Mimic: An Imitative Synthesis Algorithm For Automating Analysis And Resynthesis

Presented by:

Akash Murthy

A thesis submitted for the degree of:

Creative Music Technologies, Master of Arts

NATIONAL UNIVERSITY OF IRELAND
MAYNOOTH
Ollscoil na hÉireann Má Nuad

Department of Music

Maynooth University, NUI Maynooth

Co. Kildare, Ireland

Submission: **September 2017**

Head of Department:

Prof. Christopher Morris, BMus, MA, PhD

Research Supervisor:

Dr. Iain McCurdy, PhD

Table of Content

List of Figures

List of Formulae

List of Tables

Abstract

The idea of frequency domain analysis of sound sources has been explored in vast detail. There are now algorithms that do frequency domain conversion in real time and we have algorithms which manipulate spectral information to correct pitch on one hand and incinerate any evidence of pitch on the other. But here in this thesis project, we delve into the realm of mimicking real world sounds by automating the process of spectral analysis and gathering relevant data from the analysis to accurately recreate the original source.

We begin with a background study about various topics and concepts involved in this project. We then quickly move on to building an application using Csound and Cabbage, giving it a graphical layout and providing an interface for the algorithm. In the back end, we design an algorithm which lets the user provide a part of a source .wav file for analysis. The algorithm would then convert the source into the spectral domain, write out data logs of all necessary frame information. From this data log, we filter and map the frequencies that accurately represents the source, extract amplitude envelopes for all selected frequencies and provide an additive synthesis framework which rebuilds the source from first principles. We go a step further and provide a static window synthesizer for zeroing in on a particular section of interest in the source and the ability to play that section endlessly. We then test the application with sound files of varying duration and timbral properties. To conclude the project, limitations of the approach in use is established and steps to improve upon the existing work and targets for a future release version is illustrated.

Acknowledgement

Firstly, I would like to express my sincere gratitude to my thesis advisor Dr. Iain McCurdy for the continuous support of my Master's study and related research, for his patience, motivation, and immense knowledge. His guidance helped me throughout the time of the research and documentation of this thesis. I could not have imagined having a better advisor and mentor for my thesis.

Besides my advisor, I would like to thank members of Maynooth University's Music Department faculty: Prof. Victor Lazzarini and Dr. Gordon Delap, for their insightful knowledge and encouragement throughout the academic year which helped me grow.

Last but not the least, I would like to thank my family, my parents and to my brother for supporting me spiritually throughout writing this thesis and my my life in general.

Chapter 1: Introduction

1.1 Motivation

Samplers have been around since the late 1960's. The process of recording a sound sample, slicing it, and playing back at different rates on a keyboard has been well explored. A lot of electronic music involves sampling real world sounds and mangling them into abstractness. But the idea of recreating and re-synthesizing sound from analysing a source hasn't been explored in great detail. Partly because of the enormous overhead in analysing the spectral content of the source material and mapping their properties. The other part involves the sheer amount of processing power required to recreate a spectrally rich timbre accurately using nothing but sinusoids. So, it is quite an easy decision to opt for the sampling process over the method of imitative synthesis when the targeted sound needs to be similar or on par with the original source material. But it is an entirely new ballgame when you want to control every aspect of the source material, since all of its data is known and recorded during the process of analysis and all its properties and values can potentially be manipulated.

One of the very first lessons on Music Signal Processing offered to us at Maynooth University was on additive synthesis. The concept of additive synthesis was easy to comprehend and we soon realized the power of this method of synthesis in imitating real world sounds, as we we recreated the sound of crotales using a table of frequency values and amplitudes. But nonetheless, it was a very cumbersome process which involved a lot of manual input, checks and processes. The source file had to be acquired and analyzed through a frequency analyzer. This would display a dynamic spectrum of the source file over time, with dominant partials represented brightly, the intensity of which represented the amplitude of the partials. The user had to zero in on the graph and subjectively identify, pick and choose partials of higher strengths, determine their frequencies, starting amplitudes and the time taken for these partials to decay. This tabulated data needed to

be fed into tables or manually entered as inputs to oscillators. This, at the time, seemed like a good exercise to understand the intricacies of audio, its frequency domain representation and the elements which make up complex sounds. But then again, I felt the need for such processes to be automated, as any process a person can do, determined by a rule based methodology, could in reality be automated. This need to introduce a programmatic approach to a formally manual one, triggered my interest in this area of research.

But it wasn't until later that many other concepts were introduced which could solidify the idea into viable code. I quickly began comprehending the idea of having an application which could analyze any sound file and build a dataset of useful information from it. I envisioned that this dataset could be used as an input to a secondary application which could read the data and render or play a sound file. This meant that there would be loose coupling between the 2 components and they could be reused for different purposes. A preliminary goal was to build the first part as a Csound plugin opcode in C++ which could be used then in a Csound project to realize the second part. But due to massive time constraints and inaccuracies in planning, I bundled the two parts together into a single application which did both.

1.2 Scope of the Project

The scope of this project is contained to the usage of Csound code bundled as an application to analyze uncompressed .wav files to extract information from them and utilizing this information in recreating the source files from scratch. This project does not explore in any significant depth, the background concepts that are mentioned in the literature such as frequency domain conversion and building applications through Csound. Several references are provided at the end to facilitate learning if the user so requires.

The write-up does give a fairly in-depth review about the processes and algorithms developed and used in the project, but is in no way a substitute to a thoroughly documented manual in understanding the algorithms. The reader is expected to go over the provided code and infer and understand the algorithm with the help of the descriptive text provided in this write-up

Finally, the application associated with this project is meant to be purely experimental and educational and is not suitable for commercial use or distribution.

1.3 Goals and Deliverables

The primary goal of the project is to establish the fact that an automated approach to imitative synthesis is in-fact possible and to establish a framework for further development and improvement of the existing approach.

The secondary goal is to spark interest in the approach to creating a wide range of data processing and manipulation tools within Csound and outside it to take the existing raw data from the analysis of sound files and filter and map it in different ways to truly extract the power of this approach. The thesis makes a shallow dive into some such methods of creating quirky sounds, but it is left to the reader to be creative and imaginative with what's possible.

Towards the end of this write-up, the reader will have a thorough understanding of the aforementioned approach, and a working code base to follow along with.

Chapter 2: Background

Since we are delving into a fairly technical area of research and discussion, it is important to familiarize oneself with some concepts, processes, tools and lexicon that are needed for fully understanding the topic. Listed below are some the fundamentals required to appreciate the literature.

2.1 Additive Synthesis

Additive synthesis is a sound synthesis or compositional technique where complex sounds can be generated by adding simpler sounds together. It is established that all complex sounds can be broken down into sinusoids. A sinusoid is the fundamental building block of sound and is a pure tone, only replicable in a lab or a computer and non existent in real life. Additive synthesis makes use of the same principle applied backwards to establish that all complex sounds can be formed by selecting and adding sinusoids of different frequencies and amplitudes. Many interesting emergent properties arise from combining sinusoids, such as beating, where in a modulation in volume/amplitude is observed when 2 sinusoids of similar but unequal frequencies are introduced.

Additive synthesis can result in both harmonic and inharmonic timbre, depending on the arrangement and selection of frequencies and amplitudes for the sinusoids. Harmonic additive synthesis is a way of expressing a periodic function as the sum of sinusoids based on a certain arithmetic series or progression. When sinusoids are chosen such that their frequencies are integral multiples of a fundamental frequency, they are called partials or overtones, and generate harmonic waveform which are periodic in nature. Inharmonic additive synthesis on the other hand follows no obvious pattern nor is it integrally

associated with the fundamental frequency. These generate non periodic and inharmonic sounds such as bells and gongs.

Generally, spectrally rich timbres contain hundreds, if not thousands of distinct partials and to imitate them, the same number of individual sine wave oscillators need to be added together. Fortunately, the computer is an adding machine. With today's computational capabilities, thousands of sine wave oscillators can be combined to create and recreate any sound possible. Additive synthesis is only bound by the computational power and speed of the computing machine. However, there are more efficient and elegant ways and synthesis techniques available to us to generate rich timbral sound, such as subtractive synthesis, frequency modulation(FM), granular synthesis, etc. But the true power of additive synthesis comes into play when a sound source needs to be mimicked accurately. All other techniques provide a qualitative and experimental way to achieve the same result, and with enough tries and experiments, a similar sound can be achieved. But with additive synthesis and spectral tools for analyzing the frequencies and amplitudes of waveforms, a programmatic approach can be adopted where the decision to select partials always result in the required tone.

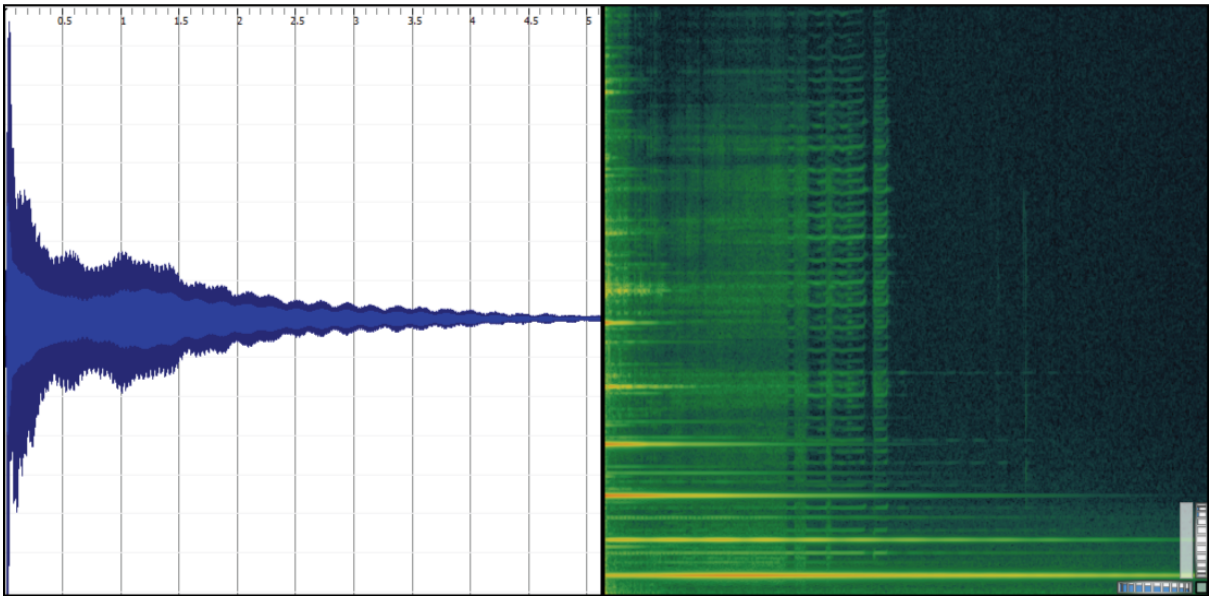

Figure 2.1: Waveform and Spectrum of a bell sound sample

Here is an example of a spectrogram analysis of a bell sound sample. It is evident from looking at the spectrogram that there are spectral peaks of high energy spread across the spectrum of frequencies, and these represent the dominant partials present in the waveform. Since we are trying to achieve a programmatic automation of this analysis, we

need to know about what a spectrogram is and how a waveform can be converted from time domain to frequency domain, where all the magic happens.

2.2 Frequency Domain Conversion

A sound file in the digital format exists in a time domain representation. It tells us the pressure level of the sound wave over time. But to determine the frequency from this representation is an awkward task. If it is a simple signal, the intervals can be measured and the frequency can be inferred. But for complex waveforms this determination becomes clouded and often inaccurate. The shape of the waveform determines the timbre and the resultant frequencies associated with the sound. So then certainly, frequency is intricately linked with the time domain representation, but it is not obvious. Since frequency and pitch are primordial components in analysing sound, a better representation was needed.

As previously mentioned, all complex signals can be broken down into a sum of sinusoidal signals. It was Joseph Fourier, a French mathematician and physicist who mathematically proved that such a transformation was possible. We shall now explore some of the mathematical tools and processes that have been derived over time for practically realizing some of the techniques of spectral signal processing. The theoretical foundation was established for continuous signal in the analog domain, but was later modified to process digital signals in a discrete time and discrete frequency version called the Discrete Fourier Transform (DFT).

2.3 Discrete Fourier Transform

The equation applied to discrete signals with a bound value for time is as follows:

$$X(k) = \frac{1}{N} \sum_{t=0}^{N-1} x(t)\, e^{-j2\pi kt/N} \quad where\ k = 0,\ 1, ...,\ N-1$$

t : *Discrete time index* (*normalized time*, $T = 1$)

k : *Discrete frequency index*

$$\omega_k = 2\pi k/N \ : Frequency\ in\ radians$$
$$f_k = f_s k/N \ : Frequency\ in\ Hz\ where\ f_s\ is\ sampling\ rate$$

Formula 2.1: DFT equation

Here we can see that both time and frequency are limited to *N* discrete steps. The frequency component is substituted by the index *k*, which refers to the set of frequency points. The slice of input signal in time *t*, is multiplied by a complex exponential *e*, determined by the Euler identity. Complex exponentials are signals which are in-turn composed of both the real and imaginary parts which can be visualized in a 2 dimensional complex plane. Complex exponentials are theoretical functions which have no significance in the real world, where real signal are present. But it gives us a calculable standpoint upon which to build our equations. Euler formulated that his identity could be expressed as a sum of sinusoids, one phase shifted by 90 degrees from the other. The Euler formula is represented as follows:

$$e^{j\varphi} = cos\varphi + jsin\varphi$$
$$cos\varphi = (e^{j\varphi} + e^{-j\varphi}) / 2$$
$$cos\varphi = (e^{j\varphi} - e^{-j\varphi}) / 2j$$

Formula 2.2: Euler formulae

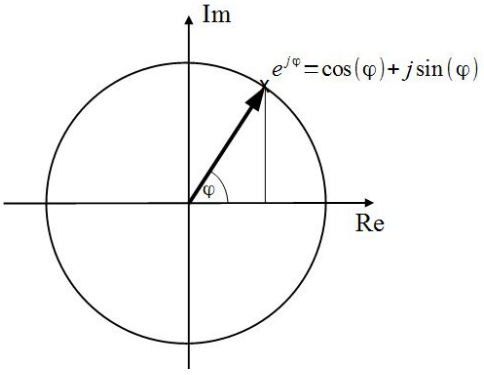

Figure 2.2: Complex plane representation

The equation for discrete fourier transforms can be rewritten as follows:

$$X(k) = \frac{1}{N} \sum_{t=0}^{N-1} x(t)[cos(2\pi kt/N) - jsin(2\pi kt/N)] \qquad where\ k = 0,\ 1,\ ...,\ N-1$$

Formula 2.3: DFT equation

The equation clearly shows that each slice of input signal *x*, sampled over a discrete time interval of *t*, is then multiplied by a complex signal consisting of two sinusoids with a phase difference of 90 degrees and an angular frequency of $2\pi kt/N$. The resultant signal is a

frequency domain signal *X* with both real and imaginary parts. The frequency and phase components of the signal can be extracted by decomposing the signal into its polar form, where the magnitude gives the frequency and the angle gives the phase. The polar form, as opposed to the cartesian form, of any complex signal is represented by the formulae:

$$Frequency(A) = mag(X) = \sqrt{a^2 + b^2}$$
$$Phase(\varphi) = angle(X) = tan^{-1}(\frac{b}{a})$$

Formula 2.4: Polar Form equation

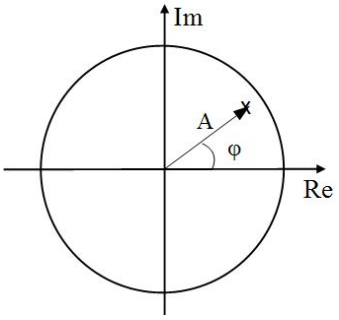

Figure 2.3: Polar Form Representation

To visualize the process and effects of the DFT operation, we can apply certain test values to the equation and generate a graph of the input signal, the magnitude spectrum of the output signal and the phase spectrum of the output signal.

For a complex signal *x*, when *N* is assigned to 64 discrete time intervals and when *k* is a frequency of 7.5, we get the following graphs:

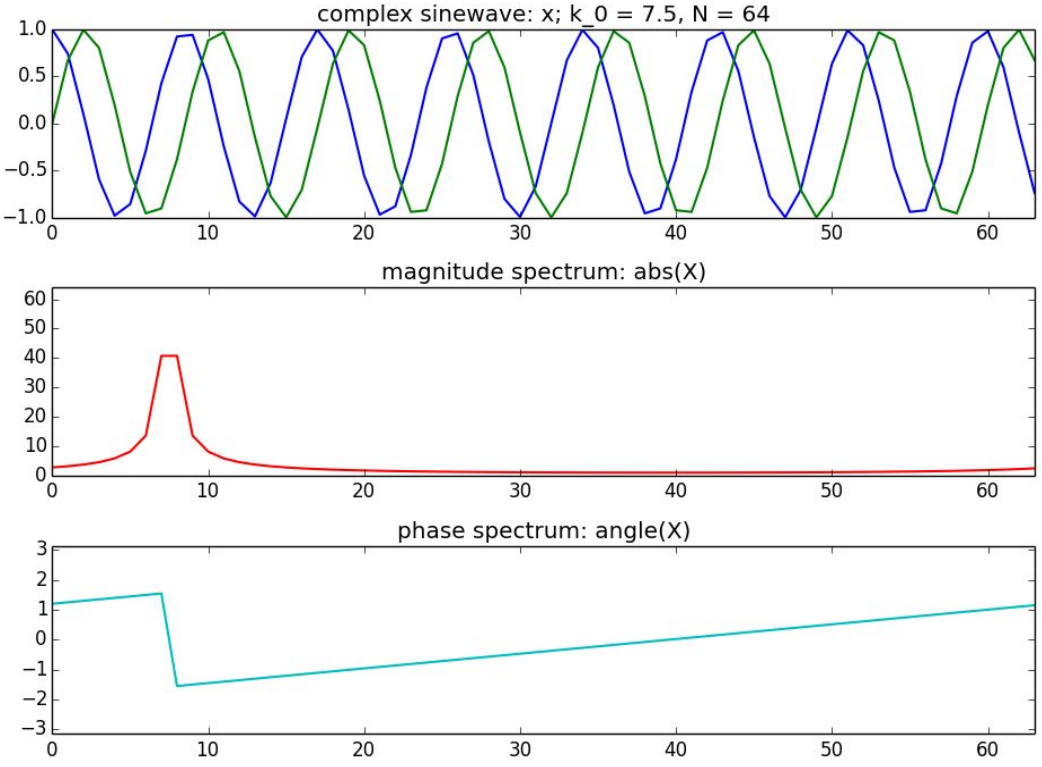

Figure 2.4: DFT of complex signal

As we can see, the input signal is a combination of 2 sinusoids, phase shifted by 90 degrees. This constitutes both the real(*cos*) and imaginary(*sin*) parts of the complex signal. Since the frequency chosen k = 7.5, is a not integral number and cannot be resolved discretely within the 64 steps, the magnitude spectrum is clouded around the 7-8 region, where a theoretical maximum exists. But magnitude values are still non-zero around the region and throughout the spectrum as well. If we had chosen k to be 7, say, since it would've been an integral number easily resolvable with the 64 discrete steps, we would have a theoretical maxima on the frequency spectrum at precisely 7 on the x-axis and zero values for every other value on the x-axis. The phase spectrum is not obvious to deduce and since we are focusing our attention on frequency analysis, irrelevent to our discussion.

As mentioned previously, we usually deal with real signal, where the imaginary part is zero. So now, let us consider a real signal with the same test input data applied. We get the following result:

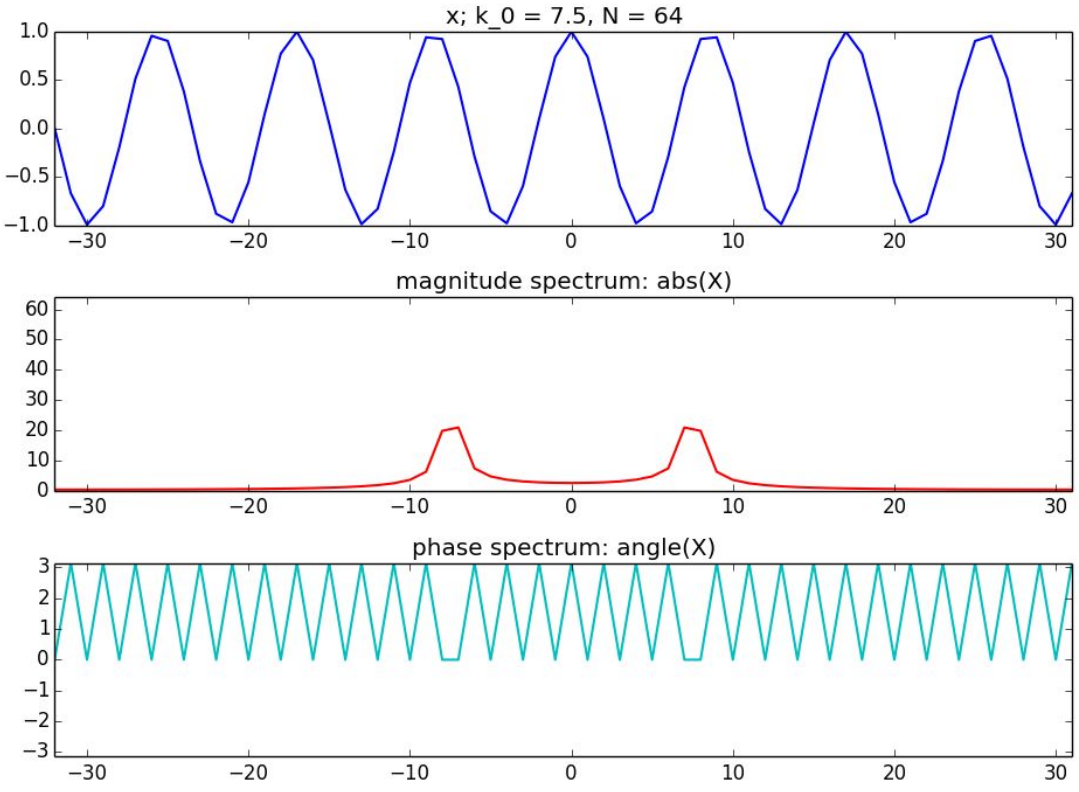

Figure 2.5: DFT of Real signal

The input signal *x*, is centered around 0 for convenience. We can see that a similar maxima occurs on the frequency spectrum at around 7-8 Hz. But there exists a similar maxima on

the negative side of frequency spectrum. Moreover the spectrum is an even function, which is symmetric around 0 Hz. We can also observe that for the *N* samples being analyzed, the frequency resolution was only *(N/2) -1.* Thus when we sample a sound file at *N=44100*, we get a frequency resolution up to the Nyquist frequency of *N/2*. Since the magnitude spectrum is an even function, we just need to retain the positive spectrum and discard the negative spectrum, as no data is lost in doing so. Though it is not obvious from the graph above, the phase spectrum is anti-symmetric around 0.

2.4 Fast Fourier Transforms

Through DFT, we are able to retrieve a frequency spectrum snapshot of a sampled region in time. But DFT is a computationally expensive operation. DFT is an idealistic mathematical formula for determining the frequency spectrum, but when applied as an algorithm, is highly inefficient as the number of computations needed to be performed is in the order of N^2. If the sampling rate *N* increases, so does the time required to complete the computation. Thus an alternative algorithm had to be devised to exploit all the properties of Fourier transforms and to eliminate redundant calculations. Then came the Fast Fourier Transform (FFT) algorithm, originally called the Cooley-Tukey algorithm, which recursively breaks down the DFT of power-of-2 sized signal into two pieces of size *N/2* until it can no longer be decomposed into smaller signals.

Thus, FFT, as a family of algorithms, is a faster way of computing the DFT. The heart of the algorithm is based on the Divide and Conquer algorithm. Rather than working with big signals, we divide our signals and perform DFT on those smaller signals. In the end, we add all the decomposed DFT operations to get the DFT of the main signal.

Through our Divide and Conquer method, if we divide our N-point signal into two signals *S1* and *S2* of length *N/2* each, and then perform DFT of these smaller signals, we will be performing $(N/2)^2$ multiplications for each of the smaller signals. So a total of $2 * (N/2)^2$ calculations are performed for calculating the DFT of both the signals. Then, to get the actual DFT of the N-point signal, we need to add these components. Since the single addition is an inexpensive operation as compared to the rest, it can be ignored. In the end, we need approximately $(N^2)/2$ operations, which is half that in the case of DFT of the same N-point signal.

This reduction of half is a result of dividing the signal once. But if we continue dividing the signal, we end up reducing the amount of computations in each step. The overall number of computations that it boils down to using this approach is of the order of $Nlog(N)$. To understand the significance of the reduction in computational size, the following graph illustrates the number of operations needed and the time taken and the order of increase of computations when the size of N is increased.

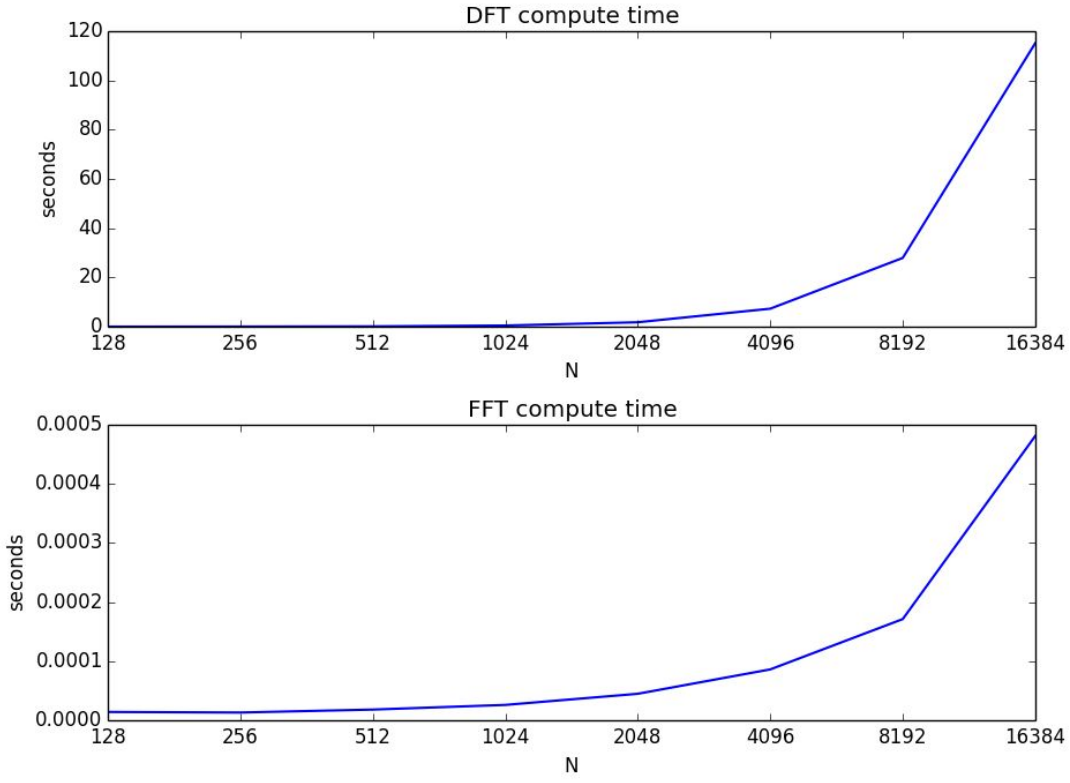

Figure 2.6: Computational analysis of DFT vs FFT

2.5 Short Time Fourier Transform

The process of FFT produces a snapshot of spectral activity for a small slice of signal frozen in time. To produce a dynamic spectrum of time varying frequency response we need to take several snapshots of audio samples and merge them together. This is where Short Time Fourier Transforms (STFT) come into play.

The first part of the STFT process involves dividing the input signal into several time frames, each containing a block of samples. This frame is then multiplied by a window function. A window function is a simple shape function which is non-zero over the area of analysis and zero everywhere else. When the window is multiplied with the frame, the magnitude of the window function is transferred to the frame. Popular window shapes include the Rectangular, Hamming, Hanning and Blackman window. Below is a graph comparing a few different windows with each other over the size of the analysis frame.

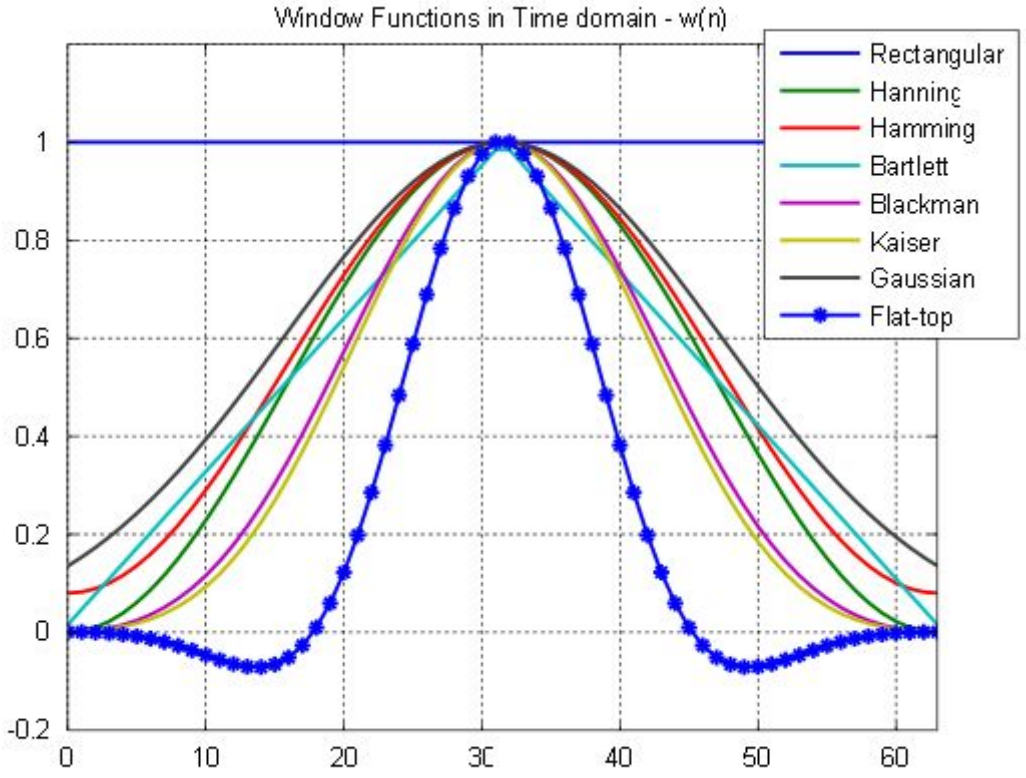

Figure 2.7: Common window functions

After the transformation, each frame is subject to STFT. The equation describing the STFT operation is as below:

$$X_l[k] = \sum_{n=-N/2}^{N/2-1} w[n]x[n+lH]e^{-j2\pi kn/N} \qquad where\ l = 0, 1, 2, ...$$

w : *Analysis window size*

l : *Frame number*

H : *Hop size*

Formula 2.5: STFT equation

The equation describes a time variant analysis where an FFT operation is carried out for every frame l. The window w is scaled with the analysis frame determined by the hop size H. The STFT operation can be seen as a sequence of FFTs or as a bank of bandpass filters spaced equally from 0Hz till the Nyquist frequency.

Before the analysis, several parameters can be set. The first one being the size or length of the frequency bands or bins, which determines the frequency resolution. The number of bands (256, 512, 1024), which is always a power of 2 for efficient FFT analysis, determines the size of each bin. Bins are equal lengthed regions spaced equally between 0 Hz and the Nyquist frequency. The analysis creates a mirror of the frequency response on the negative axis as well, but this is discarded and the magnitude of the response on the positive axis is doubled to compensate.

There exists a relationship between the number of bins, the bandwidth, the sampling rate and the frame size. The relationship between these terms and a table consisting of sample common configuration values are listed below:

$$bandwidth = (sample\ rate / 2) / number\ of\ bins$$

$$framesize = sample\ rate / bandwidth$$

$$number\ of\ bins = framesize / 2$$

Formula 2.6: Bandwidth, frame-size, number of bins

Number of bins	Framesize	Bandwidth	Sample rate
256	512	86.13	44100
512	1024	43.07	44100
1024	2048	21.53	44100
2048	4096	10.76	44100

Table 2.1: Relationship between Number of bins, Frame-size and Bandwidth

2.6 Framesize, Bandwidth and Number of Bins

As the number of frequency bins increases, the bandwidth decreases and this corresponds to a high degree of accuracy in frequency resolution and pinpointing the right frequencies. But as the number of frequency bins increase, so does the framesize and the number of samples considered for analysis. This reduces the time resolution and smears the analysis if the pitches vary frequently in the sound sample being analysed. Suppose we would want a frequency resolution of 1Hz, we would need to sample a frame of 44100 samples for the same sampling rate. This would mean analysing a window/frame size of 1 second in duration. Since the snapshots taken are a second long, any changes in pitch within the source material is not registered as a change, rather all possible frequencies present within that window are registered. Thus there exists a fine balance between time resolution and frequency resolution, and the values to be configured should be chosen depending on the source material.

2.7 Hop Size and Overlap Factor

Another factor which is considered for analysis is the Hop Size or the Overlap Factor. Both these terms can be used interchangeably, but the value associated with them change. The hop size is the number of samples to skip till the start of the next frame/window for analysis. If the hop size is the same as the framesize, then there is no overlap. We need to overlap the analysis frames since a window function is applied to each frame. Thus, to balance out the loss in amplitude at the side bands, and to avoid sharp clips (in the case of rectangular windows) or to avoid amplitude modulation, overlaps are needed. The term Overlap Factor determines the number of analysis windows covering each other at any given point. A higher overlap factor is useful for future time-dilation, however this may also cause smearing.

2.8 Effects of Window Functions

The primary usage of windows functions are for smoothing the framed signal over the edges of the time interval and gradually reducing it to zero to avoid clipping and glitches. But on a subtle note, the spectral characteristics of a window function are used to determine the balance between detection and resolution.

Detection refers to detecting a certain signal in the presence of broadband noise. Resolution on the other hand, refers to the ability to distinguish narrow band spectral components. The following emergent properties arising from choosing a particular type of window function is essential for answering the above question.

Main Lobe Width: The main lobe width is linked to frequency resolution. Narrower the main lobe width, the better the frequency resolution. With this decrease in width, the energy is transferred to the side bands, which increases spectral leakage, which decrease the ability to detect a certain frequency.

Side Lobe Level: The side lobes appears at either side of the main lobe. Side lobes determine how much of the spectral energy or frequency component is spread across adjacent bins. It is measured relative to the peak amplitude(dB) of the main lobe. As previously inferred, lesser the side lobe level, the better the ability to detect and pinpoint frequencies.

An ideal Window Function has:

1) Narrow main lobe width.
2) Low level side lobes.
3) Side lobes that drop off quickly.

The following is a graphical representation of the frequency responses of some of the common window functions when their amplitudes are normalized.

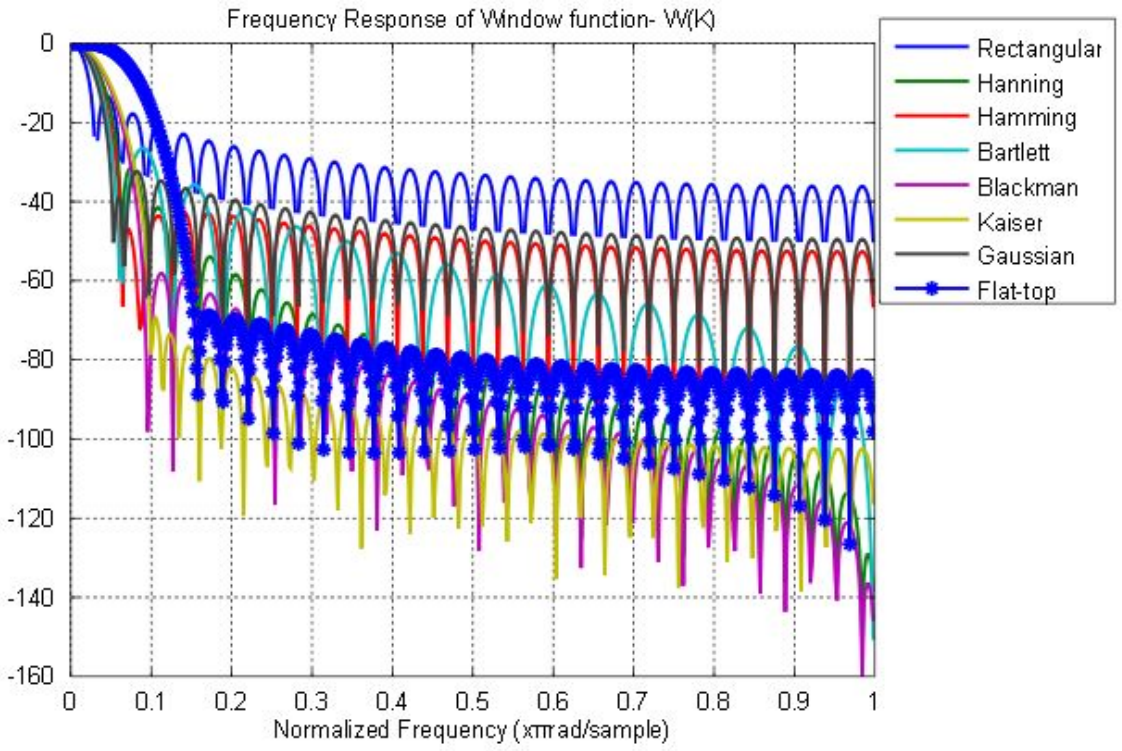

Figure 2.8: Frequency response of window functions

2.9 Resynthesis

The conversion from time domain to frequency domain of an audio signal provides a lot of insight on the spectral characteristics of the sound. But the real power lies in the digital signal processing in the spectral domain, which can produce a wide range of possibilities in terms of modifying the signature of the sound and is key in many aural effects that we hear such as spectral delays and pitch warpers. But the process is useless if we cannot convert the frequency domain signal back to the time domain, which can then be played back. Thus arises the topic of resynthesis of spectral signals through inverse FFT by means of Overlap-add method or the Oscillator Bank resynthesis method.

The equation for Inverse Fourier transform is quite similar to the forward DFT equation, except that the summation is over discrete frequency intervals rather than time. The equation for IDFT is as shown below:

$$x[n] = \sum_{k=0}^{N-1} X[k]\, e^{j2\pi kn/N} \quad where\ k = 0,\ 1, ...,\ N-1$$

k : *Discrete frequency index*

$\omega_k = 2\pi k/N$: *Frequency in radians*

$f_k = f_s k/N$: *Frequency in Hz where* f_s *is sampling rate*

Formula 2.7: Inverse DFT equation

The overlap-add method is a DSP technique where the following steps are done:

1) Decompose the primary signal in the frequency domain into simple components based on the parameters used to convert it into the spectral domain in the first place.
2) Process each individual component by applying the inverse FFT function and applying a synthesis window.
3) Add to the output at the current position to the recombined audio signal and advance the location of the next component to be added depending on the hop size chosen.

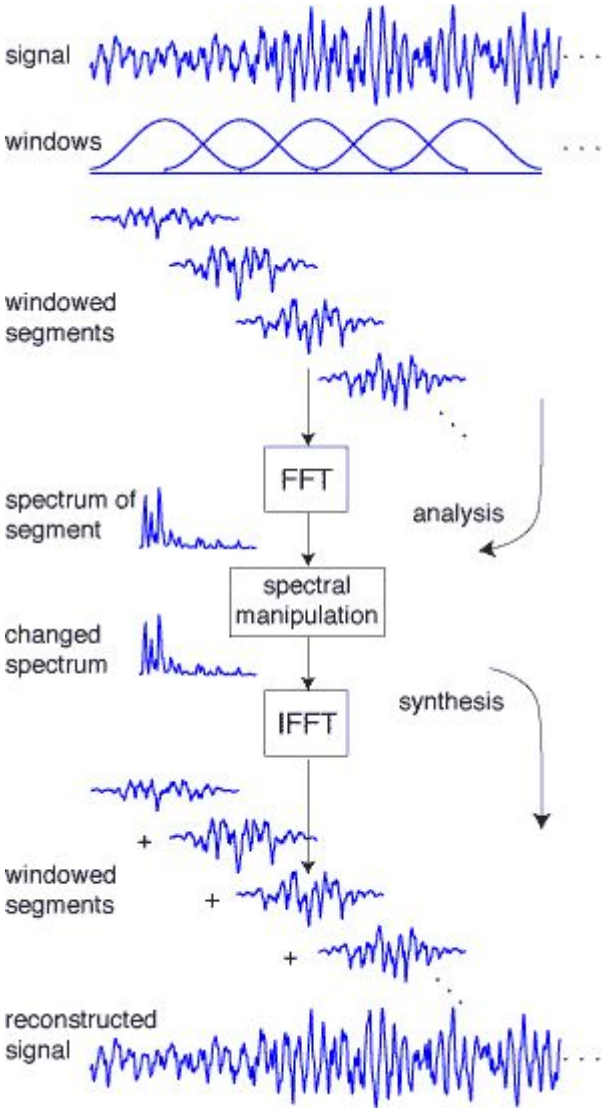

Figure 2.9: Life cycle of signal through time and frequency domains

2.10 Language and Tools

The analysis and processing is done on Csound, a text based sound rendering programming language. Csound is an esoteric language since it is much different from the modern programming languages available today with varied styles and paradigms. Csound, on the looks of it seems like an assembly language with instructions being executed sequentially. But there are several detailed and high level audio processing modules or opcodes that enable the user to create and prototype an effect or synth they need, rather quickly. Csound code, in the end, is interpreted as C code. While Csound is almost perfect for audio processing, it is limited and cumbersome for anything else, such as data processing. Thus it was a choice between choosing C++ and Csound for developing the application relating the thesis study. C++, while being an exceptionally powerful language and great for building audio applications when used in conjunction with the Juce application framework, was a hard option for me to choose since I was limited by time and was not oblivious to the many bugs and errors that I would potentially encounter. In the end Csound was the obvious choice for me and it worked out well.

For testing the application and visualizing the generated output signal, a free software called Sonic Visualizer, is used. This tool helps in graphically representing a signal in both the time domain waveform as well as a frequency domain heat-map.

Chapter 3: The Application and The Algorithm

The background research for my project was built on top of a paper written by Joachim Heintz from Incontri HMTM Hannover, titled *Mimesis and Shaping -Imitative Additive Synthesis in Csound.* The abstract of the paper describes his works as follows:

> *" This paper is to introduce a realisation of Imitative Additive Synthesis in Csound, which can be employed for the realtime analysis of the spectrum of a sound suitable for additive synthesis. The implementation described here can be used for analysing, replaying and modifying sounds in a live situation, as well as saving the analysis results for future use. "*

3.1 Explanation of the Build

The application was programmed in Csound v6.09.1 using the IDE and plugin rendering software, Cabbage v2.0.13. Cabbage provides a well typed text editor and a rich array of widgets for a graphical plugin editor. To run the application, both Csound and Cabbage needs to be installed. Please refer to the References section for download links on where to download these softwares. Also provided in the Reference is a link to the Git Repository for this project. Since it is hosted as a Git project, any changes made to the project can quickly be pulled down and provides a platform for continuous integration. Running the code is a fairly straightforward procedure. The first step is to download and install the latest version of Csound. Once done, download and install the latest version of Cabbage 2.

Cabbage 2 at the moment is in a beta release stage with a canonical version being offered from the forum. This may change later, thus refer to the readme file from the Git project to get hold of a stable version in the future.

3.2 Layout of Plugin

The plugin window is populated with the following widgets:

Figure 3.1 GUI of the application window

Open File: This opens a native browser window to select a file for analysis. Files supported depend on the current release of libsndfile library, but it supports all the popular lossless formats such as WAV, FLAC, OGG etc. The browser window opens at the working directory where the .csd file is running from. A WAV file can be chosen from any directory that it may reside at. Once the file is selected, it is populated and projected onto the Waveform Display

Input Waveform Display: This helps the user to visualize the time domain waveform of the source file that they select. The user can also select a region in time by moving the Source Selection sliders to mark a region. This region is considered for analysis. When no file is loaded (at the beginning), the display is blank and displays the text "(No audio file loaded)".

Input Source Selection: This is a 2-way horizontal slider which determines the region of selection of the input waveform. Two number boxes exist on either side which determine the time in seconds of the selection being made.

Replay Selection: This button plays the input source over the duration that is specified by the Input Source Selection. This is useful to quickly listen and figure out the right duration needed for analysis.

Analyze: This is the first of the 2 buttons that needs to be clicked in order to start the core of the application. When no file is chosen, this button is greyed out.

Process: This is the second button that needs to be clicked in order to continue where the Analyze button left off. This button is greyed out till the Analysis is complete. Once the control is relinquished from this input, the output is rendered on to the Snapshot Window.

Play: This button is pressed after the Process phase is completed in order to trigger the additive synthesis process and play the output resultant sound. This button is greyed out till the Process is complete.

Threshold: This slider controls the threshold level for analysis, and determines the number of bins selected after analysis.

FFT Size: This combobox has a list of power of 2 values which are used as the FFT size when converting the input signal to the frequency domain.

Waveform Type: This combobox determines the shape of the waveform used during resynthesis. By default, and for most purposes, a Sine wave is used. But other options include Sawtooth and Buzz.

Snapshot Window: This window will display the resynthesized version of the selected input region after the processing event is done.

Seek: This slider is used to pinpoint the window or frame approximately along the snapshot window, for use in tandem with the static synthesizer.

Envelope: This group consists of a conventional exponential ADSR envelope with attack, decay, sustain and release values with appropriate bounds.

Modulation: This group consists of a couple of sliders which control the rate of modulation of volume and pitch with a fixed depth of modulation.

3.3 Analyze source file

Once the region of the source file for analysis is selected and the Analyze button is pressed, the application kicks off by taking in the samples required from the source file as a stream of audio rate values from a table using a *phasor* and *tablei* opcode. This is a mono stream and the application, for this version, produces mono sound only.

Next, the audio file in the time domain is subjected to a phase vocoding analysis, by using one of the real time streaming PV opcodes from Csound's arsenal called *pvsanal*. This internally converts the source file into a frequency domain representation based on a stream format called f-signals. *pvsanal* handles the FFT, STFT and phase vocoding operations behind the scene. The Background literature of this thesis report contains an exhaustive report on how these 3 operations are executed. Suitable values of fftsize, overlap factor, window size and window shape are chosen beforehand.

Next, a unique operation is applied to this *fsig* stream, called *pvsftw*, which recursively goes through the entire file and generate values for 2 tables to store the amplitude and frequency values of each analysis bin present in the stream. This gives us the complete data of every frequency resolved by the process and it's weight over the duration of the sample source. This data is then stored in a new file called fulldata.txt, a text based Comma Separated Values (CSV) file. Data is appended to this file in the following format:

bin_number, frame_number, amplitude, frequency

Where *bin_number* represents the index of the bin that is producing the information, the *frame_number* is the index of the analysis frame being analyzed. This number stays constant till all the bins of this frame have been read. *amplitude* and *frequency* are self describing quantities extracted from each bin. *amplitude* is a zero-DB full scale value represented linearly between 0 and 1. And *frequency* refers to the center frequency of each bin present between 0 HZ and the Nyquist frequency.

The task of writing all this data to a file is a costly operation and the time taken and the file size achieved is subject to change depending on the length of source file chosen for analysis and the parameters chosen for spectral analysis. The file size of *fulldata.txt* far exceeds the relative size of the chosen area for analysis, so please make sure that there is sufficient disk space on the drive running the csd file.

While this file is being written, another filtered version of this data is written to a separate file, *maxdata.txt*, simultaneously. This filters out and retains all the entries with their amplitude values above a certain threshold which the user can modify. The file that is generated is drastically smaller than the other file with all the data. This data is now contains only frequencies that are prominent and important to us for analysis. Since a spectrally rich sound source can contain thousands of partials and frequencies, we need to put a cap on selection process and select only the most prominent partials.

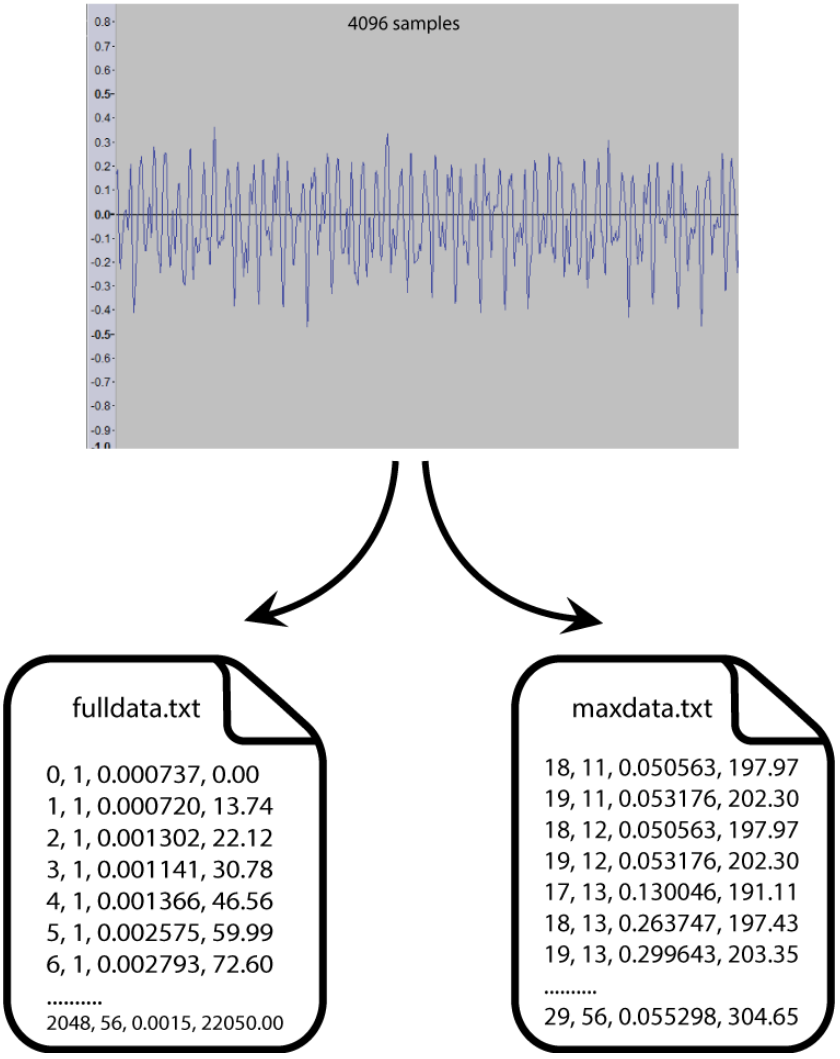

Figure 3.2: Extracting data from source

This marks the end of the analysis phase, at the end of which, data required to process is generated and saved, which can then be potentially reused for other applications.

3.4 Processing the Data

Once the analysis phase is done, and once the data is accumulated in two text files, one containing all the data and one contain the data for only the strongest partials, the Process button gets active. When the process button is pressed, a new instrument on Csound gets triggered to process and make sense of the data available.

The first step is to identify all the unique bin values available in the file *maxdata.txt,* upto a certain maximum allowable upper limit. This file will contain bins and their values in the ascending order for each frame, and all the frames are arranged ascendingly as well. Thus there is quite a lot of repetition of bin values for each frame. Moreover, bins present on one frame may not be available in others. What we want is an array of all unique bin numbers which will give us a holistic picture on which partials to concentrate our effort on. This file is parsed once and each value in the CSV file is analyzed and bin values are written to the array *giBinArr*, if the value was previously non existent. Corresponding frequency values are written to another array *giFreqArr*. Though the bin frequencies can easily be calculated using the formula :

$$Center\ frequency\ =\ (bin\ number\ *\ bandwidth)\ -\ (bandwidth/2)\ ,\ where$$
$$bandwidth = (sample\ rate\ /\ 2)\ /\ number\ of\ bins$$

Formula 3.1: Center frequency of each bin

...it is easier and more efficient to fetch from the file where they are readily available.

giBinArr[]

Index	Value
0	16
1	17
2	20
3	22
4	23
....	
(giBinCount)	50

Table 3.1: Sample data in array of bin numbers (giBinArr)

giFreqArr[]

Index	Value
0	187.70
1	161.73
2	204.03
3	238.88
4	241.26
....	
(giBinCount)	537.81

Table 3.2: Sample data in array of frequencies of each bin (giFreqArr)

Now that all the required partials are available, we need to map their behaviour over the duration of analysis. It is no good to just have frequency information as this will give a purely static output without any dynamic changes. We know that all the partials have separate attack times, decay times, and dynamic volume changes in a fairly dynamic sound source. So the next step is to map out the dynamic changes in volume of each of these partials. For this, we need the entire sample set of information. Thus, we retained the previous file *fulldata.txt.* In the first revision of the code, I parsed over this file multiple times and each time, I targeted one of the partials from *giBinArr* and recorded only the amplitude value and mapped it onto a 2-dimensional array of bin number vs amplitude, *giBinAmpEnv*. This meant that I had to iterate over the entire data set the same number of times as the number of unique partials available. For a spectrally rich sound, compounded with a large sound source to analyze, this took a massive toll on the application and rendered the processing process very slow. It would take upto a minute at times to process a source of a few seconds in duration. Thus I revised the code such that all the processing would happen in a single pass, and instead of querying through a lot of redundant data, the algorithm would now disregard and pass over iterations that have already generated the required result. The result was a qualitative and observed reduction of time by more than half. The end result will give us amplitude envelopes of every partial. We can then synthesize these partials as individual sine wave oscillator synths with their own amplitude envelopes and frequencies.

giBinAmpEnv[][]

Index	**Bin**	**Amp**					
		0	1	2	3	...	*giNumWindows*
0	16	0.050	0.263	0.263	0.482		0.025
1	17	0.053	0.299	0.299	0.616		0.048
2	20	0.039	0.130	0.130	0.117		0.002
3	22	0.046	0.199	0.199	0.288		0.022
4	23	0.032	0.067	0.067	0.004		0.000
...							

(giBinCount)	50	0.046	0.075	0.075	0.082		0.055

Table 3.3: Sample data in 2-D array of bin amplitude envelopes (giBinAmpEnv)

To conveniently use the amplitude envelope, we map *giBinAmpEnv* to an array of GEN02 tables, called *giAmpEnv[]*. To debug this information and to check if the values being filtered and mapped are relevant and accurate, the data contained within *giAmpEnv* is formatted and written out to a text file, *debug.txt.* This marks the end of the processing phase of the application. At the end of this phase, we have all the required data, both frequencies and amplitudes of the source file, which when applied with specific DSP processes, gives us the complete power to manage and the mangle the sound as we please.

3.5 Providing a Voice to the Data

Now that we have the data with us, the most obvious thing we can do with this data would be to play it and replicate the original source. To do this, we just need to click the Play button. Once this button is triggered, instrument number 4 is initiated in the code and a recursive User Defined Opcode (UDO) is called which allocates separate memory blocks for generating data for each of the partials contained within *giBinCount*. A *poscil*, a table lookup oscillator is called, per partial, and is multiplied with its associated table of amplitude envelopes. The outputs of these oscillators are mixed together and a global envelope is applied to the mix to smooth it out. The resultant sound is similar to the source sound, but definitely not the same. The threshold value could be adjusted to get a closer resemblance to the source sound, but as the threshold value closes in on the zero mark, the processing becomes heavier and the effectiveness of this method of synthesis is lost. To be silly and to quickly understand the power of this method, a drop down box of common oscillator types are added. For all intents and practical purposes, this should be left as Sine by default. But if other values are chosen, poscil uses the selected table wave shape to produce the sound. Thus, with a simple change in oscillator type, a massive change in spectral character of the source sound is observed, though this may not be pleasant to listen to.

3.6 Snapshots and Synths

As we demonstrated playing the data sample out loud to mimic the original sound, we can build an infinite sustain synthesizer with the spectral characteristics of a part of the original sound. Thus we would disregard the dynamics of the original sound and concentrate on a 'snapshot' from the subset of area that we analyzed to lock in on the partials and their amplitudes present at that point in time. This is essentially freezing the source sound at one point in time and elongating that sound and using that for synthesizing the output sound.

We can easily get frequency and amplitude data from any window considered for analysis, since we have information of all the windows saved. It is just a function of mapping a user input to focus on a single window. Below the Snapshot Window we have a horizontal slider which the user can use to visually target a slice of source to be captured. Once this position is selected, and a MIDI note is triggered either via the the in-built keyboard on the application or via a physically connected MIDI keyboard/interface, the following takes place:

1) The user input on the Seek slider in recognized and mapped to the total number of windows analyzed, and an integral frame number is extracted from this.
2) Depending on this frame number, all the frequencies and amplitudes are extracted for that window.
3) This information is passed to the UDO AdditiveSynth.

Many controls are overlaid and built for processing the output from this synth.

1) An ADSR envelope introduces a common global functionality for attack, decay, release times and a sustain level.
2) Pitch modulation
3) Amplitude modulation

Chapter 4: Testing

This chapter covers some of the behavioral aspects of the application. It involves end-to-end system testing, with a small subset of use cases exploring different source files and different values that the user can dial in. These values include FFT size and threshold, which are detrimental in the analysis process. As mentioned, the test cases cover only a few scenarios, and further testing would need to be conducted, such as boundary value analysis and performance testing to validate the application for bugs and efficiency. Several bugs were found during the development and test processes, but these bugs were hard to replicate on a per-scenario basis and seemed to crop up naturally and without warnings. While inconsistencies in the code definitely contribute to bugs, several versions of Cabbage seem to behave differently and the propagation of bugs is different in different programming environments and operating systems. For consistency and repeatability, the tests were conducted on a Macbook Air 2015, on the version of Csound and Cabbage as mentioned in section *3.1 Explanation of the Build*. All the audio files that have been used in the testing process are linked in the Reference section. Since the audio files have been protected by Creative Commons license, it would not be advisable for me to redistributed them with my application.

4.1 Source Variety

The first test is to find out how the application fares in the real world by providing a variety of timbrally different input source materials and observing the resultant output. This section compares the input spectrum with the output spectrum and analyzes how different the resultant sound is, compared to the sound it is trying to mimicking. All the sample sounds that are used here are taken from *freesounds.org*. Links are provided in the Reference section, with their indices being referred to in this section.

4.1.1 Oboe

> *" Help me Oboe Wan Kenobi. You are my only hope. "*

The oboe is a harmonic wind instrument. We use the sample sound of a person playing a few notes on an oboe in a short interval of time. Here, we try to test a harmonic instrument with variability in pitch and notes and analyze the entirety of the waveform and see if the application picks up pitch changes accurately.

Sound sample: [Reference 16]
Duration of analysis: 8 seconds
FFT size: 2048
Threshold: 0.05
Number of bins selected: 33

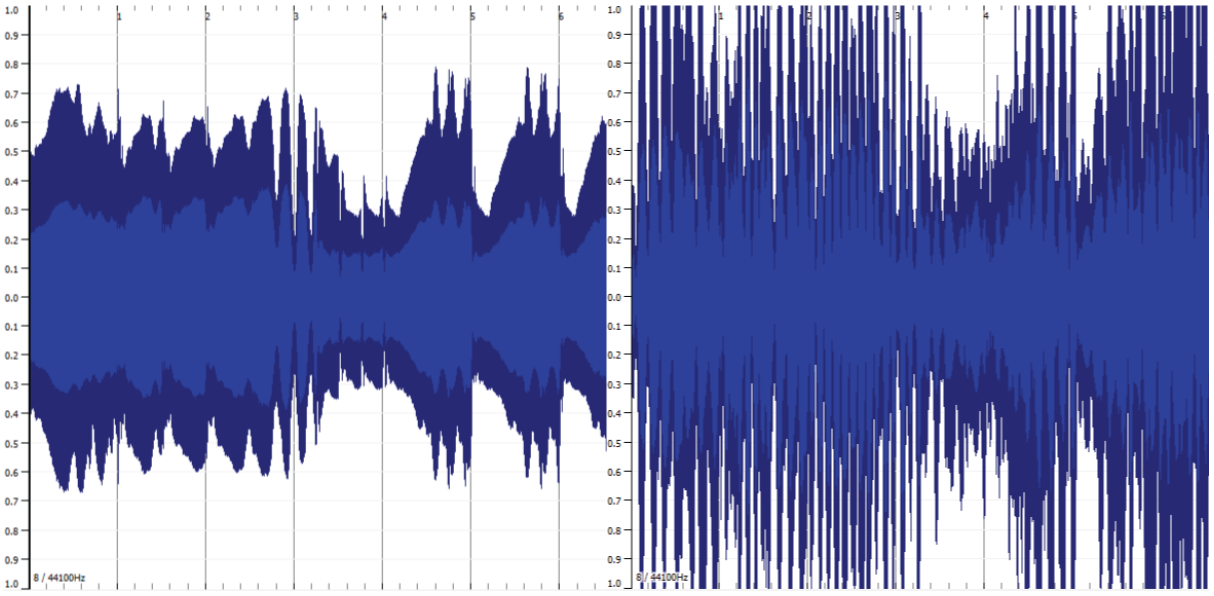

Figure 4.1.1 Waveform of Oboe source input (left) and it's corresponding rendered output (right)

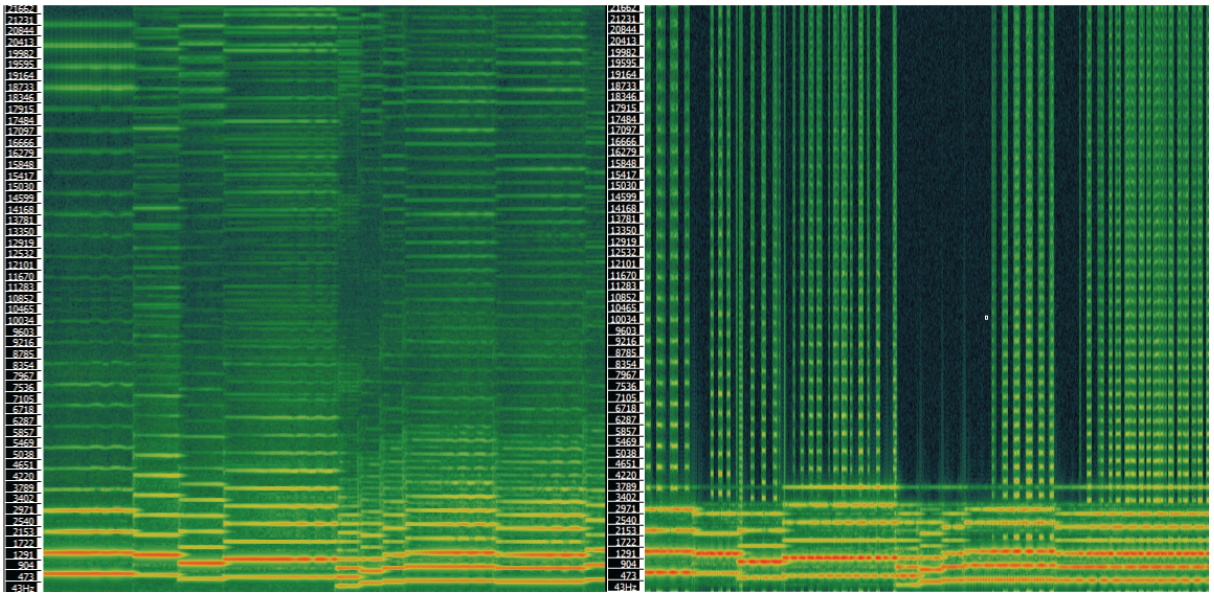

Figure 4.1.2 Spectrogram of Oboe source input (left) and it's corresponding rendered output (right)

We can see that the output waveform is clipping at quite a few places. This should be addressed in the release version which will scale down the render to a suitable level which will not clip. The low frequencies are quite well represented and the changes in frequencies are captured very accurately. The vertical lines on the spectrogram of the rendered file is due to artifacts of clipping.

4.1.2 Tibetan Singing Bowl

The Tibetan singing bowl is a bell-like metallic and semi-harmonic instrument. The audio file taken for this example contains a bit of background noise. It's a good experiment to see if any of that is captured.

Sound sample: [Reference 17]
Duration of analysis: 12.3 seconds
FFT size: 2048
Threshold: 0.05
Number of bins selected: 11

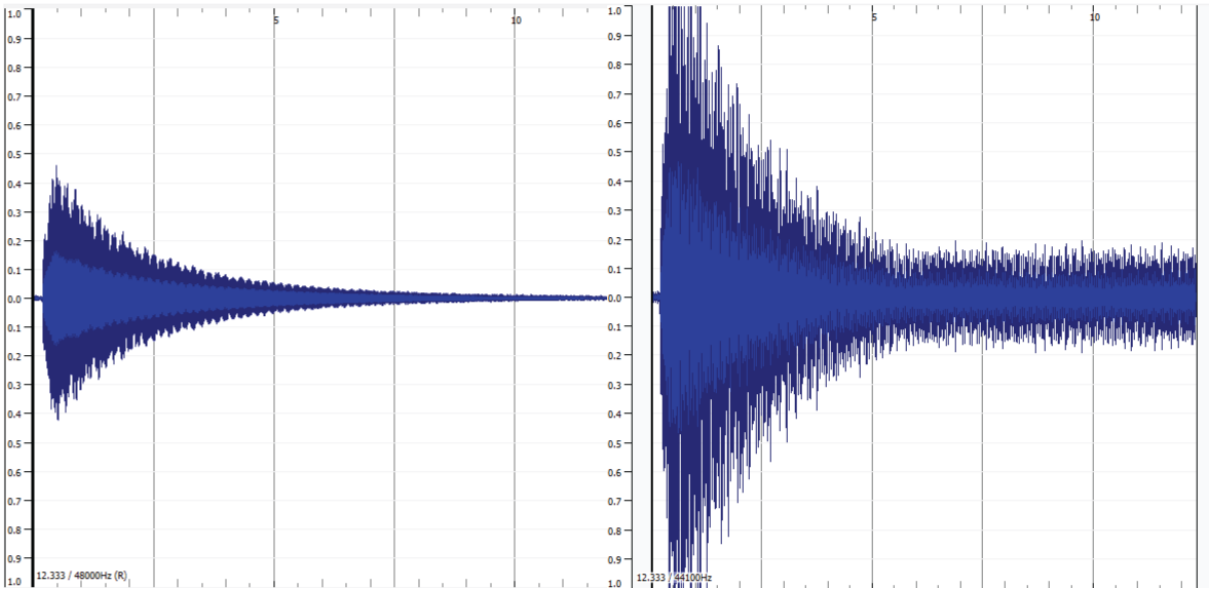

Figure 4.1.3 Waveform of Tibetan Bowl source input (left) and it's corresponding rendered output (right)

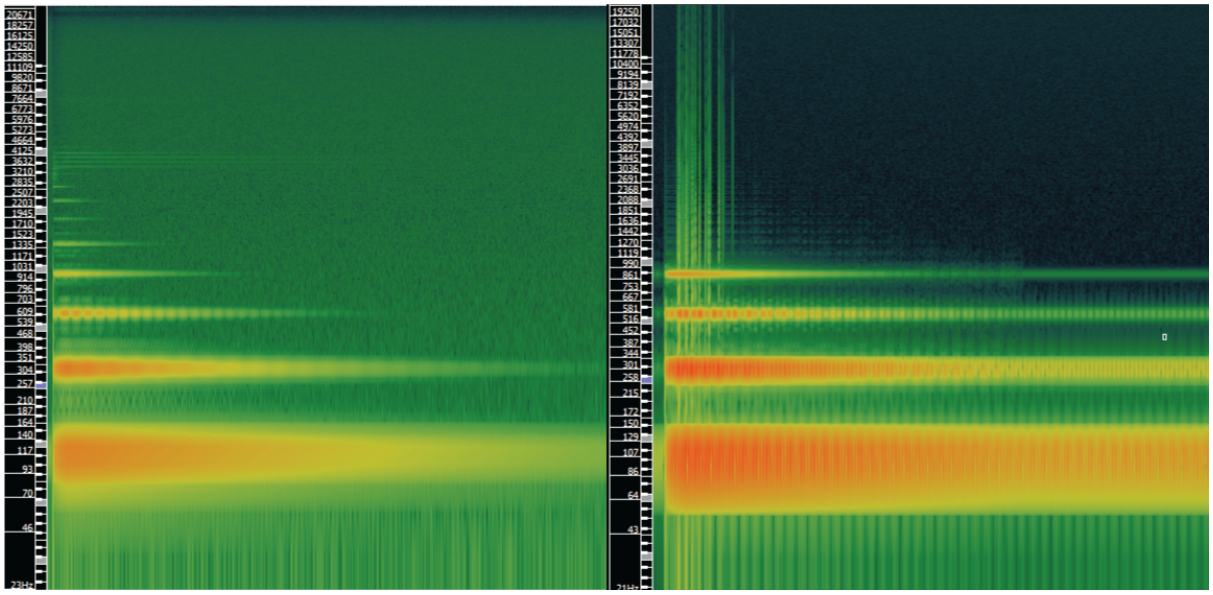

Figure 4.1.4 Spectrogram of Tibetan Bowl source input (left) and it's corresponding rendered output (right)

We can see that the background noise is rejected. So are the upper harmonic partials that are present in the original, since these partials are lower in amplitude than the set threshold. The spectrum is analyzed through the logarithmic scale to zoom in on the lower frequencies where all the activity is present. Another quirk that is observed is that the partials die out toward the end of the file, but continue to persist in the render. Aurally, the output sound is similar to the original, expect that it doesn't have a rich attack sound of the bowl being struck.

4.1.3 Vocals Soprano

No test scenario is complete without including a test sample of the human voice. Here we have a classical soprano singer singing 4 notes as part of a warm up routine. In this test we try to mimic the subtle frequency changes manifested as vibrato.

Sound sample: [Reference 18]
Duration of analysis: 3.6 seconds
FFT size: 2048
Threshold: 0.05
Number of bins selected: 145

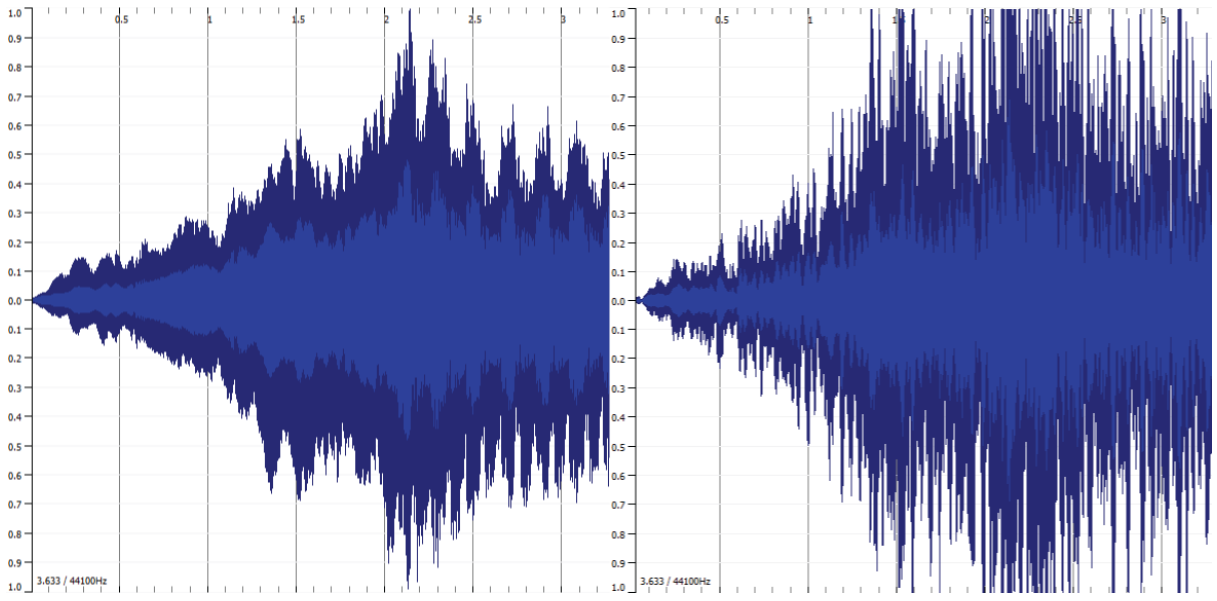

Figure 4.1.5 Waveform of vocal source input (left) and it's corresponding rendered output (right)

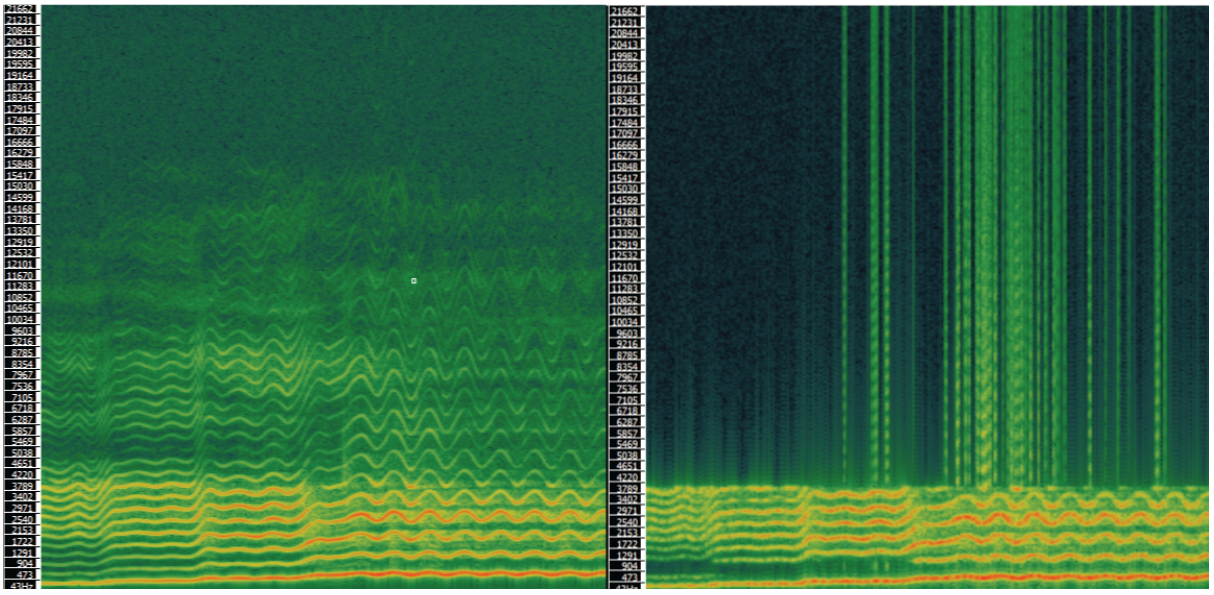

Figure 4.1.6 Spectrogram of vocal source input (left) and it's corresponding rendered output (right)

Again, the low frequencies are quite well established and prominent and thus the application picks these up quite well. The vibrato is also captured accurately, except the top most partial on the rendered file's spectrogram. Here we can see that the spectrum is sharply cut horizontally in-between a potential vibrato. This is because the application views it not as a vibrato, but as bands of frequencies which are ripe for selection. The bands which hold the rest of the vibrato movement is not selected since they have no prominence in any of the analysis windows in the duration selected.

4.1.4 Pink Noise

Pink noise is a signal with a frequency spectrum such that the power spectral density (energy or power per frequency interval) is inversely proportional to the frequency of the signal. It is similar to white noise, with the difference being that the frequencies in pink noise are intensified in the lower frequency range. This is not a particularly clever test in anyway, since simulating broadband noise through is additive synthesis is wasteful. But it does give us a broad idea on what happens when such a signal is passed as the input.

Sound sample: [Reference 19]
Duration of analysis: 5 seconds
FFT size: 2048
Threshold: 0.05

Number of bins selected: 63

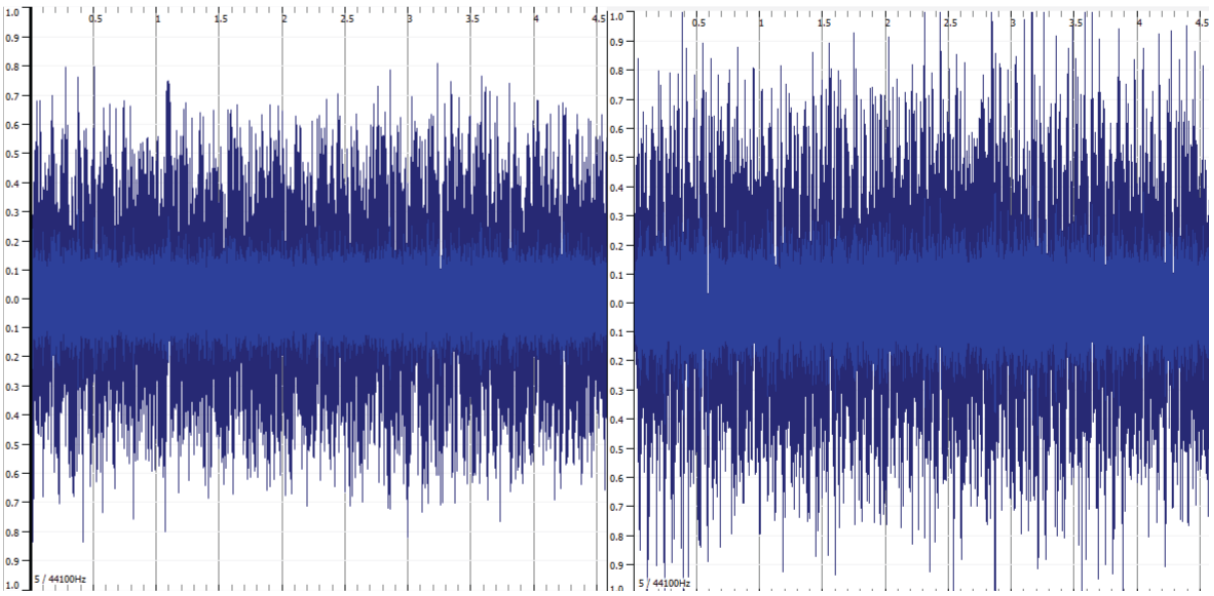

Figure 4.1.7 Waveform of pink noise source input (left) and it's corresponding rendered output (right)

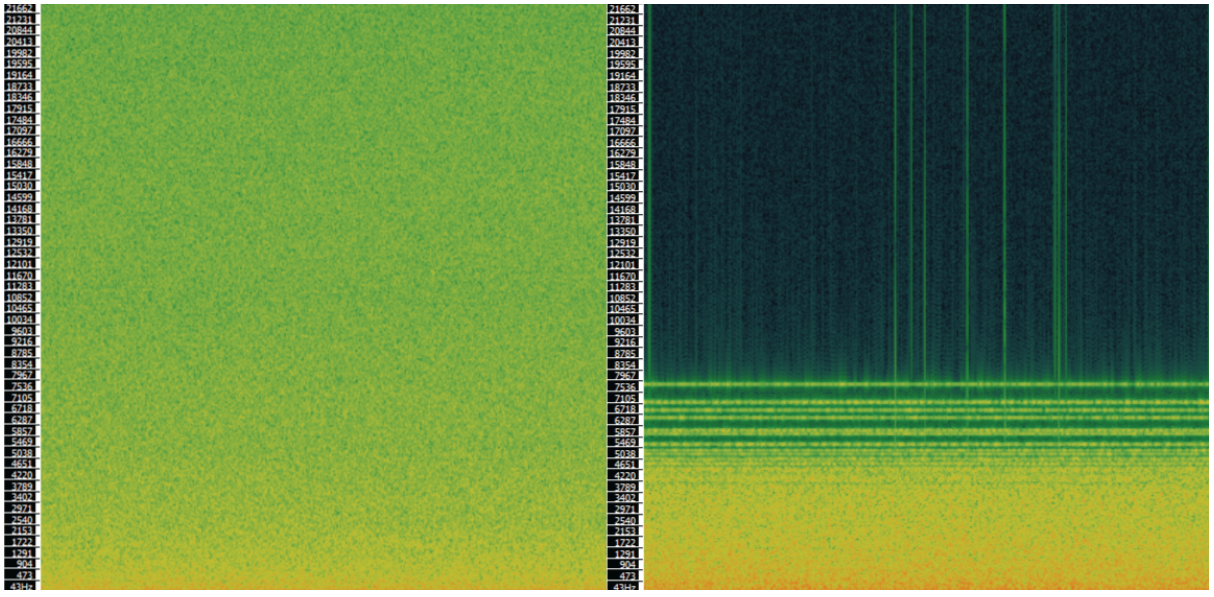

Figure 4.1.8 Spectrogram of pink noise source input (left) and it's corresponding rendered output (right)

A strange observation is that only around 63 bins were selected to be above the threshold, far less than vocal sample at the same threshold level. Ofcourse, if the threshold were low enough, the algorithm would chose all of the (FFTsize/2 + 1) bins. We can also see that the rendered spectrum has distinct lines which represent the bins chosen, rather than the fuzzy noise spectrum observed in the input. In the case of pink noise, the application acts as a low pass filter, with the characteristics of a very sharp cut-off curve.

4.2 Change in FFT Size

The FFT size determines how many windows are considered for analysis, which in-turn defines the frequency resolution factor of the algorithm. For testing the behavior of the analysis with respect to change in FFT size, we keep the threshold of analysis and duration of the source to be constant. We choose a simple bell sound as our test source. Illustrated below, is a table of observed values and spectrum of the resultant output sound.

Constants - Threshold: 0.05, Length of sample: 6.80 seconds			
FFT size	**Analysis Bins**	**Selected Bins**	**Number of Windows**
512	257	38	4685
1024	513	29	2343
2048	1025	27	1172
4096	2049	23	586
8192	4097	23	293

Table 4.1 Change in FFT size test

4.2.1 Expectations

The result of the test was very counter intuitive for me as it produced a result that was the opposite of what I had expected. I expected that the the number of bins that are selected would have been higher as the FFT size increased. Since a higher FFT size correlates to a better frequency resolution, there is more number of bins available for occupation. Partials that would've been squashed into one bin at a lower FFT size, would be free to occupy adjacent bins since there is better resolution. I estimated that a higher number of analysis bins automatically meant a higher number of partials that are available for selection. The data quite clearly proves me wrong.

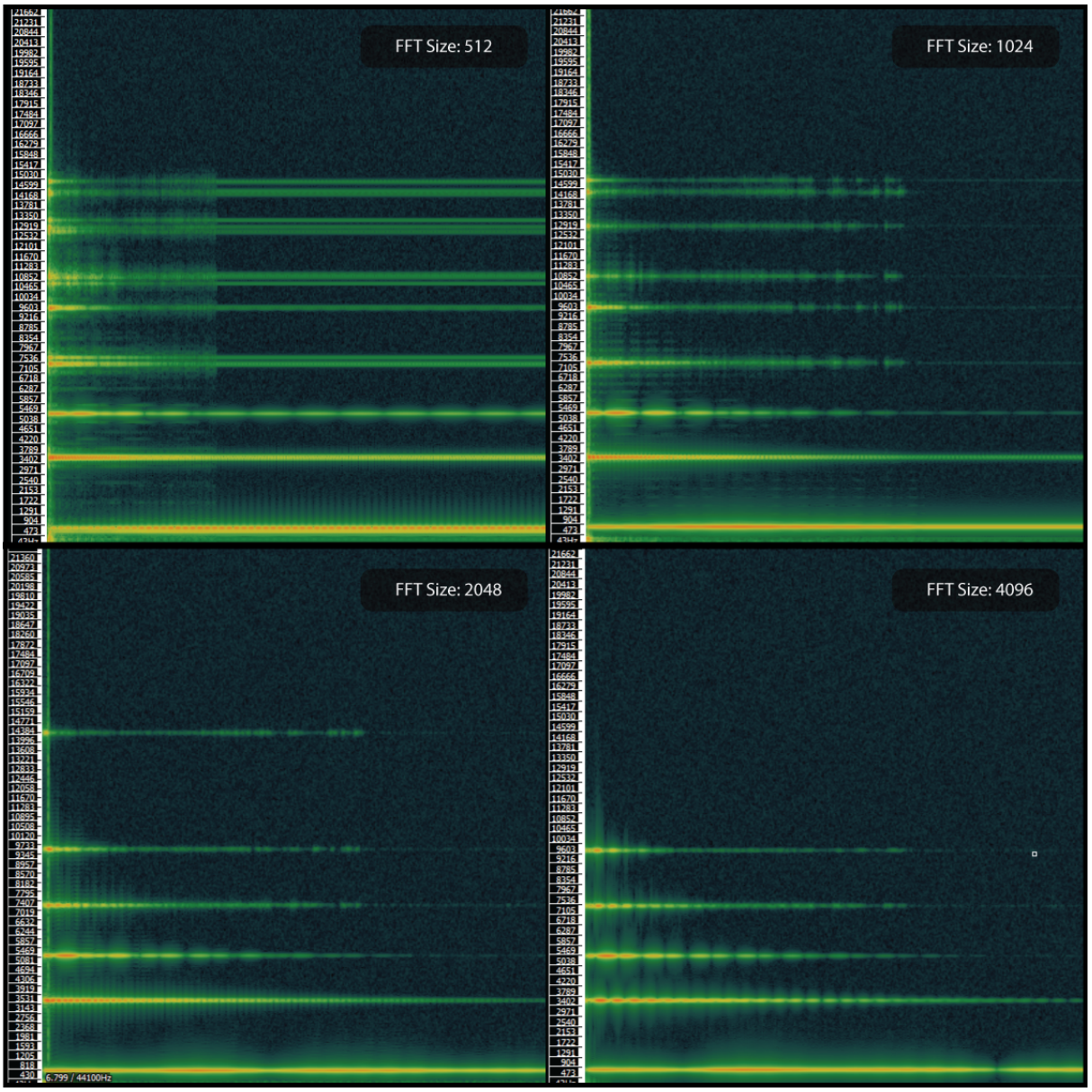

Figure 4.2 Spectrogram of Bell source for different FFT sizes

Another strange artifact that I noticed was that the effect of beating was quite profound for lower FFT sizes. This again proved quite counter intuitive. If one were to listen to the rendered sound file of the bell sample at an FFT size of 512 versus that at an FFT size of 4096, they would hear a slightly more detailed version at a lower FFT size, and the effect of beating being prominent. This is quite evident even from the spectrograms, as the render at 512 has more high frequency content than the 4096 one, and the partials are not very defined and several of them are closer to each other. The 4096 sample on the other hand has a very well defined and sharp spectral response as indicated by the sharp and crisp lines of high intensity.

4.2.2 Explanation

The aforementioned observations can be explained as listed below, but several scenarios and source samples could give rise to different end conditions which are difficult to predict. Nonetheless, a generic explanation is inferred:

1) The excessive high frequency content produced by the render at FFT size of 512 is partly due to artifacts resulting from small FFT sizes.
2) Analysis windows always overlap with each other, and thus, a single partial can appear in 2 or more adjacent bins. There exists a critical bandwidth in the theory of pitch perception which determines the minimum frequency difference needed for the listener to perceive 2 frequencies as fundamentally different. The larger the FFT size, the smaller the difference between the center frequencies of adjacent bins. Thus, partials present on adjacent bins, in higher FFT renders, are perceived by the human ears as a single partial. Thus for low FFT sizes, we tend to perceive these bins as slightly different from each other, which gives rise to the beating effect.
3) The law of conservation of energy states that the total energy of a system is constant while the distribution of it can vary. If we consider amplitude to be a measure of energy, we can deduce that amplitudes across the frequency range needs to be distributed between the available bins. Since at a lower FFT value, the number of bins are low, the amplitudes in these bins are high. As the FFT size gets larger and more bins are introduced, the amplitude gets diluted across the bins. This explains why the lower FFT render had more selected bins than a higher FFT render. It is an interesting relationship wherein to compensate for this dilution, the threshold could be mapped as an inversely proportional function to the FFT size. But to do this accurately is rather experimental and difficult and is left out in this release.

4.3 Change in Threshold

The threshold is the value of linear amplitude which determines the selection of analysis bins. Bins of amplitude above the threshold value are chosen for further processing and the bins with their amplitudes below said value are rejected. The threshold parameter determines a key balance between the accuracy of the resultant sound and the performance and efficiency attained. As the threshold is decreased, the accuracy is

increased but the load and overhead of synthesis increases. Striking a good balance depends on the source sound, whether it is normalized or not, how loud the input file is, how timbrally rich it is and also vaguely on the duration of analysis. We can take the same bell sample that we used in the previous section as the input, and apply different values of threshold and record the results.

Constants - FFT size: 2048, Length of sample: 6.80 seconds	
Threshold	**Number of Selected Bins**
0.01	213
0.02	87
0.025	61
0.03	43
0.05	27
0.1	12
0.2	5

Table 4.2 Change in threshold value test

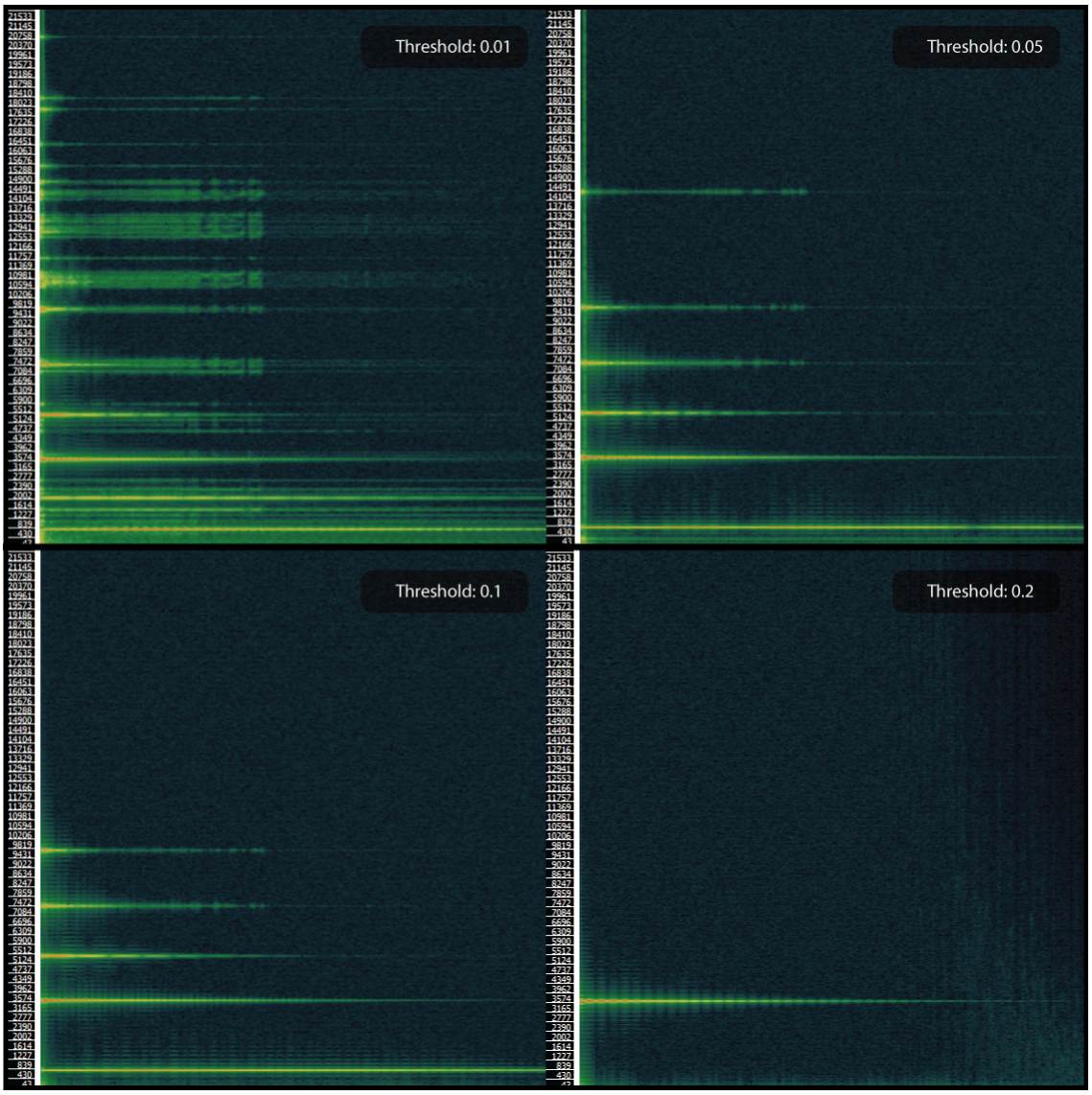

Figure 4.2 Spectrogram of Bell source for different threshold values

The result of the analysis is fairly straightforward and the spectrograms speak for themselves. As the threshold decreases, the number of bins chosen for analysis increases. Care must be taken to change the threshold accordingly for different files and experiment around to get a suitable balance. There is a non-editable variable for limiting the number of bins that can be chosen. Currently, this is set at 600. Thus, as the threshold keeps decreasing, we will hit a barrier on the number of bins that can be chosen and once this barrier is hit, any bins above this range, though they may be prominent, are ignored. Thus this limit should not be reached and a threshold value that keeps the number of selected bins below this limit should be chosen.

Chapter 5: Limitations

Like any other method of synthesis, this approach is marred by a few inconsistencies and dents. There are quite a few limitations to taking this approach of synthesis, all of which can however be solved. The section of Suggested Improvements and Further work takes all the problems listed in this section and tries to provide implementable solutions which can be incorporated in a future release of this project. However, for this current release, the project acts as a Minimal Viable Product (MVP) and a Proof of Concept (POC) which can be cloned and replicated in any respect to improve upon the existing structure or propose to change or implement changes as and when deemed appropriate and needed.

5.1 Simulation of Transients and Attack

A transient is a short-duration signal that represents a nonharmonic attack phase of a sound source.

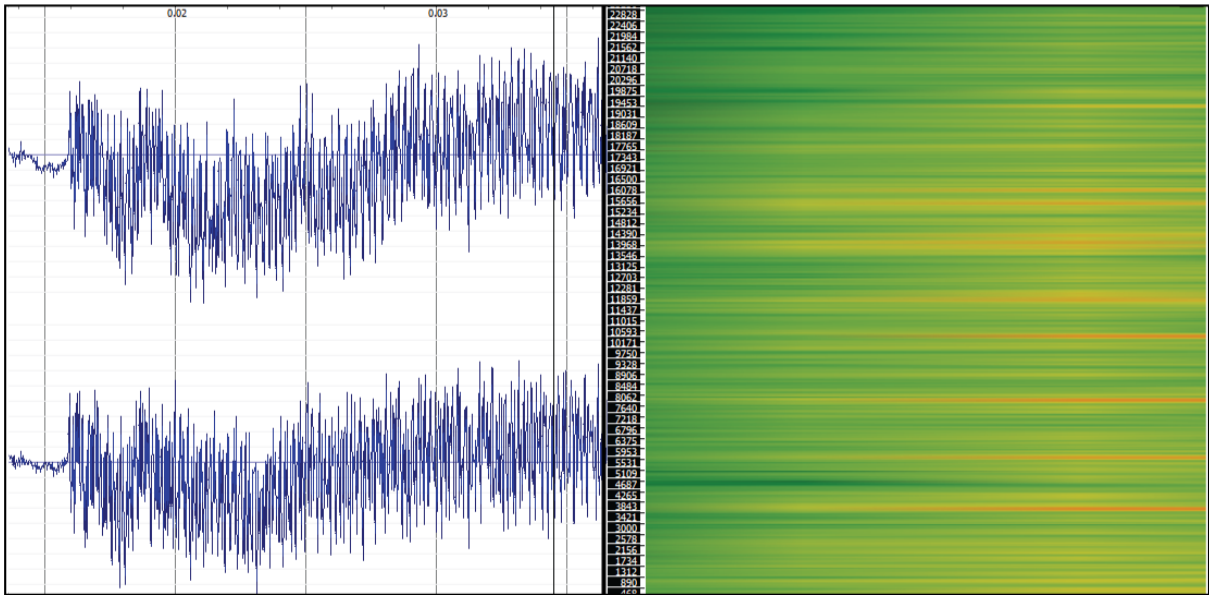

Figure 5.1 Transient phase of the bell sample

Transients contains a high degree of nonperiodic components and a higher magnitude of high frequencies than the harmonic content of that sound. Transients do not directly depend on the frequency of the tone they initiate.

Transients are hard to simulate since they contain a very rich spectrum of sound which can almost be accounted as short bursts of white noise. When the algorithm encounters this signal, it would consider it as any other signal and start listing and documenting the frequencies or partials whose strengths are are above a certain threshold. Since the attack phase contains potentially thousands of partials at high amplitude, all these partials become candidates for analysis. There is a failsafe implemented to avoid taking in too many partials in for analysis and this skews the result since it is an incrementally ascending analysis. The algorithm starts picking values from 0Hz upwards, and stops the analysis when it encounters a partial whose count is that of the designated higher limit. Later, in the processing phase, when all these partials which are selected are subject to dynamic amplitude analysis over the duration of the source selection, only a few window frames where the transient exists are highlighted with activity and many partials would have zero amplitude values throughout the rest of the window frames. Since several potentially valid high frequency partials were never considered for analysis, the effect of the resultant sound would be similar to that of the source sound being run through a low pass filter. The effect of low pass depends on several key criteria:

- **The partial limit:** This is quite straightforward, as partial limit increases, the cut off frequency of the virtual low pass filter increases.
- **FFT size:** As the FFT size decrease, the cut off frequency increases. This is because the frequency resolution decreases as the FFT size decreases, and this means that a bigger chunk of frequencies are considered for analysis, which decreases the overall number of partials which exist. Ideally, to avoid this conflict, the partial limit should be a function of FFT size.
- Transient strength and richness of sound which follows the attack.

5.2 High computational overhead for polyphony

Additive synthesis is a fairly straightforward approach and should not significantly hamper performance on a fairly modern CPU. But with the suggested approach, depending on how spectrally rich the source file is, and the value of partial capping limit, there may be tens to thousands of oscillators that are required to monophonically recreate the sound. But when polyphony is used compounded with a large release time and a frenzied user who plays a lot of notes, this number could significantly increase and cause audio dropouts to occur since the processor is just not able to handle the load. These audio dropouts may occur as glitches and pops with audio breaking from time to time, or could potentially cause the application to crash.

5.3 Mono or Stereo ?

Currently, the application's output is false stereo, meaning that the input source to the application is always mono, and the application processes this signal and generates 2 signals, virtually indistinguishable from each other for the stereo left and right. Though the input source file may be stereo or multichannel, the individual channels are summed into a mono file before being handed over to the application for analysis. This is done to avoid the computational overhead of analysing 2 separate signals. It would virtually double the overhead required and the time taken to analyse and process. However, the user may want to retain the integrity of source file and generate output which is a reflective stereo mimic of the input. This is currently not supported by the application and is viewed as a limitation. A future release may address this issue.

5.4 Bugs!

Like any beta release of an application, expect a few bugs to be present within the normal operation as well as extreme/boundary value testing of the application. Apart from the bugs on the application, different versions of Cabbage might behave differently when exposed to the application, since Cabbage itself is under constant development. However, on the versions of Csound and Cabbage mentioned in the introduction to The application and the Algorithm section, and with the operating system that it was developed it, namely Mac OSX Sierra 10.12, the application should be relatively stable. Some of the bugs in the application are hard to address since they are not reproducible for similar input conditions. Further investigation is needed to address all the possible vulnerabilities and flaws in the code and further testing is needed as well. But for now, when the behavior is not as expected, re-running the application should resolve the problem.

5.5 Volume and Normalization

The values that the Threshold knob can take were decided keeping a normalized input signal in mind. A normalized signal is one which has its amplitude scaled such that the highest point on the signal has the value of 0dbfs or 1.0 on the linear amplitude scale. Normalized signal ensures amplitude uniformity among inputs and the threshold knob would behave objectively and consistently for the same value over different signals. Without normalization, an input signal which is weak, might not have sufficient energy in terms of amplitude for even the lowest value of threshold to accurately represent the signal. A future release may include an option to normalize the input signal before the application analyses it.

Chapter 6: Suggested Improvements and Further Work

This section illustrates algorithms and procedures to improve the application in the future. Some suggestions provided may be quite challenging to implement, but possible nonetheless. Some of the suggestions offered counter the section which listed out the limitations of the application, while others suggest ways of optimizing the application of making it more efficient.

6.1 Avoiding transients in Analysis

The problem statement is well documented in the Limitations section. In brief, we need to a way to exclude transients is the selection to be analyzed. Let's say, a certain section of the selection had some interesting sonic elements that we want to capture but it starts with a transient attack phase or has bursts of transient glitches or pops which we need to avoid. In such a case the follow steps are proposed:

- Consider placing a switch on the user interface to turn the transient capture on or off. This algorithm should ideally be an optional choice as it changes the integrity of the resultant sound as opposed to the source.
- The very first step of analysis remains the same, where all the data is collected and written down to *fulldata.txt*.
- The next step is to filter out all the partials which have their strengths above a certain threshold value, *while not placing a cap on the maximum number of partials allowed*. This completes the analysis, after the filtered result is written to *maxdata.txt*.

- While processing this information, we filter this list further to extract all the unique partial values that are present in *maxdata.txt.* This step remains the same as that employed in the current release version.
- The next step is to parse through the bulk data present in *fulldata.txt* and map out the amplitude values of each of the partials into a 2-D array. This array will contain all the partials captured and their individual amplitude over the duration of the selection. Some of these partials may include the transient frequencies we must avoid.
- Next, we parse through the 2-D array and record the number of zero values present in the amplitude envelope of each partial, and record that in a separate one dimensional array. This resultant array will contain a single count value which represents the number of times the amplitude envelope is zeroed during the selection. We will also have a count of the total number of values available in this amplitude envelope, with which we can compare and scrutinize the former value. Since values in the amplitude envelope are never really zero, we can take the baseline value of zero to be -60db or -120db.
- Next, we can define a percentage value which defines the percentage of time with respect to the total selection time, below which a partial can be considered to be transient or vestigial. For example, let's say that there are 250 distinct points on the amplitude envelope for a time selection of 1 second. This means that 1 second is split into 250 windows, over which we can observe the amplitude value of each bin in each of these windows. Now, we want to determine what would be an ideal transient duration. Let's say, anything less than 0.1 second in duration is a transient. This equates to 10% of the time duration of 1 second. Translating this number to the windows present, it would mean that there would be non-zero values in less than or equal to 25 windows. Thus, 225 windows or more should have zero values for that particular frequency to be considered transient. This percentage value is a purely experimental value, and would require trial and error to accurately phase out transients. If the percentage value is very low, it would not eliminate all the transients present, and if it is too high, it might affect fast and quick pitch changes in the source to be excluded.
- Now that we have a function which defines if a particular value in iNumZeros[] should be considered for synthesis, we can parse through this array, apply the function and publish the determinant value of false(0) or true(1) in a separate one

dimensional array called iPartialSelect[]. This will have the same size as that of iNumZeros, but would contain only zeros or ones.

iNumZeros[]

Index	Value
0	12
1	2
2	240
3	238
4	84
…..	
giBinCount	159

Table 6.1: Sample data in array of zero amp values (iNumZeros)

iPartialSelect[]

Index	Value
0	1
1	1
2	0
3	0
4	1
…..	
giBinCount	1

Table 6.2: Sample data in array of boolean values for partial selection (iPartialSelect)

- Now, we can modify the AdditiveSynthesis UDO to implement a check against iPartialSelect[] and only the partials that are marked as 1's are included in the synthesis and allocated memory.

6.2 Adding Artificial Transients

This section describes a mode which is in essence opposite to the previous section which described a way to eliminate transients in sounds. At times, it is aesthetically important to have transients in the synthesized sound for it to sound more authentic. This is especially true if it is a percussive sound, like a bell striking or a guitar string being plucked.

As we have previously established, transients are basically short bursts of broadband noise. This impulse can be finely controlled with a few graphical interface parameters.

Adding this impulse to the start of our synthesized sound could give us an imitation of striking or percussive sound. We can use Csound opcodes *noise* or *pinkish* to generate white noise. The parameter are explained below:

Strength: This controls the amplitude or strength of the broadband noise.
Decay Time: This controls the time taken for the the noise to decay to zero. This should ideally be an exponential decay.
Filter: This controls a damping factor to filter out the highs through a low pass filter.

6.3 Wavetable synthesis

Since additive synthesis is performance heavy as mentioned several times already, we can adopt a different method of synthesis, which differs slightly from the original method. Oscillators such as *poscil* require a table to refer to while generating sound. They lookup a table containing one period or cycle of a wave and oscillate over it the same number of times per second as the frequency value provided. By default, *poscil* looks up a period of a sine wave. But a custom wavetable can be provided to this opcode. But a wavetable is a static representation of sound. Through wavetable synthesis, we cannot recreate the dynamics or the spectral morphing of sound as observed in the source, but can only be utilized for re-creating static timbre, as observed in the Synth section of our application.

We already have the ingredients to create the wave table, ie, the frequencies and their corresponding amplitudes at a certain window location. We can convert the frequencies to partial numbers by choosing a base frequency or a fundamental frequency. This can be any frequency in our list of frequencies, but it is good practice to choose the lowest frequency. On the basis of this fundamental frequency, the partial numbers of other frequencies can be calculated by:

$$partial\ number\ = selected\ frequency\ /\ fundamental\ frequency$$

This could result in partial numbers which may not be integral numbers. This is perfectly fine as the Gen routines we use can make use of fractional partial numbers.

GEN09 / GEN19

As illustrated in the Csound manual, the syntax or markup of GEN09 is as shown below:

```
f # time size 9 pna stra phsa pnb strb phsb …
```

Where *time* - The time offset from the invocation of this statement to when the table is actually created.
size - The size of the table in samples. Should be a power of 2 for efficiency.
pna, pnb… - Partial number relative to a fundamental of sinusoid a, sinusoid b, etc. Must be positive, but need not be a whole number.
stra, strb... - strength of partials *pna, pnb*, etc. These are relative strengths, since the composite waveform may be rescaled later.
phsa, phsb... - Initial phase of partials *pna, pnb,* etc., expressed in degrees (0-360).

GEN19 is similar in every way, except that there is an added argument for DC-offset. These Gen routines look good on the face of it, but there is no real way to recursively put the partial number and strengths on to the syntax. Thus, we need to make use of GEN34.

GEN34

```
f # time size  34  src nh scl
```

Where *src* - Source table number. This table contains the parameters of each partial in the following format, in possibly a GEN02 table:
stra, pna, phsa, strb, pnb, phsb, …
nh - Number of partials
scl - Amplitude scale.

But the problem then arises that when trying to generate inharmonic sound using this approach, we observe clicks and pops throughout the waveform. This is because inharmonic sound is inherently not periodic in nature, and when we try to introduce periodicity to it, there are sudden jumps to zero values that are made while drawing the table. This occurs in essence, when partial numbers are non integral. To solve this problem, we multiply all the partials by a large value, say 1000, and then round them off to the nearest integer. Now the fundamental frequency has to be divided by this same number, 1000, to nullify the scaling. This should accurately represent the non periodic sound without a large loss in integrity and with clicks and pops very occasionally.

Chapter 7: Conclusion

The research conducted in this thesis proved to be quite an eye opener for me in several fields. Firstly, it equipped me with the scientific method and knowledge of conducting research on an area which hasn't been explored in great detail. It helped me understand how to navigate in uncharted waters. It also provided valuable insight into the topic of discussion, which involved analysing real world sounds and recreating them from first principles, and enhanced my curiosity in this domain of Digital Signal Processing research.

In the end, I was successful in what I set out to do - to automate the process of selection of frequency partials from analysing an input source file and faithfully mimic and reproduce an output with a degree of variability that is acceptable. I was also able to provide an interface and mechanism to realize this process with relative ease. I strived to make the algorithm and application as portable as possible within the constraints of time that was set, such that it could be reused for other applications and programs. There are certainly ways to go before this application is commercial-ready. But it's a start.

The true power of this approach of imitative synthesis has only been skimmed across in the discussion of this research. Data is power. With the data that is acquired, filtered and mapped, in a way which can be utilized in processing the signal, it only requires a creative mind to figure out what to do with this data. Many interesting possibilities could arise in terms of sound design and sound manipulation which retains an essence of the original input. It gives the sound designer a palette of information to work with and sculpt a sound that they desire.

I hope to invest more time in this project in the future and tweak its performance, accuracy and portability such that it is more accessible and more reliable for others to use.

References

1. **GitHub: Mimic - An Imitative Synthesizer**
Akash Murthy, 27 August, 2015
https://github.com/Thriffleganger/MimicImitativeSynth
2. **Mimesis and Shaping - Imitative Additive Synthesis in Csound**,
Joachim HEINTZ, Incontri HMTM Hannover,
joachim.heintz@hmtmhannover.de
http://lac.linuxaudio.org/2011/papers/20.pdf
3. **The Canonical Csound Reference Manual**
Version 6.06, Barry Vercoe, MIT Media Lab
http://www.csounds.com/manual/html/
4. **Csound Download Page**
http://csound.github.io/download.html
5. **Cabbage Audio Home Page**
Rory Walsh
http://cabbageaudio.com/
6. **Sonic Visualizer Home Page**
Centre for Digital Music, Queen Mary, University of London
http://www.sonicvisualiser.org/
7. **Know Your Language**
Michael Byrne, July 22, 2015
https://motherboard.vice.com/en_us/article/pga53v/know-your-language-csound-may-be-ancient-but-its-the-audio-hacking-future
8. **Tools for Real-time Spectral Processing**
PVS opcodes, The Csound Community
http://ecmc.rochester.edu/ecmc/docs/csound/csound-manual-5.14/SpectralRealTime.html

9. Introduction to Computer Music - Chapter Four: Synthesis
Indiana University, Center for Electronic and Computer Music
Prof. Jeffrey Hass, 2013
http://iub.edu/~emusic/etext/synthesis/chapter4_pv.shtml
10. Music and Computers - A theoretical and historical approach. Section 4.2: Additive Synthesis
Phil Burk, Larry Polansky, Douglas Repetto, Mary Roberts, Dan Rockmore, 2011
http://cmc.music.columbia.edu/MusicAndComputers/
11. Better Explained: An Interactive Guide To The Fourier Transform
Kalid Azad, December 2012
https://betterexplained.com/articles/an-interactive-guide-to-the-fourier-transform
12. Gaussianwaves: Window Functions – An Analysis
Mathuranathan, 10 February, 2011
http://www.gaussianwaves.com/2011/02/window-functions-an-analysis/
13. EE Times: FFT convolution and the overlap-add method
Steven W. Smith, Ph.D, 6 July 2007
http://www.eetimes.com/document.asp?doc_id=1275412
14. Wikipedia: Pink Noise
https://en.wikipedia.org/wiki/Pink_noise
15. FreeSound: Reception Bell
Contributor: cdrk, 21 February 2015
https://freesound.org/people/cdrk/sounds/264594/
16. FreeSound: 120 Oboe
Contributor: Thirsk, 21 May 2011
https://freesound.org/people/Thirsk/sounds/120994/
17. FreeSound: Tibetan Singing Bowls
Contributor: the_very_Real_Horst, 20 June 2014
https://freesound.org/people/the_very_Real_Horst/sounds/240934/
18. FreeSound: FemalePhrase1
Contributor: HerbertBoland, 28 January 2007
https://freesound.org/people/HerbertBoland/sounds/30084/
19. FreeSound: Independent Pink Noise
Contributor: NoiseCollector, 12 July 2005
https://freesound.org/people/NoiseCollector/sounds/4029/