The Fast Fourier Transform and Spectral Manipulation in Max/MSP and Jitter

Jack Walters

Creative Practice Music Project 2019

Abstract

This document is a write up for the 2019 Creative Practice Music Project at the University of Edinburgh by Jack Walters. The project this document accompanies explores the mathematics of the Fast Fourier Transform (FFT), their application in the programming environment Max and the subsequent spectral effects that are possible in Max's openGL extension Jitter. This document is both a reference for the Max code, as well as a commentary on the project aims, research, my reflection on the project and its place in the wider programming community.

The project page can be viewed here: https://cycling74.com/projects/the-fast-fourier-transform-and-spectral-manipulation-in-maxmsp-and-jitter-beta.

A YouTube video demonstrating sections 1.2.1 – 1.2.2, and 1.4.1 – 1.6.3 can be watched here: https://www.youtube.com/watch?v=U3hjo-Jnmms&feature=youtu.be.

The GitHub repository can be viewed here: https://github.com/dwambyman/The-Fast-Fourier-Transform-and-Spectral-Manipulation-in-Max-MSP-and-Jitter.

For those interested in learning more about the process of building this project, the developmental blog can be found here: https://digital.eca.ed.ac.uk/spectral-manipulation/.

Table of Contents

Navigating the Project

The folder 'Max' in this project contains all the code I have written that explores the FFT in Max, with the files marked with a blue tag in folders 1 – 1.6.3. being intended for use by the user. This numbering system in the 'Max' folder denotes both a grouping and chronology to the project, and for the user to experience the totality of the project's functionality I recommend it is explored in order. In each subfolder in the 'Max' folder are videos demonstrating that subfolder's function, and it's my hope that users will be able to use the software and accompanying video to provide context to both the project annotation found in this write up, and the project's developmental blog. This document is structured according to the layout of the project, and sections '1.1 main' through to 'playback' correspond to their own subfolder in the Max project.

Sections 'Introduction', 'Research' and 'Aim of the Project' in this document provide an overview of the conception of this project and what I planned to achieve when starting out. In the following section 'The Fourier Transform' I provide an explanation of the principles of the FFT in abstract, and sections '1.1 main' through to 'playback' are all annotations of code that explore and develop these principles in the Max environment. Sections 'User Experience', 'Reflection' and 'The Project in the Community' explore my review of the project after having designed the software, the methods through which it will interact with the community and my intentions for the project's development in the future.

Introduction

Since learning about the mechanics of the of the FFT in June of 2018, I have become fascinated with its application in a musical context. Between September and December of 2018 I had the opportunity to study the mathematics that underpins the FFT on the university module 'The Musical Applications of Fourier Theory and Digital Signal Processing' (MAFTDSP). Having studied digital signal processing and sound design in object orientated programming environments like Pure Data and SuperCollider for two years prior, I was able to claim a reasonable level of knowledge in relation to signal processing in higher-level programming. However, I wanted to explore processes that are encapsulated fairly neatly in software like Pure Data and SuperCollider in more granular detail. In MAFTDSP we used the programming environment MATLAB to emulate the FFT and digital filters: this experience was very impactful on me and my intended direction of where I wanted to take my studies. In MATLAB the programmer has a considerable amount of control over all aspects of audio processes that are encapsulated in single objects in Pure Data and SuperCollider, meaning you have to comprehensively understand the principles that underpin the audio processes before you can implement them.

At around the same time I took a module called 'Sonic Structures' that was taught in the programming environment Max/MSP. While I was sceptical to adopt another new language at first, especially since I had spent the best part of two years programming in Pure Data, I quickly came to realise the power that Max encapsulated. At first, my new-found interests in MATLAB and Max didn't seem to hold a lot in common: there was a clear dichotomy in relation to their application in acoustics and the digital arts respectively. The modules that taught the two programming languages were run by separate departments in the music department at the University of Edinburgh and in the lectures of one module, there was little to no mention of topics covered by the other. This was understandable, as they were approaching digital signal processing from different directions and didn't have mutual intended learning outcomes. Still, I found myself wondering as to where the overlap between the disciplines of acoustics and object-orientated programming could be found.

Contextual Research

Having been introduced to the FFT on the MAFTDSP module, I was able to consolidate my understanding of the principles that underpin the process in James McClellan, Ronald Schafer and Ronald Williams' *Signal Processing First*. While being written primarily for MATLAB users, *Signal Processing First* was extremely influential on my project as it provided a complete analysis of the FFT and its various potential applications in signal processing. Through studying the principles both in abstract and applied in MATLAB, I was able to begin the research of the FFT applied in Max with complete understanding of all aspects of the process. Chapter 13 'Computing the Spectrum' in *Signal Processing First* was where I learnt about the Discrete Short-Time Fourier Transform (DSTFT): a variant on the FFT which enables the user to view the frequency representation of an infinitely long sample, rather than just a shorter one that would generally last less than a 1/10th of a second.[1] The DSTFT is the transform on which this whole project is based, and learning about the principles that underpin it inspired me to explore the topic further in Max. Being relatively new to Max upon instigation of this project, I read through relevant chapters in both Volumes 1 and 2 of Alessandro Cipriani and Maurizio Giri's *Electronic Music and Sound Design: Theory and Practice with Max/MSP*. In these books Cipriani and Giri explain how to conduct various signal processing procedures in Max, and from this I was able to garner a stronger general understanding of how Max handles data and signal flow. In particular, it was Section IB.6 on iterative structures in Volume 1[2] and Chapter 7T on dynamics processing in Volume 2[3] that were influential on how I implemented data handing structures and understood the effect of EQ spectral effects on the spectrogram.

Despite being written in C++, both ICRAM's 'Audio Sculpt'[4] and Michael Klingbeil's 'SPEAR' provided me with inspiration as to how I could organise the interface of my project. In particular it was Klingbeil's incorporation of direct interaction with the spectrogram and functionality for the selection of regions in SPEAR that influenced the masking component in section 1.3 of this project.[5]

There are three Max projects I have discovered, whose research has become the bedrock of my work. First and foremost is Jean François-Charles' article 'A Tutorial on Spectral Sound Processing Using Max/MSP and Jitter', the contents of which and their accompanying patches have provided not only myself but a plethora of other Max programmers with the practical knowledge for handling DSTFT data inside the

[1] James McClellan *et al.*, *Signal Processing First* (Upper Saddle River: Pearson Education International, 2003), 389-424.
[2] Alessandro Cipriani and Maurizio Giri, *Electronic music and sound design: theory and practice with Max-MSP* (Rome: ConTempoNet, 2010), 462-466.
[3] Alessandro Cipriani and Maurizio Giri, *Electronic music and sound design: theory and practice with Max-MSP, Volume 2* (Rome: ConTempoNet, 2014), 308-351.
[4] 'AudioSculpt', accessed 19th April 2019, http://anasynth.ircam.fr/home/english/software/audiosculpt.
[5] 'SPEAR', accessed 19th April 2019, http://klingbeil.com/spear/.

Max environment.[6] In terms of collating that data in the first place, François-Charles references the patch 'jitter_pvoc_2D' of Luke Dubois which is located in the example directory in Max 8.[7] Dubois' work provides the framework through which I was able to construct the code in section '1.1 main' of this project. The most influential work on my own was that of programmer Tadej Droljc, whose comprehensive project 'STFT Analysis Driven Sonographic Sound Processing in Real-Time using Max/MSP and Jitter' was the central inspiration behind my decision to create this project.[8] Droljc also created a YouTube video that showcased the extent of the manipulation the programmer can exercise with DSTFT data in Max.[9] Watching this video proved to be the instance in which I began to understand how deeply one can explore the DSTFT in a Max environment. It has been through Droljc's work that I have been able to formulate the kind of project I have built.

[6] Jean François-Charles, 'A Tutorial on Spectral Sound Processing Using Max/MSP and Jitter', *Computer Music Journal* 32 (3), accessed 26th October 2018, https://doi.org/10.1162/comj.2008.32.3.87.

[7] Luke Dubois, 'jitter_pvoc_2d', directory: /Cycling '74/Max 8/Examples/jitter-examples/audio/jitter_pvoc.

[8] Tadej Droljc, 'STFT Analysis Driven Sonographic Sound Processing in Real-Time using Max/MSP and Jitter' (BSc diss., University of Hull, 2011).

[9] 'Sonographic Sound Processing in Max/MSP and Jitter', last modified 17th April 2013, https://www.youtube.com/watch?v=0PrIO5tweeo.

Aim of the Project

My goal for the project was to create a Max environment in which the user can explore the sonic potential of the FFT spectra manipulation. It was also important that the project be structured in a modular fashion as not only would this forefront what was most important to me, the FFT data processing, but this structure would also lend itself well to subsequent explanation. Using Max's abstraction feature (where a single patch can be used in many iterations across the project) is something that hasn't been utilised as thoroughly as it should have been in other projects I have investigated. Doing so not only eases the computation load of the software, but also enables the user to understand the commonalities that a lot of the sections of the project share with one another. This approach is also sensical when it comes to thinking about the project's longevity beyond this submission, as I will easily be able to build on the structure in the future. The project's modular structure also means other programmers will able to extract useful parts from it: after discovering a spectral effect in this project that would be useful in their own, the user will then be able to easily take and re-use that section of code. Through studying the Max software and text help files, this write-up and the accompanying blog, fellow programmers should be able to lift sections from my project in full understanding of how it works both in isolation and in context.

Through studying Jean François-Charles, Tadej Droljc and Luke Dubois' work, I came to realise Max's potential for fulfilling my aims of creating FFT-driven analysis in an interactive environment. As I began to learn about how many spectral effects are possible in the context of Max, the clearer it became how much I would need to create a structure in which data and computational load is shared as evenly as possible. In order to showcase each effect to the extent that I wanted and be able to contextualise its explanation in the broader project and the FFT, I knew that each effect would have to be placed into a strong and flexible framework that could support it. Both François-Charles and Droljc also mentioned how many of the Jitter operations in their projects are able to be conducted in GPU-based SLAB processes. The benefit of doing so is that more of the processing load is shared between the CPU and GPU, increasing the speed of the project overall. I was aware of how taxing some of the processes I planned to build would be on the CPU, so implementing SLAB processes at every possible juncture was essential.

The more I began to envision my project in its finished state, the less I could see of it being a performance piece and the more I imagined it being a tool that could prove useful to those in the Max community. I hope that the project will be able to function as an educational tool for both those who want to learn more about the DSTFT and the flexibility in Jitter in the context of Max. Creating a piece of software that is accessible to beginner Max users was also something that I was not interested in. The FFT is not an easy concept to grasp, and to explore it to the level that I planned would require the user to be familiar with signal processing concepts such as waveforms and sampling rate, as well as general programming structures such as matrices and data types.

The Fourier Transform

There are countless formulae which exhibit the Fourier Transform in different contexts: some are theoretical and impossible to perform on a computer and serve the purpose of demonstrating the mechanics of the transform, while others are not capable of performing the FFT on a sample longer than 1 frame (around 1/10th of a second). It was not until the 9th week of lecture notes in the module MAFTDSP that I found the formula for the DSTFT, shown in Figure FT.1.

$$X_m[k] = \sum_{n=0}^{N-1} X[mH_a + n]w[n]e^{j\frac{2\pi kn}{N}}$$

Figure FT.1 – the equation outlining the DSTFT

While certainly intimidating to look at first, what this formula outlines is a process where a signal is broken down into a series of shorter horizontal segments called 'frames', where each frame is then analysed according to a certain number of vertical frequency 'bins'. The bottom 5 waveforms in Figure FT.2 represent 5 frames of the waveform in the top row:

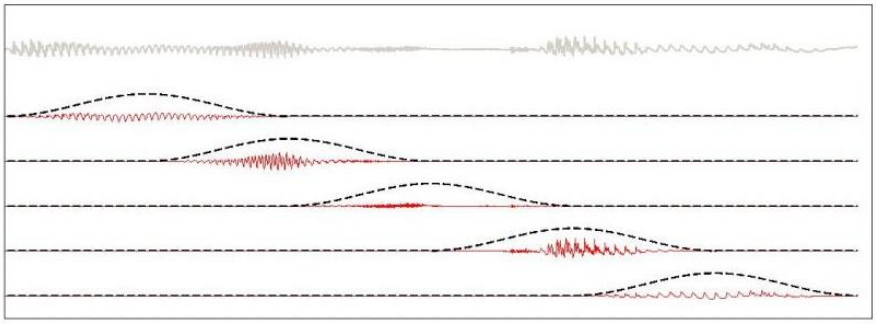

Figure FT.2 – a waveform split into 5 frames

Each bin represents a complex exponential analysis signal 'spinning' at a certain frequency and is represented by its index *k*. The Fourier analysis rests on the idea that a signal can be represented as the summation of a set of N of these complex exponentials, which are a combination of 'real' cosine waves and imaginary' sine waves. In my project, the spectral frame size N is 1024, meaning there are 1023 bins in each frame (as we analyse from bin index $k = 0...N - 1$). Say we take a look at bin index $k = 5$, we can calculate what frequency this bin is analysing through the equation in Figure FT.3 (S_r = sample rate):

$$f_k = \frac{kS_r}{N}$$

Figure FT.3 – Equation to calculate central frequency of bin index *k*

Thus, in my project bin index $k = 5$ contains a complex exponential spinning at 215.33Hz. By taking the inner product in this bin, what we are effectively doing is multiplying the input signal with a cosine and sine wave oscillating at 215.33Hz, and if there is a clustering of energy around 215.33Hz in the input signal, a large value for $X_m[k]$ is found. This scenario is demonstrated in Figure FT.4; it is worth bearing in mind that the graph is in the frequency domain while Figure FT.2 is in the time domain.

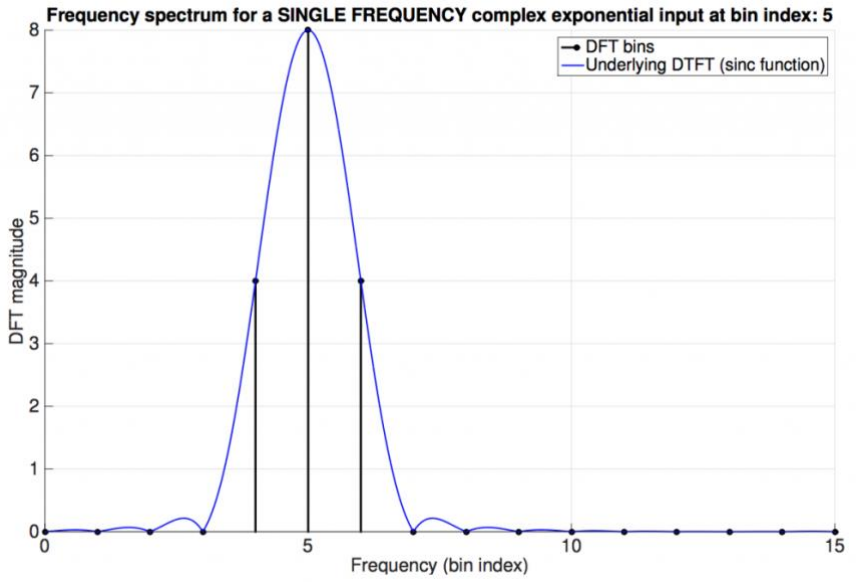

Figure FT.4 – bin index $k = 5$ exhibiting a peak

So far we have discussed attaining the amplitude information of the signal, but there is one other aspect of the signal that is essential for signal reconstruction later on in the process: the phase. Phase refers to the point of oscillation that the signal is in when it is multiplied with the complex exponential in frequency bin $k = 5$. Phase is a relative term (from 0 to 2π), so it is the difference in phase between successive frames that becomes important. Consider the example where we considered an input signal in relation to bin index $k = 5$ (215.33Hz); say our input signal actually has a clustering of energy around 225Hz, you would think that this clustering would then be picked up by bin index $k = 6$? Bin index $k = 6$ contains a complex exponential spinning at 258.4Hz, which is slightly awkward because the clustering in our input signal is now located between two frequency bins. In this instance, we can calculate the difference in phase between the clustering in our input signal and bins 5 and 6. Calculating the difference in phase value between successive frames in the input signal, allows us to calculate the frequency deviation of our input signal in relation to a specific frequency bin. The difference in phase between frames is calculated through the equation in Figure FT.5, where m represents the frame number:

$$\Delta\phi_{m,k} = \phi_{m,k} - \phi_{m-1,k}$$

Figure FT.5 – equation for calculating phase difference between frames m and m-1 in bin index k

The phase value between successive frames is what's known as the 'running phase', and is the summation of phase differences between the first frame $m = 0$ and the current frame M at bin index k:

$$\phi_R = \sum_{m=0}^{m=M} \Delta\phi_{m,k}$$

Figure FT.6 – equation for calculating running phase at bin index k

The equations in Figures FT.5 and FT.6 allow us to calculate the frequency deviation of the clustering of energy in our input signal in relation to a bin's central frequency, meaning that every frequency in the input audio can be reconstructed regardless of its distance from its neighbouring bin's central frequencies. A variant on the DSTFT, the equation in Figure FT.7 calculates the true frequency f_t of bin index k through a summation of difference in phase information across N samples in window number m, which is then multiplied by the central frequency of bin index k over 2π.

$$f_t = f_k + \Delta\emptyset \times \frac{S_r}{2\pi \times N}$$

Figure FT.7 – equation for calculating frequency deviation at bin index k

Project Annotation

From this point onwards, unless specified I will be referring to the patches in the project in 'Patching Mode', rather than 'Presentation Mode' which they will open in.

1.1 main

Through a combination of studying Luke Dubois' jitter_pvoc_2D[10] and Cycling 74's article on implementing phase vocoders in Max, I have constructed the 'matrix_fill' patch in the _main folder.[11] One important note in the article is on the difference between the 'fft~' and 'fftin~' objects. Perhaps the most noticeable difference between the objects is that 'fftin~' has to be used in a 'pfft~' abstraction (which encapsulates the whole process). The 'pfft~' abstraction manages frame overlap and provides the user with 5 in-built window choices (square, hanning, triangle, hamming and blackman). Figure 1.1.1 shows Cycling 74's demonstration of the older 'fft~' object in use.

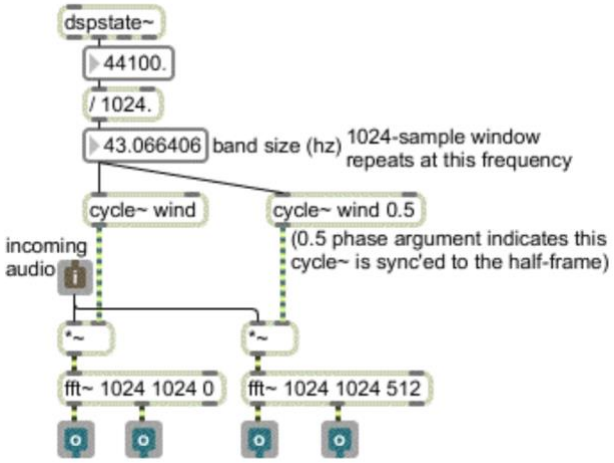

Figure 1.1.1 – Performing the DSTFT through the traditional 'fft~' object

For efficiency's sake, in my project inside the 'matrix_fill' abstraction the 'fftin~' performs a real FFT, which is faster than the complex FFT performed by the 'fft~' in Figure 1.1.1. In the real FFT, the imaginary aspect of the Cartesian coordinates is set to 0, and the real aspect is used as both the real and imaginary parts of a complex FFT half the size of the real FFT in 'fftin~'. The mathematics of this are beyond the purview of the project and not critical to understanding the FFT in Max aside from one crucial conversion factor. The resultant size of the complex FFT being half of that in the real FFT may at first seem like an issue, but it is important to consider the principle of the Nyquist frequency in relation to the data.

[10] Luke Dubois, 'jitter_pvoc_2d', directory: /Cycling '74/Max 8/Examples/jitter-examples/audio/jitter_pvoc

[11] 'Tutorial 26: Frequency Domain Signal Processing with *pfft~*', accessed 30th October 2018, https://docs.cycling74.com/max5/tutorials/msp-tut/mspchapter26.html.

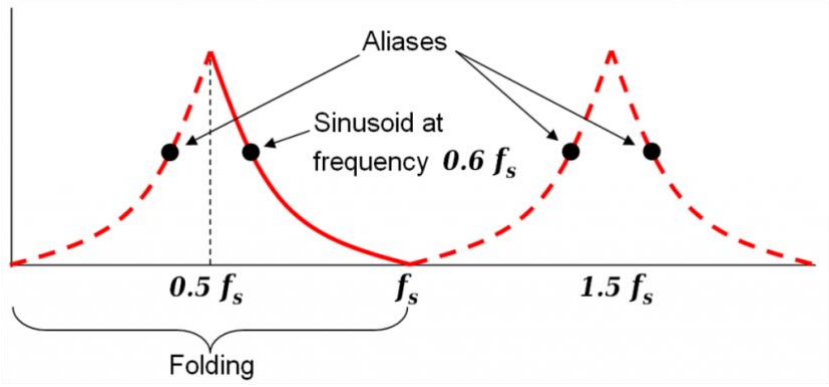

Figure 1.1.2 – Nyquist folding above 0.5fs

In a signal processing environment, analysing any frequency f above half of the sample rate ($0.5S_r$) results in that frequency folding back about $0.5S_r$ and being sampled at frequency $0.5S_r$ - (f - $0.5S_r$). This is known as aliasing and is emblematic of under-sampling: not only is it a waste of computational power but it also produces undesirable artefacts in the reconstruction of the signal. This means the fact that the complex FFT is only half the size of our real FFT performed inside the 'fftin~' object is not an issue. All we need to do is set the FFT size (referred to as the 'frame size' in Max) on the 'pfft~' object in '_main' to be twice as big as our desired FFT size (referred to as the 'spectral frame size' in Max): in my case I set the 'pfft~' FFT size to 2048 to acquire 1023 FFT bins.

At this point I came to the first crossroads of the project. Cycling 74's aforementioned article outlines 2 methods for storing amplitude information from outlet 1 of 'cartopol~', and phase information from 'phasewrap~'. The first method was storing the information in a 'buffer~', which I came to realise would be impractical for 2 reasons: firstly, the 'buffer~' object does not allow for processing of all its contents simultaneously which means all the Jitter-based spectral processing I hope to implement down the line would not be possible. Secondly the 'buffer~' memory would have to be allocated according to frame-size and overlap factor of the FFT and even then it would not be exact enough for my spectral effects in the project.

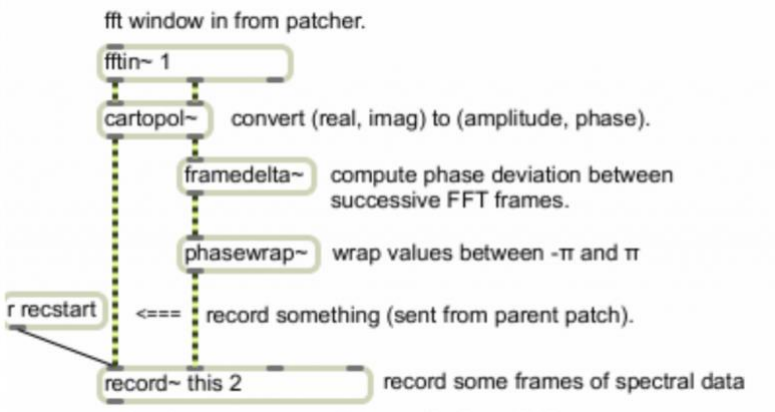

Figure 1.1.3 – Recording FFT data into a buffer~

The other method outlined by Cycling 74, and subsequently the method adopted by all research I have mentioned thus far including my own, is storing the amplitude and phase information from the DSTFT in a 'jit.matrix'. The 'jit' prefix on 'jit.matrix' is short for Jitter; Jitter is an extension to Max that allows for storage and manipulation of matrices of data. Arguably the most important object in my whole project, the 'jit.matrix' hosts a matrix of numbers that just like the 'buffer~' object, can be edited and retrieved from any instance of the object from around the project.

The patch in Figure 1.1.4 'matrix_fill' contains the object 'framedelta~' which encapsulates the equations outlined in the 'The Fourier Transform' section of this paper in Figures FT.5 and FT.6.

$$\phi_R = \sum_{m=0}^{m=M} \Delta\phi_{m,k}$$

Figure FT.6 – equation for calculating running phase at bin index k

$$\Delta\phi_{m,k} = \phi_{m,k} - \phi_{m-1,k}$$

Figure FT.5 – equation for calculating phase difference between frames m and m-1 in bin index k

Phase difference in these equations is calculated between 0 and 2π but as we want to know if the frequency deviation is positive or negative, the phase difference needs to be wrapped between -π and π, which is done in the 'phasewrap~' object pictured above. For 'framedelta~' and 'phasewrap~' to operate, they must receive signal in Polar coordinate form. The object 'cartopol~' is necessary because in its original form, Fourier analysis of a signal outputs data in Cartesian coordinates, i.e. as a coordinate in relation to real (cosine) and imaginary (sin) axes on the complex plane. By its very definition, the imaginary aspect of this measurement is theoretical and incomputable so 'cartopol~' converts the Cartesian coordinates into amplitude and phase information and outputs them through outlets 1 and 2 respectively.

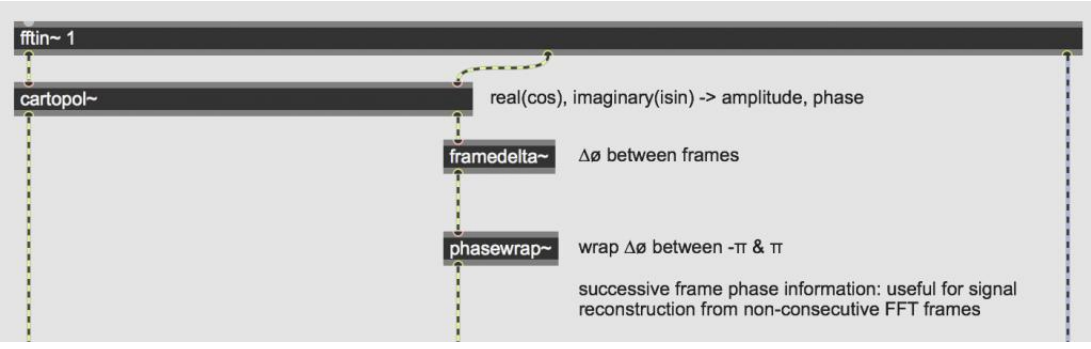

Figure 1.1.4 – The DSTFT, Cartesian to Polar coordinate conversion, and running phase calculations in Max

Figure 1.1.6 exhibits the right-hand side of matrix_fill, which performs two functions. First and foremost it provides horizontal sync for the 'jit.poke~' objects in Figure 1.1.5. To do so, the 'buffer~' in '_main' containing the audio sample is analysed by 'info~' at (1), where we can convert the number of samples in the buffer into the number of frames by dividing it by the hop size. The number of frames at (2) is then packed with the spectral frame size at (3) and set at the dimensions of the 'jit.matrix' 'DSTFTrec' where we will store our DSTFT data. The 'count~' object at (4) has its target value set by the sample count from (1) and is triggered by a message via the 'playthrough_start' send object in the '_main' patch window. The signal output from 'count~' is divided by the spectral frame size and dictates the 'jit.poke~' writing speed in terms of frames. It is worth mentioning that the amplitude information output from the left outlet of 'cartopol~' and the phase information output from 'phasewrap~' in Figure 1.1.4 are being fed through the 'selector~' object in Figure 1.1.5 (which prevents data being passed into the 'jit.matrix' when the sample is not playing in the '_main' window), and then into planes 0 and 1 of the 'jit.matrix DSTFTrec' respectively.

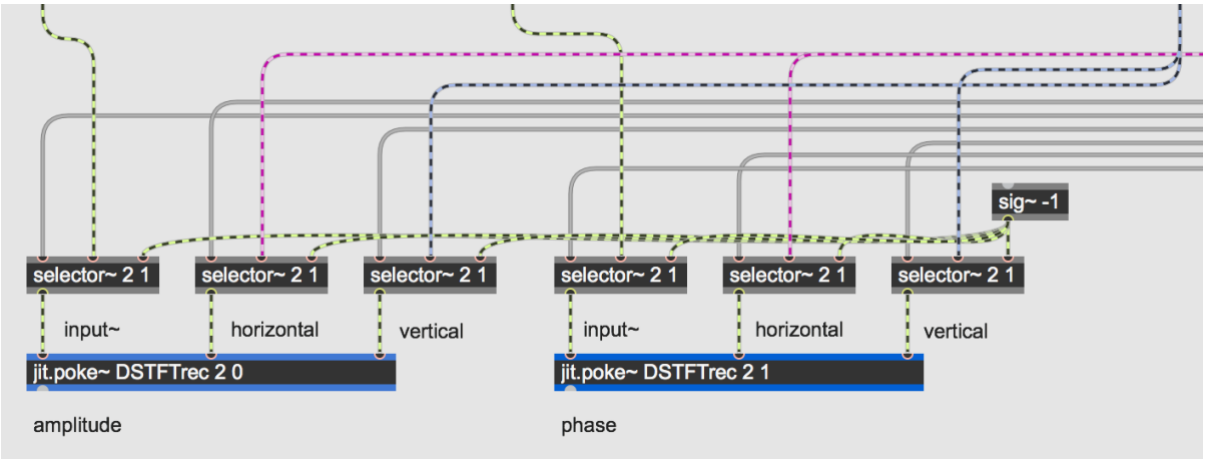

Figure 1.1.5 – 2 jit.poke~ objects filling jit.matrix 'DSTFTrec' from Figure 1.1.5

Shown in Figure 1.1.6, the number of frames information at (2) is also fed into the 'sprintf' object at (6), along with the frame size, spectral frame size, hops size and the name of the audio file (sent from _main to the file_name receive object at both instances of (7). What this does, is create a text file containing all the relevant information about the audio sample called 'fft_info.txt' in the 1.1 main folder, and also saves the contents of the 'jit.matrix DSTFTrec' as a .jit file called 'DSTFT_matrix.jxf.jit' at (6). Due to the abstracted nature of my project I wanted the 5 values stored in 'fft_info.txt' to be readable from any abstraction inside the project, as they are fundamental to the operation of nearly every patch. In the same vein, the .jit file that is saved when the audio sample has finished its play through is readable from any spectral process in the project (provided the user has added folder 1.1 main to the Max file directory). This saves on CPU power as once the .jit file has been created, all the spectral processes in the project can run independently of the DSTFT process in 1.1 main as they are simply reading the matrix file. This does come at the price of being able to use a microphone or any sort of 'live' audio input into the system, but since the spectral processes in this project all allow the user to change playback speed, with some spectral effects being more noticeable at unnatural playback speeds like 0.5 or -1.0, live input is not essential by any means.

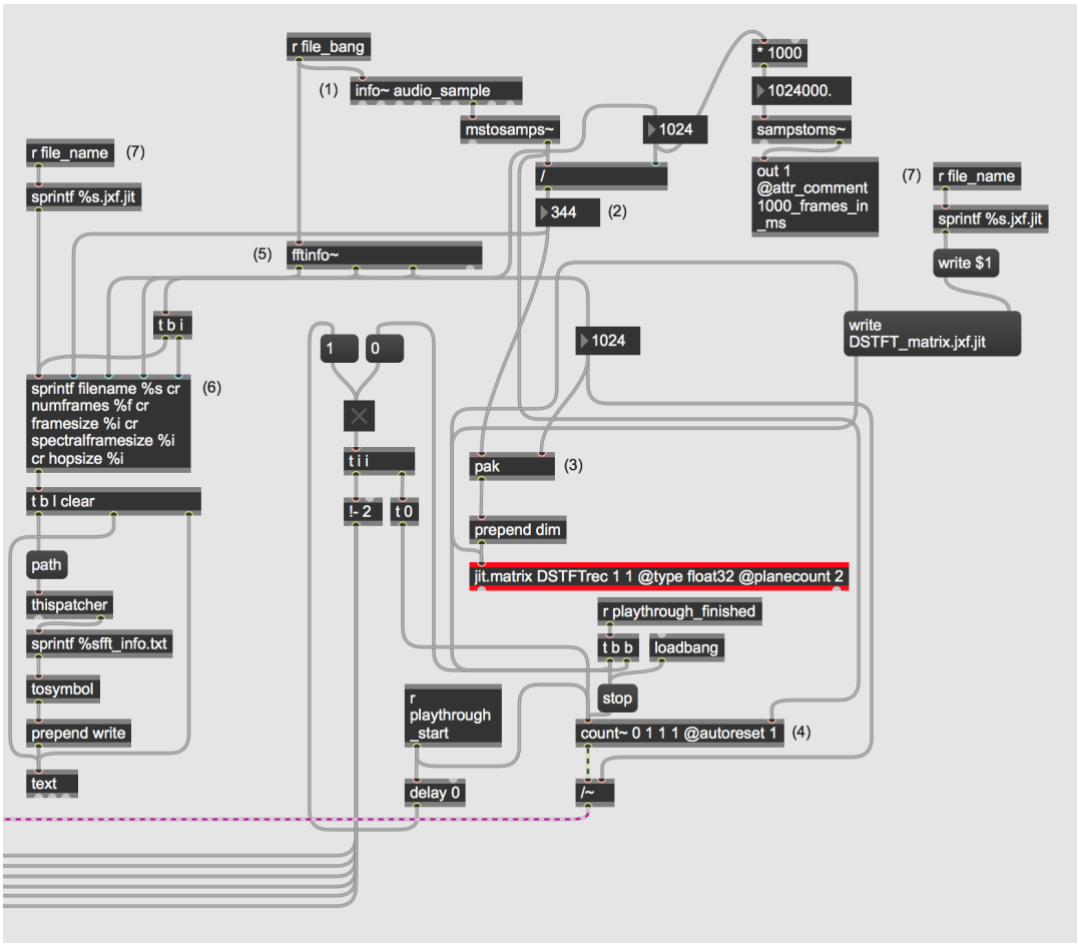

Figure 1.1.6 – Horizontal sync and file naming in matrix_fill.

1.2.1 linear_spectrogram

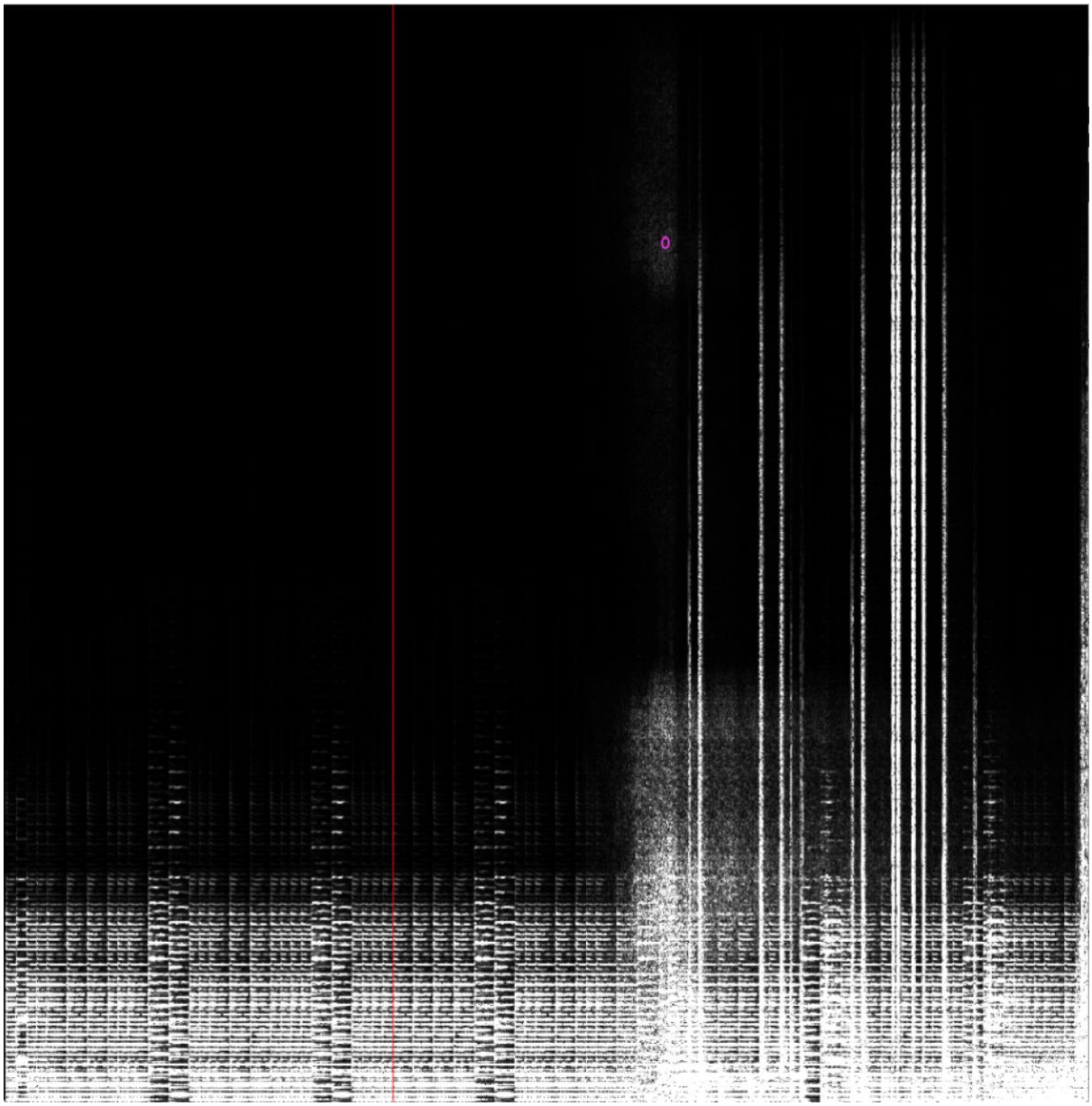

Figure 1.2.1.1 – The linear spectrogram

The abstraction 'text_file_read' in Figure 1.2.1.2 is at the top right of every patch from 1.1 – 1.6.3. It reads the text file 'fft_info.txt' created in 1.1 main and sends each line of information to its relevant 'v' object, where it is stored and called upon throughout the rest of the patch.

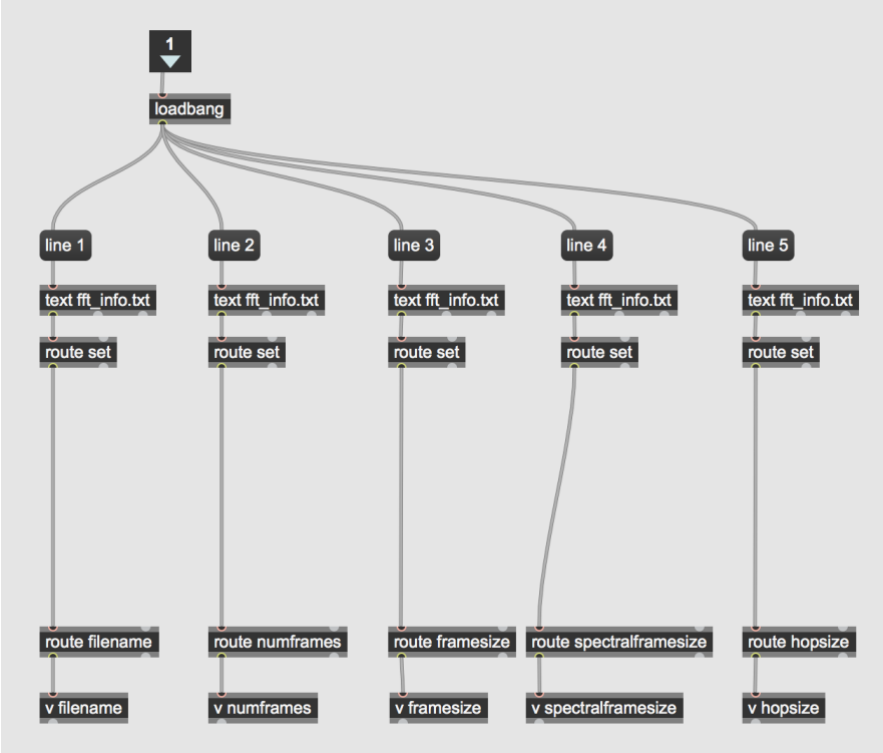

Figure 1.2.1.2 – 'text_file_read' abstraction

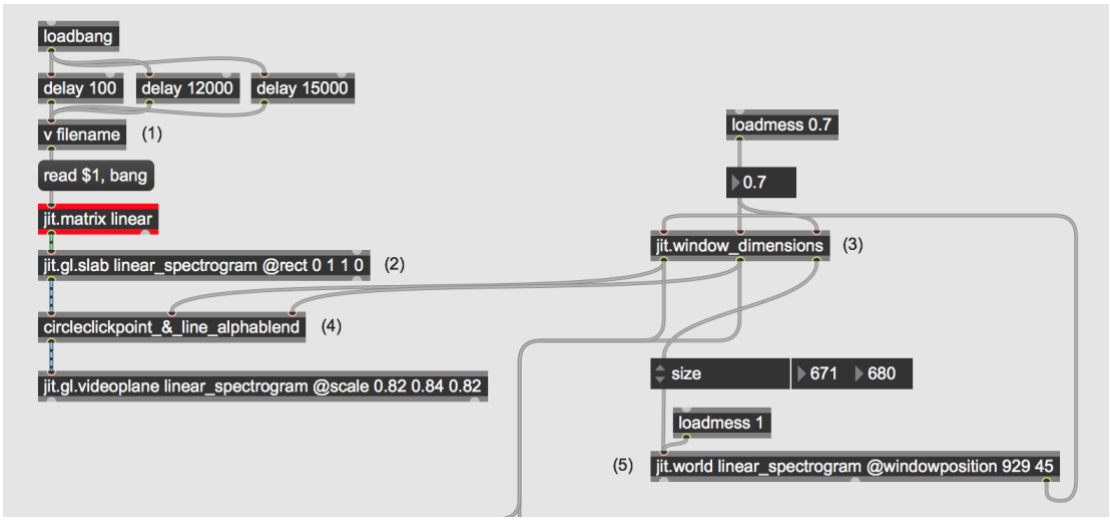

Figure 1.2.1.3 – Main patch window in linear_spectrogram

Figure 1.2.1.3 depicts the top section of 'linear_spectrogram', and is the essence of a structure I use for the patches in 1.2.2 and 1.4.1 - 1.6.3. Object 'v filename' at (1) outputs the name of the .jit file that was created in 1.1 main and prepends it with a read message. The 'jit.matrix linear' now contains the unprocessed DSTFT matrix created in 1.1 main and this matrix in then converted into a shader file and inverted in 'jit.gl.slab linear_spectrogram' at (2): now the matrix is in openGL shader format and being processed by the GPU, easing the computational load on the CPU. The 'jit.window_dimensions' abtraction at (3) is concerned with the scaling of the window. In Figure 1.2.1.4 at (1), the x and y dimensions of the user's screen are output and divided by the number of frames in and spectral frame size of the sample respectively. These ratios are then multiplied by a user-specified window scaling factor at (2). These ratios are then multiplied with the number of frames in and spectral frame size of the sample at (3) and the values output here are dimensions of the jit.window, having been scaled by both the user's screen size and their optional input scaling factor. This structure also ensures that regardless of whether the sample is 1 or 1000 frames long, the window always opens at the same size. Inlet 1 of the abstraction receives raw click point information from the window, the x and y coordinates of the click point are then scaled at (4) by the same factor used to scale window dimensions, ensuring that the correct frame number and frequency bin are output, regardless of the size of the window.

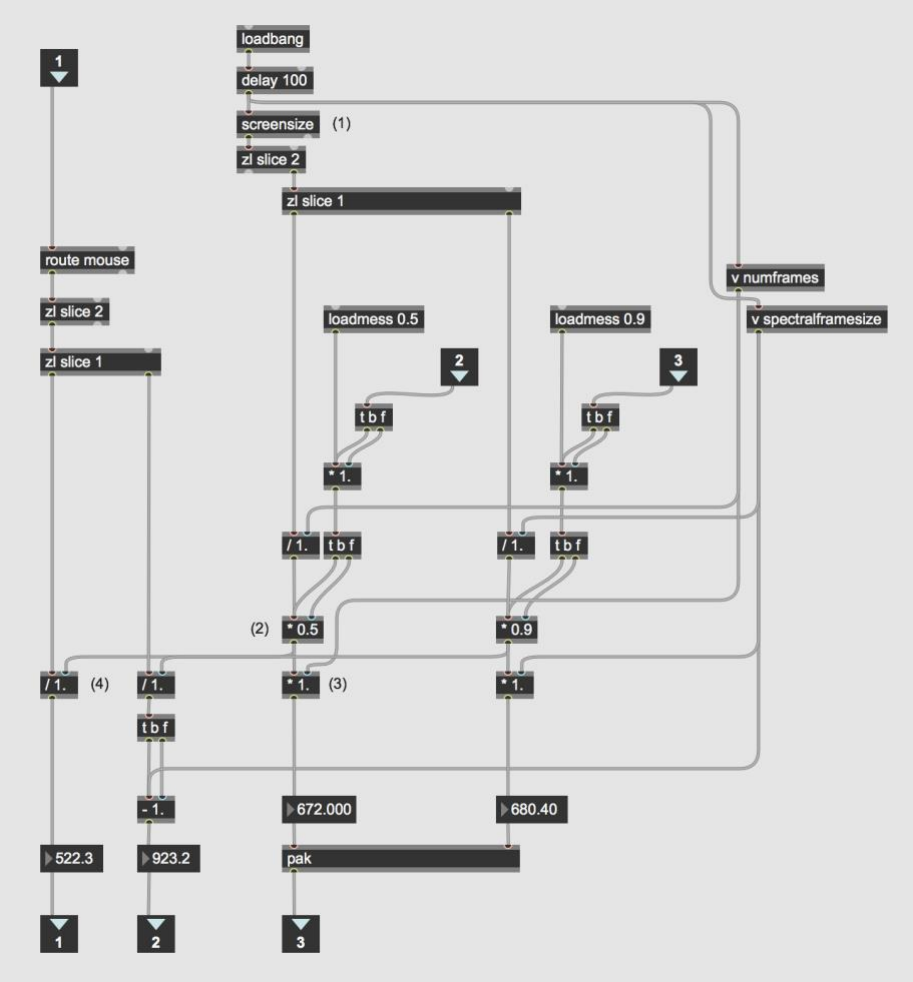

Figure 1.2.1.4 – 'jit.window_dimensions' abstraction

The ‘circleclickpoint_&_line_alphablend’ abstraction at (4) in Figure 1.2.1.3, performs the role of blending a red line to designate playback point and a purple oval to designate click point with the spectrogram (both visible in Figure 1.2.1.1). At (1) in Figure 1.2.1.5, the frame number and frequency bin at click point output from ‘jit.window_dimensions’ are concatenated in ‘pak’ and are used to outline the extremities of the purple oval at click point. (2) shows how the y value (bin number) has both 4 added to and subtracted from it with the values from these two operations being interpreted at the top and bottom points of the oval; a similar operation is carried out on the x (frame number) value that outputs the left and right points of the oval. This message, along with three others that dictate the colour of the oval, the background colour and the pen (drawing point) size are then sent to a ‘jit.lcd’ object at (3). Similarly, at (4) this jit.lcd object draws the red line playback point indicator, the x value of which is received by whichever of five playback options the user has selected (see section ‘playback’ for more information). Frustratingly, the ‘jit.lcd’ object does not offer the functionality to have its dimensions adaptively scaled like other jitter objects, this meant that I had to write JavaScript code which would remove the two previous ‘jit.lcd’ objects, create two new ones and then connect them to their surrounding objects. This JavaScript inside the object ‘js jit.2lcd_creator’ at (5) receives the number of frames and spectral frame size (which will be the dimensions of the ‘jit.lcd’ objects) as values to be passed into its function. After the ‘jit.lcd’ objects are created (see Figure 1.2.1.6 for more detail) they are combined in the ‘jit.op @op +’, the result of which is sent into a ‘jit.matrix’ of char data type (single byte values) where the planes are mapped 1 1 2 3. If the ‘@planemap’ argument is not specified, by default the ‘jit.matrix’ with char data type at (6) will map its input to planes 0 1 2 3, with planes 0-2 containing red, green and blue information respectively and 3 being the alpha channel. It is essential to leave plane 0 free which will be occupied by the spectrogram being passed through inlet 1 and to map the 4th channel (alpha channel) to plane 3 as this is the channel which will be blended with the spectrogram in ‘jit.gl.pix @gen alphablend’ at (6) (to overlay the red line and purple oval). It’s interesting to see how mapping the red data to plane 1 when it should be plane 0, means that it overwrites the green data and then blends with the blue data to make the oval purple, when I specified it should be red in the ‘frgb 255 51 255’ message. It is also worth noting that the abstraction ‘line_alphablend’ is a simplification of ‘circleclickpoint_&_line_alphablend’ that only displays the red line; I used this patch in patches where due to the spectral effects being applied, looking for specific bin values would be impossible.

Figure 1.2.1.6 shows the JavaScript which performs the ‘jit.lcd’ creation. Lines 3-7 outline the global variables that will be used in the function ‘object_creation’: variables ‘x_dim’ and ‘y_dim’ are mapped to inputs 1 and 2, as they will be receiving the number of frames and spectral frame size respectively. Inside the function ‘object_creation’, lines 11-21 and 23-29 both perform the same operations on the oval in ‘jit.lcd’ at (3) and the line in ‘jit.lcd’ at (4) in Figure 1.2.1.5:

1. The previous ‘jit.lcd’ is removed.
2. A new ‘jit.lcd’ is created whose arguments are ‘4’, ‘char’ and the x_dim and y_dim variables from lines 3 and 4.
3. The ‘jit.lcd’ is given a scripting name.

4. The ‘jit.lcd’ is connected to specific objects around it.

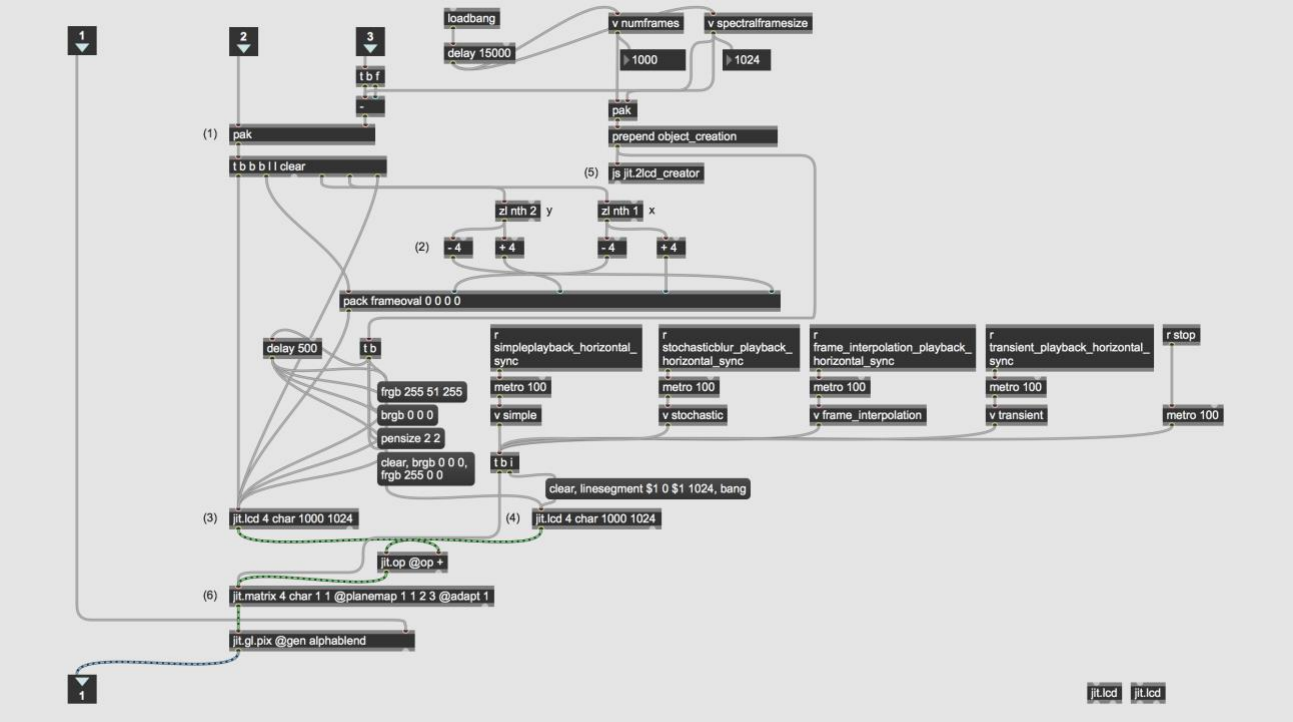

Figure 1.2.1.5 - ‘circleclickpoint_&_line_alphablend’ abstraction

```
autowatch = 1;

var x_dim = input1;
var y_dim = input2;
var point_jitlcd = 0;
var line_jitlcd = 0;
var object_size = 30;

function object_creation (x_dim,y_dim) {

    this.patcher.remove(point_jitlcd);
    this.patcher.remove("point_jitlcd");
    point_jitlcd =  this.patcher.newdefault(237, 525, "jit.lcd", 4, "char", x_dim, y_dim);
    point_jitlcd.varname = "point_jitlcd";
    this.patcher.connect(this.patcher.getnamed("frgb"),0,point_jitlcd,0);
    this.patcher.connect(this.patcher.getnamed("brgb"),0,point_jitlcd,0);
    this.patcher.connect(this.patcher.getnamed("pensize"),0,point_jitlcd,0);
    this.patcher.connect(this.patcher.getnamed("pack_framerect"),0,point_jitlcd,0);
    this.patcher.connect(this.patcher.getnamed("trigger"),5,point_jitlcd,0);
    this.patcher.connect(this.patcher.getnamed("trigger"),0,point_jitlcd,0);
    this.patcher.connect(point_jitlcd,0,this.patcher.getnamed("jitop"),1);

    this.patcher.remove(line_jitlcd);
    this.patcher.remove("line_jitlcd");
    line_jitlcd = this.patcher.newdefault(550, 525, "jit.lcd", 4, "char", x_dim, y_dim);
    line_jitlcd.varname = "line_jitlcd";
    this.patcher.connect(this.patcher.getnamed("line_colour"),0,line_jitlcd,0);
    this.patcher.connect(this.patcher.getnamed("linesegment"),0,line_jitlcd,0);
    this.patcher.connect(line_jitlcd,0,this.patcher.getnamed("jitop"),0);

}
```

Figure 1.2.1.6 – ‘2lcd_creation.js’ JavaScript

The ‘spectrogram_interaction_top_level linear’ sections 1.2.1 and 1.2.2 serve to provide the user with information from the spectrogram in relation to where their cursor is. It is useful to think of this abstraction as working in 3 modules:

1. frame2ms

To calculate what a frame number represents in milliseconds, the equation in Figure 1.2.1.7 can be used. Value n is sent through inlet 1 in 'frame2ms' (Figure 1.2.1.8), and $\frac{WindowSize}{OverlapFactor}$ is represented in 'v spectralframesize' (as the frame size is 2048 and the overlap factor I use is 2), which is multiplied by n. In isolation, the 'sampstoms~' object in Figure 1.2.1.8 calculates the duration of 1 sample by taking the reciprocal of the sampling rate ($\frac{1}{S_r}$), and in this context it is multiplying the value of $\frac{1}{S_r}$ by whatever value is coming out of the multiplication object.

$$t = n \times \frac{WindowSize}{OverlapFactor} \times \frac{1}{S_r}$$

Figure 1.2.1.7 – Equation to calculate millisecond equivalent t of frame number n

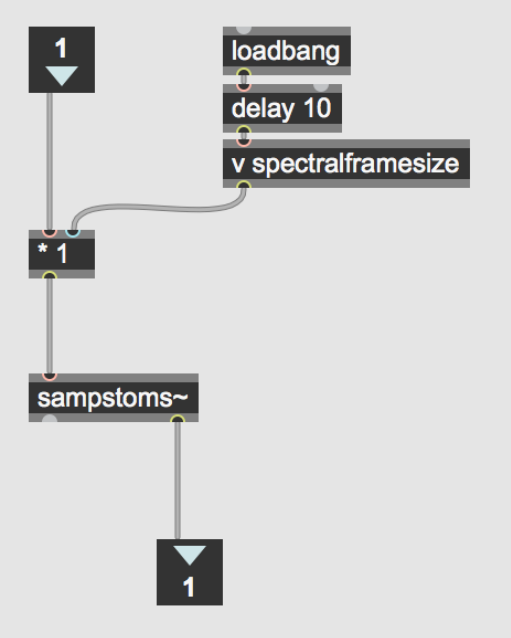

Figure 1.2.1.8 – 'frame2ms' abstraction

2. bin2centrefreq

For considering the second module of 'spectrogram_interaction_top_level linear', it is useful to recall Figure FT.3 from section 'The Fourier Transform', we can use this equation to calculate the central frequency f_k of bin index *k*. The abstraction 'bin2centre_freq' in Figure 1.2.1.9 encapsulates this equation: bin index *k* is passed through inlet 1 and multiplied by the S_r output by object 'dspstate~', the result of which is divided by N (represented by object 'v framesize').

$$f_k = \frac{kS_r}{N}$$

Figure FT.3 – Equation to calculate central frequency of bin index *k*

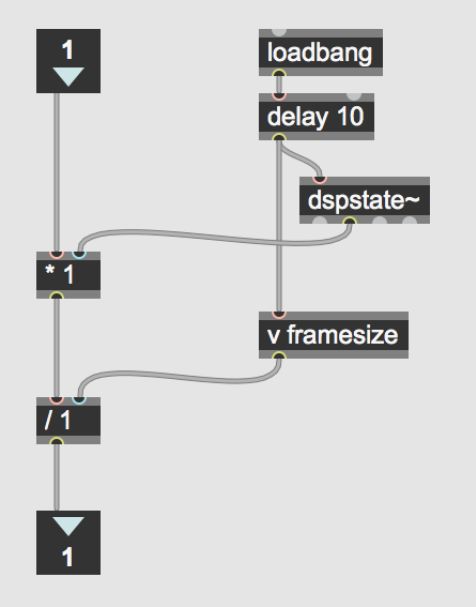

Figure 1.2.1.9 – 'bin2centrefreq' abstraction

3. ΔØ_&_amplitude

The third module of 'spectrogram_interaction_top_level linear' performs two functions. Firstly it displays the amplitude value in of bin *m,k* (*m* and *k* being the frame number and bin number respectively) that the user has chosen through clicking on the spectrogram. The 'getcell' command at the top of Figure 1.2.1.10 is fed the values *m* and *k*, when then causes 'jit.matrix linear' (containing the same contents as the jit.matrix in Figure 1.2.1.) to output the amplitude and running phase ΔØ information at bin *m,k*. The amplitude information is output direct, while the running phase ΔØ information is sent into Max objects that perform a rearranged version of the equation for calculating true frequency f_t, outlined in Figure FT.7 from section 'The Fourier Transform'.

$$f_t = f_k + \Delta\emptyset \times \frac{S_r}{2\pi \times N}$$

Figure FT.7 – equation for calculating true frequency at bin index k

$$f_t = \frac{\Delta\emptyset \times S_r}{(\frac{N}{2\pi})} + f_k$$

Figure FT.7' – rearranged equation for calculating true frequency at bin index k

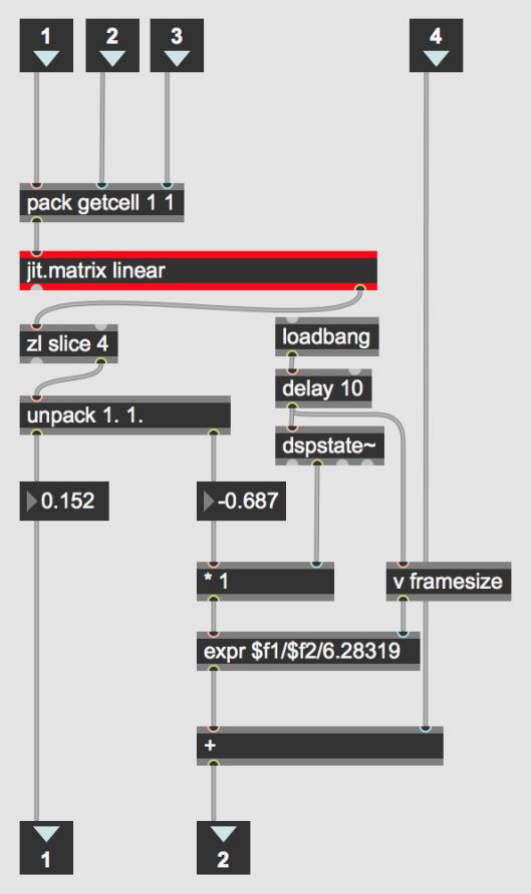

Figure 1.2.1.10 – 'ΔØ_&_amplitude' abstraction

The right outlet of the 'unpack 1. 1.' object sends $\Delta\emptyset$ to be multiplied by the sample rate S_r output from 'dspstate~', the result of which is sent into the 'expr' object and divided by $\frac{N}{2\pi}$ (N being represented by $f1 and 2π being represented by 6.28319). As Figure FT.7' shows, we still need to add the central bin frequency of index k to this result, which is done in the addition box above outlet 2 in Figure 1.2.1.10. Now 'spectrogram_interaction_top_level linear' is outputting time in ms, amplitude and true frequency at bin index m,k, which are all displayed in the 'linear_spectrogram' main patch.

1.2.2 logarithmic_spectrogram

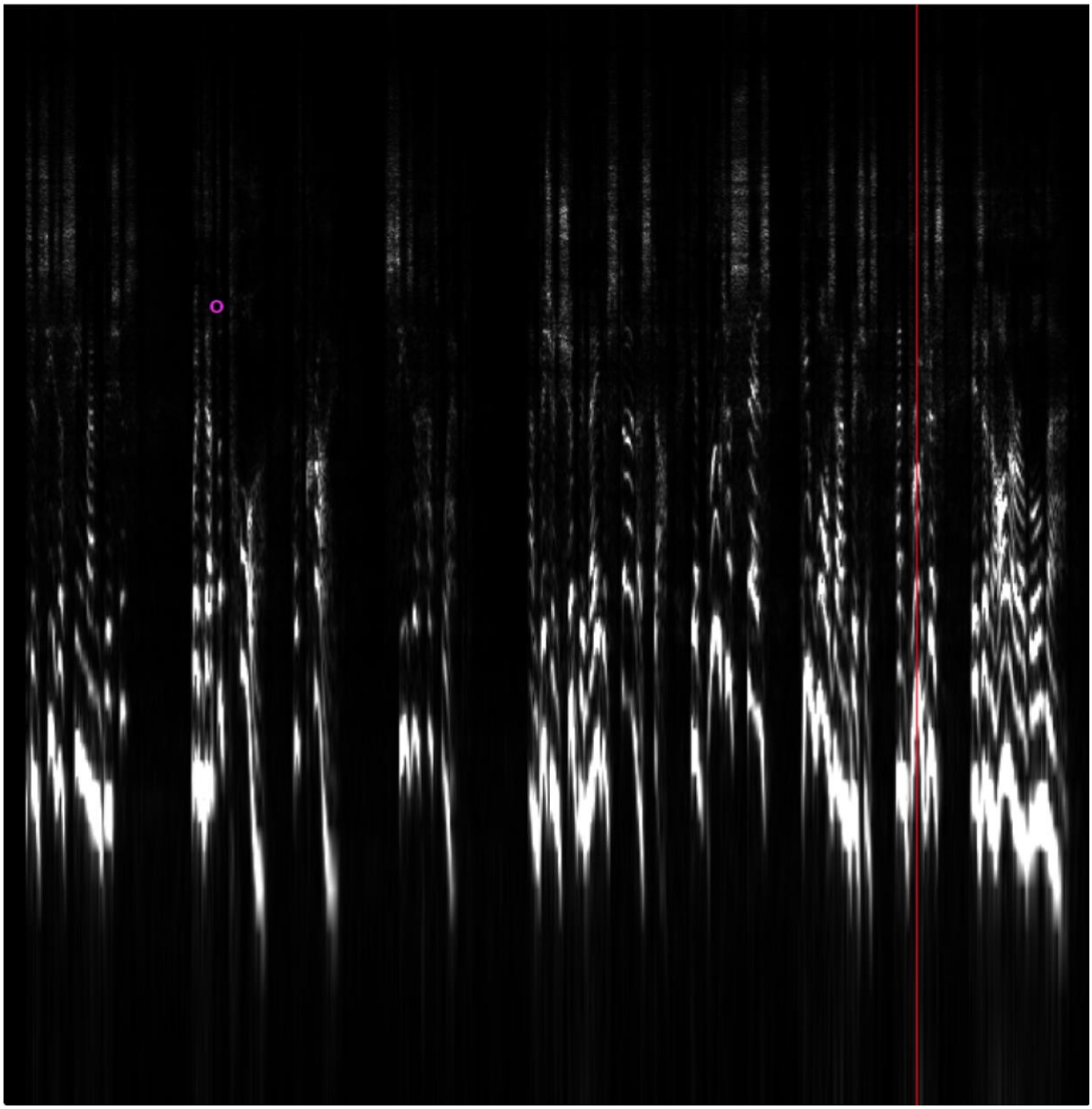

Figure 1.2.2.1 – The logarithmic spectrogram

'1.2.2 logarithmic_spectrogram' is structured very similarly to the linear spectrogram in section 1.2.1. The central difference between the two is inclusion of the abstraction 'logarithmic_mapping' at (1) below the 'jit.matrix', exhibited in Figure 1.2.2.2.

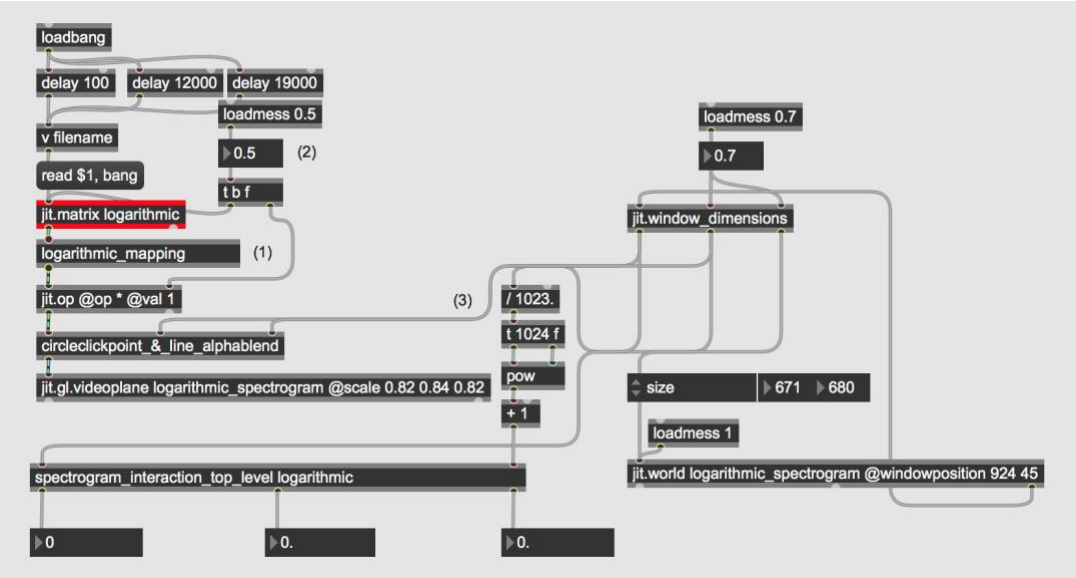

Figure 1.2.2.2 – Main patch window in logarithmic_spectrogram

At (1) in Figure 1.2.2.3 a 'counter' set comfortably over the spectral frame size (1024) is driven by a metro object at a 15 millisecond interval. Every 15 milliseconds this object outputs an integer 1 higher than the last, which is used as the y coordinate in 'setcell', and the numerator in the exponent of 1023 in the 4 objects above (2). As a result, the number output from (2) gets exponentially larger, as demonstrated in a

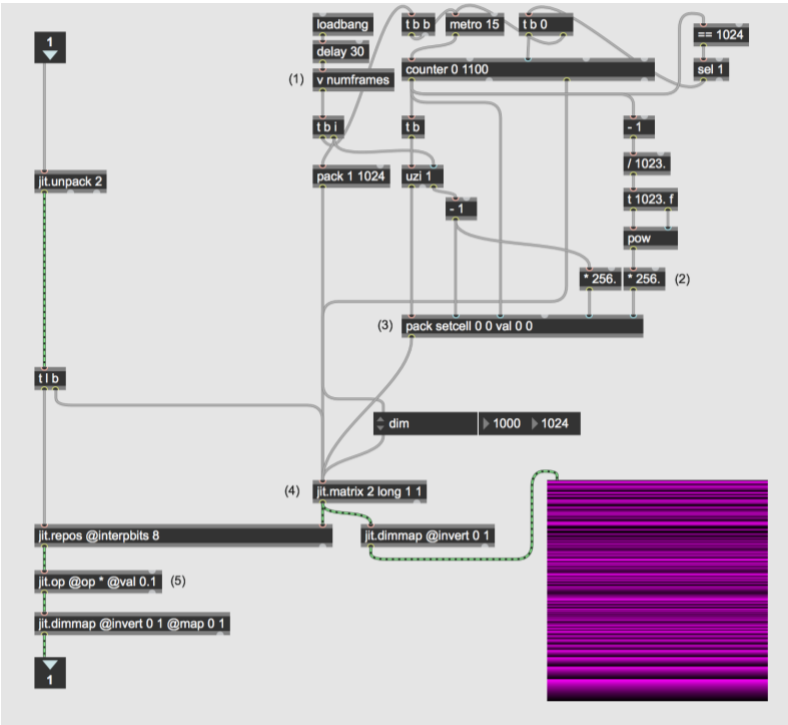

Figure 1.2.2.3 – 'logarithmic_mapping' abstraction

graph I sketched in MATLAT in Figure 1.2.2.4. Every time the 'counter' object progresses, an 'uzi' object also fills out all values in that row, meaning the matrix at (4) is filled row by row over the course of 15 seconds.

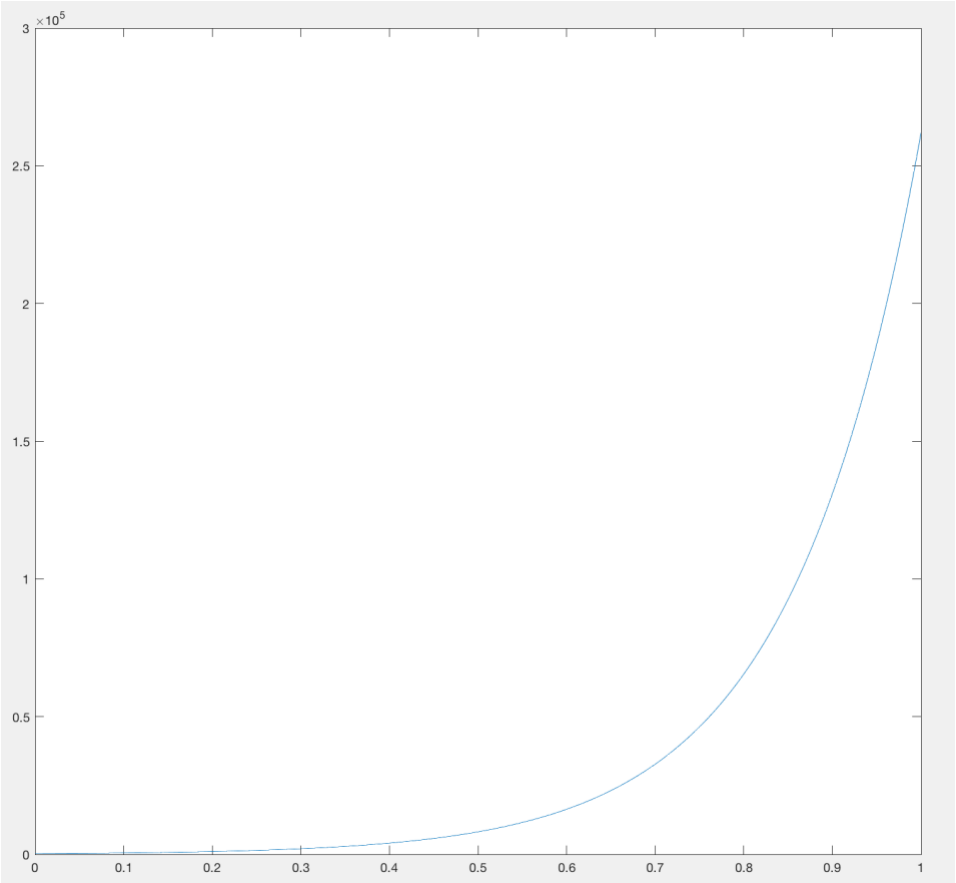

Figure 1.2.2.4 – x axis represents exponent in $1023^{\frac{x}{1023}}$ against output of object at (2) on y axis

Clearly the curve in Figure 1.2.2.4 is the opposite of a logarithmic curve, but as the matrix containing our DSTFT data that has entered Figure 1.2.2.3 through inlet 1 is itself inverted, scaling it exponentially is the equivalent of scaling a non-inverted matrix logarithmically. This exponentially scaled inverted matrix is then amplified appropriately and inverted at (6). The user can also manually adjust the amplification of their logarithmic spectrogram at (2) in Figure 1.2.2.2 if they wish to highlight certain regions. (3) in Figure 1.2.2.2 demonstrates the logarithmic scaling of the bin number required before being interpreted by the 'spectrogram_interaction_top_level': without this the abstraction would still be reading the 'jit.window' information as though it was linear.

1.3 mask_draw

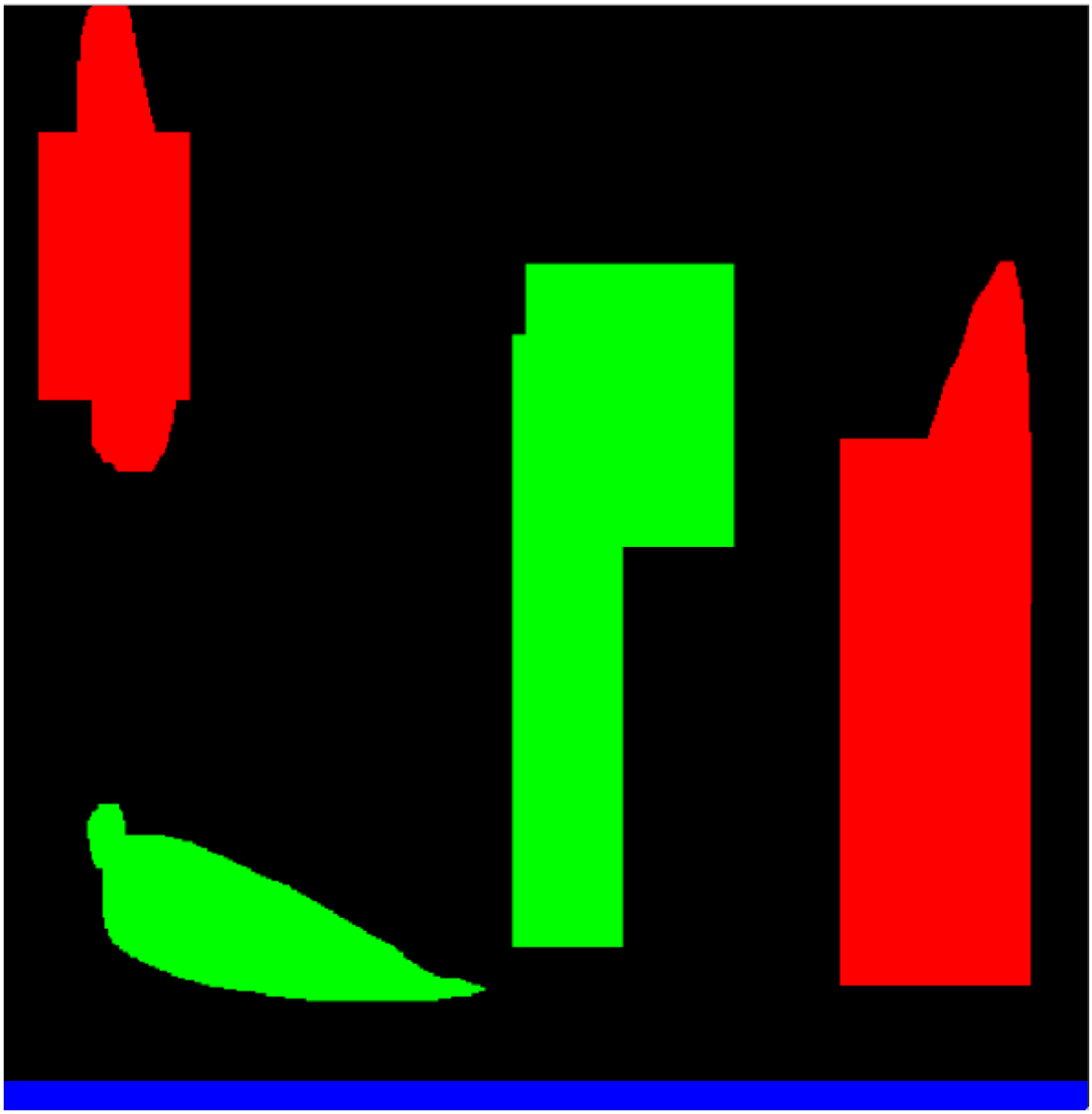

Figure 1.3.1 – Three masks drawn by user

For spectral effects 1.4.1 – 1.6.3, masks are all a crucial part of their operation. In essence, a mask allows the user to specify exactly where in the spectrogram they want a spectral effect to be applied. Spectral effects 1.4.1 – 1.6.3 all come in three forms: custom mask, rectangular mask or no mask. Custom mask, explained in this section between pages 32 and 33, allows the user to draw up to three masks by hand over a logarithmic spectrogram of the audio sample they have chosen. Rectangular mask allows the user to move and scale the size of a rectangular mask which contains spectrally processed DSTFT data on the fly while the audio is playing back. No mask applies the spectral effect to all of the DSTFT data in the audio sample.

'1.3 mask_draw' is a patch in two halves. The ultimate purpose of the patch is to allow the user to draw shapes which represent masks on a logarithmic spectrogram of their audio sample. Through analysing how this was achieved by Tadej Droljc, I was inspired to provide the user with functionality to draw both rectangular and free-hand shapes onto the same mask.[12]

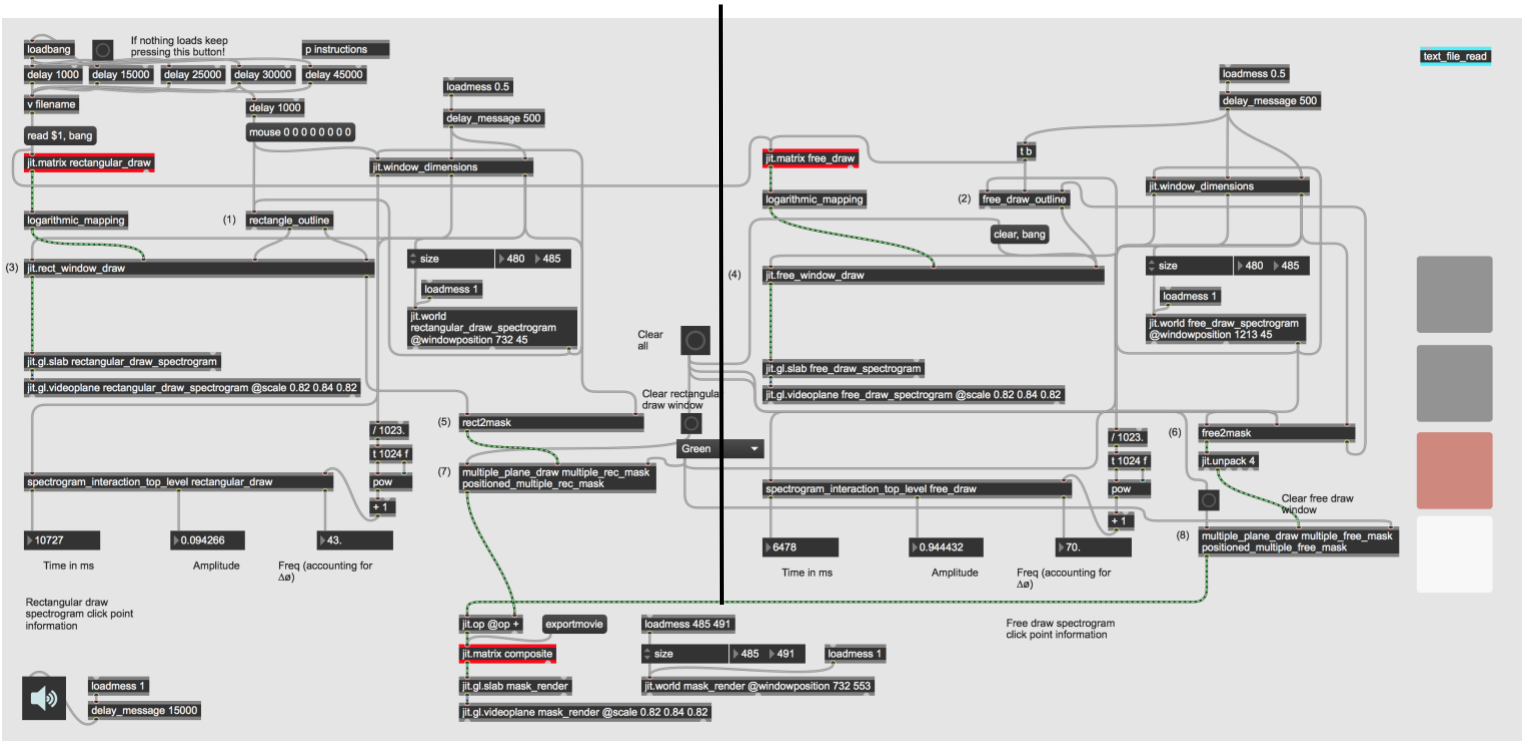

Figure 1.3.2 – 'mask_draw' main patch window

The line intersecting Figure 1.3.2 represents the division in the patch as a whole. To the left of the line is the code associated with drawing a rectangular shape, and to the right is the code associated with the drawing of free-hand shapes. Below the line, the rectangular and free-hand shapes drawn by the user are combined and displayed in the 'mask_render' window (depicted in Figure 1.3.1).

[12] Tadej Droljc, 'STFT Analysis', 121-146.

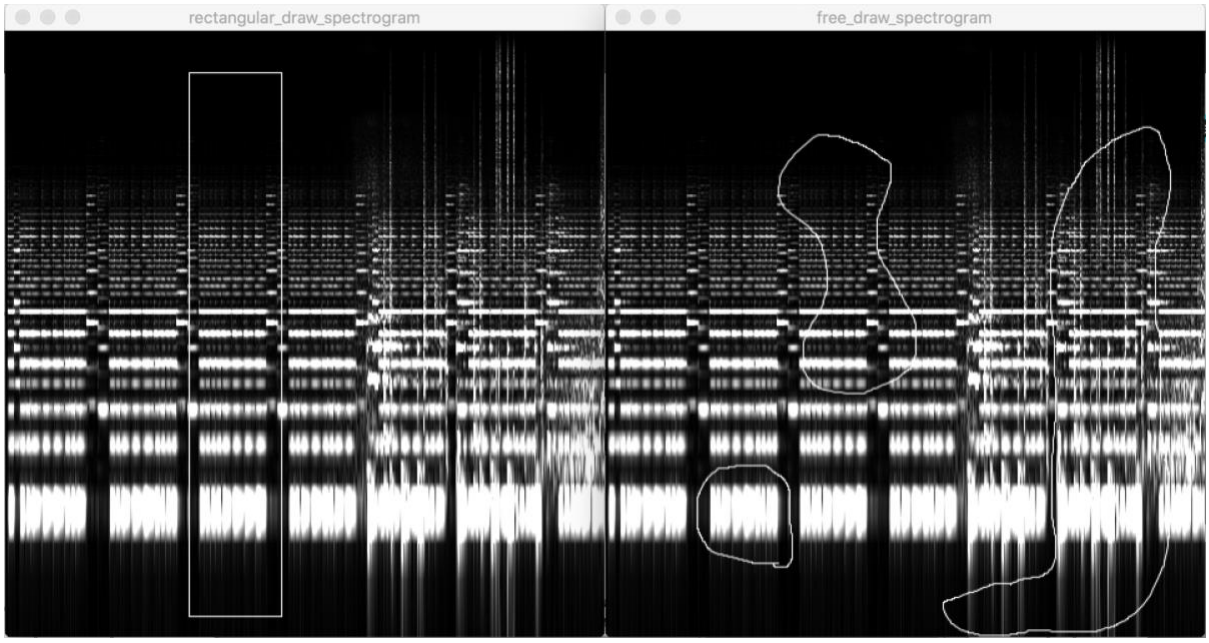

Figure 1.3.3 – 'rectangular_draw_spectrogram' and 'free_draw_spectrogram' windows

As the same principles run in parallel down each half of Figure 1.3.2 in the creation of the two spectrograms in Figure 1.3.3, I will describe the function of each abstraction in Figure 1.3.2 and its relation to its equivalent abstraction on the other side of the black dividing line (I highly recommend that this commentary is followed with the patch '1.3 mask_draw' open). 'rectangle_outline' at (1) and 'free_draw_outline' at (2) in Figure 1.3.2 both perform the function of converting the mouse coordinates output from the 'rectangular_draw_spectrogram' and 'free_draw_spectrogram' windows into messages for erasing previous shapes and subsequently drawing a rectangle or line that can be interpreted by a 'jit.lcd' in the abstractions at (3) and (4). The abstractions 'jit.rect_window_draw' and 'jit.free_window_draw' at (3) and (4) respectively are essentially identical, aside from 'jit.rect_window_draw' includes a bypass for the mouse position coordinates. Both patches are sent the dimensions of their 'jit.window' through inlet 1 which set the dimensions of the 'jit.matrix' which is fed into the 'jit.lcd'. Through inlet 2 both patches receive the DSTFT data which is input into the 'jit.matrix' whose dimensions were set through inlet 1. Through inlet 3 the messages created in abstractions 'rectangle_outline' at (1) and 'free_draw_outline' at (2) are sent into their respective 'jit.lcd'. The colouration of both 'jit.lcd's are inverted and then overlaid over the original 'jit.matrix' which contains the DSTFT data.

While performing the same function, abstractions 'rect2mask' and 'free2mask' at (5) and (6) do so in completely different ways. 'rect2mask' receives the rectangle draw command message (which outlines the coordinates of the 4 corners of the rectangle) from (1), then clips and scales them to be in the range of the 'jit.window' 'rectangular_draw_spectrogram' in Figure 1.3.3. Enabling the user to be able to sketch on a logarithmic spectrogram of their audio sample for greater control over regions where frequencies generally cluster, means that the rectangle they drew needs to be scaled from logarithmic to linear (as the rest of my project displays the spectrograms linearly). This is done so through a similar techniques used in (3) of Figure

1.2.2.2. At the bottom of 'rect2mask', the top left coordinates followed by the bottom right coordinates of the rectangle are used as the start and end points respectively of 2x2 'jit.matrix rec_mask' (containing white char data) over 'jit.matrix positioned_mask' (containing black char data) which is scaled to the dimensions of the DSTFT data of the user's audio sample. This creates the white rectangle in a black background depicted in Figure 1.3.4.

Figure 1.3.3 – example mask output from 'rect2mask'

Abstraction 'free2mask' also clips and scales the mouse coordinates from the 'jit.window' according to the size of the window itself, but this new scaled data is used to sketch a closed contour vertex on a polygon in openGL. For this section of the patch I drew heavily from Tadej Droljc's work as openGL sketching is not a topic I am familiar with, but what the various message commands that are sent 'jit.gl.sketch' do is always sketch a closed polygon, regardless of where the user started and ended their line (as demonstrated in Figure 1.3.4.[13] At the bottom of 'free2mask' the openGL video plane is read by 'asyncread'

Figure 1.3.4 – incomplete user shape scaled and automatically completed in 'free2mask'

[13] Droljc, 'STFT Analysis', 143.

and exported as a char matrix. The abstraction 'multiple_plane_draw' at (7) and (8) splits the matrices output by 'rect2mask' and 'free2mask' into 3 planes of red, green and blue and allows the user to discern when they are drawing into which plane. These composite matrices are they output from their respective abstractions and combined in the 'jit.op @op +' object at the bottom of the main patch window in Figure 1.3.2. The 'exportmovie' message over 'jit.matrix composite' allows the user to save the contents at a .mov file and reuse their custom masks with spectral effects later in the project.

Custom Mask Application

As I mentioned at the beginning of this section, spectral effects 1.4.1 – 1.6.3 allow for the custom masks drawn in '1.3 mask_draw' to be used to apply the spectral effect to specific regions of the DSTFT data. Whichever of the three masks is selected function as the region in which the spectral effect is active, as demonstrated in Figure 1.3.5. The four regions in this image are where the spectral effect '1.5.3 spectral_smear' is being applied (more details on this effect in section 1.5.3).

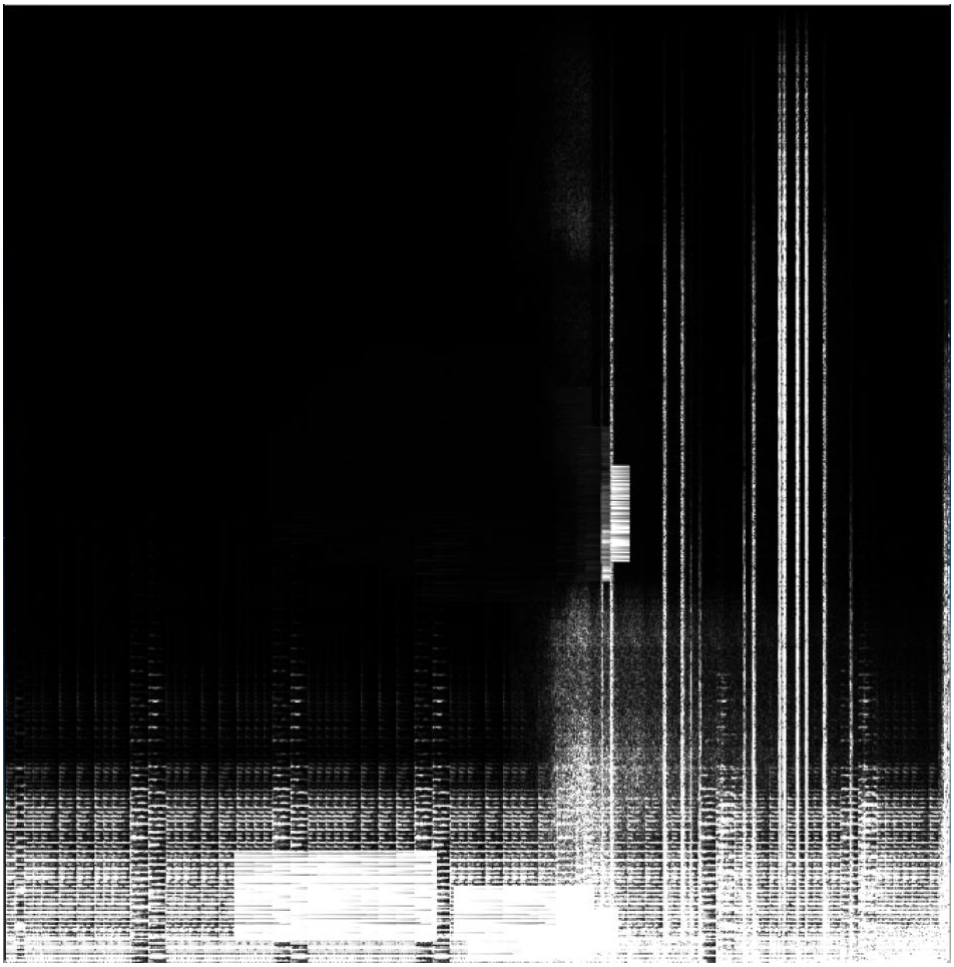

Figure 1.3.5 – 'spectral_smear' effect applied to 4 mask regions on the spectrogram

To apply this mask to our original DSTFT spectrogram requires fairly straightforward matrix multiplication whereby the mask area is cut from the spectrogram through multiplication, processed and then added back.

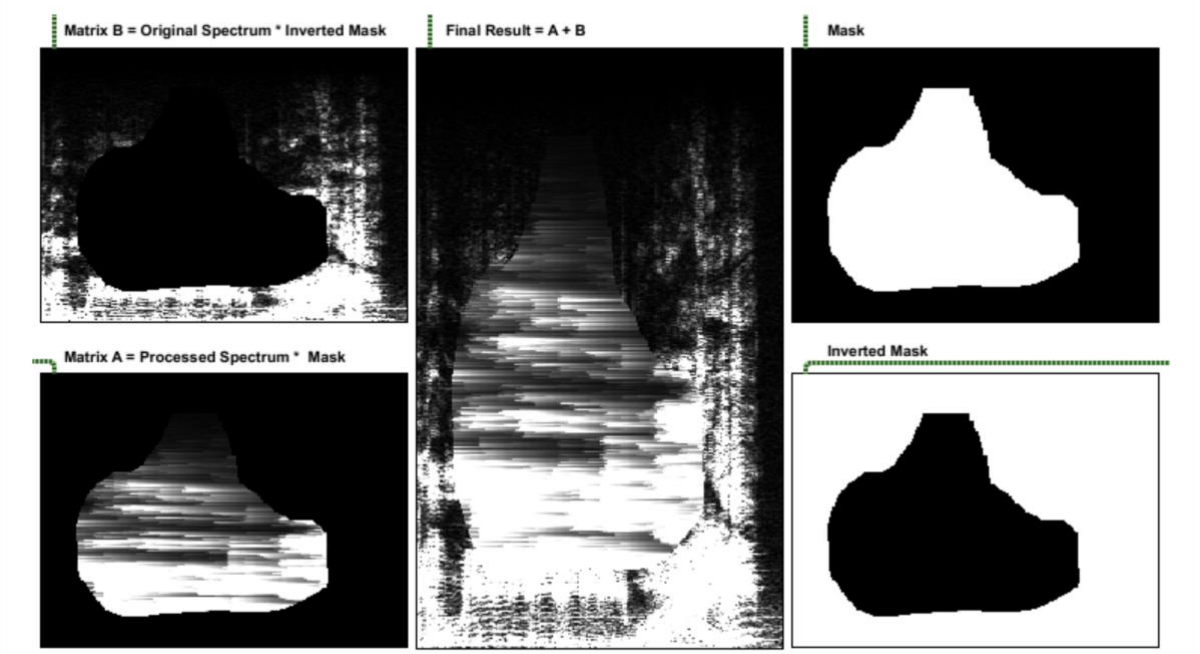

Figure 1.3.6 – The process of applying a mask to a spectrogram

This process is contained in the abstraction 'mask_application', which is present in every (custom) iteration of spectral effects 1.4.1 – 1.6.3. Once again using '1.5.3 spectral_smear' as an example, the 'mask_application' abstraction is inserted in the patch after the after a conversion to SLAB process (to save CPU power by processing on the GPU in openGL) and before the masked matrix is blended with the red line playback head in 'line_alphablend'.

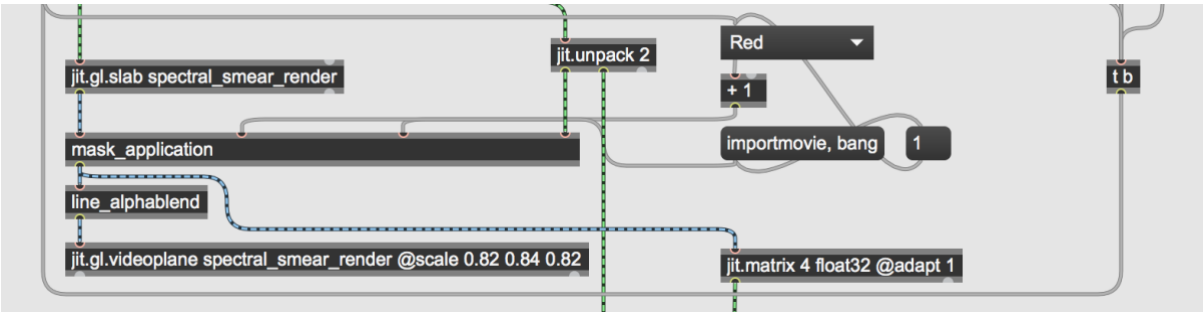

Figure 1.3.7 – 'mask_application' in the main patch window

Rectangular Mask Application

The rectangular mask iteration of spectral effects 1.4.1 – 1.6.3 has the added benefit of presenting the user with a mask that can be moved and resized while the patch is in playback mode. Figure 1.3.8 depicts the spectral effect 'spectral_smear' (more details on this effect in section 1.5.3) being applied to the original DSTFT matric via the rectangular mask.

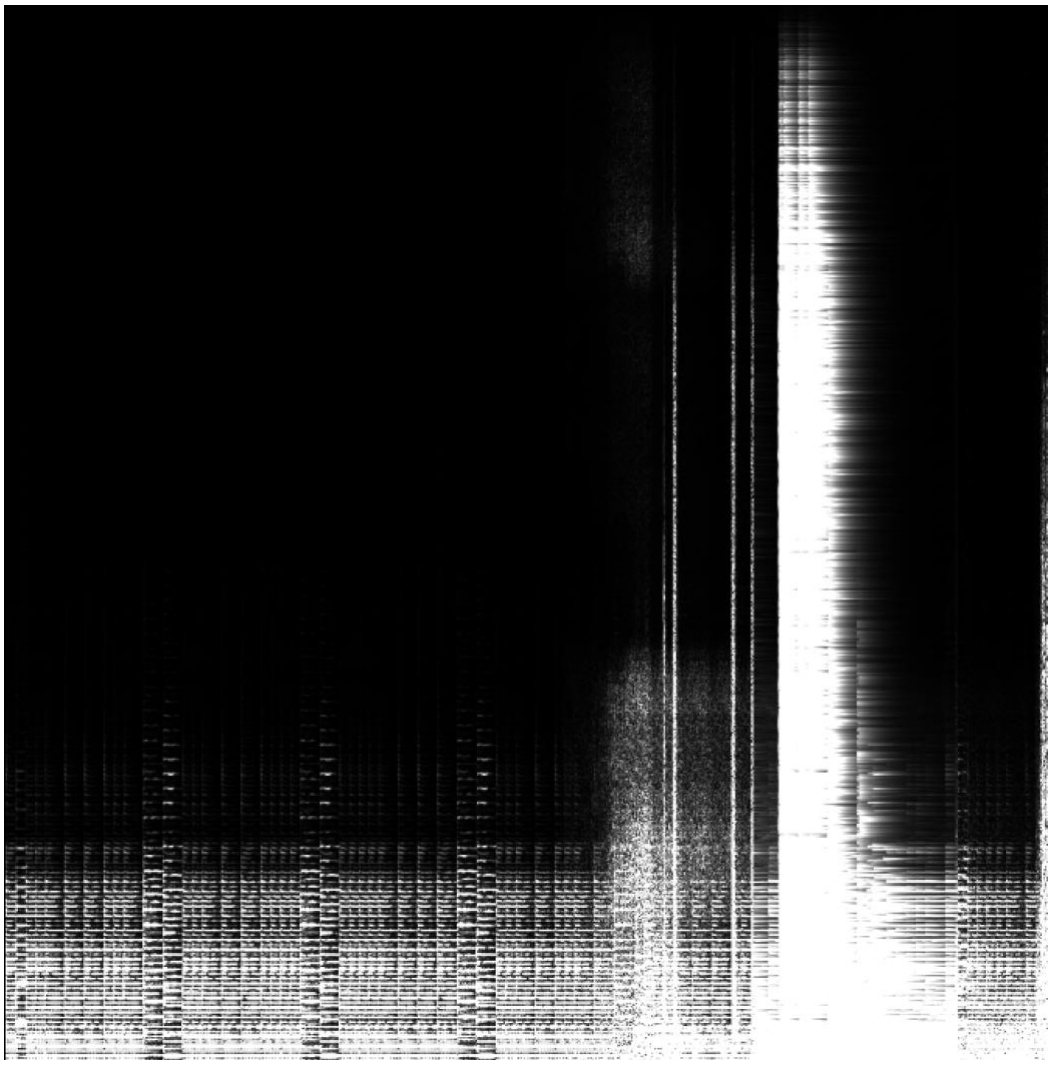

Figure 1.3.8 – 'spectral_smear' effect applied to rectangular mask

I refer to '1.5.3 spectral_smear(rectangular)' for example's sake here, but this applies to the (rectangular) iteration of spectral effects 1.4.1 – 1.6.3. To achieve this effect, a submatrix is taken from our original DSTFT matrix and is processed individually before being added back to the original matrix. This process happens in two halves: extracting the submatrix and then later combining it with the original matrix after processing. The abstraction 'processing_area_size' in Figure 1.3.9 receives a value from the slider (the size of which has been set by the frame number) and is used as the offset from the start of the original

'jit.matrix' that the submatrix will be extracted from. Though not visible in presentation mode, the user also has the option of changing vertical dimensions of submatrix.

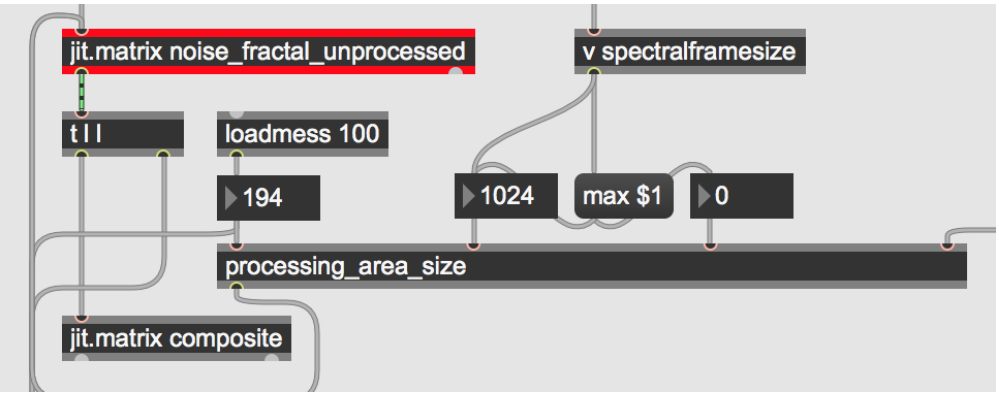

Figure 1.3.9 – 'processing_area_size' in the main patch window

After the submatrix has been processed, it is added back through a similar process to how it was taken out. Inside the 'coordinates' abstraction, the slider value is appended with the value plus the user-specified processing area size and the spectral frame size as a 'dstdimstart' and 'dstsimend' message. This is instructing the 'jit.matrix composite' matrix (the dimensions of which have been set to the size of our original DSTFT matrix) to take on the submatrix) in Figure 1.3.1.10 to take on the submatrix at the same coordinates that the submatrix was taken out at.

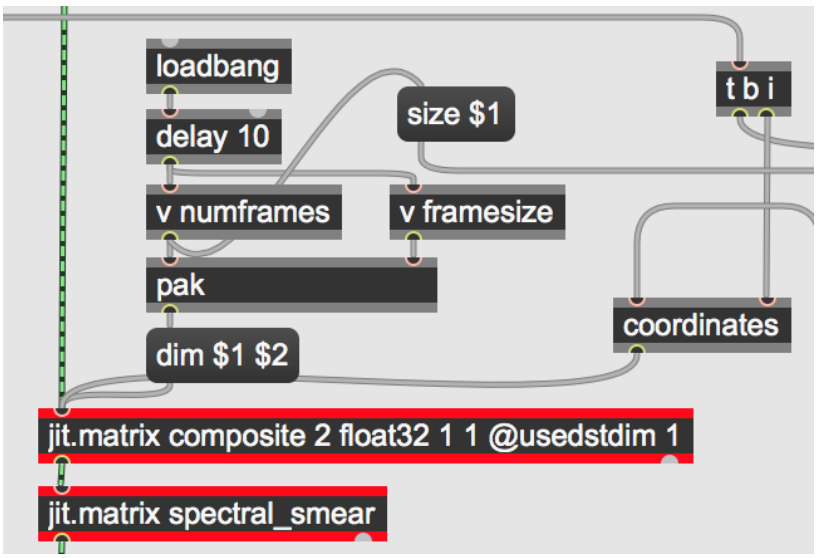

Figure 1.3.10 – 'coordinates' and submatrix re-addition in the main patch window

1.4.1 gaussian_blur

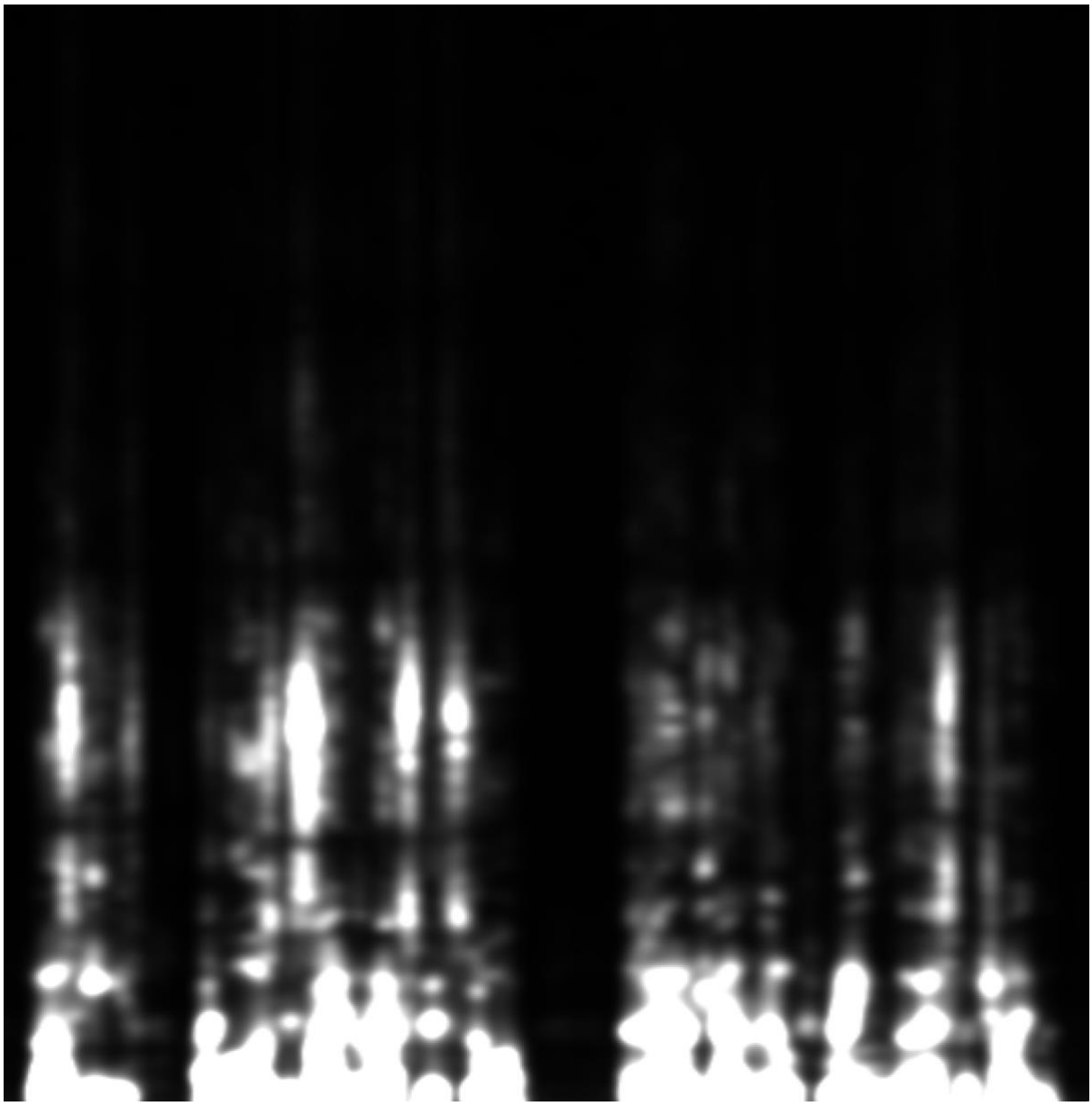

Figure 1.4.1.1 – Gaussian blur applied to a spectrogram

The Gaussian blur is a blurring effect produced through multiplying the texture in our 'jit.gl.slab' object by values on the Gaussian curve. '1.4.1 gaussian_blur' is perfect for users who wish to achieve a hazy and ephemeral quality from their sample. As demonstrated in Figure 1.4.1, using several 'jit.gl.slab' objects with the 'cf.gaussian.2p.jxs' shader loaded into them in conjunction produces a blurring effect over the whole spectrogram and subsequent audio. In my project I have implemented the Gaussian blur in openGL, as doing so ensures that as much processing is done on the GPU as possible. Figure 1.4.2 depicts the SLAB chain encapsulated in abstraction '2pass_gassian_shader' in which the Gaussian blur is processed.

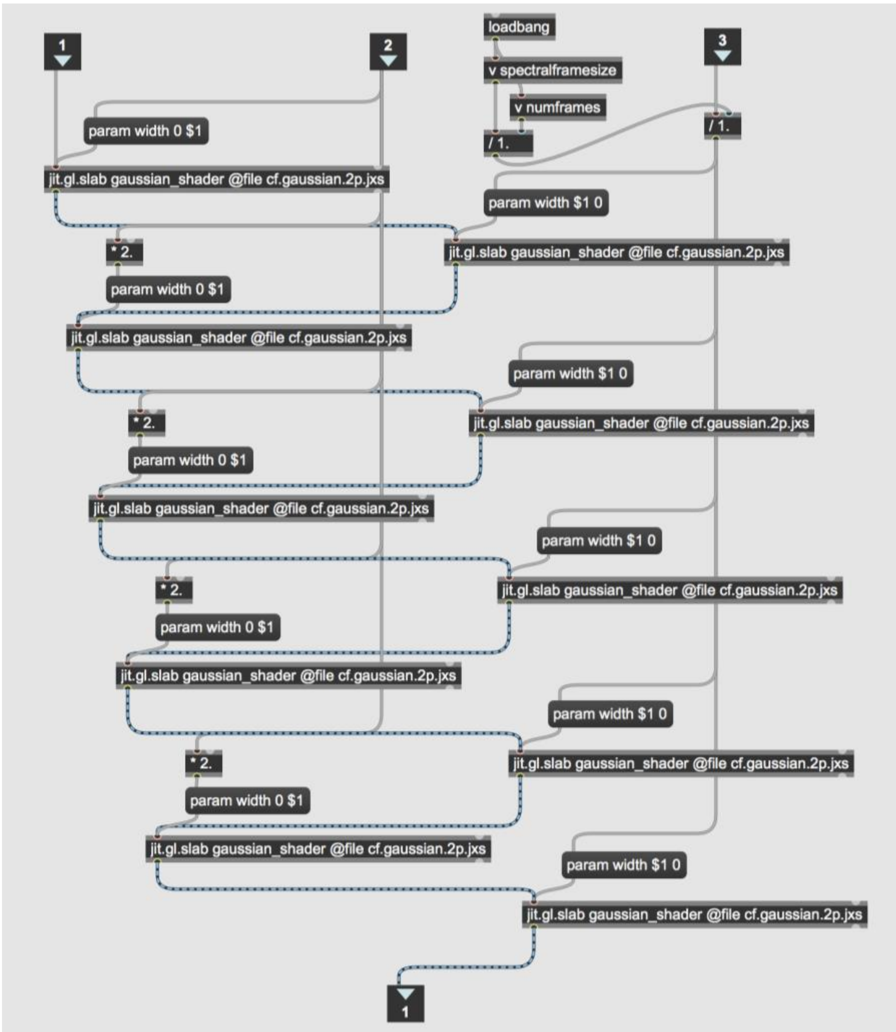

Figure 1.4.1.2 – SLAB process producing the Gaussian blur in '2pass_gaussian_shaders'

Our DSTFT matrix is passed through 10 'jit.gl.slab' objects, with the 5 down the left hand side representing Gaussian blue being applied on the y-axis (controlled through inlet 2), and the 5 on the right hand side applying Gaussian blur on the x-axis (controlled through inlet 3). As is evident in Figure 1.4.2, the texture in our SLAB chain alternatively has y-axis then x-axis Gaussian blur applied to it.

1.4.2 noise_fractal

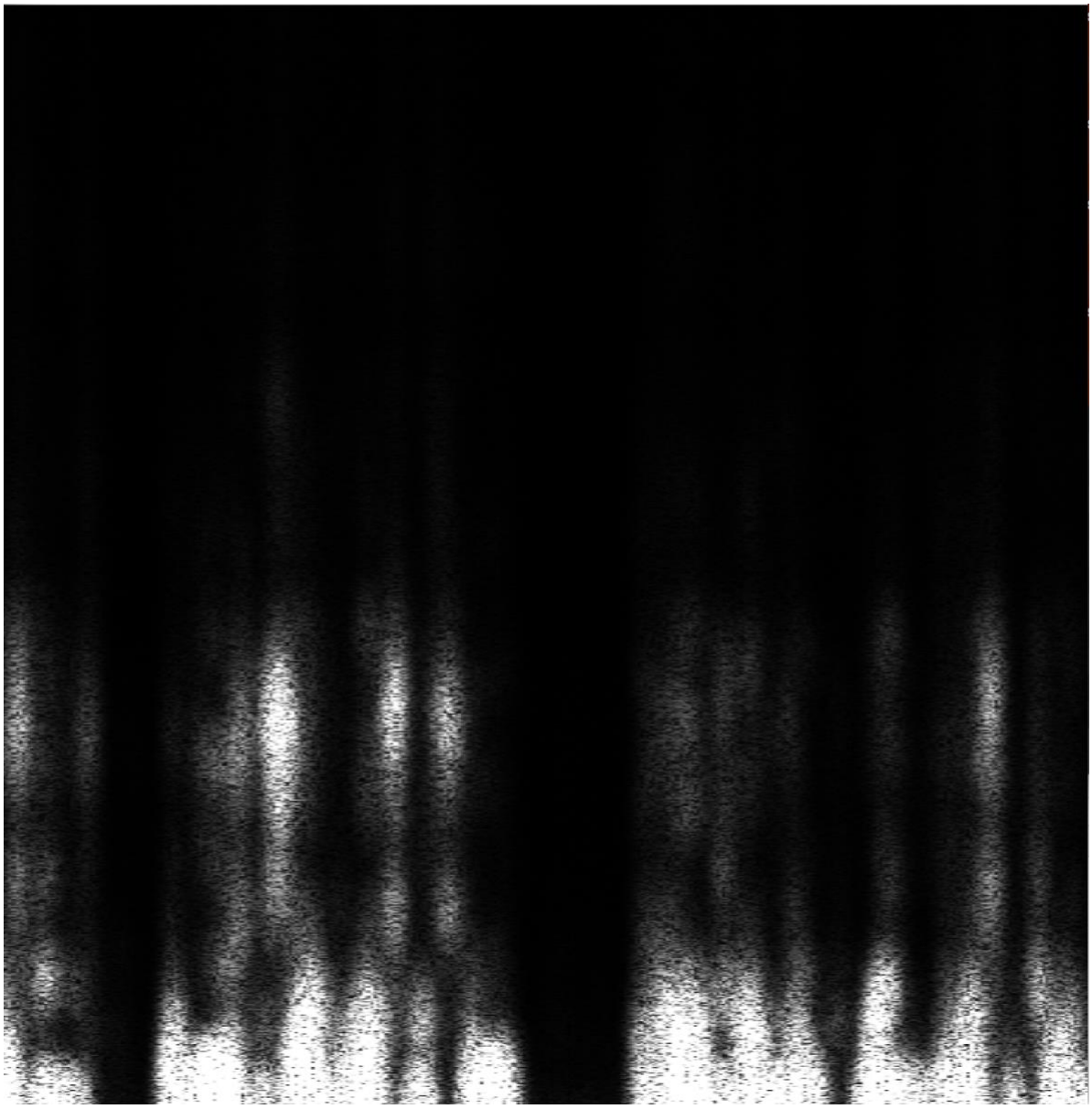

Figure 1.4.2.1 – Noise fractal applied to a spectrogram

'1.4.2 noise_fractal' builds off the structure of '1.4.1 gaussian_blur', with the addition of which noise in the masking process. '1.4.2 noise_fractal' is ideal for users that want to capitalise on the softening effect of the Gaussian blur, with an added dryness of static noise. In the main window of this patch, a matrix is filled with random values in 'jit.noise' (the dimensions of which are set to the maximum matrix size in this project: 1000x1024) and refreshed every 100 milliseconds by a 'qmetro'. This matrix is then combined with our original DSTFT matrix processed with Gaussian blur in the multiplication shader file 'op.mult.jxs' in 'jit.gl.slab gaussian_shader', shown in Figure 1.4.3.

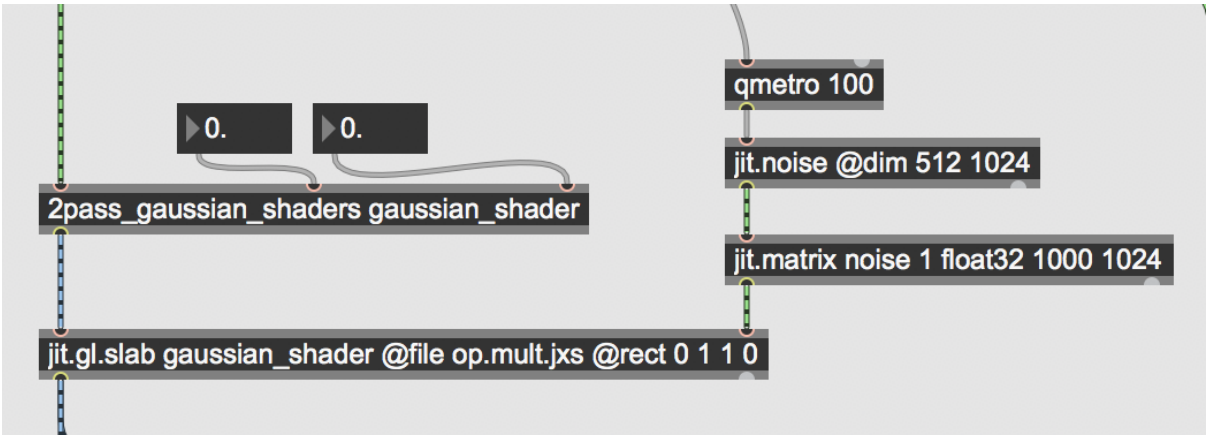

Figure 1.4.2.2 – 'jit.noise' multiplied with Gaussian blur matrix in shader file 'op.mult.jxs' in 'jit.gl.slab gaussian_shader' in the patch main window

1.5.1 blur_water

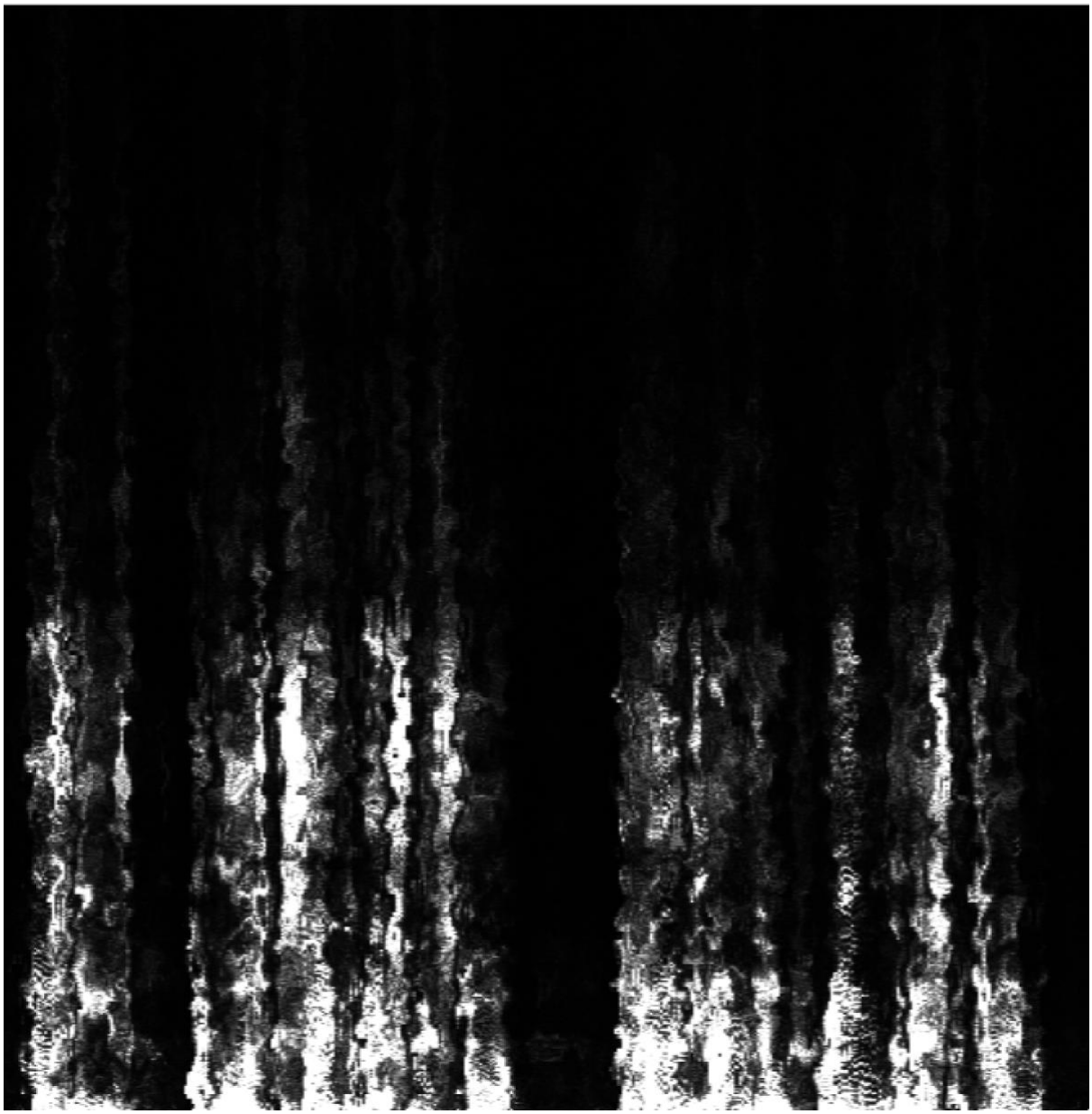

Figure 1.5.1.1 – Blur water effect applied to a spectrogram

'1.5.1 blur_water' is an effect for those who wish to achieve a muffled distancing phenomenon in their processed audio. This effect is extremely effective at making the sound source and the DSTFT data sound and look like they are below rippling water. In a similar vein to spectral effect 1.4.1 and 1.4.2, this effect is achieved through up-sampling a matrix that is being filled with random values from 'jit.noise'. In Figure 1.5.1.2 inlets 3 and 4 set the dimensions of said matrix at (1), then the matrix is split and scaled by 2^n at (2), where n is the 'interpbits' factor being passed in through inlet 2. Setting interpbits to a higher value firstly increases fractional repositioning in the 'jit.repos' object at (3), but also increases the factor by which the matrix at (1) is automatically scaled, thereby proving itself to be one way the user can increase the harshness of the blurring effect. Additional scaling if offered through inlets 5 and 6, before the two scaled white noise-filled matrices are packed and interpolated into 'jit.matrix long_noise'; the fact they are interpolated via '@interp' 1 here is key, as this interpolation ensures smooth edges on the spectrogram.

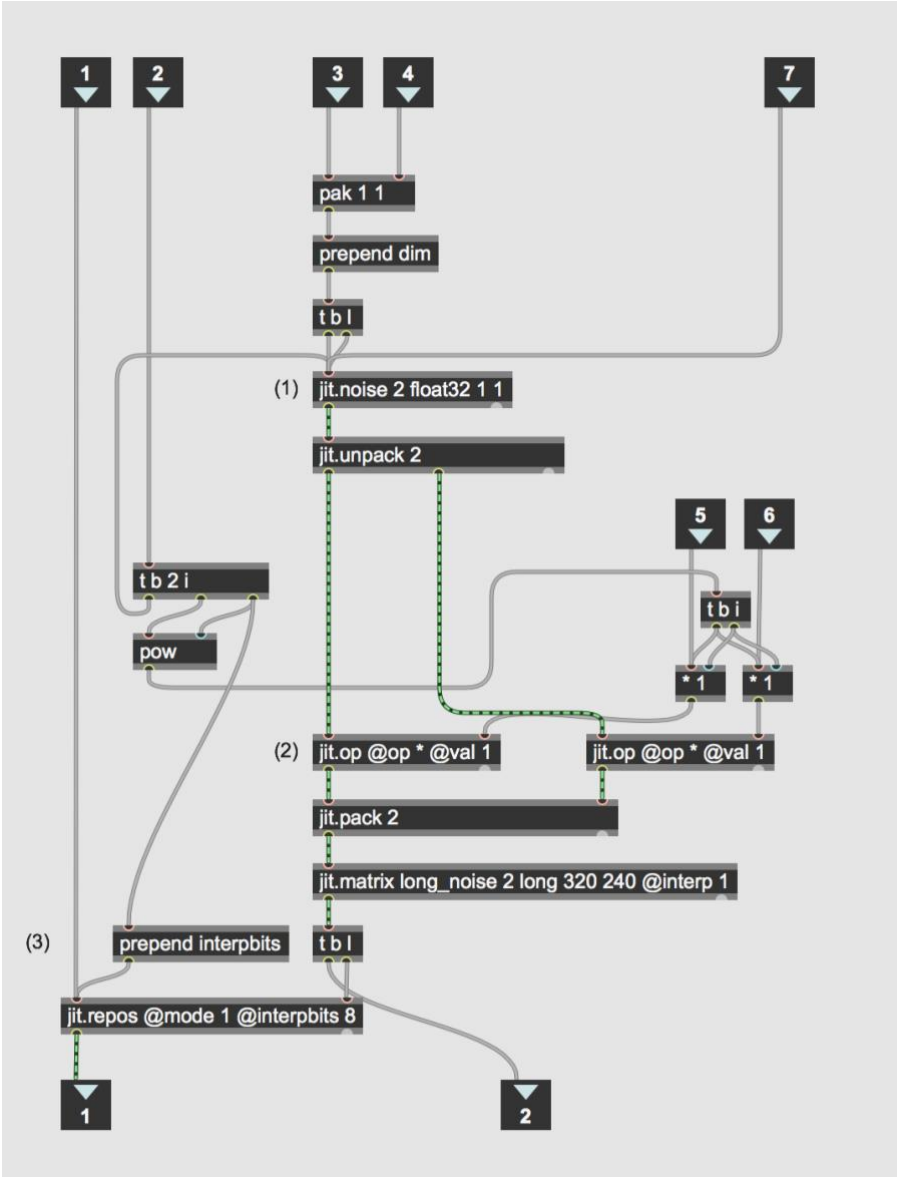

Figure 1.5.1.2 – Blur water effect applied to a spectrogram

1.5.2 time_scramble

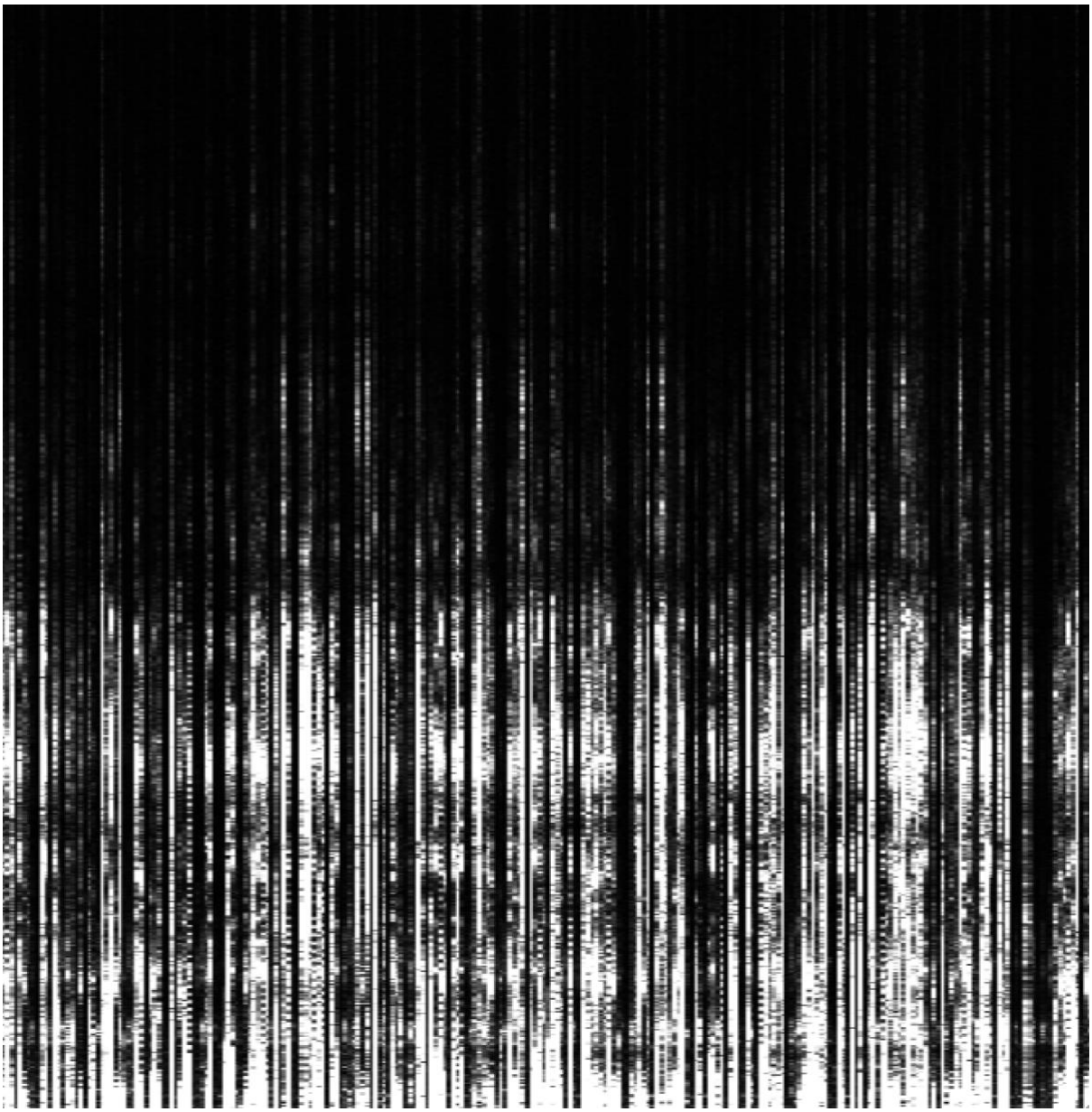

Figure 1.5.2.1 – Time scramble effect applied to a spectrogram

Spectral effect '1.5.2 time_scramble' is an effect whereby the DSTFT matrix frames are randomly rearranged at a user-defined rate. This effect completely disintegrates any coherence in the audio, and unless a subtle custom or small rectangular mask is applied this effect will scramble the audio beyond recognition. This effect should be used if the user wants a section of their sample to be unrecognisable, and it is interesting to see how when all samples are scrambled in such a fashion they all begin to sound similar.

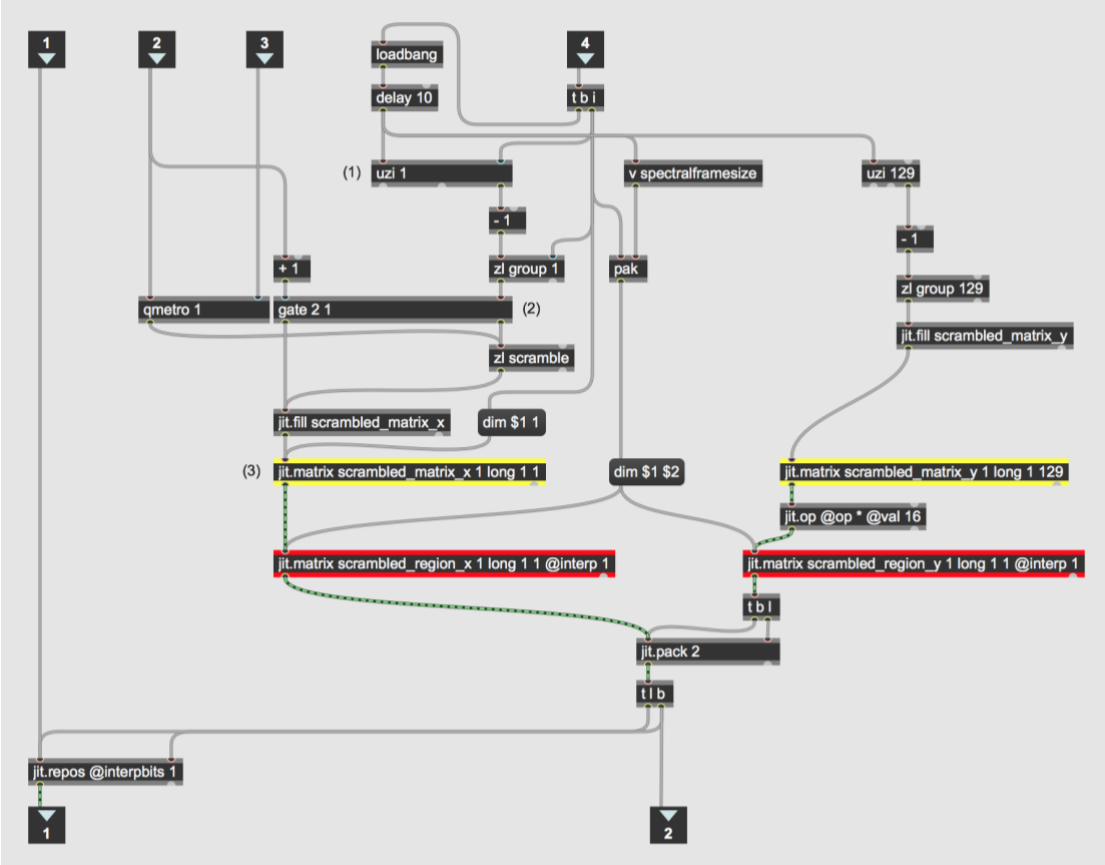

Figure 1.5.2.2 – 'time_scramble_process' in main patch window

Inside the 'time_scramble_process' abstraction in the patch's main window, similar to spectral effect 1.5.1, our spectral process relies on the 'jit.repos' object. To create the scrambling repositioning matrix that is combined with our original DSTFT matrix at the bottom left of Figure 1.5.2.2., we need to create two planes representing the x-axis repositioning and y-axis repositioning respectively. To scramble the frames in this matrix, an 'uzi' object counts from 0 to the number of frames in the sample and this count is grouped in the object 'zl group'. If the time scramble toggle is enabled in the main patch window, this list of values is passed through the 'gate' at (2) and then randomised in 'zl scramble'. These values are then sent into a 1-dimensional matrix. As we are not repositioning the y-values of the matrix, an equivalent process happens down the right-hand side of the abstraction, but the values are not scrambled. The scrambled x-axis values and untouched y-axis values are packed together into 2 planes, and then this 2-plane matrix is used to reposition our original DSTFT matrix.

1.5.3 spectral_smear

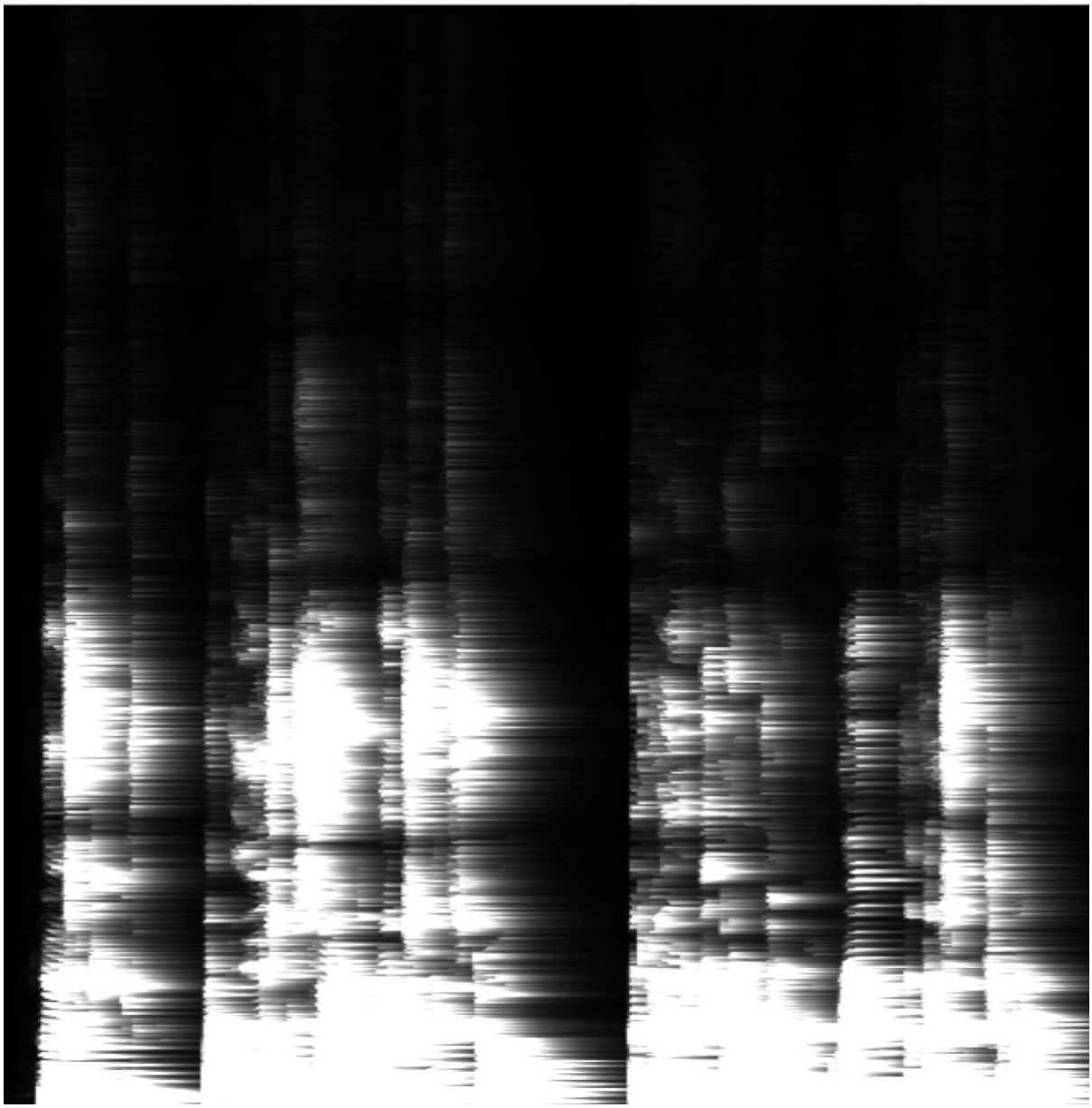

Figure 1.5.3.1 – Spectral smear effect applied to a spectrogram

Spectral effect 1.5.3 is simplest effect of all the those offered in this project to implement in Max because the Jitters 'jit.scanslide' object's spatial envelope creation, and its perfect suitability for implementing such an effect. After applying the Spectral smear effect to their DSTFT matrix, users will notice a trail proceeding the peaks in their spectrogram. The effect is similar to applying excessive amounts of reverb to the sample.

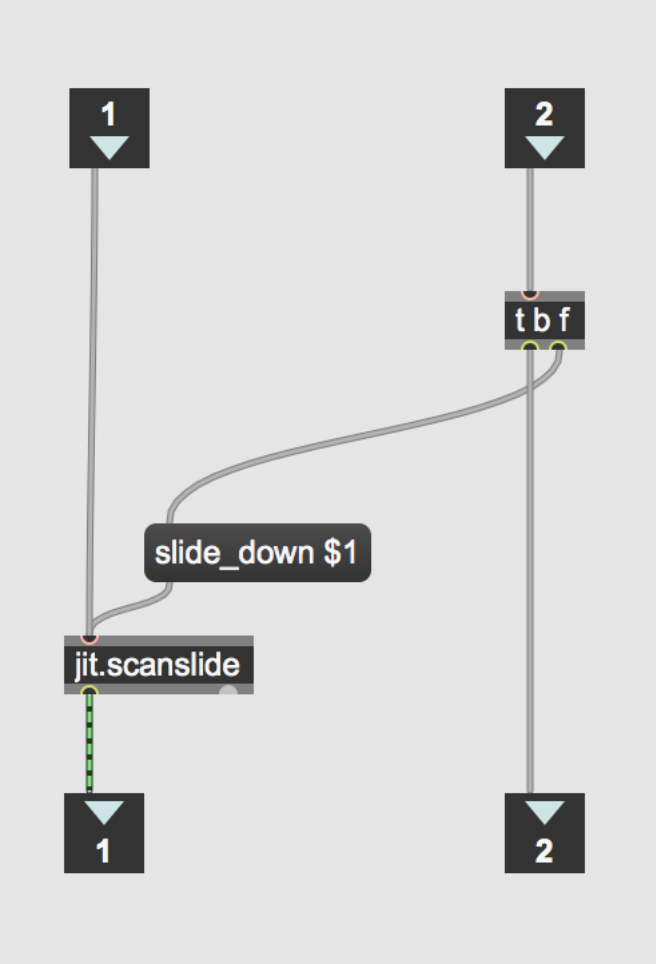

Figure 1.5.3.2 – 'spectral_smear_process' abstraction

1.6.1 frequency_warp

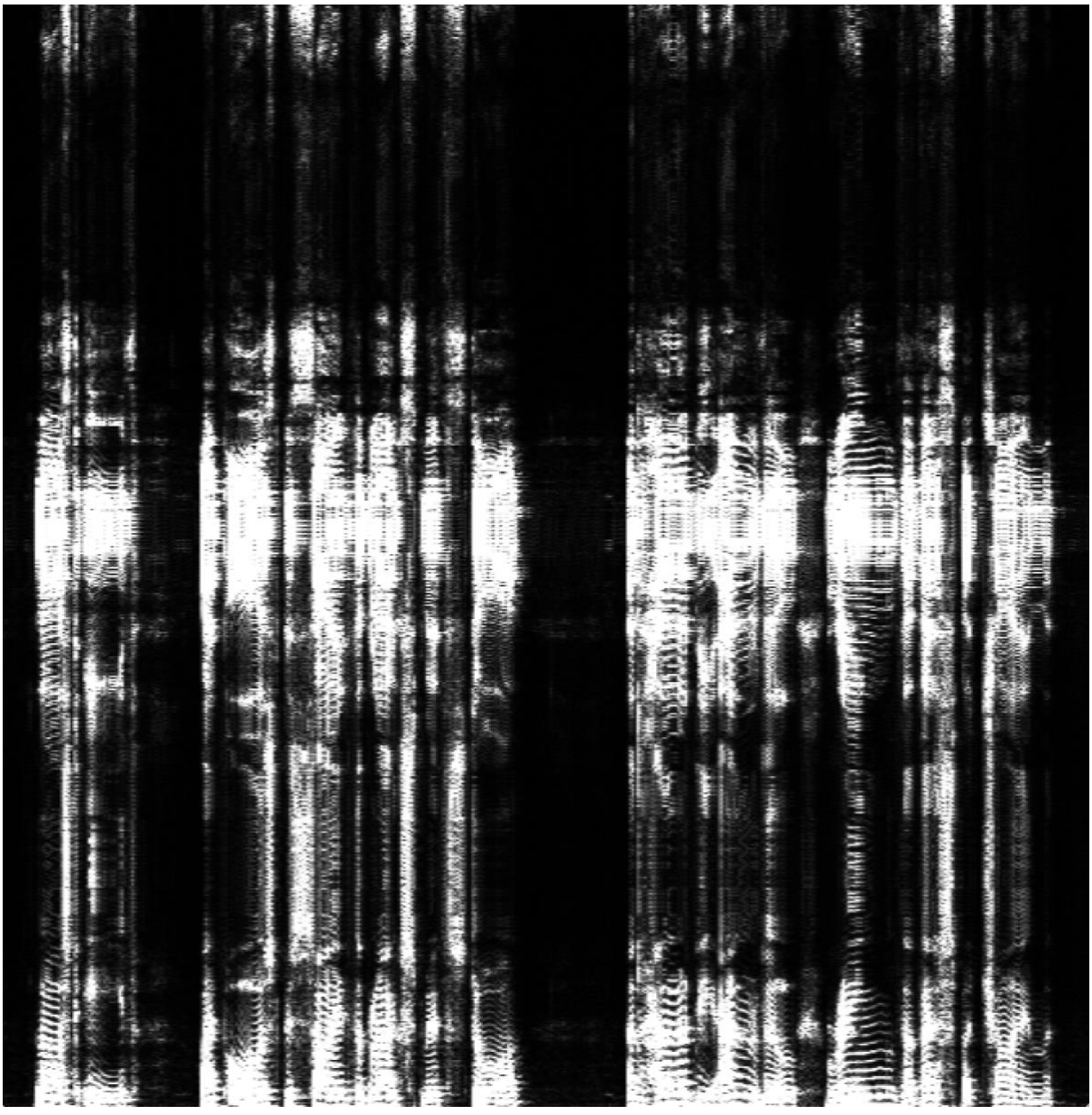

Figure 1.6.1.1 – Frequency warp effect applied to a spectrogram

Effect '1.6.1 frequency_warp' is the first spectral effect to offer the user control over individual parts of the frequency spectrum. This effect enables the user to displace the clustering of frequency between different frequency bins. This is a useful tool for users who want to maintain the temporal quality of their sample while experimenting with how altering the clustering of frequency in the sample will affect how the listening understands audio.

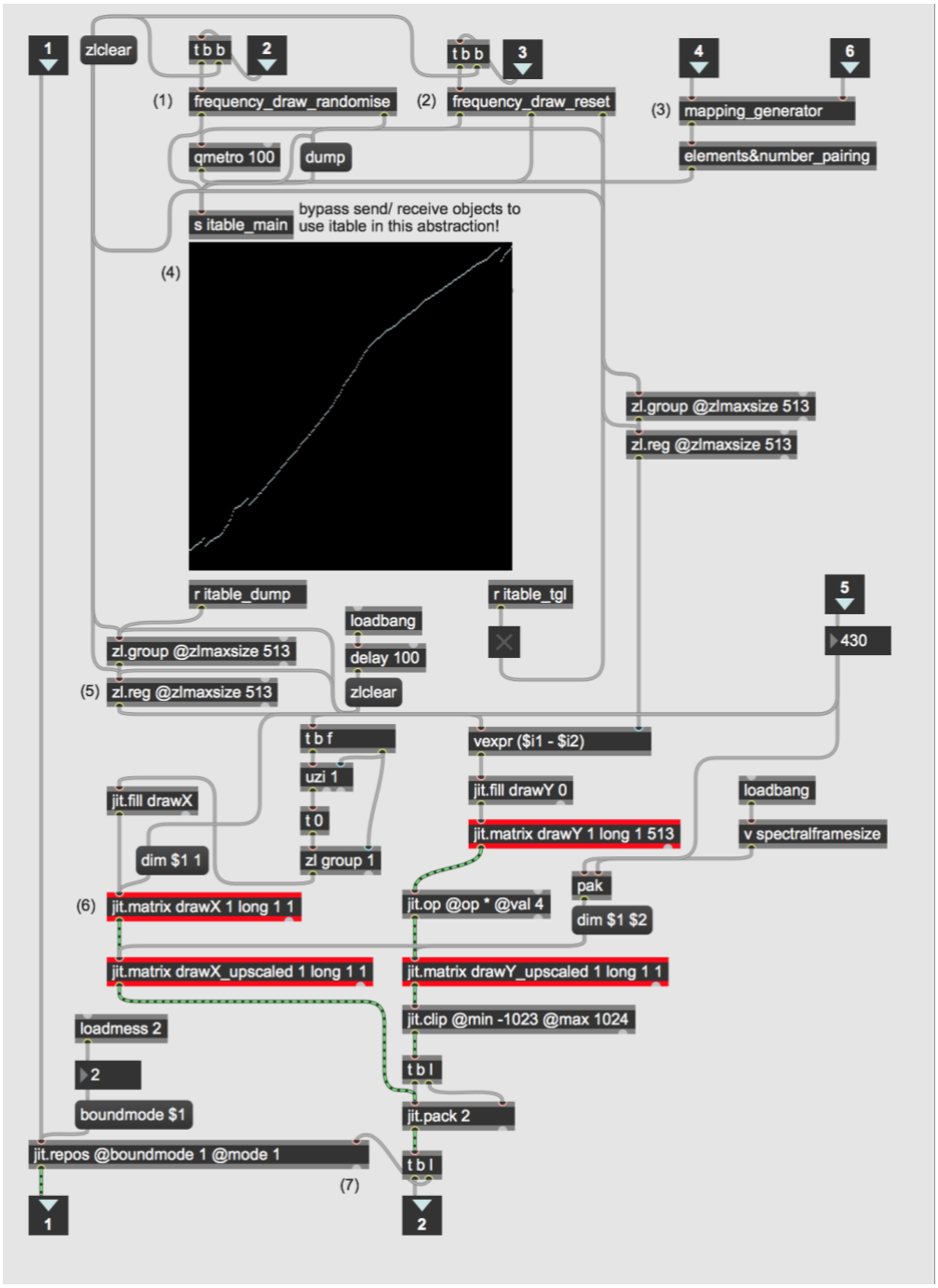

Figure 1.6.1.2 – 'frequency_warp_process' abstraction main window

Figure 1.6.1.2 depicts the main window of 'frequency_warp_process'. Once again our spectral manipulation process hinges on the 'jit.repos' object and in a similar fashion '1.5.2 time_scramble', we are combining a repositioned matrix with our original DSTFT matrix in order to warp its frequency distribution. The GUI of the abstraction in Figure 1.6.1.2 is an 'itable' object at inside the bpatcher 'frequency_draw' at (4), which provides a visual display of a table with which the user can interact. Through a series of 'if' statements inside the bpatcher, a toggle object is enabled only when the user is clicking on the window. This toggle enables a metro which dumps the information of the table out every 100 milliseconds. Above the 'itable' at (4) in Figure 1.6.1.2, are three abstractions which fill the 'itable' with information if the user does not wish to do it themselves, which were inspired by Tadej Droljc.[14] At (1) in 'frequency_draw_randomise' an 'uzi' object successively pairs x-values with random y-values which are then sent into the 'itable', so that the horizontal distribution of the matrix is maintained but the vertical distribution is randomised. At (2) in 'frequency_draw_reset' the same process happens as inside 'frequency_draw_randomise', apart from the y-values are not randomised. Abstraction 'mapping_generator' at (3) generates either an exponential or logarithmic curve similar to process to that in 'logarithmic_mapping' in section 1.2.2 and then pairs the curve data with unaltered x-axis data inside abstraction 'elements&number_pairing'. If the user chooses to draw on the 'itable' themselves, the data is sent out in x and y pairs as standard.

At (5) in Figure 1.6.1.2 the dumped information from the 'itable' is grouped together and then inverted in the 'vexpr' object before being sent into 'jit.matrix drawY' and clipped to be in the range of the spectral frame size. Meanwhile at (6), a number of 0s corresponding to the number of frames are sent into 'jit.matrix drawX', so that the x-axis of the matrix is not scaled. These two matrices are then packed together and use the repositioning matrix with our original DSTFT matrix in 'jit.repos at (7).

[14] Droljc, 'STFT Analysis', 152-157.

1.6.2 spectral_rolling_pin

Figure 1.6.2.1 – Spectral rolling pin effect applied to a spectrogram

'1.6.2 spectral_rolling_pin' is a spectral effect in which the user can manually disperse the frequency groupings in their spectrogram. This effect offers the most direct interactivity with the spectrogram of those in this project, insofar as the user can literally drag the spectrogram into any shape they want. The possibilities of this effect are quite endless but it is perhaps most useful for the process of spectral modelling whereby the user can mould the spectrogram of one sample into the shape of another. Perhaps if the user has the spectrogram of a sample of thunder in mind, they could model a spectrogram of speech into the same shape as the thunder crashing. Unlike effects before this, both the frequency clustering and temporal information of the DSTFT matrix can be altered, meaning that the user can alter the sampler beyond recognition in some regions while maintaining the integrity of others.

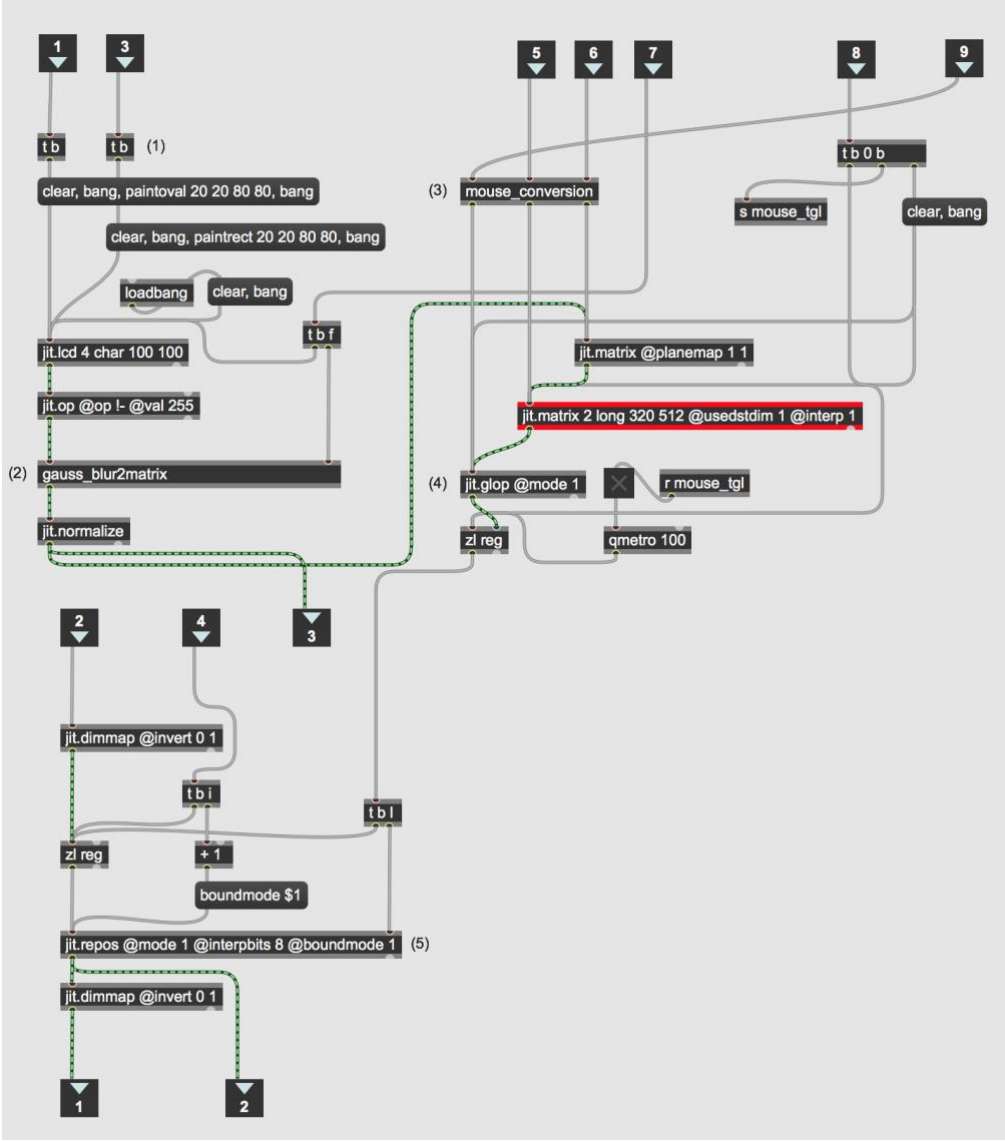

Figure 1.6.2.2 – Abstraction 'spectral_rolling_pin_process'

For this spectral effect, I was inspired again by the work of Tadej Droljc.[15] The crux of this effect is based on the object 'jit.glop', which feeds the matrix back onto itself with optional gain control for the process. A similar effect can be achieved through the + operator of 'jit.op', but 'jit.glop' is particularly useful in this context because feedback direction can be specified (i.e. up, down, left or right). The process works in three stages: first the rolling pin head is rendered, then its feedback trail is saved in the 'jit.glop' object and finally the output of jit.glop is used as a repositioning matrix in a 'jit.repos' object; this process is depicted in Figure 1.6.2.3.

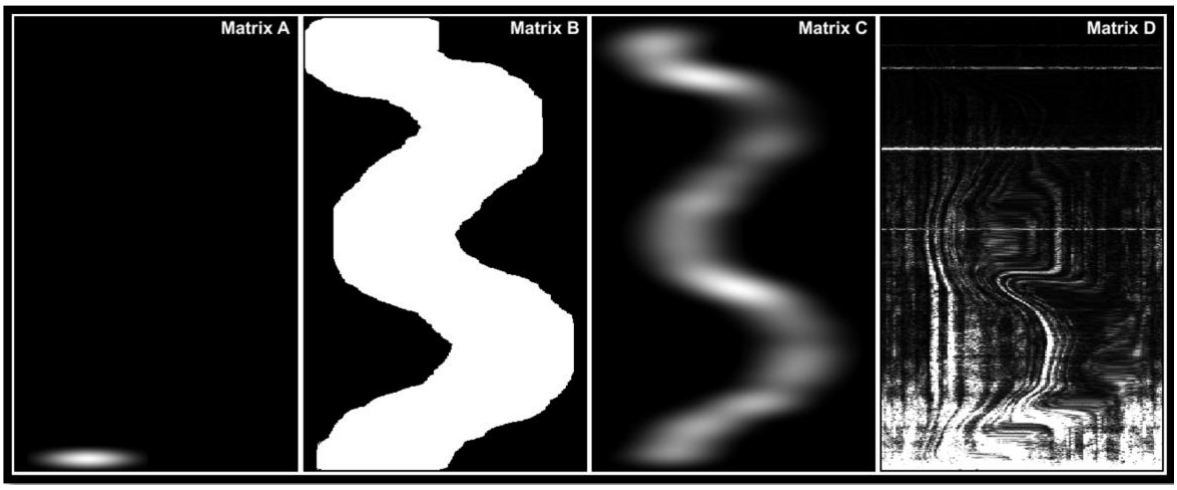

Figure 1.6.2.3 – Spectral rolling pin effect overview

Matrix A in Figure 1.6.2.3 depicts the rolling pin head, Matrix B shows the trajectory of this rolling pin's feedback in the matrix, C is the normalised version of B and Matrix D shows the effect of using Matrix C as a repositioning matrix for a DSTFT matrix in the 'jit.repos' object. At (1) in Figure 1.6.2.2 are the messages used to draw either a rectangle or circle pin head into the 'jit.lcd' below, the output of which is inverted and sent into abstraction 'gauss_blur2matrix' at (2). Inside 'gauss_blur2matrix' the rolling pin head has a user-defined amount of Gaussian blur applied to it through the same process as in section 1.4.1. The values in this Gaussian blurred pin head matrix are then normalised, and then mapped to plane 1 in a 'jit.matrix' (this need to be done at the output of 'jit.lcd' is always in 4 planes). Abstraction 'mouse_conversion' at (3) performs two functions: it monitors the direction the cursor is travelling and outputs gain direction that is sent into 'jit.glop' at (4), and also outputs 'dstimend' and 'dstdimstart' messages for the two 'jit.matrix' objects below. This ensures the pin head is correctly positioned in the 'jit.matrix' (which is set to the dimensions of the jit.lcd in this spectral effect's main window) above 'jit.glop' at (4). The output of 'jit.glop' is then used as the repositioning matrix in 'jit.repos' at (5). Here I additionally offered the user the option of changing this 'jit.repos' object's boundmode in the main window as I wanted to ensure the user had maximum control over how their pin head affects the DSTFT matrix.

[15] Droljc, 'STFT Analysis', 165-169.

1.6.3 filter_mask

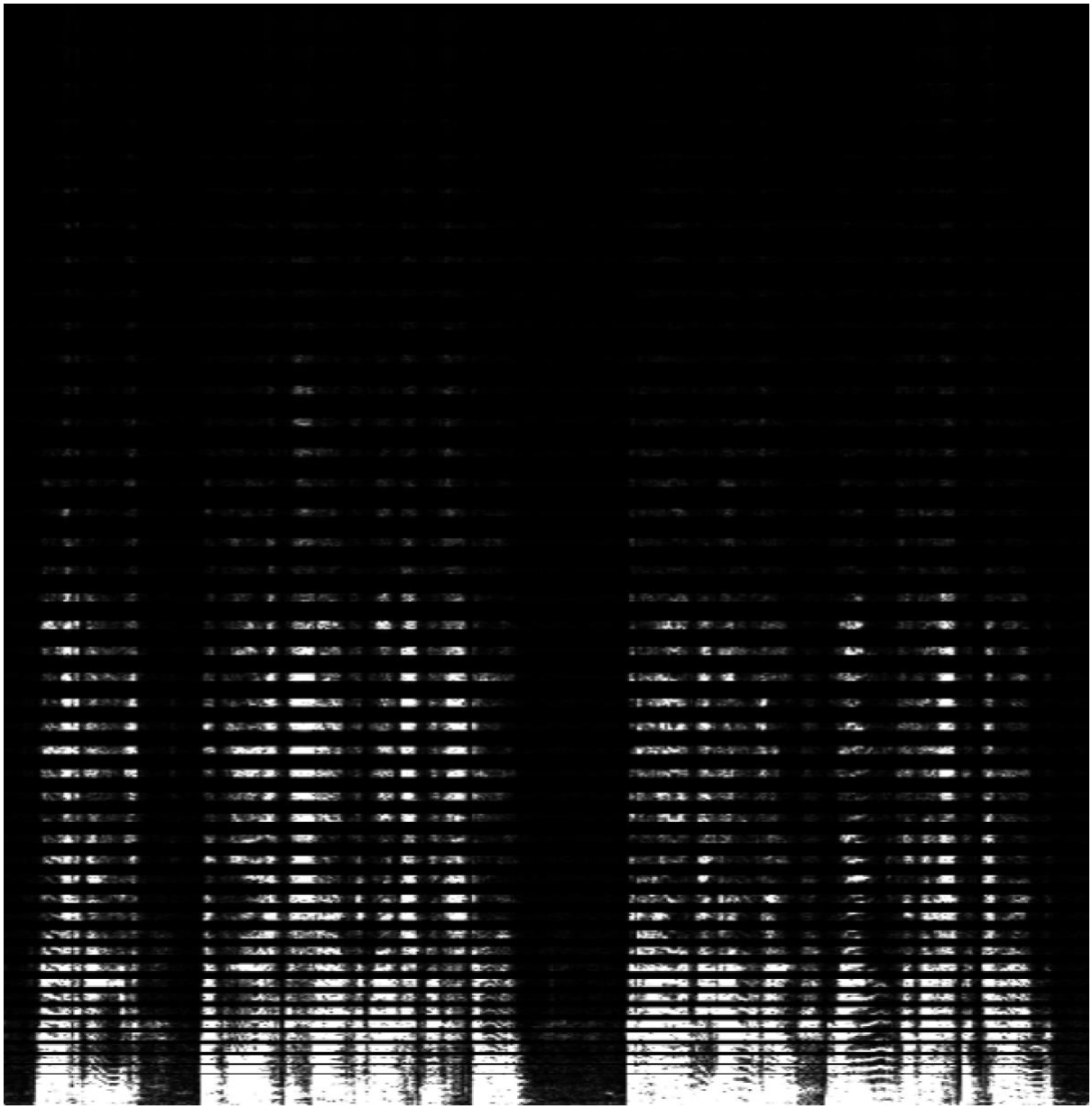

Figure 1.6.3.1 – Filter mask effect applied to a spectrogram

Spectral effect '1.6.3 filter_mask' offers the user with the functionality to apply three different masking effects to their spectrum. This effect is ideal for users who want to subtly change the timbral quality of their audio, but wish to do so with more control over the process than in spectral effects 1.4.1 - 1.5.3. This effect is not built as one which will completely change the audio sample beyond recognition (this can be done, but at that point the effect is being used beyond what it was designed for), but rather as a sensitive veiling effect whose target frequencies and distributions can be altered.

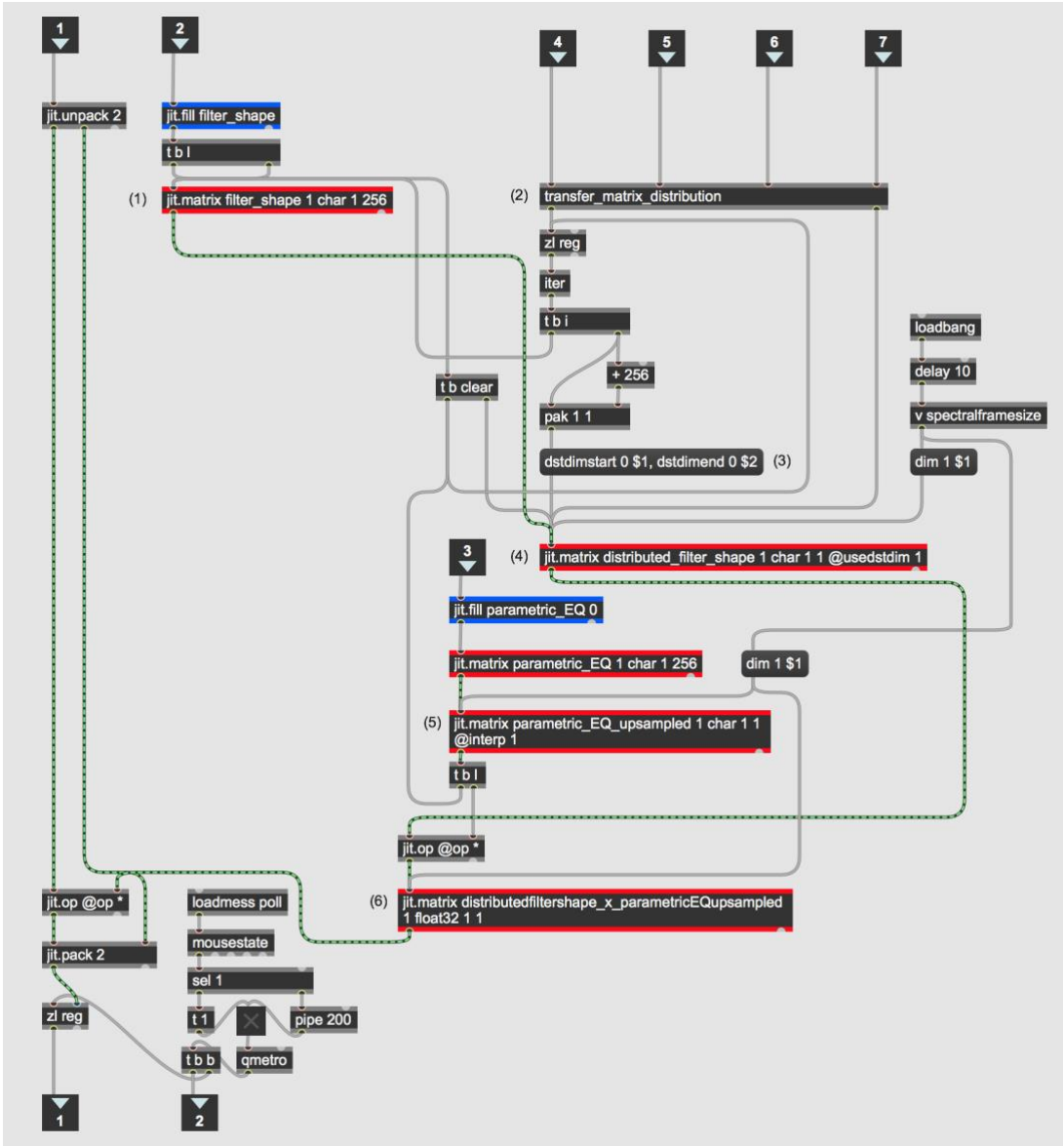

Figure 1.6.3.2 – 'filter_mask_process' abstraction main window

This spectral effect operates on the basis two masks ('Band pass filter' and 'Secondary EQ') and a distribution filter for the band pass filter. The band pass filter shape drawn by the user is input through inlet 2 in Figure 1.6.3.2 and this data is used to fill 'jit.matrix filter_shape' at (1). Through inlets 4, 5, 6, and 7 the abstraction receives the number of filters, exponent of the distribution, the fundamental operator of the distribution, and the distribution interpolation factor and these values are sent into abstraction 'transfer_matrix_distribution'.

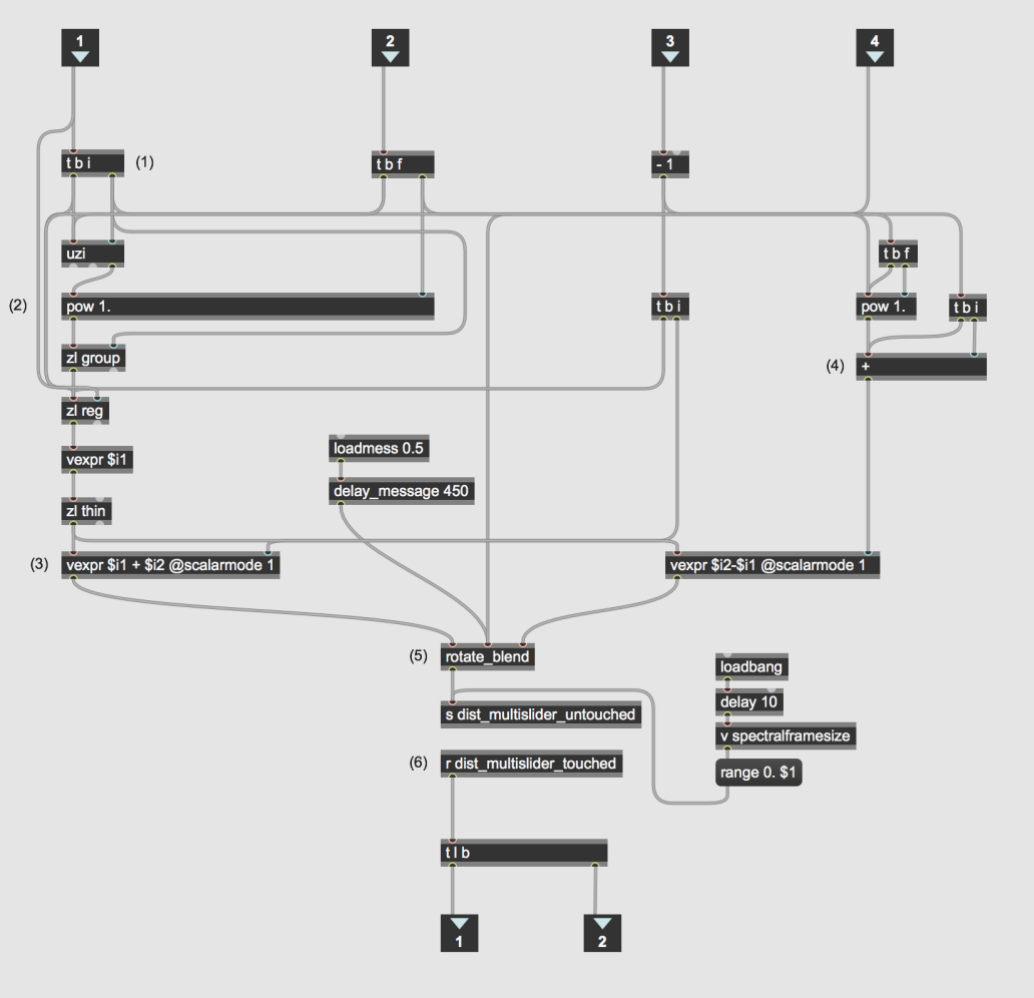

Figure 1.6.3.3 – 'transfer_matrix_distribution' abstraction main window

Figure 1.6.3.3 shows the number of filters coming in through inlet 1 at (1) and being multiplied by the exponent coming in through inlet 2 at (2). These values are then set as integers added to the user-specified fundamental at (3). Down the right-hand side of the patch at (4), an opposite exponential distribution is created: the abstraction 'rotate_blend' at (5) allows the user to interpolate between these two opposing lists, with the interpolation factor being adjustable through the value input through inlet 4 in Figure 1.6.3.3. What 'transfer_matrix_distribution' is doing is setting how the band pass filters at (1) in Figure

1.6.3.2 are distributed over the frequency spectrum. This means that if an exponential curve is drawn in the distribution filter, then the band pass filters will be more active at the lower end of the spectrum, gradually becoming more dispersed towards the top of the spectrum.

In Figure 1.6.3.2 the values output from abstraction 'transfer_matrix_distribution' at (2) are grouped together into lists of length 256 (list lengths higher than this begin to slow the spectral effect's processing speed) and then broken up again. This is done so that the 'zl reg' object list is grouped and banged when a change is made by the user in the band pass filter. These values are successively sent as the y-value of a 'dstdimstart' message and the y-value + 256 of a 'dstdimend' end message. This is setting individual start and end points for each iteration of the band pass filter. At inlet 3 the values from 'Secondary EQ mask' are input and up-scaled into the 'jit.matrix parametric_EQ_upsampled'. This 'jit.matrix' is used to scale the distributed band pass filter in the 'jit.op' object below. The amplitude plane of our original DSTFT matrix is then scaled by this EQ-scaled, distributed band pass filter matrix. The resultant matrix from this operation is the final filtered 'jit.matrix' that is displayed in the 'jit.window'.

EQ

Should programmers wish to use any of the EQ abstractions in their own projects, be sure to route them after their spectral effect(s).

Compressor

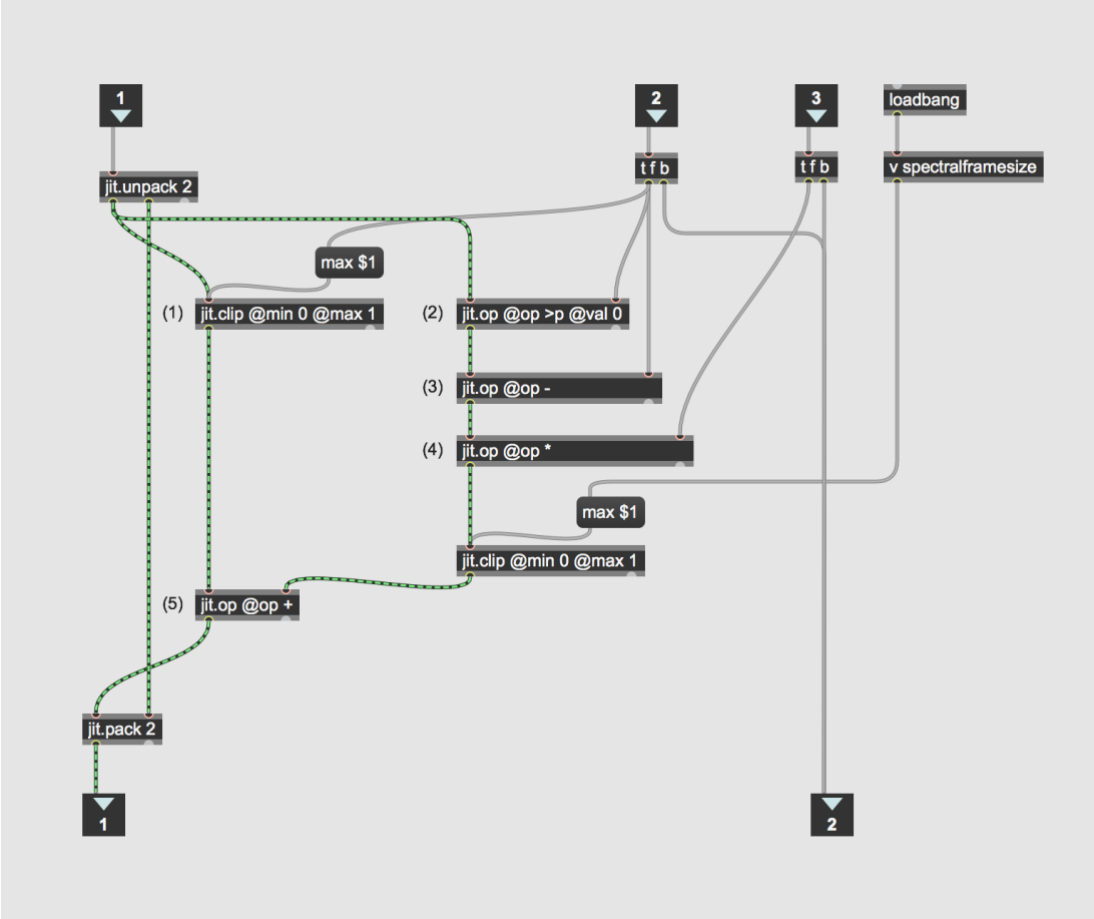

Figure EQ.1 – 'compressor' abstraction main window

Figure EQ.1 depicts the abstraction that encapsulates spectral compression in this project. Inlet 2 receives the user-specified threshold of the compressor, and this value is used to clip all values in the original DTSFT amplitude plane at (1). At (2) this 'jit.op' object only lets amplitude values passed if they are above the user specified threshold and then the difference between the amplitude of the matrix and the user-specified threshold is multiplied by the user-specified ratio at (4), and the result of this is then added back onto the original amplitude plane. Say the threshold is 20 and the ratio is 0.5 and an amplitude in the matrix is 25, the difference between the threshold and the amplitude value (5) is multiplied by 0.5 (2.5) and added back to the original clipped amplitude plane, so the resultant amplitude will be 22.5.

Denoiser

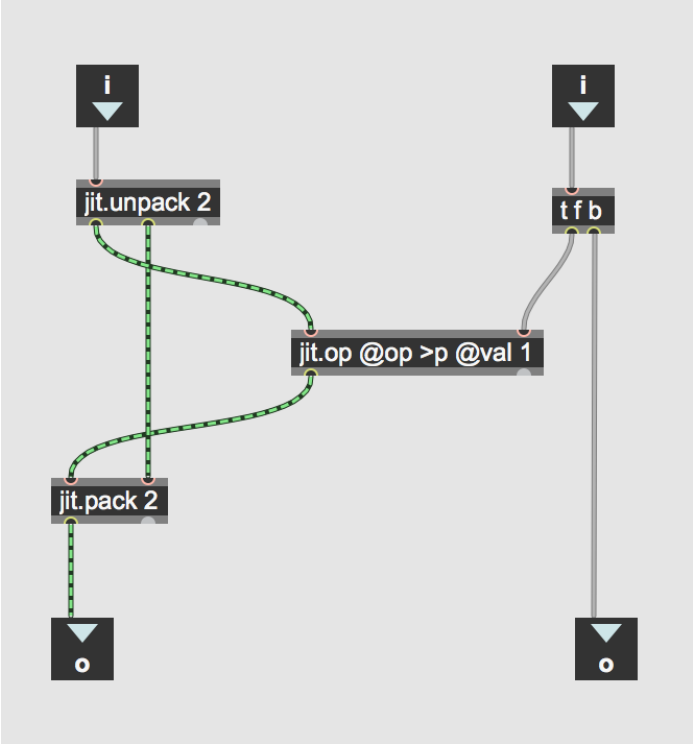

Figure EQ.2 – 'denoiser' abstraction main window

As with the compressor EQ, this object uses the 'jit.op' with the operator to limit which amplitude values can pass through. In this abstraction only the amplitude values that are below the user specified value can pass.

Limiter

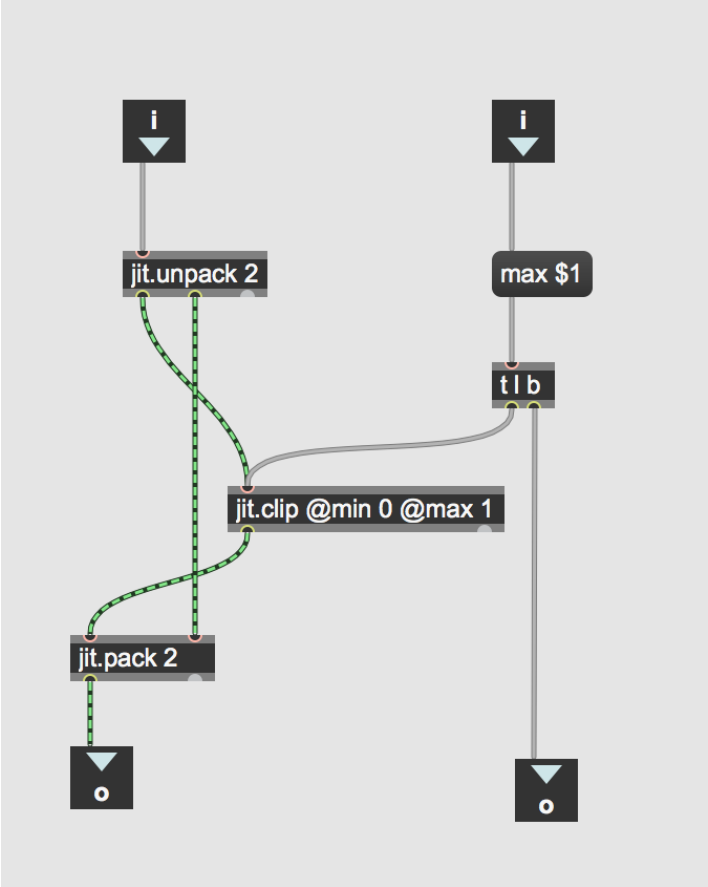

Figure EQ.3 – 'limiter' abstraction main window

The 'limiter' abstraction uses the other aspect of the compressor EQ that was not utilised in the limiter. In this abstraction a 'jit.clip' object allows values in the matrix plane of our DSTFT matrix to pass if they are below a user-specified maximum amplitude.

playback

The true frequency f_t calculation process outlined in section 'The Fourier Transform' of this paper in Figure FT.7 is encapsulated in the 'horizontal_sync_playback_pfft~' and 'no_horizontal_sync_playback_pfft~' patches in the 'playback' folder.

$$f_t = f_k + \Delta\emptyset \times \frac{S_r}{2\pi \times N}$$

Figure FT.7 – equation for calculating true frequency at bin index *k*

The equation in Figure FT.7 performed inside the 'frameaccum~' and 'poltocar~' objects in figure P.1, where the running phase output from 'frameaccum~' is used to convert the Polar coordinates of the amplitude and phase information back into Cartesian form accounting for frequency deviation. This information is then processed by the Inverse Discrete Short-Time Fourier Transform (IDSTFT) inside the fftout~ object, which outputs the reconstructed audio from the matrix.

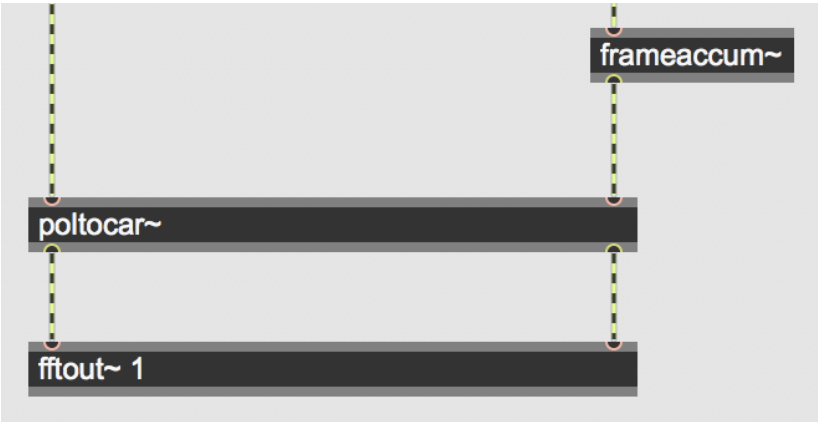

Figure P.1 – The true frequency calculation, Polar to Cartesian coordinate conversion and IDSTFT processes encapsulated in Max

To play back the sample, two operations have to happen: the processed DSTFT matrix needs to be read inside a 'pfft~', and a 'phasor~' is required to drive the speed at which this happens. Inside all 4 playback abstractions is the abstraction 'phasor~_playback_drive', which provides the playback speed as a signal and then sent into the individual playback abstraction's 'pfft~'. The 'phasor~_playback_drive' abstraction encapsulates the equation in Figure P.2, in which the playback rate is user specified, therefore *f* is dependent on the user's playback rate.

$$f = \frac{S_r}{N \times HopSize} \times playback\ rate$$

Figure P.2 – The equation for calculating the original playback speed of the sample

Figure P.3 shows the abstraction 'phasor~_playback_drive' which performs the operation in Figure P.2. Hop size is input through inlet 1, N through inlet 2, the user specified playback rate through inlet 3, the sample rate (S_r) through the 'sampstoms~' object in tandem with the '!/ 1000' math operator object.

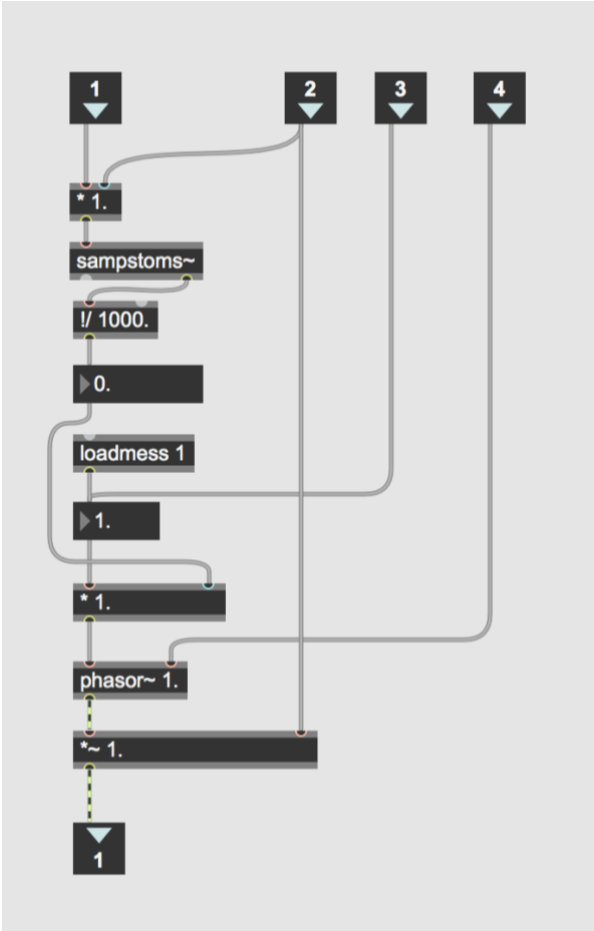

Figure P.3 – The 'phasor~_playback_drive' abstraction

simpleplayback

Inside the 'simpleplayback' abstraction the 'phasor~_playback_drive' in Figure P.3 sends the playback frequency *f* into the 'pfft~ horizontal_sync_playback_pfft~'. This value is also converted into the numerical domain by the 'snapshot~' object and either sent (through the 'v simple' object) to the 'circleclickpoint_7_line_alphablend' or 'line_alphablend' abstractions in the main window of whichever spectral effect is currently open to drive the red line playback point.

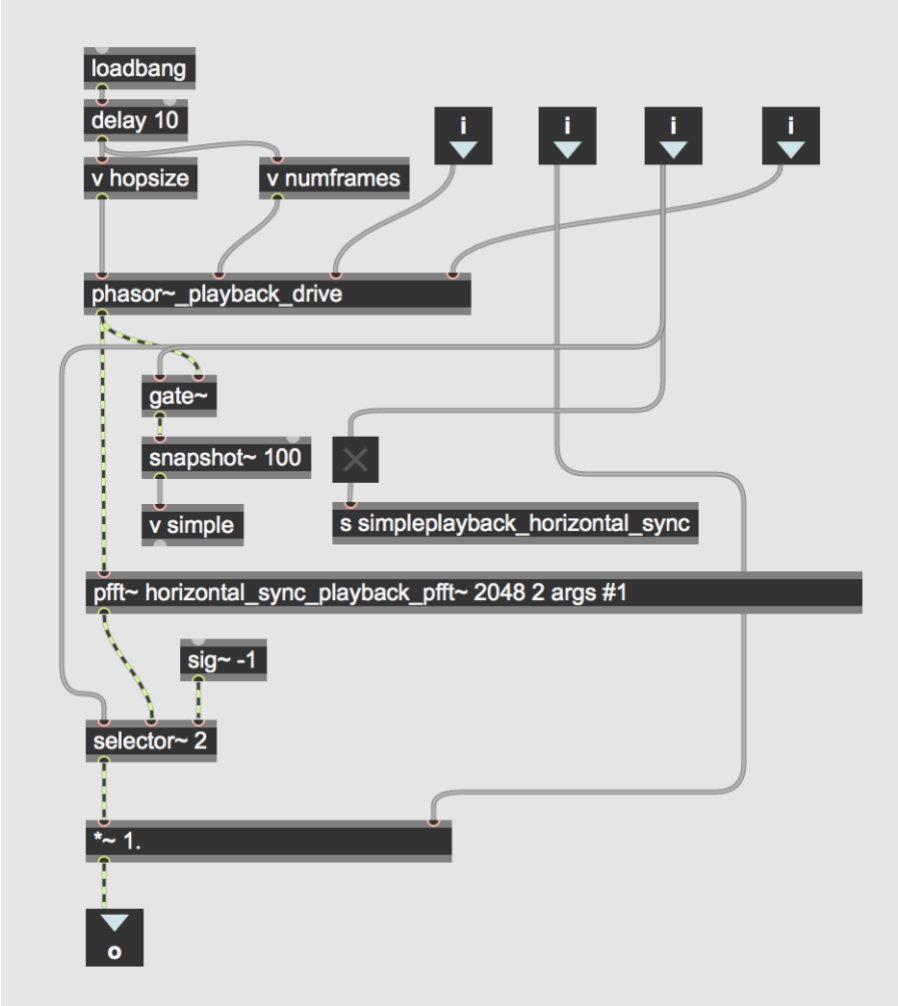

Figure P.4 – The 'simpleplayback' abstraction

The 'pfft~ horizontal_sync_playback_pfft~' is exhibited in Figure P.5. The playback frequency output from the 'phasor~_playback_drive' abstraction in Figure P.3 is output from the left outlet of the 'fftin~' object in Figure P.5 into the 'sah~'. The 'sah~' is held by the bin index output from the right outlet of the 'fftin~', ensuring that the two 'jit.peek~' objects below, reading the x and y values of the amplitude plane of the processed DSTFT matrix respectively are only driven in sync with the bin index. The signals output from the 'jit.peek~' objects are then converted into audio via the IDSTFT described at the beginning of this section.

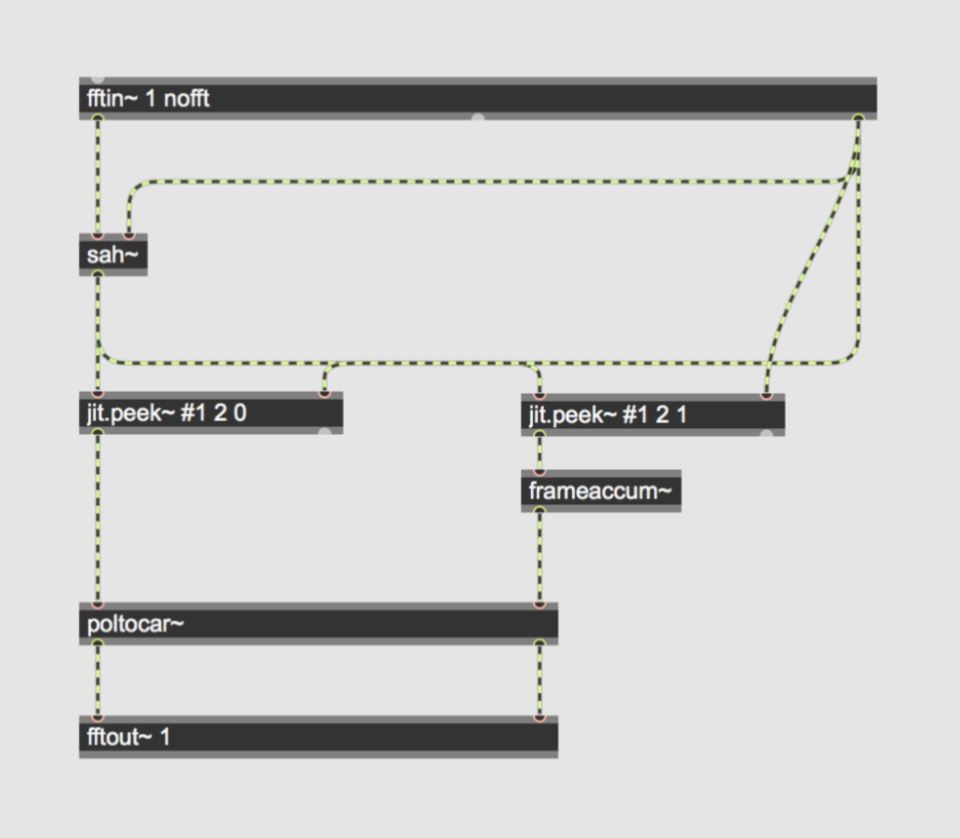

Figure P.5 – The 'pfft~ horizontal_sync_playback_pfft~' abstraction

stochasticblur_playback

The 'stochasticblur_playback' abstraction is identical to the 'simpleplayback' abstraction aside from the 'sah~' is removed in the pfft~, meaning that horizontal sync is no longer active when the 'jit.peek~' objects read from the processed DSTFT matrix. To accentuate this lack of sync hence creating horizontal blur stochastically, the user has the added option of increasing the scale of a 'noise~' object which is being added to the playback rate *f* signal output from 'phasor~_playback_drive'. Increasing the amount of white noise in the signal means that the 'jit.peek~' objects are more likely to read from the 'wrong' frame (increasing the blurring effect in subsequent signal reconstruction) when outputting values from the processed DSTFT matrix.

frame_interpolation_playback

As is made evident by the top half of Figure P.6, the abstraction 'frame_interpolation_playback' operates on the same principles as 'simpleplayback' and 'stochasticblur_playback'. What differentiates this abstraction from the previous two is its creation of a composite frame (that I refer to as *n*0.5) from frames *n* and *n* + 1. At (1) a 1x1024 size 'jit.submatrix' is created, whose offset (in relation to the processed DSTFT 'jit.matrix' is frame *n*, and at (2) the same operation happens aside from it's for frame *n* + 1. At (3) the 'jit.xfade' crossfades between frames *n* and *n* + 1 at a fade factor specified by the user. The resultant composite frame (*n*0.5) is passed through the 'poking_with_noise' abstraction, where a 1x1024 size 'jit.noise' matrix filled with random 1s and 0s is multiplied with frame *n*0.5, randomly filling the frame with 0s (the 'switch' object below allows the user to disable this effect if they desire'. The resultant matrix is then sent into 'jit.matrix interpolated_frame' and read by the 'pfft~' object below, with the 'sah~' horizontal sync removed.

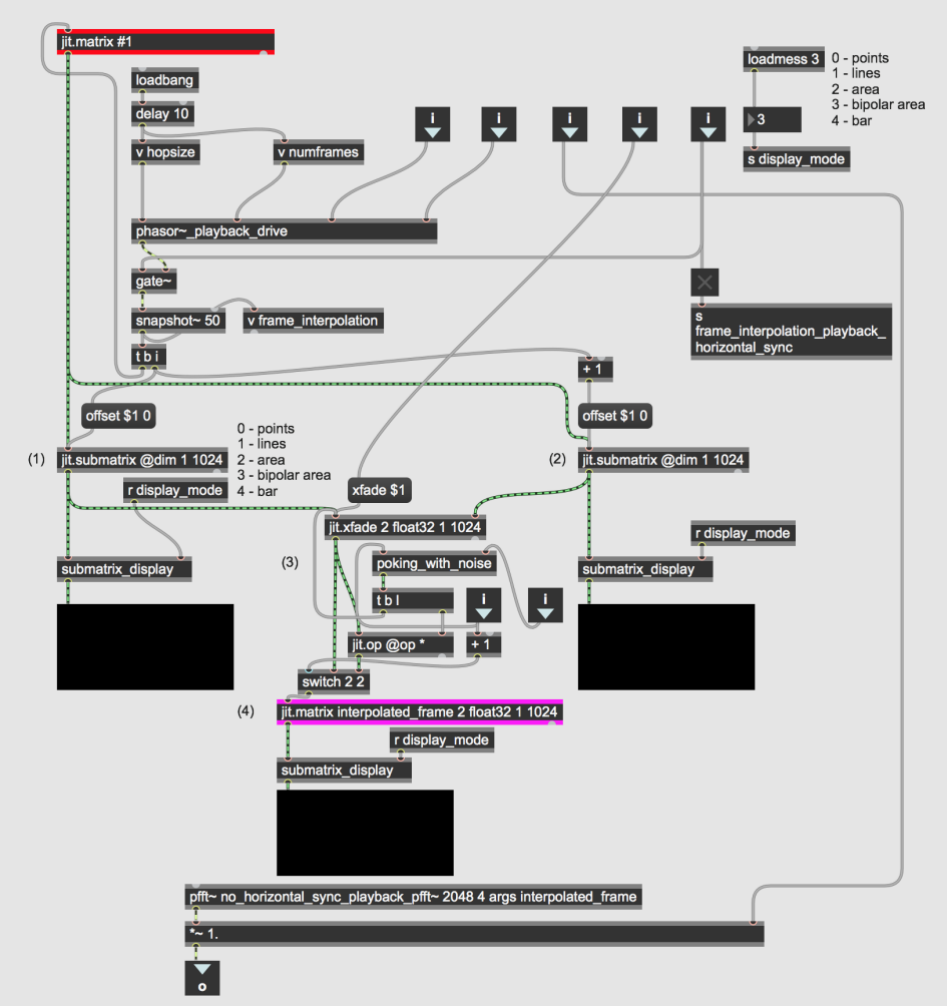

Figure P.6 – The 'frame_interpolation_playback' abstraction

transient_playback

This playback abstraction was inspired by the 'Transient Preservation' system outlined by Jean François Charles.[16] The abstraction is based on the principle of extracting the relative transient value (normalised from 0 to 1) of each frame n from $n - 1$, and using this value to affect the playback frequency f. Figure P.7 depicts the Max structure in which this happens.

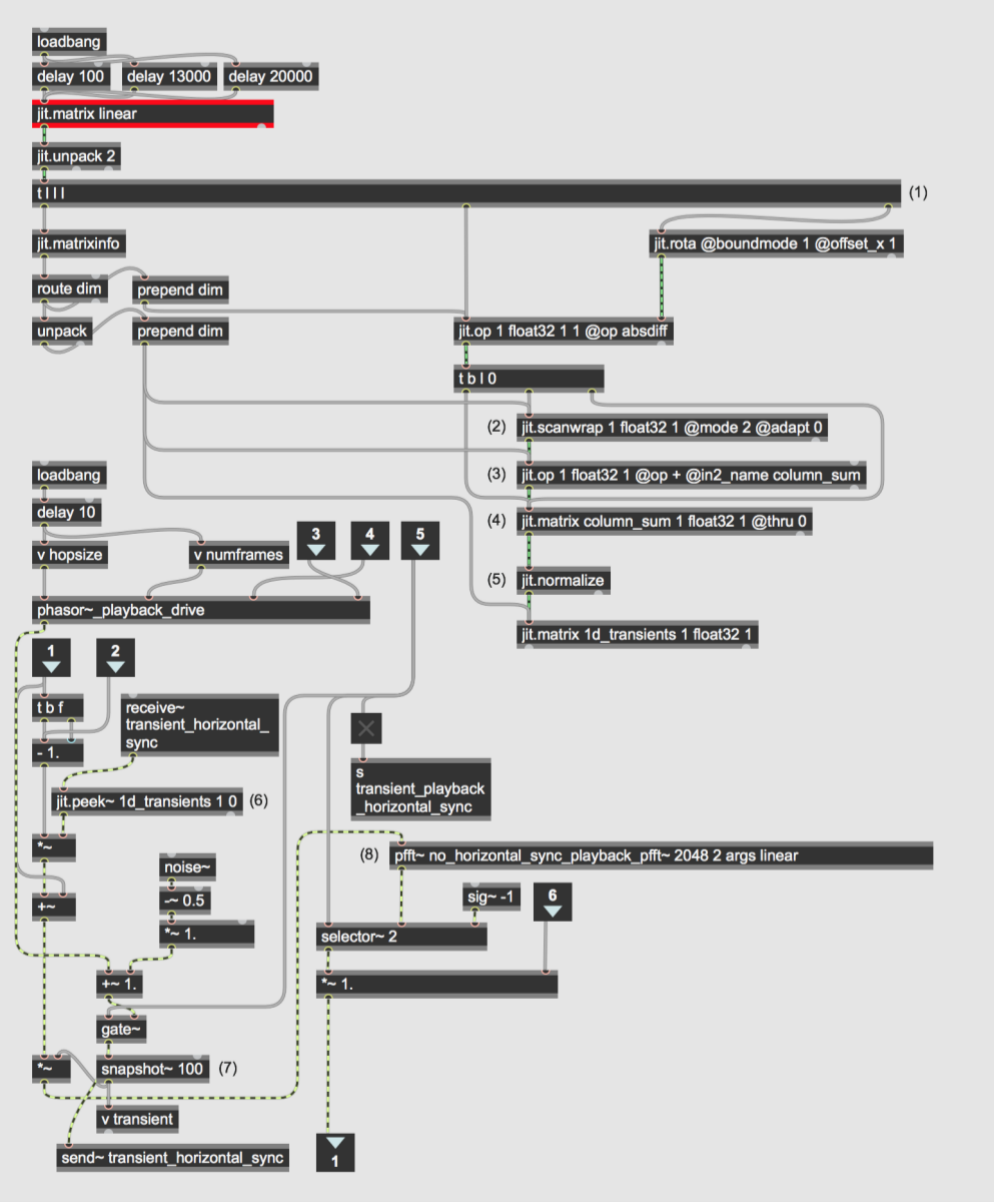

Figure P.7 – The 'transient_playback' abstraction

[16] Charles, 'A Tutorial', 94 -95.

At (1) from the right outlet of the 'trigger' object, our processed DSTFT amplitude plane is offset by 1 to become frame $n + 1$, and the absolute Euclidean transient distance between itself and frame n is calculated via the 'absdiff' operator in the 'jit.op' object below. At (2), this matrix is wrapped into a 1 column width with the 'jit.op' object at (3) taking the running sum of this transient absolute difference before being collected in 'jit.matrix column_sum' at (4). At (5) the matrix values are normalised so that the transient curve is visible, before this matrix is read by 'jit.peek~' at (6). For calculating playback speed, this formula is outlined by Charles:

$$r_{inst} = r_{stat} + tr \times (r_{trans} - r_{stat})$$

Figure P.8 – Equation for calculating instantaneous playback speed

The equation in P.8 is performed (and kept in horizontal sync via the 'snapshot~' object) in the multiplication and addition objects between (6) and (7). The value r_{inst} from this final multiplication object is sent to the 'pfft~ no_horizontal_sync_playback' abstraction where the processed DSTFT matrix is read via two 'jit.peek~' objects at the frequency r_{inst}.

User Experience

Coming to the end of building the project, I began to think about how I could make it simple to use, yet still offer all the functionality I had built into it. Max's 'presentation mode' feature has proven extremely useful here, enabling me to hide certain parts of the patch from the user that I didn't deem necessary for them to see. Constructing a Graphical User Interface (GUI) through this presentation mode was only the beginning of attempting to make this software as user friendly as possible. The modular approach I have taken to building this project has also come in useful when thinking of how a first time user will experience it. The numbered structure of each folder in the 'Max' folder of this project is an attempt by myself to make clear the order through which the user should navigate this project. After organising the interface of each patch in the project, I have also included instructions and information sub-patches in patches I felt that the user could benefit from added guidance.

In order to assess how user-friendly the project was, in April 2019 I organised two sessions with two users at disparate ends of the Max experience spectrum. In these sessions, one of which was with a university lecturer who specialises in Max and other being with a friend who had never used the software before, I simply placed my laptop in front of them with nothing running on it aside from the open project directory. While I have explained in the 'Aims of the Project' section of this paper that this project is not meant for first-time users of Max, I still thought it would be useful to see how my friend would get to grips with the interface. I told both beforehand I hoped to talk as little as possible as I wanted to understand how a first time user's mind sought to explore the patch, and I sat behind them as and took notes as I watched them use the software in the project. Both gave me a running commentary as they explored the project, and as I had hoped, I gathered a lot of valuable feedback about what was immediately clear to them and more importantly, what wasn't. Following these sessions, I added more help files around the patch and rearranged some of the interfaces for the spectral processing effects. What I enjoyed most from observing both users, was watching them essentially trying to break the patch once they understood what the parameters did. This kind of destructive testing was not the type that I spent much time on during the construction of the project, as after I had finished building the spectral effect I was working on, I would quickly check the bounds of all relevant parameters before moving on to building the next.

Reflection

This project has undoubtedly been the biggest I have undertaken in my academic career, and has also been my most fulfilling. Having been fascinated by digital signal processing for the past two years, being able to explore a topic such as the FFT in about as much detail that Max can offer has been immensely gratifying. Having chosen to create a more functional project in a programming language that is generally for audio/ visual performance has definitely presented challenges, most of which I have been able to overcome. With projects in the past and in particular with this one that I have done in Max, at a certain point you start to hit a glass ceiling in terms of the dexterity that Max can offer as a programming language. What Max offers in terms of instantaneous visualisation and signal flow, can come as a detriment or limitation when you want to perform an operation fractionally different to what a Max object was built for.

Saying this, I have been pleasantly surprised overall with how much malleability Max has offered in terms of DSTFT data collection and spectral effects rendering. While there are certain frustrating bugs that affect Max, as there are with all programming languages, it has generally been straightforward to problem solve when code was not working and I haven't hit any obstacles that were impossible to overcome in my time building this project. Of course there is additional functionality I would like to have added if time had allowed, such as 3D rendering but overall I my project displays all the functionality I intended it to 6 months ago. In terms of the project's portability to the community, I am very thankful for Max's extremely active online presence and hopeful that the beta will be viewed and used by Max users who wish to learn more about the FFT in the context of Max. Frustratingly, it may well be possible that users with older laptops or limited RAM may struggle to run some of the latter spectral effects as efficiently as I would hope, but this is an issue all programmers have to deal with in one way or another. There is scope for the project to be implemented in a structure through which all spectral effects can be opened from a central Max window and communicate with a central 'jit.matrix'. Doing so would not only simplify the interface for the user, but also eliminate some bugs that become apparent when trying to run two spectral effects concurrently. I also hope in future to be able to explore the FFT in other programming environments in the future, as I'm sure that the functionality that was lacking in Max will be available in in other programming languages and vice versa.

The Project in the Community

As I mentioned in the 'Aims of the Project', my aim for this project was to build modules of code that all built on one another, making it both easier for the user to navigate my project, and for them to adapt code they wanted to use in their own work.

Having navigated Cycling 74's projects page in October 2018, it feels satisfying to come full circle and submit my work on site. This section of Cycling 74's website is perhaps where my project will be viewed by the selection of people closest to my intended demographic. Most projects submitted here are generally of a high calibre, and I am hopeful it will attract more specialist users who will be able to provide insightful feedback on my project.
https://cycling74.com/projects/the-fast-fourier-transform-and-spectral-manipulation-in-maxmsp-and-jitter-beta.

Another subsection of Max's extremely active online community is the private group on Facebook 'Max/MSP'. This community is both an extremely responsive question and answer medium through which I have seen countless users helped with problems they are experiencing with their projects, and also a platform through which programmers can share projects they are working on. While Cycling 74's projects website section does offer a reply section, the Facebook group's comment sections are always far more active, so I look forward to receiving feedback from fellow users.
https://www.facebook.com/groups/maxmspjitter/permalink/10158296964374392/.

Code sharing platform GitHub is the third way in which I have shared my project. Traditionally a platform for text-based programming languages, GitHub is great platform for sharing my project because of how it presents code repositories. When looking at an upload on the site, you can navigate individual directories without having to download the whole project. This method of sharing will hopefully inspire other GitHub users to explore my project for modules they can use, or just browse until they find a spectral effect they wish to explore.
https://github.com/dwambyman/The-Fast-Fourier-Transform-and-Spectral-Manipulation-in-Max-MSP-and-Jitter.

In order to immediately accentuate the functionality of the project to the community, all posts have accompanied by a YouTube video showcasing clips of each spectral effect. Where possible, all posts have been accompanied with relevant hashtags and tags.
https://www.youtube.com/watch?v=U3hjo-Jnmms&feature=youtu.be.

I have also written a blog which chronicles the six-month development of this project, for those who are interested in a more informal look at the issues in constructing the code and the various design decisions I had to make over the last six months.
https://digital.eca.ed.ac.uk/spectral-manipulation/.

Regardless of platform, I will also be mentioning that the project is in beta in the hope that I will receive constructive feedback from fellow programmers. While I have completed the aims I set out to achieve from the inception of this project, I am still aware of potential bugs that may affect users on different operating systems or setups different to my own. It has been open-source software and commentary that has facilitated much of my work, I am looking forward to being able to develop the project further alongside others and in doing so give back to the Max community.

Bibliography

Cipriani, Alessandro, and Maurizio Giri. *Electronic music and sound design: theory and practice with Max-MSP*. Rome: ConTempoNet, 2010.

Cipriani, Alessandro, and Maurizio Giri. *Electronic music and sound design: theory and practice with Max-MSP, Volume 2*. Rome: ConTempoNet, 2014.

Droljc, Tadej. 'STFT Analysis Driven Sonographic Sound Processing in Real-Time using Max/MSP and Jitter.' BSc diss., University of Hull, 2011.

François-Charles, Jean. 'A Tutorial on Spectral Sound Processing Using Max/MSP and Jitter.' *Computer Music Journal* 32 (3). Accessed 26th October 2018, https://doi.org/10.1162/comj.2008.32.3.87.

McCellan, James, and Ronald Schafer, and M.A. Yoder. *Signal Processing First*. Upper Saddle River: Pearson Education International, 2003.

Figure(s)

Figure 2 – Owen Green, https://digital.eca.ed.ac.uk/sounddesignmedia/2012/11/20/spectral-processing-and-the-fft/.

Figure 4 – Brian Hamilton, MAFTDSP Lecture 8.

Figure 1.1.1 - https://docs.cycling74.com/max5/tutorials/msp-tut/mspchapter26.html.

Figure 1.1.2 - https://en.wikipedia.org/wiki/Nyquist_frequency#/media/File:Aliasing-folding_SVG.svg.

Figure 1.1.3 - https://docs.cycling74.com/max5/tutorials/msp-tut/mspchapter26.html.

Figure 1.6.2.3 – Droljc, 'STFT Analysis', 167.

Figure PB.7 – Charles, 'A Tutorial', 95.

B081743

Directory

Dubois, Luke. /Cycling '74/Max 8/Examples/jitter-examples/audio/jitter_pvoc.

Sample(s)

last_eve - https://freesound.org/people/acclivity/sounds/435535/

drone - https://freesound.org/people/toiletrolltube/sounds/466198/

spring_in_town - https://freesound.org/people/inchadney/sounds/466528/

thunder - https://freesound.org/people/Erdie/sounds/24003/

Website(s)

Cycling 74. 'Tutorial 26: Frequency Domain Signal Processing with pfft~.' Accessed 30th October 2018. https://docs.cycling74.com/max5/tutorials/msp-tut/mspchapter26.html.

icram. 'AudioSculpt.' Accessed 19th April 2019. http://anasynth.ircam.fr/home/english/software/audiosculpt.

Klingbeil. 'SPEAR.' Accsessed 19th April 2019. http://klingbeil.com/spear/.

YouTube. 'Sonographic Sound Processing in Max/MSP and Jitter.' Last modified 17th April 2013. https://www.youtube.com/watch?v=0PrIO5tweeo.