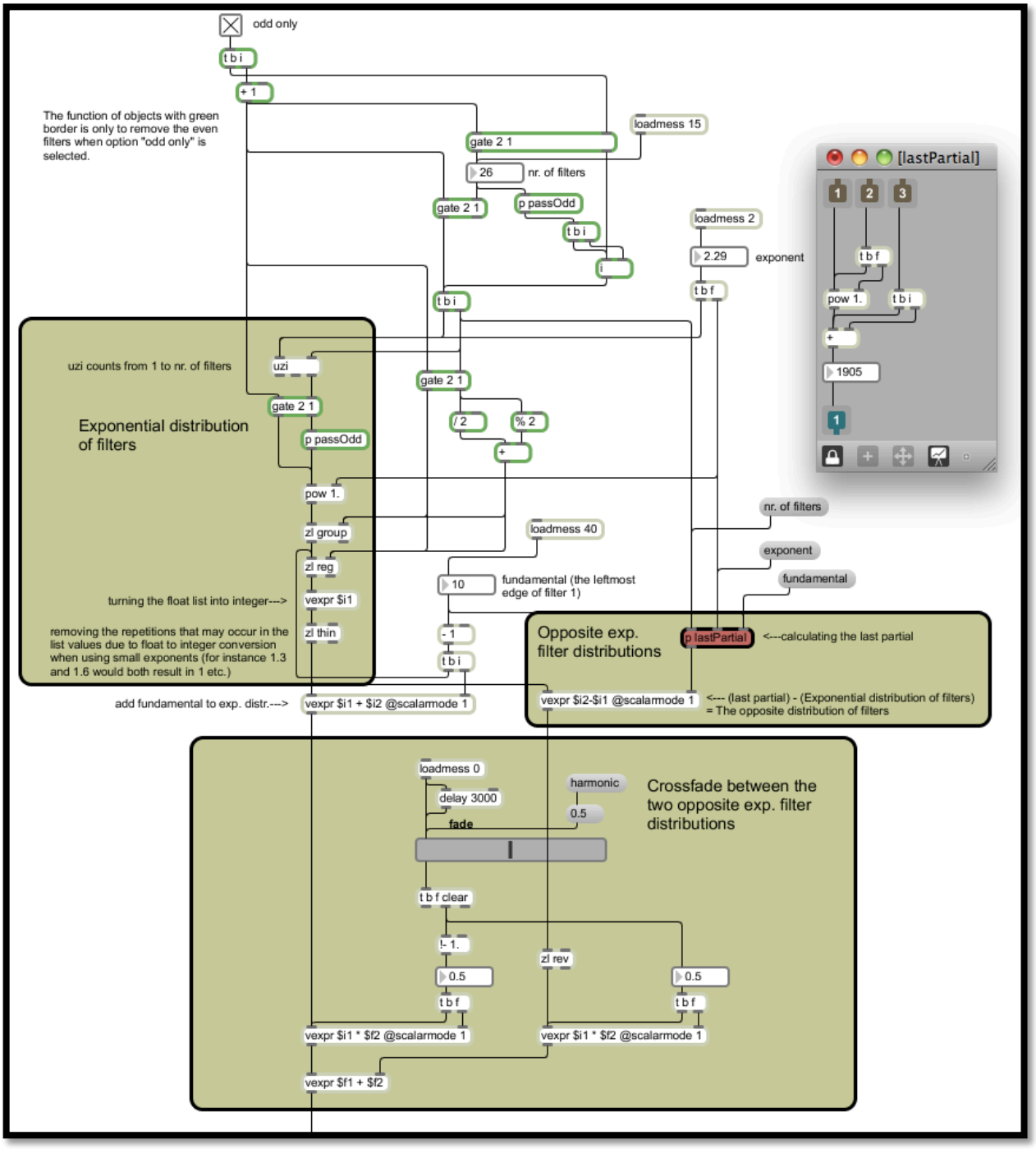

Figure 5.26 Implementation of primary filter mask distribution. The patch chord at the very bottom is attached to the *multislider* that can be seen on figures 5.24 (original).

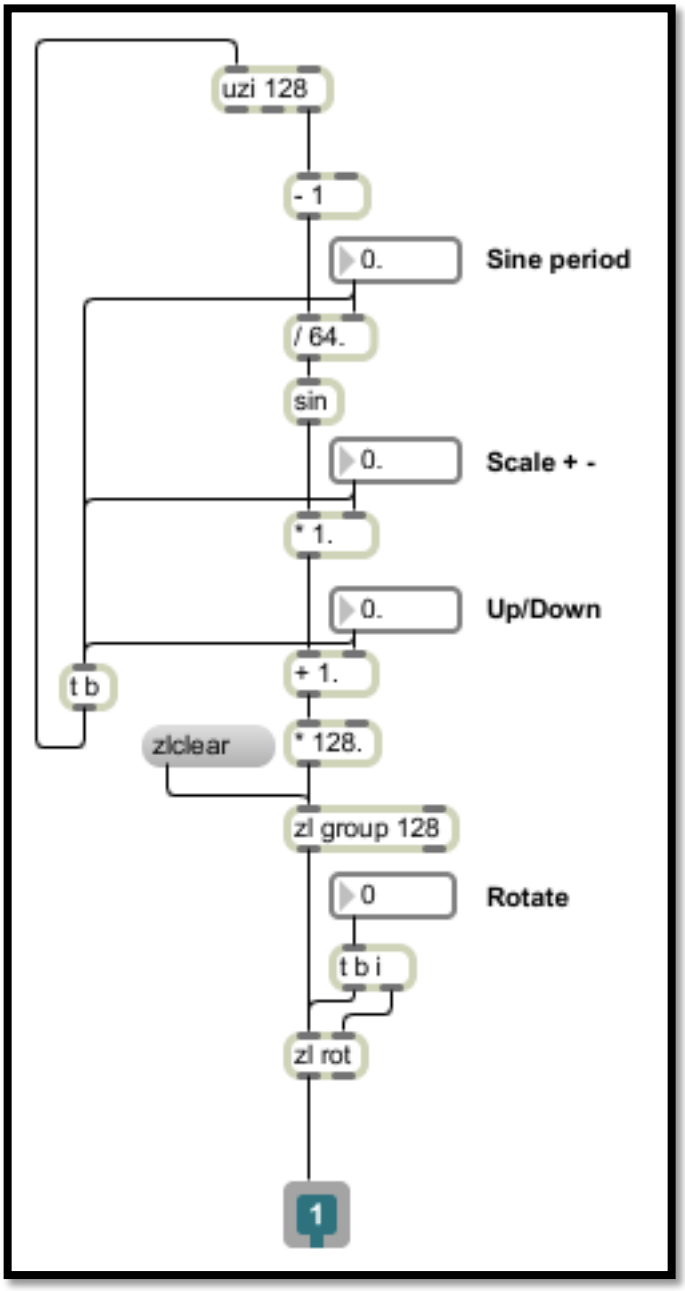

Figure 5.27 Using *uzi* object to write values of sine function into list of size 128. Two different methods of processing data were used: using *zl* objects to process data as lists or using *, / and + objects on each integer output of *uzi*. There is also an option of using *vexpr* object to perform mathematical operations on lists (original).

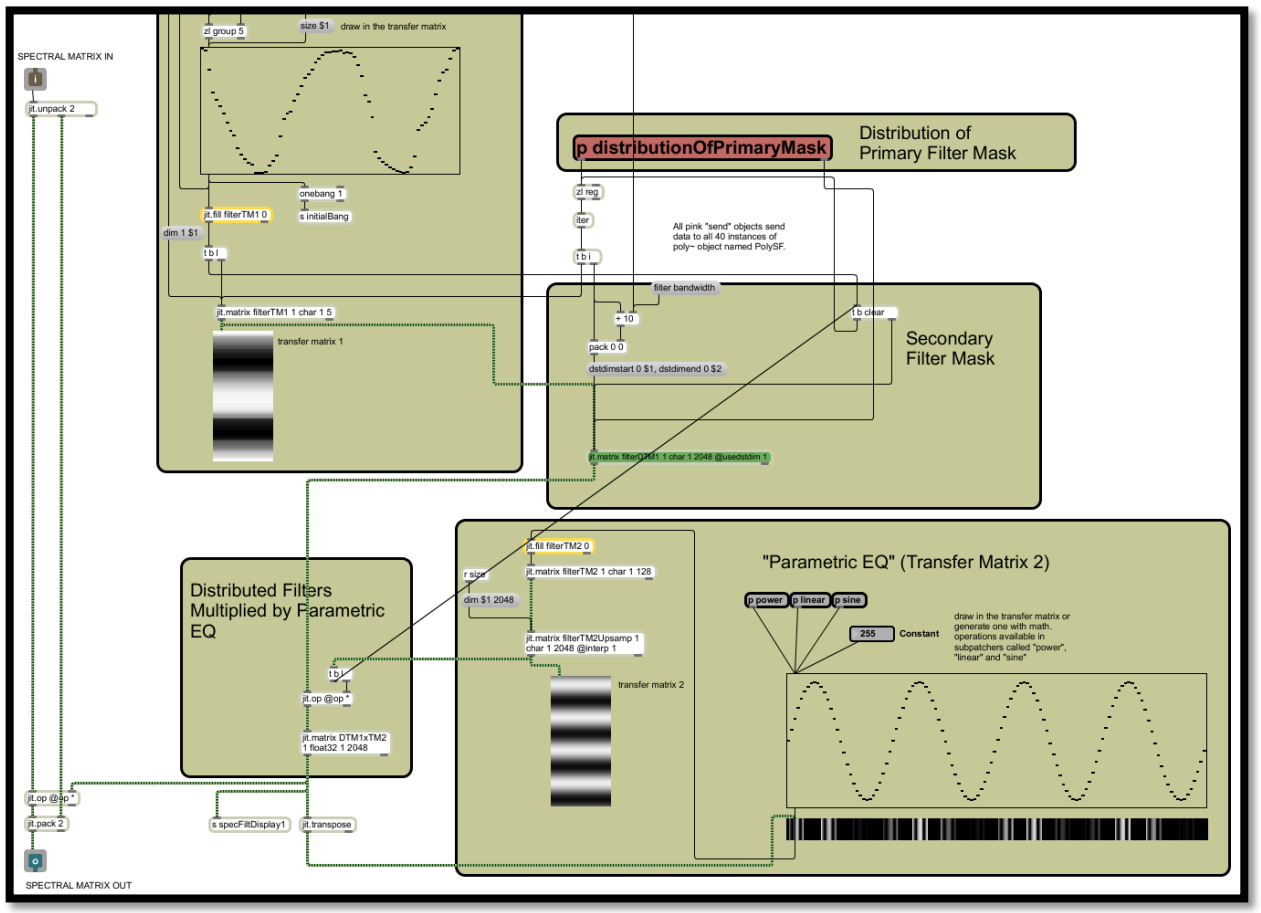

Figure 5.28 The main patch. The content of subpatchers *sine* and *distributionOfPrimaryMask* can be seen on figures 5.27 and 5.26 (original).

5.3.1.2 Applying a Mask to a Spectrum

Applying a filter mask to a spectrum is very straightforward. It is all about matrix multiplication of the filter mask with the amplitude spectrum. On the other hand applying a mask to localize a specific spectral effect demands different approach.

The mask area has to be cut out from the spectrum by the process of *multiplication*, processed separately and *added* back. If we would be using only rectangular matrices, we could perform the task simply by cutting-processing-pasting. But since that method is limited to rectangular shapes only (due to rectangular disposition of matrix), we need to cut with the process of multiplication and paste with the process of addition (fig. 5.29).

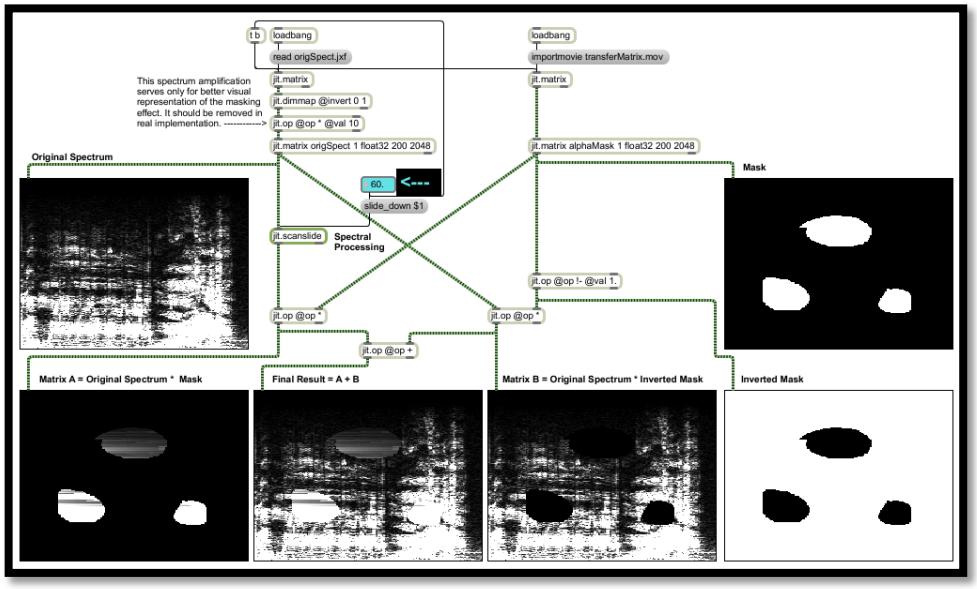

Figure 5.29 Multiplication of the mask and its inverse with the spectrum generates matrices A and B. After processing matrix A, matrices A and B are added back together. For this example, an extreme spectral smear was used caused by object *jit.scanslide* (original).

In case we would like to gradually introduce the spectral effect (temporally) we can apply horizontal Gaussian blur to the mask. The same holds for blurring the frequency borders in vertical directions (fig. 5.30).

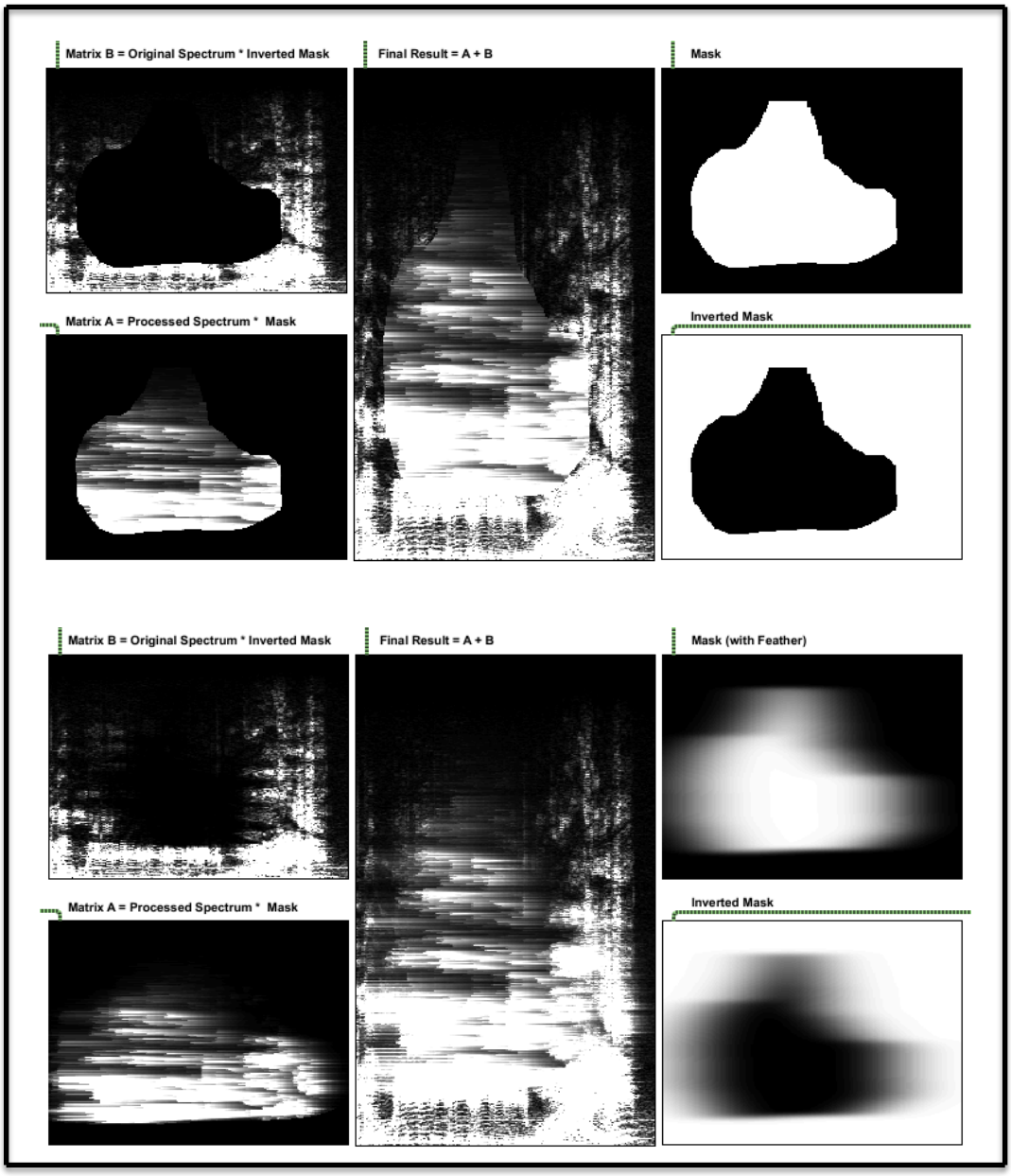

Figure 5.30 The masking process with (lower picture) and without (upper picture) Gaussian blur. For this example an extreme spectral smear was used caused by object *jit.scanslide*. Also a significant amount of Gaussian blur in x and y directions was used. The reason why vertical blur is hardly noticeable is because we are watching a matrix with 2048 rows in much smaller display (original).

Since Gaussian blur in Max/MSP exists only in openGL, we have to move from CPU to GPU and back again to CPU. For a faster transfer we can use *uyvy2rgba* and *rgba2uyvy* shaders. UYVY colormode exist only for the sake of faster transfer between CPU and GPU. But since making the mask feather with blurring is not something that would demand maximum performance, using the mentioned shaders is not necessary.

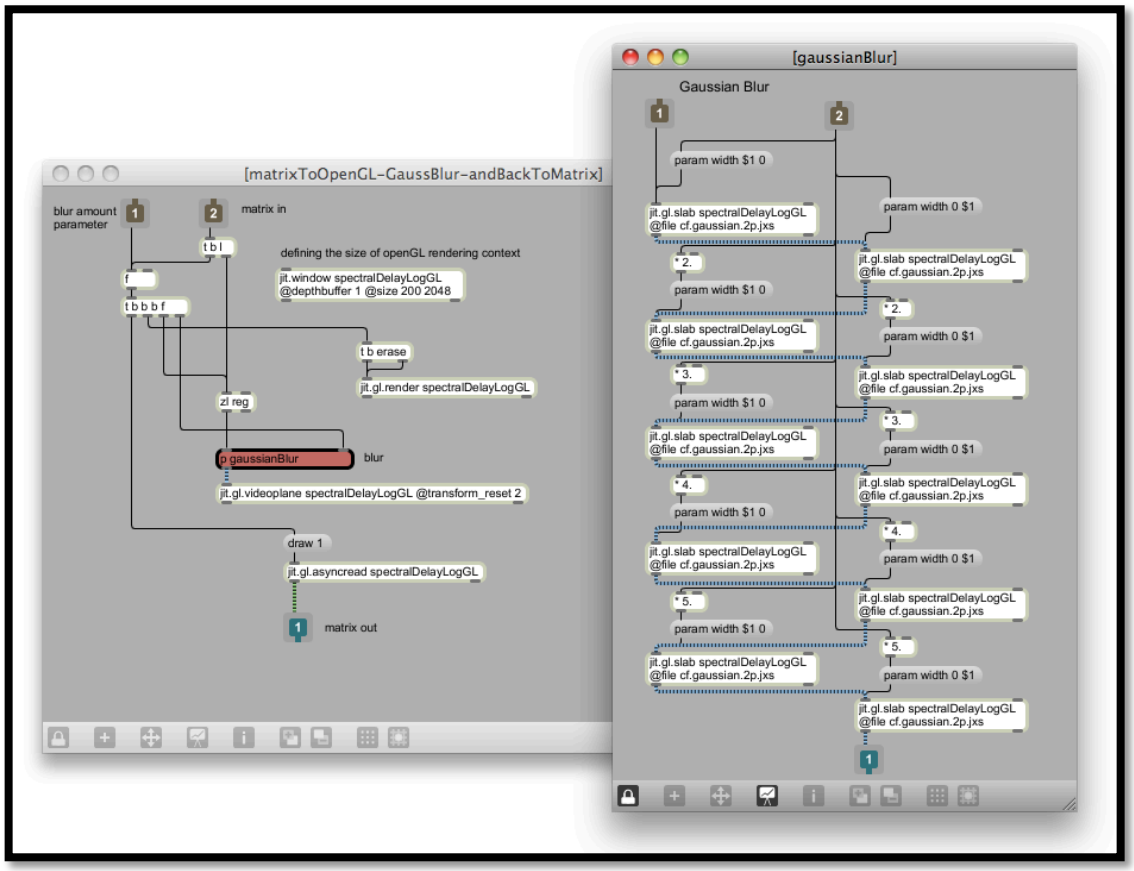

Figure 5.31 CPU-GPU-CPU transfer and the Gaussian blur subpatcher. Left column in subpatcher performs horizontal blur and right column performs vertical blur (original).

The good side of using openGL (GPU) for processing masks or basically any other data based transfer matrices, is to unburden the CPU. When doing the CPU-GPU-CPU transfers, it is necessary to define the size of openGL rendering context, that should be the same size as an incoming matrix. In our case the rendering context needs to have dimensions 200 x 2048. If we use smaller rendering contexts, the out-coming matrix suffers a loss of resolution. Now the problem that appears at this stage is, that the dimensions of openGL rendering context can be defined either with *jit.window, jit.pwindow* or *jit.*matrix (any of these objects needs to have the same name as defined in *jit,gl.render*). Ideally for spectral processing that does not end on GPU, is to define the context with *jit.matrix,* because we do not need to display the openGL content. But as soon as *jit.matrix* and *jit.gl.render* share the same name, it is impossible to define the matrix size anymore (even by using the *dim* attribute). That might be a software bug, since the computer used during the research *always* crashed (MacBook Pro with 2.4 Ghz Dual Core Intel processor and NVIDIA GeForce 8600M GT graphic card) when trying to define the size of the matrix in question. The problem could be of course also in the computer system or hardware. Various experiments showed, that the only reliable way to define the size of rendering context is to use *jit.window*. That means that whenever we use CPU-GPU-CPU transitions, we need to display the rendering context through a window. Luckily, the openGL rendering is of course processed on GPU.

When applying a feather mask, the area of the feather does not gradually decrease or increase the amount of chosen spectral effect but rather presents a dry/wet relationship. In order to literally change the amount of the effect, we need to control the responsible parameter of chosen effect. That can be achieved by parallel reading of additional transfer matrix, that stores the modulation data.

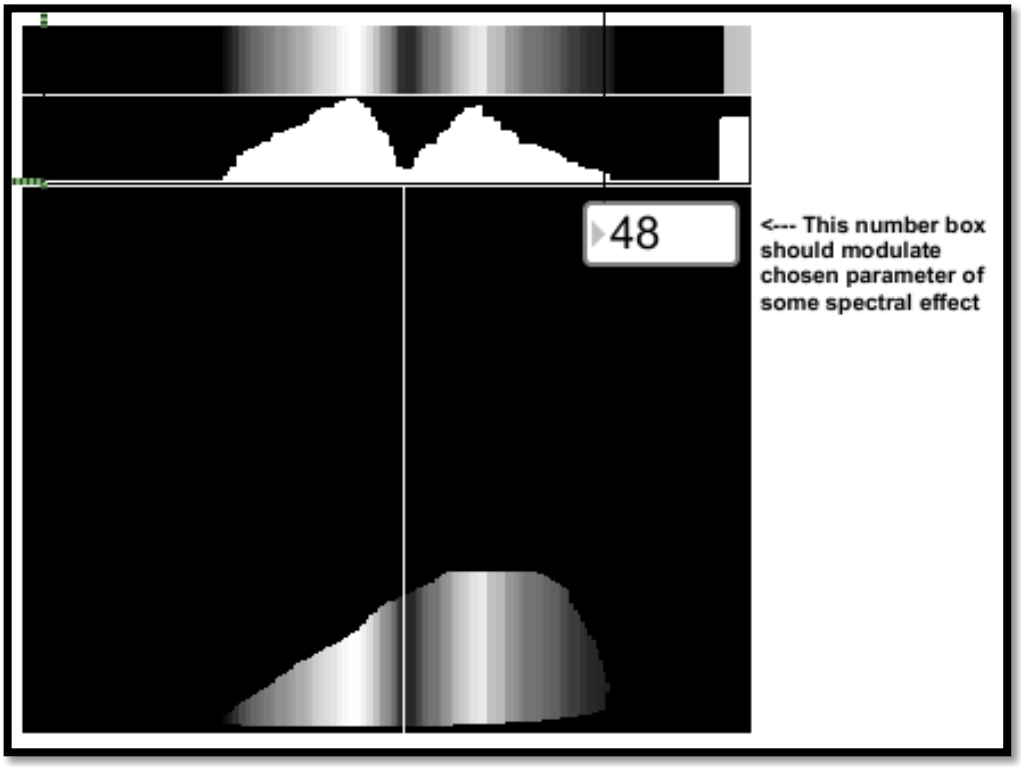

Figure 5.32 The horizontal reading position performs a parallel reading on a modulation transfer matrix. In real patch, the modulation matrix should not be added to the mask, as presented in lower window (original).

In this chapter we were using only premade masks. Generating masks in Max/MSP and Jitter is covered in the following sections.

5.3.1.3 Rectangular Masks

The first problem when drawing rectangular masks appears already at the stage of drawing a rectangle. If we want to select a rectangular area on the spectrum, we need a frame of a rectangle, that grows as we click and drag the mouse across the spectrogram. If we use the *jit.lcd*

object and command *framerect $1 $2 $3 $4,* we need to constantly erase the previous frame in order to see only the last one. As we click on *jit.pwindow,* the coordinates for top-left ($1 and $2) of a rectangle has to be stored and used along the drawing process. As we drag the mouse, the bottom-right coordinates ($3 and $4) are constantly changing. As we release the mouse button, the last bottom-right coordinates have to be packed together with the top-left ones and sent further (fig. 5.30 and 5.31).

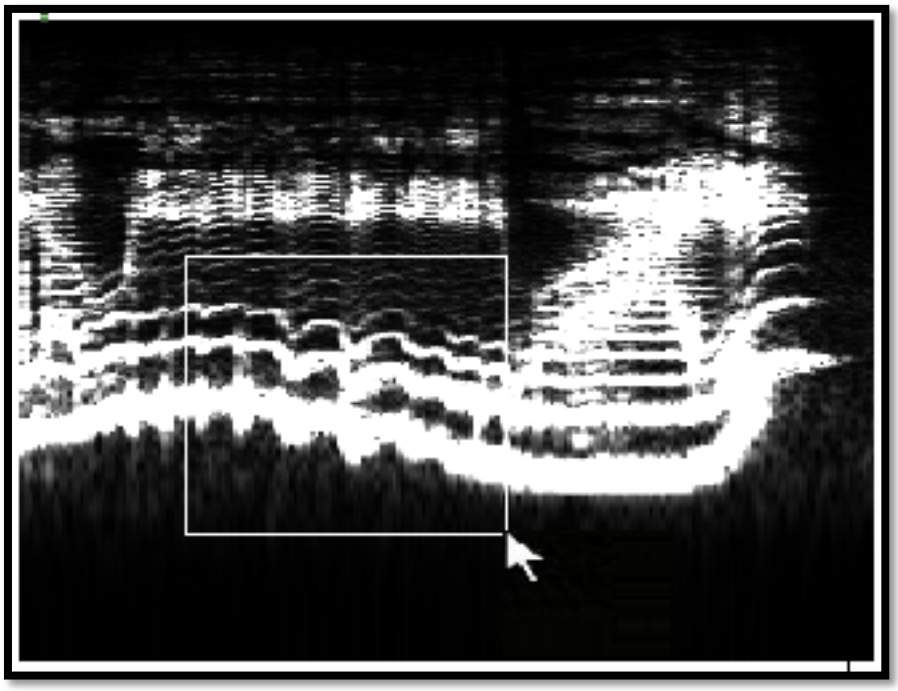

Figure 5.33 Drawing the flexible rectangle frame on a logarithmic spectrogram by dragging a mouse. (original).

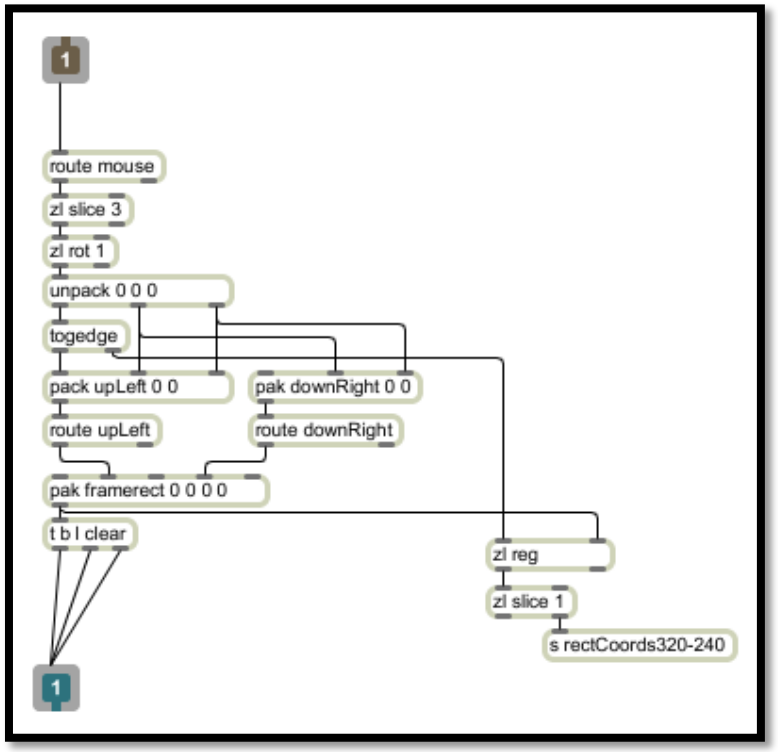

Figure 5.34 The implementation of flexible rectangle frame. The inlet of the *subpatcher* receives mouse coordinates from *jit.pwindow* and the outlet is connected to *jit.lcd.* The coordinates of final rectangle frame are sent further with the *send* (*s*) object (original).

Since we are drawing a rectangle on logarithmically scaled spectrogram, displayed in 320 x 240 window, we need to exponentially scale the rectangle coordinates to create a mask, that would cover the marked area in linear, 200 x 2048 spectral matrix. Exp. scaling was already presented in chapter 4.4 (see figures 4.5 and 4.6), so we will show only the way to generate a mask from already exponentially scaled coordinates (fig. 5.35). Rectangular masks can be also generated with the method presented in sub-sub-section 5.3.1.4 (arbitrary masks).

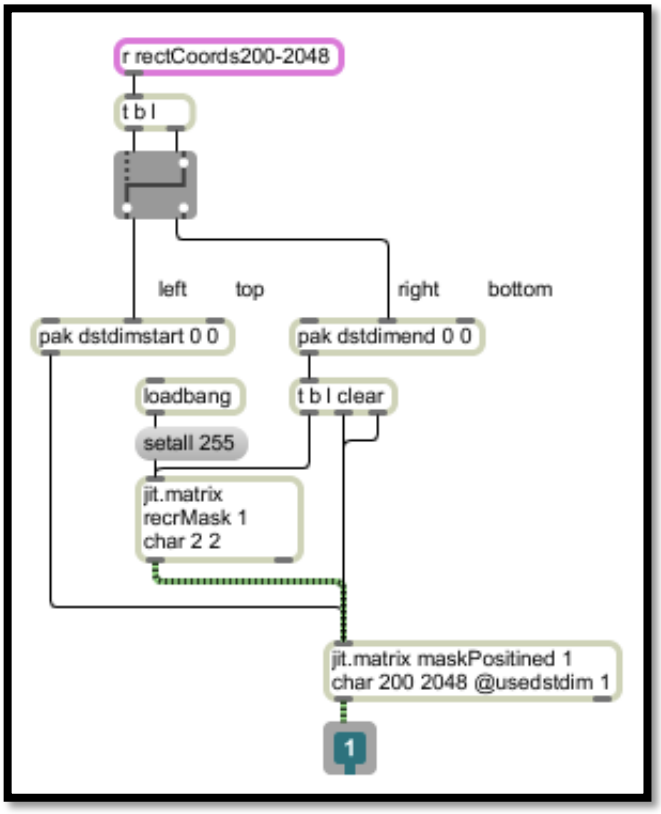

Figure 5.35 The exp. scaled coordinates comes into the *subpatcher* in the form of two successive lists (one list for each opposite angle). One list is passed to the empty ("black" or filled with zeros) 200 x 2048 matrix together with *dstdimstart* message, and the other one with the *dstdimend* message. With these two messages, we position a "white" 2 x 2 matrix into the "black" 200 x 2048 matrix. Positioning at the same time acts as a mask stretching since the original mask is initially stored as a 2 x 2 matrix (original).

Next figure (5.33) shows the positioning of a drawn rectangle in a 200 x 2048 matrix that represents the mask.

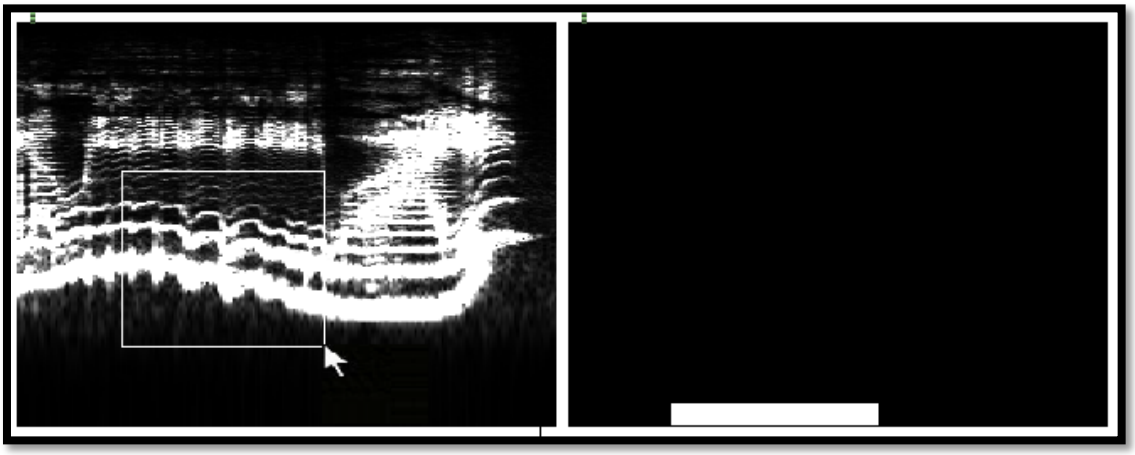

Figure 5.36 At the right picture we see a ("linear") mask as a 200 x 2048 matrix that is a derived from selected rectangular area in logarithmic spectrogram. The selected frequency area was approximately between 1250Hz and 50Hz (original).

If we want to generate a mask from multiple rectangles, we can implement a feedback loop, using the *plus* operator of object *jit.op* as presented on figure 5.38. An additional option is to generate various masks, where each mask is stored as a separate plane of a matrix. If the masks consist of char data, we can use RGB planes to create three different masks for three different effects (figure 5.37 and 5.38

If we want to display generated masks also on the spectrogram, as presented on figure 5.37 (left picture), we need to either generate another set of masks out of non scaled data, or scale back the actual masks (right picture of fig. 5.37) using the inverse mapping.

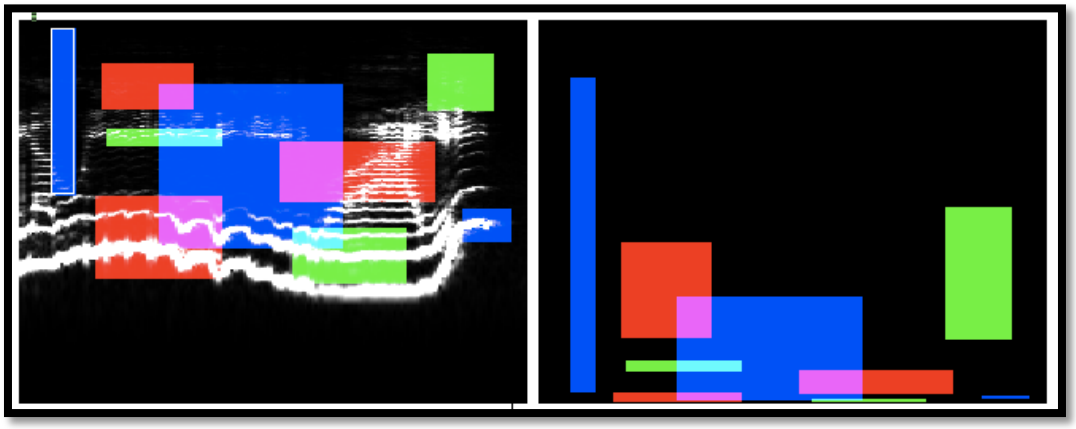

Figure 5.37 Using three planes of four plane ARGB matrix for storage of three different masks, that should be used on three different spectral effects. The right picture presents the actual size of masks in 200 x 2048 matrix displayed in 320 x 240 window. Each plane on its own (R, G and B) is a graysclae matrix, consisting of values 0 (black) and white (255) (original).

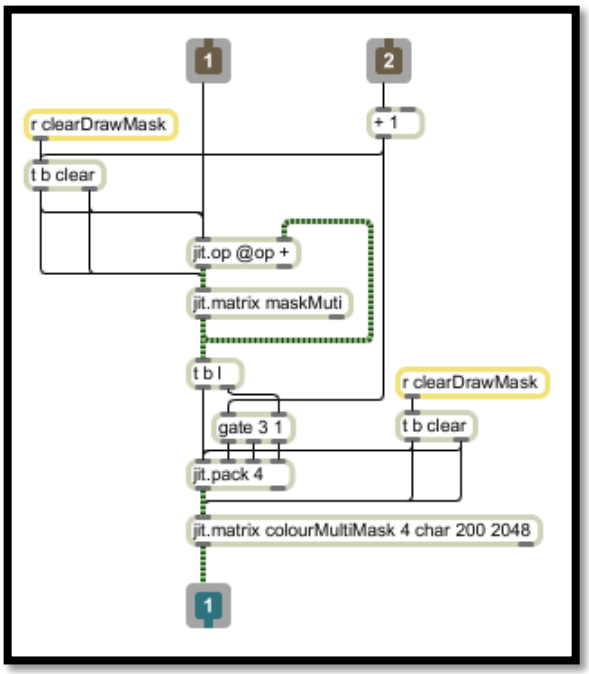

Figure 5.38 To create a multiple-rectangle single plane mask, the outlet of patch presented on fig. 5.35 is passed to inlet 1. Multiple rectangles are added to matrix named “maskMulti” through a feedback loop using *jit.op*. Inlet 2 accepts numbers from 0 to 2 that selects the target plane of ARGB matrix “colourMultiMask”. 0 selects the red plane, 1 selects the green plane and 2 selects the blue plane (original).

5.3.1.4 Arbitrary Masks

The object that is capable of identifying the geometry on the basis of arbitrary vertices is *jit.gl.sketch.* Using that object we can generate closing geometries from three or more points. Since any scribble consists of more than three points, *jit.gl.sketch* gives us a freedom to generate arbitrary masks (fig 5.39).

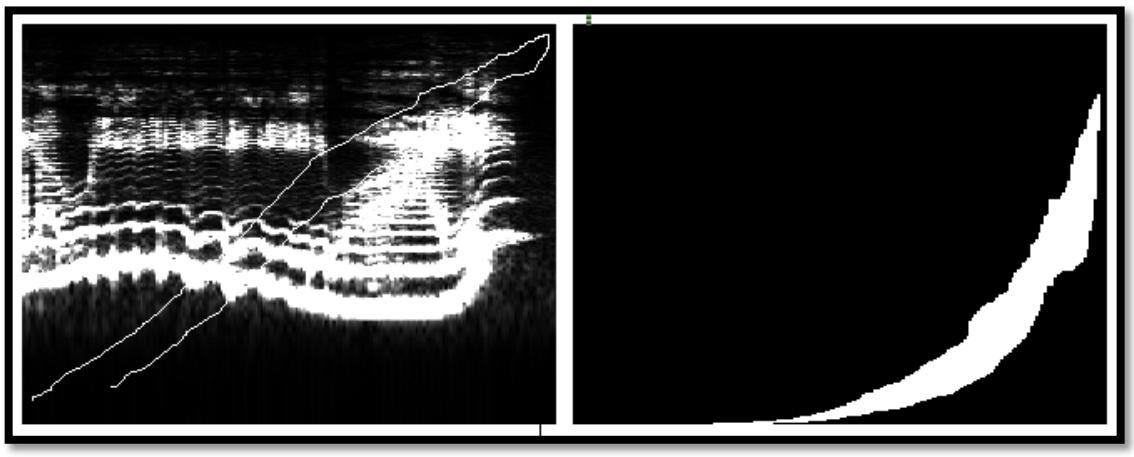

Figure 5.39 Making a mask from hand drawn open curve. On the left hand side we see curve drawn on logarithmic spectrogram and on the right hand side is a closed geometry (mask) as a 200 x 2048 matrix (original).

jit.gl.sketch is an openGL object so we have to again define the openGL rendering context with *jit.window*. If we define the rendering context with *jit.matrix* and leave the default, size the resulting masks are practically useless since the vertical resolution is approximately 10 times smaller.

Drawing with mouse on the display using the *jit.lcd* object is covered in *jit.lcd* help file and will be therefore not presented here. We will rather take a look at an example of generating a closing geometry with *jit.gl.sketch* (fig. 5.40).

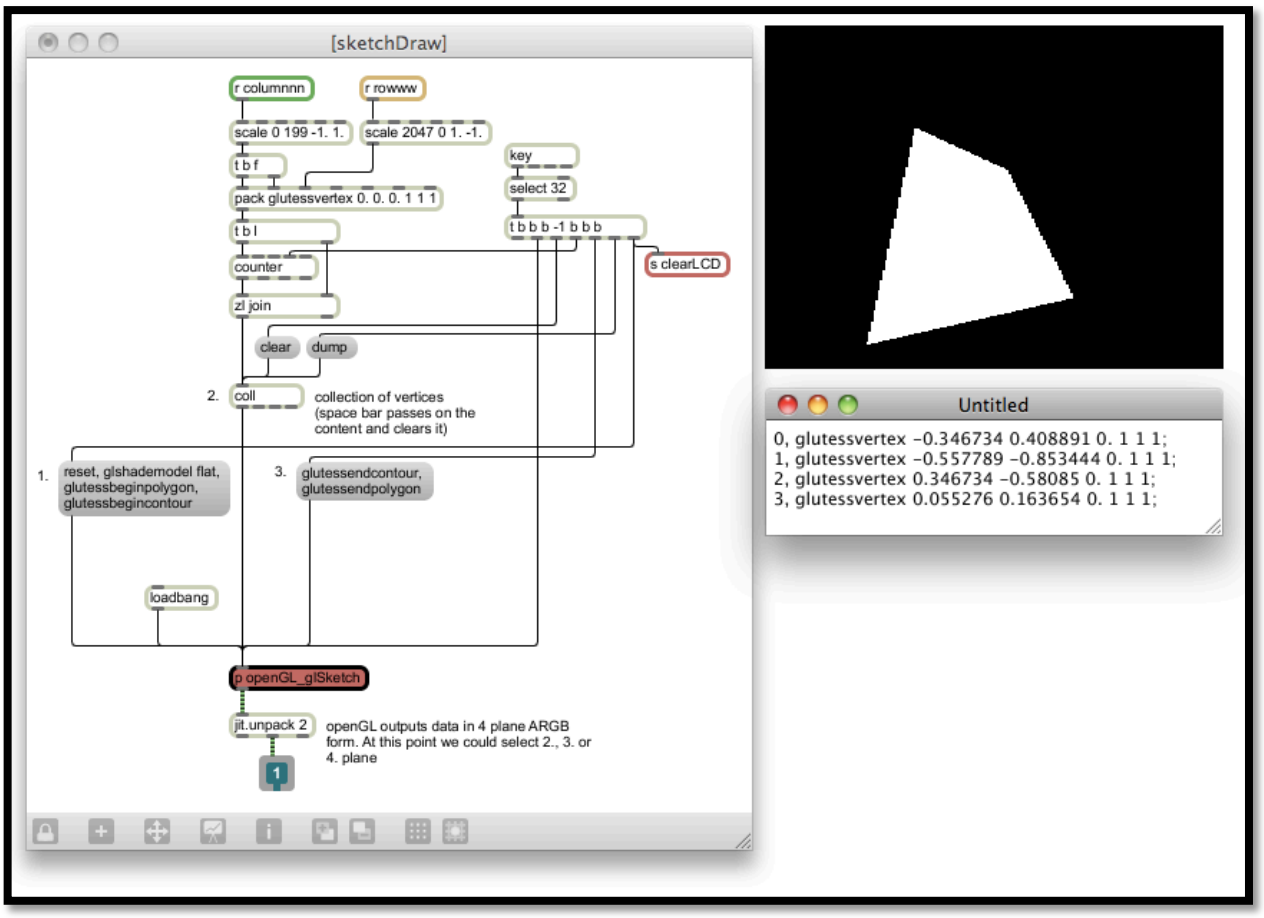

Figure 5.40 Generating a closed geometry from four points or vertices using *jit.gl.sketch* . *jit.gl.sketch* object is hidden in *subpatcher* "openGL_glSketch" and is the first object in openGL chain. In other words, all the messages from *patch* "sketchDraw" are received by *jit.gl.sketch* (original).

jit.gl.sketch demands many commands or messages to create a geometry. Numbers 1 to 3 in figure 5.40 present the order of events that are passed to *jit.gl.sketch* every single time the geometry is created. Set of messages under successive number 1 resets the object, defines the shade model, beginning of the polygon and beginning of the contour. The next set of messages (number 2) can be seen on the right hand side of fig. 5.40 under the display. That is the content of the *coll* object that is filled with vertex coordinates and colours (for the example in question only four vertices were used). There are two important things to notice when looking at the coordinates. Firstly we see that third coordinate (z) is always the same (0). Since we need a 2D mask, there is no need for third dimension. Secondly, the x and y coordinates are scaled between -1. and 1. This can be imagined as matching the center of the spectrogram (as 200 x 2048 matrix) with the center of the openGL 3D coordinate system. This guarantees us 0 mask offset when the geometry comes back from the openGL land. The third set of messages (number 3) defines the end of contour and polygon. The already exponentially scaled coordinates comes into patch "sketchDraw" via receive objects. The openGL part is hidden in *subpatcher* "openGL_glSketch" and object *jit.gl.sketch* is the first in openGL processing chain.

Probably the most elegant way of getting the openGL data back into matrix form (GPU to CPU transition) when using *jit.gl.sketch,* is by setting the name of the *capture* attribute of *jit.gl.sketch* to the same name as is the *name* attribute of *git.gl.texture.* That way we can capture anything that *jit.gl.sketch* draws as a texture. That method works perfectly when openGL rendering context is defined with *jit.matrix.* But then again we have the problems when defining the size of the rendering context (see sub-sub-section 5.3.1.2). This issue might be connected with computer used or is a software bug. The answer unfortunately could not be found during the time scope of

this research. Hence we are again using the same method as presented in chapter 5.3.1.2 based on object *jit.gl.asyncread* (fig. 5.31).

Multiple RGB arbitrary masks can be generated in the same way as in chapter 5.3.1.3.

5.3.2 Repositions

Many spectral effects in terms of ADSSP and PV can be considered as controlled repositions of spectral data. This chapter will be dealing with the creation of spatial maps for object *jit.repos* through user interface. First we will take a look at pros and cons of two Max objects that provide user interface for data manipulation. We will continue to work with relatively large FFT sizes (4096 samples long windows) as this imposes additional problems when creating spatial maps.

5.3.2.1 Creating Spatial Maps with Multislider Object

Filling the content of a matrix with *multislider* object was already covered in chapter 5.3.1.1 and 5.3.1.2. In the same way as we were creating transfer matrices that served as filters we can create spatial maps for *jit.repos*. The only difference is that created (transfer) matrix now serves as a set of instructions for *jit.repos* on how to reposition the matrix data.

Multislider object in general presents the easiest way to fill a matrix. It accepts *lists* that determine the number of its sliders and it outputs a list of all slider values as soon as the value of any single slider is changed. With the help of *jit.fill,* the lists can be easily written into a matrix. Also it has an option for vertical or horizontal layout that enables more intuitive approach to spatial map creation (depending on horizontal or vertical data repositioning). But when creating spatial maps for *jit.repos,* multislider presents one major obstacle.

The basic idea behind using spatial maps for repositions is to initially generate a map, that would leave the content of spectral matrix intact. For instance if all sliders of *multislider* object are set to 0 (or any other preferred initial slider layout), the spectral matrix should stay unchanged. As we interact with *multislider,* the reposition of spectral data occurs. Therefore we first need to generate a map that would cause zero change to our spectral matrix. Due to a relatively large amount of frequency bins, considering much smaller vertical size of user interface, Y spatial maps impose some extra problems. The same would hold for X spatial maps if we would be using larger amount of FFT frames.

When using 4096 samples large STFT windows, we need a 1 column and 2048 rows big Y spatial map, filled with values that go from 0 to 2047 (when using *jit.repos* in absolute mode). That kind of spatial map would leave the content of spectral matrix intact (*one column matrix can be of course stretched to preferred width if needed). But since the maximum number of rows (or columns), that can be created with *multislider* is 256, we need to upsample and (linearly) interpolate the matrix. For instance if we generate a matrix, that consists of 1 column and 257 rows, filled with values that increase linearly from 0 (first row) to 256 (last row), we can multiply the matrix by 8 and upsample it to the size of 2049 rows. Using the linear interpolation the result is linear increase from 0 to 2048. As we cut the last row we get the Y spatial map that

causes no change to spectral matrix (linear increase from 0 to 2047). Actually we do not have to cut the last row but only upsample the matrix to 2048 instead of 2049 rows.

Now the problem is that *multislider* can not generate a matrix with 257 rows or columns. Therefore the upper limit for using multislider for such purposes is 129 sliders. Since relatively small accuracy can be achieved when using 129 sliders to control the frequency content of a 2048 row matrix (1 single slider controls 16 bins at the same time), *multislider* is not a good choice for spatial map generation. On the other hand, when using small FFT sizes, *multislider* presents no problems. If upsampling is not needed, *multislider* can control a frequency content of a spectral matrix at bin accuracy level for up to 512 samples large FFT's.

When using *jit.repos* in relative mode, we face the same problems. In relative mode, our initial spatial map consist of only zeroes. But since the deviation from zero has to go from +2047 to -2047, we again need an initial matrix that consist of $2^n + 1$ rows.

5.3.2.2 Creating Spatial Maps with Itable Object

Itable object is a table with graphical user interface that can be imagined as *multislider* with "unlimited" amount of "sliders". The values of data in the *itable* or the "position of sliders" can be also controlled by drawing straight lines (automatic linear interpolation between two drawn vertices) in the itable *display* that is large bonus. On the other hand the major weakness of the object in question is its practical awkwardness. It does not accept and output lists and it does not

output the data as we change the content by drawing interaction. So first we will present some workarounds on how to overcome mentioned problems.

Using the *mousestate* object in combination with few *if* statements enable us to get a 0/1 flag when clicking on *itable* surface. *Itable* should be placed in upper left corner of *bpatcher* to make the use of *mode 1* attribute of *mousestate* object as simple as possible. *Mode 1* attribute gives us a relative coordinate system considering the whole screen or an absolute coordinate system inside the *bpatcher* itself. Wherever we place the *bpatcher* in the main patch, the output of *mousestate* will always show (0, 0) in the upper left corner. Therefore we can freely move the *bpatcher* without the need to adjust the *if* conditions that are restricting the area of the *itable*.

In order to write a list in a table we first need to break up a list in a series of elements. Then we need to pack each element with its successive number and send the generated pairs to *itable* (see *subpatcher pairListElementsWithSuccessiveNumbers* in fig. 5.41.). To read the *itable* we can send the *dump* message to the *itable*. The *dump* message is triggered each time a new list is received or can be driven by *metro* object on the other hand (see *subpatcher readtable* in fig. 5.41.). In order to preserve CPU power, the *metro* should be turned on only when we click on the *itable,* and turned off as we release the mouse (*see 0/1 flag in previous paragraph).

At the output of *itable* we need to generate a list again from separate elements. The incoming and out coming list and also the size of the table should all be the same.

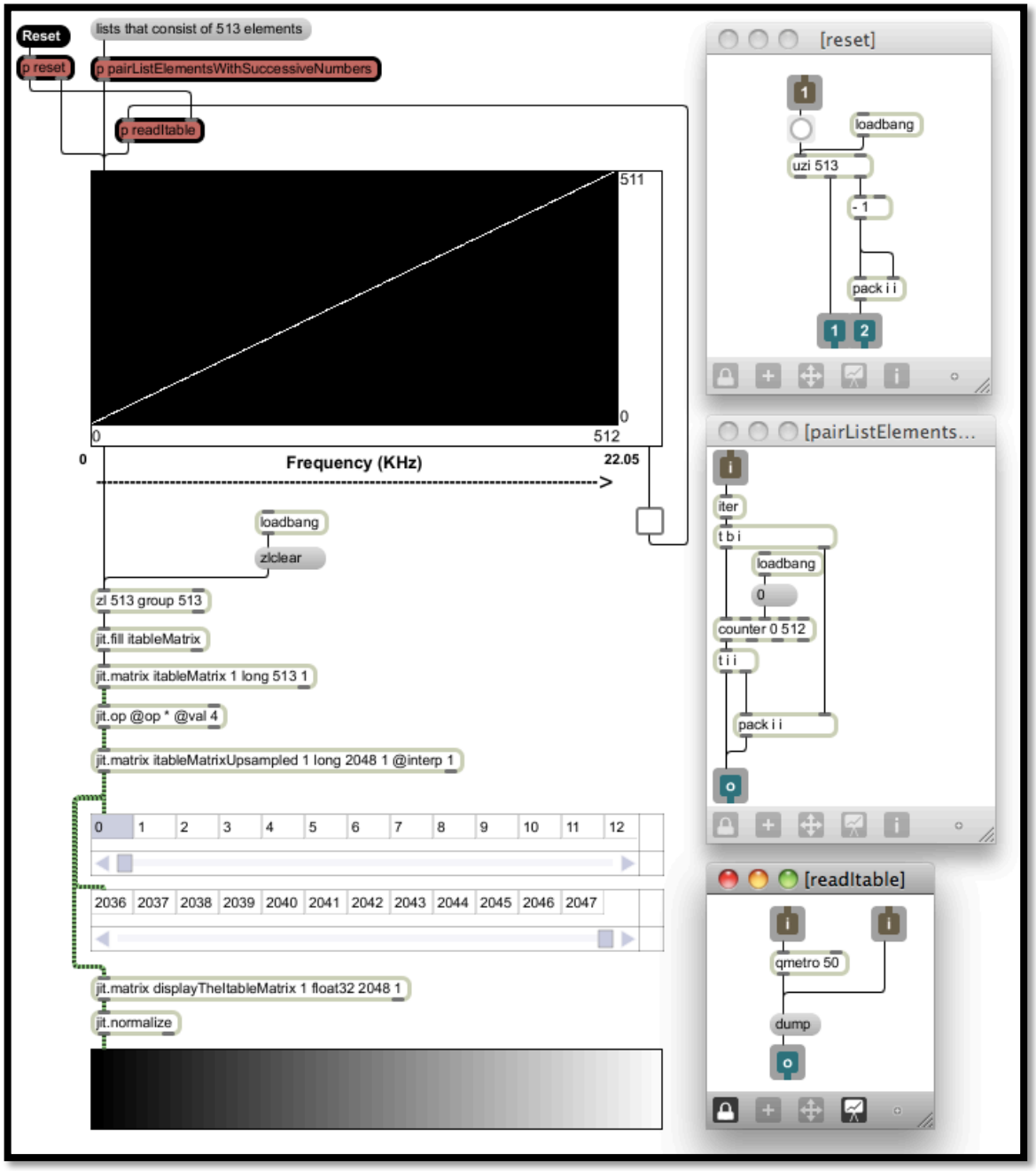

Figure 5.41 Using the *itable* of size 513 x 512 and linear transfer function to generate a 2048 x 1 relative spatial map for *jit.repos* with the method of *upsampling* and *linear interpolation.* The two *jit.cellblock* objsects display the first and the last 13 values of 2048 x 1 matrix. The whole content of a matrix is also visualized in *jit.pwindow* (original).

With presented workaround the *itable* becomes a very powerful user interface for generating spatial maps. The “only” weakness in terms of user interface is its inability to be positioned horizontally.

When using the *itable* of size 513 to generate linear Y spatial maps to control 2048 matrix rows its accuracy is one “slider” per 4 rows. Since we probably do not want larger *itables,* but we want better accuracy in the lower part of the spectrum at the same time, the solution comes again in a form of logarithmic scaling.

The most optimal solution found in the scope of this research suggests that we always generate linear spatial maps and then logarithmically scale them. The solution also suggests that we use relative maps rather then absolute ones. If we take a look again at example presented in figure 5.41, we see that the diagonal (transfer function) in the *itable* creates the initial absolute spatial map that has no effect on repositioning. If we subtract the absolute map from itself, we get an initial relative map (2048 x 1 matrix that consist of only zeroes) for the same diagonal layout of “sliders”. The benefit of using relative maps is their initial “0 state”. Considering the fact that anything to the power of 0 equals 1 and that the logarithm of 1 equals 0, we can easily keep the diagonal layout as our initial state regardless of further mapping (see sub-sub-sub-section 5.3.2.2.1 for further explanation). That way we can switch between various scaling methods and keep the linear diagonal as a transfer function that generates the initial “0” matrix. On the other hand when using absolute spatial maps, we would need to draw an inverse function to the scaling function into the *itable* to achieve the initial 0 state. Although there are exceptional cases where that would not be needed (if X and Y spatial maps would be equally scaled).

In the following practical examples we will be using logarithmic scaling. That way we can achieve a bin accurate control for the lower part of the spectrum. Also it is worth mentioning that with log. scaling some of the first “sliders” become either inactive or have control over one and the same bin (that depends on implementation). On the other hand the later “sliders”, and especially the last one, control larger amount of bins or matrix rows. That is an extra reason why it is handy to have the ability to draw lines in *itable* object.

5.3.2.2.1 Frequency Warp

The idea and the name behind this spectral processing technique come from FreqWarp plug-in from GRM (Groupe de Recherches Musicales) tools ST bundle. With FreqWarp plug-in one can “transform a sound by rearranging its frequency components in a completely free and creative way” (GRM, p. 22). This is done by remapping the input spectrum to the output spectrum by drawing a transfer function. When transfer function is linear, going from low-left corner to high-right corner, the input and output spectrums are the same. When transfer function goes from high-left corner to low-right corner in linear way, the spectrum is inversed (fig. 5.42).

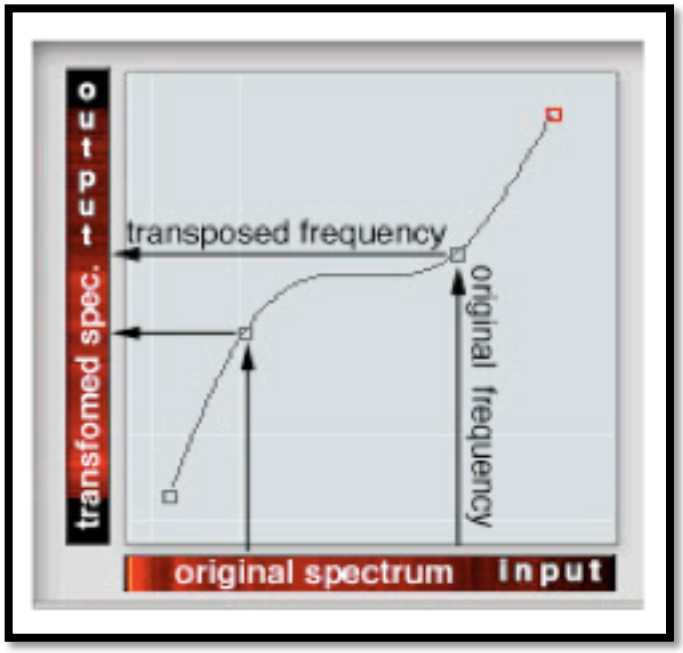

Figure 5.42 The role of transfer function in GRM FreqWarp plugin. Both axis represent frequency spectrum – X is the input spectrum while Y is the output spectrum (GRM).

Creating a frequency warp with linear scale is not much further from the example presented in figure 5.41. All we need to do is change the row of matrix called *itableMatrixUpsampled* into a column and stretch is to the preferred width (the width of processing area or the whole spectrum measured in FFT frames) so we have identical columns. That is necessary since *jit.repos* works on cell to cell basis. The next step is packing together the Y spatial map with X spatial map and send altogether to *jit.repos* as 2 plane matrix or XY spatial map. That would give us an absolute spatial map for *jit.repos* in absolute mode.

When using relative mode the X spatial map should be full of zeroes (*X spatial map is constant during all the processing since we are remapping only the frequency content of the spectrum). To make the absolute Y map relative we need to store first the content of an *itable* in its initial stage (transfer function as linear diagonal) into a list. That list should be subtracted from each list that

comes out from *itable*. Transfer function as flat line at the bottom of the *itable* should therefore result in a 513 elements long list with values going linearly from 0 to -512. Flat line at the top of the table would give us values from 512 to 0 and the diagonal transfer function would give us only 513 zeroes.

The things get a bit more complicated when using log. scaling. The problem is that our Y spatial map represents the input and output spectrum at the same time (see fig. 5.43).

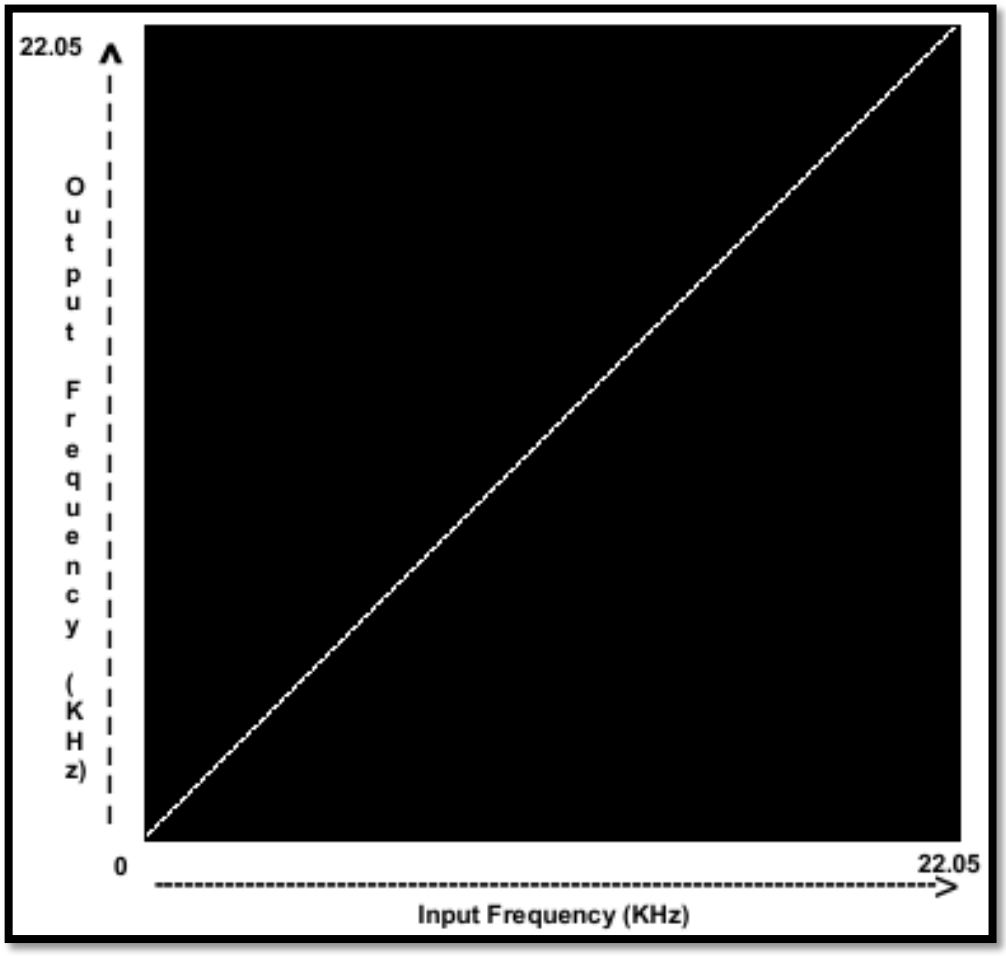

Figure 5.43 Transfer function that controls the input and output spectrum by defining Y spatial map. The cell *positions* of Y spatial matrix present input spectrum while cell *values* present the output spectrum (original).

To avoid the confusion at this point we should think of Y spatial map only as a one column matrix. The *values* in the column relates to Y axis while the cell *positions* relates to X axis. For instance if 6th cell holds the value 5 (6. Number as we count from 0), the 6th frequency bin is mapped to 6th frequency bin resulting in no change. Therefore we need to perform two different scaling methods in order to equally scale X and Y axis. For X axis we need *jit.repos* since we are dealing with *positions* while for Y axis we need either a lookup table or mathematical expression on each *value*. Hence the X axis needs to be scaled as a matrix (due to *jit.repos*), while Y axis can be easier scaled as a list (we process the list before we convert it into a column of a matrix). See figure 5.44 for details on scaling.

The additional obstacle when using logarithmic scaling presents the fact that logarithm is not defined for negative numbers. Therefore we need to "extract" the negative (and positive) sign of each list element and calculate logarithm only for positive values. After the calculation we need to attach the original sign back to each list element. This can be done by using only two *clip* and one *vexpr* object (see subpatcher LogScaleY in fig. 5.44; the *vexpr* object has green colour).

The last obstacle in terms of logarithmic scaling poses the inability of *expr* and *vexpr* objects to calculate logarithms with arbitrary base (only bases 10 and *e* are available). Hence we need to use a mathematical trick to calculate the log. with base 512.

$$log_{512}(x) = \frac{log_n(x)}{log_n(512)}$$

where n is an arbitrary base.

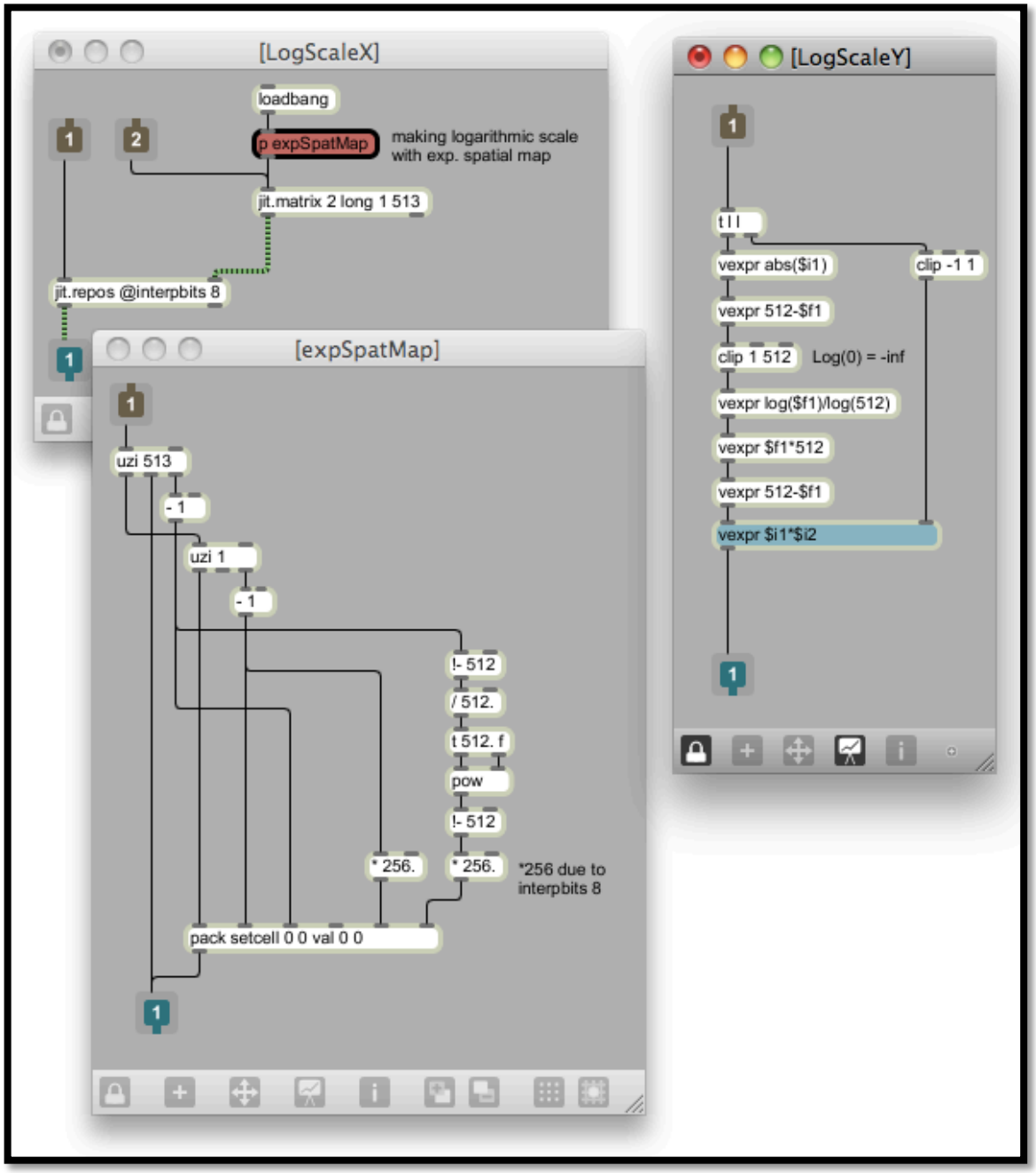

Figure 5.44 Scaling the X and Y axis of *itable*. The X axis is logarithmically scaled with *jit.repos* through exponential spatial map. On the other hand Y axis is scaled by logarithmic expression (*note that log. expression is the inverse of exponential one used for spatial map). The *values* from *itable* that represent Y axis are scaled as *lists* while the *positions* that represent the X axis are scaled or repositioned as a *matrix*. Both scalings are done before the initial matrix is upsampled to the size of 2048 rows (original).

The resulting one column matrix is at the end upsampled and interpolated to preferred size, packed with X spatial map (constant zeroes) and sent to second *jit.repos* as final XY spatial map. The effect of frequency warp effect can be seen in figure 5.45.

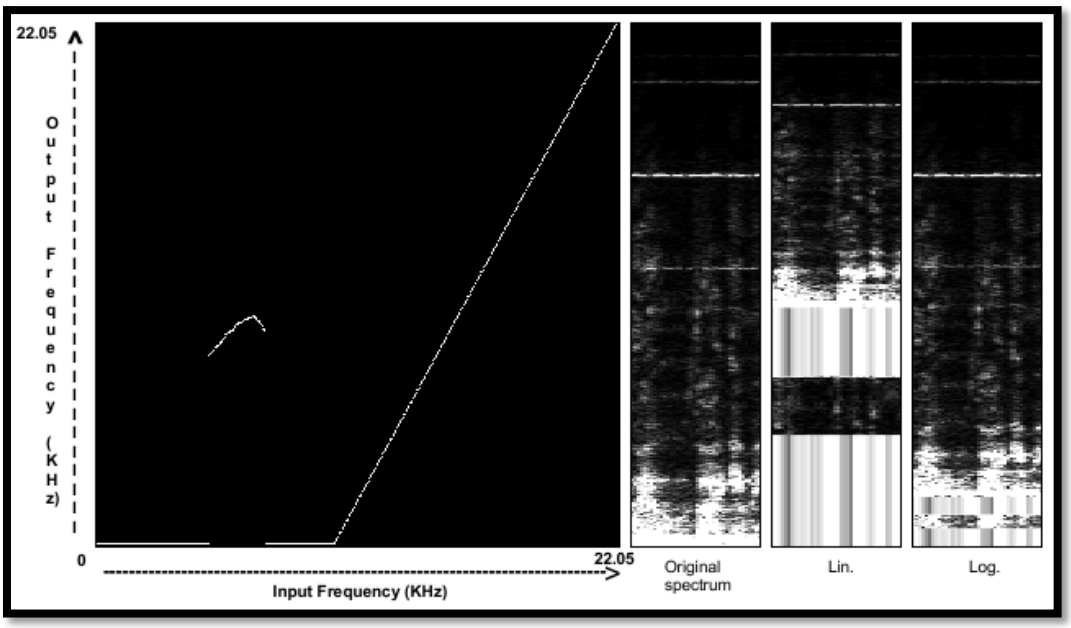

Figure 5.45 From left to right: transfer function drawn in *itable*; original spectrum; the effect of transfer function on the spectrum when using linear scaling; the effect of transfer function on the spectrum when using logarithmic scaling (original).

One of the potential problems of frequency warp effect is its ability to spread one frequency bin over the significant range of the spectrum as it can be seen in figure 5.45. This results in a very distinctive "FFT sound" that might be undesirable, although is really only a matter of taste. In order to get rid of that side effect, the frequency warp effect can be combined with set of band-pass spectral filters. This seems to be the case in actual FreqWarp plug-in from GRM.

All reposition should be done on both planes of spectral matrix.

5.3.2.2.2 Spectral Delay

"Delaying individual FFT bins in a short-time Fourier transform, can create interesting musical effects that are unobtainable with more traditional types of delay techniques" (Kim-Boyle, 2004). Here we present a sonographic version of the spectral delay concept, presented in the paper by David Kim-Boyle (2004).

As suggested by David Kim-Boyle, spectral delay can be implemented in Max/MSP by using time domain *delay~* object inside the *pfft~* subpatcher. By storing a (potentially different) delay time for every single FFT bin in a *buffer~* object (the size of buffer in samples should be the same as the number of FFT bins where each *buffer~* sample holds the delay information for corresponding frequency bin) we can create arbitrary spectral delays. If the *buffer~* is being read at the same speed as the *fftin~* object outputs FFT data, each buffer sample can dynamically change the delay time of the *delay~* object for every single FFT bin. Using spectrogram as a PV data storage, this can be implemented in Jitter by moving the rows of spectral matrix backwards (negative delay time) and forwards (positive delay time). The delay feedback on the other hand can be simulated by using the technique presented in sub-section 5.2.2. Also we could simulate the feedback effect by copying, scaling and adding the scaled spectrogram over the top of existing one with certain offset.

The role of the *buffer~* object will be replaced by one column matrix or X spatial map for *jit.repos*. Therefore each cell of one column X spatial map will correspond to single row in spectral matrix. The transfer function that will generate the map will be drawn in *itable* object, and since we are using Jitter to process the reposition data, we have much more flexibility

available as when using MSP object *buffer~*. Even when using the spectral delay technique from David Kim-Boyle that is not based on PV and spectrogram, we suggest using matrix for the delay time storage instead of a *buffer~*. Since matrices can be read with audio signals, sample accuracy can be obtained.

The main benefits of using matrices as the delay time storage objects are their ability of upsampling and interpolation, logarithmic repositioning and "smoothing" their data (delay time information). That way we can again achieve desirable accuracy when working with large FFT sizes (see sub-sub-sub-section 5.3.2.2.1). The "smoothing" of delay time data can be done by using vertical Gaussian blur on spatial map. Various examples of practical use of sonographic spectral delay (with matrix as a data storage object) can be seen on figures 5.46 to 5.51. Also it is worth mentioning, that in case of spectral delay it would be handier if *itable* could be positioned horizontally and scaled to the height of spectrum display.

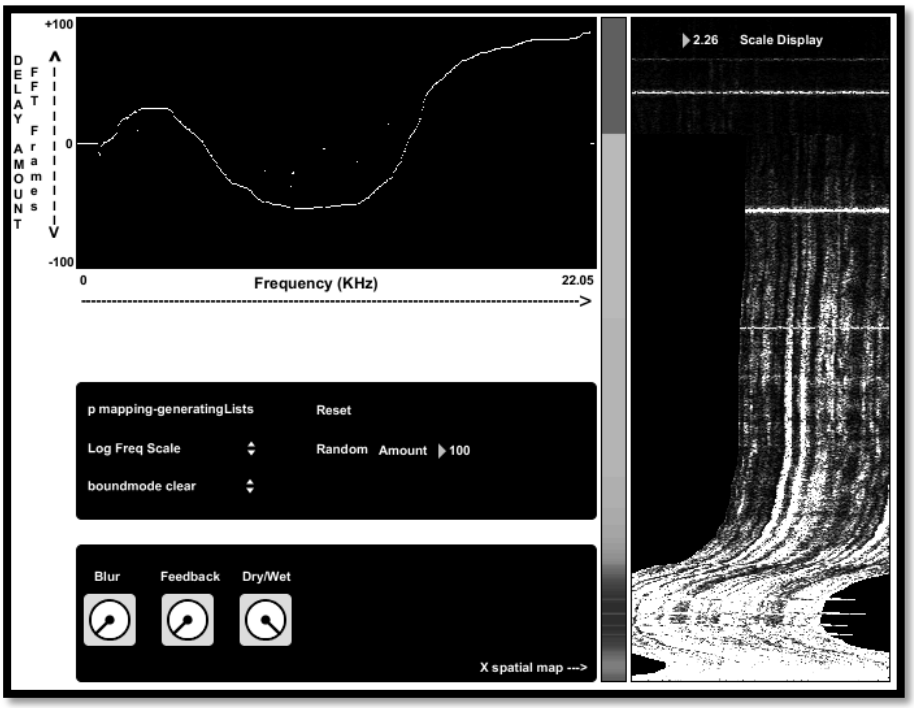

Figure 5.46 Transfer function drawn in itable, X spatial map (thin vertical display) and the effect of spectral delay on the spectrum. In this example, logarithmic frequency scale was used (original).

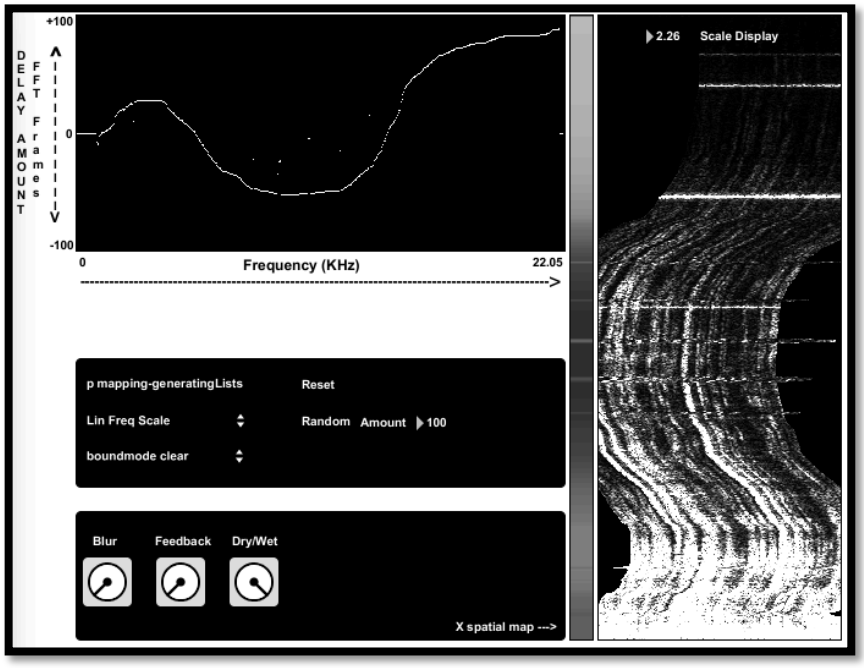

Figure 5.47 Transfer function drawn in itable, X spatial map (thin vertical display) and the effect of spectral delay on the spectrum. In this example, linear frequency scale was used (original).

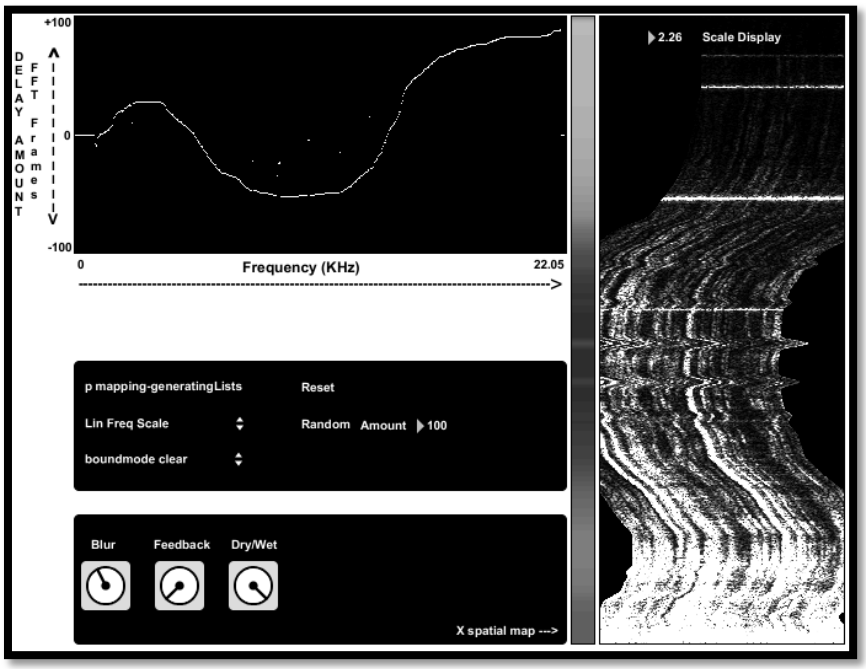

Figure 5.48 Example from fig. 5.47 with vertical Gaussian blur added to X spatial map. Applied Gaussian blur to spatial map results in bigger continuity of spectral transformation (original).

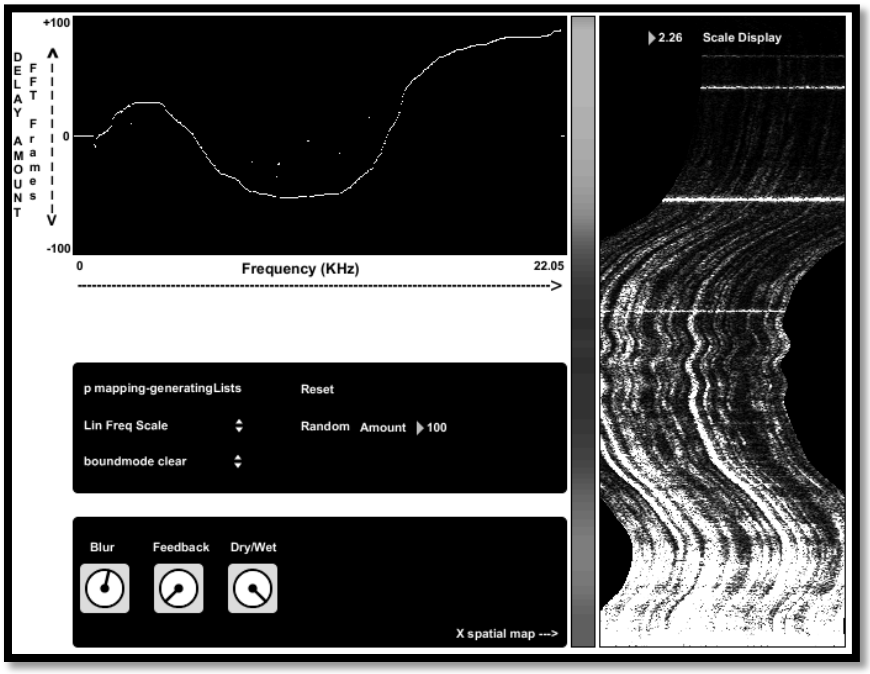

Figure 5.49 Example from fig. 5.48 with bigger amount of Gaussian blur (original).

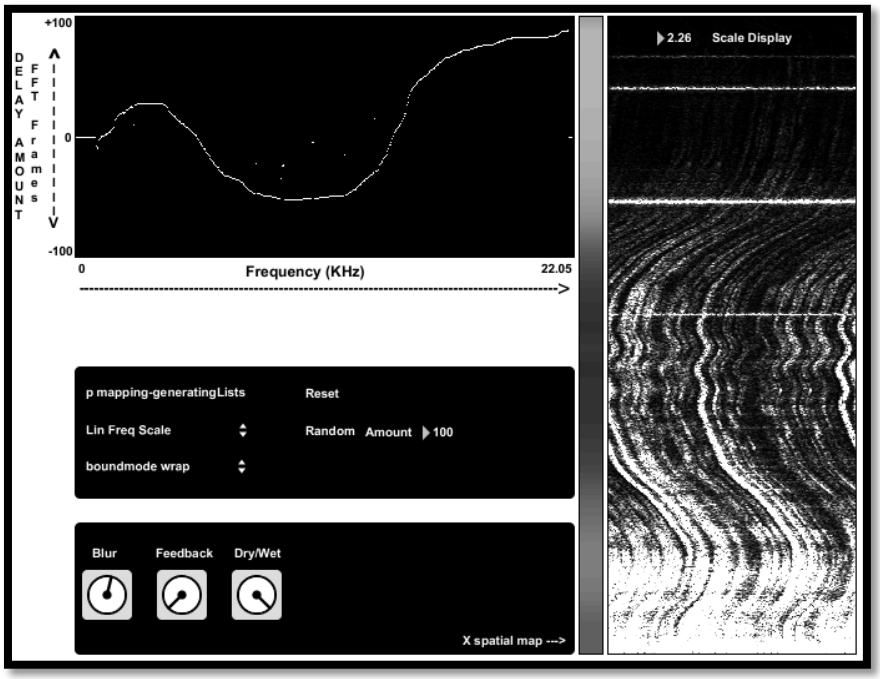

Figure 5.50 Example from fig. 5.49 using the *wrap* mode of *jit.repos* object. The missing elements of the spectrum due to repositioning are wrapped back. That way we get rid of the discontinuity that appears when looping the spectrogram by jumping from last to the first column of spectral matrix (when using *phasor~* to do the reading in X direction for instance) (original).

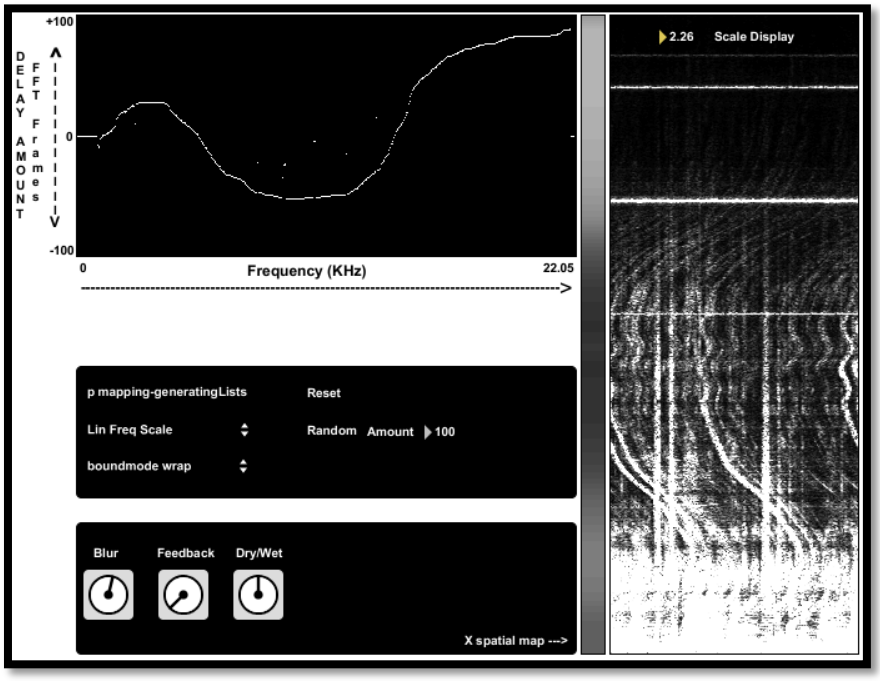

Figure 5.51 Example from fig. 5.50 using dry/wet function in 1:1 relation. We can see the unprocessed spectrum in the background (original).

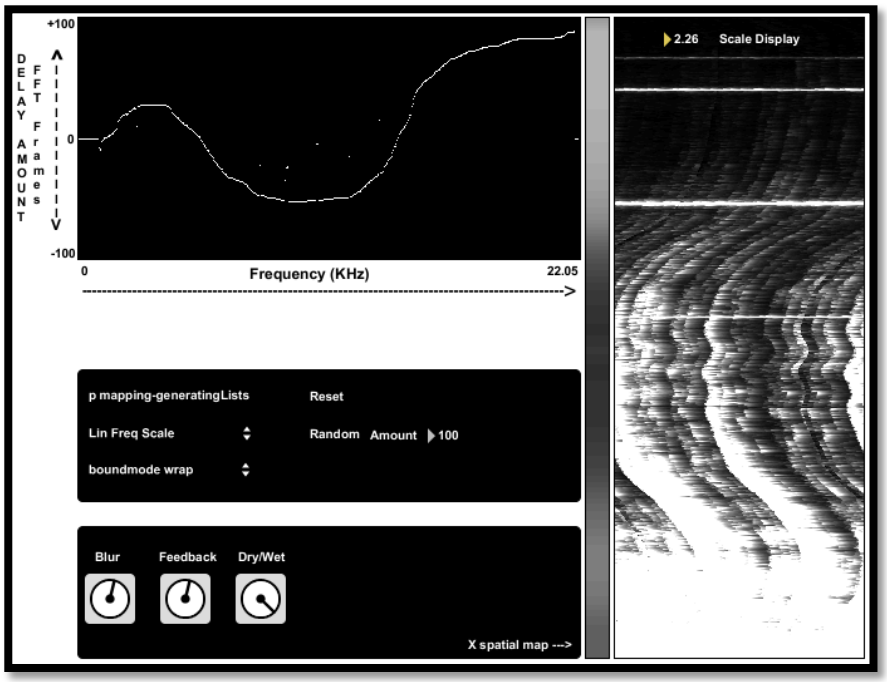

Figure 5.51 Applying a feedback using the technique presented in sub-section 5.2.2 (original).

The implementation of spectral delay is a conglomerate of concepts already presented in this paper (fig 5.52). As opposed to frequency warp example, we are repositioning X spatial map this time. The *values* of one column X spatial map represent delay times, while the *positions* relates to frequency spectrum. In order to use the log. freq. scale, we therefore do not need to scale the *values* of a matrix this time, but only the *positions*. Hence we need two *jit.repos* objects – first one for logarithmic scaling of X spatial map and the second one for the actual reposition of the spectrum. Y spatial map should be filled with zeroes as we use the main *jit.repos* in relative mode.

Spectral delay presented in figure 5.52 has the delay time range of +/- 100 FFT frames. From figure 5.52 is not evident that the initial values of X spatial map are going from 0 to 200 and not from -100 to +100. The reason for that is the use of object *jit.gl.asyncread* to get the matrix out of openGL, where Gaussian blur takes place. Since *jit.gl.asyncread* supports only char data, we can not work with negative numbers. Hence we scale the range from 0 to 200 into -100 to 100 after the openGL part. That results in grey colour of X spatial map for 0 delay times, black colour for delay times of – 100 FFT frames and almost white colour (value 200) for delay times of +100 FFT frames.

All reposition should be done on both planes of spectral matrix.

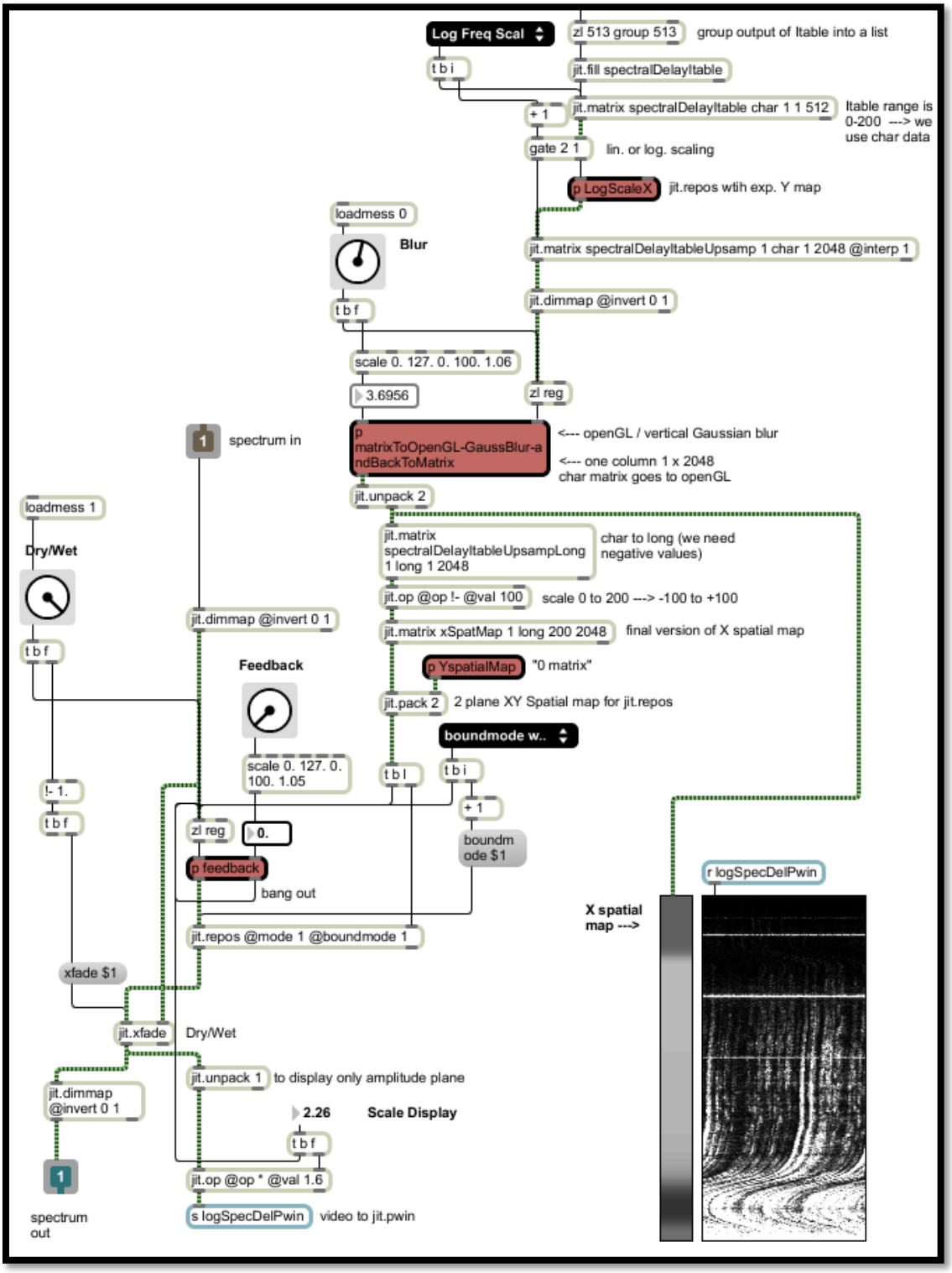

Figure 5.52 The implementation of spectral delay with the ability to switch between linear and logarithmic frequency scale. The *zl* object in *group* mode, at the top of the picture, turns the output of *itable* (not visible) into a *list* (original).

5.3.2.2.3 Spectral Rolling Pin

If we imagine spectrogram as a dough, then we can use the spectral rolling pin to roll any area of the spectrogram in the direction of rolling. That would give us intuitive, flexible and sensible way to reposition the spectral data. By turning the mouse cursor into a “rolling pin”, we can achieve various visual effects that depend on shape of the rolling pin, direction of rolling and the time of rolling (time in that case can be also considered as a force). The example of rolling pin in use can be seen in figures 5.53 and 5.54.

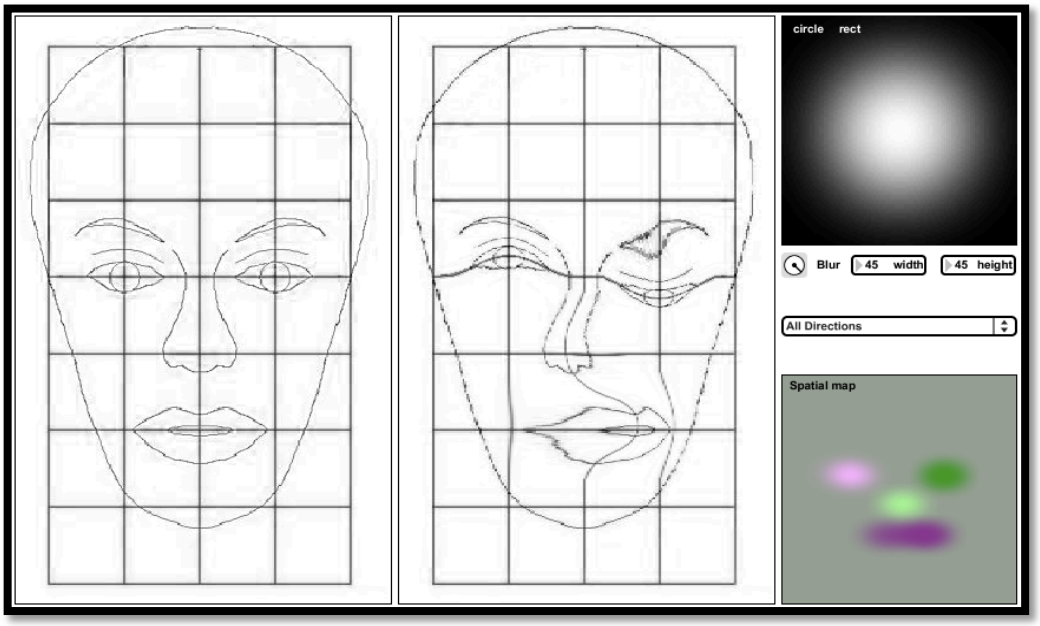

Figure 5.53 At the left hand side is the original picture and at the right hand side is the repositioned one. The repositioning was achieved by dragging the mouse over the left picture in various directions. Left eye was repositioned in upper direction, right eye in lower direction, nose in left direction and mouth in right direction. The shape of the “rolling pin” used can be seen in upper right corner (circle with radius 45 pixels, blurred and normalized). Normalized version of spatial map that caused the repositioning, and was generated during the mouse dragging, is located in lower right corner. (original).

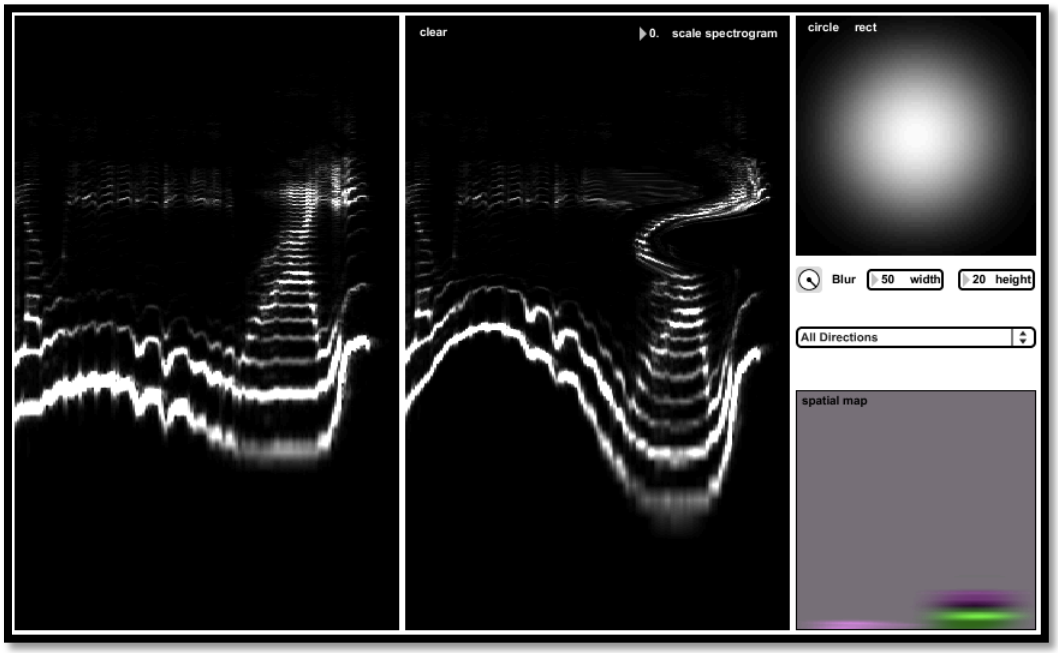

Figure 5.54 Using spectral rolling pin on logarithmically scaled spectrogram. On the left hand side is the original spectrum, while the repositioned one is on the right hand side. The shape of the rolling pin used was blurred ellipse, obtained from circle in the right top corner by stretched it to width 50 and height 20 pixels. Linear and normalized spatial map for linear FFT matrix can be seen in lower right corner (original).

The implementation of spectral rolling pin is based around the object *jit.glop,* that offers feedback with gain stage applied either to input our output. The core idea for the implementation was taken from the unnamed *patch* made by Cycling'74 forum member "nesa". The concept behind the implementation will be explained on linear example using reposition in only one direction (fig. 5.55).

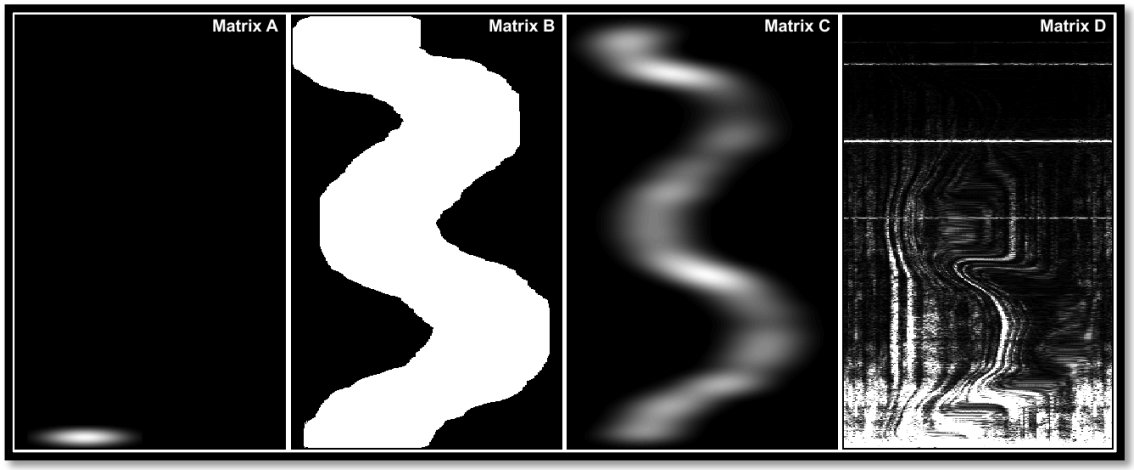

Figure 5.55 From left to right: Moving matrix of adjustable size containing ellipse ("rolling pin") inside the 200 x 2048 matrix A. The trajectory of "rolling pin" is defined by mouse movements over the spectrogram; Matrix B - Tracing the trajectory by using the feedback (the values do not clip at 255 as it seems on the picture but can go all the way up to the maximum possible values determined by chosen data type); Matrix C - Normalized matrix B from second picture due to only better visual representation; Matrix D - The effect of matrix B, as X spatial map, on the spectrogram (original).

As we can see in figure 5.55, the matrix that represents spatial map is matrix B. Matrix B is generated by feeding matrix A into *jit.glop* with feedback gain stage applied to input. The function of *jit.glop* could be replaced with feedback loop using the + operator of object *jit.op* (see fig. 5.38 as an example). The reason why we are using *jit.glop* lies in its ability to apply positive or negative feedback gain on arbitrary matrix plane.

Matrix A and B are both two plane matrices, since we need X and Y spatial maps for *jit.repos*. First plane represents the X spatial map, while second plane represents Y spatial map. Therefore if we want to achieve a reposition in right direction, we apply negative feedback to the first plane of matrix A. Similarly positive feedback on first plane of matrix A causes reposition in left direction (this example is presented in fig. 5.55). The same holds for applying negative or positive feedback to the second plane of matrix A that results in up and down repositions.

Since we are adding a feedback to input matrix A, the values in matrix B are changing only at the area of current "rolling pin" position. Due to the fact that "rolling pin" is the only non zero value in matrix A, feedback has no effect on surrounding "black" area.

If we subtract the current mouse coordinates from the previous ones, when moving ("rolling") across the spectrogram, we can determine the direction of mouse movement (*this should be done separately for X and Y direction). When the direction of mouse movements controls the target (plane) and the type of feedback (positive or negative), we get the repositions that are in chord with the mouse movements. Also the feedback process should be performed only "on mouse down".

When we visualize matrix B as a whole two plane XY spatial map, filled with all 4 possible types of information, the result is a mesh of four different colours. For clearer visualization we have to normalize the matrix B, since matrix B can contain extremely huge values. The result is Matrix C, that resembles the colourful drawing with spray. This can be seen in lower right corner in figures 5.53 and 5.54.

Since the output of *jit.glop* can be very rapid when "mousing", it can pose a huge CPU burden when using repositioning on large matrices. That may very likely introduce unwanted audio interruptions on majority of computers, when reading and transforming the spectrogram at the same time. To overcome that problem, the output of *jit.glop* can be stored into the *zl* object using the *reg* argument. The output of *zl* object can be controlled by using *metro* at preferable speed, that should be turned on as we click on the spectrogram, and turned off as the mouse button is released (fig. 5.56).

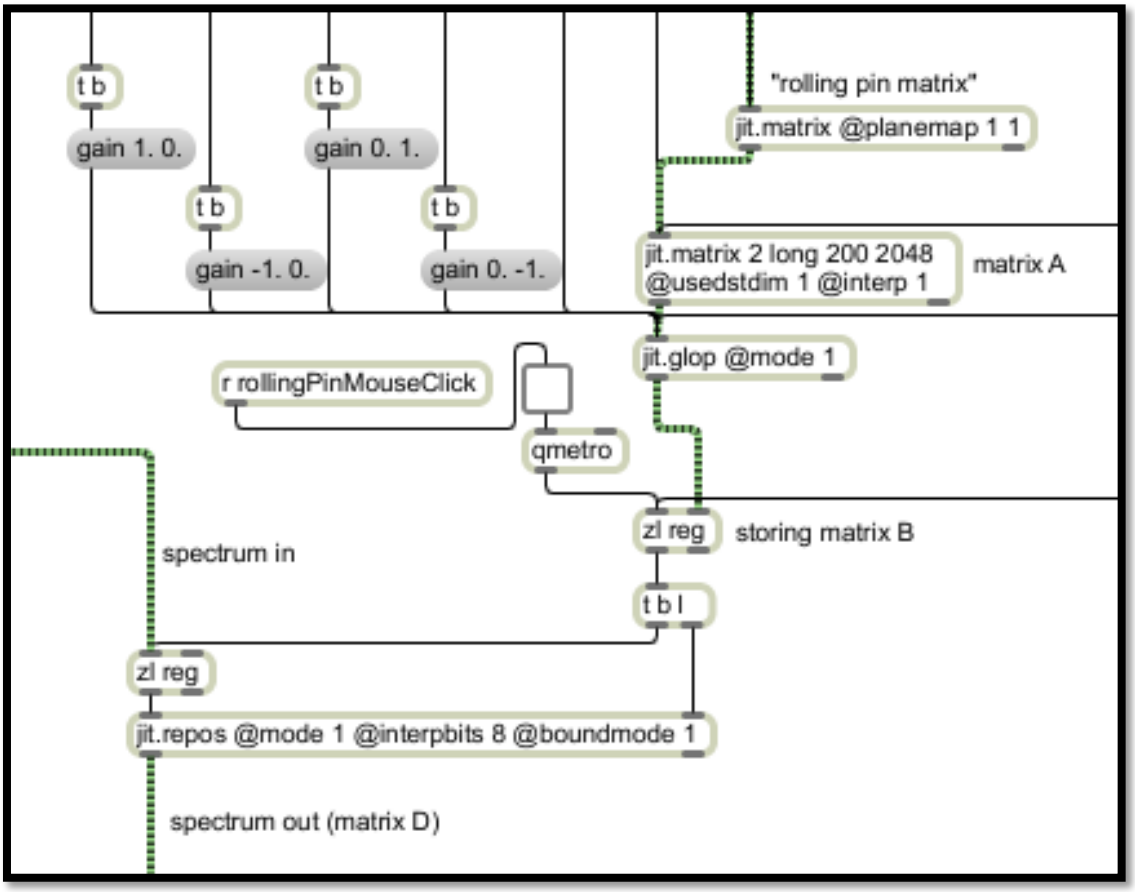

Figure 5.56 Storing matrix B in *zl* object. The output of *zl* is caused by *metro* object, that is turned on and of by clicking on the spectrogram. This prevents undesired audio interruptions when reading and transforming the spectrogram at the same time (original).

The solution presented in fig. 5.56 is applicable to any method of spectral processing when experiencing either audio or video glitches due to too intensive processing.

When using very small “rolling pins”, for accurate repositioning, it is very handy to have the undo option. That could be implemented by separately storing one or more previous matrices B.

5.4 Spectral Interactions

Processing techniques labeled as spectral interactions are considered as techniques, where spectrum of one sound modifies the spectrum of the other sound, or even its own spectrum. This could be done via direct spectral interaction (various forms of cross-synthesis) or indirect spectral interaction (FFT data as a modulation source)

5.4.1 Cross-Synthesis Using Masks

> *Cross-synthesis means different things depending on the system being used (LPC, convolution, phase vocoder, wavelets, etc.). In general, it refers to techniques that start from an analysis of two sounds and use the characteristics of one sound to modify the characteristics of another sound, often involving a spectrum transformation.*
>
> (Rhoads, 1996, p. 208).

Once in the frequency domain, convolution and many basic types of cross-synthesis are very straightforward. It is all about multiplication, summation, and swapping of two spectral planes of individual sounds. Therefore we present here a technique that takes the advantage of sonographic processing – sound morphing via masks.

Fast convolution, frequency domain equivalent of circular convolution, is defined as multiplication of amplitude spectrum and addition of phase spectrum of two sounds. When using masks in Jitter, to localize the spectral interaction, that poses an additional problem. That is why an example where both the addition and multiplication take place has been chosen. Actually we will not be presenting a technically correct convolution, because we will be summing phase differences instead of phases, but the methods and obstacles are exactly the same. The main purpose of this chapter is hence not to present cross-synthesis or convolution, but to emphasize the problem that appears when using masks for multiplication of two spectral planes.

In order to perform localized convolution, regardless of using phases or phased differences, the mask has to be applied to both spectral planes. If using masks with feather, the feather should be applied only to amplitude plane mask, since we want the phases intact or unscaled. Also the mask should be applied to only one sound or spectrum.

When adding together phase spectrum of one sound, and masked phase spectrum of the other sound, there are no problems. The black area of the masked phase spectrum, or zero values, does not affect the phases of the other sound since x + 0 = x. The problem appears with multiplication, in this case the multiplication of amplitude planes, because we need the change the 0 values of masked amplitude spectrum into 1 (x * 1 = x). In classical programming, the problem would be easily solved by a simple if statement: "If value in the matrix equals 0, then change it to 1, else do nothing". But since there is no *if* object in Jitter, we need to make one ourselves by using *jit.expr*. Here is an example of required if statement equivalent:

```
(in[0]==0.) + ((in[0]>0.)*in[0])
```

When input equals 0, the result of the equation is:

$$(1) + (0*0) = 1$$

When input is more that 0, the result is:

$$(0) + (1 * \text{input}) = \text{input}$$

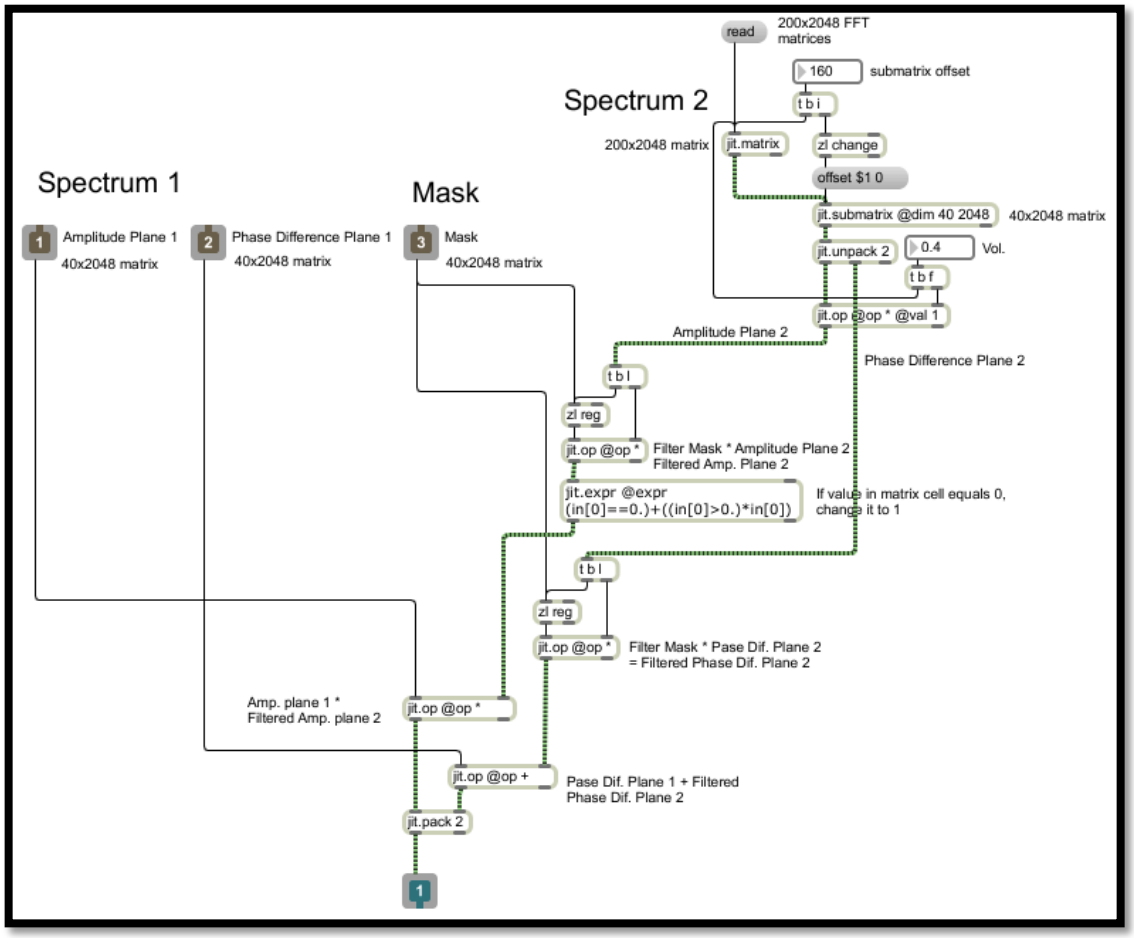

Figure 5.57 Implementation of phase differences based "convolution" using masks. Implementation of technically correct fast convolution, using phase plane, would be exactly the same, only phase difference plane should be replaced with phase plane (original).

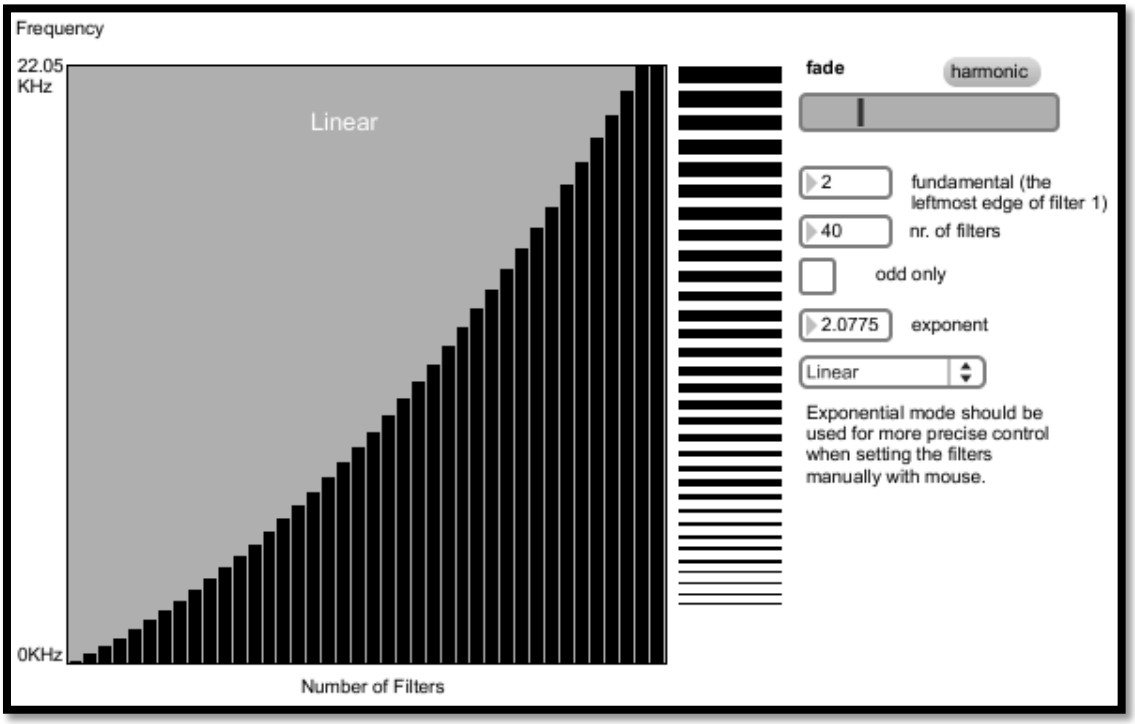

Figure 5.58 An example of mask used for localized convolution. Using the interpolation fader that morphs between two opposite exponential distribution, the brightness of convolution morph can be controlled (original).

If integrating convolution, or any other form of cross-synthesis in a compressor, presented in chapter 5.2.10, the amplitude threshold of spectral morph could be controlled. FFT based compressor splits the amplitude spectrum into two parts – above and below a certain threshold. Hence instead of compressing (scaling) the upper part, we could morph only partials that are loud enough to exceed the threshold.

One of potential problems when multiplying two amplitude planes is that two spectrums can significantly cancel each other out (with or without using masks). That happens when one spectrum is very weak in the frequency range where the other is strong and vice versa. This probably unwanted effect of spectrum multiplication can be reduced to great amount by making

the morph most prominent only in the area, where one spectrum has the most of its energy. The center of that area is called spectral centroid.

5.4.2 Centroid Based Spectral Morphing

"The (individual) centroid of a spectral frame is defined as the average frequency weighted by amplitudes, divided by the sum of the amplitudes, as follows

$c_i = \frac{\sum f_i a_i}{\sum a_i}$ " (Burk, Polansky, Repetto, Roberts and Rockmore, 2011). Individual spectral centroid, measured in Hz, tells us where the energy center of one FFT frame is located. When taking all the FFT frames into account, the resulting centroid can be used as a measure of sound brightness. "If two sounds have a radically different centroid, they are generally perceived to be timbrally distant" (Burk, Polansky, Repetto, Roberts and Rockmore, 2011).

When calculating the individual centroid from FFT data, the *f* from equation $c_i = \frac{\sum f_i a_i}{\sum a_i}$, can be replaced by *k*, where *k* is a bin number. The reason why we can do so, is that center bin frequency is defined as multiplication of bin number and fundamental FFT frequency. When translating the resulting frequency back into bin numbers, in order to position the centroid in a matrix, we have to divide the result by FFT fundamental.

$$c_i = \frac{\sum f_i a_i}{\sum a_i} = \frac{\sum F_{FFT} k_i a_i}{\sum a_i} = \frac{F_{FFT} \sum k_i a_i}{\sum a_i}$$

$$c_{BINi} = \frac{F_{FFT} \sum k_i a_i}{F_{FFT} \sum a_i} = \frac{\sum k_i a_i}{\sum a_i}$$

From phase vocoder analysis data a more accurate centroid can be calculated, using the true bin frequencies instead of center bin frequencies. True bin frequencies can be calculated using the equations 3.13 and 3.14. But the centroid based spectral morphing technique presented in this sub-section, is based on morphing a larger area around one sound's individual centroid. So there is no need for a very accurate centroid estimation.

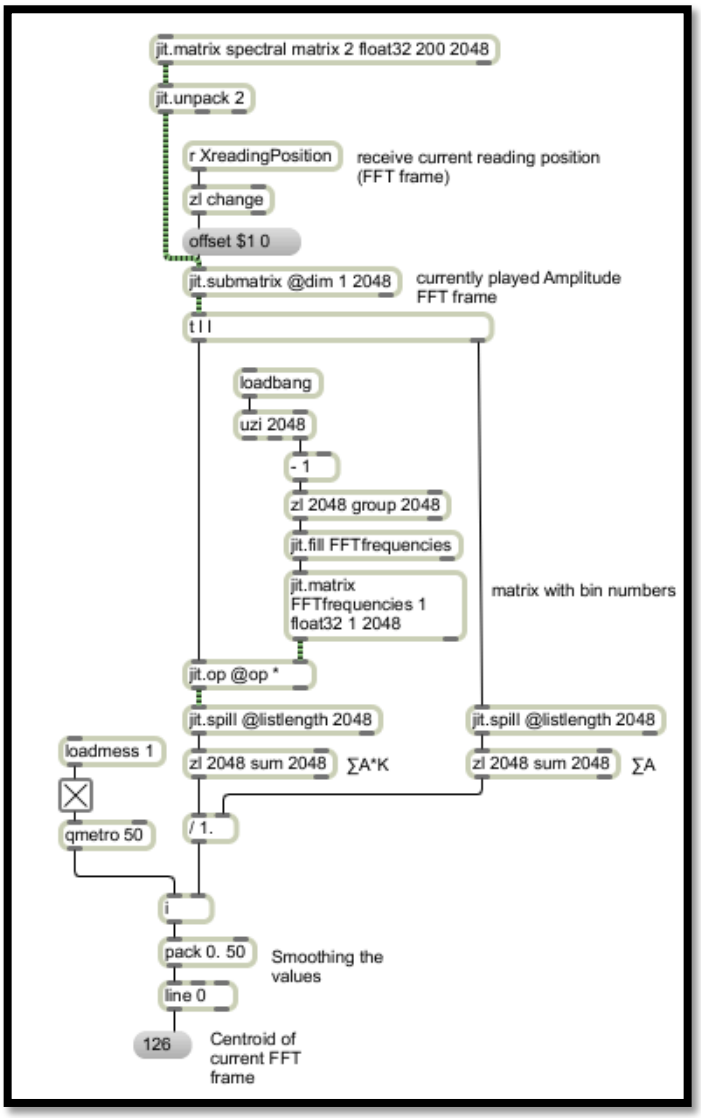

Figure 5.59 Calculating the centroid of currently played FFT frame (original).

Since there is no object in Jitter that would calculate the sum of a matrix row or a column, the matrix can be transformed into a list, where the summation of its elements can be calculated using the *sum* attribute of the *zl* object (*there are externals available that calculate the desired sum in a matrix) (fig. 5.59).

If individual centroid is used as a modulation source of morphing mask position (individual centroid at the same time determines the center of a mask), the result is a brighter morph with less spectral cancelations. When morphing between two sounds of equal length, the reading position of one sound can also present the reading position of the other sound. In such case, we get the static centroid or static morphing mask when reading speed is 0. For all other reading speeds, the morphing area is dynamically changing.

Centroid can be also calculated from interpolated FFT frame.

5.4.3 Indirect Spectral Interaction

If spectrum of one sound controls the parameters of certain spectral effect, that is applied to other or the same sound, we call that indirect spectral interaction. The spectral modulation can be implemented on bin to bin basis. For instance each amplitude FFT bin of one sound's single FFT frame, can control a spectral delay of the other sound at bin accuracy. That way we can impose certain spectral characteristics of one sound onto the other. Also the feedback loops are possible, when spectrum of one sound controls its own spectral parameters and vice versa. This can be

implemented by dynamically generating, or in a way modulating, spatial maps for *jit.repos* with FFT data.

5.4.4 Target-Based Concatenative Synthesis

"Concatenative sound synthesis (CSS) methods use a large database of source sounds, segmented into *units*, and a *unit selection* algorithm that finds the sequence of units that match best the sound or phrase to be synthesised, called the *target*" (Schwarz, 2006). The *unit* of CSS can be a single *FFT frame*.

Concatenative synthesis is a very complex subject, so we will present only the most basic approach to that synthesis. The purpose of this section is to describe a general approach to target-based concatenative synthesis and to give some conceptual guidelines for more elaborated and efficient methods.

The basic idea of target-based CSS starts with a bank of various sounds, where each FFT frame of the whole sound bank is equipped with a set of timbre descriptors. These descriptors can be considered as various coordinates in multi-dimensional coordinate system, that determine the multi-dimensional vector, representing all descriptors. Therefore the number of descriptors defines the number of dimensions. All multi-dimensional vectors of all FFT frames present a timbre identification database.

In order to perform target-based CSS, an additional sound is required. That sound is also analyzed and multi-dimensional vector is determined in real-time for each FFT frame. After the

vector is specified, a *unit selection* algorithm finds the best matching vector from stored database and replaces the original FFT frame of incoming sound with timbraly most related FFT frame from the database. For more details on that subject please see publications from William Brent (2009).

In this section we will not be dealing with vectors, because only one timbral descriptor will be used – spectral centroid. In the same way as we calculated an individual spectral centroid for currently played FFT frame in sub-section 5.4.2, we can calculate centroids for the whole FFT matrix of various sounds, and store them in a list. That list presents a timbre identification database, derived from a sound bank FFT matrix (various sounds or sound bank should be organized in preferred sound editor and exported as a single sound).

Once the centroid (*we will refer to individual spectral centroid only as centroid from now on) database is stored, we have to calculate in real-time the centroid of the incoming sound we want to resynthesize. For each FFT frame of incoming sound, the current centroid is subtracted from each element in centroid database. The lowest absolute value of the result, identifies the most closely related centroid. Once we know the position of the best match in the list, we know the number of FFT frame that will be used for CSS resynthesis. In other words, the position of best match in the list, determines the target in the sound bank matrix. Hence the position of the best match presents the horizontal reading position of the sound bank matrix (fig. 5.60).

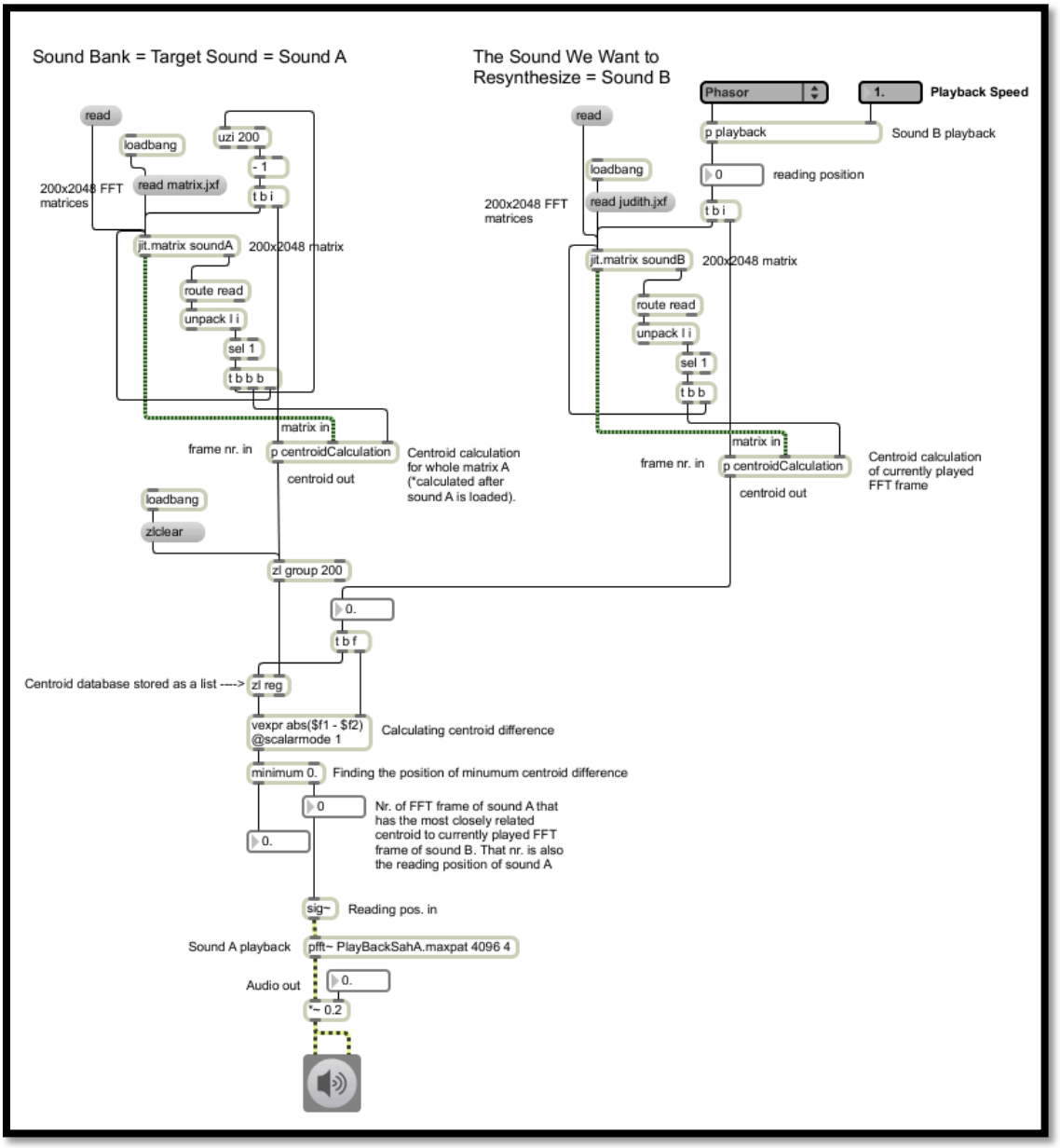

Figure 5.60 Implementation of most basic target-based concatenative sound synthesis using only one timbre descriptor (original).

If we would extend presented method with additional descriptor, our 1D list elements would become 2D vectors, existing in X-Y coordinate system, where X would be a centroid, while Y could be brightness or any other timbral descriptor. Instead of vector, we could also think of our

resulting descriptor as a single point in 2D space. The descriptor database would hence look like a myriad of points in 2D coordinate system. Each FFT frame of the sound we would want to resynthesize, would also generate a point in 2D space, and the nearest neighboring point would present the most closely related timbre. Therefore we would need to calculate Euclidean distance between the current point (descriptor of currently played FFT frame) and all the points from the database. The shortest distance would identify the best match.

When working in n dimensions, we can still use the formula of Euclidian distance to find the best match. The distance between point p and q in n dimensional space equals

$$d(p,q) = d(q,p) = \sqrt[2]{(q_1 - p_1)^2 + (q_2 - p_2)^2 + \cdots + (q_n - p_n)^2}$$

Beside using Euclidean distance instead of difference, when working with more that one descriptor, normalization of data is also required. This needs to be done since various descriptors are measured in various units and spans across various ranges. Without normalization, one descriptor could totally override the others when calculating the Euclidean distance. Normalization (of list values) in Max/MSP can be done by using object *scale,* by setting the output range from 0. to 1. These conclusions are based on the conversation I had with William Brent.

5.5 FFT Data Processing on Graphic Cards

> *The performance in CPUs over the past half century has more or less followed Moore's Law, which predicts the doubling of CPU performance every 18 months. However, because the GPU's architecture does not need to be as flexible as the CPU and is inherently parallelizable (i.e. multiple pixels can be calculated independently of one another), GPUs have been streamlined to the point where performance is advancing at a much faster rate, doubling as often as every 6 months. This trend has been referred to as Moore's Law Cubed.*
>
> (Dobrian, 2001, Jitter tutorial 42)

OpenGL was used in previous sections to process masks or spatial maps that were based on 8-bit char data. The techniques used to transfer char matrices from GPU to CPU are not appropriate when transferring 32-bit data matrices. This section hence presents the solution to transfer 32-bit FFT data, or any other matrix types, from GPU to CPU. It is also worth mentioning that if time-domain signals are written in a *jit.matrix* instead of a *buffer~*, they could also be processed on GPU without the need of any specially designed shaders.

Getting the 32-bit floating point data matrix to and from the GPU is actually a very easy thing to do in Jitter. Despite the simplicity, finding the solution to the problem in question was one of the most time consuming tasks of this research.

All we need to do is to put object *jit.gl.slab* at the end of a *slab* chain, and set its dimension attribute to the size of rendering context. Since *jit.gl.slab* has a texture form output, we need to

attach four-plane matrix to the output of *jit.gl.slab.* When sending only one plane matrix to openGL, the processed matrix is stored in last three planes of "ARGB" matrix. Hence we can choose any of those three planes for further processing on CPU.

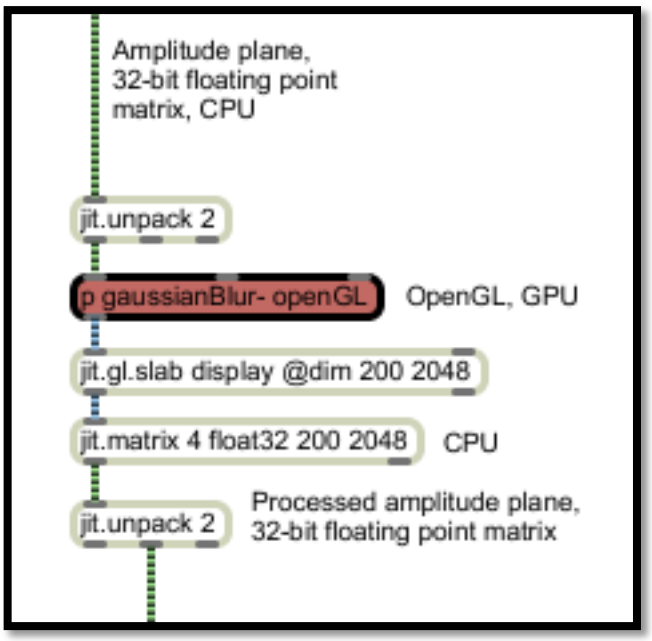

Figure 5.61 Processing FFT data as 32-bit float matrix on GPU, and using *jit.gl.slab* for GPU to CPU transition. Part of unavoidable openGL objects are also *jit.window* and *jit.render* (original).

Since the input to *jit.gl.slab* can have a form of matrix or texture, we can use that object for GPU based mathematical calculations between CPU generated matrixes and GPU generated textures. Example presented on fig. 5.62, shows the GPU based multiplication of blurred amplitude plane by noise fractal, that results in some kind of unusual spectral amplitude modulation.

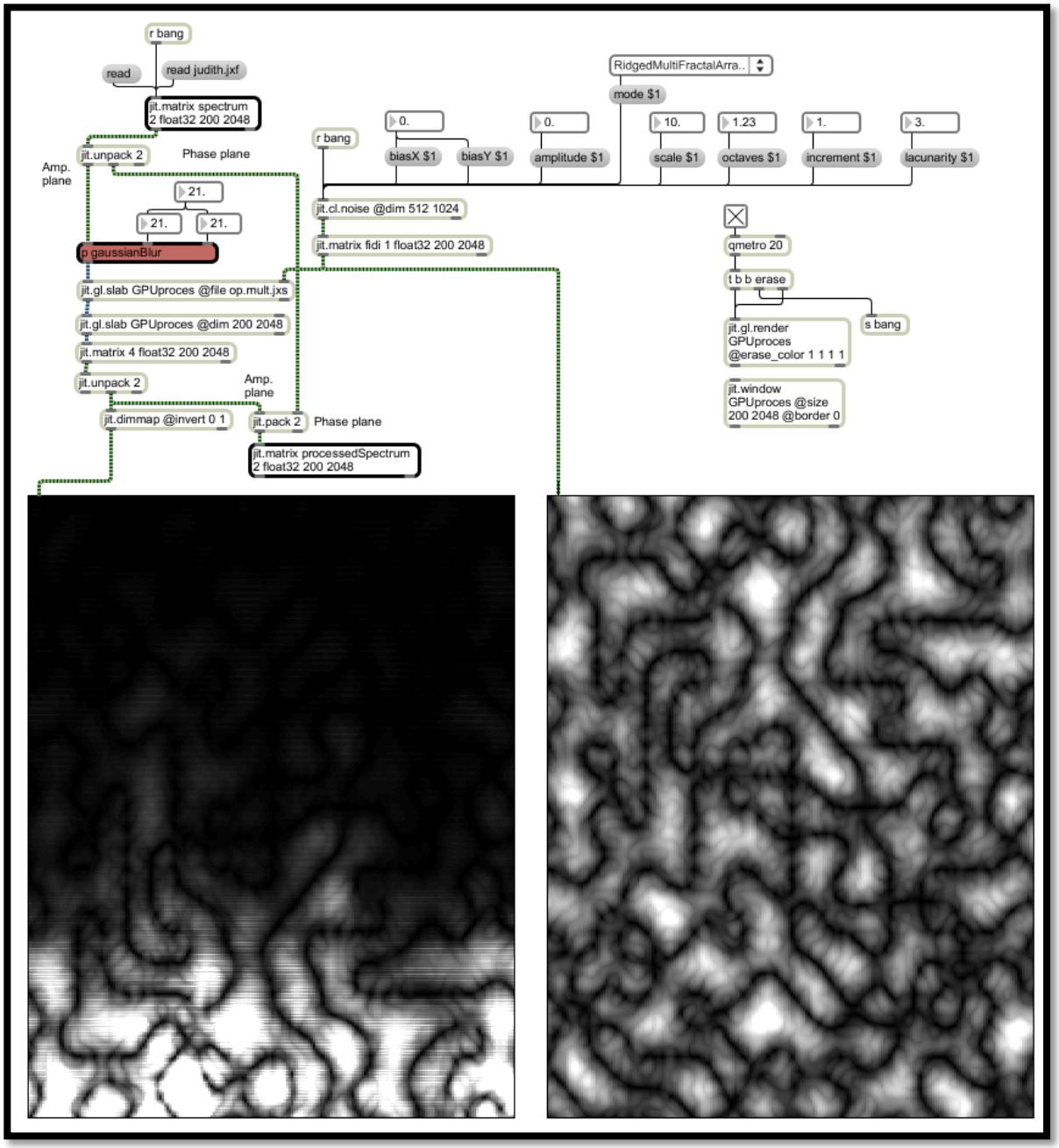

Figure 5.62 Using *jit.gl.slab @file op.mult.jxs* to perform multiplication on GPU between data from CPU and GPU. Left: blurred amplitude plane multiplied by noise fractal; Right: Noise fractal (original).

The fractal on fig. 5.62 was generated with external *jit.cl.noise,* that uses openCL library to calculate noise generated functions on GPU, and outputs a matrix (see green patch cords at the output on fig. 5.62). Similar fractal could be generated by object *jit.bfg* that uses CPU. The reason why external *jit.cl.noise* was used is to demonstrate an additional possibility of using GPU for data processing.

* Many techniques that were presented in this paper could also be done on GPU. There are slab versions of various jitter objects, such as rota, op, repos etc. For instance the slab version of *jit.rota* could be instantiated by typing the reference slab file name into the jit.gl.slab object: *jit.gl.slab renderContextName @file td.rota.jxs.* For other examples see folder *jitter-examples/render/slab-helpers.*

Jumping from CPU to GPU and back, which should be performed only on smaller matrixes when working with 32-bit data, would probably not make sense for only one task – for instance doing a rotation on GPU. Therefore the power of GPU becomes very handy when using longer slab chains, that means doing a larger amount of processing on GPU.

With jit.pix, that is an object from Gen family, we can also make our own slabs. When working with longer slab chains, an openGL version of jit.unpack and jit.pack objects might be very handy. That can be done with jit.pix by using swiz and vec operators (See patch *03_pack-unpack_on_GPU.maxpat).* For other examples on GPU processing see notes in sections 5.1.3 Frame Interpolation and 5.2.3 Lock to Grid.

Chapter 6

Conclusions and Future Work

The research succeeded in categorizing ADSSP techniques of many kinds on the basis of their functionality and underlying data scaling or repositioning methods. Through suggested categorization, a person interested in ADSSP, using Max/MSP and Jitter, can find proposed implementations of either specific spectral effects or their underlying data manipulating techniques, potentially applicable to many not yet explored ADSSP methods. How successful these implementations are is hard to say, since it was impossible to find something to compare them to. The research also succeeded in implementing some probably new spectral effects and also in finding many advantages of real-time ADSSP. One of the beneficial side results of the research is the explanation of Phase Vocoder and FFT, that combines conceptual and mathematical approach.

Max/MSP/Jitter turned out to be very powerful tool for real-time ADSSP, although it has some unnecessary limitations that could be hopefully be Theremoved with future versions of the software. In combination with various externals, the power of Max/MSP and Jitter can be even extended. The option of processing FFT data and spatial maps on graphic cards using openGL is also a very strong attribute of the software.

6.1 Future Work

The majority of presented ADSSP techniques can be combined in various creative ways in order to develop new spectral effects. They could also be integrated into a larger assembly such as phase vocoder based synthesizer. Using modulation matrices in addition to parallel and serial processing could have a synergetic effect on spectral sound synthesis, as is the case in time-domain signal processing.

Processing FFT data on graphic cards is also worth further investigation since their processing power is rising much faster than the power of CPU's. Many ADSSP techniques presented in this paper can be executed on graphic cards, especially because all the mathematical and logical operators of object *jit.op,* are available in openGL. OpenGL objects also offer a sea of new possibilities for ADSSP. Even though ADSSP is all about scaling and repositioning data, openGL in combination with fast graphic card offers a faster performance that should result in higher sound quality - especially when modulating various parameters in different kinds of spectral effect at the same time.

According to uninterrupted audio and video stream during the research, using MacBook Pro with 2.4 Ghz Dual Core Intel processor and NVIDIA GeForce 8600M GT graphic card, we can conclude that ADSSP is suitable for live performance. Visualization of signal processing with spectrogram yields synesthetic emotions and also offers the audience a deeper insight into the complex process of electronic music creation. Beside the emotional, also the rational side of listening to music can be evoked in the audience. The synesthetic effect of seeing sounds and hearing moving images, can be probably even extended by using 3D representation of a spectrogram and also a 3D or stereo projector.

References

Allen, J. B., Rabiner, L. R., 1977. PhaVoRIT: A Unified Approach to Short-time Fourier Analysis and Synthesis. *Proceedings of the IEEE 65*, Dayton, Ohio. pp 1558-1564.

Battier, M.1996. *AudioSculpt – User's Manual* [Online Manual]. 2nd edition, Paris: IRCAM. Available: homepages.gold.ac.uk/ems/pdf/AudioSculpt.pdf [Accessed 15 August 2011].

Bogaards, N. & Röbel, A. 2005. An interface for analysis-driven sound processing [Online Article]. *Audio Engineering Society Convention Paper.* 119 (October), New York, New York USA. Available: articles.ircam.fr/textes/Bogaards05b/index.pdf [Accessed 15 August 2011].

Branble, S. *QPSK Modulation Demystified.* [Online Article]. Available: http://www.simonbramble.co.uk/techarticles/qpsk_modulation/qpsk_modulation_demystified.htm/ [Accessed 25 July 2011].

Burk, P., Polansky, L., Repetto, D., Roberts, M. and Rockmore, D. 2011. Morphing. In: *Music and Computer: A Theoretical and Historical Approach* [Online Book], Columbia University, Available: http://music.columbia.edu/cmc/MusicAndComputers/chapter5/05_06.php [Accessed 15 September 2011].

Brent, W. 2009. *A Timbre Analysis and Classification Toolkit for Pure Data.* [Online Article]. Department of Music and Center for Research in Computing and the Arts University of California, San Diego. Available: http://williambrent.conflations.com/papers/timbreID.pdf [Accessed 3 October 2011].

Brent, W. 2009. Cepstral Analysis Tools for Percussive Timbre Identification. [Online Article]. *Proceedings of the 3rd Pure Data Convention.* Sao Paulo.. Available: http://williambrent.conflations.com/papers/features.pdf [Accessed 3 October 2011].

Charles, J. F. 2008. A Tutorial on Spectral Sound Processing Using Max/MSP and Jitter. *Computer Music Journal* 32(3) pp. 87–102.

Cooley, J. and Tukey, J. 1965. An Algorithm for the Machine Computation of Complex Fourier Series. *Mathematical Computation,* 19 (December). pp. 297-301.

Dobrian, C. 1998. FTM & Co. Tutorial, Gabor - Adding a Matrix Into MSP [Online Tutorial]. *IRCAM* (Institut de Recherche et Coordination Acoustique/Musique). Available: http://ftm.ircam.fr/downloads/tutorials/intro/f05-AudioGrains.html [Accessed 30 July 2011].

Dobrian, C. 1999. Programming New Realtime DSP Possibilities with MSP [Online Article]. *Proceedings of the 2nd COST G-6 Workshop on Digital Audio Effects (DAFx99)*. NTNU, Trondheim. Available: http://music.arts.uci.edu/dobrian/DAFX99-Dobrian.htm [Accessed 5 August 2011].

Dobrian, C. 2001. *Max 5 Help and Documentation* [Online Help Documentation]. Cycling ’74. Available: http://www.cycling74.com/docs/max5/vignettes/intro/docintro.html [Accessed 13 July 2011].

Dobson, R. 1993. *The Operation of the Phase Vocoder – A non-mathematical introduction to the Fast Fourier Transform.* [Online Article]. Composers Desktop Project. Available: http://people.bath.ac.uk/masjpf/CDP/operpvoc.htm [Accessed 2 August 2011].

Dodge, C. and Jerse, T.A. 1997. *Computer Music: Synthesis, Composition and Performance.* 2nd edition, New York: Thomson Learning

Dolson. M.1986 The Phase Vocoder: A Tutorial. *Computer Music Journal*, 10(4) pp. 14–27.

Dudas, R., Lippe, C. 2006. *The Phase Vocoder - Part I* [Online Article]. Available: http://cycling74.com/2006/11/02/the-phase-vocoder-%E2%80%93-part-i/ [Accessed 3 August 2010].

Eckel, G. 1990. A Signal Editor for the IRCAM Musical Workstation. *Proceedings of the International Computer Music Conference*, Glasgow. pp. 69–71. [Online Article]. Available: http://iem.at/~eckel/publications/eckel90.html [Accessed 16 August 2011].

Flanagan, J. L., Golden, R. M., 1966. Phase Vocoder. *Bell System Technical Journal* [Online Article]. 45 pp. 1493-1509. Available: www.ee.columbia.edu/~dpwe/e6820/papers/FlanG66.pdf [Accessed 4 August 2010].

Fourier, J. 1822. *Théorie analytique de la chaleur*. Paris: Firmin Didot Père et Fils

Gabor, D. 1946. Theory of communication. [Online Article]. *Journal of the Institute of Electrical Engineers* 93(3) pp. 429–457. Available: bigwww.epfl.ch/chaudhury/gabor.pdf [Accessed 15 August 2011].

Gabor, D. 1952. Lectures on communication theory. [Online Technical Report]. *Technical Report 238.* Research Laboratory of Electronics. Cambridge, Massachusetts, Massachusetts Institute of Technology. Available: http://dspace.mit.edu/bitstream/handle/1721.1/4830/RLE-TR-238-14266376.pdf?sequence=1 [Accessed 15 August 2011].

GRM. GRM Tools ST user's guide. . [Online]. Available: http://www.google.com/url?sa=t&source=web&cd=5&ved=0CDwQFjAE&url=http%3A%2F%2Fwww.dtic.upf.edu%2F~lfabig%2Fresources%2Fmaterials%2Fsession8-spectral_processing2%2FGRMToolsSTVSTXPDoc16Eng.pdf&rct=j&q=grm%20tools%20documentation&ei=bxZiTvyRCJPO4QSLgt3MCg&usg=AFQjCNGaGPv4tmdDJlsHd7nun194_A04ig&sig2=SvfCdnGMweQ7aG4ujhUXOQ&cad=rja [Accessed 30 August 2011].

Heideman, M. T., Johnson D. H. and Burrus, C. S. 1984. Gauss and the History of the Fast Fourier Transform, *IEEE ASSP Magazine*, 1, (4). pp. 14-21.

Jones, R. and Nevile, B. 2005. Creating Visual Music in Jitter: Approaches and Techniques. *Computer Music Journal* 29(4) pp. 55–70.

Kahane, J.P. & Lemarie-Rieusset, P.G. 1995. *Fourier Series and Wavelets*, New York: Routledge [Online Book]. Available: http://mathdoc.emath.fr/PMO/PDF/K_KAHANE-70.pdf [Accessed 5 July 2011].

Kim-Boyle, D. 2004. Spectral Delays with Frequency Domain Processing [Online Article]. *Proceedings of the 7th Int. Conference on Digital Audio Effects (DAFx04)*. Italy, Naples. Available: *dafx04.na.infn.it/WebProc/Proc/P_042.pdf* [Accessed 20 August 2011].

Laroche, J., Dolson, M. 1997. Phase Vocoder: About this Phasiness [Online Article]. *Proceedings IEEE Workshop on Applications of Signal Processing to Audio and Acoustics*, New Paltz, New York. Available: www.ee.columbia.edu/~dpwe/papers/LaroD97-phasiness.pdf [Accessed 28 July 2011].

Laroche, J., Dolson, M. 1999. New Phase-Vocoder Techniques for Pitch-Shifting, Harmonizing and Other Exotic Effects. *Proceedings IEEE Workshop on Applications of Signal Processing to Audio and Acoustics*, New Paltz, New York. pp. 91–94.

Lyon, E. 2006. *A Sample Accurate Triggering System for Pd and Max/MSP*. [Online Article]. Queen's University Belfast, Sonic Arts Research Centre, School of Music and Sonic Arts. Available: http://www.sarc.qub.ac.uk/~elyon/LyonPapers/SampleAccurate-Lyon-ICMC2006.pdf [Accessed 8 August 2011].

Mathcentre. 2004 $a\cos x + b\sin x = R\cos(x-\alpha)$ [Online Article]. Universities of Loughborough, Leeds and Coventry. Available: http://www.mathcentre.ac.uk/resources/uploaded/mc-ty-rcostheta-alpha-2009-1.pdf [Accessed 13 July 2011].

Miranda, E. R. 2002. *Computer Sound Design: Synthesis Techniques and Programming.* 2nd edition, Oxford: Focal Press

Moorer, J. A. 1978. The Use of the Phase Vocoder in Computer Music Applications. *Journal of the Audio*. [Online Article], 26 (1/2) pp. 42-45 Available: www.jamminpower.com/PDF/Phase%20Vocoder.pdf [Accessed 4 August 2011].

Portnoff, M. 1976. Implementation of the Digital Phase Vocoder Using the Fast Fourier Transform. *IEEE Transactions on Acoustics, Speech, and Signal Processing* [Online Article], 24(3) pp 243-248. Available: *www.ee.columbia.edu/~dpwe/papers/Portnoff76-pvoc.pdf* [Accessed 4 August 2011].

Puckette, M. 1995. Phase-Locked Vocoder [Online Article]. Proceedings IEEE Workshop on Applications of Signal Processing to Audio and Acoustics, New Paltz, New York. Available:

http://www-crca.ucsd.edu/~msp/Publications/mohonk95.ps [Accessed 23 June 2011].

Redmon, N. 2002. *A gentle Introduction of FFT.* [Online Article]. Available: http://www.earlevel.com/main/2002/08/31/a-gentle-introduction-to-the-fft/ [Accessed 24 July 2011].

Rehberg, P. 2010. Fenn O'Berg: In Stereo/ Nice Music / Yes/ No. *Tokafi* [Online Article]. Available: http://www.tokafi.com/news/fenn-oberg-stereo-nice-music-yes-no/ [Accessed 25 August 2011].

Reif, J. H. 2004. *Mechanical Computation: Its Computational Complexity and Technologies* [Online Article]. Duke University, Department of Computer Science, Durham, NC. Available: http://csis.pace.edu/~marchese/CS396x/Computing/MechComp.pdf [Accessed 6 July 2011].

Roads, C. 1996. *The Computer Music Tutorial.* Cambridge, Massachusetts: The MIT Press

Roads, C. 2001. *Microsound*. Cambridge, Massachusetts: The MIT Press.

Sakodna, N. 2000. *MSP Granular Synthesis Patch v2.5* [Online Max/MSP Patch]. Available: http://web.mac.com/nsakonda/sakoweb/download.html [Accessed 25 August 2011].

Schwarz, D. 2006. Concatenative Sound Synthesis: The Early Years. Journal of New Music Research 35(1) pp. 3–22.

Sedes, A., Courribet, B., Thiebaut, J. B. 2004. Visualization of Sound as a Control Interface. Proceedings of the 7th International Conference on Digital Audio Effects. Naples, Italy, pp. 390–394.

Smith, J. O. 2007. DFT Definition. In: *Mathematics if the Discrete Fourier Transform (DFT)*, W3K Publishing [Online Book]. Available: https://ccrma.stanford.edu/~jos/mdft/DFT_Definition.html [Accessed 27 July 2011].

Smith, J. O. 2008. *Spectral Audio Signal Processing*, Stanford University [Online Book]. Available: https://ccrma.stanford.edu/~jos/sasp/sasp.html [Accessed 3 August 2011].

Smith, J. O. 2010. Dudley's Vocoder. In: *Physical Audio Signal Processing*, W3K Publishing [Online Book]. Available: https://ccrma.stanford.edu/~jos/pasp/Dudley_s_Vocoder.html [Accessed 7 July 2011].

Sprenger, S. M. 1999. Pitch Scaling Using The Fourier Transform. *Audio DSP pages*. [Online Book]. Available: http://docs.happycoders.org/unsorted/computer_science/digital_signal_processing/dspdimension.pdf [Accessed 5 August 2010].

Thomson, W. 1878. Harmonic Analyser [Online Article]. *Proceedings of the Royal Society of London*, London, England. 27 (May). pp. 371-373 Available:

http://zapatopi.net/kelvin/papers/harmonic_analyzer.html [Accessed 6 July 2011].

Warburton, D. 2009. Dan Warburton on the sound sorcerors of EAI. *The Wire* [Online Article]. Available: http://www.thewire.co.uk/articles/2165/print [Accessed 25 August 2011].

Ward, R. L. 1994. *Imaginary Exponents and Euler's Equation* [Online Article]. Drexel University at Philadelphia. Available: http://mathforum.org/dr.math/faq/faq.euler.equation.html [Accessed 14 July 2011].

Weeks, M. 2007. Digital Signal Processing Using MATLAB and Wavelets. Hingham, Massachusetts: Infinity Science Press.

Wenger, E., and Spiegel, E. 2005. *MetaSynth 4.0 User Guide and Reference*. San Francisco, California: U&I Software.

Wishart, T. 1996 *On Sonic Art. A New and Revisited Edition*, London: Routledge