UNIVERSITY OF CALIFORNIA, SAN DIEGO

# Objective Analysis of Time Stretching Algorithms

A dissertation submitted in partial satisfaction of the
requirements for the degree Doctor of Philosophy

in

Music

by

Cooper Everett Baker

Committee in charge:

Miller Puckette, Chair
Anthony Burr
Thomas Erbe
Tamara Smyth
Shahrokh Yadegari

2015

The Dissertation of Cooper Everett Baker is approved, and it is acceptable in quality and form for publication on microfilm and electronically:

_____

_____

_____

_____

_____

Chair

University of California, San Diego

2015

TABLE OF CONTENTS

LIST OF GRAPHS

# VITA

| | |
|---|---|
| 1996 – 1998 | Violin Performance, DePauw University, Greencastle, Indiana |
| 1998 – 2000 | Music Technology, University of Oregon, Eugene, Oregon |
| 2002 – 2004 | Bachelor of Fine Arts, Music Technology, California Institute of the Arts, Valencia, California |
| 2004 – 2006 | Master of Fine Arts, Composition – Experimental Sound Practices, California Institute of the Arts, Valencia, California |
| 2008 – 2015 | Doctor of Philosophy, Music, University of California, San Diego, California |

# PUBLICATIONS

Baker, Cooper, and Tom Erbe. "Pd Spectral Toolkit." *Proceedings of the International Computer Music Conference* (2013): 410-413.

# FIELDS OF STUDY

Major Field: Computer Music

Studies in Spectral Analysis
Professor Miller Puckette

Studies in Spectral Signal Processing
Professor Tom Erbe

Studies in Audio Programming
Professors Miller Puckette, Tom Erbe, and F. Richard Moore

ABSTRACT OF THE DISSERTATION

Objective Analysis of Time Stretching Algorithms

by

Cooper Everett Baker

Doctor of Philosophy in Music

University of California, San Diego, 2015

Professor Miller Puckette, Chair

This research describes and implements new techniques for objective analysis and comparison of common time stretching algorithms. A set of test signals and samples is used in conjunction with spectral analysis to generate objective error measurements and subsequent comparisons of the algorithms. The time stretching techniques are treated as black-box processes, and are regarded as members within a general family of analyze-

modify-resynthesize algorithms. This black-box approach of output signal analysis provides a basis for objective evaluation of the more general class of algorithms. Six common time stretching algorithms are analyzed: the Classic Phase Vocoder, Oscillator Bank Phase Vocoder, Peak Tracking Phase Vocoder, Phase-Locked Vocoder, Synchronous Overlap Add technique, and Overlap Add technique. The six algorithms are then compared within a uniform framework to provide greater insight regarding the behavior of their output signals. Subjective listening evaluations are also performed, then all results contribute to the development of criteria that evaluate and rank the algorithms' suitability in various signal processing situations.

## 1 – Introduction

Time stretching algorithms change the duration of a signal without altering the signal's pitch. Several commonly used versions of these algorithms are considered canonical and have been published in the musical signal processing literature, but an objective comparison of the signals generated by these algorithms does not exist. Six of these algorithms are evaluated here, and are chosen due to their primacy within the two main families of time stretching techniques. Most of the included algorithms serve as general foundations for more specialized techniques, and analysis of their output may provide insight about common time stretching algorithms, while also illuminating the behavior of more specialized techniques. This research focuses on new methods of analysis for objective evaluation of these common time stretching algorithms, and describes the similarities and differences of their output signals in detail.

Time stretching algorithms each have their own idiosyncratic implementations which result in slightly different output signals. Since the output signals are of primary interest for this research, the time stretching techniques are considered to be black-box algorithms, and observations are generated based solely on comparisons of these output signals. However, as it is instructive to understand how various artifacts within the output signals might arise, so the internal operation of each algorithm is also described in detail. The combination of descriptions and output evaluations is intended to provide insight for creative practitioners and algorithm designers who need to employ a common algorithm or implement a new time stretching technique.

Typically, time stretching algorithms are chosen for particular applications based on theoretical knowledge of how they work, and how they might operate on input signals.

For example, a granular technique might be better suited for noisy, transient filled percussion, while a spectral technique might work better with clearly defined partials that evolve over time. These assumptions are based on theoretical strengths and weaknesses that are fundamental to the two families of granular and spectral techniques. Spectral techniques operate within the frequency domain and are well suited to pitched signals with little noise, but spectral techniques suffer from time versus frequency trade-offs, where good frequency resolution translates to poor time resolution and vice versa, making noisy transients difficult to handle. Granular techniques are unable to operate in the frequency domain, and operate instead in the time domain, without suffering from the same time-versus-frequency problems while more easily handling noisy signals with percussive transients. However, granular methods have difficulty manipulating the frequency spectrum, which is of primary importance when operating on pitched signals.

In general, internal operations shared among the members of each family of algorithms are somewhat similar, but specific techniques at work within each algorithm can differ significantly, necessitating a black-box approach that evaluates only the algorithms' output signals, and provides meaningful comparisons regardless of how the algorithms work internally. This approach considers any type of time stretching algorithm to be part of a general class of analyze-modify-resynthesize techniques that change signal duration without affecting frequency content. The black-box approach enables general observations about this class of techniques, and facilitates judgments about their strengths and weaknesses, regardless of the algorithms' individual implementations.

In order to objectively evaluate and compare output signals of the algorithms without considering their inner workings, three spectral comparative analysis techniques are employed. These techniques work in conjunction with special test and evaluation signals to illustrate how the time stretched output signals differ from ideal output signals, and how the output signals differ from each other. Additional analysis techniques that evaluate time stretched real-world signals are also applied to provide deeper insight into the stretching algorithms outside of the special test conditions.

All of these analysis techniques generate observations and comparisons based on magnitude. One technique discards time and considers average spectral magnitude, while the other two focus on spectral magnitudes as they change over time. Spectral analysis provides information about both magnitude and phase, but phase can be unpredictable, and different algorithms can manipulate phase very differently to create similar output signals. Magnitude is relatively predictable, and the behavior of magnitude versus frequency over time can be reliably predicted and compared. Consequently, magnitude is chosen as the primary characteristic of study for this research.

This document focuses on these topics in detail, and includes mathematical notation to describe the time stretching and analysis algorithms, as well as extensive graphs to help illustrate the analysis results. The rest of the document is organized into six chapters with various subsections that cover each topic. Chapter Two discusses the special analysis signals and real-world signals that are used throughout this research. Chapter Three describes the time stretching algorithms, and contains subsections for each phase vocoder and granular technique, with supporting mathematical notation. Chapter Four introduces the analysis techniques and describes each one in detail with text and

mathematical notation, while also describing the parameters that are used to configure the analysis and time stretching algorithms while they operate on the signals. Chapter five contains extensive graphs and comparative discussions of each signal group that is emitted by all of the algorithms, organized by analysis technique. Chapter six provides a summary of the analysis results and general ranking of each algorithm in terms of stretched material.

As a supplement to the main text, two appendices are provided. Appendix A contains all of the graphs generated during this research. Appendix B contains every MATLAB script used to perform the signal generation, time stretching, analysis, and comparison operations. These supplemental materials and all audio files are also available for download at http://dissertation.cooperbaker.com.

## 2 – Audio Files

Two classes of audio files are stretched and analyzed. The first consists of three specially designed synthetic waveforms, intended to make the differences between stretching algorithms easily observable. The second class consists of three recorded audio samples taken from diverse sources. The samples were selected for specific audio features, which can be challenging to stretch, to illustrate how different algorithms handle these challenges. Additionally, the samples were chosen to provide stretched material containing perceptually significant and easily discernable differences. All audio files have a sampling rate of 44100 Hertz, the synthetic waveforms use 32 bit floating-point samples, and the recorded samples use 16 bit fixed-point samples.

### 2.1 – Synthetic Waveforms

Synthetic waveforms may be created with specific features and durations. Ideally, a sine wave that has been stretched should simply result in a longer sine wave with the same frequency and amplitude. By using specially designed synthetic signals to test a time stretching algorithm, the ideal output signals may be synthesized, and used to perform comparative evaluations of the algorithm's actual stretched output signals. Three synthetic waveforms were designed to test the time stretching algorithms: a sine wave, a sine wave mixed with a sinusoidal frequency sweep, and a band limited square wave.

The sine wave is created with a frequency of 1000 Hertz at full amplitude to provide a pure, steady signal for the time stretching algorithms. Any amplitude variations resulting from time stretching are easily detectable, and there is no need to account for anything other than the single sinusoid. The input waveform file is displayed

in Graph 2.1.1, and the MATLAB scripts that generate the input and ideal files are contained in Appendix B, Sections 8.1.1 and 8.1.2.

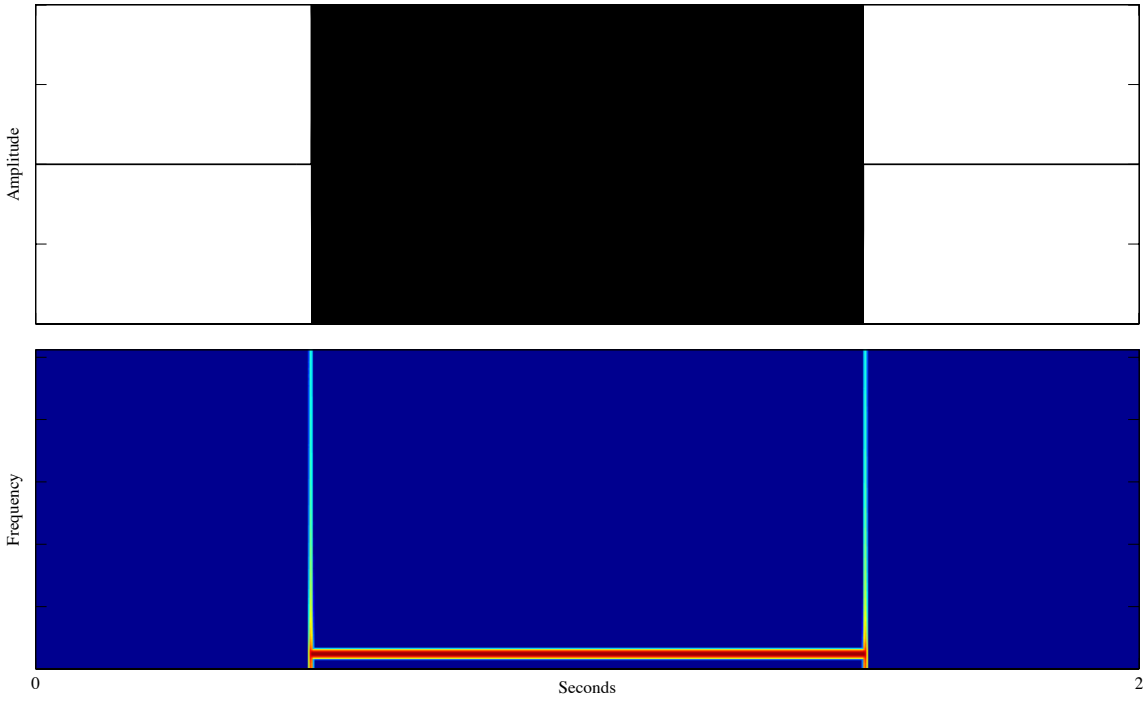
The band limited square wave is synthesized at full amplitude with a fundamental frequency of 1000 Hertz and nine harmonics, for a total of ten evenly spaced sinusoids from 1000 Hertz to 10,000 Hertz within the audio spectrum. This arrangement provides another steady signal for amplitude analysis, as well as insight into the behavior of the stretching algorithms as they manipulate a harmonic spectrum. The input waveform file is displayed in Graph 2.1.3, and the MATLAB scripts that generate the input and ideal files are contained in Appendix B, Sections 8.1.3 and 8.1.4.

The sine wave and sinusoidal frequency sweep consist of a 1000 Hertz sine wave mixed with a logarithmically swept sinusoid that moves from 50 Hertz to 21,000 Hertz, resulting in a file with full amplitude throughout. These values cause the sweep's frequency to cross the sine wave's frequency directly in the middle of the file, making the crossing region easier to find during analysis. This arrangement of sinusoids is difficult to resynthesize and highlights errors that arise during the time stretching process. The input waveform file is displayed in Graph 2.1.2, and the MATLAB scripts that generate the input and ideal files are contained in Appendix B, Sections 8.1.5 and 8.1.6.

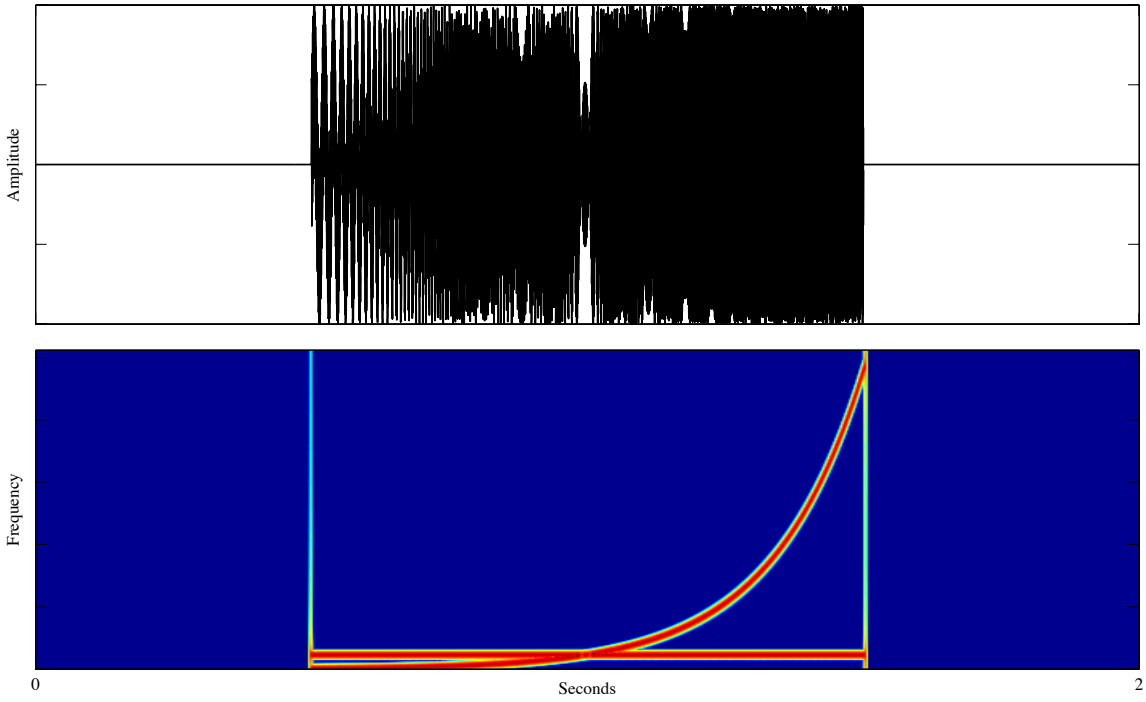
The synthetic waveforms all occupy the middle half of their respective files, with silence at the beginning and end of each file. Each waveform starts with a phase of zero, so that there are no discontinuities between silence and sound. The silent regions are included to provide transitions between silence and signal for analysis, in order to observe how each time stretching algorithm behaves when the waveforms begin or end.

Two sets of the three waveform files are used. One set is fed to the algorithms as input, and the second set is used as the ideal output for comparison. The first set of input waveform files begins with a half-second of silence, followed by one second of waveform, ending with another half-second of silence for a total length of two seconds. The ideal output files are twice as long (for analyzing double-length stretching) and begin with one second of silence, followed by two seconds of waveform, then one second of silence, for a total length of four seconds. The MATLAB script that generates the waveform file displays is contained in Appendix B, Section 8.4.1.

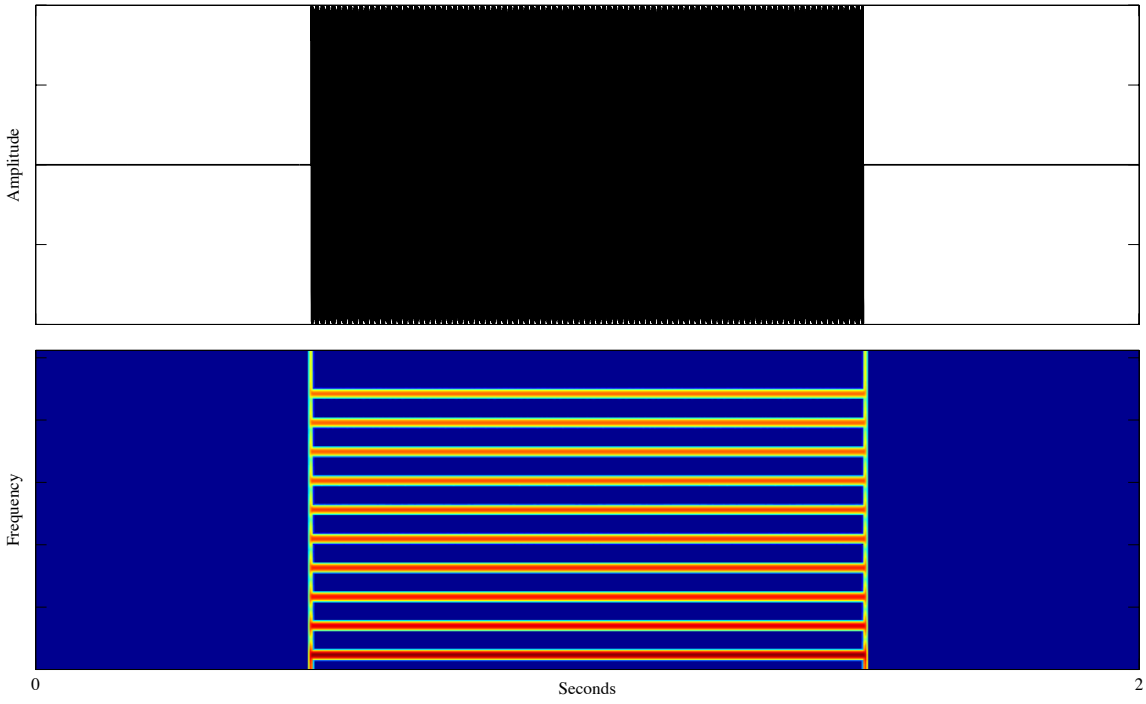
**Graph 2.1.1 –** Sine Wave File



**Graph 2.1.2 –** Sine Wave and Sweep File




**Graph 2.1.3: –** Square Wave File



**2.2 – Recorded Samples**

Three recorded samples were selected based on their particular audio features. Some features are difficult to time stretch while others provide familiar timbres for subjective listening evaluations.

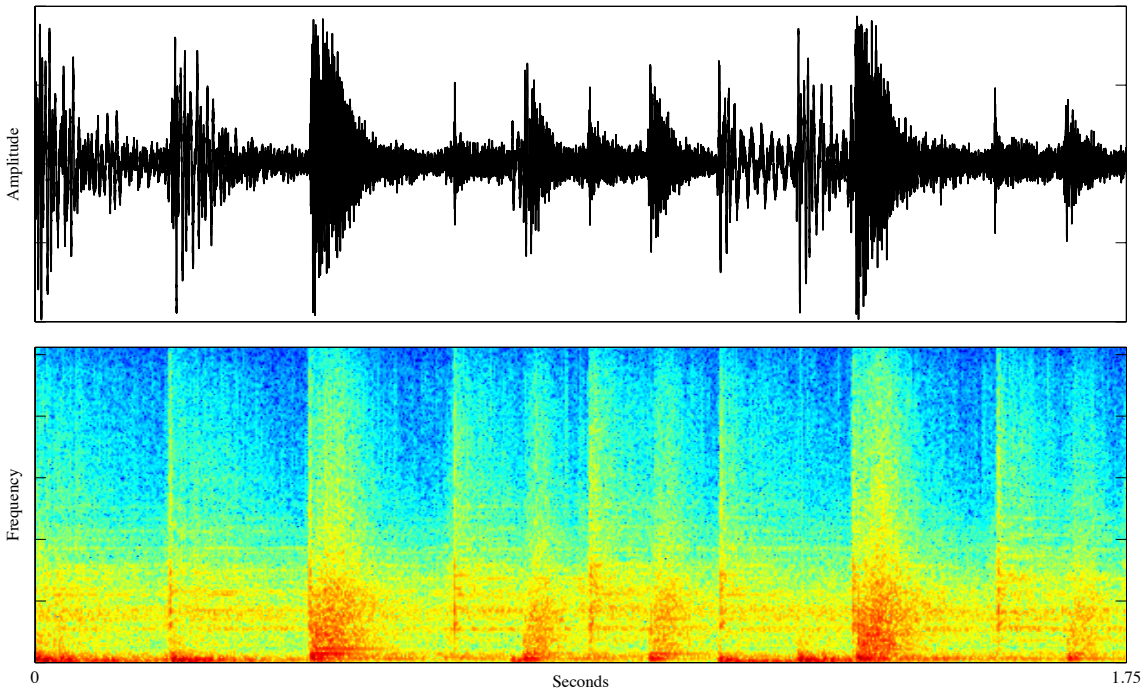
The first sample is a 1.8 second long, single bar of the well known four-bar *Amen Break*, a widely sampled drum kit solo, or break, from the song *Amen Brother*, recorded in 1969 by *The Winstons* [Win69]. This solo drum material contains many percussive transients filled with inharmonic partials, and is difficult to time stretch without introducing artifacts into the resulting audio signals. Inaccurate transients are visible in analysis and are also easy to perceive when listening subjectively to the stretched signals. The sample is displayed in Graph 2.2.1.

The second sample is an excerpt from the song *Autumn in New York*, recorded in 1975 by *The Singers Unlimited*, a four-part vocal jazz group [Sin75]. This 6.5 second sample consists of four vocalists singing in parallel motion, and covers the bass to alto vocal range. It contains vocal timbres with many shifting and overlapping harmonics, as well as sibilant sounds. These features provide rich analysis material, and are also good for subjective listening since the human ear is exceptionally adept at discerning variations in vocal timbres. The sample is displayed in Graph 2.2.2.

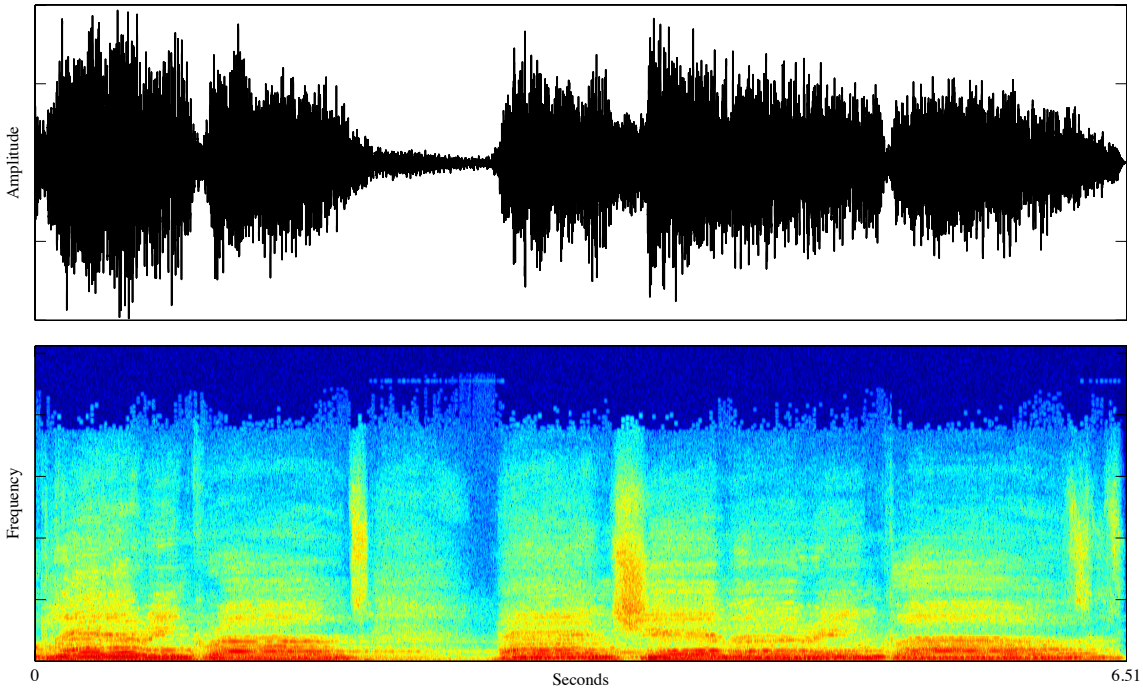
The third sample is a 9.5 second long excerpt from the beginning of the Frank Zappa song *Peaches en Regalia*, from his 1969 album *Hot Rats* [Zap69]. This sample is timbrally dense, containing piano, keyboards, bass, and drums with ringing cymbals. It is full of harmonically related overtones as well as inharmonic partials, transients, and sustained notes of several timbres, while spanning a large portion of the audible

frequency spectrum. These characteristics make it very interesting for both objective analysis and subjective listening. The sample is displayed in Graph 2.2.3, and the MATLAB script that displays these sample files is contained in Appendix B, Section 8.4.1.

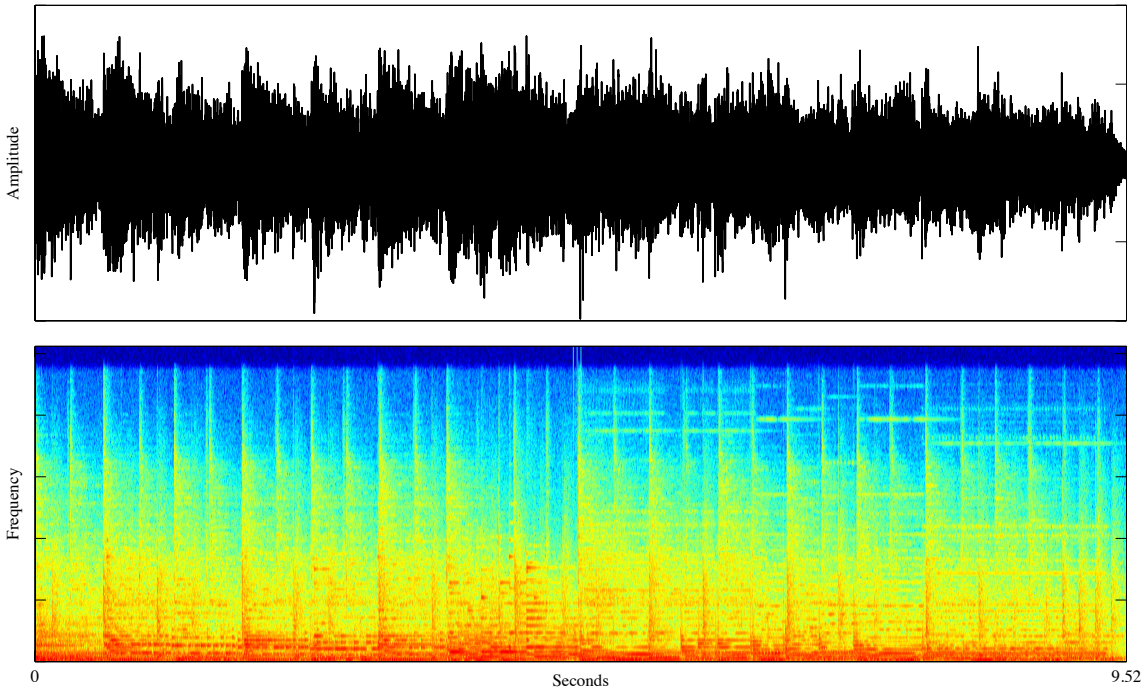
**Graph 2.2.1 –** One Bar of the *Amen Break*



**Graph 2.2.2 –** Excerpt from *Autumn In New York*




**Graph 2.2.3 –** Excerpt from *Peaches en Regalia*



## 3 – Time Stretching Algorithms

Six commonly used time stretching algorithms were selected to perform time stretching operations on the synthetic waveforms and audio files. Five of these algorithms are taken from published works and are intended to represent canonical methods of time stretching, widely used for musical signal processing. The unpublished technique was created by the author and fills a gap in the granular realm where there are few canonical techniques. Four of the algorithms are phase vocoders, and the other two are granular techniques.

The phase vocoders all rely on the Fast Fourier Transform (FFT) to analyze audio signals and generate data for the stretching algorithms. Three of the phase vocoders use the Inverse Fast Fourier Transform (IFFT) to resynthesize audio signals from analysis data after it has been manipulated, and the fourth algorithm relies on an oscillator bank to resynthesize sound. These algorithms are referred to as 1) FFT to IFFT Classic Phase Vocoder, 2) FFT to IFFT Phase-Locked Vocoder, 3) FFT to IFFT Peak Tracking Phase Vocoder, and 4) FFT to Oscillator Bank Phase Vocoder. The two granular algorithms comprise overlap add techniques and are referred to as 1) OLA, for Overlap Add, and 2) SOLA, for Synchronous Overlap Add.

All algorithms are configured to stretch their input by two times, using identical parameters, resulting in output files twice as long as their corresponding input files. Identical parameters provide more meaningful comparisons of artifacts and errors between the different algorithms. The frame size is set to 1024 samples, the overlap factor is set to 4, and each algorithm uses the Hann windowing function. These parameters are chosen as basic, general-purpose settings, and are intended to provide

reasonable results with the diverse input signals chosen for analysis. All output files use a sampling rate of 44100 Hertz and consist of 32 bit floating-point samples. The output of each algorithm is normalized and saved into a file with additional metadata that contains the normalization coefficient for later use.

### 3.1 – Phase Vocoders

The phase vocoders under consideration in this work use the common frame-by-frame technique, sometimes referred to as the block-by-block technique, to generate output signals that are different in length from their input signals. The input frames are the same size as the output frames and use the same windowing function. However, their hop sizes differ, resulting in signals of different durations. In the case of time stretching, the output hop size is longer than the input hop size, essentially taking frames from the input signal and spreading them across more time to create a longer output signal. The Fourier transform provides information about each frame's constituent frequency components. These components' phases may be manipulated to create continuous transitions between successive frames that smoothly fill the gaps without significantly changing frequency content. The signal is analyzed, modified, and resynthesized. However, these modifications may alter phase in uncoordinated ways, contributing to so-called "phasiness" in the stretched signal. Often, computer music practitioners observe that phase vocoders have a characteristic and somewhat undesirable sound associated with these phase errors, and much effort has been spent to minimize them. [Lar97]

Each phase vocoder algorithm chosen for study here implements the frame-by-frame approach differently. Some algorithms pay special attention to phase by using particular techniques that attempt to mitigate phasiness through additional phase

manipulations, while other algorithms simply calculate the expected phases and do nothing to address phasiness.

### 3.1.1 – FFT to IFFT Classic Phase Vocoder

This algorithm is arguably the most commonly implemented version of a phase vocoder in the world of audio signal processing. It is relatively simple, well known, and contains no special phase manipulations designed to decrease phase error resulting from the time stretching process.

Frames are copied from the input buffer and windowed, then their samples are shifted to place the zero frequency component at the center of the spectrum. An FFT is performed on the frame, and the resulting Cartesian data is transformed to polar data. The polar phase values are unwrapped and stretched, then accumulated to create new phase information for resynthesis. This technique maintains the instantaneous frequencies of each bin between input and output frames, and the resulting polar data is transformed back to Cartesian data. The Cartesian data is fed to an IFFT, the zero frequency component is shifted back to the center of the spectrum, and the frame is windowed again. Finally, the output frame is overlapped onto the output buffer to create a stretched version of the input signal. The MATLAB script that implements this algorithm is contained in Appendix B, Section 8.2.1. [Dol86, Fla66, Zöl11]

The Classic Phase Vocoder algorithm may be represented mathematically by the following equations. First, a Discrete Fourier Transform (DFT) creates spectral data in $X$, using overlapped and Hann windowed portions of a signal. The window is represented by $w$, $x$ is the signal, $H$ is the hop size in samples, $S$ is the stretch factor, $N$ is the window size in samples, and $F$ is the number of frames:

$$X[f,k] = \sum_{n=0}^{N-1} w[n]\, x\left[n + f\frac{H}{S}\right] e^{-2\pi i k\left(\frac{n}{N}+\frac{1}{2}\right)} \quad , \quad k = 0, \dots, N-1 \quad , \quad f = 0, \dots, F-1$$

Each frame's signal is shifted to place zero-time at the center of the window by $\left(\frac{n}{N}+\frac{1}{2}\right)$ in the exponent. This is analogus to *frame = fftshift( frame )* in the MATLAB language, where *frame* is the windowed time-domain input to *fft()*. Next, intermediate phase values are calculated as *A*:

$$A[f,k] = \angle X[f,k] - \angle X[f-1,k] - 2\pi\frac{Hk}{SN} \quad , \quad k = 0, \dots, N-1 \quad , \quad f = 0, \dots, F-1$$

Following this, phase deltas are calculated as *B*:

$$B[f,k] = \mathrm{mod}(A[f,k] + \pi, 2\pi) - \pi + 2\pi\frac{Hk}{SN}$$

$$k = 0, \dots, N-1 \quad , \quad f = 0, \dots, F-1$$

After the phase delta calculations, output phases are stretched and accumulated in *C*:

$$C[f,k] = C[f-1,k] + B[f,k]S \quad , \quad k = 0, \dots, N-1 \quad , \quad f = 0, \dots, F-1$$

Once *C* contains the proper output phases, the output spectrum is calculated as *Y*:

$$Y[f,k] = |X[f,k]|e^{iC[f,k]} \quad , \quad k = 0, \dots, N-1 \quad , \quad f = 0, \dots, F-1$$

Finally, the output spectrum is used to synthesize the stretched signal, *y*, using an overlapping windowed DFT:

$$y[f, n+fH] = y[f-1, n+fH] + w[n]\frac{1}{N}\sum_{k=0}^{N-1} Y[f,k]e^{2\pi i k\left(\frac{n}{N}+\frac{1}{2}\right)}$$

$$n = 0, \dots, N-1 \quad , \quad f = 0, \dots F-1$$

### 3.1.2 – FFT to IFFT Phase-Locked Vocoder

The Phase-Locked Vocoder was developed by Miller Puckette and refines the classic method described above by locking the phases of neighboring bins to mitigate phase errors [Puc95]. It is also notable for its mathematical elegance. The algorithm is implemented simply and uses no trigonometry, staying completely in the complex domain and relying on significantly fewer operations than other phase vocoders.

This algorithm uses the same frame-by-frame approach to stretch signals, however, instead of using one frame at a time, it copies two frames from the input buffer, with one frame offset by the analysis hop size. The input frames are windowed and an FFT is performed on each of them, then phase accumulation is calculated with the complex values of the previous frame and the first input frame. Three copies of the accumulated phase are used to create the phase-locked spectrum by shifting two copies in opposite directions spectrally, by one bin, then adding all three together. This encourages adjacent bins to stay in phase by causing them to synchronize with whichever bin has the strongest amplitude. The output spectrum is then calculated using the locked spectrum and offset input spectrum, resulting in a less phasy and more present output signal. [Puc14, Puc07] The MATLAB script that implements this algorithm is contained in Appendix B, Section 8.2.2.

The Phase-Locked Vocoder is shown by the following formulas. First, a DFT is used to generate spectral data, where $X$ contains the spectra, $w$ is the window function, $x$ is the signal, $H$ is the hop size in samples, $S$ is the stretch factor, $N$ is the window size in samples, and $F$ is the number of analysis frames:

$$X[f,k] = \sum_{n=0}^{N-1} w[n]\, x\left[n + f\frac{H}{S}\right] e^{-2\pi i k \frac{n}{N}} \quad , \quad k = 0, \dots, N-1 \quad , \quad f = 0, \dots, F-1$$

The next step is phase accumulation where *P* contains accumulated phase, over-bar signifies the complex conjugate, and *Y[f-1,k]* represents the previous output spectrum:

$$P[f,k] = \frac{\overline{X\left[f\frac{H}{S},k\right]}\, Y[f-1,k]}{|Y[f-1,k]|} \quad , \quad k = 0, \dots, N-1 \quad , \quad f = 0, \dots, F-1$$

Following phase accumulation, phases are locked together into *L*, with the notation *k-1* and *k+1* signifying bin offsets, so that the phases of three adjacent bins are combined:

$$L[f,k] = P[f,k-1] + P[f,k] + P[f,k+1] \quad , \quad k = 0, \dots, N-1 \quad , \quad f = 0, \dots, F-1$$

Once all phases are accumulated, the output spectra are calculated as *Y*:

$$Y[f,k] = \frac{L[f,k]X\left[f\frac{H}{S} + H,k\right]}{|L[f,k]|} \quad , \quad k = 0, \dots, N-1 \quad , \quad f = 0, \dots, F-1$$

Finally, the output signal is windowed and overlap-added into *y*:

$$y[f, n + fH] = y[f-1, n + fH] + w[n]\frac{1}{N}\sum_{k=0}^{N-1} Y[f,k] e^{2\pi i k \frac{n}{N}}$$

$$n = 0, \dots, N-1 \quad , \quad f = 0, \dots F-1$$

### 3.1.3 – FFT to IFFT Peak Tracking Phase Vocoder

This algorithm was developed by Jean Laroche and Mark Dolson as an extension of the traditional phase vocoder, and is considered to be an enhancement of the Phase-Locked Vocoder described in the previous section [Lar99]. Sometimes referred to as a Peak-Tracking Phase-Locked Vocoder, or a Scaled-Phase-Locking Vocoder, this scheme starts with the familiar frame-by-frame approach. Following FFT analysis, the algorithm detects significant spectral peaks and treats them as areas of interest for phase

manipulation. These peaks are connected to the closest peaks of the previous frame, and the previous peak phases are propagated forward into the current peaks, assuming linear frequency motion between frames. The phases of bins surrounding each peak are rotated equally to maintain phase coherence, and then the spectrum is fed to an IFFT that synthesizes the output frame.

The Peak Tracking technique is said to significantly reduce phase dispersion, or phasiness, but assumes that input sounds are comprised primarily of quasi-sinusoidal components, and attempts to propagate their phases accordingly [Zöl11]. The MATLAB script that implements this algorithm is contained in Appendix B, Section 8.2.3.1.

Due to loops and conditional evaluations, this algorithm is difficult to express mathematically in its entirety. The significant portions are described by the following formulas. First, a DFT is used, where $X$ contains the spectral data, $w$ is the window function, $x$ is the signal, $H$ is the hop size in samples, $N$ is the window size in samples, and $F$ is the number of analysis frames. Again, the signal is shifted to place zero-time at the center of the window by $\left(\frac{n}{N}+\frac{1}{2}\right)$ in the exponent, analogus to *frame = fftshift( frame )* in the MATLAB language, where *frame* is the windowed time-domain input to *fft()*.

$$X[f,k] = \sum_{n=0}^{N-1} w[n]\, x[n+fH] e^{-2\pi i k\left(\frac{n}{N}+\frac{1}{2}\right)} \quad , \quad k = 0, \ldots, N-1 \quad , \quad f = 0, \ldots, F-1$$

After the spectral data is created, a set of peaks is chosen for each frame. Peaks are defined as bins whose magnitudes are greater than those of their surrounding four neighbors. These peaks are represented by $K$, and *p[f]* represents the number of peaks per frame:

$$K[f,m] \quad , \quad m = 0, \ldots, p[f]$$

Next, output phase is computed based on the peaks as follows: if *k* matches one of the peaks in *K[f,m]* then set phase in *P[f,k]* according to the Classic Phase Vocoder technique using the previous output spectrum, *Y[f-1,k]*, and the current spectrum to maintain the input phase across frames:

$$P[f,k] = \angle Y[f-1,k] + X[f,k] - \angle X[f-1,k]$$

$$k = 0,\dots,N-1 \quad , \quad f = 0,\dots,F-1$$

Otherwise, let *k'* denote the nearest peak chosen from *K[f,[0,…,p[f]]* and calculate the phase in *P[f,k]* to maintain the input phase between bins according to:

$$P[f,k] = \angle Y[f,k'] + \angle X[f,k] - \angle X[f,k'] \quad , \quad k = 0,\dots,N-1 \quad , \quad f = 0,\dots,F-1$$

After phase is properly massaged, it is used to synthesize the output spectrum in *Y*:

$$Y[f,k] = |X[f,k]|e^{iP[f,k]} \quad , \quad k = 0,\dots,N-1 \quad , \quad f = 0,\dots,F-1$$

Finally, the output signal, *y*, is windowed and overlap-added with stretch factor *S*:

$$y[f,n+fHS] = y[f-1,n+fHS] + w[n]\frac{1}{N}\sum_{k=0}^{N-1} Y[f,k]e^{2\pi ik\left(\frac{n}{N}+\frac{1}{2}\right)}$$

$$n = 0,\dots,N-1 \quad , \quad f = 0,\dots F-1$$

Unlike other phase vocoders, this algorithm implements its frame sliding scheme by scaling hop size during the overlap add stage of its output. Input hop size is always regularly spaced, and output hop size changes based on the stretch factor. Unfortunately, with a stretch factor of two and an overlap factor of four, the output overlap factor is reduced by half, resulting in an output overlap factor of two that spaces frames further apart. When used with the Hann windowing function, overlap add of output frames leads to significant amplitude modulation at the rate of overlap, caused by the Hann function as it sums into a sinusoid that modulates the output signal. Despite this unusual frame

sliding scheme, the Peak Tracking Phase Vocoder as published in [Zöl11] is used in this research. However, it is possible to make the algorithm behave similarly to other phase vocoders by modifying its input and output stages. First, the stretch factor *S* is included in the input stage as shown here:

$$X[f,k] = \sum_{n=0}^{N-1} w[n]\, x\left[n + f\frac{H}{S}\right] e^{-2\pi i k\left(\frac{n}{N}+\frac{1}{2}\right)} \quad , \quad k = 0, \dots, N-1 \quad , \quad f = 0, \dots, F-1$$

Then, the stretch factor is removed from the output stage as follows:

$$y[f, n+fH] = y[f-1, n+fH] + w[n]\frac{1}{N}\sum_{k=0}^{N-1} Y[f,k] e^{2\pi i k\left(\frac{n}{N}+\frac{1}{2}\right)}$$

$$n = 0, \dots, N-1 \quad , \quad f = 0, \dots F-1$$

An additional MATLAB script that implements these changes is contained in Appendix B, Section 8.2.3.2.

### 3.1.4 – FFT to Oscillator Bank Phase Vocoder

Like the Classic Phase Vocoder, this algorithm does nothing extra to manipulate phase. It is quite similar to the classic version, starting with a windowed input frame that is fed to an FFT and converted to polar coordinates. However, phase information is not manipulated to generate new data for IFFT transformation, and is instead used to calculate phase increment values that control an oscillator bank. The oscillator bank synthesizes one sinusoid at the instantaneous frequency of each bin below the Nyquist bin, with the values interpolated linearly between hops. Resynthesis occurs for the duration of each hop, instead of each frame, and these hops are concatenated, without windowing or overlapping, into the output buffer. [Zöl11]

This technique may be conceptually simpler than the Classic Phase Vocoder since phase is not stretched, but instead is simply used to drive oscillators whose phases are allowed to run free. The MATLAB script that implements this algorithm is contained in Appendix B, Section 8.2.4.

The Oscillator Bank Phase Vocoder starts with a DFT where *X* is the spectrum, *w* is the window function, *x* is the signal, *H* is the hop size in samples, *N* is the window size in samples, and *F* is the number of analysis frames:

$$X[f,k] = \sum_{n=0}^{N-1} w[n]\, x[n+fH] e^{-2\pi i k \frac{n}{N}} \quad , \quad k = 0, \dots, N-1 \quad , \quad f = 0, \dots, F-1$$

Next, an intermediate phase wrapping step is calculated as *A*:

$$A[f,k] = \angle X[f,k] - \angle X[f-1,k] - 2\pi H \frac{k}{N} \quad , \quad k = 0, \dots, N-1 \quad , \quad f = 0, \dots, F-1$$

Following phase wrapping, phase delta is calculated as *B*:

$$B[f,k] = mod(A[f,k] + \pi, 2\pi) - \pi + 2\pi H \frac{k}{N}$$

$$k = 0, \dots, N-1 \quad , \quad f = 0, \dots, F-1$$

Once the proper phase deltas are created, magnitude and phase increments are calculated to drive the oscillator bank, and are shown in this notation as frame-sized vectors of propagating magnitude and phase values. The magnitudes are contained in *M*, and calculated using the stretch factor *S*:

$$M[n + fHS, k] = M[n - 1 + fHS, k] + \frac{(|X[f,k]| - |X[f-1,k]|)}{HS}$$

$$n = 0, \dots, N-1 \quad , \quad k = 0, \dots, N-1 \quad , \quad f = 0, \dots, F-1$$

The phases are contained in *P*:

$$P[n + fHS, k] = P[n - 1 + fHS, k] + \frac{B[f, k]}{H}$$

$$n = 0, \dots, N - 1 \quad , \quad k = 0, \dots, N - 1 \quad , \quad f = 0, \dots, F - 1$$

Finally, the output signal is synthesized and concatenated piecewise into *y*:

$$y[n + fHS] = \frac{1}{N} \sum_{k=0}^{N-1} M[f, k] \cos P[f, k] \quad , \quad n = 0, \dots, HS - 1 \quad , \quad f = 0, \dots, F - 1$$

### 3.2 – Granular Techniques

Countless granular techniques have been devised to time-stretch audio signals, and these techniques are usually idiosyncratic to each person who implements them. However, there are two canonical granular techniques that are documented in the literature. These techniques are referred to as Synchronous Overlap Add (SOLA), and Pitch Synchronous Overlap Add (PSOLA). The PSOLA technique is intended for use with pitched signals, specifically the monophonic human voice, and has been omitted here because it is not a general-purpose time stretching algorithm [Zöl11]. The SOLA technique was also originally intended to operate on human speech, but does not assume the signal it stretches is pitched, and consequently works as a general-purpose time stretching algorithm. The SOLA and PSOLA methods are part of a larger class of granular algorithms that may be referred to as Overlap Add (OLA) techniques. Despite the lack of a canonical OLA technique in the literature, a basic OLA technique has been implemented in this research, which shares fundamental traits with many granular time stretching techniques.

Like phase vocoder techniques that use the frame-by-frame approach, these granular techniques take slices, or grains, of the input signal and create copies of them spaced further apart to create a longer output signal. A windowing function is applied to each grain to attenuate discontinuities at the grain boundaries. However, unlike phase vocoder techniques, the granular techniques do not employ Fourier analysis and are unable to manipulate phase. This granular approach may be thought of as a more brute-force time-domain technique that stretches signals without significantly considering their content.

### 3.2.1 – Overlap Add

This algorithm was created by the author to provide basic insight into the family of granular OLA time stretching techniques. It is as simple as possible, stretching signals by slicing the input into many identically sized overlapping grains, then spreading them apart with less overlap to create a longer output signal. The grains are windowed to prevent amplitude artifacts. However, since this can cause robotization due to the regularly repeating window function as it modulates amplitude, the input grain locations are randomized within their hop boundaries to prevent this problem. The resulting output grains are overlap added at a regularly repeating interval into an output buffer to create the stretched signal. The MATLAB script that implements this algorithm is contained in Appendix B, Section 8.3.1.

A mathematical representation of this algorithm is shown by the following equation, where $y$ is the output signal, $H$ is the hop size in samples, $S$ is the stretch factor, $w$ is the window function, $x$ is the input signal, $r$ is a vector of random values between 0 and 1, $N$ is the grain size, and $F$ is the total number of frames:

$$y[f, n + fH] = y[f - 1, n + fH] + w[n]x\left[n + f\frac{H}{S} + r[f]\frac{H}{S}\right]$$

$$n = 0, \ldots, N - 1 \quad , \quad f = 0, \ldots, F - 1$$

**3.2.2 – Synchronous Overlap Add**

The SOLA technique works similarly to the OLA technique, but takes special care to reduce artifacts by subtly time shifting each grain's crossfade point, and by manipulating overlapped grain amplitudes with special crossfades. The input signal is segmented into overlapping grains and these grains are spaced apart to create a longer output signal, one grain at a time. However, before the grains are overlapped to construct the output signal, the current grain is cross-correlated with the previous grain to determine a location of maximum similarity where the two grains overlap. Grain crossfade slopes are generated based on this location of maximum similarity, resulting in a pair of linear ramps that place the point of maximum similarity at the midpoint of each fade. The fade-in is applied to the head of the current grain, the fade-out is applied to the tail of the previous grain, and these overlapping sections are added together. The crossfaded section is appended to the end of the output buffer, followed by the rest of the current grain [Zöl11, Rou85, Mak86]. The MATLAB script that implements this algorithm is contained in Appendix B, Section 8.3.2.

This algorithm is expressed mathematically as follows. First, cross-correlation values are calculated into *R*, where *x* represents the input signal, *H* is hop size in samples, *y* is the output signal, *S* is the stretch factor, and *F* is the number of frames:

$$R[f,m] = \sum_{n=0}^{\min(m,HS-m)} x[n+fH]y\left[f-1, n+fHS-\left(\frac{HS}{2}-1\right)+m\right]$$

$$m = 0,\dots,HS-1 \quad , \quad f = 0,\dots,F-1$$

Next, the location of the maximum value in *R* is assigned to *K*:

$$K[f] = \text{argmax}\{R[f,0],\dots,R[f,HS-1]\}$$

Following this, the crossfade length $L_c$ is determined where $L_o[f\text{-}1]$ represents the previous frame's output length:

$$L_c[f] = L_o[f-1] - fHS - \frac{HS}{2} + \text{K}[f] - 1 \quad , \quad f = 0,\dots,F-1$$

Next, the current frame's output length is calculated as $L_o[f]$ with *N* representing the frame size in samples:

$$L_o[f] = L_o[f-1] - L_c[f] + N \quad , \quad f = 0,\dots F-1$$

After $L_o[f]$ is calculated, the crossfaded portion of the signal is calculated as *D*:

$$D[f,n] = y\big[f-1, n+L_o[f-1]-L_c[f]\big]\left(1-\frac{n}{L_c[f]}\right) + x[n+fH]\left(\frac{n}{L_c[f]}\right)$$

$$n = 0,\dots,L_c[f]-1 \quad , \quad f = 0,\dots,F-1$$

Following this, the location of the beginning of the crossfade is calculated as *C*:

$$C[f] = fHS - \frac{HS}{2} + K[f] \quad , \quad f = 0,\dots,F-1$$

Finally, the output signal is assembled piecewise from previously calculated segments:

$$y[f,n] = \begin{cases} y[f-1,n] & , \quad n = 0,\dots,C[f-1] \\ D\big[f, n-C[f]\big] & , \quad n = C[f],\dots,L_o[f-1] \\ x[n+fH] & , \quad n = L_o[f-1]+1,\dots,L_o[f-1]-L_c[f]+N-1 \end{cases}$$

$$f = 0,\dots,F-1$$

## 4 – Analysis Techniques

The goal of these analysis techniques is to create an objective method to compare differences between the outputs of time stretching algorithms. In order to achieve this, it is necessary to know what the ideal output of an algorithm should look like. If a known signal goes into an algorithm and comes out stretched to twice its original length, it is not necessary to know what happens inside the algorithm or how accurately it is working internally. Instead, the ideal result of the algorithm may be predicted and used to evaluate the accuracy of the algorithm's actual output. In other words, the input signal should result in an output signal that is identical in all aspects except for its duration. For example, a sine wave stretched to twice its length is simply another, longer, sine wave.

This idea leads to the creation of special synthetic input and ideal output signals, which may be used to analyze the error of time stretching algorithms by comparing their output signals with the ideal output signals. These signals are discussed in detail in chapter two, and share several important characteristics: they contain the same waveforms with the same frequencies, initial phases, and amplitudes, differing only in length. The longer signals represent the ideal output versions of the input signals, and are compared with the actual output signals of the algorithms. These synthetic waveforms are used to provide a very controlled picture of how the outputs of various time stretching algorithms differ from their expected ideal outputs, as well as how their outputs differ from each other.

Insight into the behavior of these algorithms with musical material is also important, and may be considered by using a similar technique. However, real-world signals are not perfect sinusoids with controlled amplitudes and phases, and it is

unfortunately impossible to synthesize an ideal version of a stretched musical sample. Since the ideal output of a time stretching algorithm is an identical, longer copy of its input, any musical material being stretched should result in a signal containing the same spectral content with longer envelopes. This means the resulting spectra may be predicted and compared against the original spectra. By using the original spectra this way, it is possible to compare the results of different algorithms against it and look at how the comparisons differ from each other.

This concept of comparing actual output with ideal output is used to create three analysis techniques, referred to as *Average Spectrum*, *Moving Spectral Average*, and *Error Spectrogram*. These techniques are named for what they do, and are described in Section 4.2.

### 4.1 – Settings

Parameters of the time stretching algorithms and analysis techniques are configured identically in order to create output signals and subsequent analysis data whose differences arise solely from the differences in the algorithms themselves. Any unwanted artifacts which might arise during spectral analysis of the output signals are cancelled out when the stretched output analysis data is subtracted from the ideal output analysis data. These parameters are discussed in the following three sections.

#### 4.1.1 – Time Stretching Parameters

All similar parameters between time stretching algorithms are set to the same values. This is done to prevent different configurations of DFT or grain parameters from unintentionally affecting comparisons made between the various output signals of the algorithms. Fewer variables need to be accounted for if parameters are identical, and

comparisons may focus solely on the stretching algorithms. Furthermore, the values of these parameters are selected from commonly used configurations to mimic what might be encountered in real-world audio software, and to provide a reasonable trade-off between time and frequency resolution.

Each algorithm is configured to stretch signals by two times, resulting in output signals that are twice as long as the input files. The algorithms all use the same windowing function, specifically the Hann window, a general purpose window for phase vocoders and granular techniques, widely available in signal processing software. Finally, the FFT and granular algorithms use a frame and grain size of 1024 samples with an overlap factor of four.

**4.1.2 – Analysis Parameters**

All analysis algorithms rely on Fourier analysis and use FFTs configured with the same settings so that any artifacts will cancel each other out during comparison. Overlap factor is set to four for all analysis algorithms, and they all use the same windowing function. The window size is set to 512 samples to provide good time resolution, and to generate graphs that are easier to read. At window sizes larger than 512 samples, graphs tend to increasingly reflect the spectral shape of the windowing function and its sidelobes, making the graphs less comprehensible as frequency resolution increases and more of the window shape creeps into the graphs. The windowing function chosen for analysis is the Chebyshev window, since it provides virtually flat sidelobes with configurable attenuation. In this case, sidelobe attenuation is set at -125 decibels. Among windowing functions available in MATLAB with adjustable sidelobe attenuation, the Chebyshev window provides the best main lobe width versus sidelobe attenuation,

and imparts less of its shape onto the spectra it creates. When used in conjunction with a frame size of 512 samples and four times overlap, the Chebyshev window provides good results.

An approximation of the sones scale is used to represent the magnitude of analysis data, in order to create visual representations of data that more closely match the human logarithmic perception of loudness. Additionally, Logarithmic scaling shows subtle differences in the graphs of analysis data much more clearly than raw magnitude or amplitude data. The decibel scale may also be used to accentuate these subtle differences, but its range extends to negative infinity and consequently does not provide a baseline of zero magnitude. Such a baseline is necessary for the analysis techniques this research uses to compare different algorithms, and the sones scale provides a happy medium with accentuated subtleties and a fixed baseline for comparison.

**4.1.3 – Normalization**

Due to variability in the output amplitudes of time stretching algorithms, special care is taken to normalize signals before they are analyzed. If the amplitudes are not adjusted, their overall differences introduce an amplitude offset into the comparison of the stretched and ideal signals. Two different normalization strategies are employed to best accommodate the types of signals being analyzed.

The first strategy is designed to accommodate the synthetic signals and takes into account the location of their steady-state portions. Transients generate amplitude spikes when they are processed through time stretching algorithms, and each of the synthesized input signals include transitions from silence to signal, and signal to silence. At these locations there are amplitude spikes in the stretched signals, and the sizes of these spikes

vary depending on the algorithms that created them. In order to create an objective comparison of the various transients, as well as a comparison of the steady-state portions of these signals, only the steady-state portions are used to calculate a normalization coefficient for the stretched file. This coefficient is used to scale the entire file so that the stretched steady-state portion matches the ideal steady-state portion as closely as possible in terms of Root Mean Square power (RMS).

This operation occurs before Fourier analysis, and consists of the following five steps: 1) Get the original normalization coefficient from the stretched file, saved as metadata when it was created. 2) De-normalize the stretched file, scaling it back to its original amplitude as if it came directly out of the time stretching algorithm. 3) Copy and Hann-window the portion of the stretched file that contains the steady-state signal (the middle half, i.e. ¼ to ¾ of the way through), and the portion of the ideal file that contains the steady-state signal. 4) Calculate the RMS value of each windowed copy, then create a scaling coefficient for the stretched file by calculating the ratio of the stretched value to the ideal value. 5) Scale the de-normalized stretched file by this ratio so that the stretched and ideal files match in terms of steady-state RMS power. The MATLAB implementation of this process occurs in several scripts, under sections marked "Normalization", contained in Appendix B, Sections 8.4.3, 8.4.5, and 8.4.7.

The second normalization strategy is designed for real-world signals and does not account for any particular portions of the files, since their content is not controlled. The stretched files are normalized based on their overall amplitudes as described in the following five steps: 1) Get the original normalization coefficient from the stretched file, saved as metadata when it was created. 2) De-normalize the stretched file, scaling it back

to its original amplitude as if it came directly out of the time stretching algorithm. 3) Hann-window the entire stretched file, and the entire original file. 4) Calculate the RMS value of each windowed signal, then create a scaling coefficient for the stretched file by calculating the ratio of the stretched value to the ideal value. 5) Scale the de-normalized stretched file by this ratio so that the stretched and ideal files match in terms of RMS power. The MATLAB implementation of this process occurs in several scripts, under sections marked "Normalization", contained in Appendix B, Sections 8.4.2, 8.4.4, and 8.4.6.

The RMS calculation for both normalization strategies may be expressed mathematically as shown in the following equation, where $x$ is the de-normalized audio file, $w$ is the windowing function, $A$ is the start of the signal region, $B$ is the end of the signal region, and $R$ is the RMS power value:

$$R = \sqrt{\frac{1}{B-A}\sum_{n=0}^{B-A-1}(w[n]x[n+A])^2}$$

$R$ is then used to perform power-normalization of the stretched file so that it matches the ideal file as shown by the following equation, where $x_s$ is the stretched file, $R_s$ is the stretched RMS power value, $R_i$ is the ideal RMS power value, $x_{sr}$ is the power-normalized stretched file, and $N$ is the length of the file in samples:

$$x_{sr} = \frac{x_s[n]}{R_s/R_i} \quad , \quad n = 0, \ldots, N-1$$

### 4.2 – Techniques

For this research, three significant analysis techniques were developed to analyze and compare time stretching algorithms. Each of these techniques operates by comparing a time-stretched signal with an ideal signal, and generates a measure of error between the two. Due to the arbitrary and unpredictable nature of phase in the output of time stretching algorithms, each analysis technique described here uses only magnitude information to generate data. The specific details of each technique are described in the next three sections.

#### 4.2.1 – Average Spectrum

The Average Spectrum technique works by performing a windowed, overlapped, discrete Fourier analysis on the stretched and ideal signals. Depending on the type of signal, this technique looks at different parts of the stretched signals. For synthesized signals (i.e. sinusoid, square, and sinusoidal sweep), the middle half of the stretched signal is analyzed, and for real-world musical signals, the entire stretched signal is analyzed.

To create the Average Spectrum of a signal, analysis data is converted to polar form and each successive magnitude spectrum is stored. Then, the magnitude spectra are summed together, and the resulting spectrum is divided by the number of analysis frames. This average magnitude spectrum is then converted to a sones spectrum to create data that provides a more meaningful visual representation of error. This process occurs for both signals being compared, resulting in an average sones spectrum of the stretched signal and an average sones spectrum of the ideal signal.

After both of the average spectra are created, an error spectrum is calculated by subtracting the stretched spectrum from the ideal spectrum, then by taking the absolute value of the result. This provides an absolute error spectrum that represents how much the ideal and stretched spectra differ from each other. In addition to the error spectrum, a single measure of error is created by averaging all error spectrum values together, and the maximum error value is also recorded. The MATLAB scripts that implement these techniques are contained in Appendix B, Sections 8.4.2 and 8.4.3.

The Average Spectrum technique may be expressed mathematically according to the following equations. First, a discrete Fourier transform is performed on each signal as shown by the following equation, where *X* is the spectrum, *w* is the window function, *x* is the signal, *H* is the hop size in samples, *N* is the window size in samples, and *F* is the total number of analysis frames:

$$X[f,k] = \sum_{n=0}^{N-1} w[n]\, x[n + fH] e^{-2\pi i k \frac{n}{N}} \quad , \quad k = 0, \ldots, N-1 \quad , \quad f = 0, \ldots, F-1$$

Next, all of the resulting magnitude spectra are averaged together by the following equation, where *A* is a spectrum representing the average of all magnitude spectra:

$$A[X,k] = \frac{1}{F} \sum_{f=0}^{F-1} |X[f,k]| \quad , \quad k = 0, \ldots, N-1$$

Following this averaging calculation, the windowing function is normalized out of the Average Spectrum, and the spectrum is converted to the sones scale according to the following equation, where *S* contains the sones spectrum:

$$S[X,k] = \left( \frac{A[X,k]}{\sum_{n=0}^{N-1} w[n]} \right)^{0.6} \quad , \quad k = 0, \ldots, N-1$$

Next, an error spectrum is calculated by subtracting the stretched sones spectrum from the ideal sones spectrum, where $E$ is the error spectrum, $X_i$ is the ideal spectrum, and $X_s$ is the stretched spectrum:

$$E[k] = |S[X_i, k] - S[X_s, k]| \quad , \quad k = 0, \dots, N-1$$

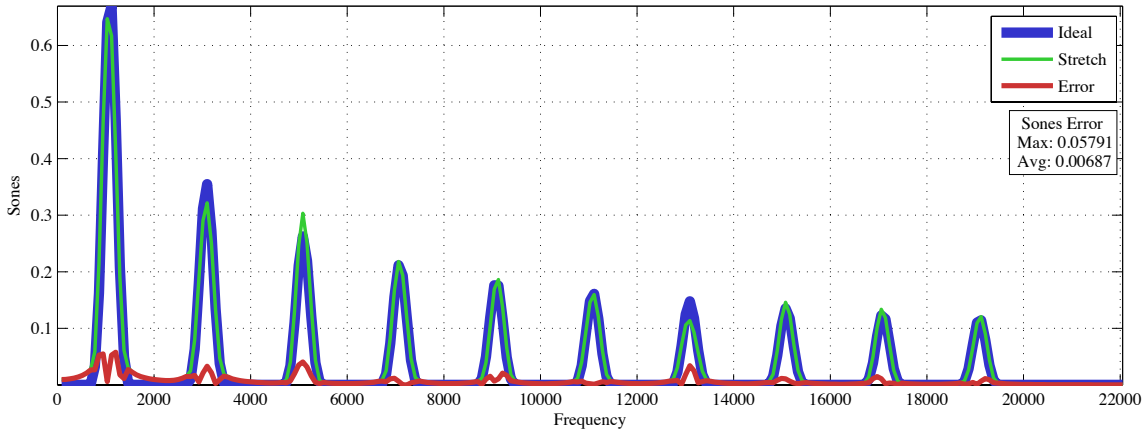
Finally, the average and maximum error values are determined according to the following equations, where $E_{avg}$ is the average value of the error spectrum, and $E_{max}$ is the maximum value of the error spectrum:

$$E_{avg} = \frac{1}{N} \sum_{k=0}^{N-1} E[k]$$

$$E_{max} = \max\{E[0], \dots, E[N-1]\}$$

Graph 4.2.1 shows the Average Spectrum analysis of a stretched square wave. The thick blue line represents the ideal spectrum, the thin green line represents the stretched spectrum, the medium red line represents the error, and average and maximum values are listed in the lower legend. In practice, the stretched spectrum tends to closely approximate the ideal spectrum, and its accuracy reflects the accuracy of the time stretching algorithm that generated it. A perfect time-stretch would result in a stretched spectrum that exactly matches its ideal spectrum, resulting in a flat error graph. In the example graph, the square wave's harmonics are resynthesized with varying degrees of accuracy, and the amount of error the stretched signal exhibits is represented by the red error plot. All Average Spectrum graphs generated by this technique are contained in Appendix A, Section 7.1.

**Graph 4.2.1.1 –** Average Spectrum Analysis of a Stretched Square Wave




### 4.2.2 – Moving Spectral Average

The Moving Spectral Average technique is similar to the Average Spectrum described in the previous section. However, instead of comparing the spectra of entire files, it compares the spectra of files over time, as they progress frame by frame.

This technique performs a windowed, overlapped, discrete Fourier analysis on two files. During analysis of each frame, the spectral data is converted to polar form and the magnitude spectra are saved. These magnitude spectra are used to calculate an average magnitude value for each frame by summing all their values and dividing by the size of the spectrum. As the Fourier analysis moves through the files, each average magnitude value is appended onto an array, resulting in a curve that shows how the average magnitude values change over time. After Fourier analysis is completed, the arrays of average magnitude values are converted to sones, resulting in a graph of the moving average sones value for the ideal signal, and a graph of the moving average sones value for the stretched signal. Next, the stretched values are subtracted from the ideal values and the absolute values of the result are stored. These absolute values represent the error over time between the two signals by showing how the stretched signal differs

from the ideal signal. The average value of the error curve is calculated, the maximum value of the error curve is found, and these numbers are stored for later comparison.

Since it is not possible to create ideal stretched versions of real-world musical signals, the analysis of these signals is performed by comparing them to their original input signals. This is achieved by performing the operations described in the previous paragraph and adding an extra step: after Fourier analysis moves through all frames, the magnitude values of the stretched signal are compressed to match the length of the ideal signal. Pairs of magnitude values are averaged together, resulting in an average magnitude curve for the stretched signal that has been compressed to the same length as the input file curve. The MATLAB implementations of these techniques are contained in Appendix B, Sections 8.4.4 and 8.4.5.

The Moving Spectral Average techniques may be expressed more concisely by the following equations. The first step requires Fourier analysis and is implemented with a DFT. *X* represents the output spectrum, *N* is the number of samples per analysis frame, *w* is the window function, *x* is the signal, *H* is the hop size in samples, and *F* is the total number of frames:

$$X[f,k] = \sum_{n=0}^{N-1} w[n]\, x[n + fH] e^{-2\pi i k \frac{n}{N}} \quad , \quad k = 0, \dots, N-1 \quad , \quad f = 0, \dots, F-1$$

The next step is the calculation of a set of moving average magnitude values over time, or magnitude curve, according to the following equation where *M* contains the magnitude curve:

$$M[X,f] = \frac{1}{N} \sum_{k=0}^{N-1} |X[f,k]| \quad , \quad f = 0, \dots, F-1$$

When analyzing real-world signals with no ideally synthesized counterparts, an extra step is required, which creates a compressed magnitude curve matching the magnitude curve of the input signal. This is accomplished by averaging adjacent pairs of magnitude values, as shown by the next equation, where $M$ contains the magnitude curve, $X_s$ is the raw stretched spectrum, and $X_c$ is the compressed spectrum:

$$M[X_c, f] = \frac{M[X_s, 2f] + M[X_s, 2f+1]}{2} \quad , \quad f = 0, \dots, F-1$$

The windowing function is then normalized out of the magnitude curve, and the magnitude curve is converted to sones as represented by $S$, shown here by the following equation:

$$S[X, f] = \left( \frac{M[X, f]}{\sum_{n=0}^{N-1} w[n]} \right)^{0.6} \quad , \quad f = 0, \dots, F-1$$

After creating the sones curves in $S$, this data is used to calculate an error curve to show how the two signals differ. For synthetic signals with ideal counterparts, the following formula shows this process where $E$ represents the error curve, $X_i$ is the ideal spectrum, and $X_s$ is the stretched spectrum:

$$E[f] = |S[X_i, f] - S[X_s, f]| \quad , \quad f = 0, \dots, F-1$$

For real-world signals with no ideal counterparts, the error curve, $E$, is calculated according to the following formula, with $X_o$ representing the original spectrum, and $X_c$ representing the compressed spectrum:

$$E[f] = |S[X_o, f] - S[X_c, f]| \quad , \quad f = 0, \dots, F-1$$

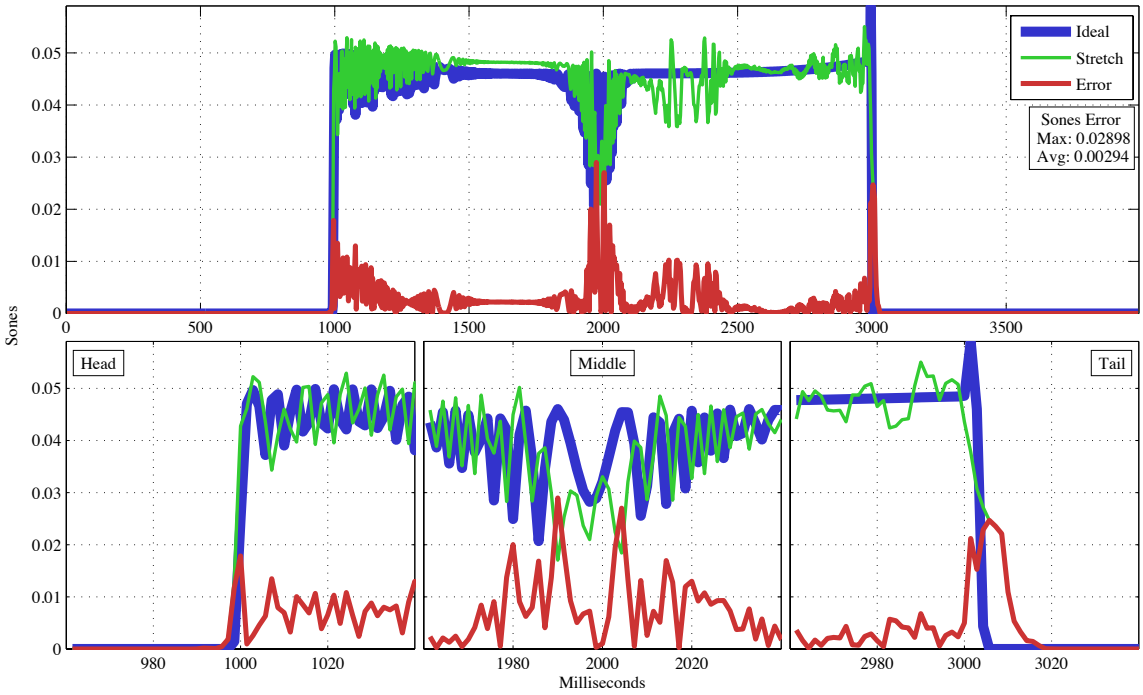
Finally, the average and maximum error values are determined according to the following equations, where $E_{avg}$ is the average value of the error spectrum, and $E_{max}$ is the maximum value of the error spectrum:

$$E_{avg} = \frac{1}{F}\sum_{f=0}^{F-1} E[f]$$

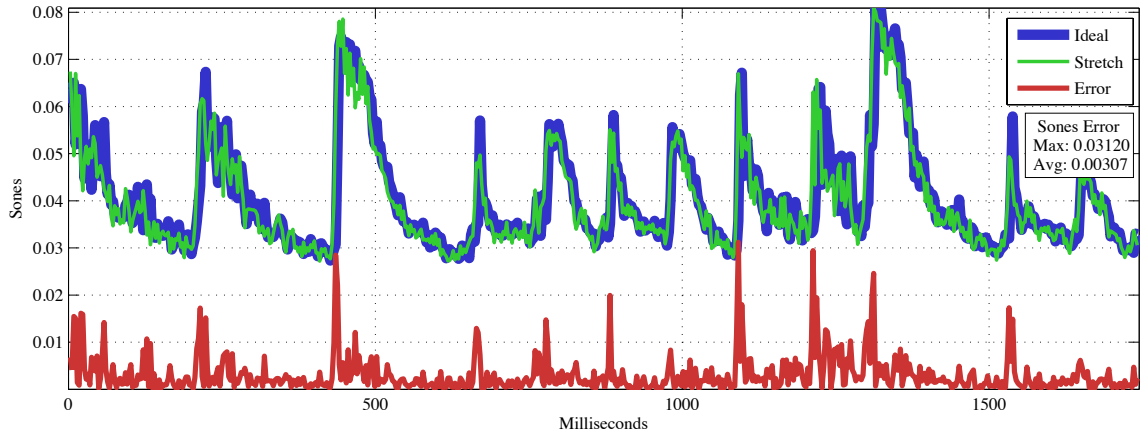
$$E_{max} = \max\{E[0], \ldots, E[N-1]\}$$

Unlike the Average Spectrum technique, this technique operates identically on the synthetic waveforms and the real-world musical samples by comparing their entire files. Since the synthetic files contain transitions from silence to signal, and signal to silence, it is interesting to observe how the various time stretching algorithms handle the regions that lie at the edges of their steady-state portions. In order to inspect these areas, the MATLAB script that displays their graphs was designed to provide an overview of the entire moving spectral analysis, as well as detailed plots of the head, middle, and tail sections of the steady-state signals. Graph 4.2.2.1 shows an example of the stretched synthetic sinusoidal sweep, and Graph 4.2.2.2 shows an example of a stretched real-world musical sample. The thick blue line represents the ideal signal, the thin green line represents the stretched signal, the red line represents the error, and average and maximum values are recorded in the lower legend. All graphs generated by the Moving Spectral Average technique are contained in Appendix A, Section 7.2.

**Graph 4.2.2.1 –** Moving Spectral Average of a Synthetic Swept Sinusoid




**Graph 4.2.2.2 –** Moving Spectral Average of a Real-World Musical Signal



### 4.2.3 – Error Spectrogram

The Error Spectrogram technique generates a spectrogram to display the difference between two files. Unlike the Average Spectrum and Moving Spectral Average techniques in the previous sections, this technique does not calculate any averages of spectral data. Instead, it simply calculates the difference between two sets of spectra and provides a visual overview of how the signals differ across time. This results in a spectrogram that represents how the stretched signals differ from their ideal counterparts.

An Error Spectrogram is created by first performing a windowed, overlapped discrete Fourier transform on the normalized ideal and stretched signals, to generate sets of magnitude spectra representing the signals' contents over time. Since it is impossible to generate ideal synthetic versions of real-world signals, stretched real-world spectra must be compressed and compared with their original counterparts. The Error Spectrogram technique employs a method similar to the Moving Spectral Average to achieve this compression. Adjacent pairs of spectra are averaged together to create a compressed set of spectra matching the size of the original signal's set of spectra.

At this point, all magnitude spectra are converted to sones spectra, then the stretched sones spectra are subtracted from the ideal sones spectra and the absolute values of the results are stored as the error spectra. These error spectra are plotted as a spectrogram that displays the absolute difference, or error, between the two input signals over time. The MATLAB scripts that implement this technique are contained in Appendix B, Sections 8.4.6 and 8.4.7.

The Error Spectrogram technique may be expressed mathematically by the following equations. First, Fourier analysis occurs with a DFT where $X$ contains the spectra, $w$ contains the window function, $x$ contains the signal, $H$ is the hop size in samples, $N$ is the window size in samples, and $F$ is the number of analysis frames:

$$X[f,k] = \sum_{n=0}^{N-1} w[n]\, x[n+fH] e^{-2\pi i k \frac{n}{N}} \quad , \quad k = 0, \dots, N-1 \quad , \quad f = 0, \dots, F-1$$

Next, the window function is normalized out of the magnitude spectra, and the sones spectra are calculated as a set contained in $S$:

$$S[X,f,k] = \left( \frac{|X[f,k]|}{\sum_{n=0}^{N-1} w[n]} \right)^{0.6} \quad , \quad k = 0, \dots, N-1 \quad , \quad f = 0, \dots, F-1$$

However, in the case of real-world signals, the stretched spectra must be compressed in time to match the length of the original spectra, so adjacent pairs of spectra are averaged before the sones spectra are calculated:

$$S[X_c,f,k] = \left( \frac{|X_s[2f,k] + X_s[2f+1,k]|}{2\sum_{n=0}^{N-1} w[n]} \right)^{0.6} \quad , \quad k = 0, \dots, N-1 \quad , \quad f = 0, \dots, F-1$$

Finally, the Error Spectrogram is calculated as $E$ by subtracting the stretched or compressed spectra, represented by $X_s$ or $X_c$, from the ideal or original spectra, represented by $X_i$ or $X_o$, and taking the absolute value of the result:

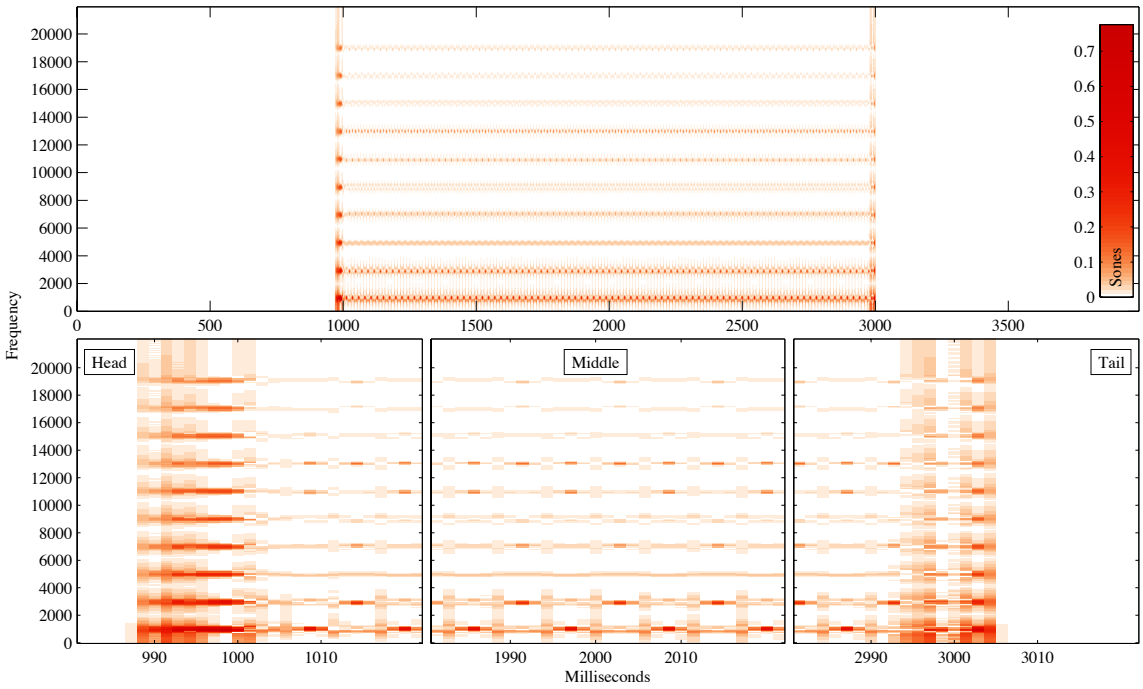
$$E[f,k] = |S[X_i,f,k] - S[X_s,f,k]| \quad , \quad k = 0, \dots, N-1 \quad , \quad f = 0, \dots, F-1$$

$$E[f,k] = |S[X_o,f,k] - S[X_c,f,k]| \quad , \quad k = 0, \dots, N-1 \quad , \quad f = 0, \dots, F-1$$

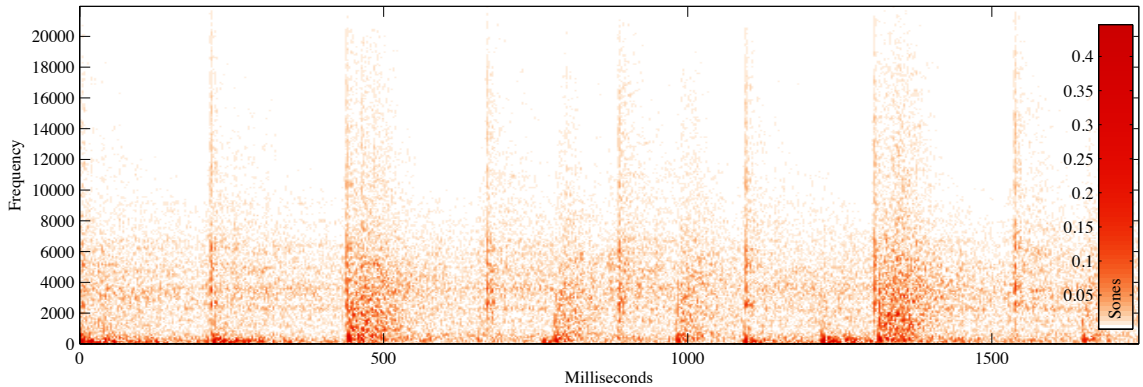
Since this technique analyzes signals over time, it is possible to inspect specific locations of interest within the signals. Similar to the Moving Spectral Average technique, this technique generates overview plots of the entire signals, as well as

detailed plots of the head, middle, and tail portions of the synthetic signals, where interesting transitions occur. Graph 4.2.3.1 shows an example of a stretched synthetic square wave, and Graph 4.2.3.2 shows an example of a stretched real-world musical signal. All graphs generated by the Error Spectrogram technique are contained in Appendix A, Section 7.3.

**Graph 4.2.3.1 –** Error Spectrogram of a Synthetic Square Wave




**Graph 4.2.3.2 –** Error Spectrogram of a Real-World Musical Signal



### 4.2.4 – Subjective Listening

In addition to mathematical analysis of the time stretched signals, subjective listening observations of these signals are included to provide a perceptual picture of the signals' characteristics and differences. The author is a classically trained musician with sensitive ears that are attuned to the subtleties of computer music. Listening observations of the time stretched material are portrayed using the vocabulary of digital signal processing, in an attempt to describe these observations in a clear and meaningful manner, while also conveying an idea of their musical characteristics.

## 5 – Analysis Results

The following four sections show the results of Average Spectrum, Moving Spectral Average, Error Spectrogram, and subjective listening analysis for all time stretching algorithms and output signals. The graphs display all error curves and are grouped by signal type so that each group of graphs shows every algorithm's results for each signal, with sones identically scaled for each group of graphs to facilitate visual comparison of the results. Within each group, the individual graphs are labeled according to their respective algorithms: "Classic" represents the FFT to IFFT Classic Phase Vocoder, "Lock" represents the FFT to IFFT Phase-Locked Vocoder, "Peak" represents the FFT to IFFT Peak Tracking Phase Vocoder, "Bank" represents the FFT to Oscillator Bank Phase Vocoder, "Sola" represents Synchronous Overlap Add, and finally, "Ola" represents Overlap Add.

### 5.1 – Average Spectrum

The following subsections display graphs of the Average Spectrum analysis results and provide a general discussion of the errors shown in these graphs. As mentioned in Chapter Four, Average Spectrum analysis considers only the steady-state sections of the synthetic signals, but considers the entire length of the real-world signals.

The Average Spectrum technique provides information about a time stretching algorithm's overall accuracy in terms of a signal's frequency spectrum. General characteristics of an algorithm's performance are represented by the Average Spectrum error curve. These characteristics include the accuracy of phase manipulations during the analysis and resynthesis stages of spectral algorithms, as well as the distribution of energy among their bins. For example, many synthetic signal error curves display skirts

of error surrounding a spectral peak, and these skirts represent energy that is spilling into adjacent bins surrounding the peak. Additionally, Average Spectrum error curves illustrate magnitude errors of spectral and granular algorithms, which can arise from phase cancellation due to poorly overlapped grains, as well as inaccurate instantaneous frequency calculations for the bins of spectral frames.

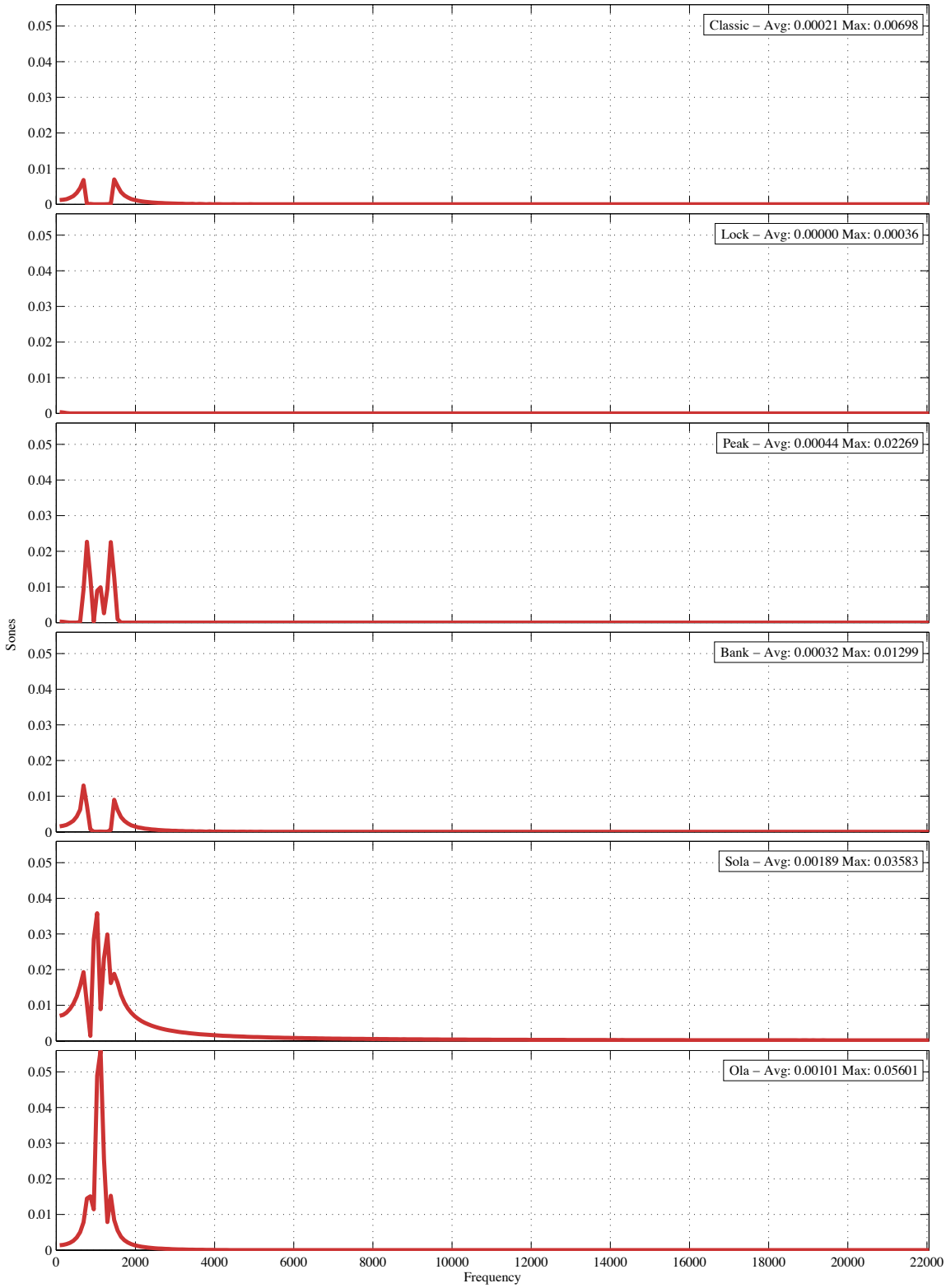
All Average Spectrum analysis graphs are contained in Appendix A, Section 7.1, and the MATLAB script that generated the graphs is contained in Appendix B, Section 8.4.8.

### 5.1.1 – Sine Wave Results

The sine wave error curves in Graph 5.1.1 show that the Phase-Locked Vocoder produces the most accurate time-stretched sine wave. This arises from its treatment of phase as it locks the phases of adjacent bins together, resulting in a purely sinusoidal output waveform at the exact frequency and amplitude of the input waveform. The Classic and Oscillator Bank Phase Vocoders are also relatively error free, but they exhibit skirts of error surrounding the spectral bin containing the input frequency. Since the Classic and Oscillator Bank Phase Vocoders do nothing to address phase, bins adjacent to the sinusoid's actual bin are resynthesized with incorrect phases, and these errors manifest as sinusoidal components with attenuated amplitudes, that are very close in frequency to the input sine wave. Furthermore, oscillator bank resynthesis shows marginally greater error due to data interpolation between frames before resynthesis. The Peak Tracking Phase Vocoder exhibits slightly more error, with visible skirts as well as small error peaks surrounding the center frequency. These errors are likely related to the algorithm's attempts to pick peaks and control the phases of adjacent bins using a signal

that contains only one spectral peak, resulting in a main sinusoid with the correct frequency but inaccurate amplitude, and low amplitude sinusoids surrounding the main sinusoid. Synchronous Overlap Add exhibits significant skirts and frequency errors, likely related to the algorithm's time-domain attempts to find and crossfade discrete locations of maximum similarity in a sinusoidal signal that is homogenous and self-similar throughout. Finally, the Overlap Add algorithm results in skirts as well as a noticeable error spike located at the sine wave's center frequency. This spike arises from phase cancellation and subsequent random amplitude modulation introduced by the algorithm as it attempts to mitigate robotization by randomly shifting input frame locations.

**Graph 5.1.1 –** Sine Wave Average Spectrum Errors



### 5.1.2 – Square Wave Results

Graph 5.1.2 illustrates square wave error curves that are very similar to those in the sine wave curves, but multiplied by the harmonics of the square wave. Again, the Phase-Locked Vocoder produces nearly error-free results since it locks the phases of adjacent bins together, and synthesizes an output spectrum that matches the input spectrum. The Classic and Oscillator Bank Phase Vocoders display error skirts surrounding the square wave's harmonics, and the Oscillator Bank shows slightly larger skirts, arising from its spectral data interpolation. These algorithms also produce mildly larger errors surrounding the fourth and sixth harmonics of the square wave. The reasons for this are unclear, showing little relation to bin frequency center or edge proximity. The Peak Tracking Phase Vocoder is next, with slightly more error, shown by taller skirts and the presence of small peaks at the harmonic frequency centers. Unlike the Classic and Oscillator Bank methods, the Peak Tracking method produces errors that seem to attenuate linearly, according to the amplitudes of the input harmonics. The Overlap Add algorithm is next, with minor error skirts but major peaks at the harmonic frequency centers, again due to random amplitude modulation. Finally, the Synchronous Overlap Add results show significant errors with both large skirts and peaks that arise from the algorithm's attempts to determine crossfade points based on cross-correlation of a homogeneous input signal.

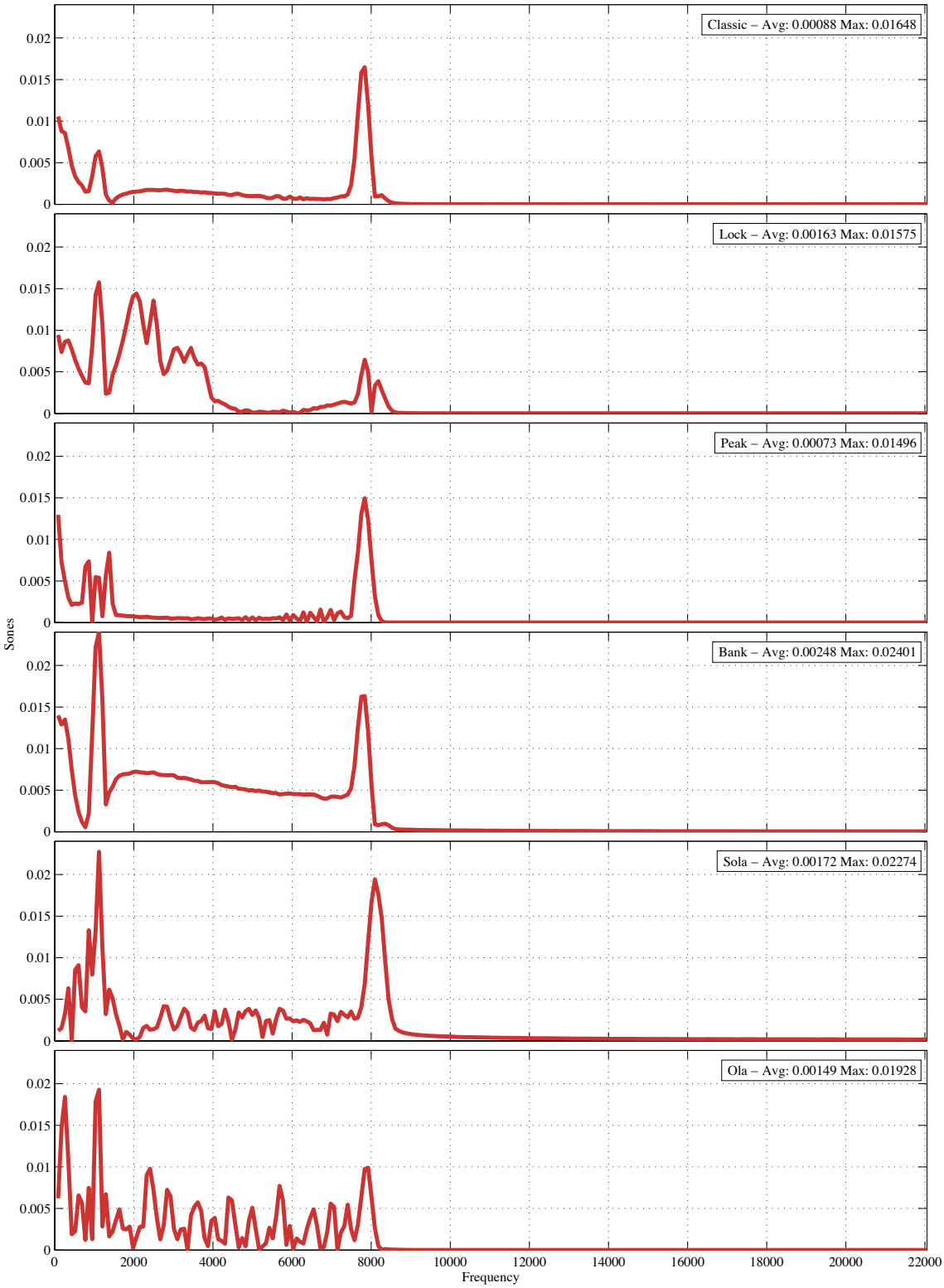
**Graph 5.1.2 –** Square Wave Average Spectrum Errors

### 5.1.3 – Sinusoidal Sweep Results

The Average Spectrum analysis results of the sinusoidal sweep are displayed in Graph 5.1.3. All algorithms display an error spike near 8000 Hz, with smaller spikes shown by the Phase-Locked and Overlap Add algorithms. This spike represents errors that occur once the swept sinusoid begins to accelerate upward in frequency, and it becomes more difficult for the algorithms to analyze and correctly predict (see Graph 5.3.5 for a clearer picture of this frequency movement over time). Additionally, all phase vocoders show low frequency errors arising from the frequency resolution of the DFT as it tries to analyze the beginning of the sweep. These low frequency errors are less present in the granular techniques, since they do not rely on spectral data to operate, and consequently do not suffer from frequency resolution issues. Aside from the approximately 8000 Hz error spike and low frequency errors, this group of graphs shows that the Peak Tracking and Classic Phase Vocoders perform best. As before, the Peak Tracking Phase Vocoder demonstrates a small error spike and skirt surrounding the 1000 Hz tone, then becomes nearly flat as the swept tone progresses upward. The Classic Phase Vocoder is similar, and shows a bit of error surrounding the 1000 Hertz tone, then the error becomes smaller and mostly flat as the swept tone progresses upward. The Oscillator Bank Phase Vocoder looks quite similar to the Classic Phase Vocoder. However, its error seems to be scaled up and is likely larger due to inaccuracies introduced by the magnitude and phase interpolation that occurs during its resynthesis stage. Phase-locking exhibits noticeable errors with this swept signal, as shown in the Phase-Locked Vocoder error curve. Since the frequency content of the spectrum is always changing non-linearly, the phase-locking algorithm attempts to synchronize

phases of neighboring bins, which are also constantly changing. The Synchronous Overlap Add technique displays a significant amount of error at the 1000 Hz tone, then shows a much smaller, albeit noisier, error curve in the swept portion of the spectrum. This arises from the algorithm's purely time based approach, and the introduction of amplitude variations as it attempts to crossfade its grains based on a signal with no self-similarity. Finally, the Overlap Add graph shows a generally greater level of error which may be attributed to its simple time based approach and randomization of grain positions.

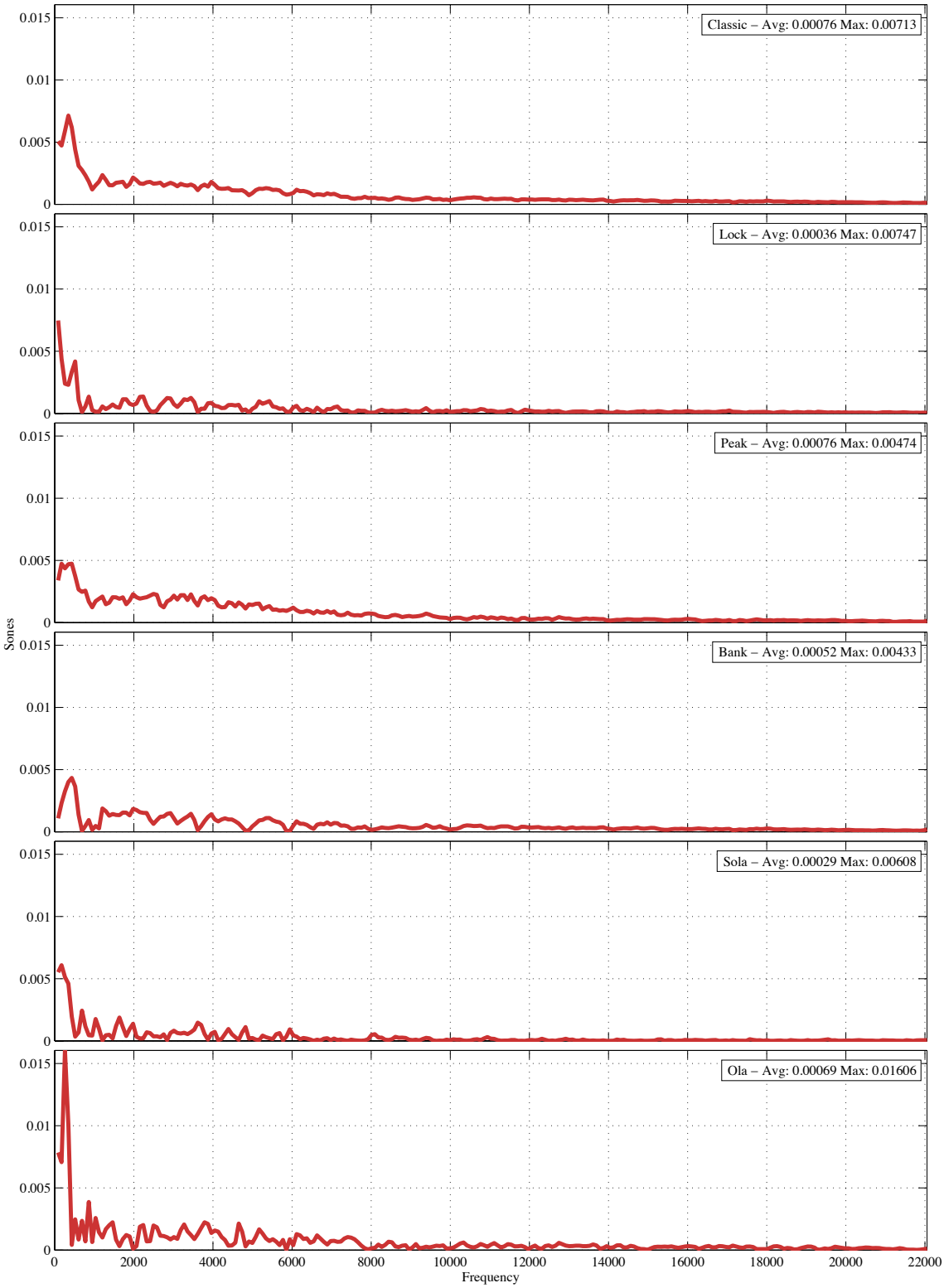
**Graph 5.1.3 –** Sinusoidal Sweep Average Spectrum Errors



### 5.1.4 – *Amen Break* Results

Graph 5.1.4 shows the error curves resulting from analysis of the *Amen Break*. Signals like this contain many unrelated frequencies. Consequently, they do not generate distinct spectral features, resulting in Average Spectrum error curves that look quite similar. This type of source material makes meaningful observations difficult, as the Average Spectrum error curves seem to mimic the overall spectral content of the original signal, exhibiting only minor variations throughout the entire spectrum. Since musical signals contain more low frequency content, and this signal specifically contains high amplitude low frequency content in the form of kick drum strikes, the majority of error occurs in the low frequency end of the spectrum, where the energy is. This may be attributed to the poor low frequency resolution of the DFT as used by phase vocoders, and amplitude variations introduced by the granular algorithms.

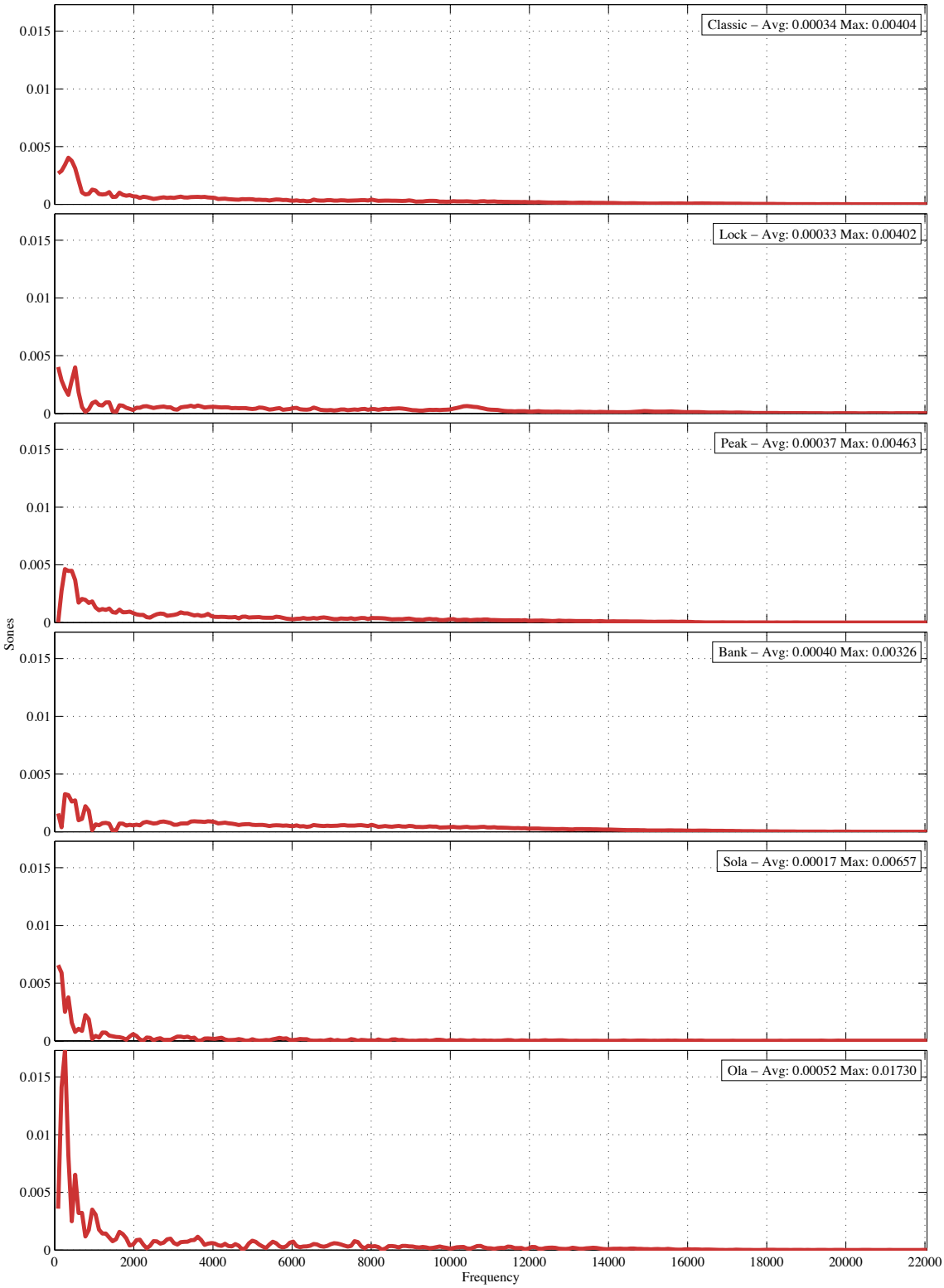
**Graph 5.1.4 –** *Amen Break* Average Spectrum Errors



### 5.1.5 – *Autumn in New York* Results

The error curves shown in Graph 5.1.5 represent the stretched *Autumn in New York* signal, and exhibit an overall contour similar to the original input signal. Again, this musical material contains more low frequency energy, displaying more errors in the low frequency end of the spectrum. Similar to the *Amen Break* error curves, the curves shown in this graph exhibit errors arising from DFT low frequency resolution inaccuracies, as well as granular amplitude errors. The Overlap Add error curve shows a significant low frequency spike, like it did in the previous graph, and this may be attributed to the random amplitude variations introduced by the Overlap Add algorithm.

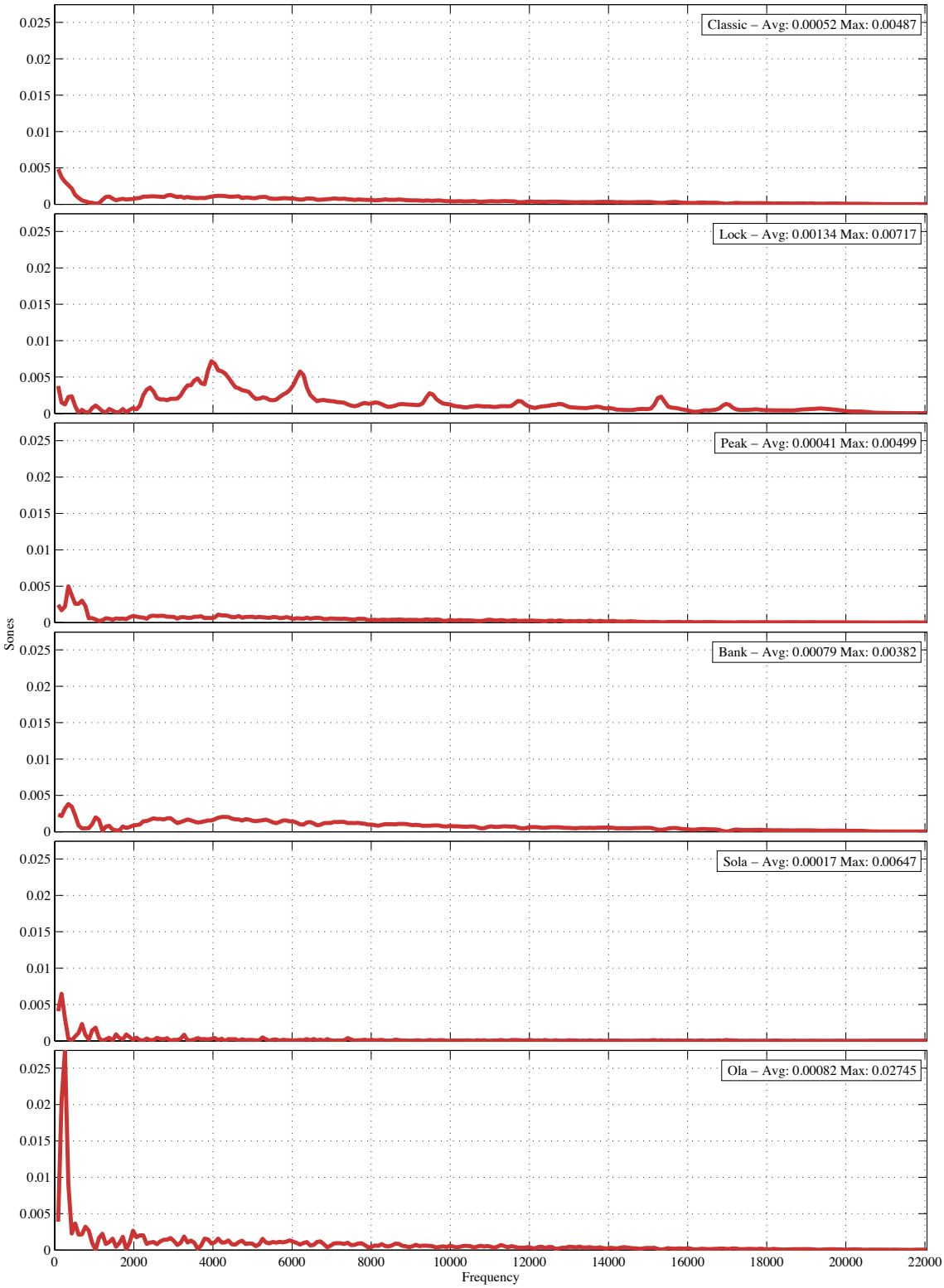
**Graph 5.1.5 –** *Autumn in New York* Average Spectrum Errors



### 5.1.6 – *Peaches en Regalia* Results

Graph 5.1.6 shows the Average Spectrum error curves for the *Peaches en Regalia* signal. Similar to the previous two real-world musical signals, the curves generated for this stretched signal mimic the original average input spectrum, and the Overlap Add curve exhibits a significant low frequency error spike. However, unlike the previous musical signal results, the Phase-Locked Vocoder curve displays notable error peaks throughout its spectrum. Since the *Peaches en Regalia* excerpt contains broader spectral energy in the form of dense instrumentation and unpitched cymbal noise, the Phase-Locked Vocoder is potentially locking phases of adjacent bins that contain unrelated spectral information, and inadvertently introducing phase errors into its output.

**Graph 5.1.6 –** *Peaches en Regalia* Average Spectrum Errors



### 5.2 – Moving Spectral Average

The following six subsections and graphs show Moving Spectral Average error curves for each signal, and provide a general discussion of the errors shown in the graphs. Each signal's error curve is shown as an overview of the entire curve, and the synthetic signals are shown in additional, more detailed views of their head, middle, and tail sections. These additional views focus on areas of interest within the synthetic signals.

The Moving Spectral Average technique generates information about the behavior of a stretched signal over time. Since this technique's error curve represents the average spectral magnitude error of each frame, it provides insight regarding a time stretching algorithm's behavior as it moves across the various parts of a signal. This is useful for inspecting transients or other significant time-domain features in order to evaluate the relative accuracy of each algorithm. In the error curves of the synthetic signals, as well as the *Amen Break*, the errors caused by these features are visibly different between algorithms, and indicate how accurately each algorithm performs when stretching the signals. Other errors may also be observed manifesting as regular wavy curves resulting from overlap add inaccuracies, or less predictable squiggles arising from sloppily shifted or poorly crossfaded grains.

All Moving Spectral Average graphs are contained in Appendix A, Section 7.2, and the MATLAB scripts that generated the graphs are contained in Appendix B, Sections 8.4.9 and 8.4.10.

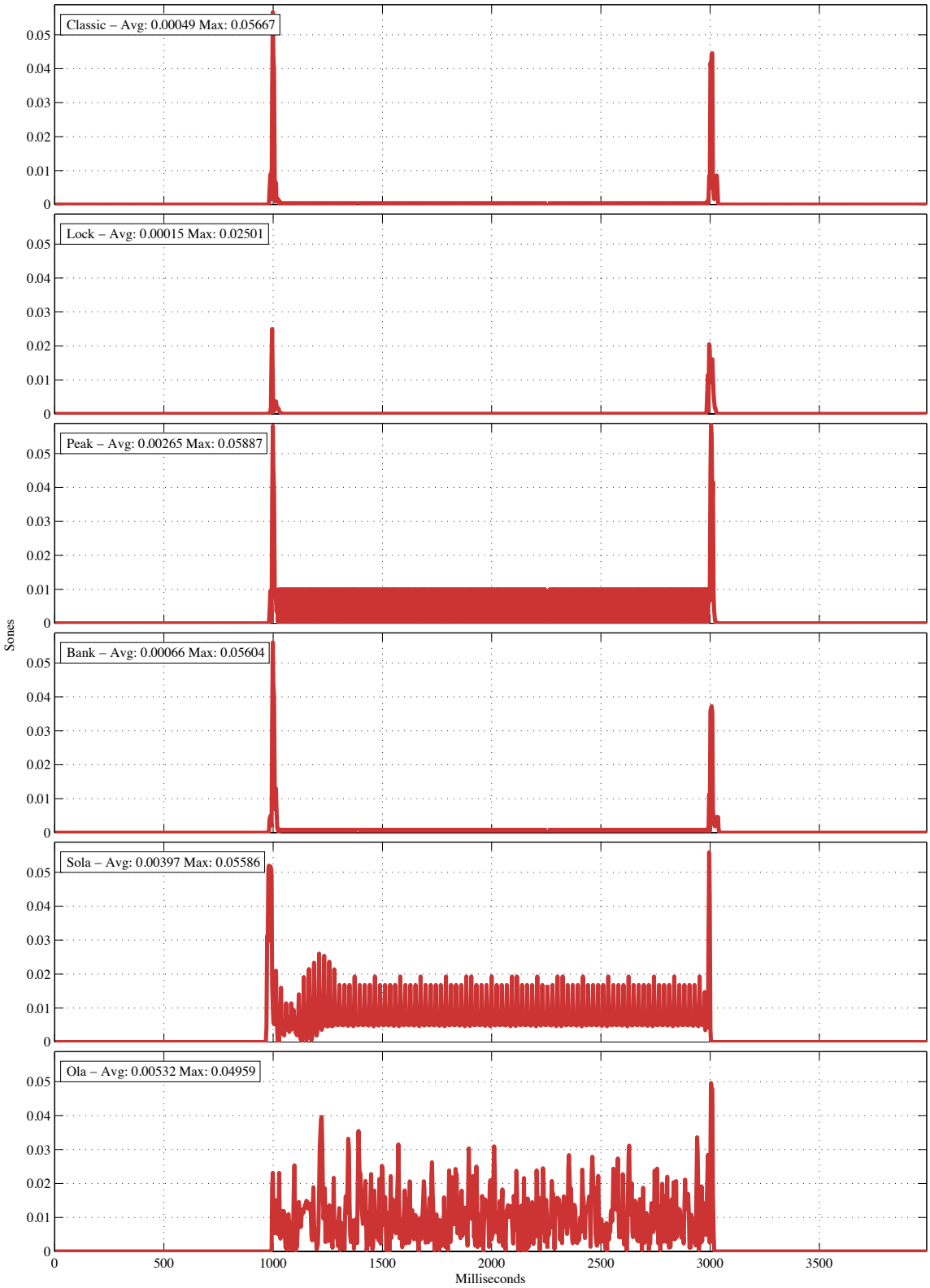
#### 5.2.1 – Sine Wave Results

Graphs 5.2.1.1 and 5.2.1.2 show the overview and detailed error curves resulting from the stretched sine wave. The common error traits shared between all algorithms are

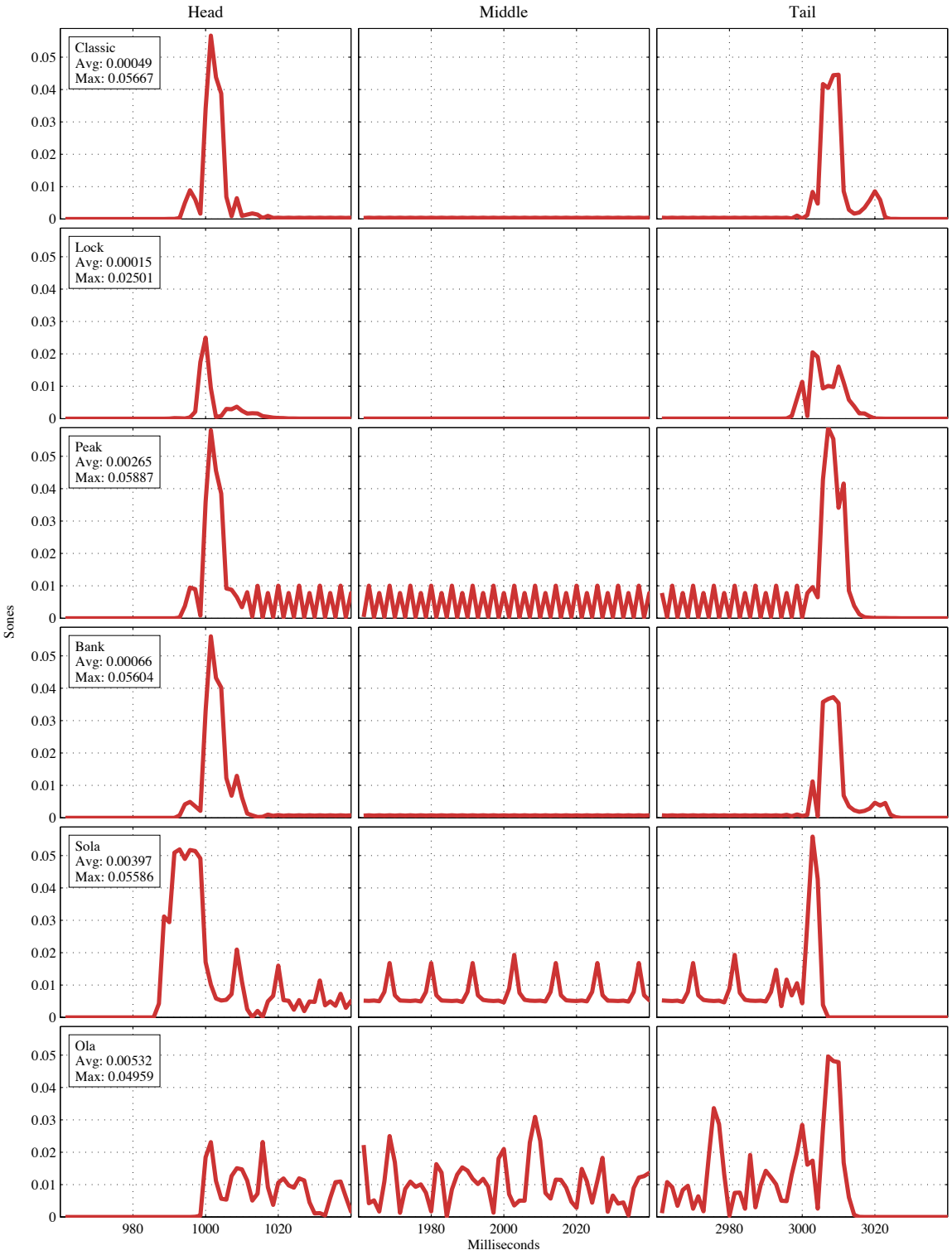
the spikes at the beginnings and ends of their output signals. These spikes indicate how accurately each algorithm transitions from silence to signal, and signal to silence. These spikes are also visible in the detailed graphs, which give a better view of each algorithm's accuracy. The Phase-Locked Vocoder spikes show the least amount of error in terms of magnitude, and a little less than average error in terms of duration. Furthermore, the head spike is centered in the graph and the tail spike is shifted by approximately five milliseconds, with a flat steady-state region in the middle. In other words, the spikes are short and slightly less than average width, with the head spike happening on time, the tail spike occurring late, and negligible error in the middle. Both the Classic and Oscillator Bank Phase Vocoders' spikes display significant magnitude errors, longer durations that are late, and very little error in the middle of their signals. Notice the similar shapes of these algorithms' error spikes, with minutely longer errors arising from the Oscillator Bank Phase Vocoder, likely related to its linear interpolation of spectral data between resynthesis frames. The Peak Tracking Phase Vocoder exhibits the highest magnitude head and tail spikes, but they are slightly narrower than the other phase vocoders. Unfortunately this algorithm also generates a significant amount of distortion in the form of amplitude modulation, visible between the spikes. This modulation almost certainly arises from the Peak Tracking Phase Vocoder's frame sliding scheme, where the output hop size is equivalent to the input hop size multiplied by the stretch factor. The Synchronous Overlap Add graph displays significant error spikes with an early and wide head spike but a narrow, and nearly on time tail spike. This algorithm also exhibits significant errors in the steady-state portion of its output, starting as an unstable lump of error that settles into a kind of miniature spike train with some slightly taller spikes. As

previously discussed, this algorithm's attempt to cross correlate a self-similar signal introduces repetitive amplitude errors into the signal as it inaccurately picks cross fade points. Finally, the Overlap Add algorithm shows more accuracy in its head portion, with a more significant and late error spike at its tail, and noticeable noisy error throughout its middle due to its randomly time shifted grain spacing scheme.

**Graph 5.2.1.1 –** Sine Wave Moving Spectral Average Error Overviews



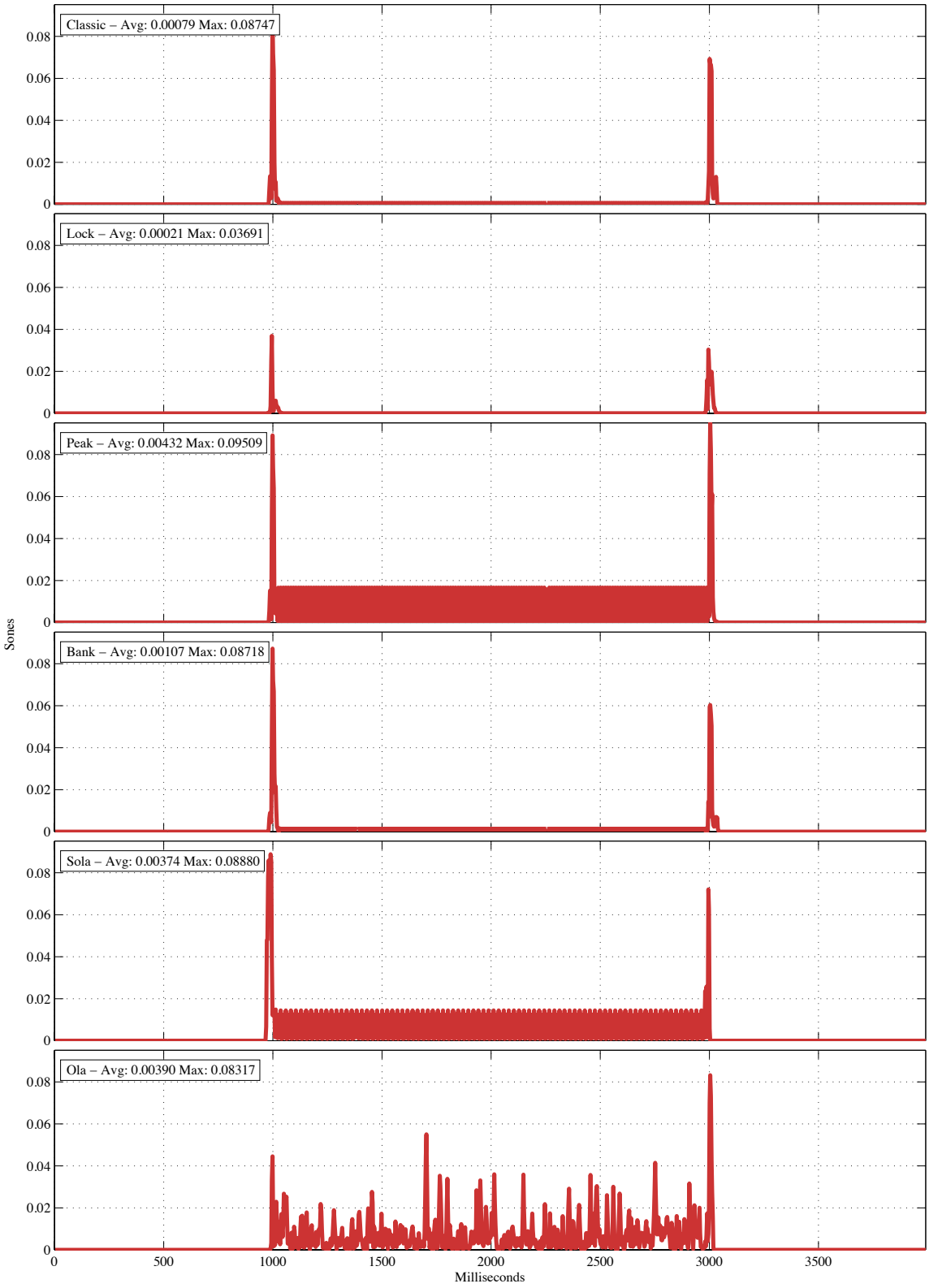
**Graph 5.2.1.2 –** Sine Wave Moving Spectral Average Error Details



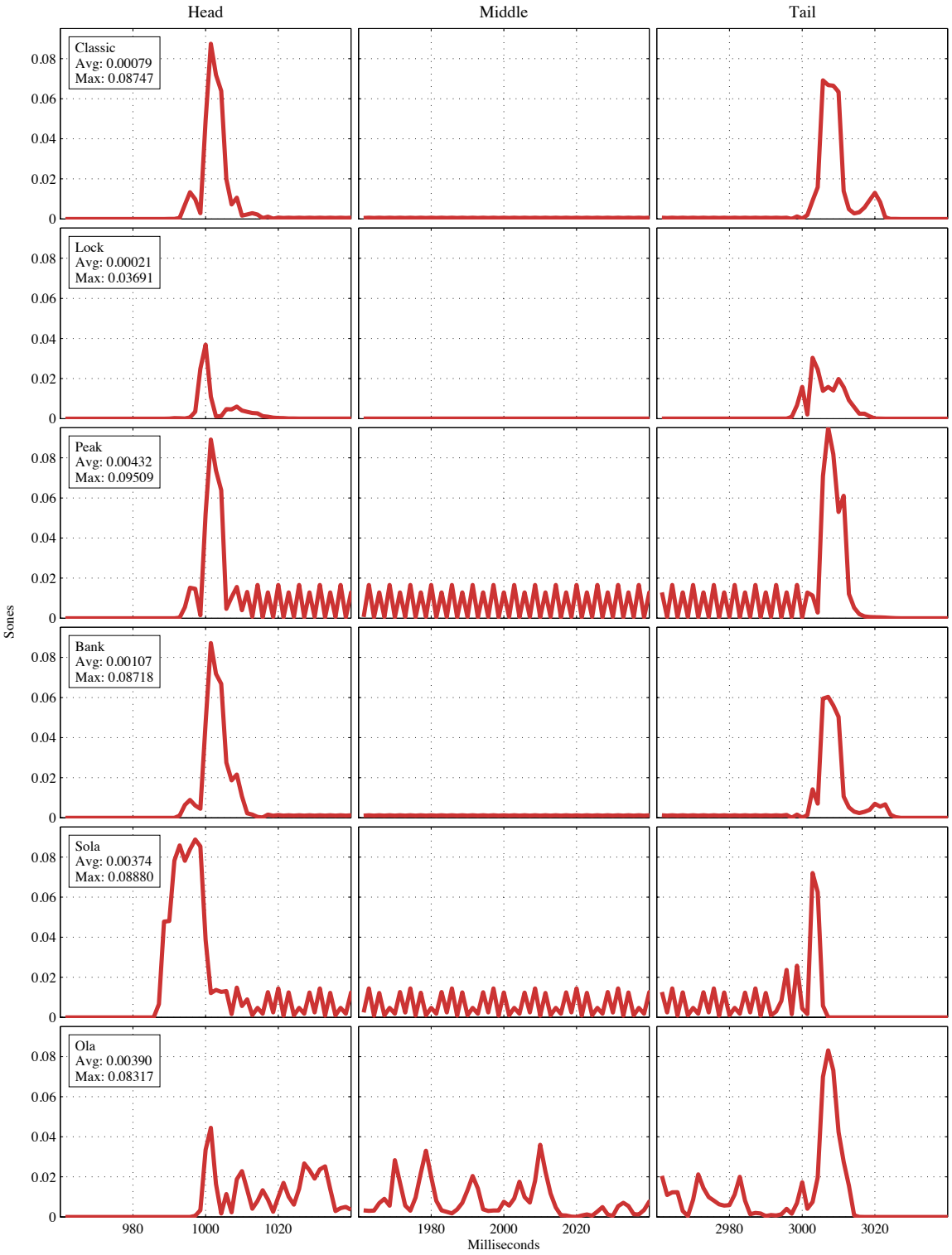
### 5.2.2 – Square Wave Results

The error curves corresponding to the stretched square wave are contained in Graphs 5.2.2.1 and 5.2.2.2. These demonstrate head and tail error spikes that are nearly identical to those contained in the sine wave Moving Spectral Average curves. This is expected since both signals are steady-state and have identical amplitude envelopes and durations. They differ only in harmonic content, which leads to slightly different errors in the middle sections of the square wave error curves. More specifically, these middle errors are shown where the Peak Tracking Phase Vocoder displays a modulation pattern of subtly alternating spikes, the Synchronous Overlap Add technique does not have a blob of error near its head, instead settling quickly into a more regular spike train, and the Overlap Add algorithm exhibits a slightly less noisy error curve.

**Graph 5.2.2.1 –** Square Wave Moving Spectral Average Error Overviews



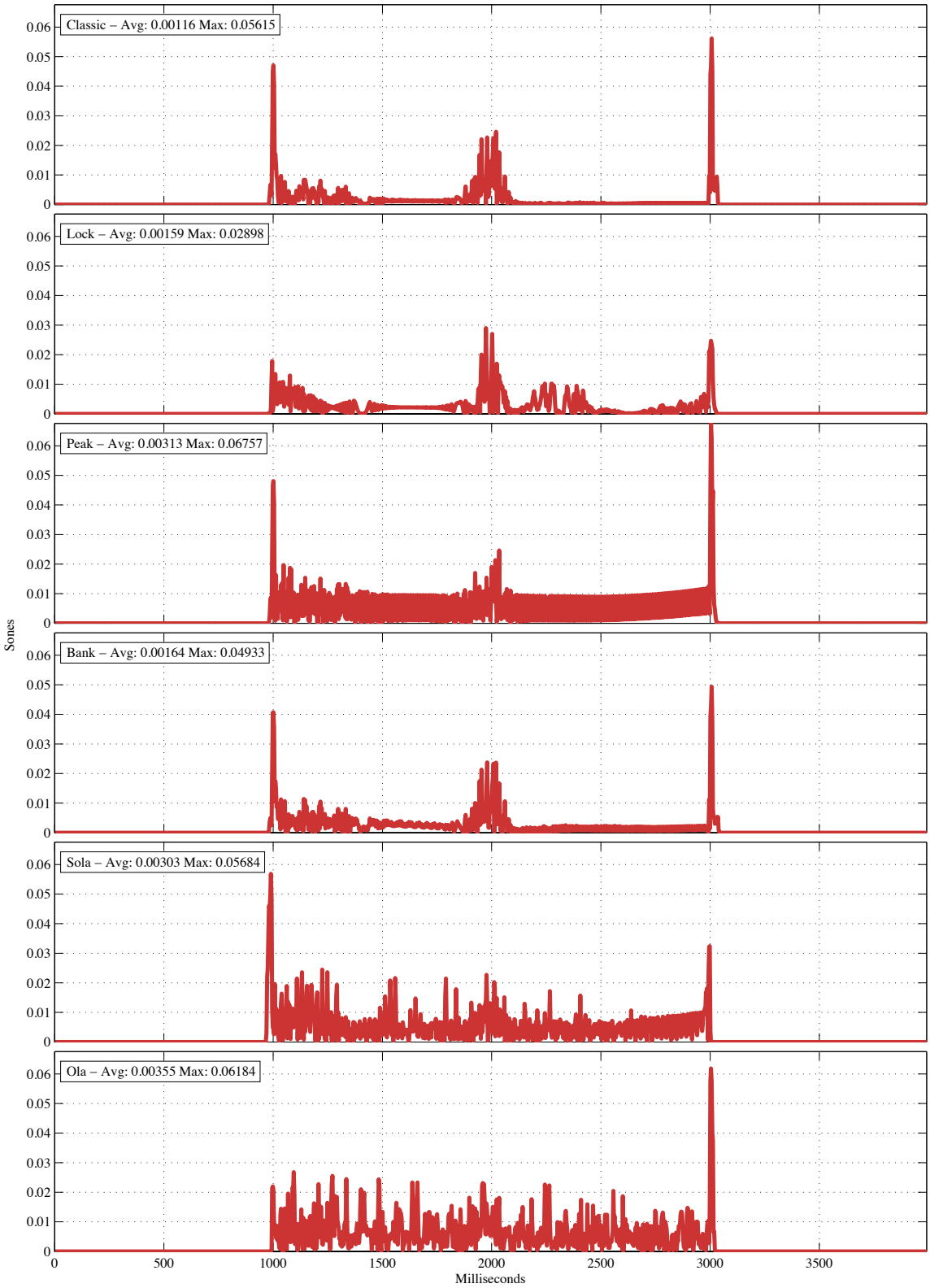
**Graph 5.2.2.2 –** Square Wave Moving Spectral Average Error Details



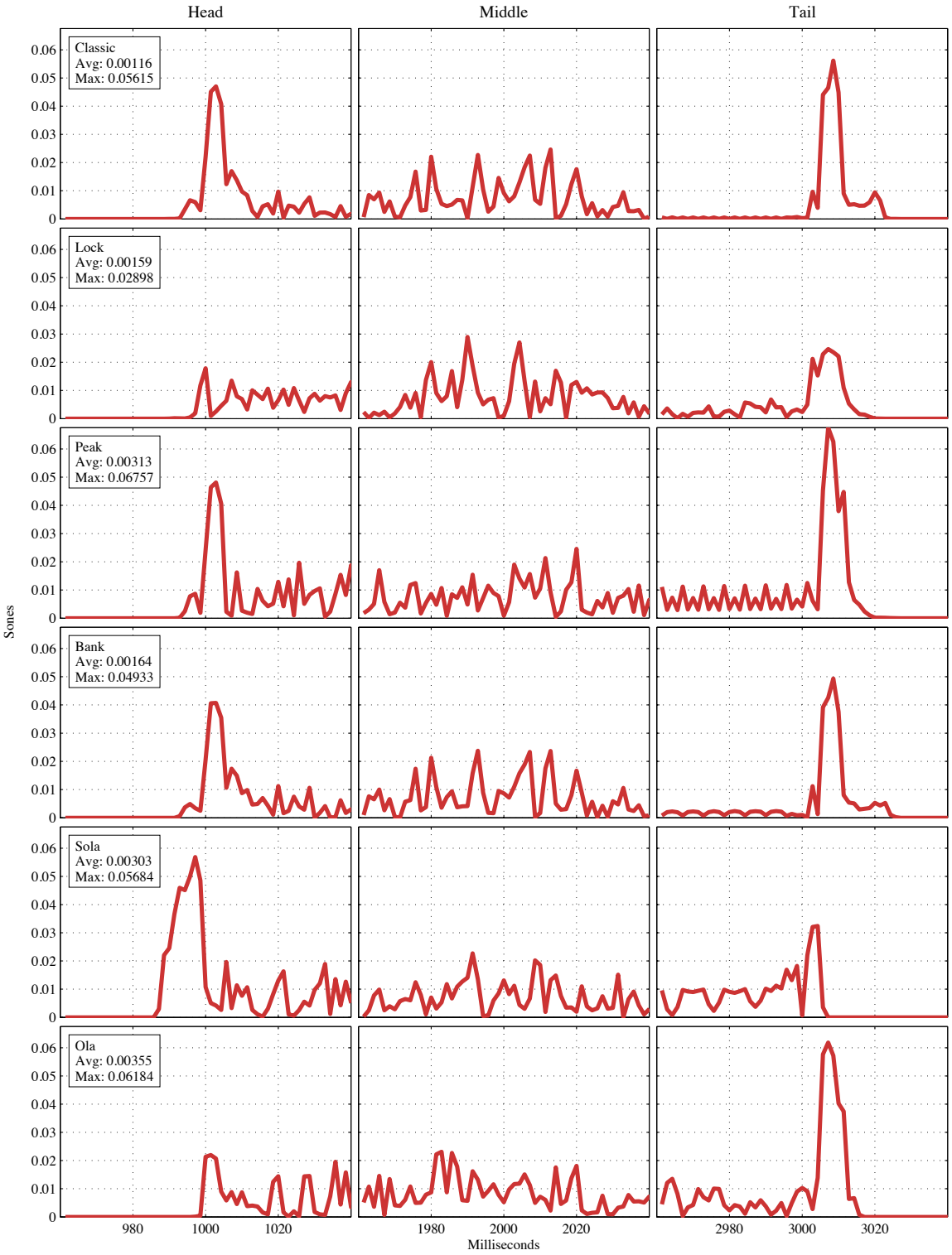
### 5.2.3 – Sinusoidal Sweep Results

The stretched sinusoidal sweep error curves are displayed in Graphs 5.2.3.1 and 5.2.3.2. This signal is deliberately designed to be difficult for time stretching, and the error curves reflect this. The most accurate algorithm appears to be the Classic Phase Vocoder, even though it exhibits late head and tail spikes, and a lump of error in the middle of the signal where the swept and fixed sinusoid frequencies cross. While the frequencies of the two sinusoids are in close proximity, their energies start to occupy the same spectral bins, leading to inaccuracies as the spectral algorithms attempt to determine which frequency merits attention. All algorithms actually appear to exhibit difficulty with this crossing, but the Classic and Oscillator Bank Phase Vocoders show the least amount of error. The Oscillator Bank error curve is quite similar to the Classic curve, and even displays a middle section that is nearly identical. However, the Oscillator Bank's error curve is slightly larger in magnitude and shows a bit more modulation. The Phase-Locked Vocoder is the next most accurate algorithm with a small, on-time head spike, and a small but late tail spike. This algorithm displays more error in the swept portion of the signal with an additional elongated lump of error after the mid point, more error leading into the tail, and a slight magnitude offset before the middle. The Peak Tracking Phase Vocoder is next, with narrower but late head and tail spikes, and more modulation error arising from its time stretching scheme. Finally, the granular algorithms both display significant errors, with the Synchronous Overlap Add technique appearing slightly more accurate than the Overlap Add technique.

**Graph 5.2.3.1 –** Sinusoidal Sweep Moving Spectral Average Error Overviews



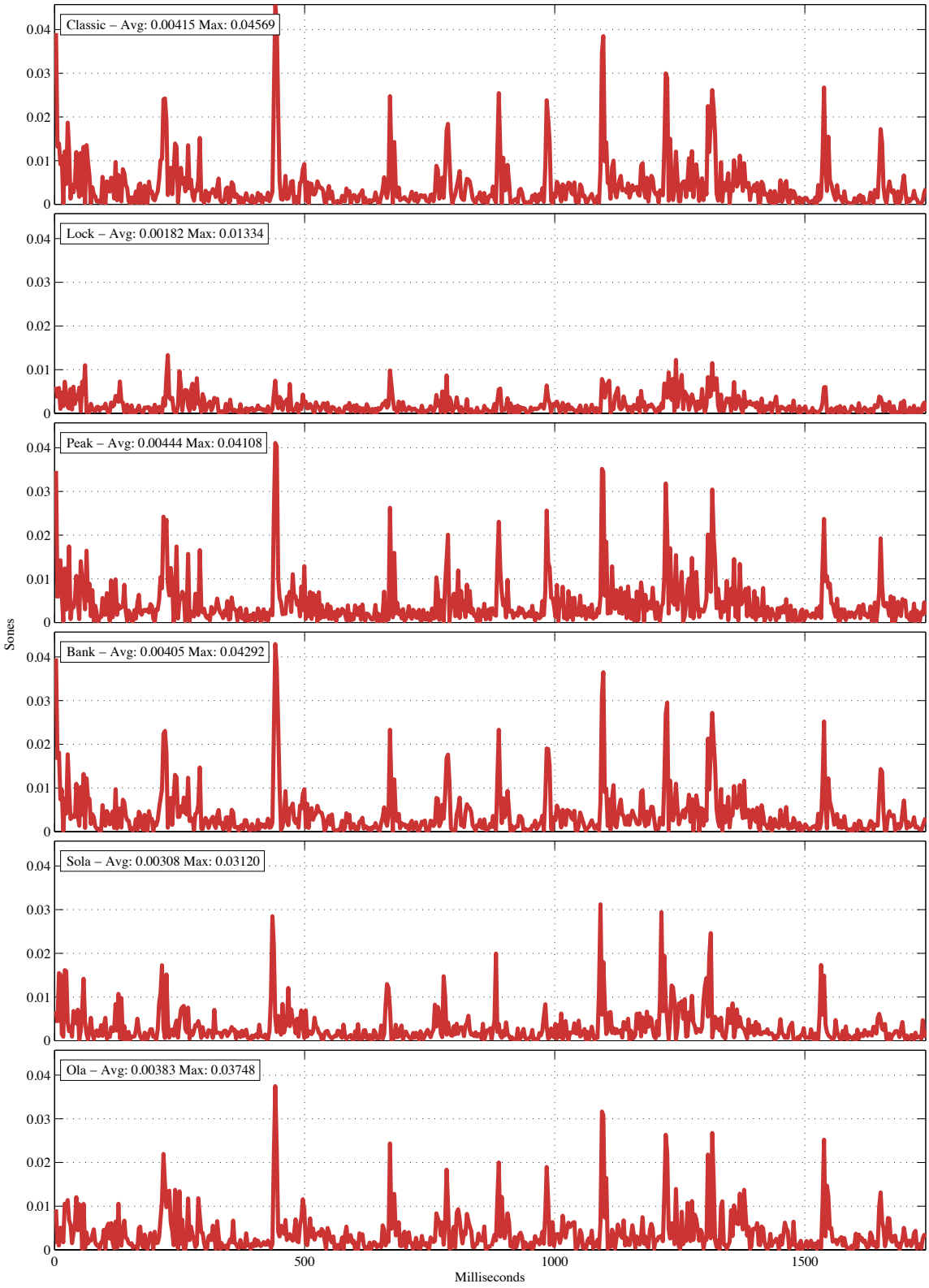
**Graph 5.2.3.2 –** Sinusoidal Sweep Moving Spectral Average Error Details



### 5.2.4 – *Amen Break* Results

Graph 5.2.4.1 displays the *Amen Break* Moving Spectral Average error curves. These graphs are quite similar to each other, with each algorithm generating errors of roughly the same magnitudes at the same locations. Percussive transients are notoriously difficult to time stretch, since they generate temporally inaccurate data within DFT bins and are coarsely smeared by granular algorithms. The curves resulting from these transients look like collections of spikes, and these spikes correspond to the locations of event onsets within the signal. More specifically, each spike occurs during the percussive attack of a drum kit sound, with the largest spikes corresponding to the loudest snares on beats two and four. These spikes are in the same location for each algorithm, however, the Phase-Locked Vocoder is notably more accurate than any other. The Synchronous Overlap Add technique appears to be the next most accurate, but its curve is only marginally better than the others.

**Graph 5.2.4 –** *Amen Break* Moving Spectral Average Error Overviews



### 5.2.5 – *Autumn in New York* Results

The error curves arising from the *Autumn in New York* signal are displayed in Graph 5.2.5.1. These graphs are all quite similar in terms of magnitude and contour, making meaningful observation difficult. The errors seem to follow the amplitude of the stretched signal, mimicking its time domain shape. The *Autumn in New York* signal is relatively dense and variable in terms of harmonic content, so it is not surprising that errors in the stretched signal tend to follow the shape of the input signal. In other words, larger errors correspond to larger input amplitudes. Again, the Phase-Locked Vocoder exhibits the most accuracy, but in this case it is not significantly better.

**Graph 5.2.5 –** *Autumn in New York* Moving Spectral Average Error Overviews

### 5.2.6 – *Peaches en Regalia* Results

Graph 5.2.6.1 shows the *Peaches en Regalia* error curves. As with the previous graphs, these error curves are remarkably similar to each other and exhibit many of the same traits that were previously observed. This signal contains drums, so most of the tall spikes correspond to snare drum strikes or loud cymbal crashes, and the overall error curve contours follow the amplitude of the input signal. The Phase-Locked Vocoder displays the most accuracy but is only marginally better than any of the other algorithms.

**Graph 5.2.6 –** *Peaches en Regalia* Moving Spectral Average Error Overviews

### 5.3 – Error Spectrogram

The following six subsections display the Error Spectrograms for each signal and provide a general discussion of each set. Again, each signal is shown in its entirety, with additional detailed spectrograms of the synthetic signals that focus on the head, middle, and tail sections. These detailed views show areas of interest within the signals, illustrating where waveforms begin, are steady or cross, and end.

The Error Spectrogram technique illustrates a two dimensional view of spectral error over time. This technique represents magnitude errors across the frequency spectrum and provides a comprehensive view of an algorithm's output. Particular error features of a signal may be observed in terms of the features' locations across both the frequency domain and time domain. Unlike error curves generated by the other two analysis techniques, the graphs created by this technique do not include specific measurements. Rather, they provide visual overviews by displaying patterns and regions of error within the stretched signals.

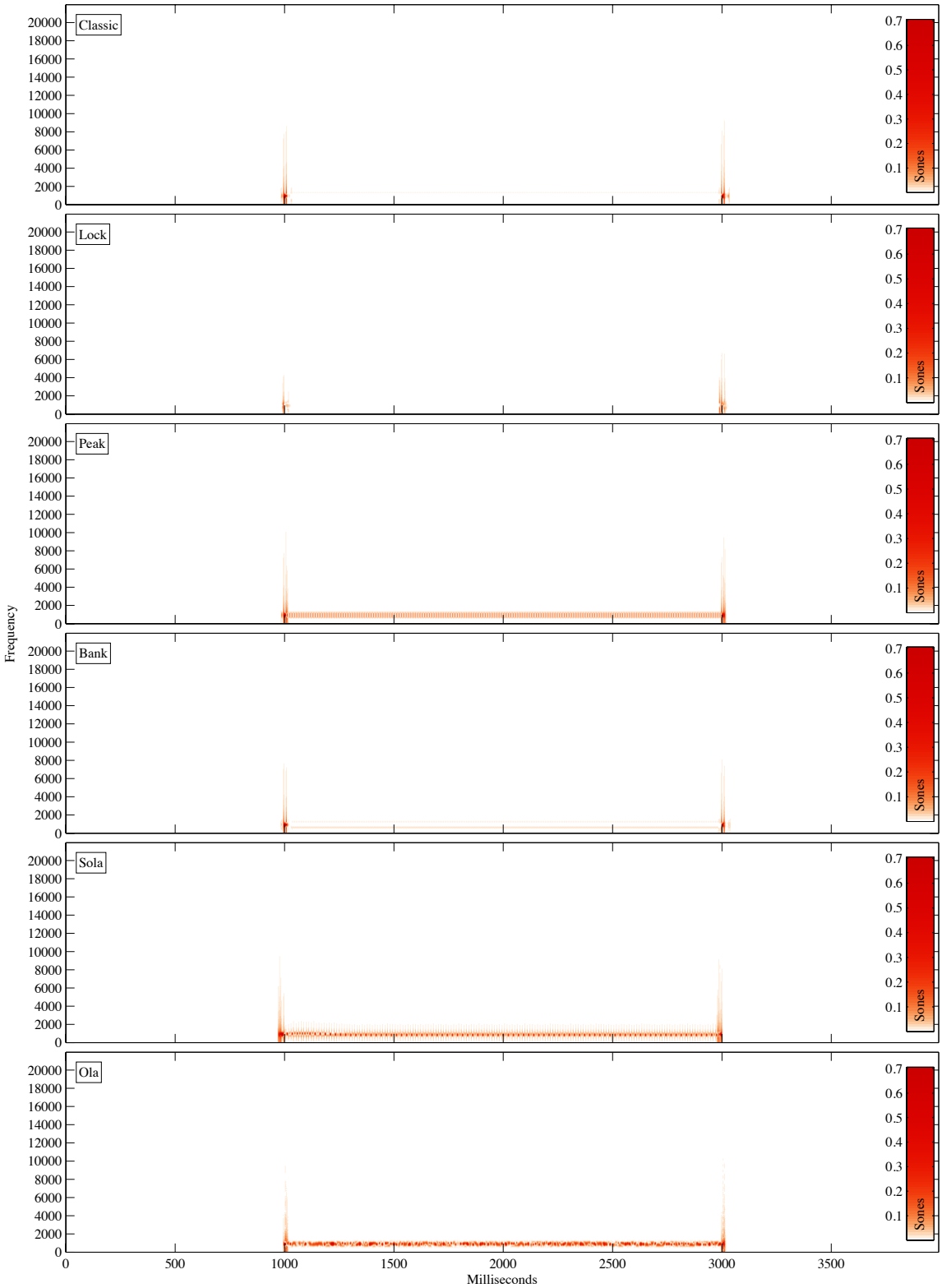
All Error Spectrograms are contained in Appendix A, Section 7.3, and the MATLAB scripts that generated the graphs are contained in Appendix B, Sections 8.4.11 and 8.4.12.

#### 5.3.1 – Sine Wave Results

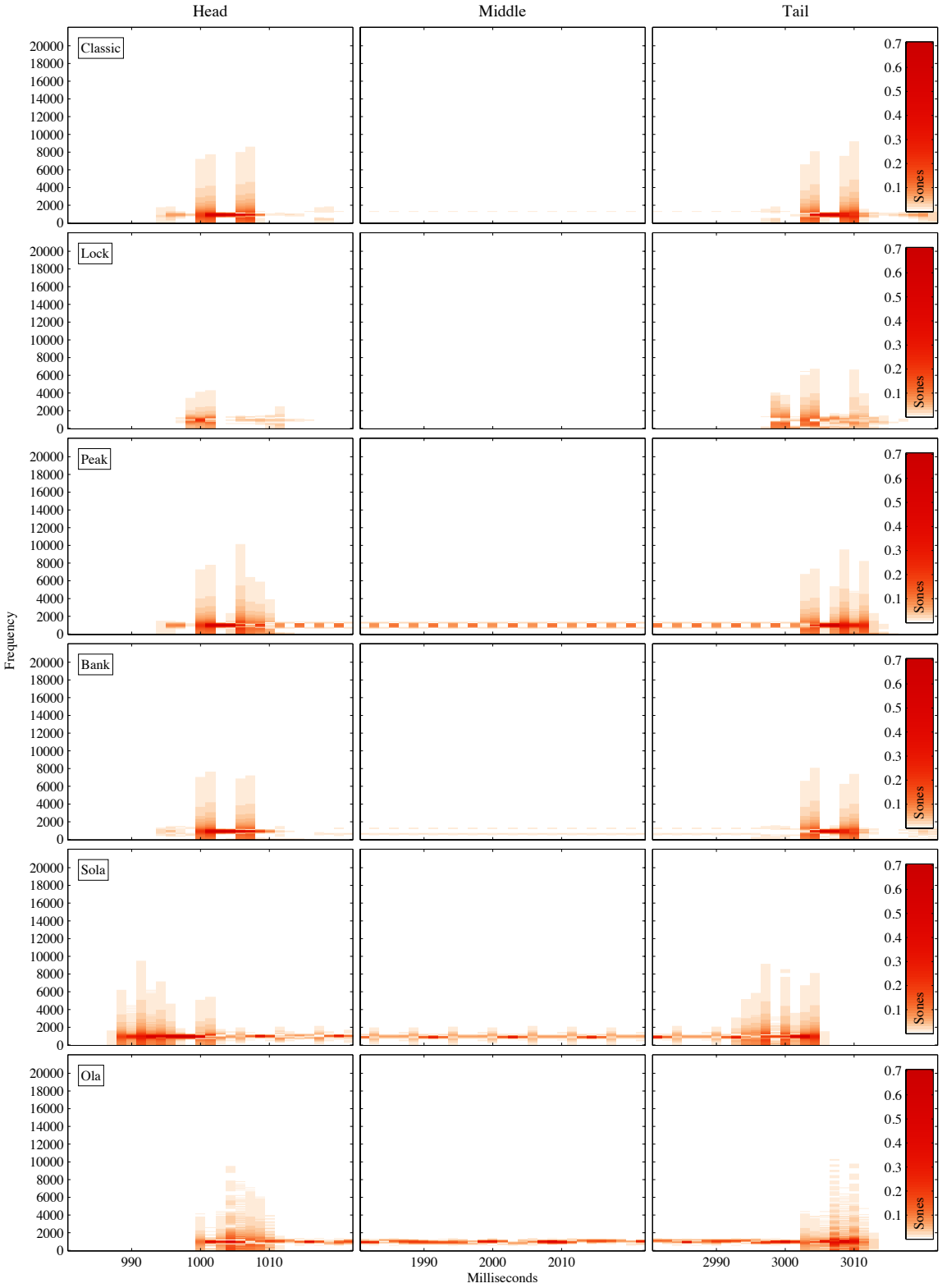
Error Spectrograms for the stretched sine wave are contained in Graphs 5.3.1.1 and 5.3.1.2. These spectrograms display similar results to the Moving Spectral Average curves in Section 5.2.1, but provide more insight about the spectral envelopes of the stretched signals. The overview and detail spectrograms clearly show that the Phase-Locked Vocoder has the least amount of error. Next, the Classic and Oscillator Bank

Phase Vocoders, have very small errors and remarkably similar head and tail sections. Following these, the Peak Tracking Phase Vocoder's modulation errors are visible as a subtle dotted line at the sine wave's fundamental frequency. The Synchronous Overlap Add technique exhibits a more distinct dotted line of error, and the middle of its detailed spectrogram displays small error pulses, interleaved with double pulses of error, spanning several spectral bins. Finally, the Overlap Add spectrogram displays a noisy line of errors around the sine wave's fundamental frequency, with random spacing and a range of a few spectral bins.

**Graph 5.3.1.1 –** Sine Wave Error Spectrograms



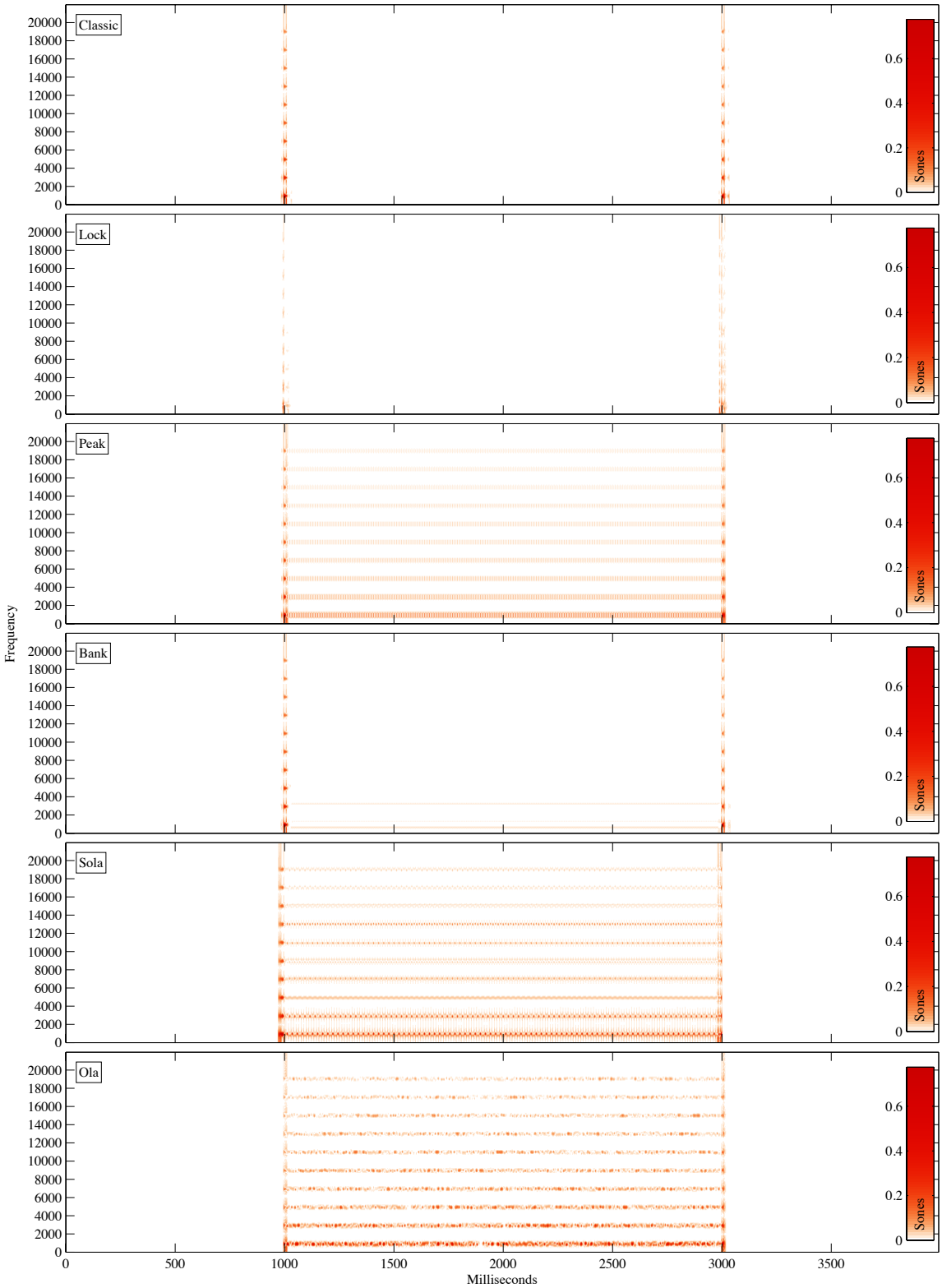
**Graph 5.3.1.2 –** Sine Wave Error Spectrogram Details



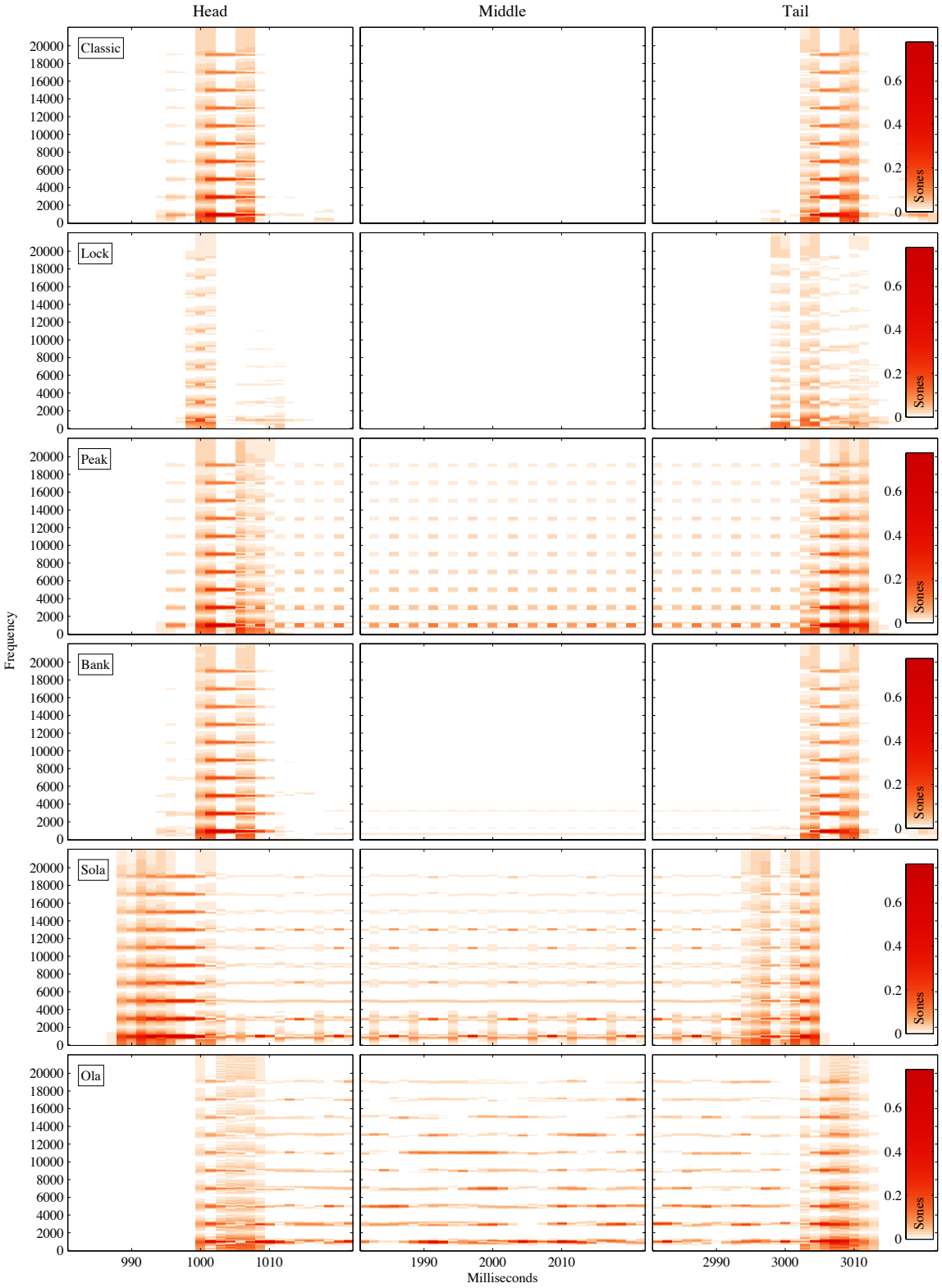
### 5.3.2 – Square Wave Results

Graphs 5.3.2.1 and 5.3.2.2 contain the stretched square wave Error Spectrograms. Again, these spectrograms display errors that are quite similar to the square wave Moving Spectral Average curves in Section 5.2.2, but with more information about the square wave's spectrum over time. The Phase-Locked, Classic, and Oscillator Bank Phase Vocoders all provide accurate results (as previously discussed), and their spectrograms provide little additional insight into their behavior. However, the other three algorithms exhibit interesting additional errors. First, the Peak Tracking Phase Vocoder displays a repetitive grid-like pattern of errors that attenuate according to the corresponding harmonic, and the magnitude of these errors appears to be directly related to each harmonic's original amplitude. As discussed in Section 5.2.1, these errors are related to the Peak Tracking Phase Vocoder's frame sliding scheme, which is the opposite of most other phase vocoders. Next, the Synchronous Overlap Add technique shows very strange errors, which manifest as alternating patterns affecting different harmonics from frame to frame. The fundamental and first harmonic errors appear to alternate every other frame, as do sets of upper harmonic errors. In between the alternating pattern, the other errors appear to be less focused on the harmonics, instead spreading between them into adjacent frequency regions. Finally, the Overlap Add algorithm displays randomized errors that affect various harmonics of the square wave, caused by the randomness in the algorithm's time stretching scheme.

**Graph 5.3.2.1 –** Square Wave Error Spectrograms



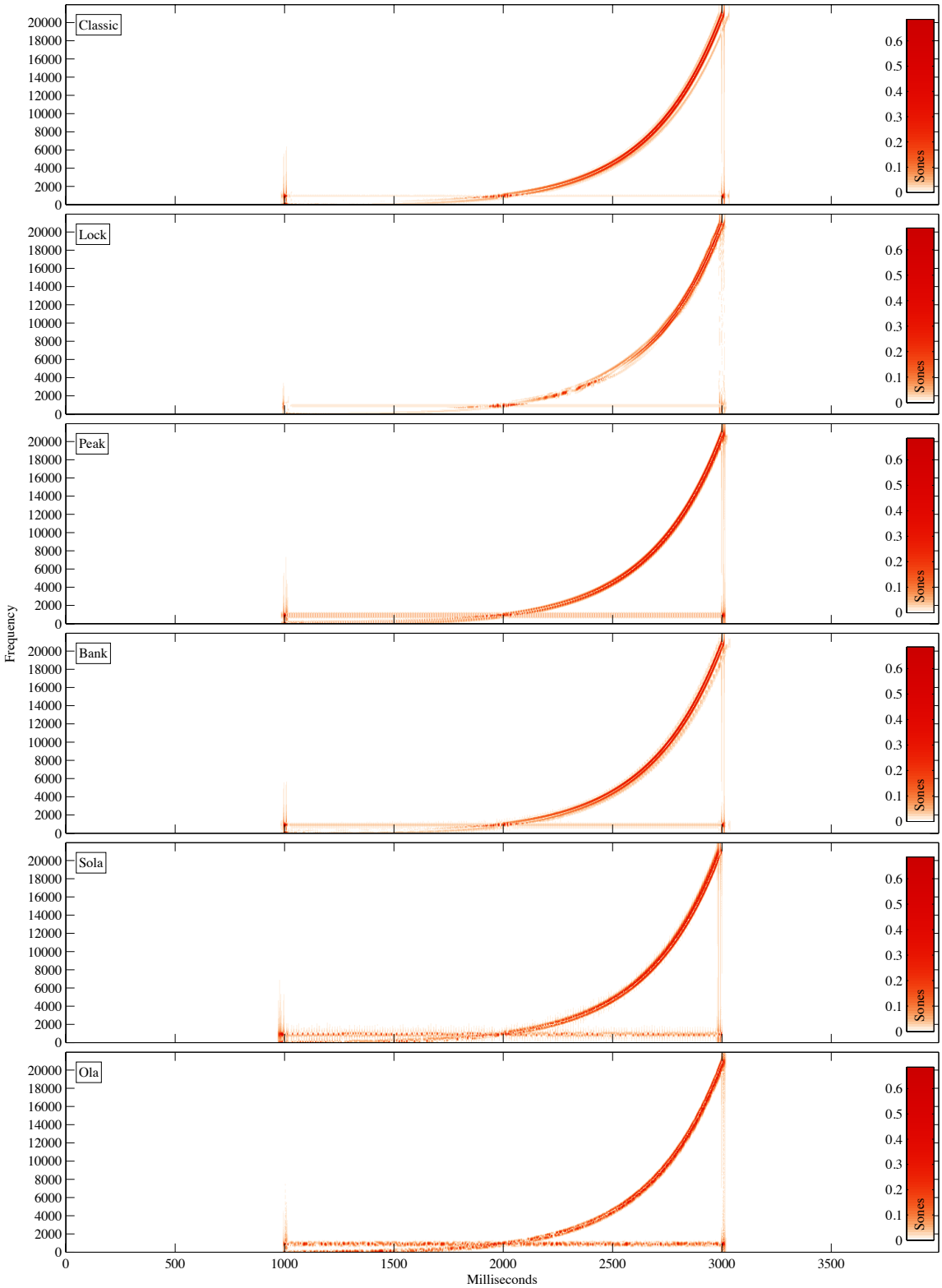
**Graph 5.3.2.2 –** Square Wave Error Spectrogram Details



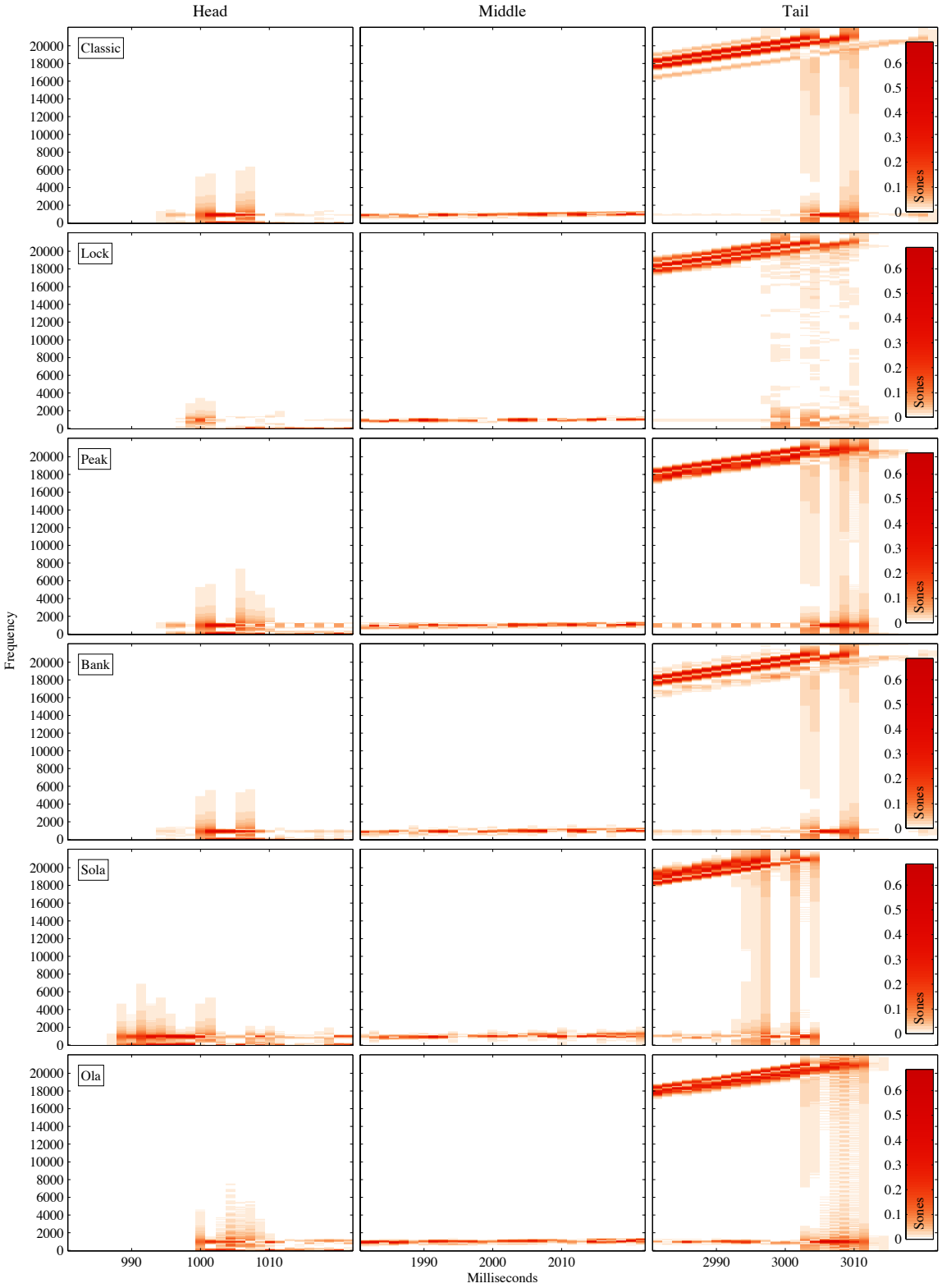
### 5.3.3 – Sinusoidal Sweep Results

The sinusoidal sweep Error Spectrograms are displayed in Graphs 5.3.3.1 and 5.3.3.2. These graphs display errors where the swept tone crosses the steady tone, and also demonstrate increasing errors as the swept tone accelerates upward in frequency towards the end of the synthesis region. The detailed graphs illustrate predictable errors during the tail sections of the Classic and Oscillator Bank Phase Vocoders, which closely follow the movement of the swept tone. The other algorithms' tail sections demonstrate less predictable errors, and the artifacts discussed in Section 5.3.2 are still visible along the steady tone.

**Graph 5.3.3.1 –** Sinusoidal Sweep Error Spectrograms



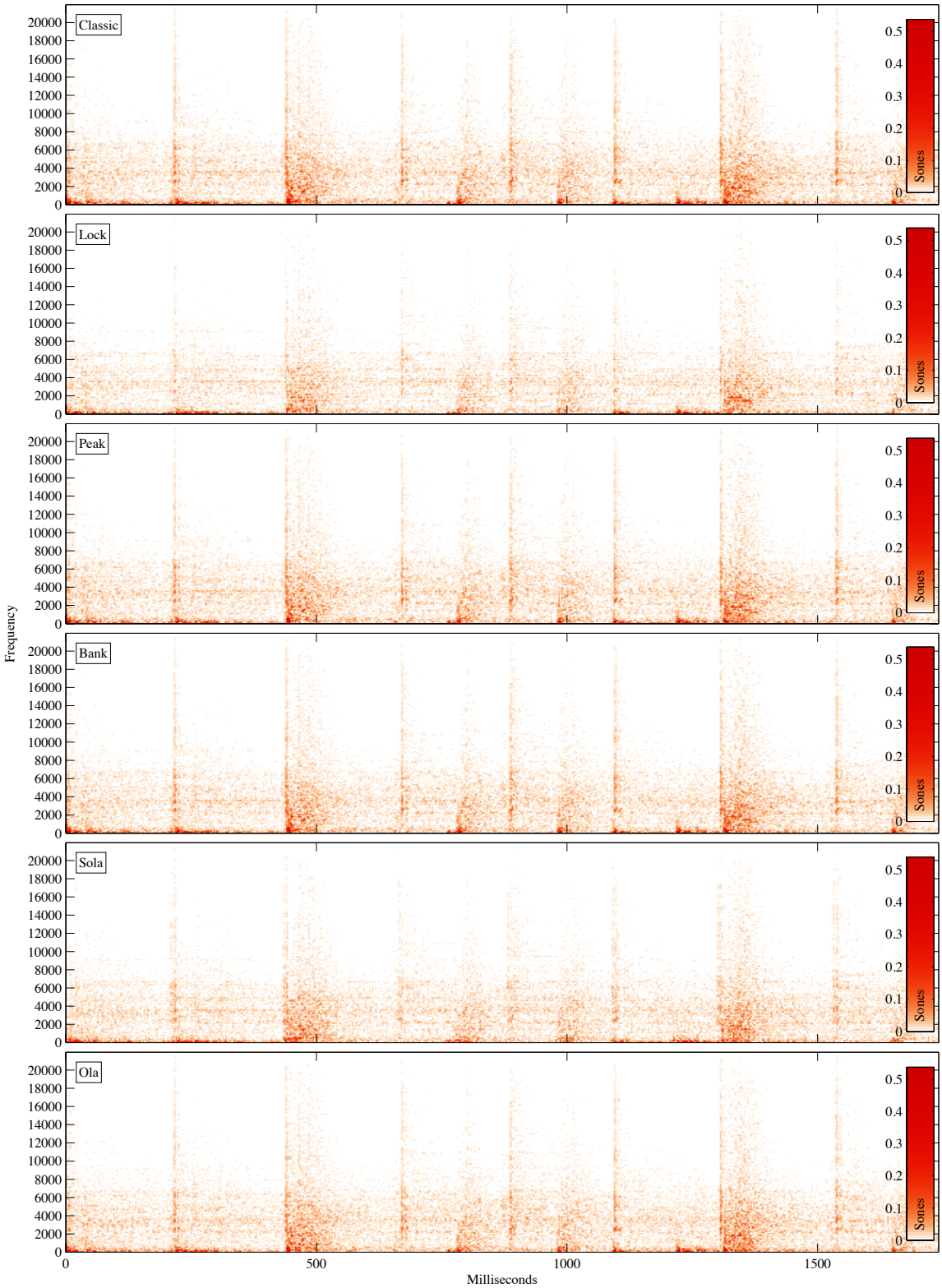
**Graph 5.3.3.2 –** Sinusoidal Sweep Error Spectrogram Details



### 5.3.4 – *Amen Break* Results

Graph 5.3.4 contains the *Amen Break* Error Spectrograms. Since real-world signals like this one are unpredictable, with constantly changing spectral content, there are not many significant features to discuss. However, the Phase-Locked Vocoder displays slightly less error in its spectrogram, which looks a bit darker than the others. All of the algorithms show errors at the onsets of percussive sounds, with the snare transients producing the most noticeable errors. Furthermore, the errors appear to mimic the spectral magnitude envelopes of the sounds, with larger errors in the lower frequencies and smaller errors in the upper frequencies.

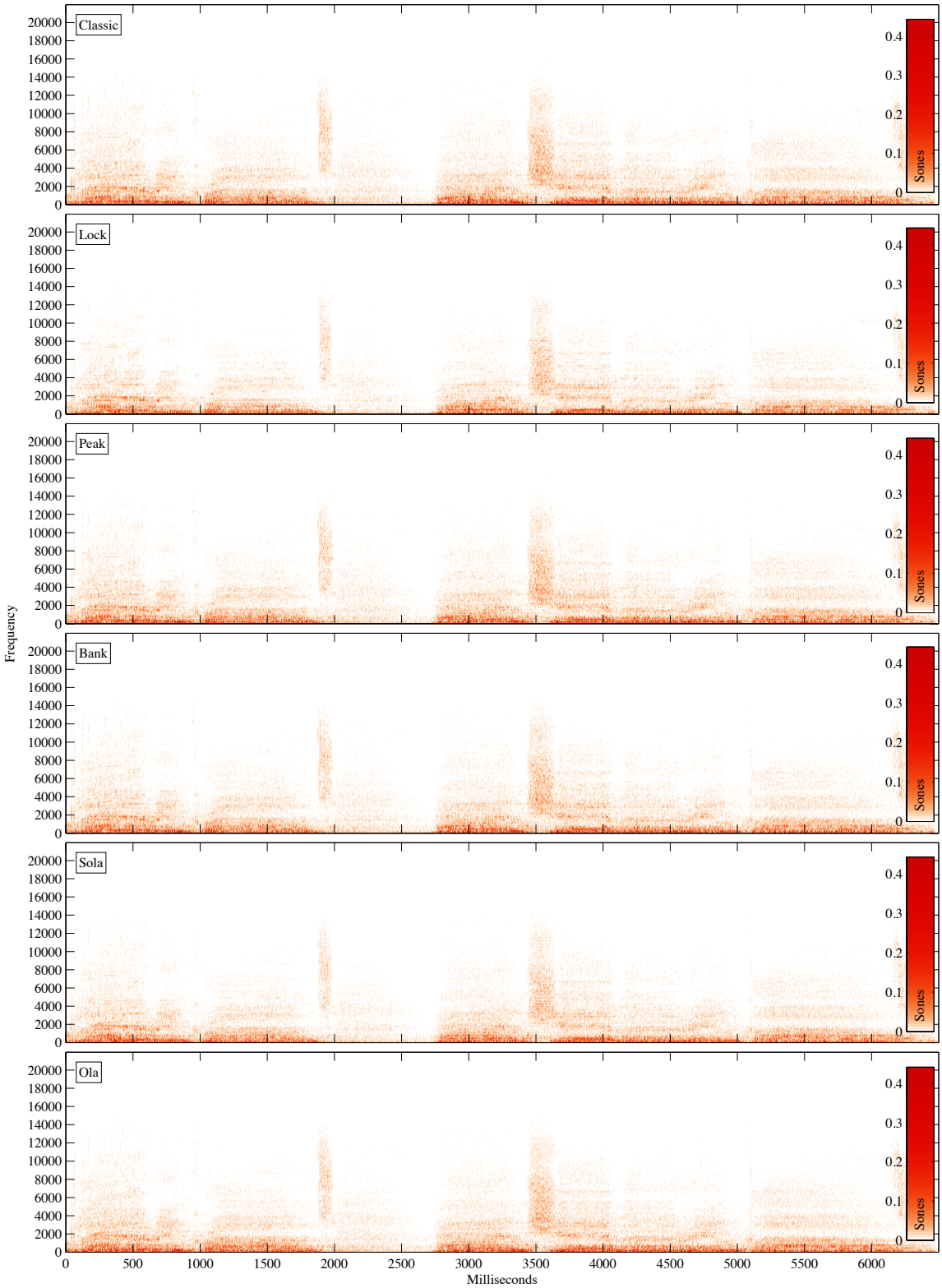
**Graph 5.3.4 –** *Amen Break* Error Spectrograms



### 5.3.5 – *Autumn in New York* Results

The *Autumn in New York* Error Spectrograms are displayed in Graph 5.3.5. The *Autumn in New York* signal exhibits few significant features for comparison. Most noticeable are the errors generated by sibilance arising from the "s" at the end of the word "crowds," and the "sh" at the beginning of the word "shimmering," occurring at 2000 and 3500 milliseconds respectively. As with the *Amen Break*, the errors shown here follow the spectral envelopes of the original sound, with more errors around the concentration of energy at low frequencies, progressing toward fewer errors in the upper frequencies.

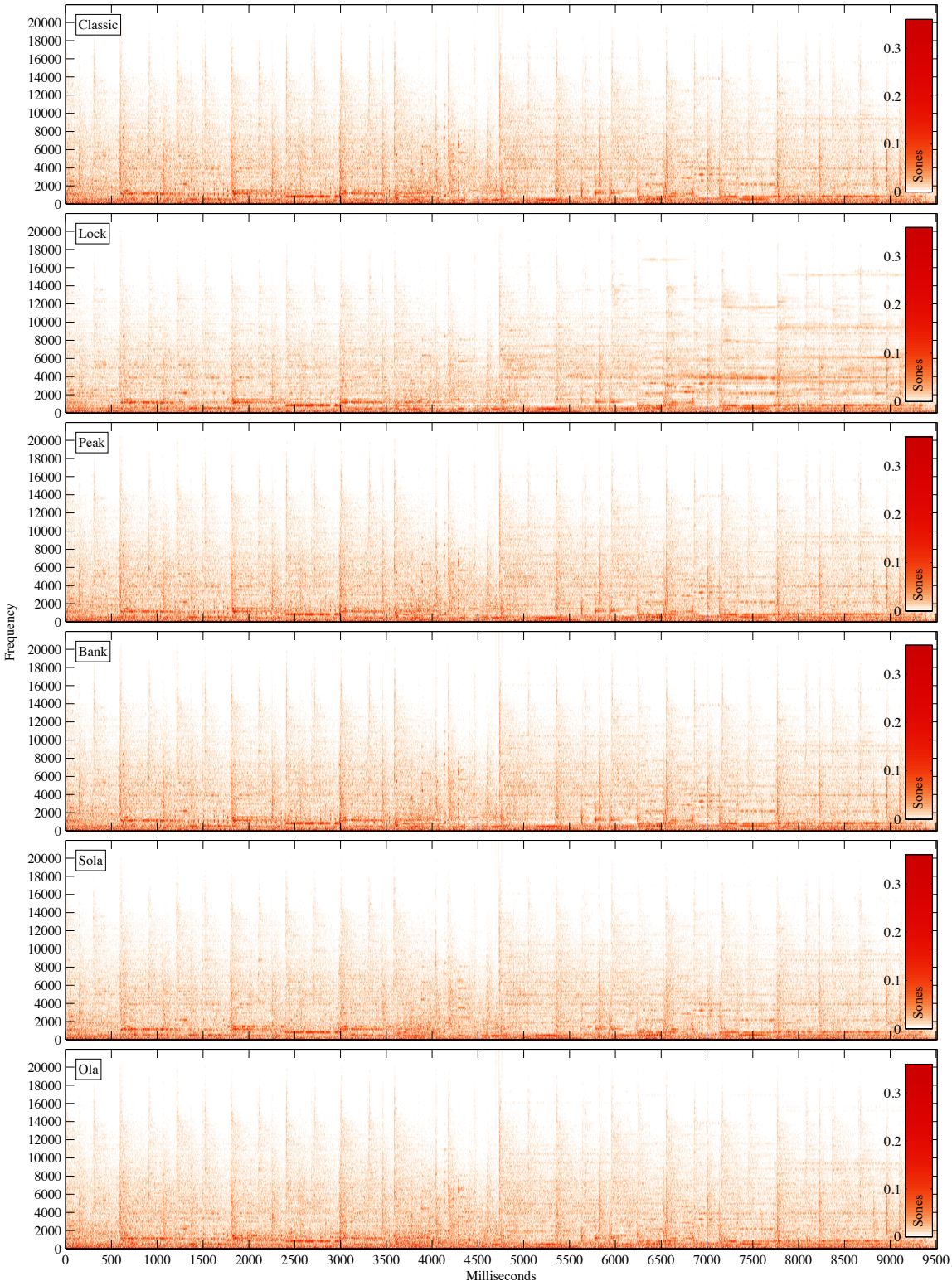
**Graph 5.3.5 –** *Autumn in New York* Error Spectrograms



### 5.3.6 – *Peaches en Regalia* Results

Graph 5.3.6 shows the *Peaches en Regalia* Error Spectrograms. This signal contains the densest spectrum of all, consequently appearing as a continuous blur of errors in the spectrograms. This signal also contains percussive events, which appear again as vertical lines, with salient snare drum transient. These spectrograms look remarkably similar, though the last third of the Phase-Locked Vocoder spectrogram exhibits more horizontal error tracks than the others. This likely occurs due to notes with harmonics that overlap among adjacent spectral bins, leading to incorrectly manipulated phases due to the phase locking scheme. These errors also arise from the inability of discrete Fourier resynthesis to handle more than one frequency per bin. Again, all spectrograms exhibit more errors at the lower end of the frequency spectrum, and fewer errors toward the upper frequency range.

**Graph 5.3.6 –** *Peaches en Regalia* Error Spectrograms



### 5.4 – Subjective Listening

Subjective listening observations are discussed in the following six sections, and are arranged by signal type. As with any subjective listening, the observations described here are perceptual in nature and may differ between listeners.

#### 5.4.1 – Sine Wave Results

The Oscillator Bank Phase Vocoder signal sounds very close to the ideal signal, is difficult to tell apart from the ideal signal, and sounds like a single sinusoid with the correct attack and release characteristics. The Classic Phase Vocoder also sounds very close and purely sinusoidal, however, its release has a slightly more noticeable click than the ideal signal. The Phase-Locked Vocoder also sounds like a pure sinusoid and has a clean attack, but its release has noticeably less of a click and sounds slightly attenuated, when compared with the ideal signal. Subtle double clicks are audible in both the attack and release of the Peak Tracking Phase Vocoder, which also contains audible amplitude modulation in its sinusoidal portion. This amplitude modulation sounds similar to pitched ring modulation, due to the regular period of the modulation pattern. The Synchronous Overlap Add signal has an elongated attack, characterized by a period of unstable amplitude modulation that settles into a more regular, ring-modulation-like pattern, after the steady-state signal begins. This signal also contains a more noticeable click when it ends. The Overlap Add signal has double clicks during its attack and release portions, but more significantly, it sounds strikingly amplitude modulated at random, giving its sinusoidal portion a diffuse and warbling character.

### 5.4.2 – Square Wave Results

Like the sine wave, the square wave sounds most accurate when stretched by the Oscillator Bank Phase Vocoder, which produces a result that is nearly indistinguishable from the ideal signal. The Classic and Phase-Locked Vocoders also sound very good with accurate attacks and pure steady-state portions, however, they exhibit slightly inaccurate releases when the signal ends, with a slightly more noticeable click from the Classic Phase Vocoder, and a slightly attenuated click from the Phase-Locked Vocoder. The Peak Tracking Phase Vocoder creates accurate attack and release portions of the stretched signal, but its steady-state portion unfortunately contains noticeable periodic amplitude modulation resembling low-frequency ring modulation. The Synchronous Overlap Add technique results in a decent sounding attack, with a release that contains two clicks instead of one, and very significant amplitude modulation that affects pitch perception and creates a second pitched tone, audible below the stretched signal. Again, the Overlap Add algorithm has doubled attack and release portions, and salient random amplitude modulation, giving it a trembling, diffuse quality.

### 5.4.3 – Sinusoidal Sweep Results

The stretched sinusoidal sweep sounds best when stretched by the Phase-Locked, Oscillator Bank, and Classic Phase Vocoders. These three algorithms produce perceptually accurate, similar results. However, the Classic Phase Vocoder sounds most accurate. It produces a signal that sounds purely sinusoidal with good attack and release portions, and consistent amplitude throughout. One noticeable error occurs where the swept tone crosses the steady tone, and sounds like a momentary amplitude attenuation. Of the three algorithms, this error seems to be less present in the Classic Phase Vocoder,

and only slightly more present in the other two. The Oscillator Bank and Phase-Locked Vocoders also exhibit additional amplitude differences that manifest as higher amplitude bass frequencies during the first half of the Oscillator Bank Phase Vocoder's output, and subtle amplitude inconsistencies during the second half of the Phase-Locked Vocoder's output. Furthermore, the Phase-Locked Vocoder's attack contains an almost imperceptibly attenuated click, while its release portion contains a more noticeably attenuated click. Next in accuracy, the Peak Tracking Phase Vocoder starts with an accurate attack, sounds ring modulated during its signal portion, contains an attenuation at the frequency crossing, then produces a release with a noticeably louder click. The Synchronous Overlap Add technique creates very similar results, but its ring modulation is significantly more salient, ending with a double click in the release portion. Finally, the Overlap Add technique has softened attack and release clicks, and a less noticeable attenuation at the frequency crossing. However, it sounds randomly amplitude modulated, and this modulation likely hides other errors.

### 5.4.4 – *Amen Break* Results

The Oscillator Bank Phase Vocoder produces the best stretched *Amen Break*, resulting in a signal with transients that sound reasonably crisp, kick drums that seem more coherent, and the least amount of phasiness in the cymbals. The Classic and Phase-Locked Vocoders produce quite similar results, but they sound slightly off in different ways. The Classic Phase Vocoder also produces reasonably crisp transients, but the kick drums sound less coherent, and the cymbals and snares sound phasier. Phasiness is less noticeable, but still audible, in the Phase-Locked Vocoder, and this algorithm's output also contains less coherent kick drums. However, the Phase-Locked Vocoder feels more

accurate in terms of rhythm, possibly attributable to its less noticeable phasiness, resulting in more accurate timbres. The second most accurate result comes from the Synchronous Overlap Add technique, which produces strikingly crisp transients and virtually undetectable phasiness in the upper frequencies. Amplitude modulation is perceptible in its signal, but unlike the ring modulation present in previous Synchronous Overlap Add signals, the amplitude modulation in this signal is more like noise or subtle distortion. The Overlap Add technique creates a signal that is quite similar to the Synchronous Overlap Add technique, but with less accuracy, resulting in transients that are less crisp, and ringing cymbals that seem to warble. The randomized amplitude modulation errors exhibited by the Overlap Add technique in previous examples are absent from this signal, which instead manifest as phasiness in the upper frequencies.

### 5.4.5 – *Autumn in New York* Results

Both the Oscillator Bank and Phase-Locked Vocoders produce the best results with this sample. They have the least amount of phasiness and maintain the pitched content of the signal quite accurately. These signals differ very subtly in the lower register, with the Oscillator Bank sounding a bit fuller from more amplitude in the bass frequencies. The only noticeable artifact is phasiness, which is noticeable during the first word in the signal. This phasiness sounds like a low amplitude, modulated difference tone. A nearly identical, but slightly phasier result is created by the Classic Phase Vocoder. Its phasiness is more noticeable throughout the signal, during loud and quiet segments, as well as pitched and unpitched segments. The next most accurate signal is generated by the Peak Tracking Phase Vocoder. However, it contains a distracting amount of periodic amplitude modulation, and this modulation begins to overwhelm the

pitched content of the signal, making pitches less distinct and slightly inharmonic. The Synchronous Overlap Add signal also exhibits amplitude modulation, but has a different character to its modulation. With this signal, the modulation is less periodic and has slightly greater amplitude, hindering pitch perception and adding a background of noise to the sample. The Overlap Add signal is even noisier due to its randomized amplitude modulation, and this signal begins to sound like a quasi-pitched diffuse whisper.

**5.4.6 – *Peaches en Regalia* Results**

Again, both the Oscillator Bank and Phase-Locked Vocoders produced the best results from this sample, with the least amount of phasiness, most accurate pitches, and good timbral reproduction. The Oscillator Bank Phase Vocoder is slightly more uniform throughout the signal, with minor phasiness, and acceptably crisp percussive transients, while the Phase-Locked Vocoder has a bit less initial phasiness, which becomes more prevalent in the last third of the signal. Additionally, the Phase-Locked Vocoder creates a cleaner sounding piano arpeggio in the middle of the signal, but also creates transients that are mildly less crisp throughout the signal. The Classic Phase Vocoder is nearly as good as the previous two, with crisper transients, but it exhibits more noticeable phasiness, especially in noisy timbres like cymbal crashes. The Peak Tracking Phase Vocoder is the next most accurate, but suffers from significant periodic amplitude modulation. This modulation gives the signal a ring-modulated character, which becomes very noticeable during the piano arpeggio, and later as a difference tone during the second half of the signal. Despite the modulation, pitch perception is still possible with this signal, and the transients are still present. The Synchronous Overlap Add technique creates a signal that is more modulated, with less perceptible pitch, less defined

transients, and more salient difference tones. These difference tones are especially noticeable during the piano arpeggio and last quarter of the signal. Finally, the Overlap Add signal is noisy due to its randomized amplitude modulation, which contributes to inharmonicity in the pitched elements. This signal also contains less crisp transients, but does not contain any pitched difference tones. Instead it contains unpitched and noisy components that contribute to a sense of diffusion.

## 6 – Summary

The analysis techniques developed in this research provide insight regarding the behavior of time stretching algorithms, and this insight is directly related to aural perception of the algorithms' output. Error graphs arising from these techniques are related to perceptual observations, and may be interpreted as analytical descriptions of how the time stretched signals differ from their ideal counterparts. Each analysis technique provides a unique class of data useful for describing the behavior of signals from different perspectives.

Average Spectrum analysis represents the general behavior of time stretching algorithms, showing overall accuracy in the frequency domain by averaging data from all analysis frames together. This illustrates prevalent recurring errors within a signal as a single representative spectrum. The characteristics described by this type of analysis include the accuracy of phase manipulations during the analysis and resynthesis stages of spectral algorithms, as well as the distribution of energy among their bins. Many synthetic signal error curves generated during Average Spectrum analysis display skirts of error surrounding spectral peaks, as well as spectral peaks that are slightly offset from their corresponding ideal peaks' locations. In addition, many of these peaks are of different magnitude compared to their ideal peaks. These differences indicate spectral energy leaking into adjacent bins, errors during instantaneous frequency calculations, and improperly scaled sinusoids during resynthesis. Each error has a characteristic audible sound, and these sounds are represented within the data. Furthermore, Average Spectrum analysis illustrates other recurring magnitude errors that arise in both families of time stretching algorithms, such as periodic window function modulation, or phase

cancellation arising from poorly overlapped grains. Again, these audible artifacts appear in the analysis data error graphs as unique features representing the artifacts. Specific detailed interpretations of Average Spectrum error graphs are contained in Chapter 5, Section 5.1.

Moving Spectral Average analysis generates information about the behavior of stretched signals over time. This technique creates graphs displaying average spectral magnitude error for each analysis frame, providing insight regarding a time stretching algorithm's behavior as it processes a signal through time. Such analysis is useful for inspecting significant time-domain features, like transients or amplitude changes, in order to evaluate the temporal accuracy of each algorithm. Since the graphs generated by this technique represent error over time, features like transients create error curves that show how the various algorithms generate echoes, or otherwise smear information that should ideally be instantaneous. The portions of the graphs representing these stretched transients correspond directly to their audible locations, lengths, and amplitudes. Errors that persist in time are also visible, including overlapping window function modulation, sloppily shifted grains, and poorly chosen crossfade points. These visible errors directly represent audible differences between the stretched and ideal signals. Detailed discussion and interpretation of Moving Spectral Average analysis is contained in Chapter 5, Section 5.2.

Error Spectrogram analysis displays a two dimensional view of spectral error over time, incorporating both time-domain and frequency-domain information. This technique illustrates spectral magnitude errors for each analysis frame in order to provide a comprehensive view of each algorithm's output. The location of an error over time, as

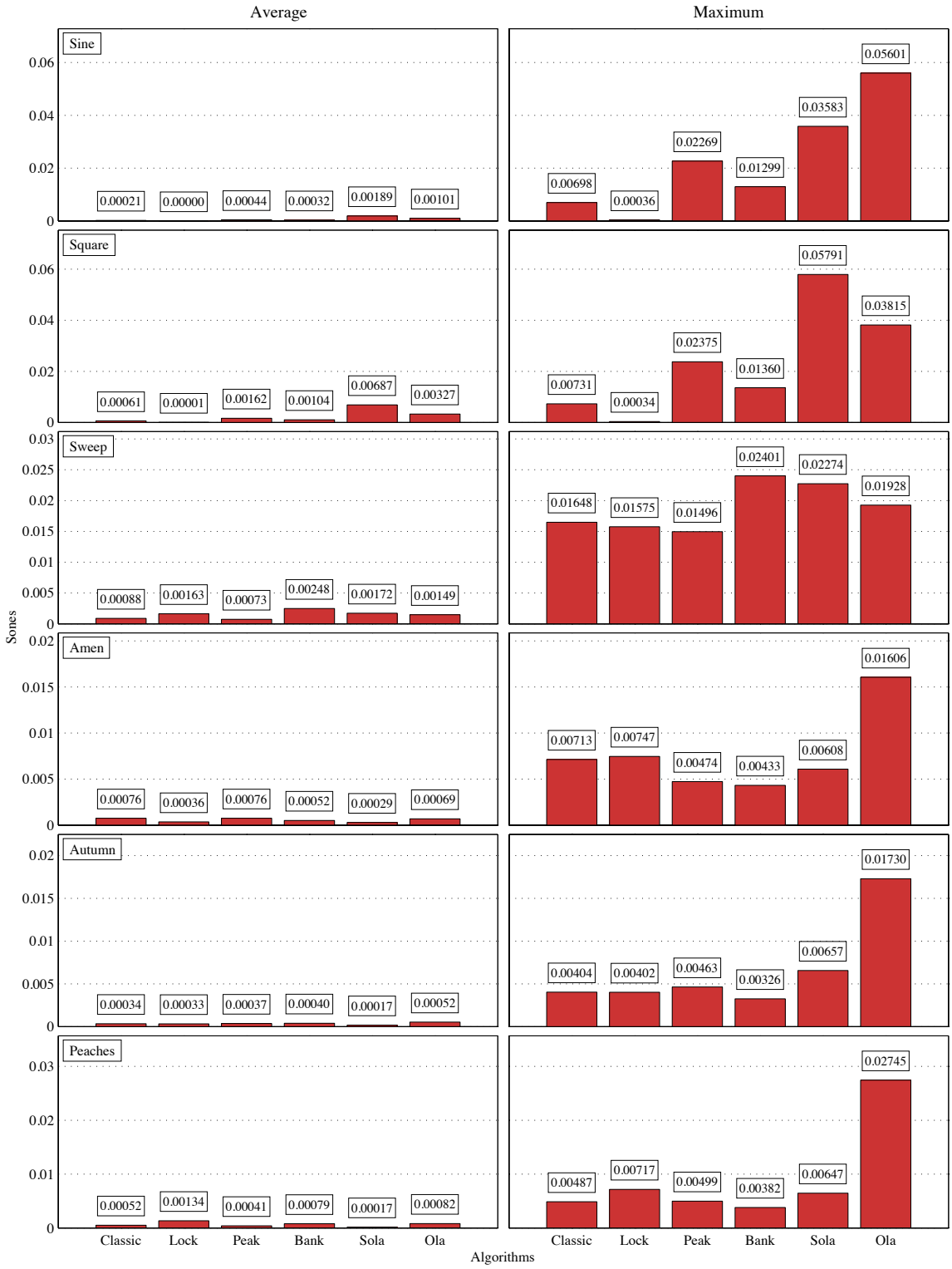
well as its location within the frequency spectrum is represented in the graphs, and these errors are directly related to aural perception of artifacts in the time stretched signals. Any differences between the ideal and stretched signals are visible, including errors such as inaccurate transients, window function modulation, or phase cancellation. Error Spectrogram graphs display these features where they correspond in time to their audible counterparts. Detailed interpretation of the Error Spectrum analysis results is discussed in Chapter 5, Section 5.3.

The next few subsections contain summaries of the time stretching algorithms, with graphs providing an overall idea of accuracy among the algorithms in terms of the signals they operate upon. The graphs contain the average and maximum error values for every signal generated by each algorithm, and each set of values is displayed with the same scaling to provide a visual relationship between the values. The legend text in the graphs corresponds to the different signals, and the X axis labels correspond to the different algorithms: “Classic” is the Classic Phase Vocoder, “Lock” is the Phase-Locked Vocoder, “Peak” is the Peak Tracking Phase Vocoder, “Bank” is the Oscillator Bank Phase Vocoder, “Sola” is the Synchronous Overlap Add technique, and “Ola” is the Overlap Add technique. The boxes above the bars display specific values for each signal and algorithm combination.

### 6.1 – Average Spectrum

Graph 6.1 shows the Average Spectrum error summary for all algorithms and signals. In general, the phase vocoders produce less error than the granular techniques, with the Phase-Locked Vocoder performing best in most cases, followed closely by the Classic and Oscillator Bank Phase Vocoders. The Overlap Add algorithm consistently produces the most error. The MATLAB script that generated this graph is contained in Appendix B, Section 8.4.13.

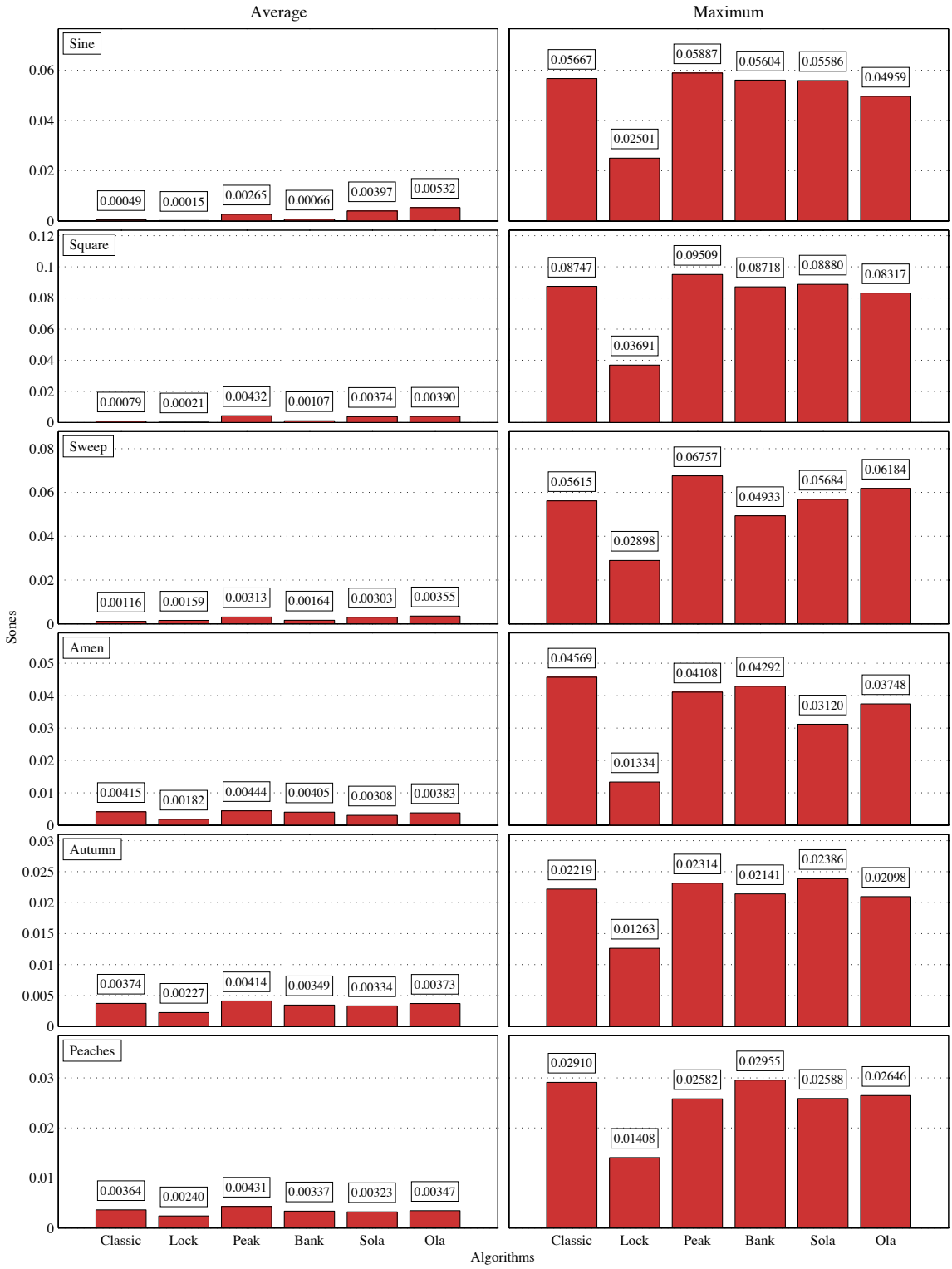
**Graph 6.1 –** Average Spectrum Summary



### 6.2 – Moving Spectral Average

The Moving Spectral Average error summary for all algorithms and signals is shown in Graph 6.2. This graph shows that the Phase-Locked Vocoder exhibits slightly less average error, and noticeably less maximum error than the other algorithms. The Classic and Oscillator Bank Phase Vocoders display similar values to each other, with slightly more error created by the Oscillator Bank Phase Vocoder. The remaining three algorithms produce comparatively greater errors, but the values of these errors vary inconsistently, and appear to be signal dependent. The MATLAB script that generated this graph is contained in Appendix B, Section 8.4.14.

**Graph 6.2 –** Moving Spectral Average Summary



### 6.3 – Subjective Listening

Results from subjective listening tests differ slightly from the previous two graphs and are summarized as follows. The sine and square waves sound best when stretched by the Oscillator Bank Phase Vocoder. These signals also sound nearly as good through the Phase-Locked and Classic Phase Vocoders, but their attacks and releases are slightly different from the ideal signals. The swept sinusoid sounds best when stretched by the Classic Phase Vocoder, but again, this signal also sounds nearly as accurate from the Oscillator Bank and Phase-Locked Vocoders. The *Amen Break* sounds best through the Oscillator Bank Phase Vocoder, but is also acceptably accurate when manipulated by the Synchronous Overlap Add technique. When *Autumn in New York* is stretched, both the Oscillator Bank and Phase-Locked Vocoders perform best. Finally, the *Peaches en Regalia* signal sounds best through the Oscillator Bank Phase Vocoder.

### 6.4 – General Ranking

Since each algorithm exhibits different strengths depending on the type of signal being processed, the data generated from these processed signals allows categorization of the algorithms according to their strengths in terms of signal type.

The Phase-Locked Vocoder works well with steady sustained signals containing sinusoidal components (i.e. sine and square waves in this research), and is closely followed by the Classic and Oscillator Bank Phase Vocoders. The Phase-Locked Vocoder's treatment of phase ensures that it accurately reproduces steady sinusoids by locking neighboring bins together and minimizing phase errors. The sine and square wave test signals have relatively sparse spectra, and the Phase-Locked Vocoder's locking scheme has no chance of accidentally locking incorrect phases from unrelated partials of

other sounds between adjacent bins. Spectral sparsity also aids accuracy in the Classic and Oscillator Bank Phase Vocoders by minimizing the amount of unrelated energy in neighboring bins during resynthesis. The Peak Tracking Phase Vocoder, Synchronous Overlap Add, and Overlap Add techniques fare poorly with steady sinusoidal signals, mainly due to repetitive time-domain amplitude errors arising from poorly implemented windowing schemes. These errors are most noticeable in the Overlap Add technique, but are also quite prevalent in the other two techniques, where they manifest as a kind of amplitude modulation at the period of window overlap.

The Classic Phase Vocoder works best with the swept sinusoid signal, followed closely by the Oscillator Bank Phase Vocoder, and more distantly by the Phase-Locked Vocoder. Both the Classic and Oscillator Bank algorithms do nothing special with phase, which is an advantage due to the nature of the signal. The swept sinusoid signal contains a fixed 1000 Hertz sinusoid mixed with a logarithmically ascending sinusoid, making phase locking and peak tracking difficult due to unpredictable frequency changes and phase procession. Furthermore, the signal contains no points of self-similarity, defeating any cross-correlation attempted by the Synchronous Overlap Add algorithm. Again, the Peak Tracking Phase Vocoder and both Overlap Add techniques suffer from repetitive windowing errors with this signal.

The Phase-Locked Vocoder is most accurate with the *Amen Break* signal, and the Synchronous Overlap Add method provides second best results. Phase locking may mitigate phasiness among inharmonic partials of percussion and ringing cymbals, but it is unclear if this is the main reason for its accuracy. The Synchronous Overlap Add scheme seems to excel with percussive material, finding well chosen crossfade points based on its

grain by grain cross-correlation data. The Classic and Oscillator Bank Phase Vocoders are less accurate, due to phase errors in resynthesis, arising from the cymbals' dense inharmonic partials, which overlap among adjacent bins. Peak tracking is of little use with percussive signals, as there are relatively few sustained spectral peaks. This algorithm mainly introduces amplitude modulation error, which is apparent in the upper parts of the signal's spectrum. Similarly, the Overlap Add technique introduces phasiness to the output signal as it randomizes input grain locations and inadvertently causes phase cancellations in the upper partials corresponding to the ringing cymbals.

With the *Autumn in New York* and *Peaches en Regalia* signals, the Classic, Oscillator Bank, and Phase-Locked Vocoders all perform best with very similar accuracy. The Phase-Locked Vocoder is slightly better with *Autumn in New York*, likely due to the signal's sustained vocal harmonies, which consist of relatively long and somewhat slowly changing partials. On the other hand, the Classic and Oscillator Bank Phase Vocoders are slightly more accurate with the *Peaches en Regalia* signal, most likely due to their simplicity when dealing with such dense phase information. The remaining three algorithms all perform somewhat less accurately with these signals, most notably introducing unacceptable repetitive or randomized amplitude modulation artifacts into the signals.

**6.5 – Recommendations**

Based on subjective listening results, the Oscillator Bank Phase Vocoder generally produces the most perceptually accurate time stretched signals. However, certain algorithms seem more accurate depending on signal type, with the Classic Phase Vocoder handling the swept sinusoid well, and the Synchronous Overlap Add technique

producing a nicely stretched *Amen Break*. Based on the Moving Spectral Average graph, the Phase-Locked Vocoder is clearly the most accurate algorithm. However, the Average Spectrum graph also indicates that the Classic, Oscillator Bank, and Phase-Locked Vocoders all exhibit good accuracy.

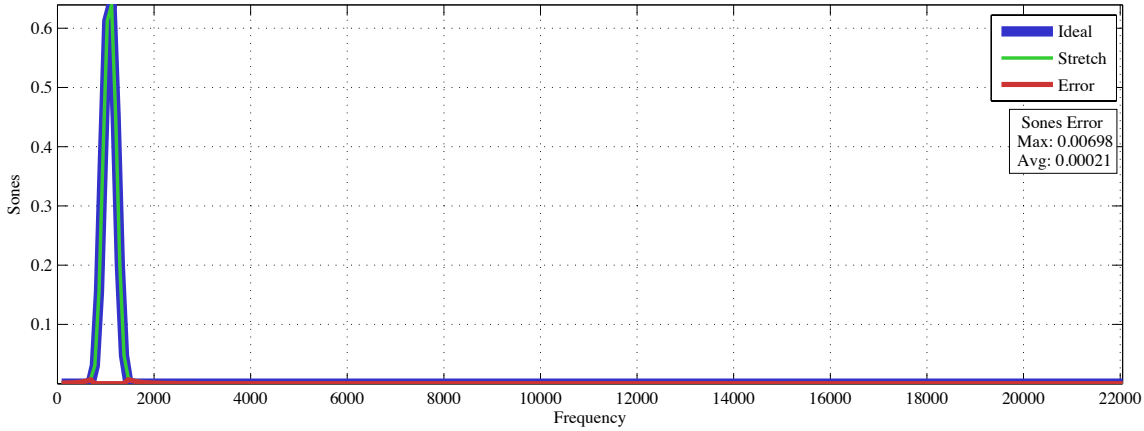
Based on mathematical analysis results, the Oscillator Bank Phase Vocoder is a good starting point for general-purpose time stretching. In cases where an accurate steady-state signal is desired, the Phase-Locked Vocoder should be the first choice, and for sinusoidal signals with changing frequencies, the Classic Phase Vocoder is a good starting point. The Oscillator Bank Phase Vocoder is good for unpitched rhythmic signals containing percussive transients, with the second choice being the Synchronous Overlap Add technique. Pitched, non-percussive rhythmic signals are best stretched by the Oscillator Bank or Phase-Locked Vocoders. Finally, complex signals with pitched components, transients, and non-harmonic elements are most accurately stretched by the Oscillator Bank Phase Vocoder.

# 7 – Appendix A

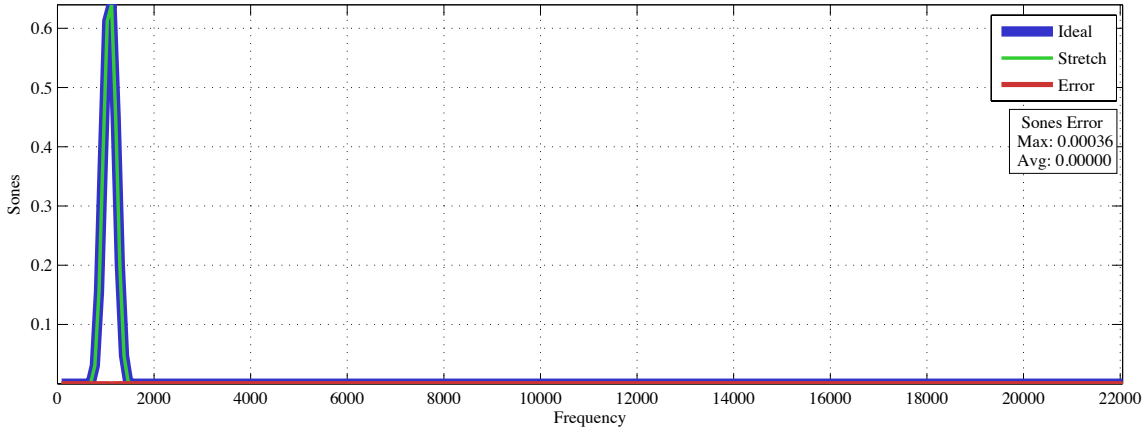
## 7.1 – Average Spectrum Graphs

### 7.1.1 – Sine Average Spectrum Graphs

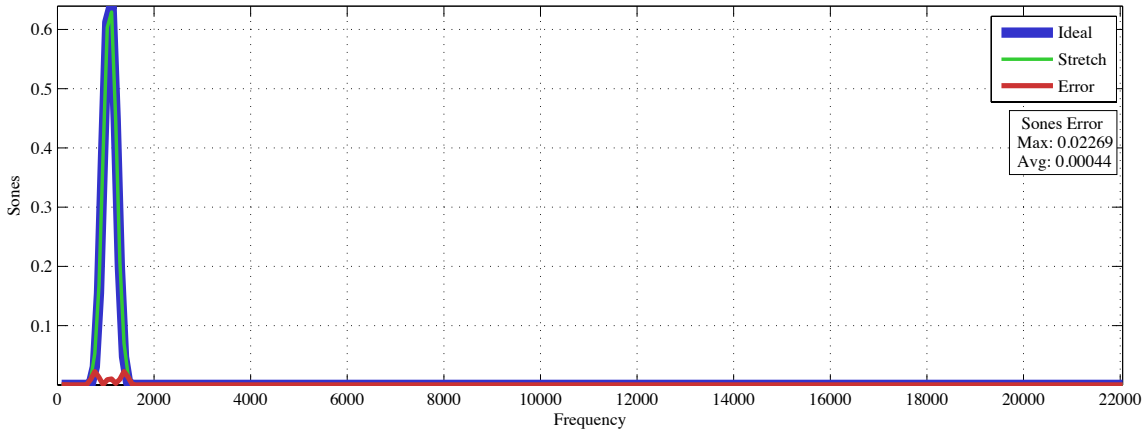
**Graph 7.1.1.1 –** Sine Classic Average Spectrum



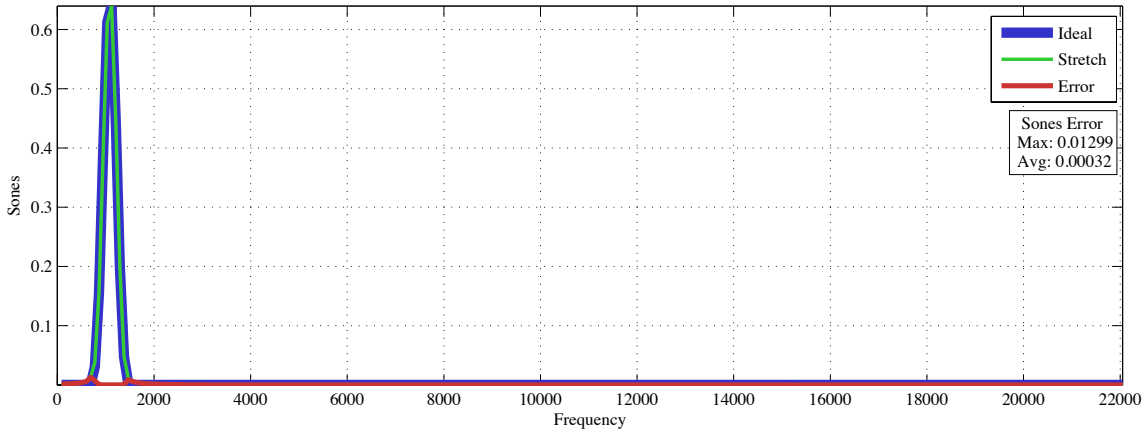

**Graph 7.1.1.2 –** Sine Lock Average Spectrum



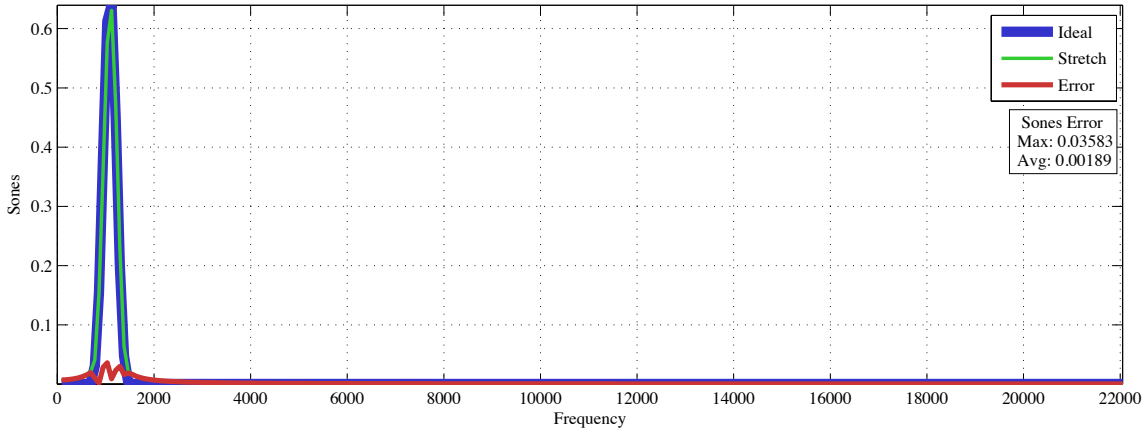
**Graph 7.1.1.3 –** Sine Peak Average Spectrum



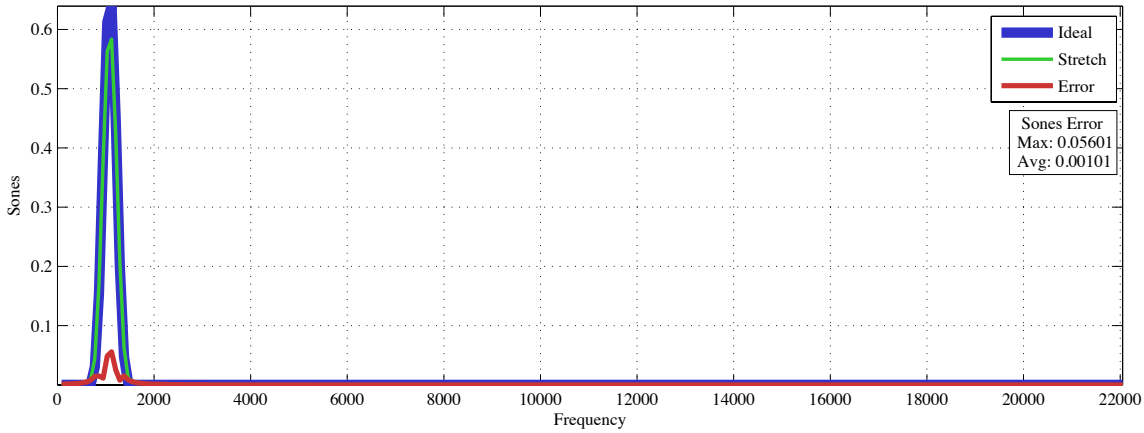

**Graph 7.1.1.4 –** Sine Bank Average Spectrum




**Graph 7.1.1.5 –** Sine Sola Average Spectrum

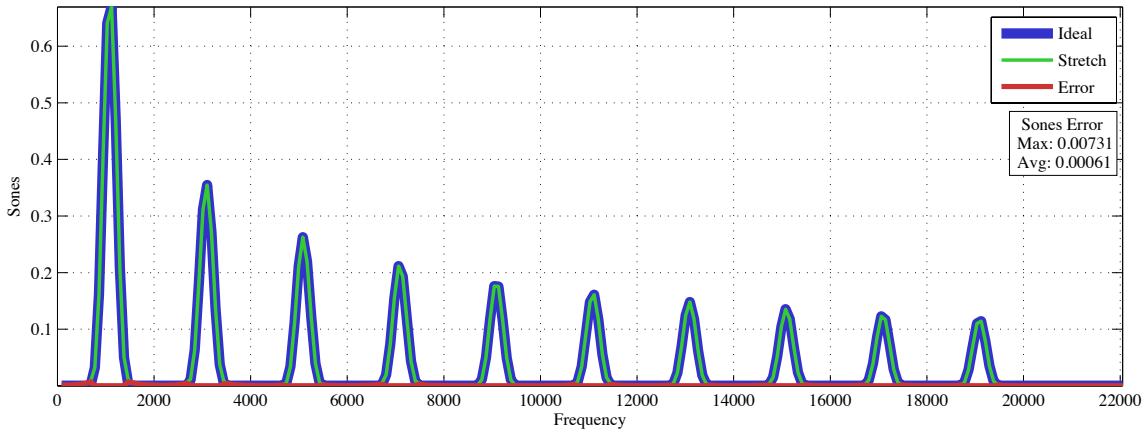


**Graph 7.1.1.6 –** Sine Ola Average Spectrum

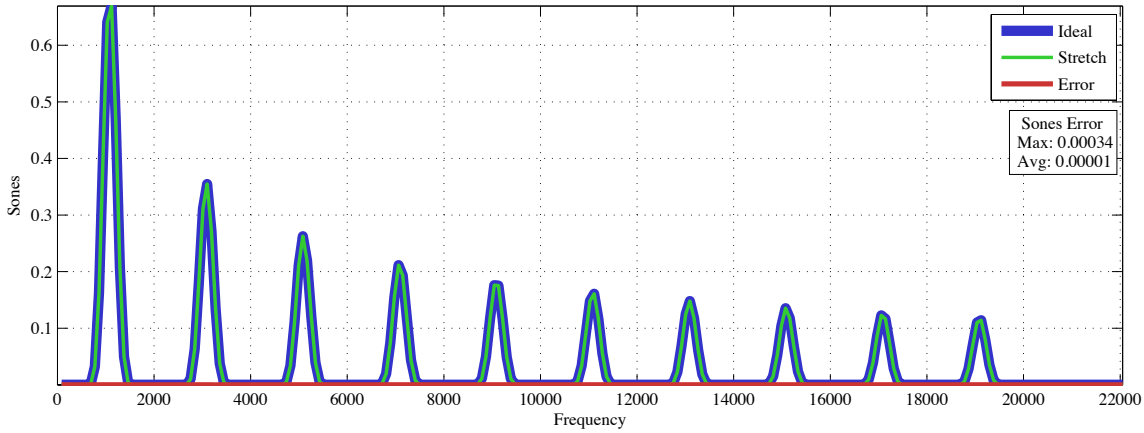



## 7.1.2 – Square Average Spectrum Graphs

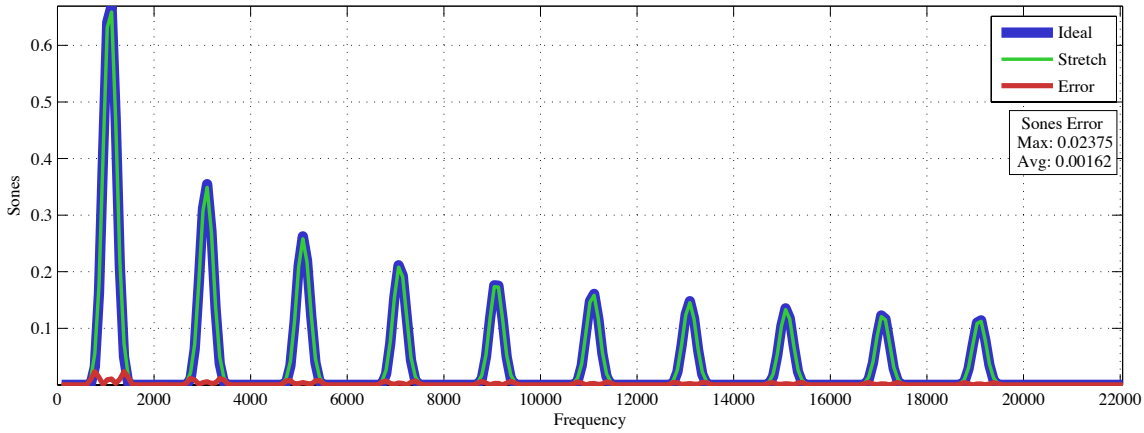
**Graph 7.1.2.1 –** Square Classic Average Spectrum



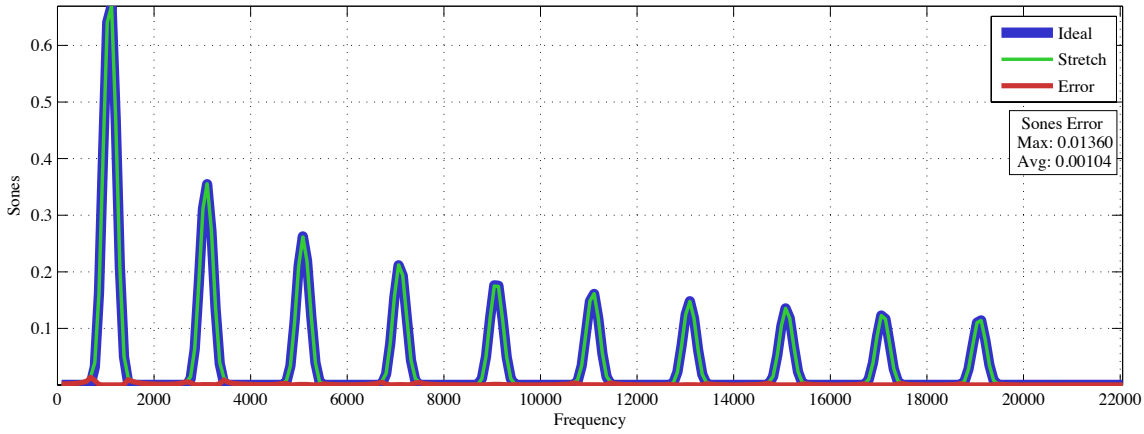

**Graph 7.1.2.2 –** Square Lock Average Spectrum



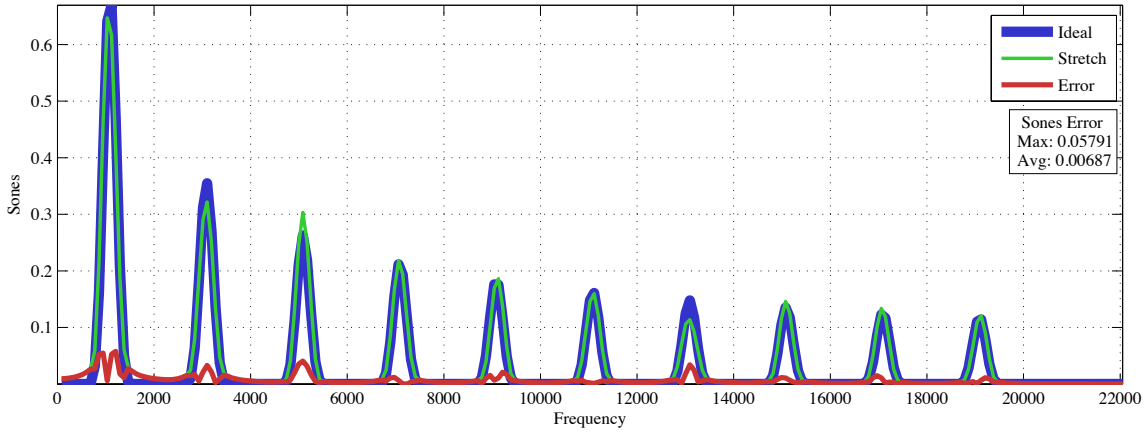
**Graph 7.1.2.3 –** Square Peak Average Spectrum



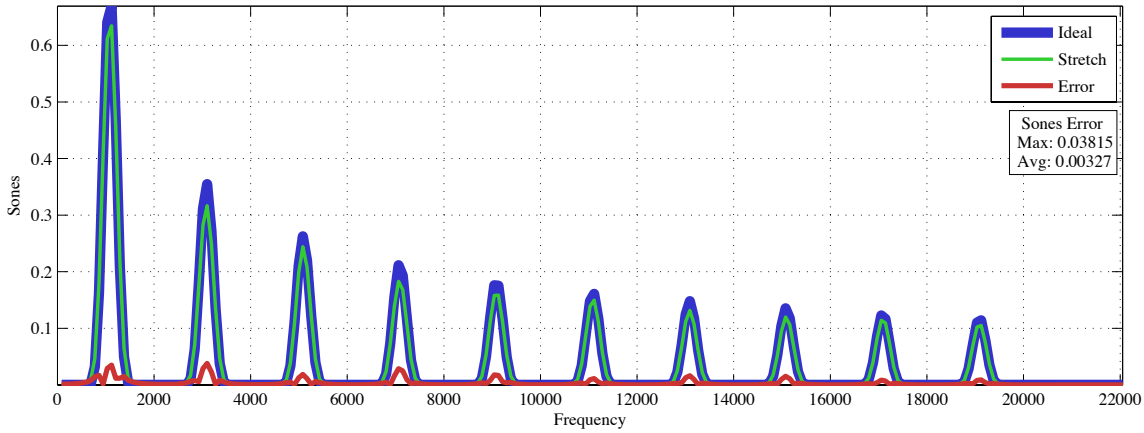

**Graph 7.1.2.4 –** Square Bank Average Spectrum




**Graph 7.1.2.5 –** Square Sola Average Spectrum

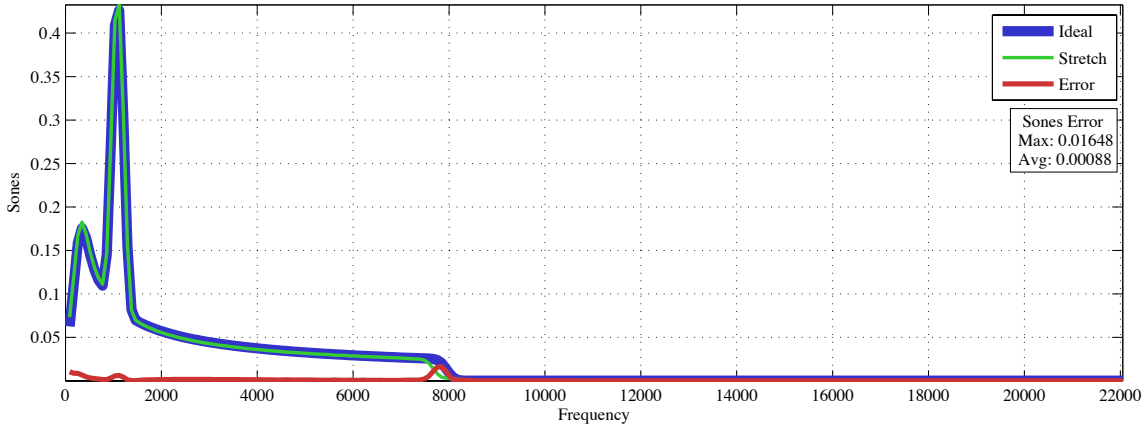


**Graph 7.1.2.6 –** Square Ola Average Spectrum

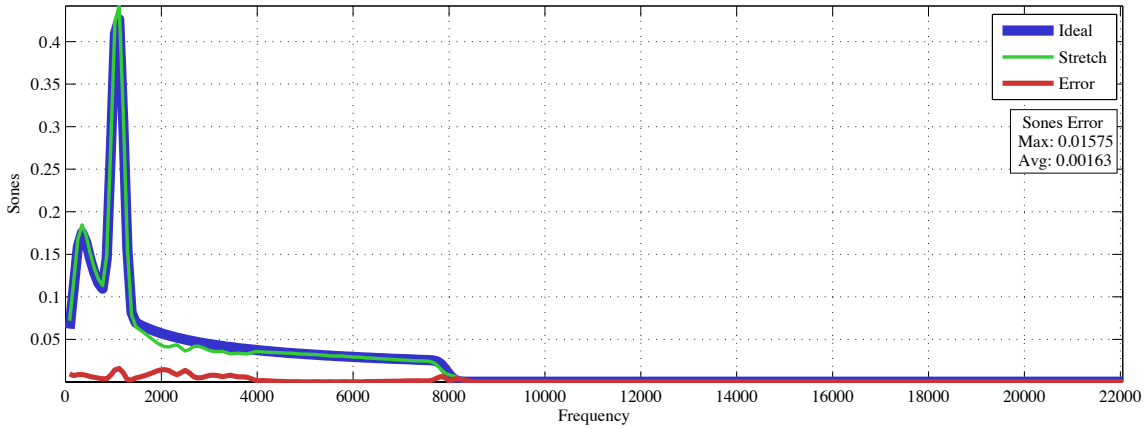



## 7.1.3 – Sweep Average Spectrum Graphs

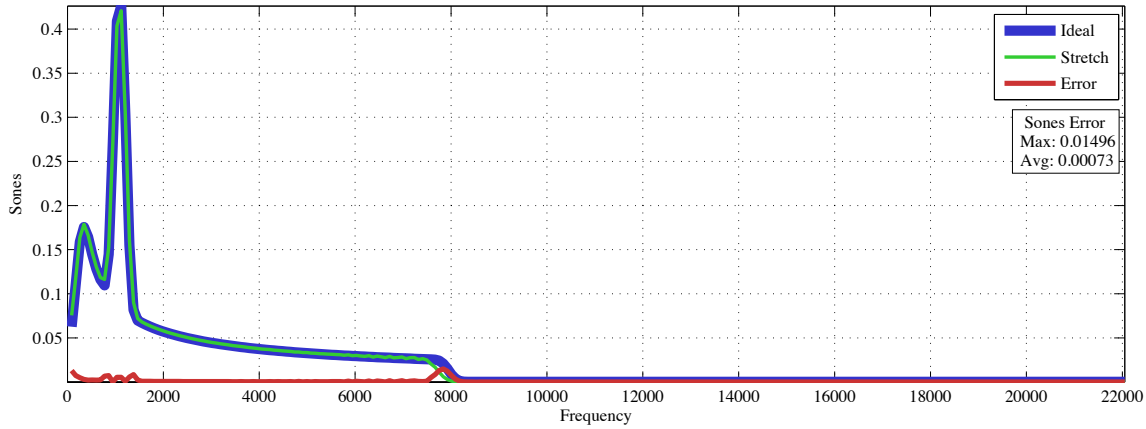
**Graph 7.1.3.1 –** Sweep Classic Average Spectrum



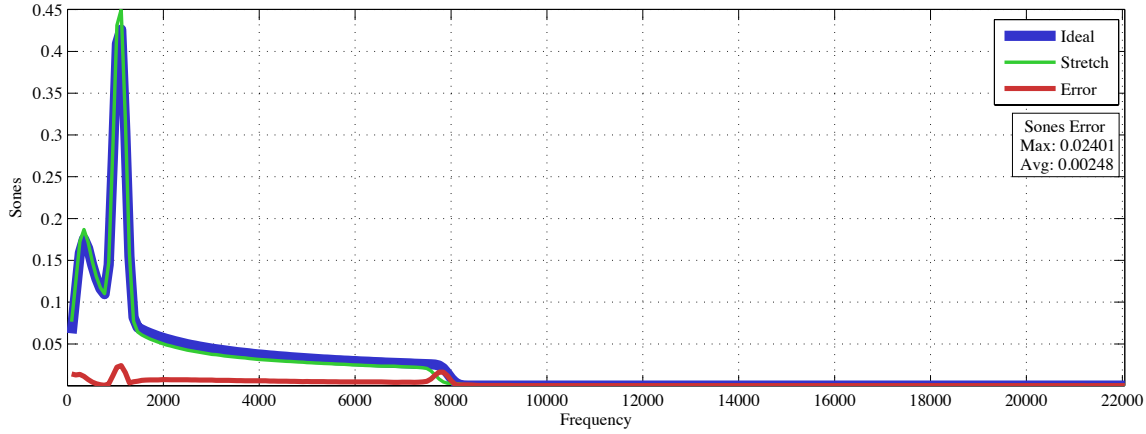

**Graph 7.1.3.2 –** Sweep Lock Average Spectrum



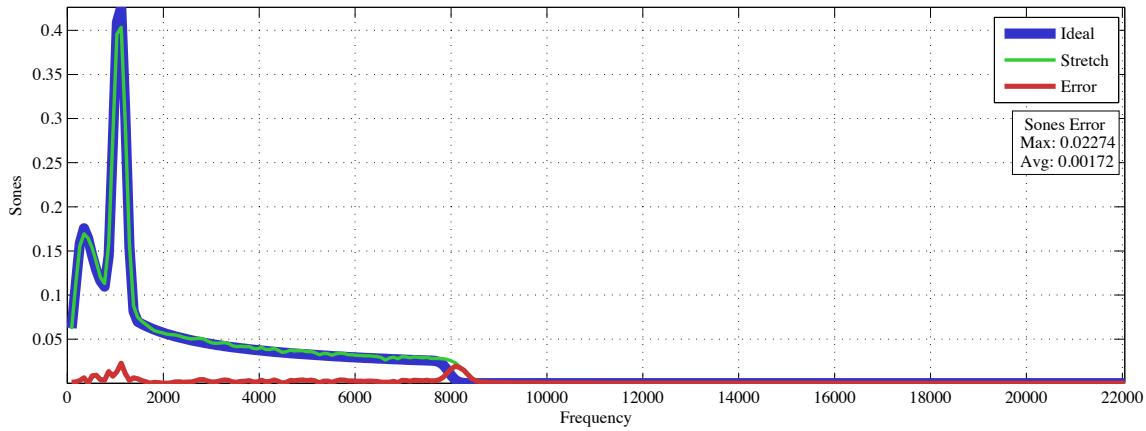
**Graph 7.1.3.3 –** Sweep Peak Average Spectrum



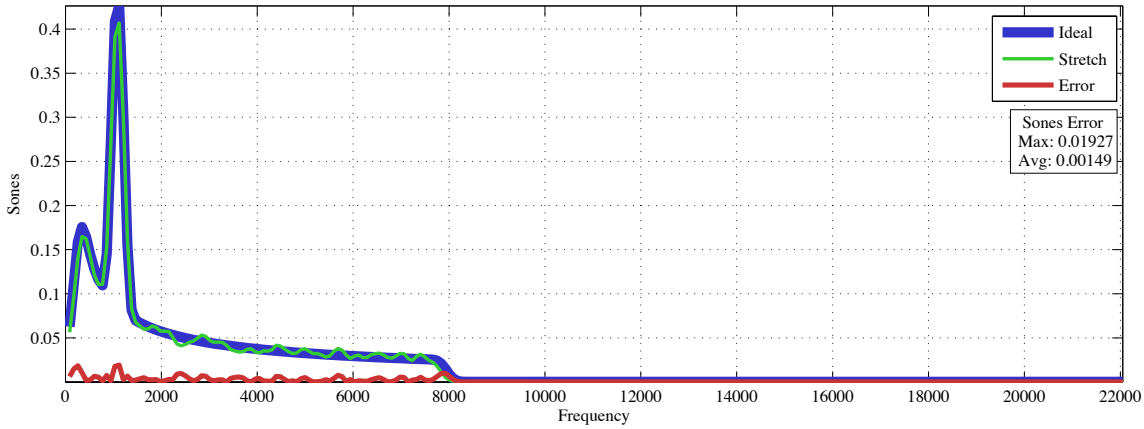

**Graph 7.1.3.4 –** Sweep Bank Average Spectrum




**Graph 7.1.3.5 –** Sweep Sola Average Spectrum

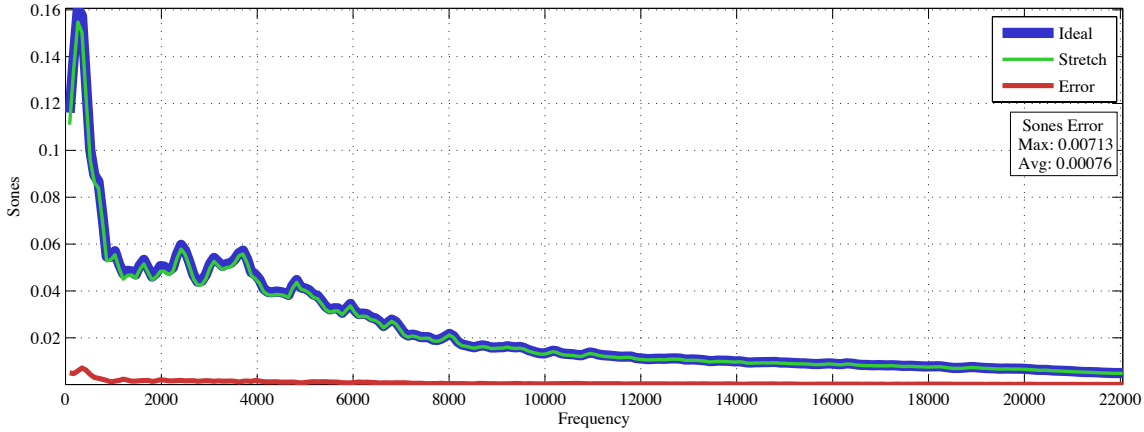


**Graph 7.1.3.6 –** Sweep Ola Average Spectrum

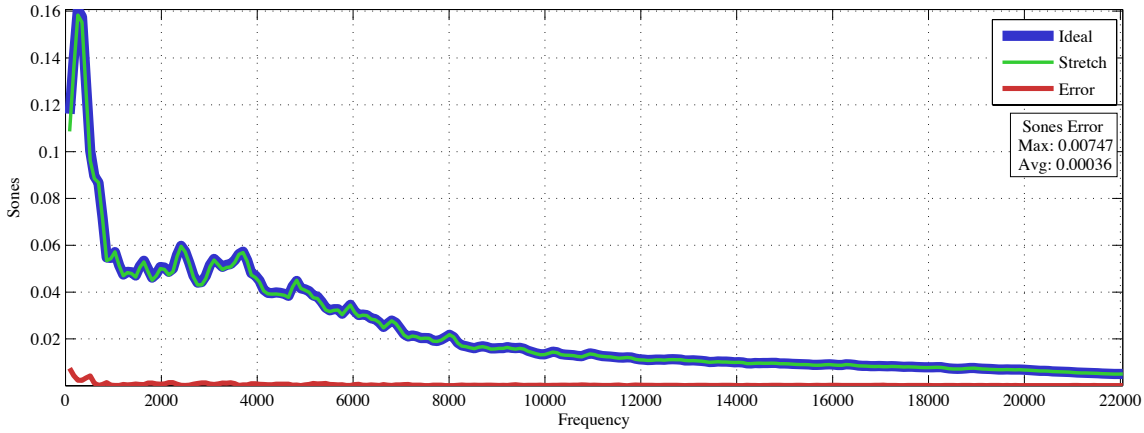



## 7.1.4 – *Amen* Average Spectrum Graphs

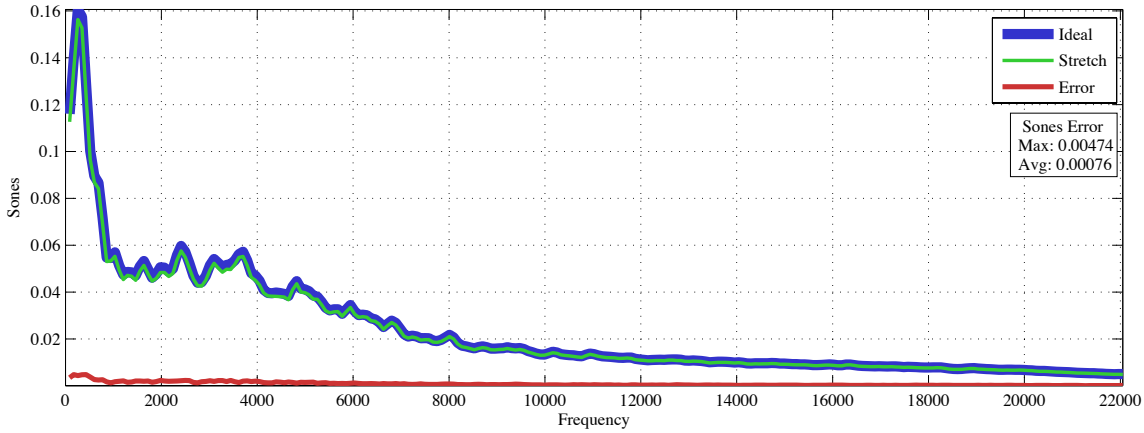
**Graph 7.1.4.1 –** *Amen* Classic Average Spectrum



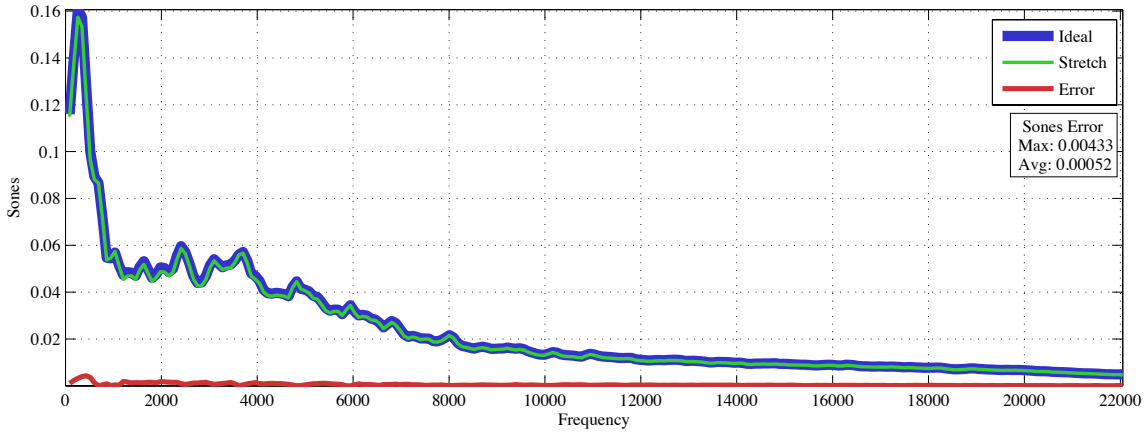

**Graph 7.1.4.2 –** *Amen* Lock Average Spectrum



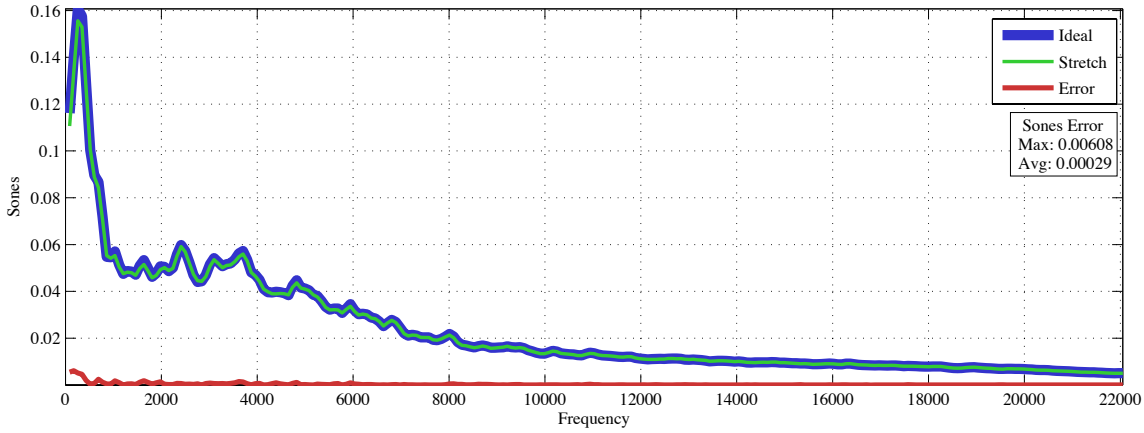
**Graph 7.1.4.3 –** *Amen* Peak Average Spectrum



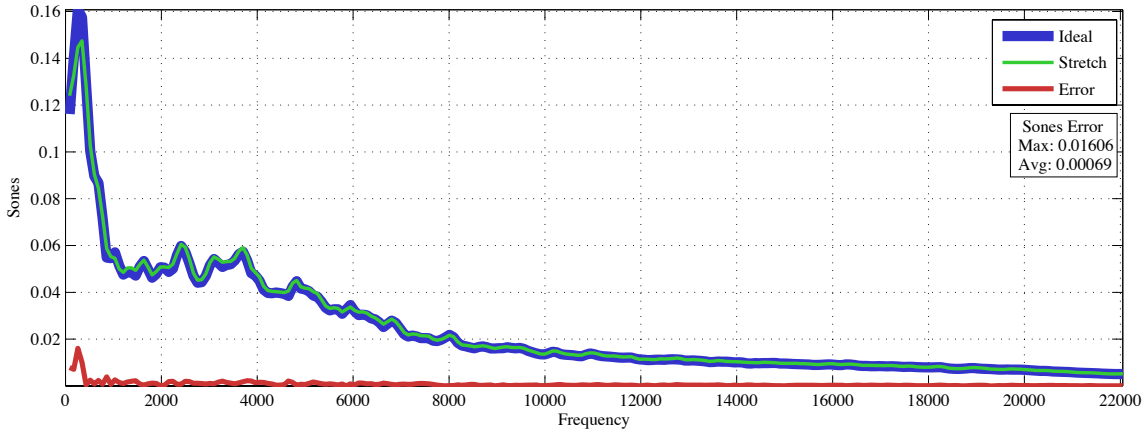

**Graph 7.1.4.4 –** *Amen* Bank Average Spectrum




**Graph 7.1.4.5 –** *Amen* Sola Average Spectrum

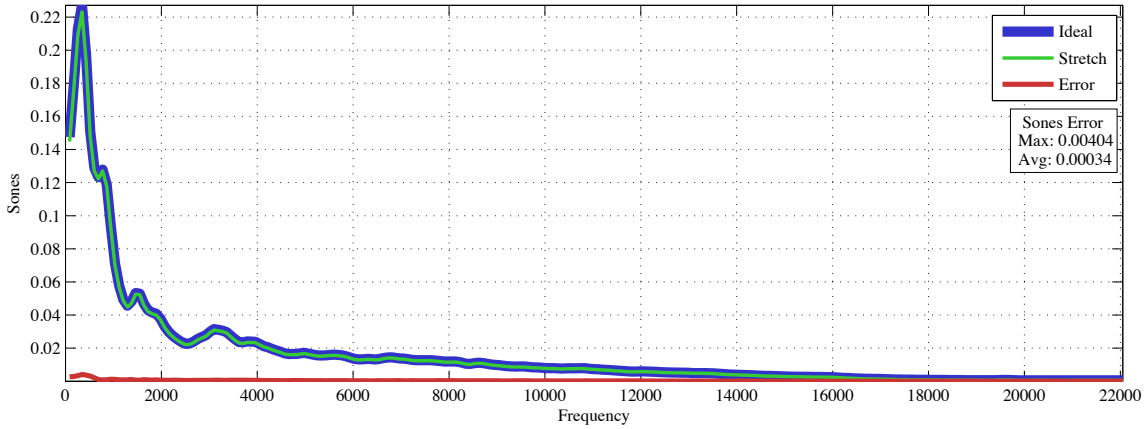


**Graph 7.1.4.6 –** *Amen* Ola Average Spectrum

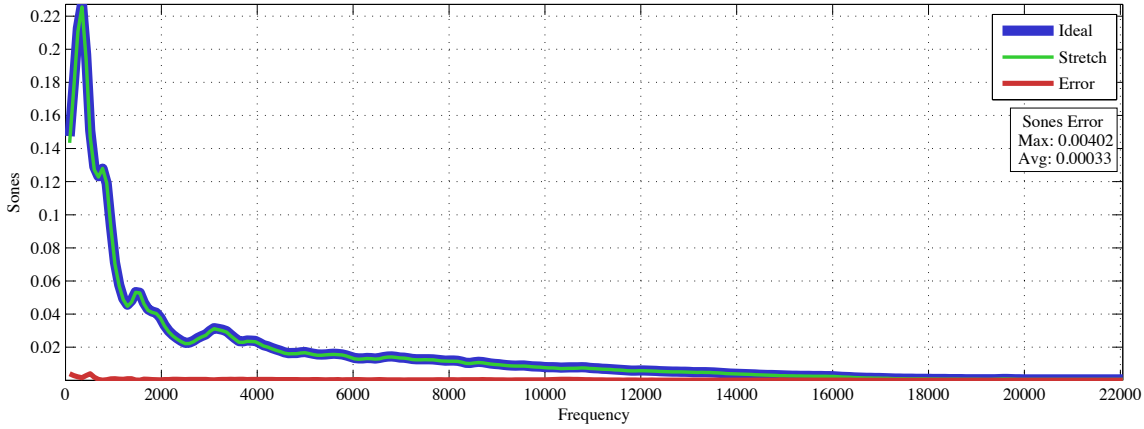



## 7.1.5 – *Autumn* Average Spectrum Graphs

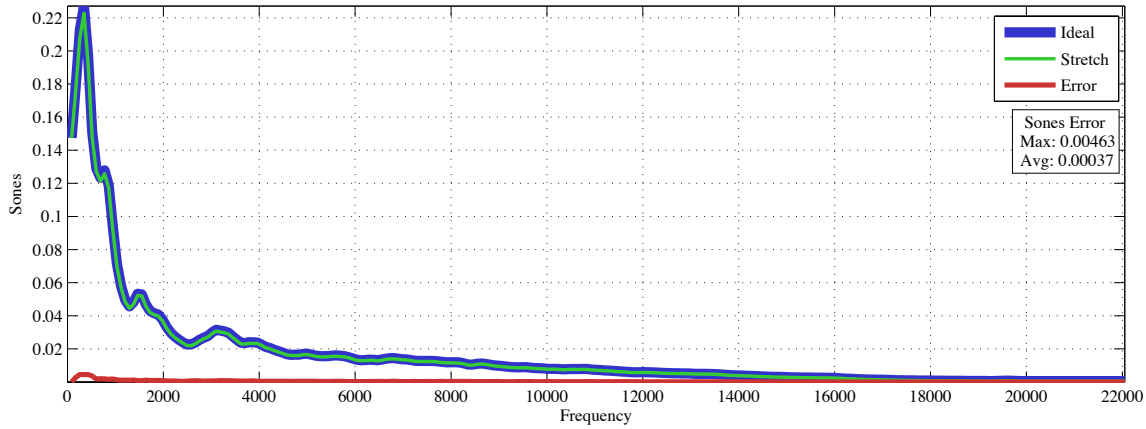
**Graph 7.1.5.1 –** *Autumn* Classic Average Spectrum



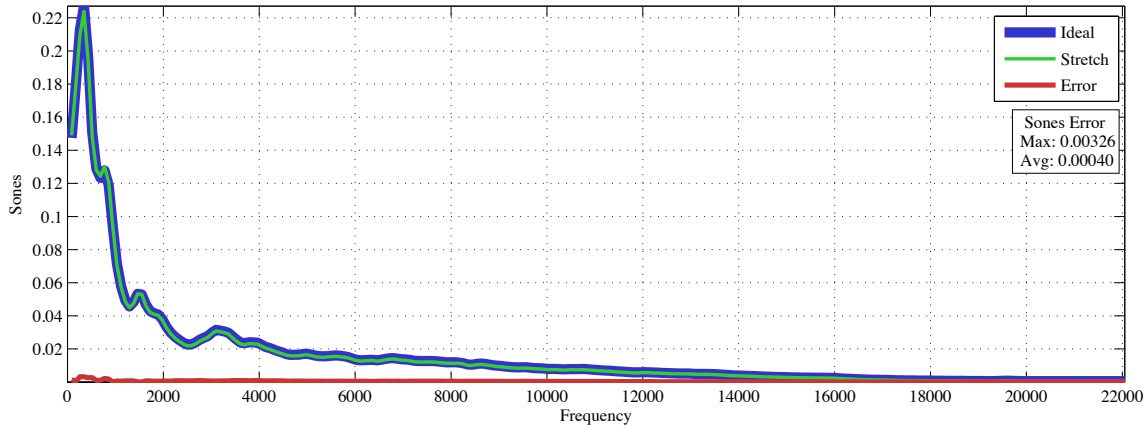

**Graph 7.1.5.2 –** *Autumn* Lock Average Spectrum



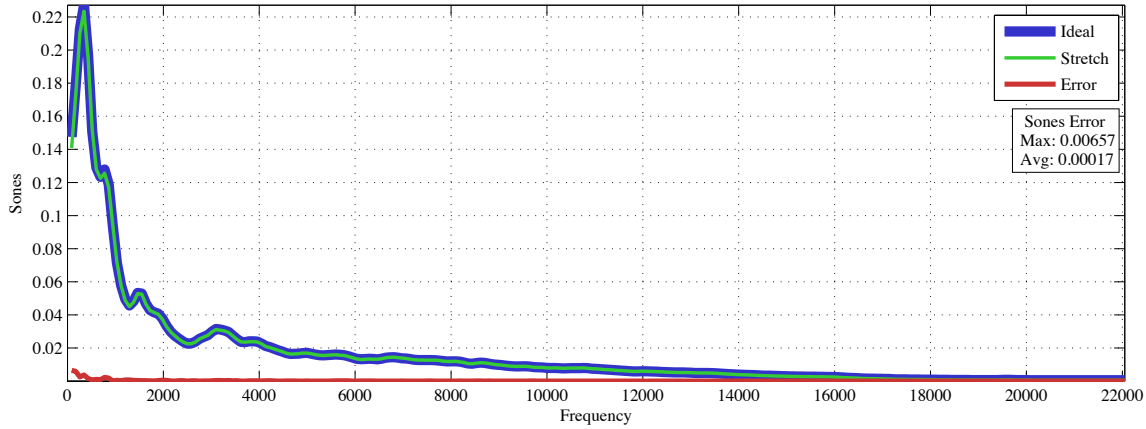
**Graph 7.1.5.3 –** *Autumn* Peak Average Spectrum



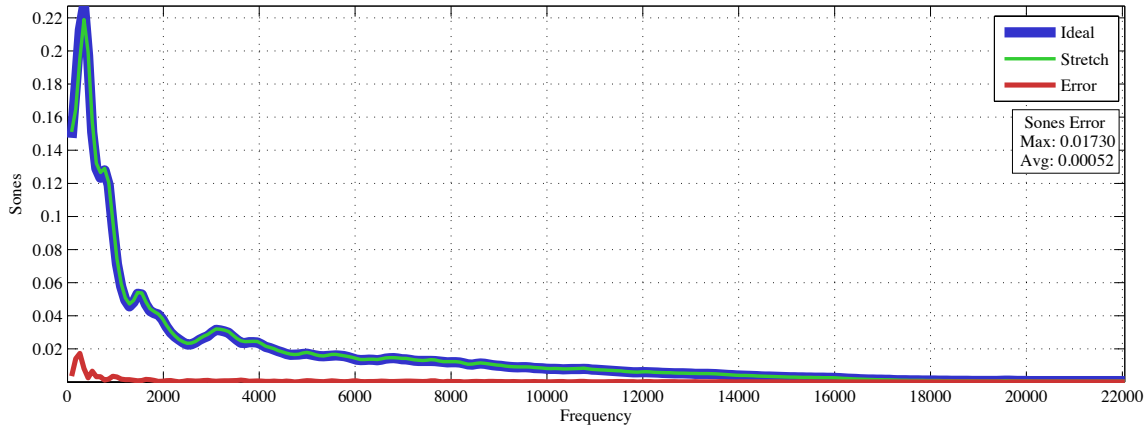

**Graph 7.1.5.4 –** *Autumn* Bank Average Spectrum




**Graph 7.1.5.5 –** *Autumn* Sola Average Spectrum

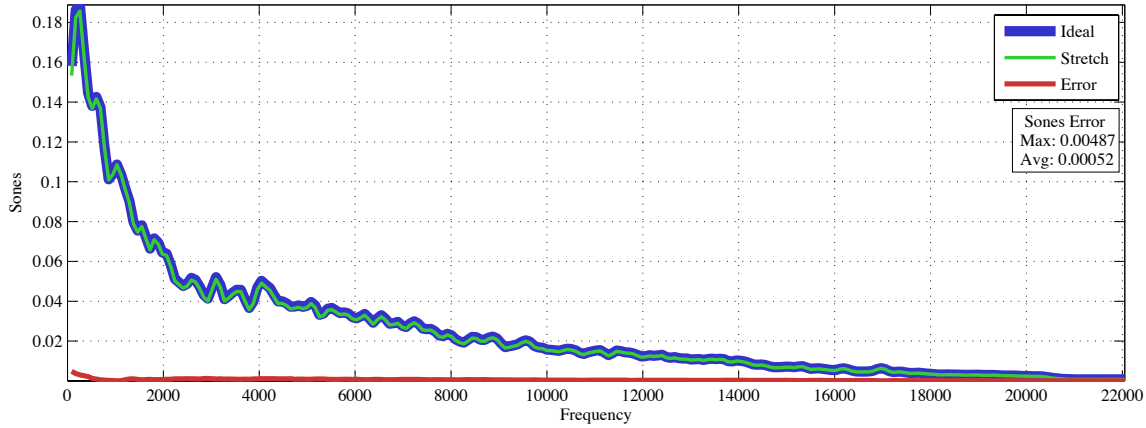


**Graph 7.1.5.6 –** *Autumn* Ola Average Spectrum

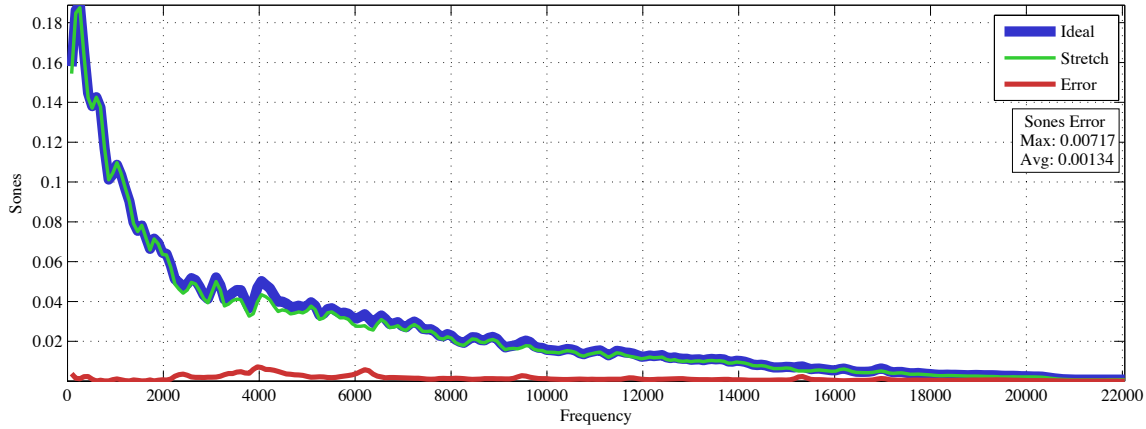



## 7.1.6 – *Peaches* Average Spectrum Graphs

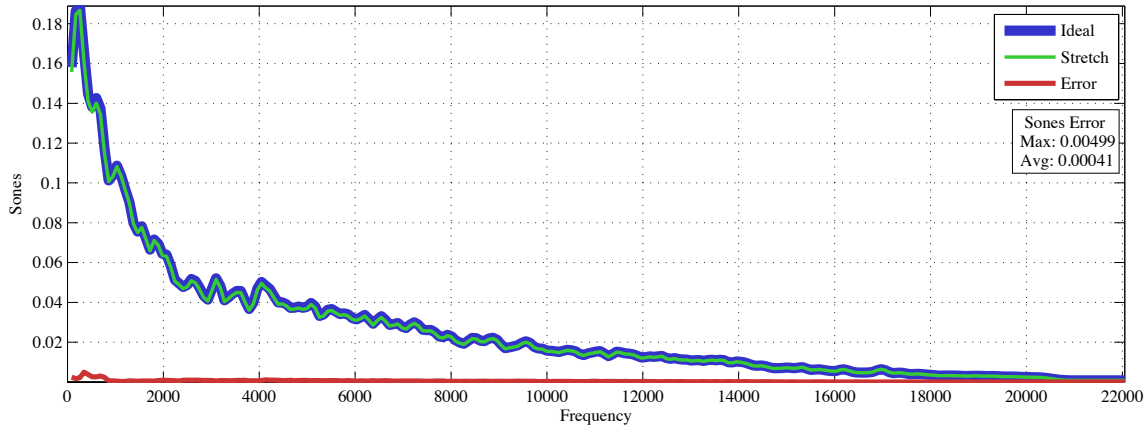
**Graph 7.1.6.1 –** Peaches Classic Average Spectrum



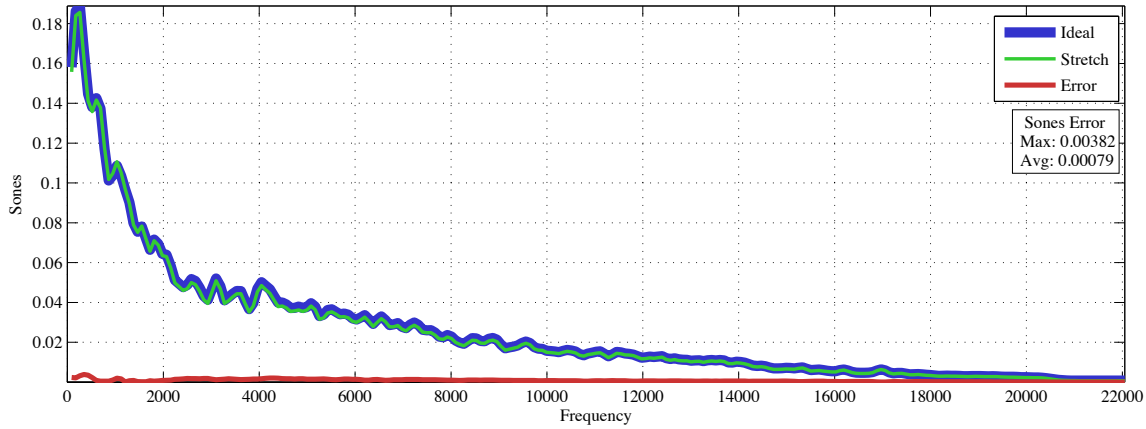

**Graph 7.1.6.2 –** *Peaches* Lock Average Spectrum



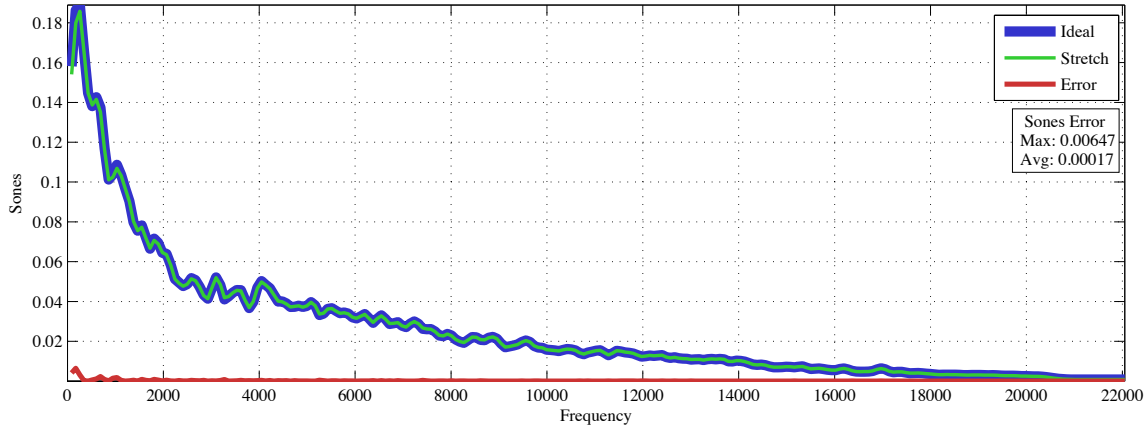
**Graph 7.1.6.3 –** *Peaches* Peak Average Spectrum



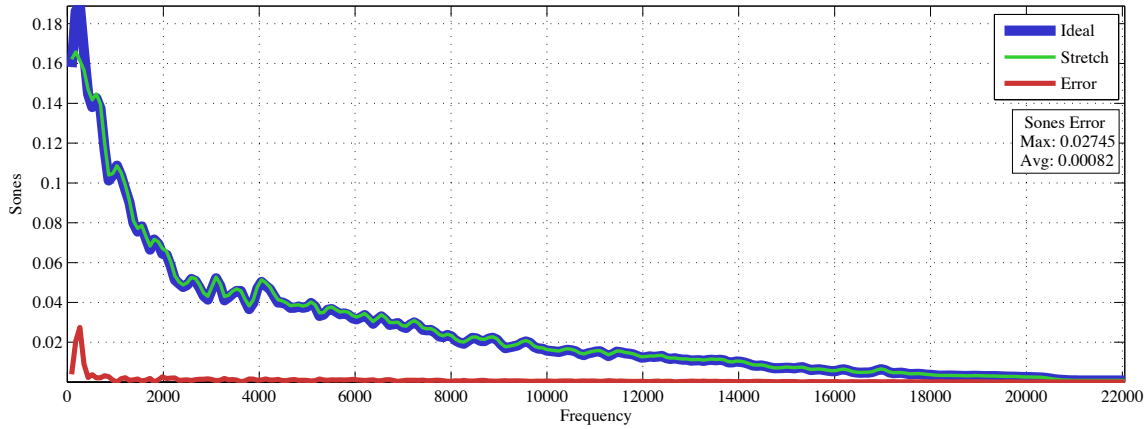

**Graph 7.1.6.4 –** *Peaches* Bank Average Spectrum




**Graph 7.1.6.5 –** *Peaches* Sola Average Spectrum



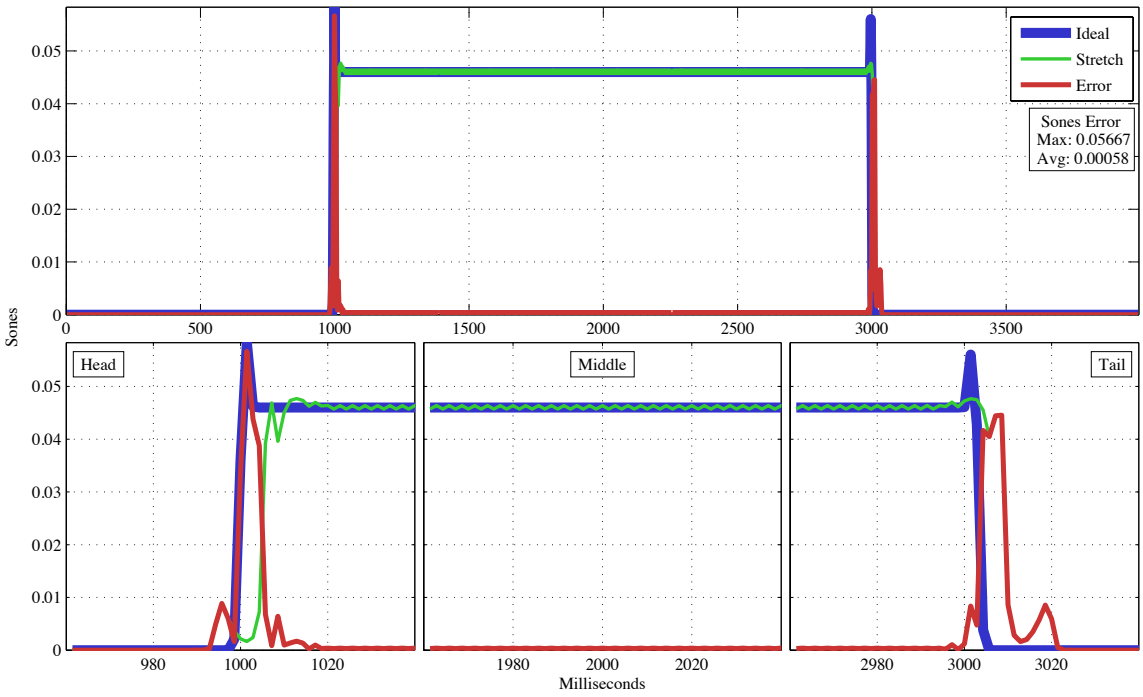
**Graph 7.1.6.6 –** *Peaches* Ola Average Spectrum



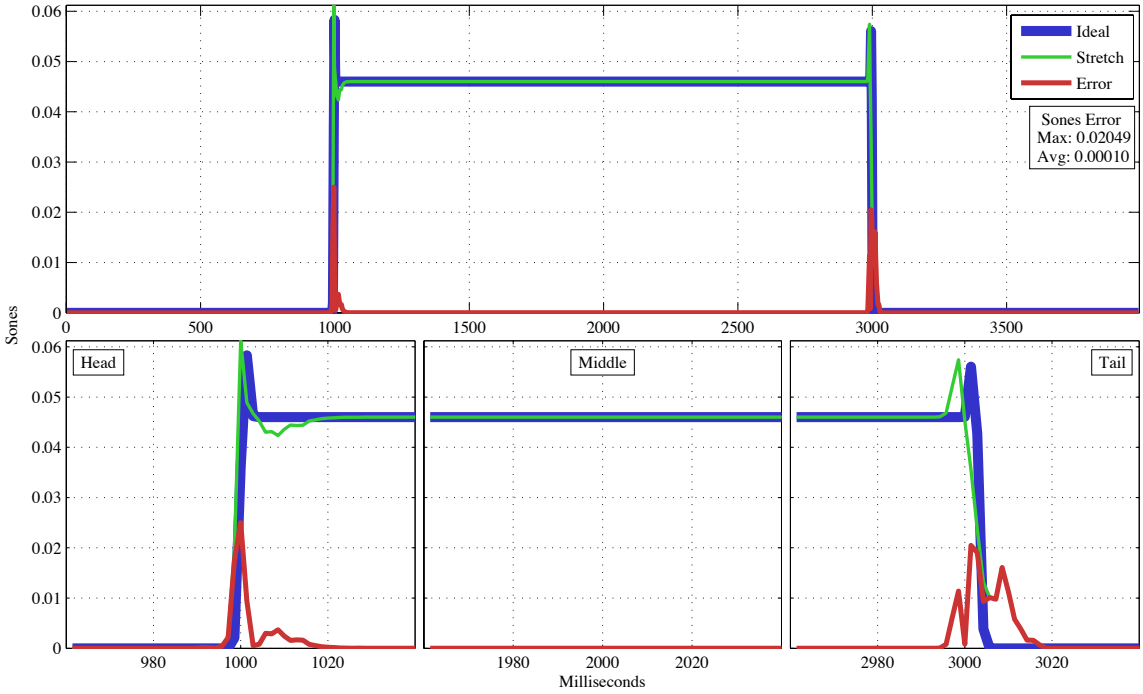

## 7.2 – Moving Spectral Average Graphs

### 7.2.1 – Sine Moving Spectral Average Graphs

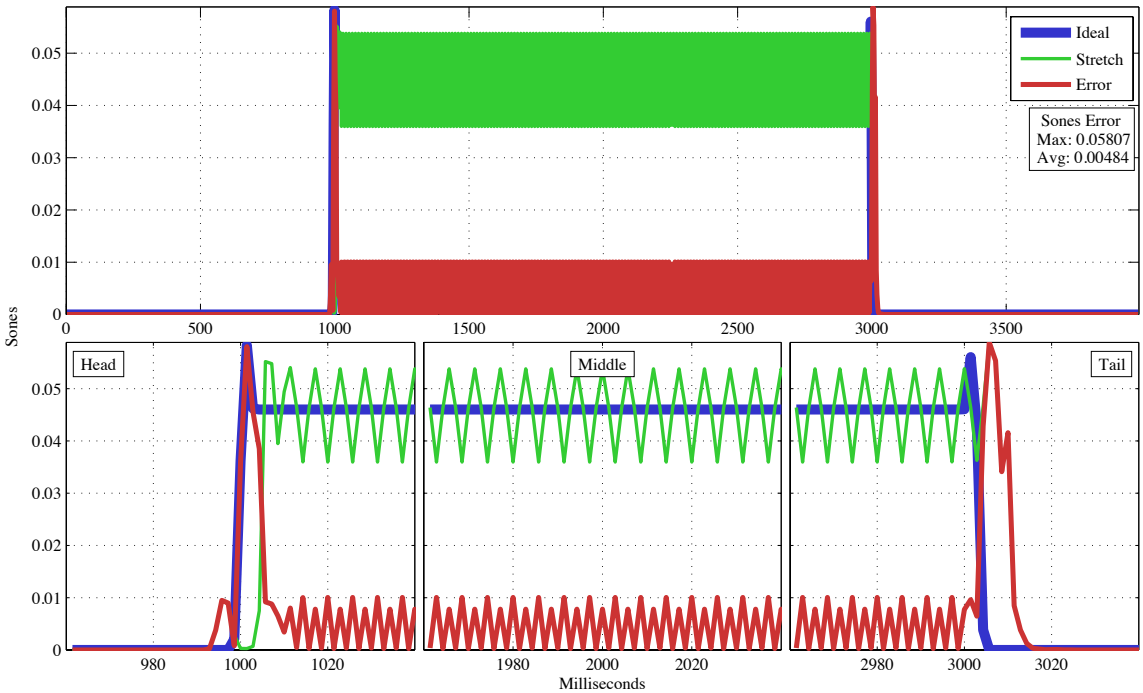
**Graph 7.2.1.1 –** Sine Classic Moving Spectral Average



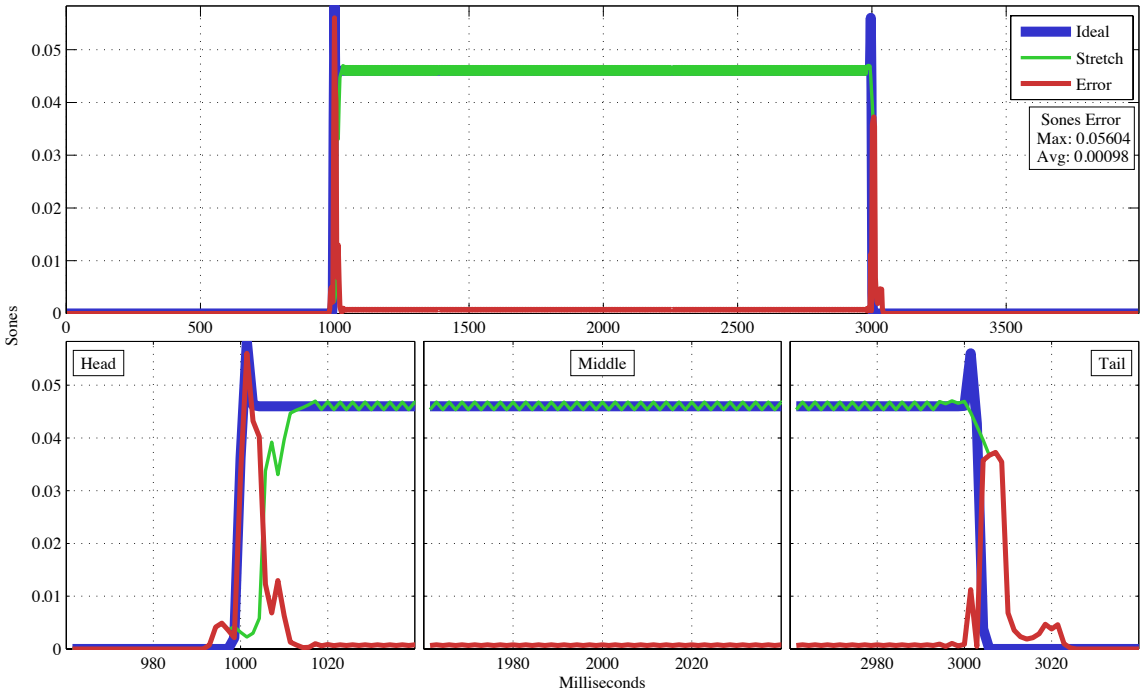
**Graph 7.2.1.2 –** Sine Lock Moving Spectral Average



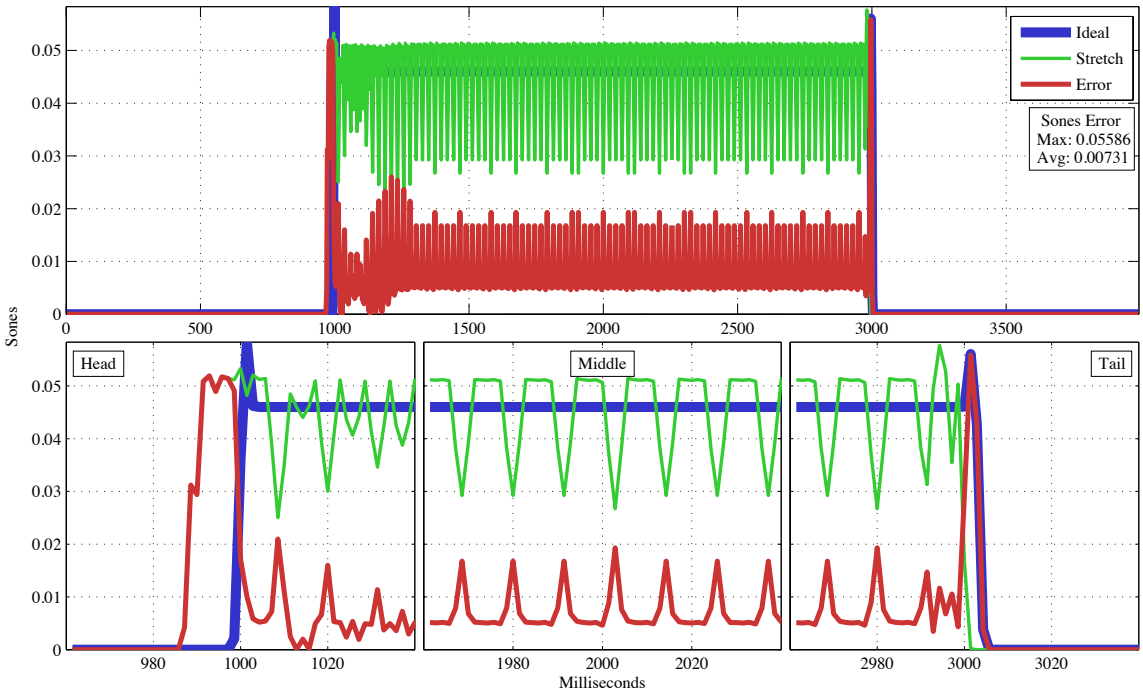

**Graph 7.2.1.3 –** Sine Peak Moving Spectral Average



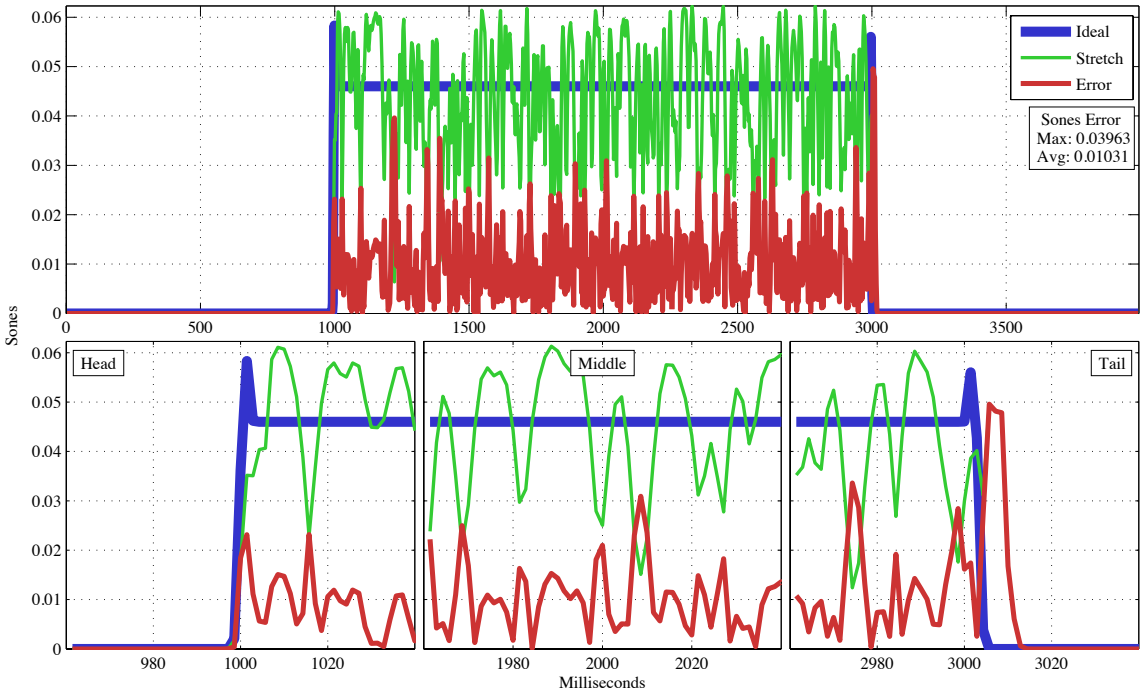
**Graph 7.2.1.4 –** Sine Bank Moving Spectral Average




**Graph 7.2.1.5 –** Sine Sola Moving Spectral Average

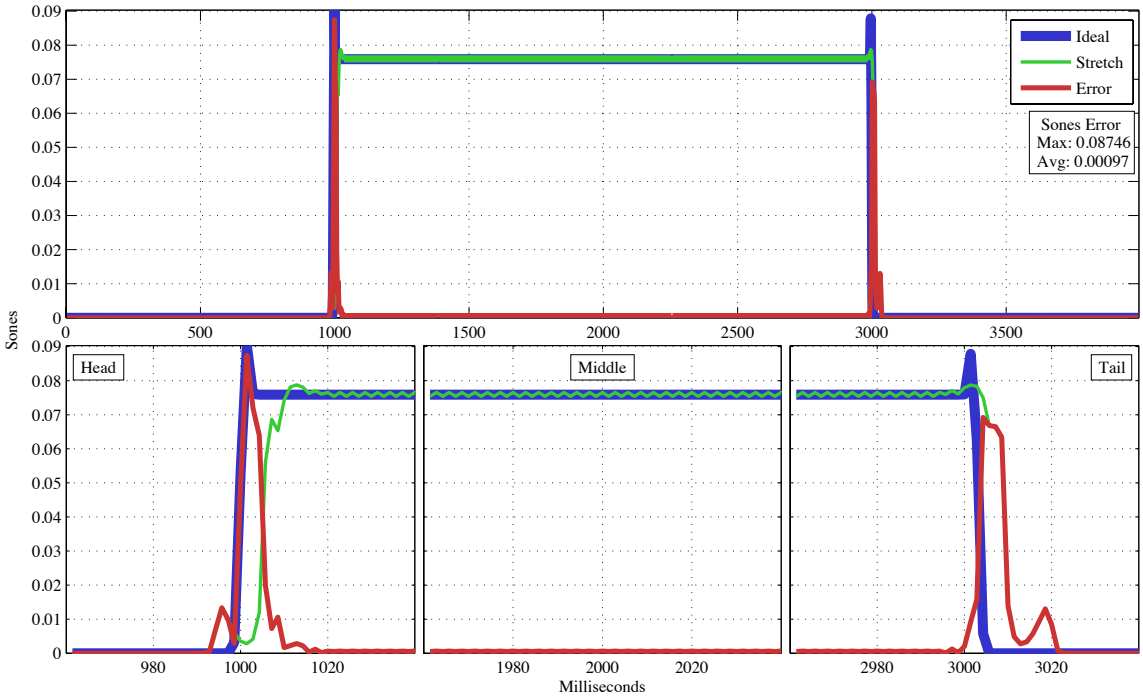


**Graph 7.2.1.6 –** Sine Ola Moving Spectral Average

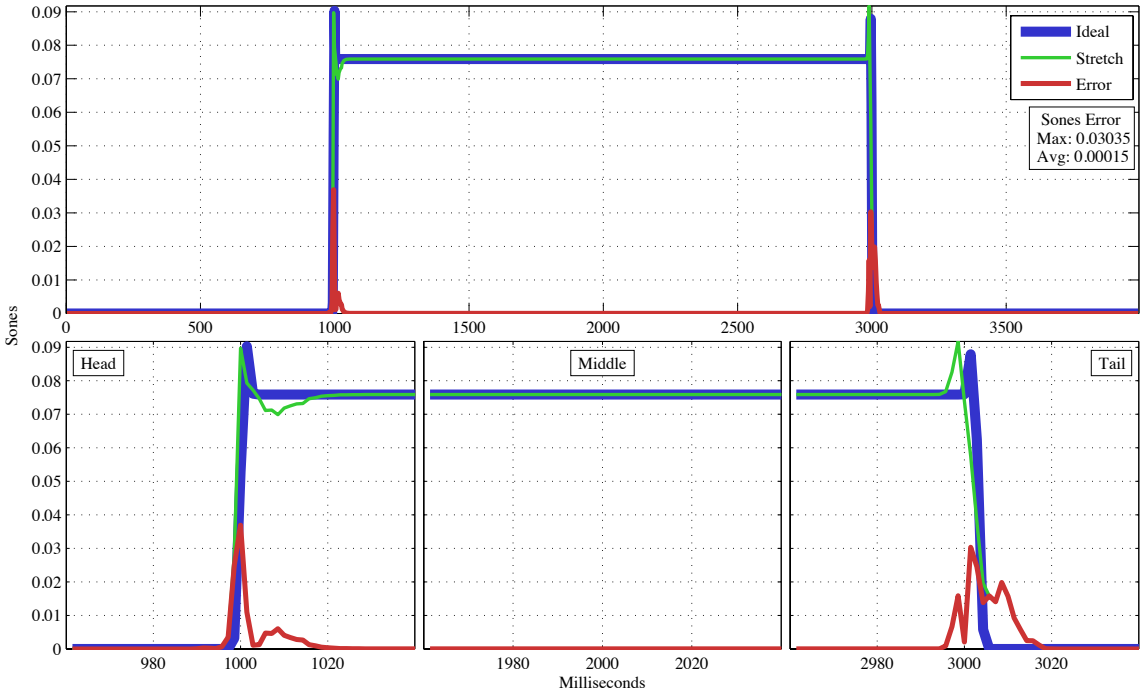



## 7.2.2 – Square Moving Spectral Average Graphs

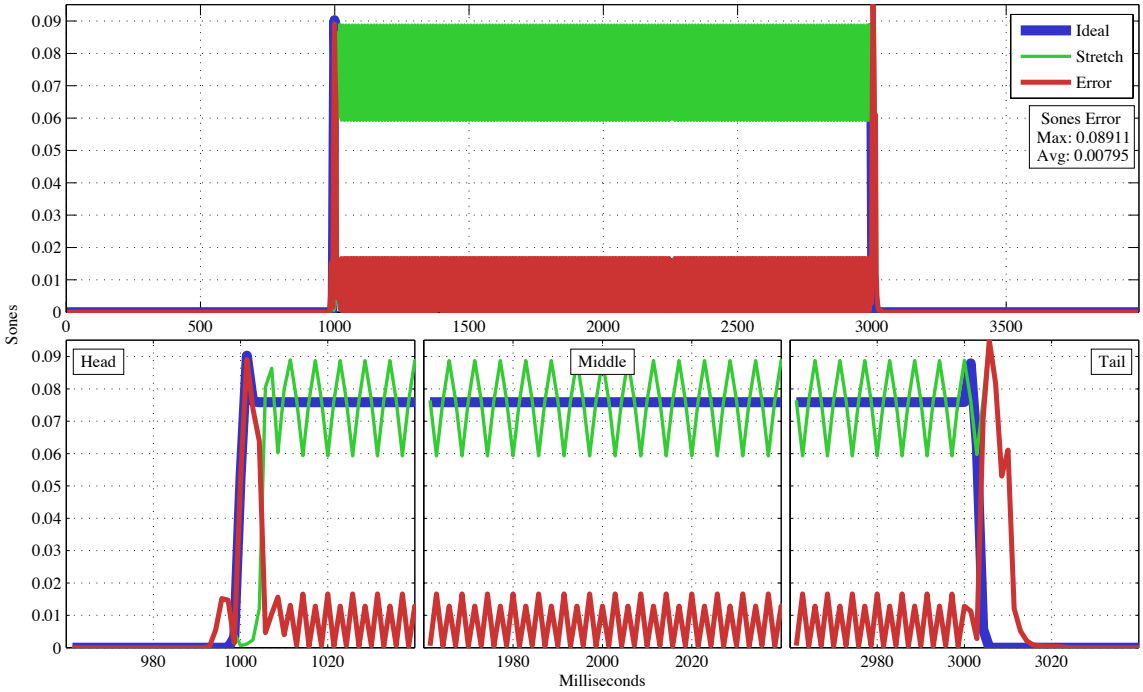
**Graph 7.2.2.1 –** Square Classic Moving Spectral Average



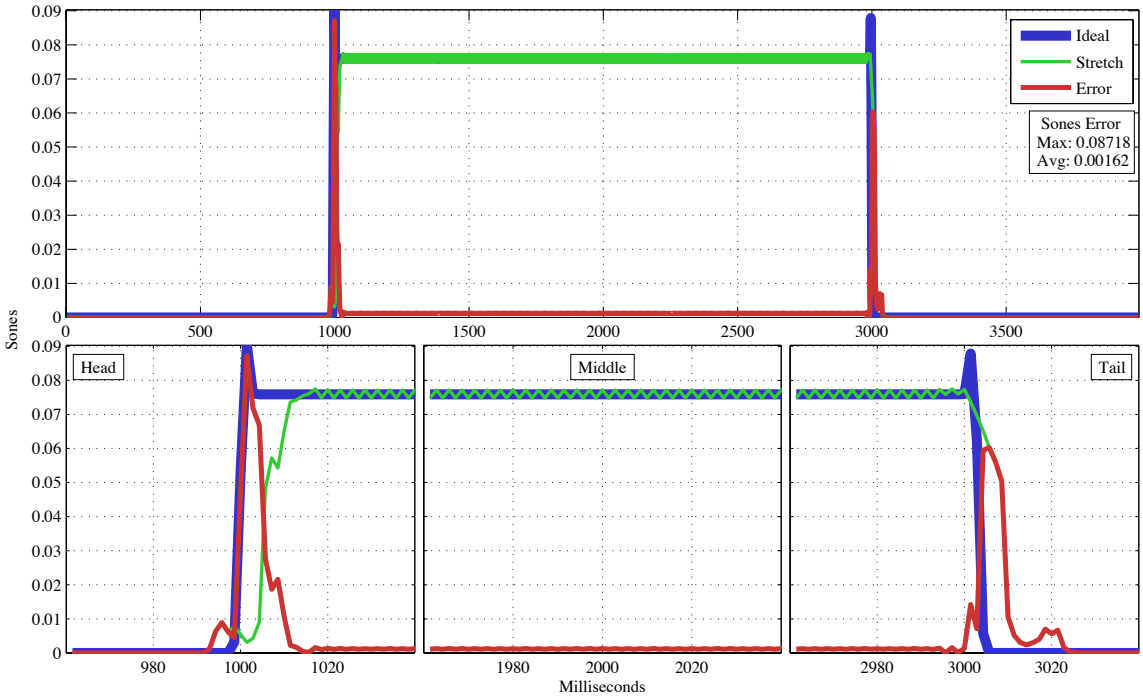
**Graph 7.2.2.2 –** Square Lock Moving Spectral Average



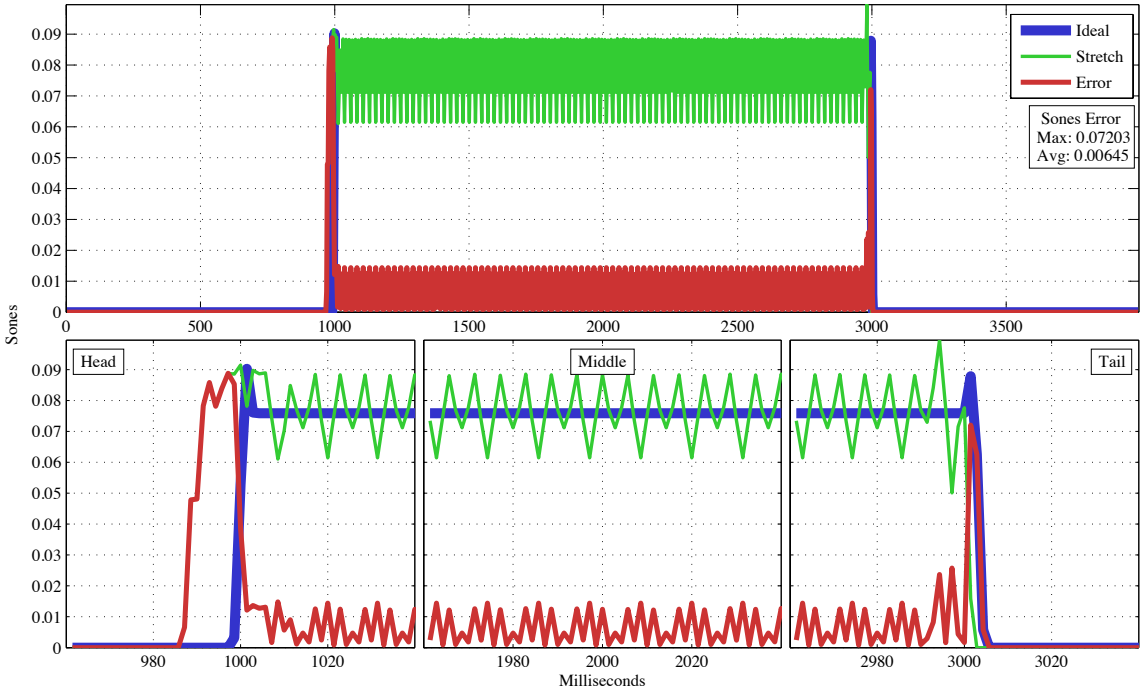

**Graph 7.2.2.3 –** Square Peak Moving Spectral Average



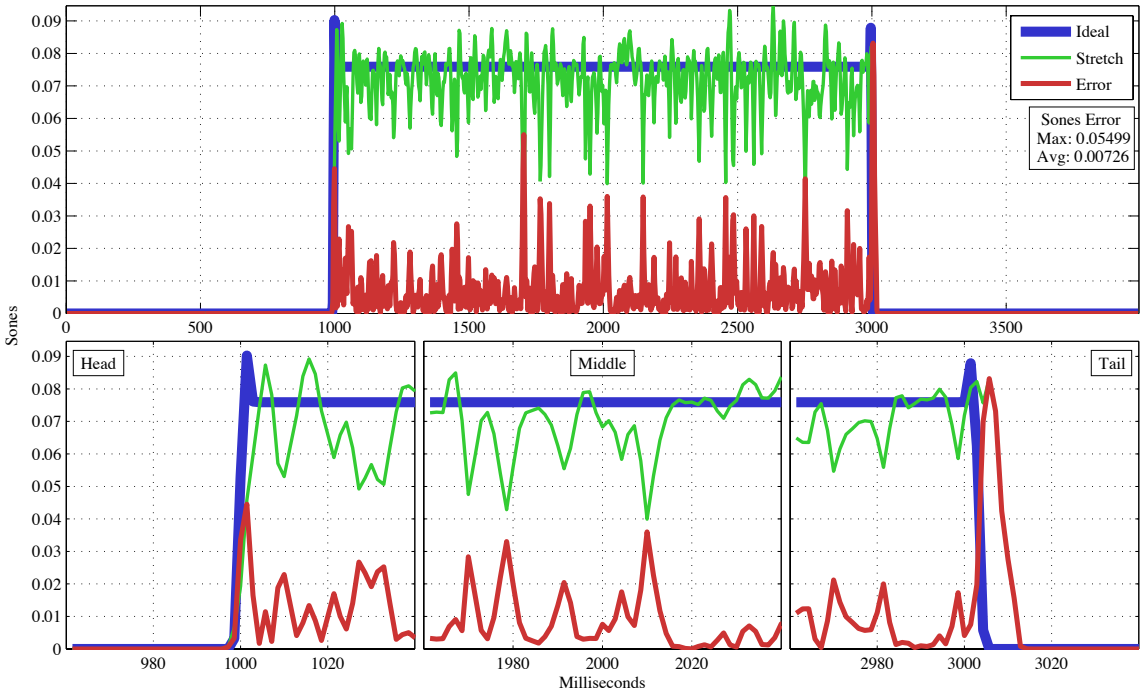
**Graph 7.2.2.4 –** Square Bank Moving Spectral Average




**Graph 7.2.2.5 –** Square Sola Moving Spectral Average

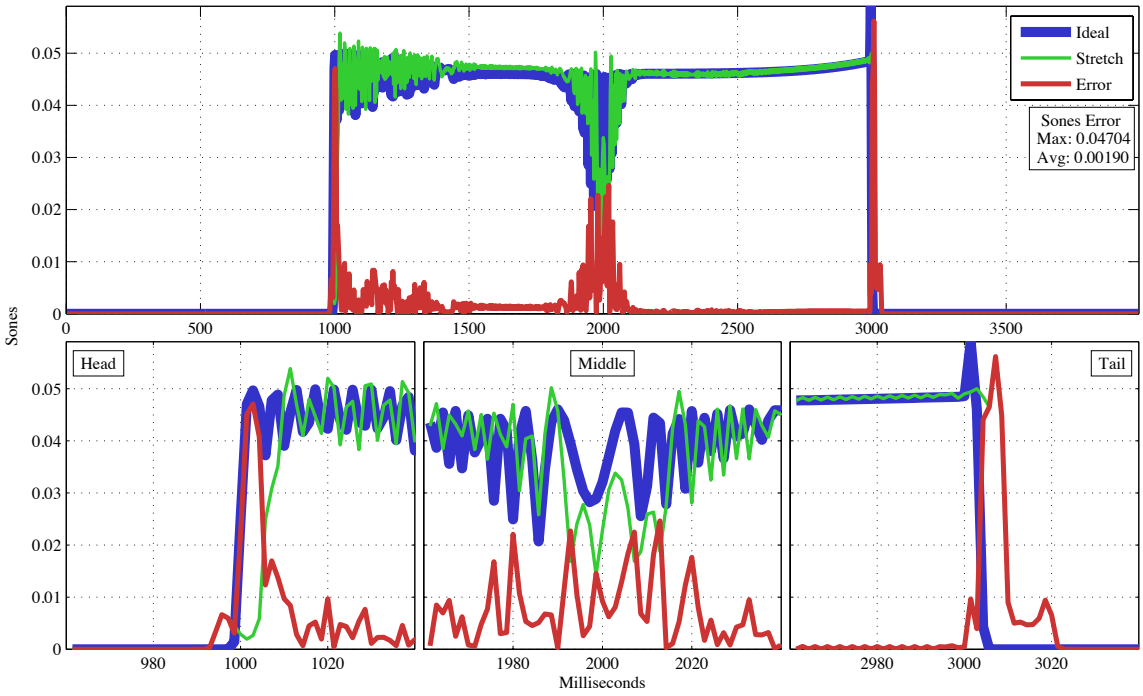


**Graph 7.2.2.6 –** Square Ola Moving Spectral Average

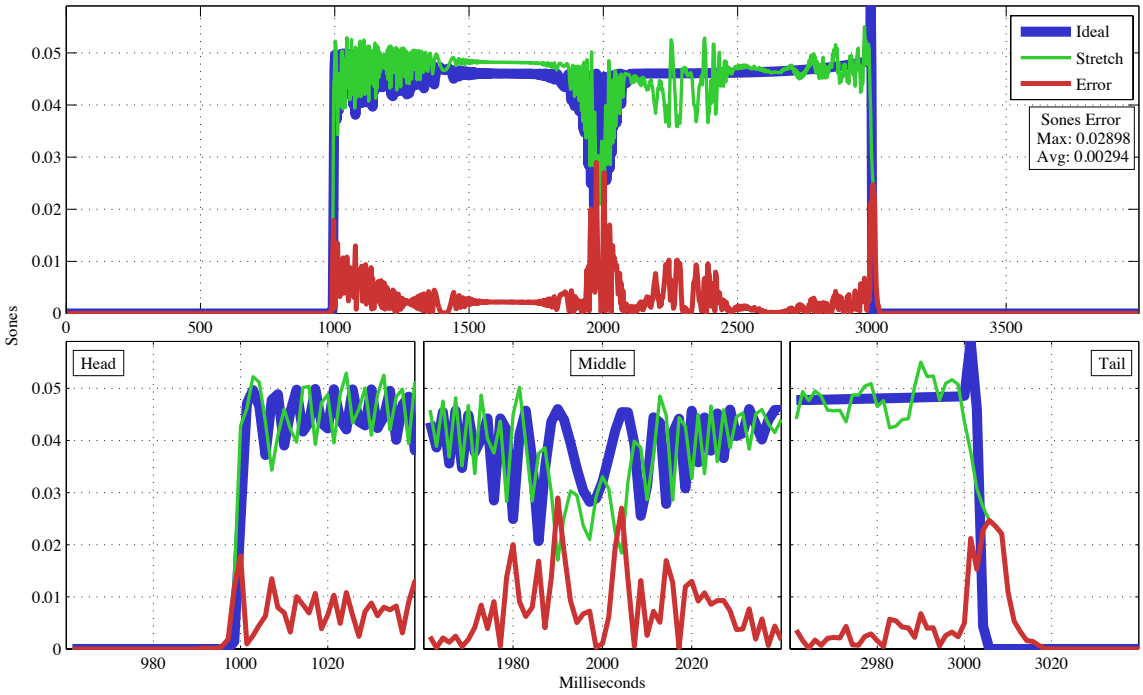



## 7.2.3 – Sweep Moving Spectral Average Graphs

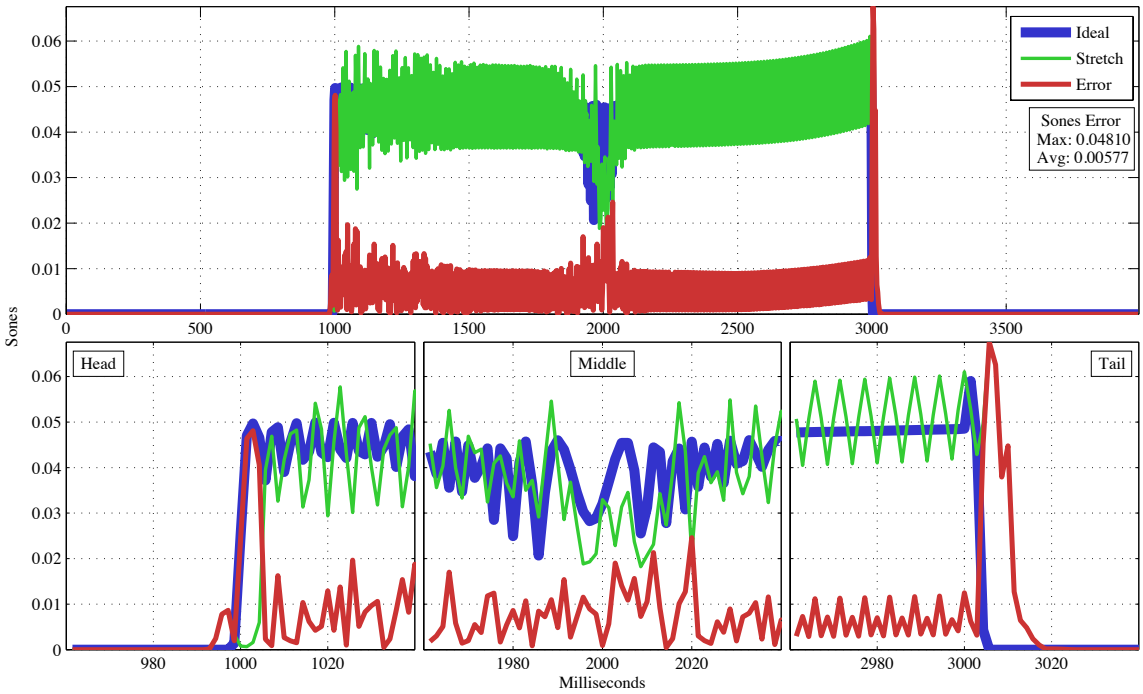
**Graph 7.2.3.1 –** Sweep Classic Moving Spectral Average



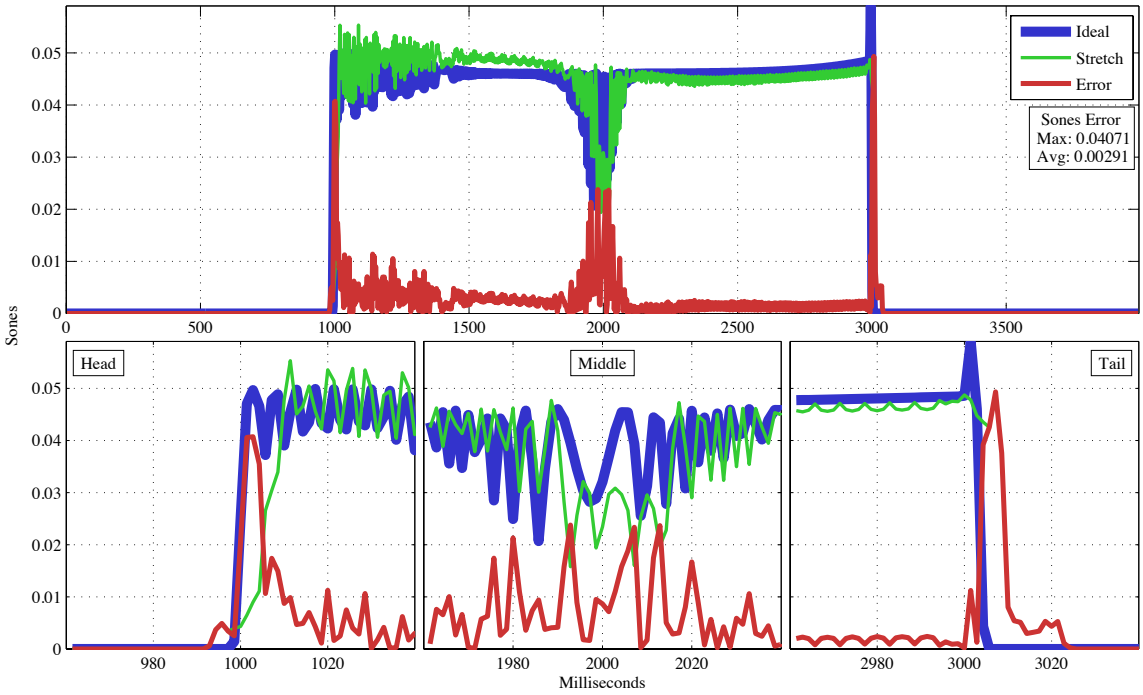
**Graph 7.2.3.2 –** Sweep Lock Moving Spectral Average



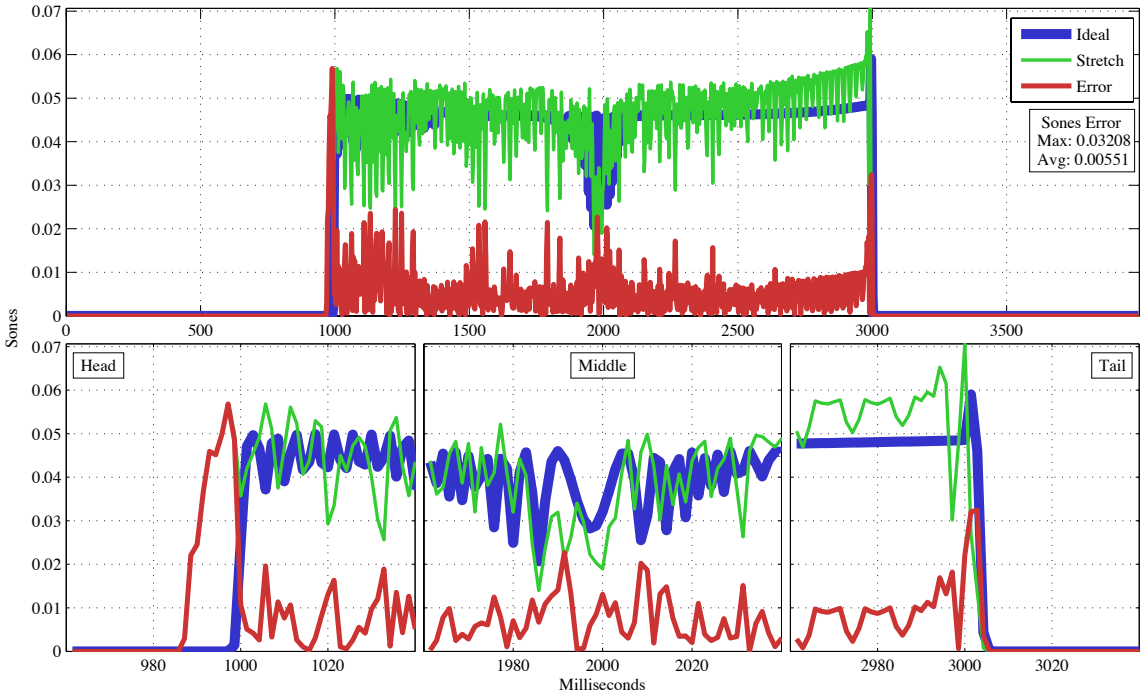

**Graph 7.2.3.3 –** Sweep Peak Moving Spectral Average



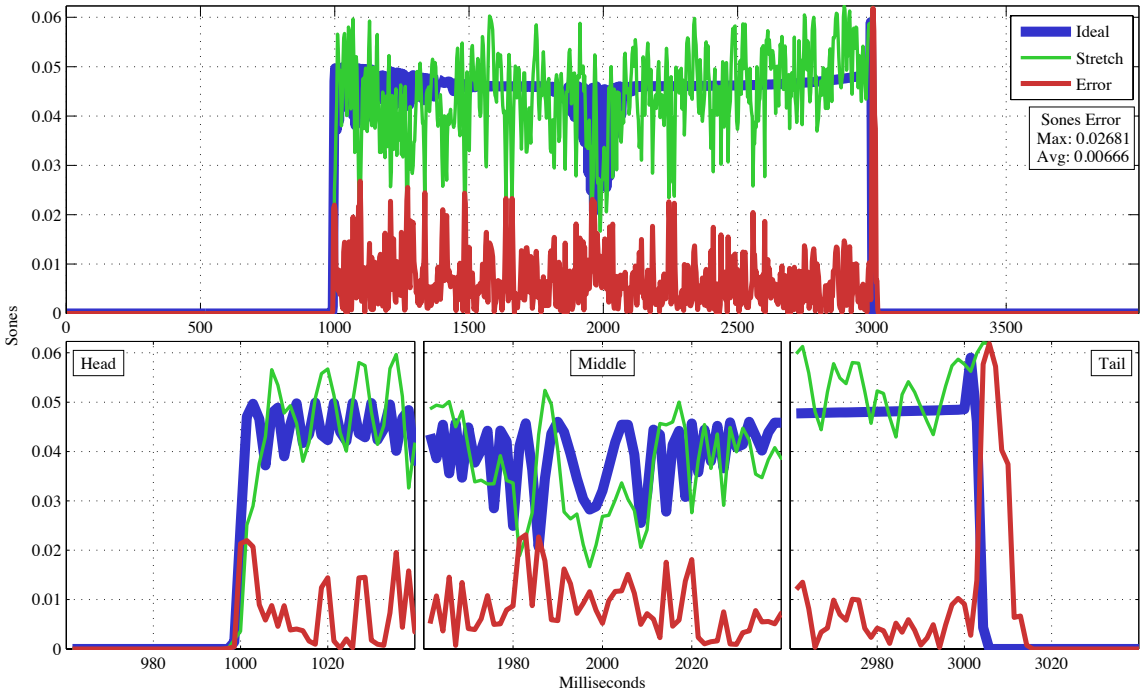
**Graph 7.2.3.4 –** Sweep Bank Moving Spectral Average




**Graph 7.2.3.5 –** Sweep Sola Moving Spectral Average

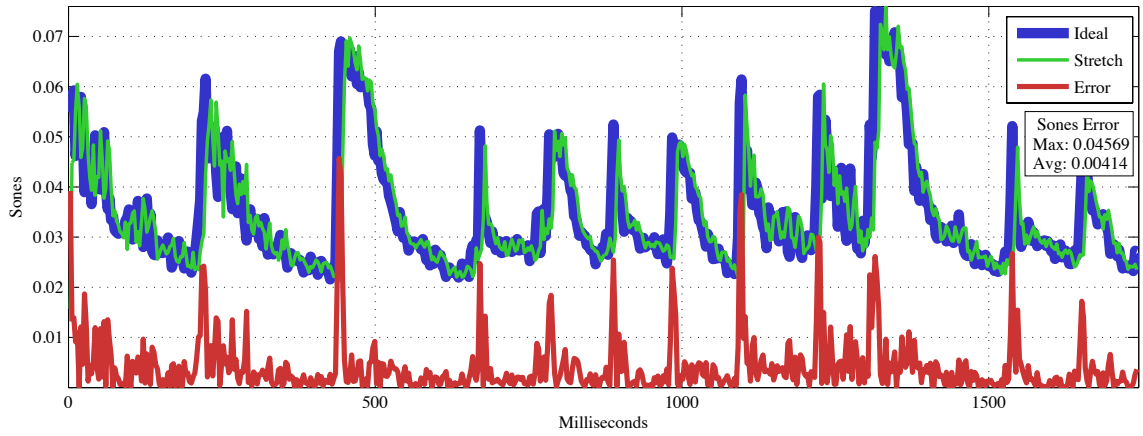


**Graph 7.2.3.6 –** Sweep Ola Moving Spectral Average

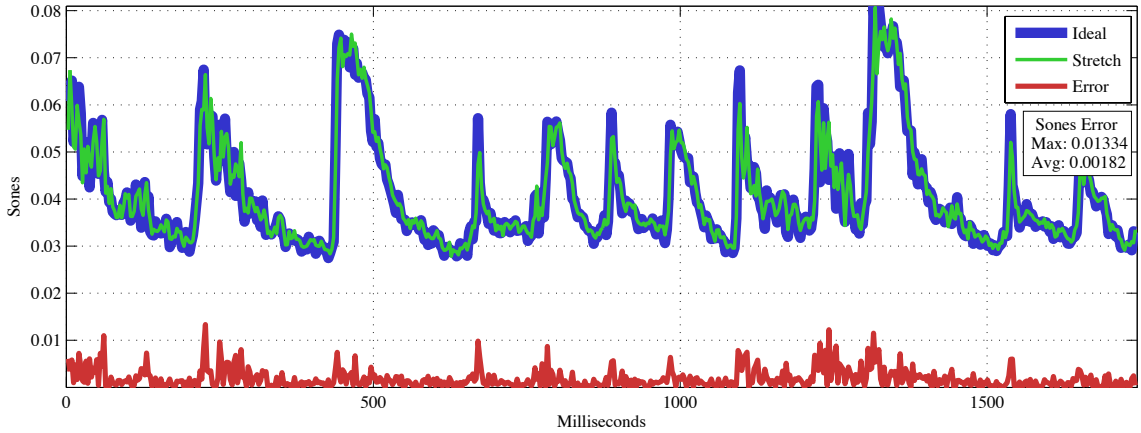



## 7.2.4 – *Amen* Moving Spectral Average Graphs

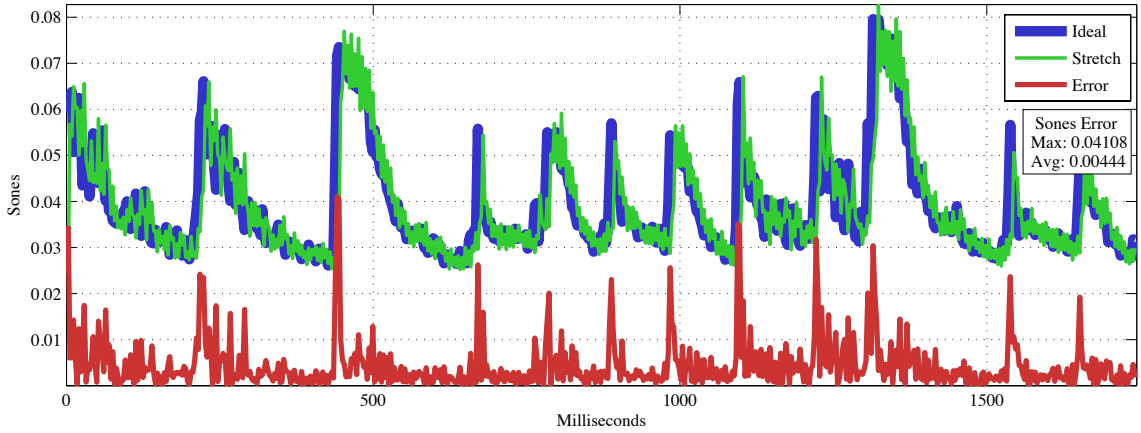
**Graph 7.2.4.1 –** *Amen* Classic Moving Spectral Average



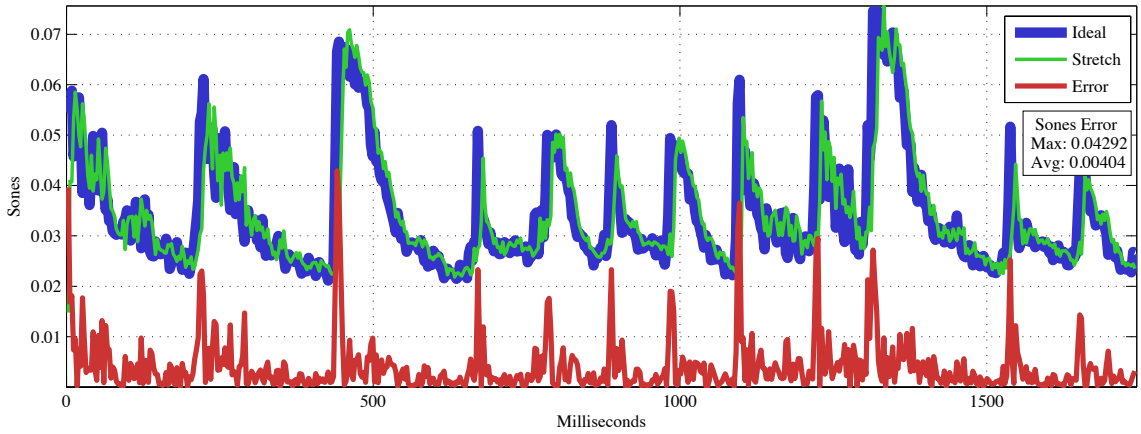
**Graph 7.2.4.2 –** *Amen* Lock Moving Spectral Average



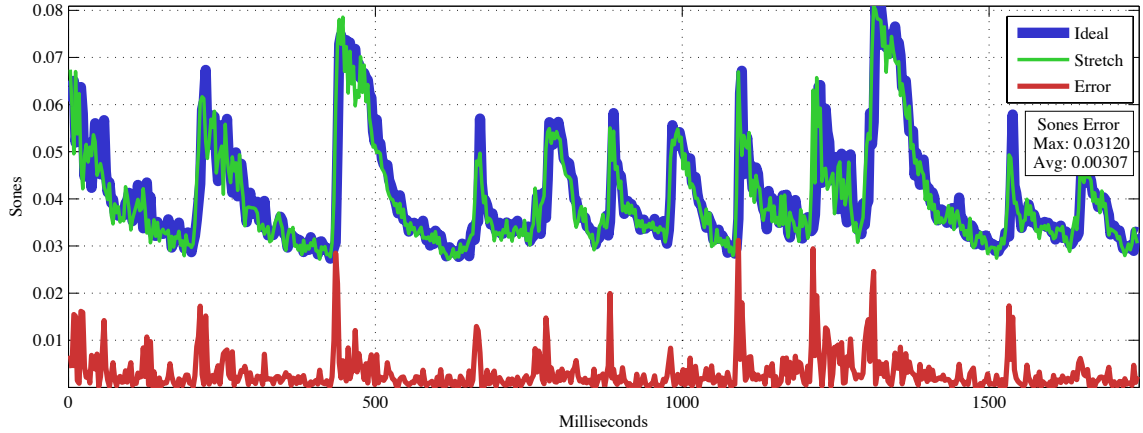

**Graph 7.2.4.3 –** *Amen* Peak Moving Spectral Average



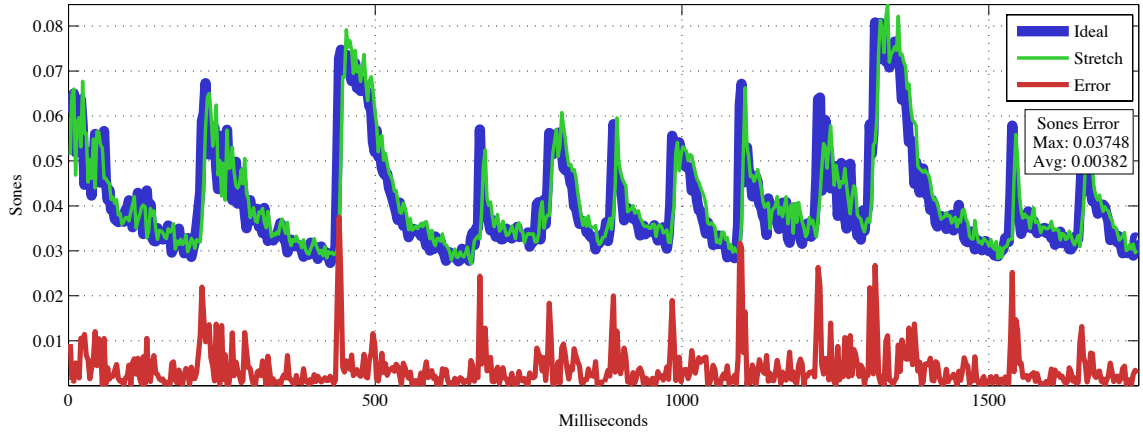

**Graph 7.2.4.4 –** *Amen* Bank Moving Spectral Average



**Graph 7.2.4.5 –** *Amen* Sola Moving Spectral Average Graph

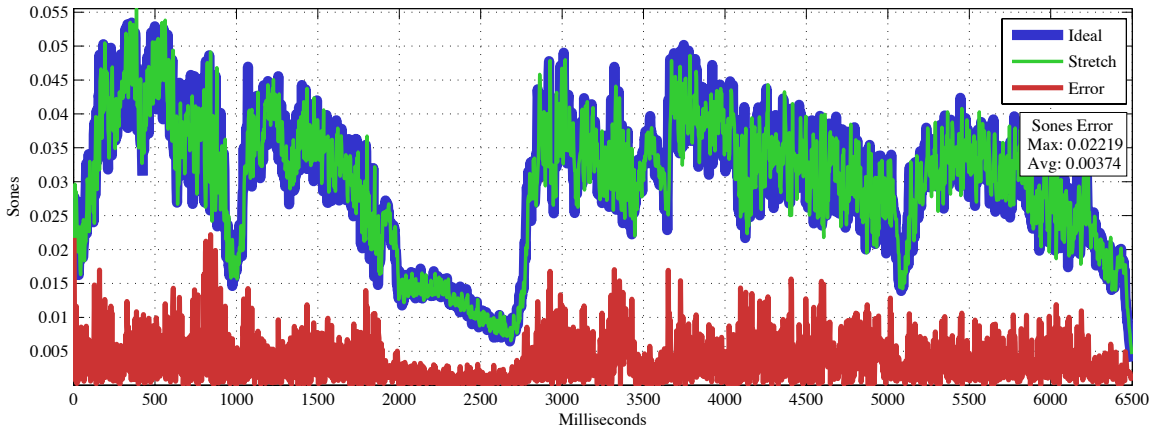



**Graph 7.2.4.6 –** *Amen* Ola Moving Spectral Average Graph

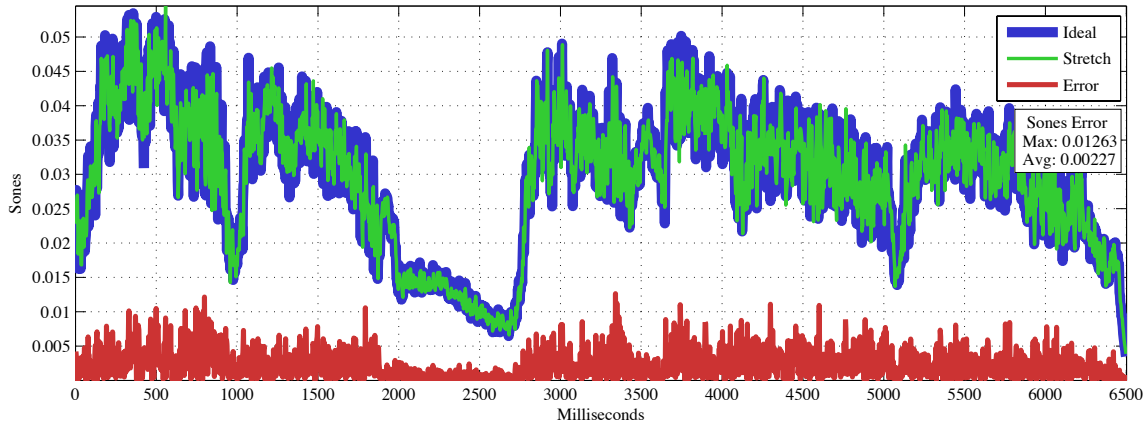



## 7.2.5 – *Autumn* Moving Spectral Average Graphs

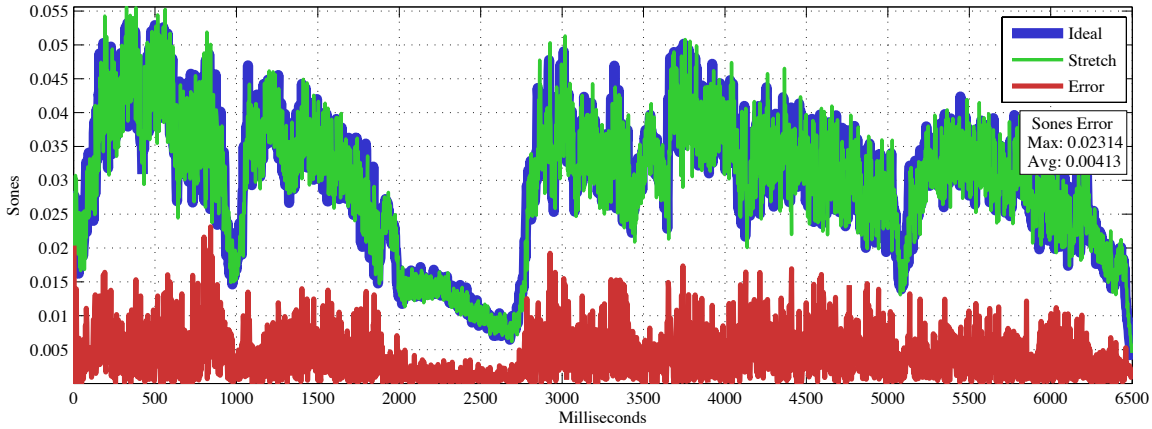
**Graph 7.2.5.1 –** *Autumn* Classic Moving Spectral Average



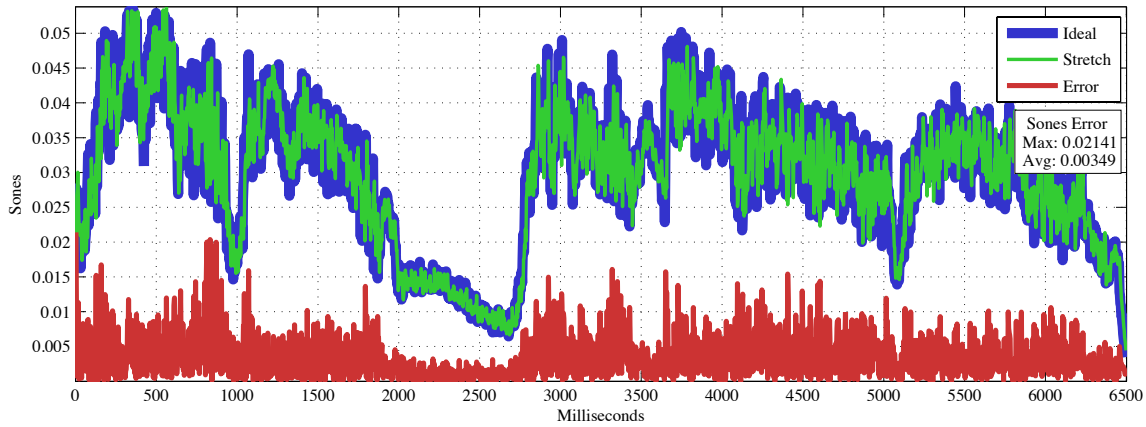
**Graph 7.2.5.2 –** *Autumn* Lock Moving Spectral Average



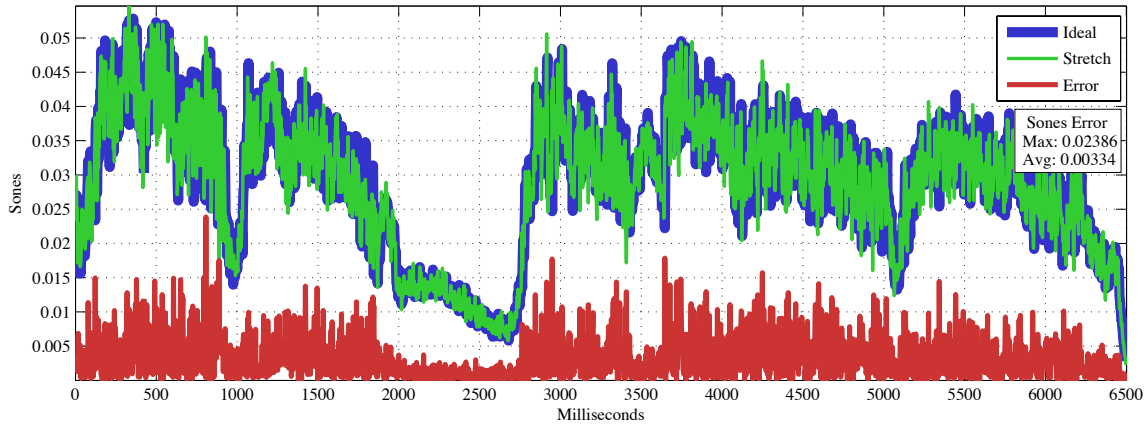

**Graph 7.2.5.3 –** *Autumn* Peak Moving Spectral Average



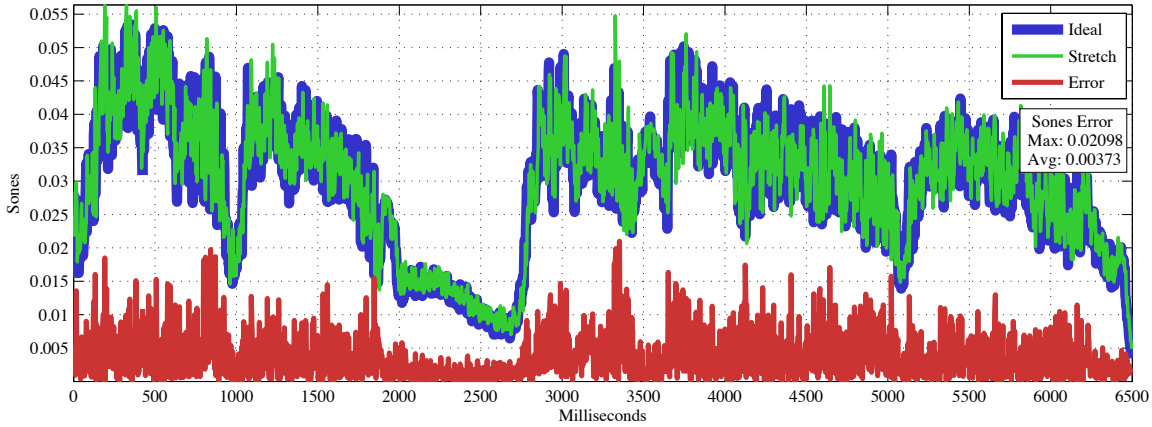

**Graph 7.2.5.4 –** *Autumn* Bank Moving Spectral Average



**Graph 7.2.5.5 –** *Autumn* Sola Moving Spectral Average

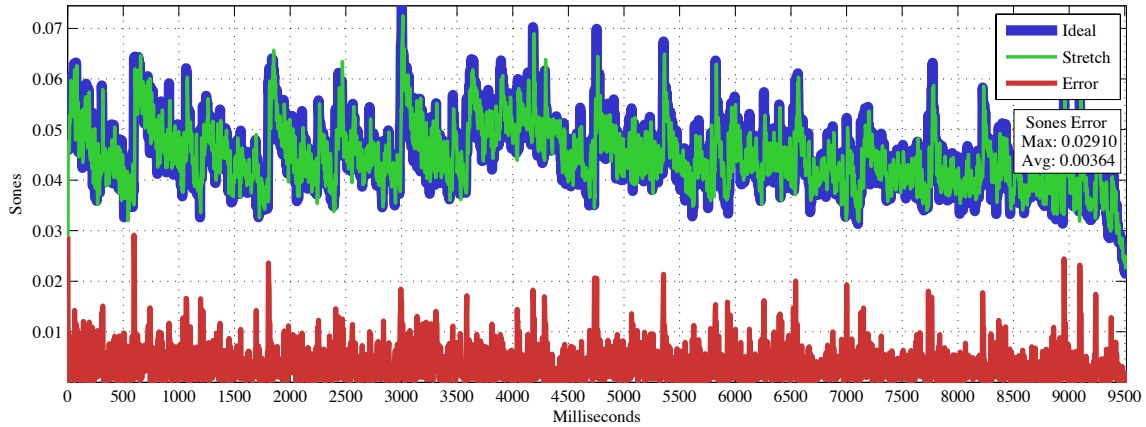



**Graph 7.2.5.6 –** *Autumn* Ola Moving Spectral Average

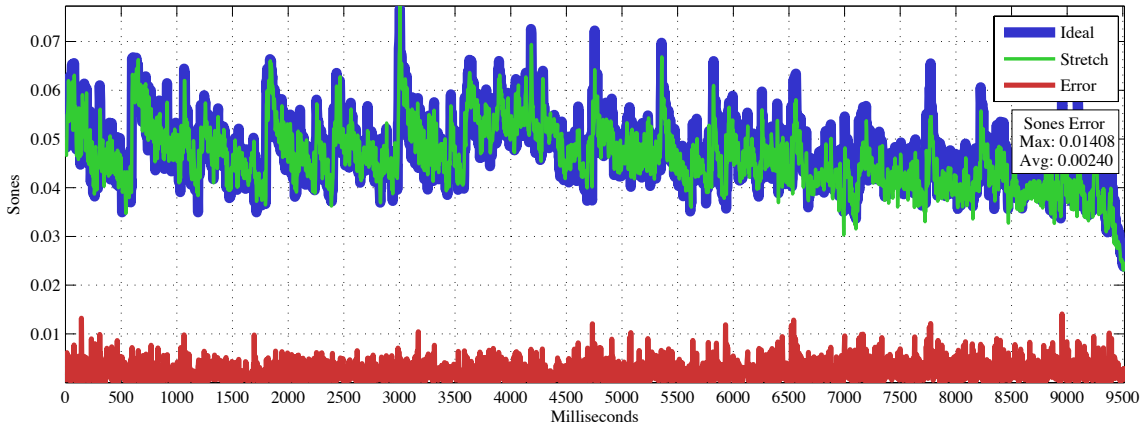



## 7.2.6 – *Peaches* Moving Spectral Average Graphs

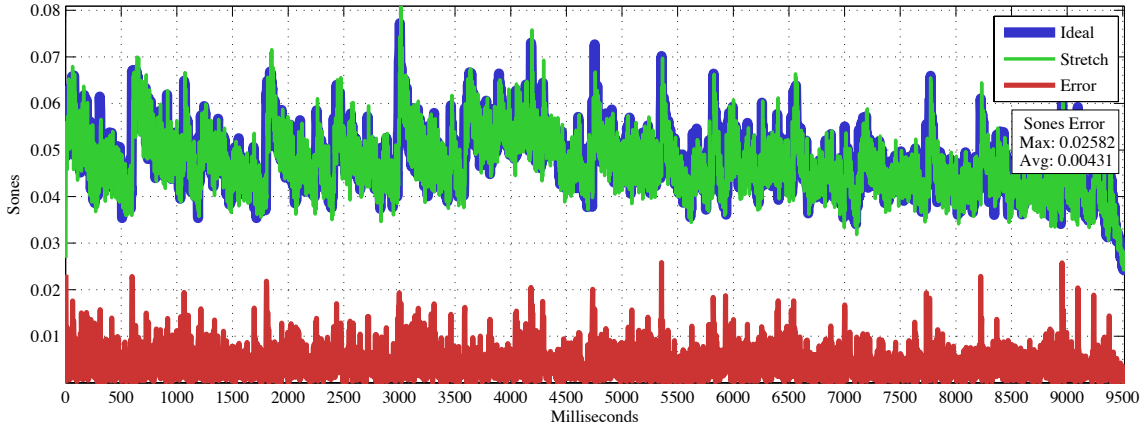
**Graph 7.2.6.1 –** *Peaches* Classic Moving Spectral Average



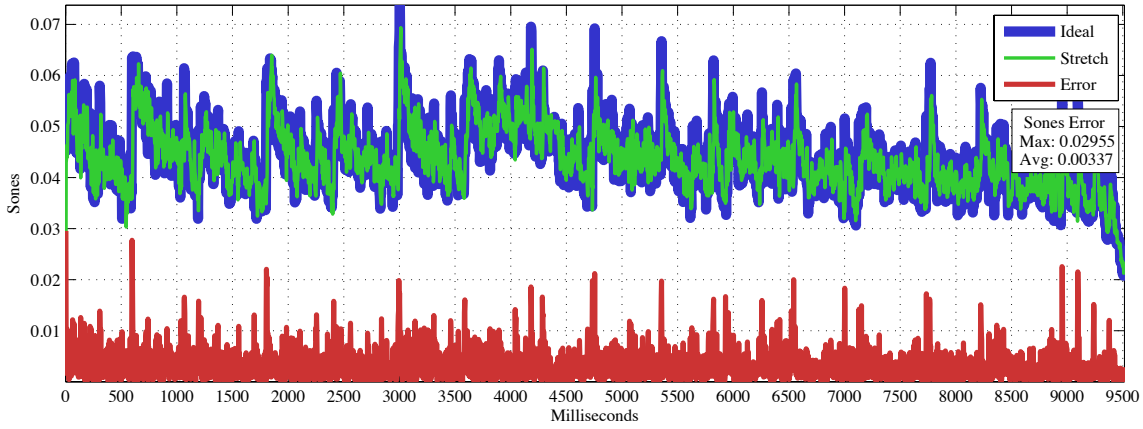
**Graph 7.2.6.2 –** *Peaches* Lock Moving Spectral Average



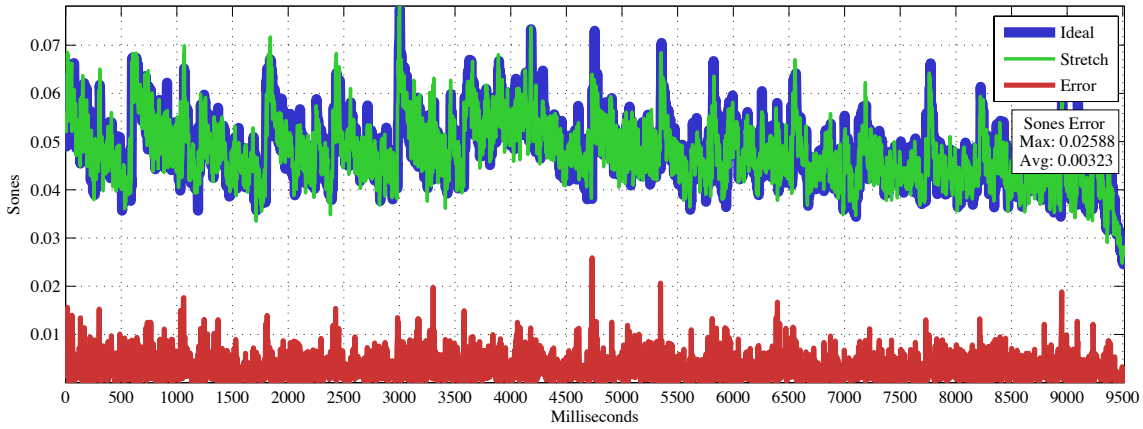

**Graph 7.2.6.3 –** *Peaches* Peak Moving Spectral Average



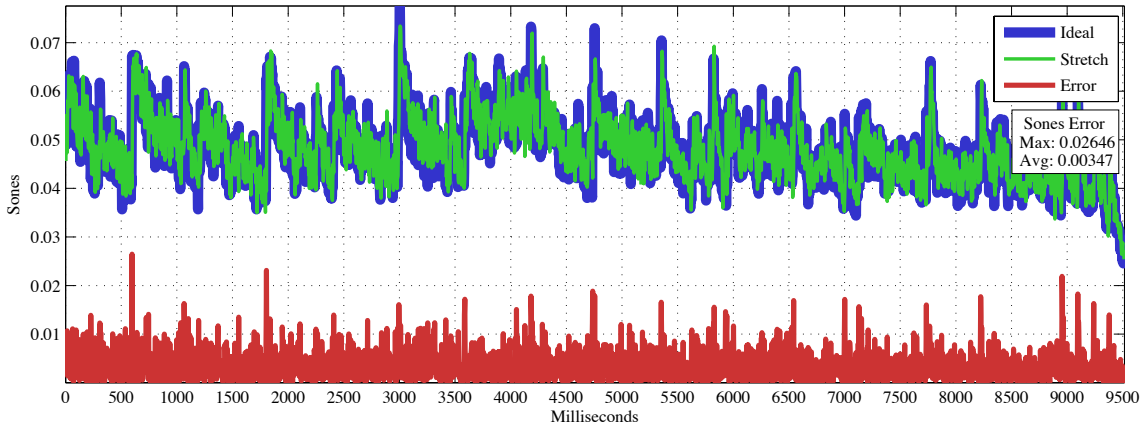

**Graph 7.2.6.4 –** *Peaches* Bank Moving Spectral Average



**Graph 7.2.6.5 –** *Peaches* Sola Moving Spectral Average



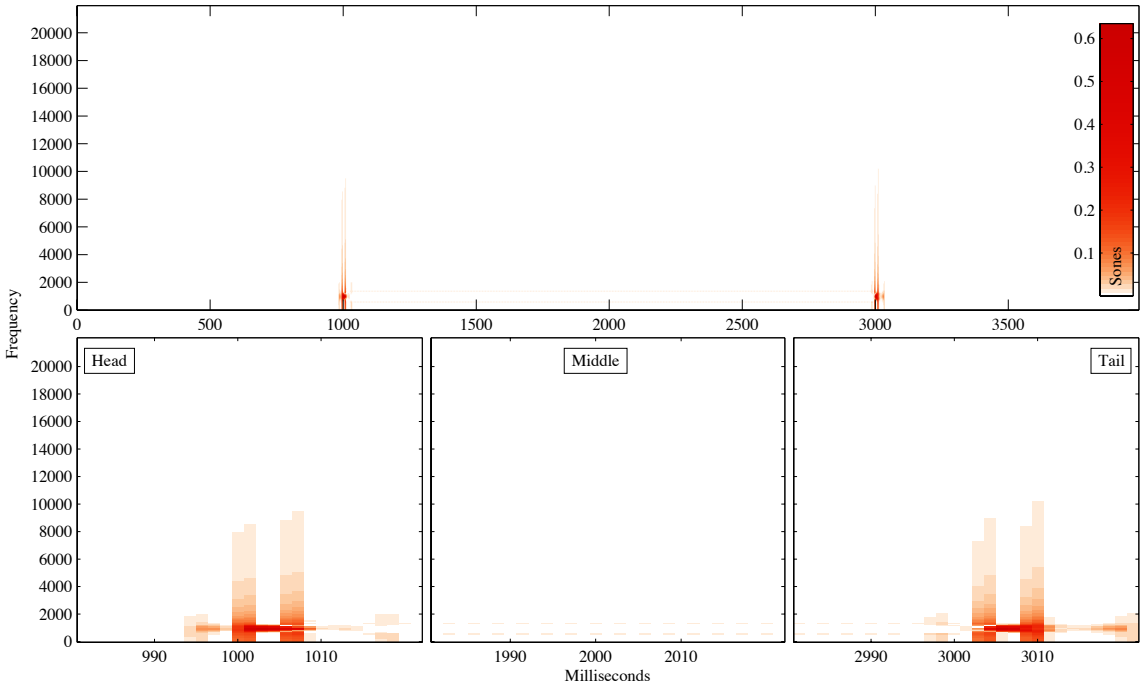

**Graph 7.2.6.6 –** *Peaches* Ola Moving Spectral Average



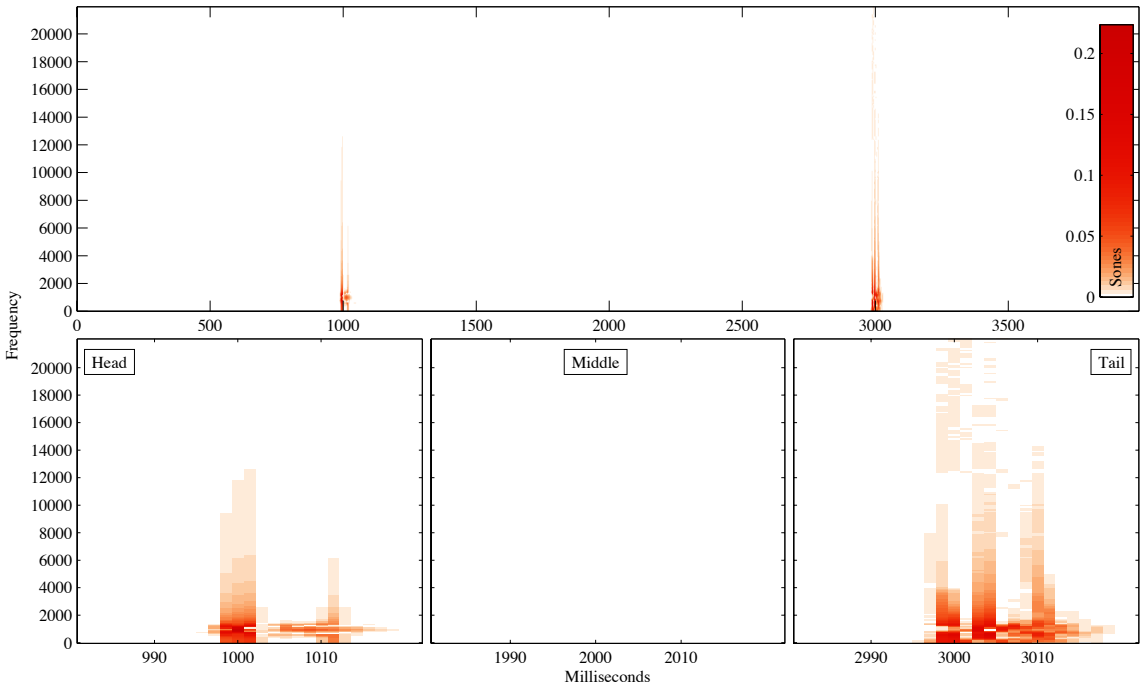
## 7.3 – Error Spectrograms

### 7.3.1 – Sine Error Spectrograms

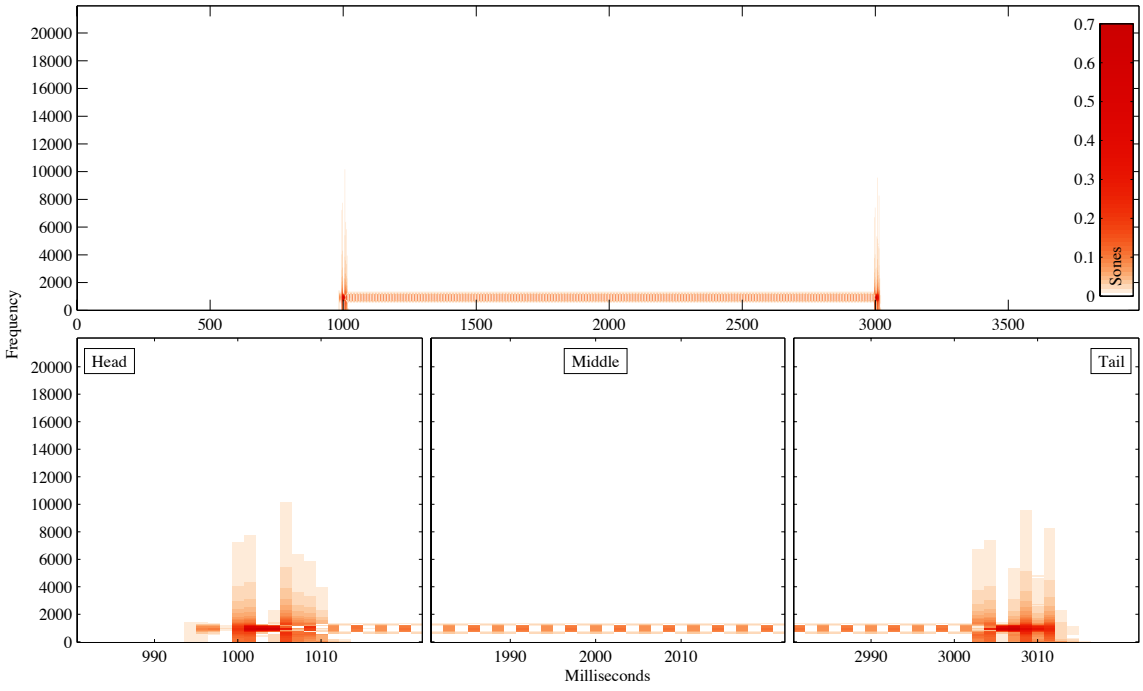
**Graph 7.3.1.1 –** Sine Classic Error Spectrogram



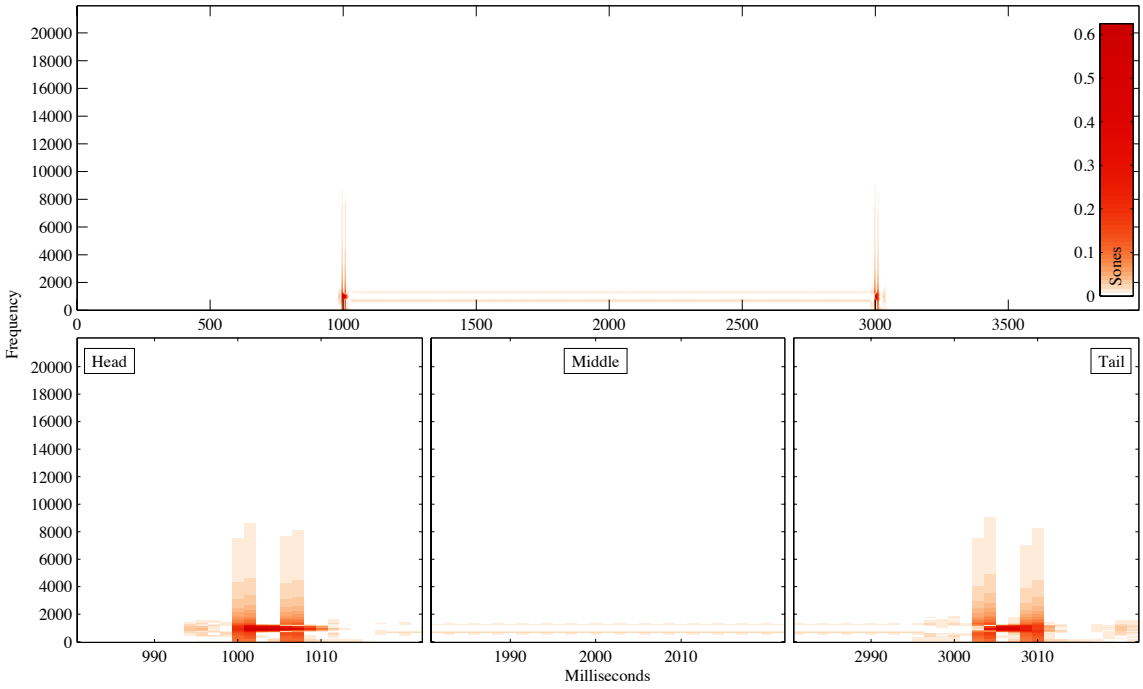

**Graph 7.3.1.2 –** Sine Lock Error Spectrogram



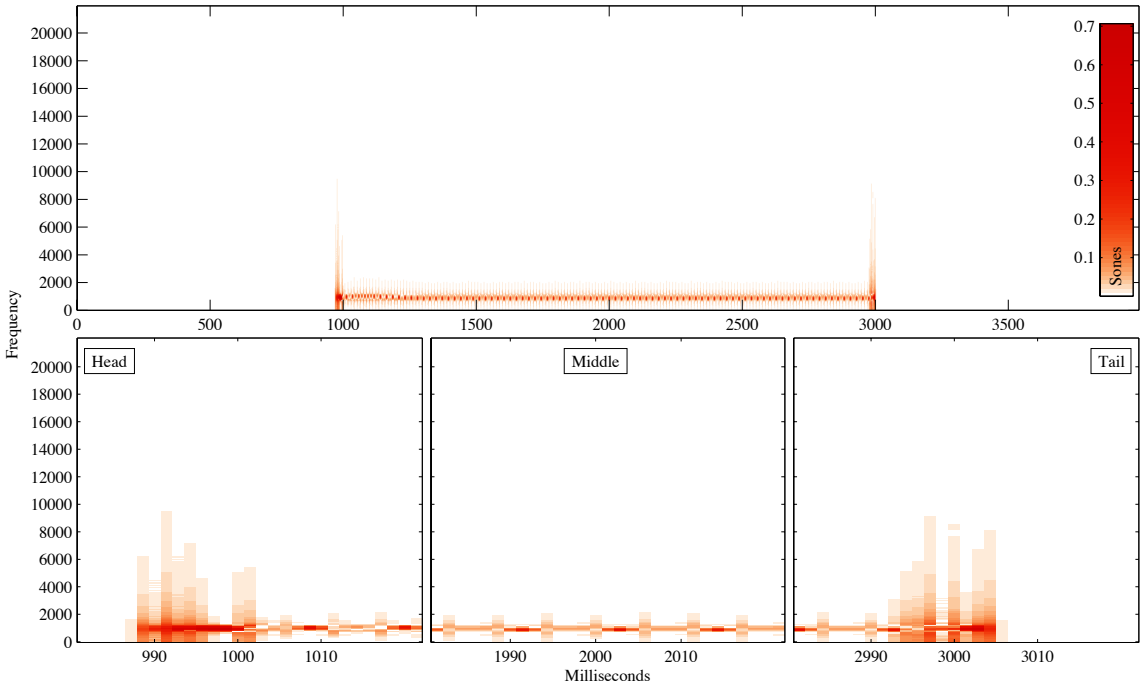
**Graph 7.3.1.3 –** Sine Peak Error Spectrogram



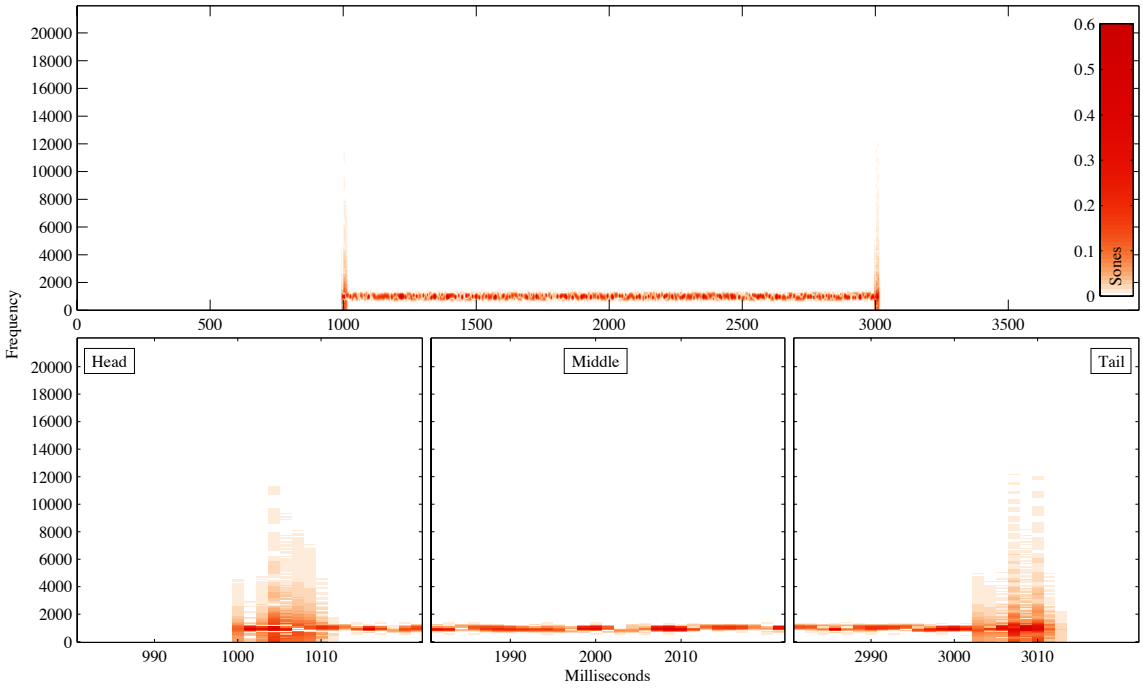

**Graph 7.3.1.4 –** Sine Bank Error Spectrogram



**Graph 7.3.1.5 –** Sine Sola Error Spectrogram

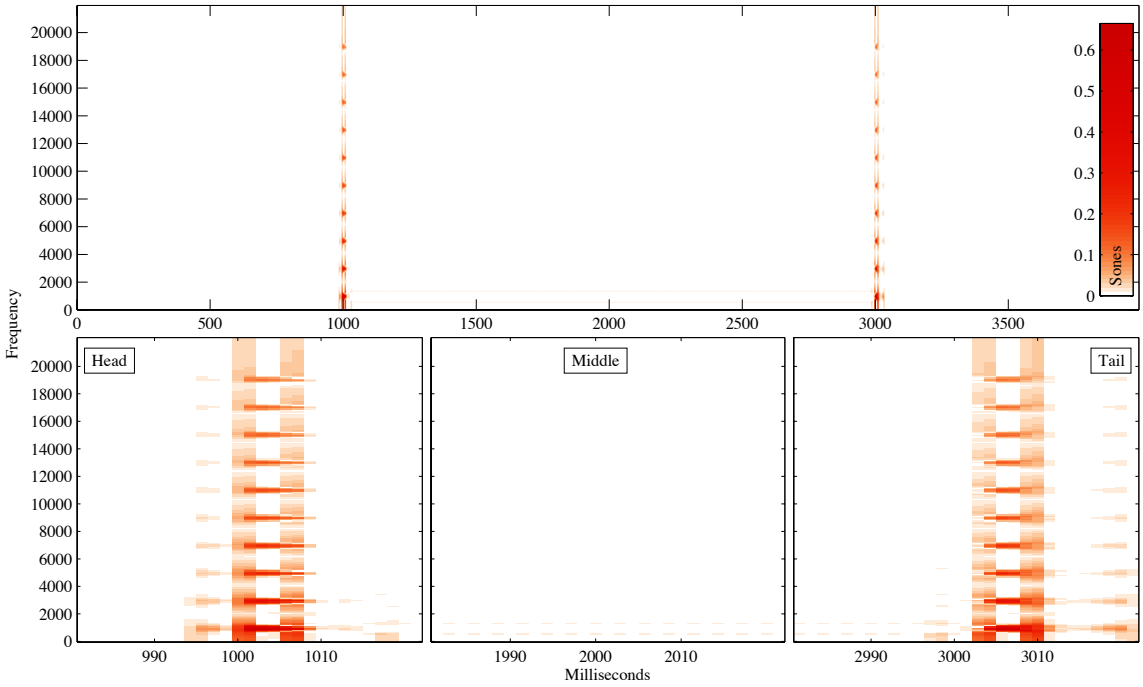



**Graph 7.3.1.6 –** Sine Ola Error Spectrogram

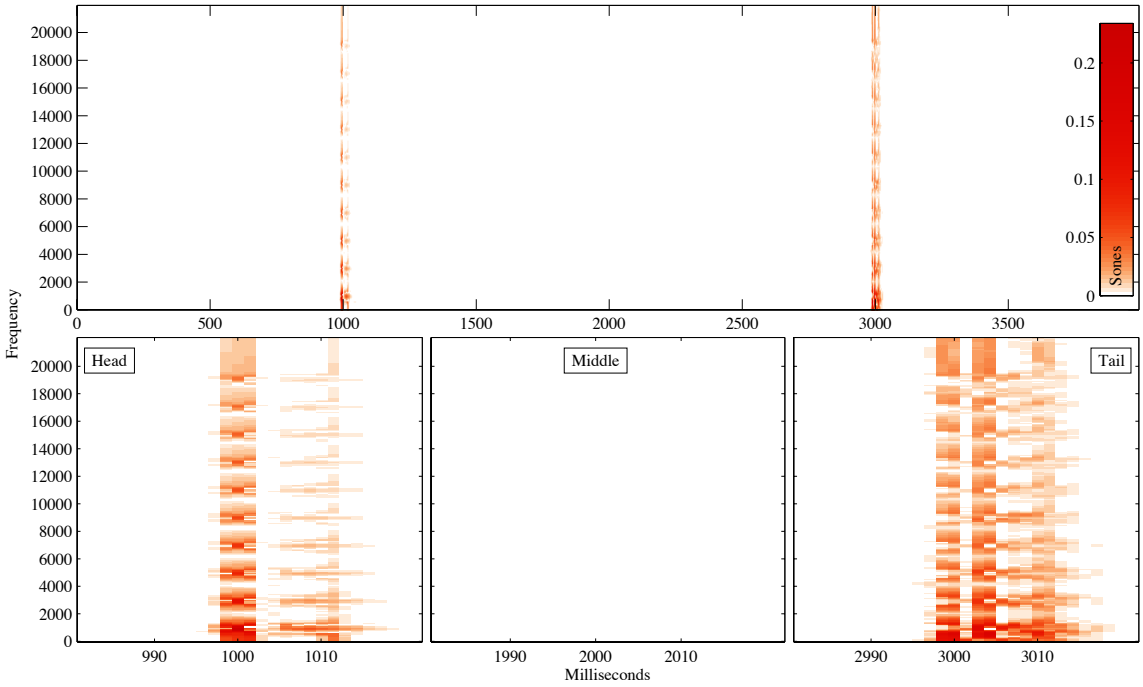


### 7.3.2 – Square Error Spectrograms

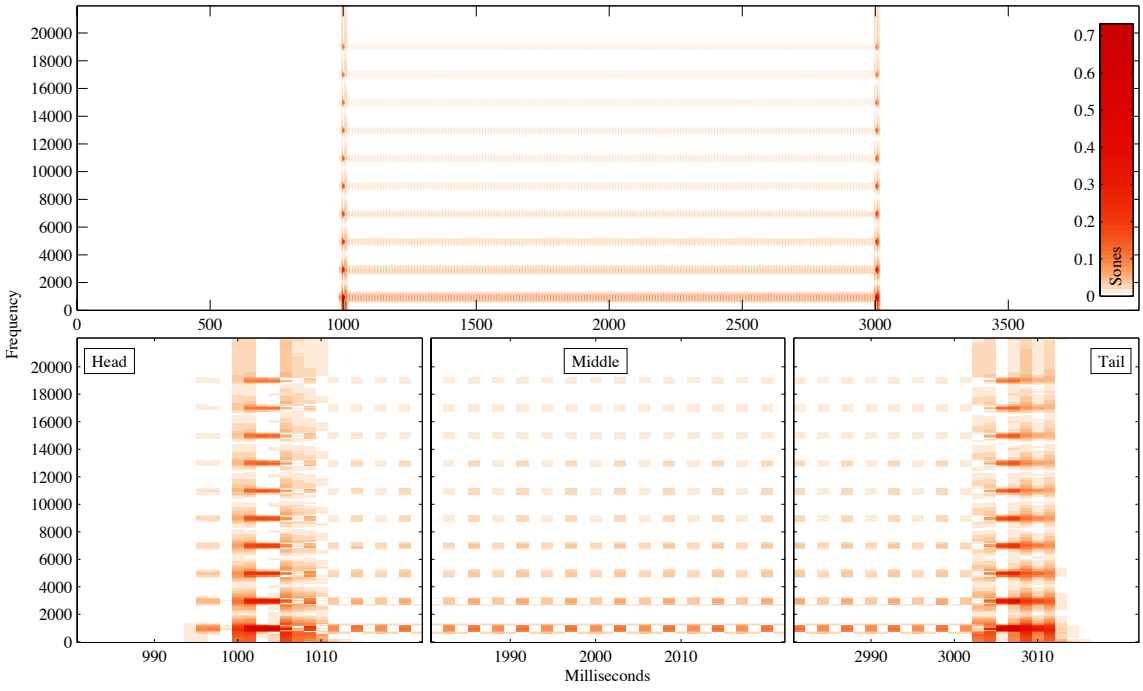
**Graph 7.3.2.1 –** Square Classic Error Spectrogram



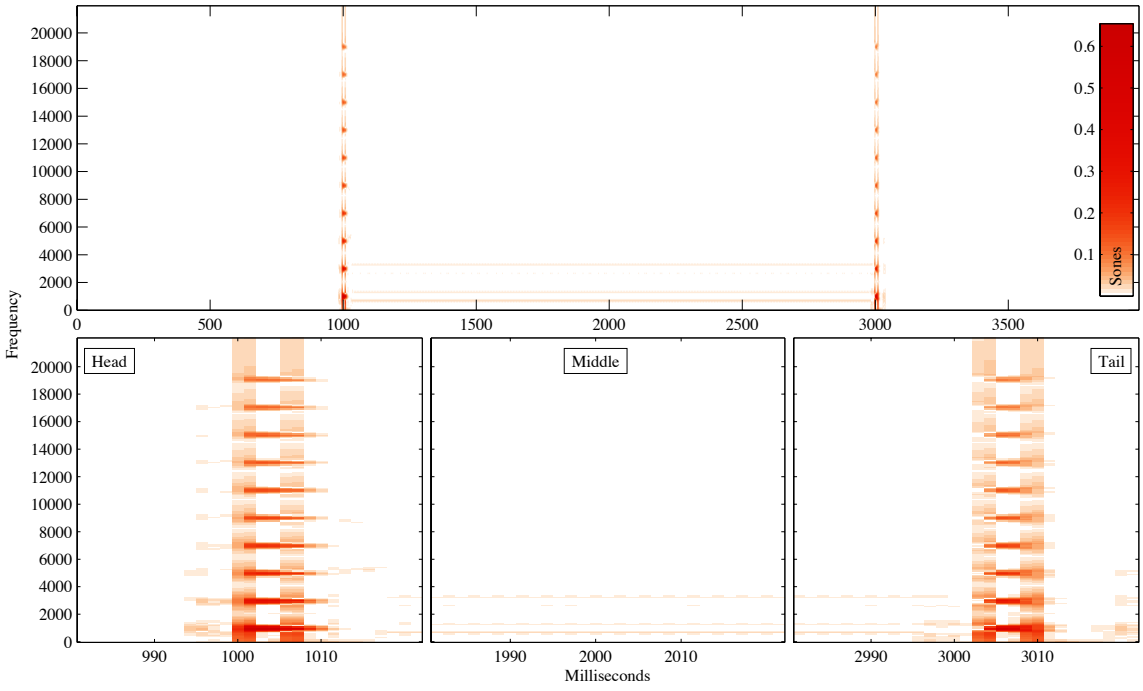

**Graph 7.3.2.2 –** Square Lock Error Spectrogram



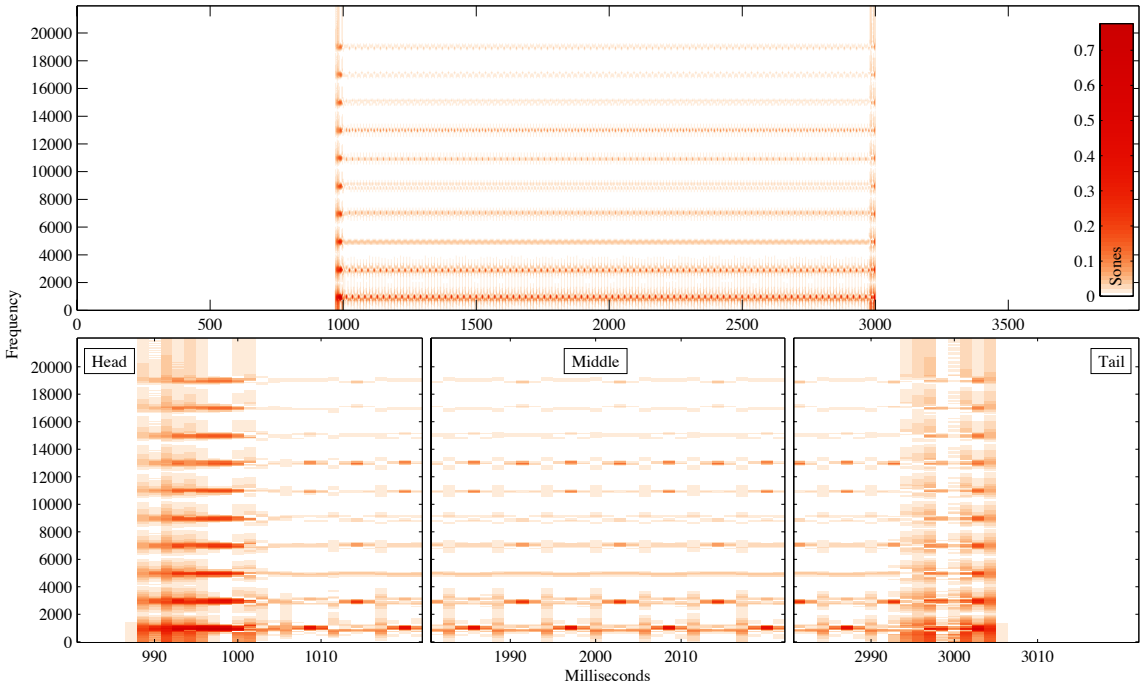
**Graph 7.3.2.3 –** Square Peak Error Spectrogram



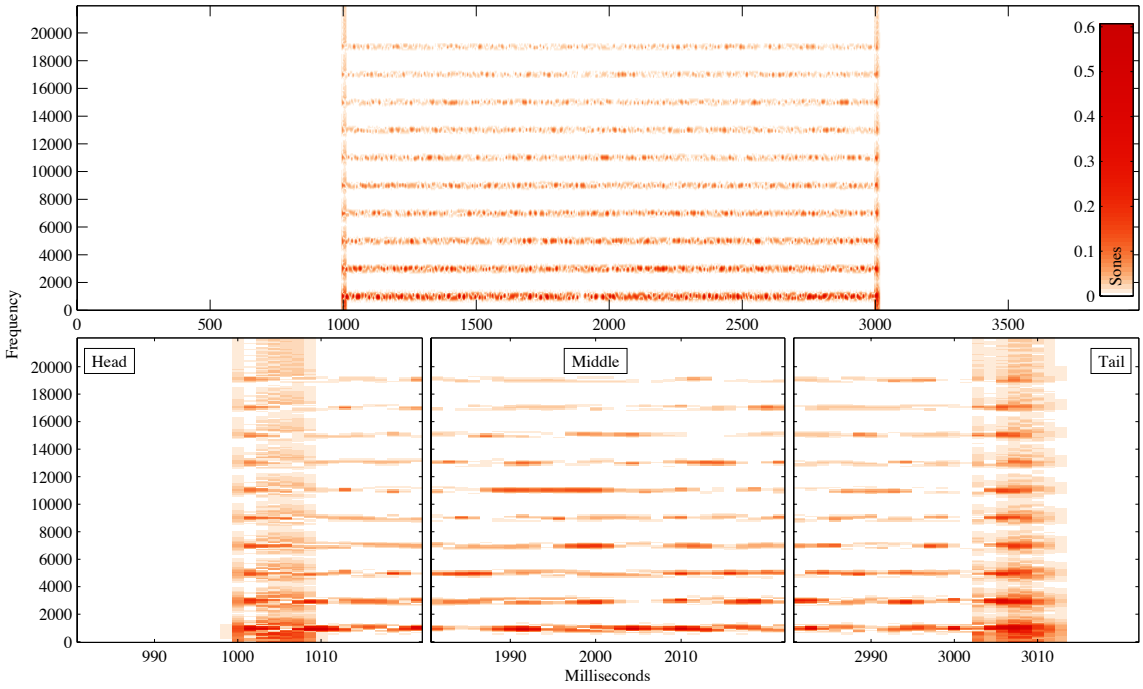

**Graph 7.3.2.4 –** Square Bank Error Spectrogram



**Graph 7.3.2.5 –** Square Sola Error Spectrogram

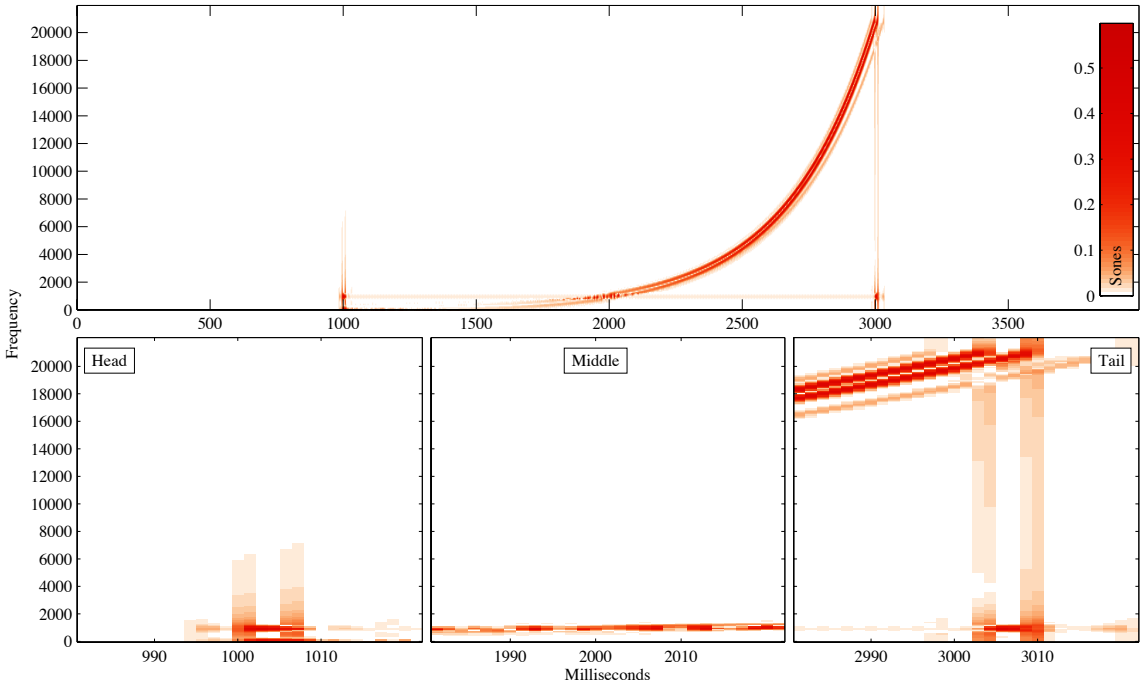



**Graph 7.3.2.6 –** Square Ola Error Spectrogram

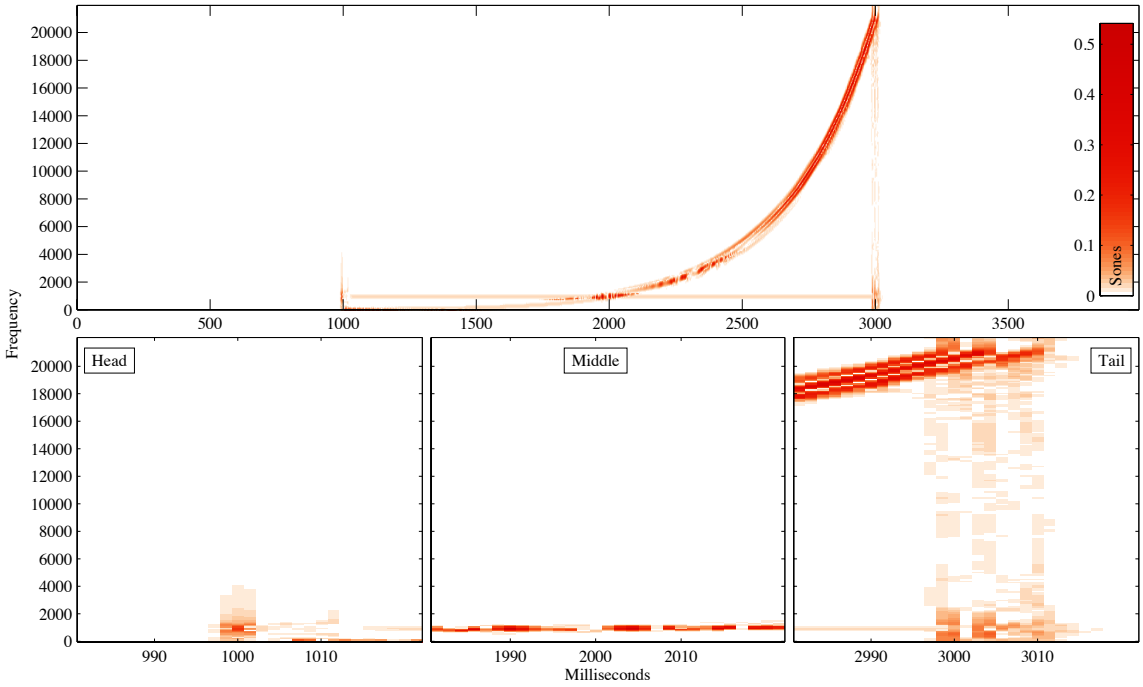


### 7.3.3 – Sweep Error Spectrograms

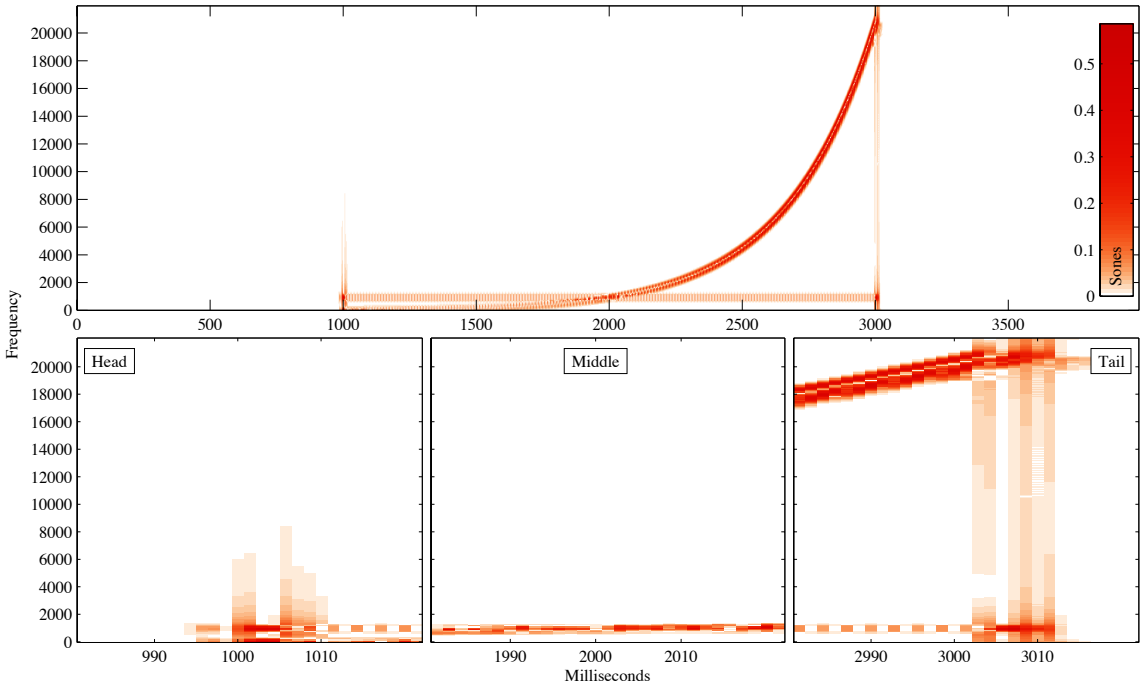
**Graph 7.3.3.1 –** Sweep Classic Error Spectrogram



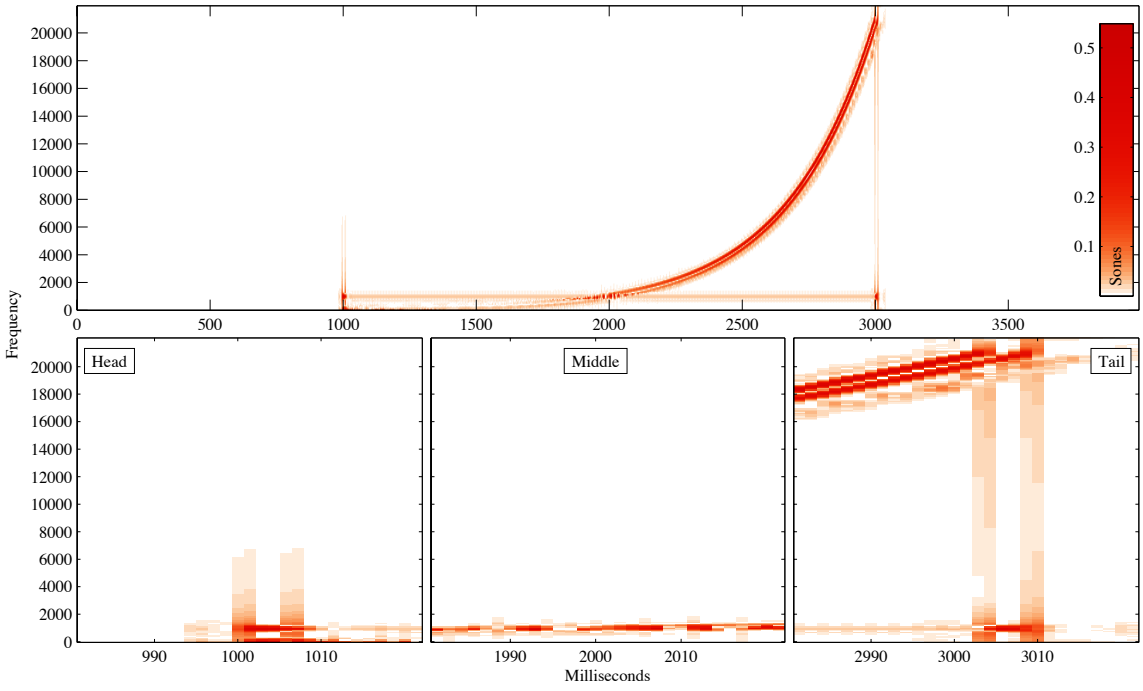

**Graph 7.3.3.2 –** Sweep Lock Error Spectrogram



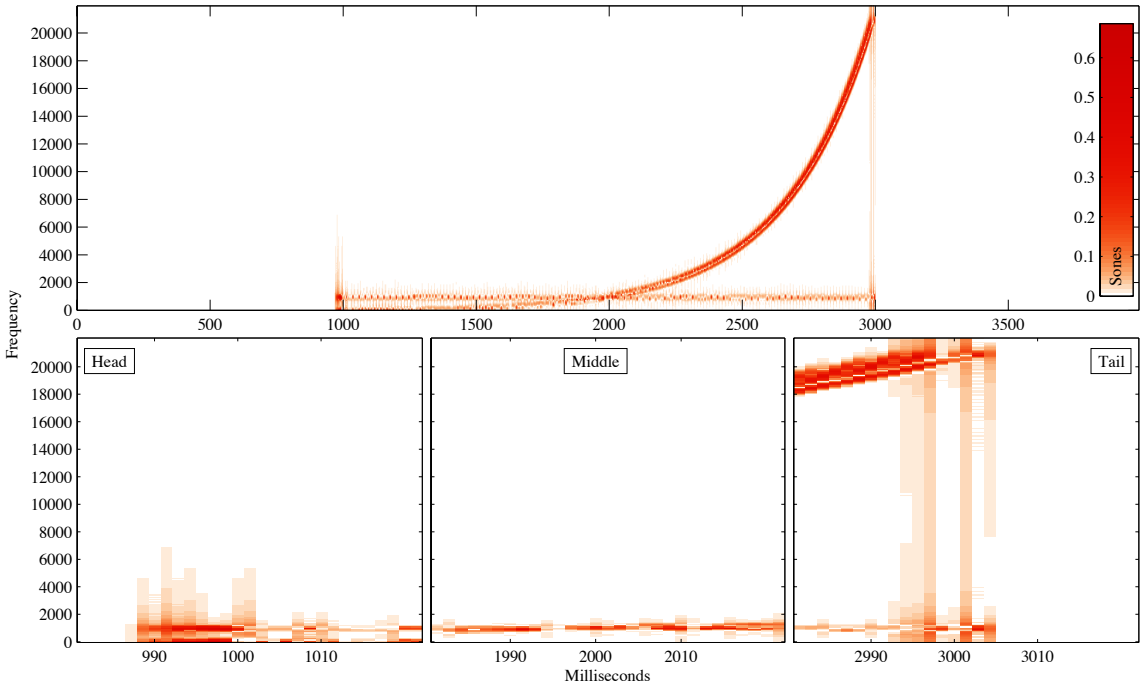
**Graph 7.3.3.3 –** Sweep Peak Error Spectrogram



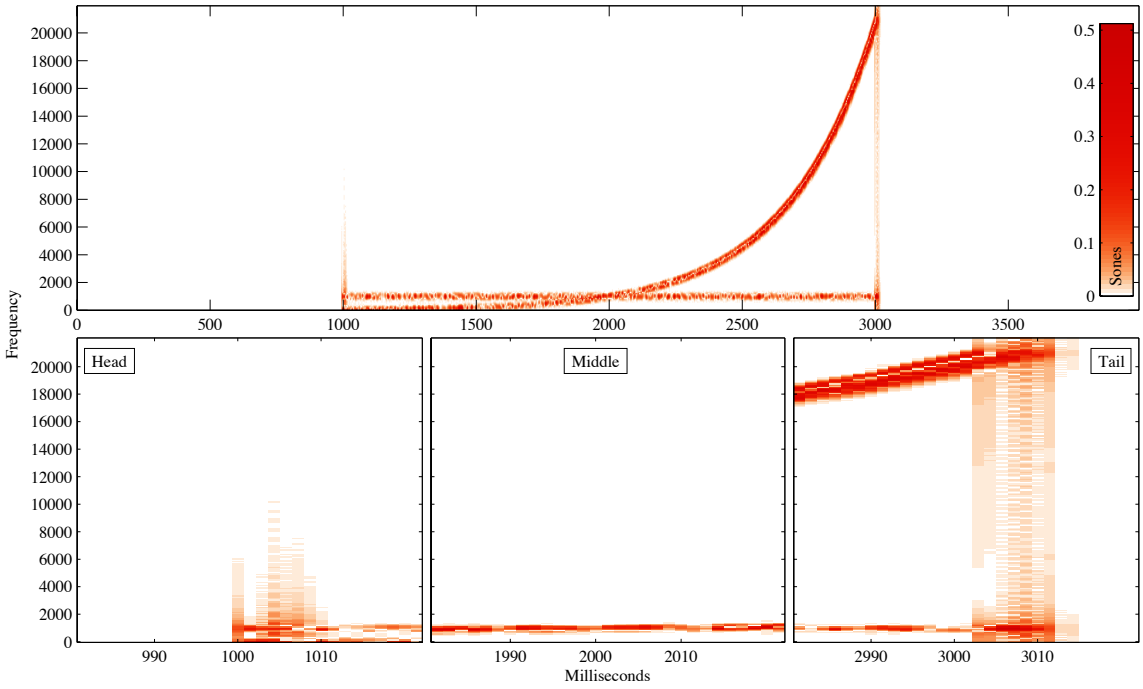

**Graph 7.3.3.4 –** Sweep Bank Error Spectrogram



**Graph 7.3.3.5 –** Sweep Sola Error Spectrogram

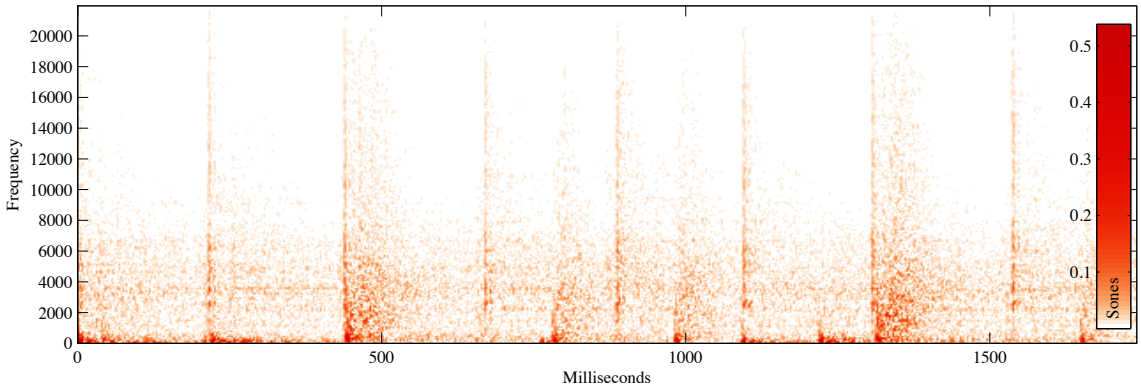



**Graph 7.3.3.6 –** Sweep Ola Error Spectrogram

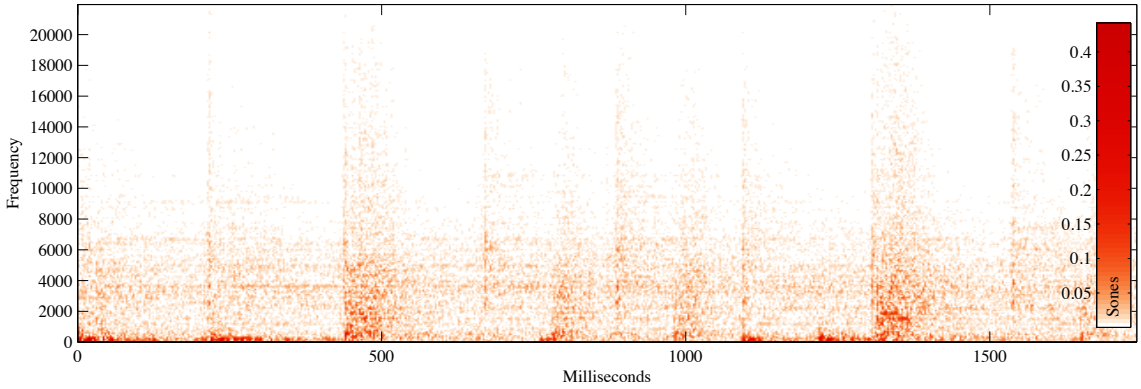


### 7.3.4 – *Amen* Error Spectrograms

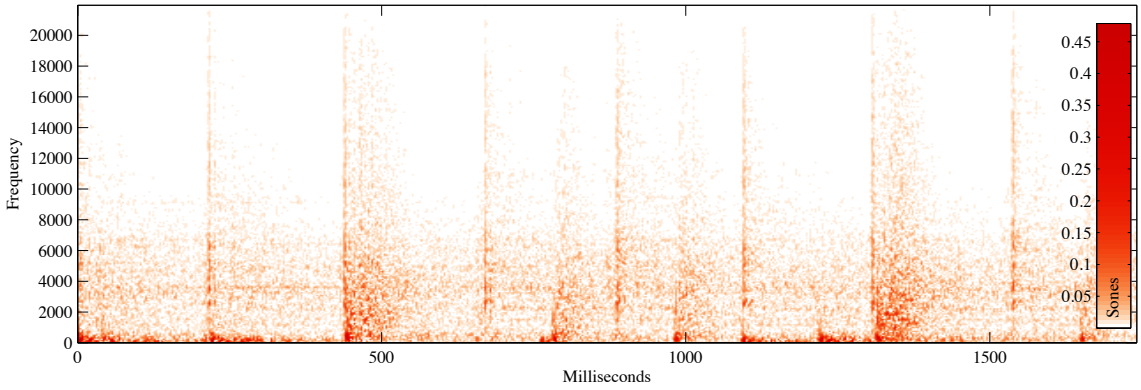
**Graph 7.3.4.1 –** *Amen* Classic Error Spectrogram



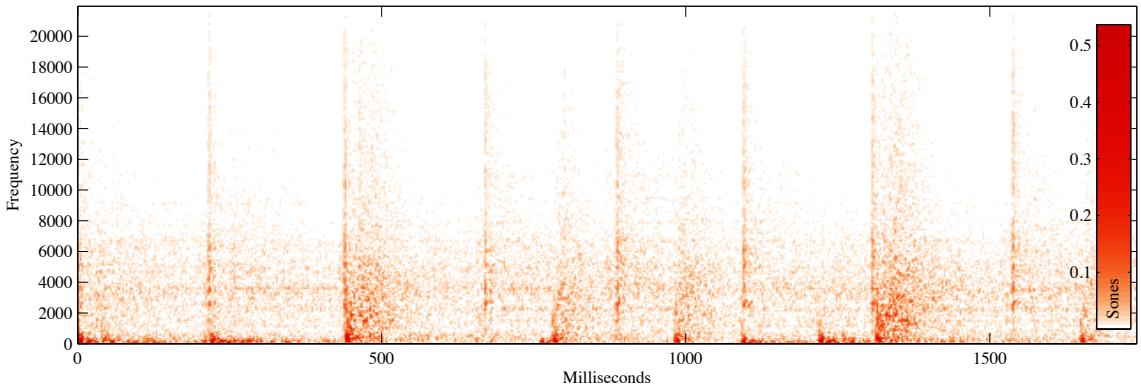

**Graph 7.3.4.2 –** *Amen* Lock Error Spectrogram



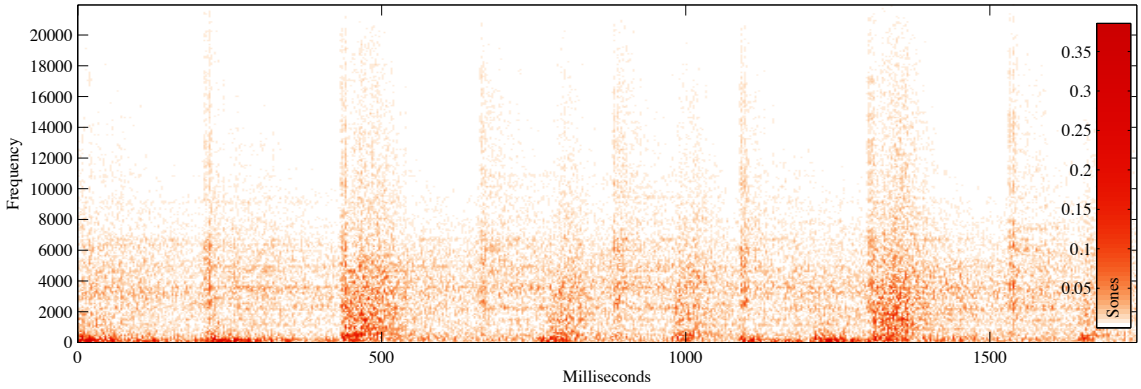

**Graph 7.3.4.3 –** *Amen* Peak Error Spectrogram



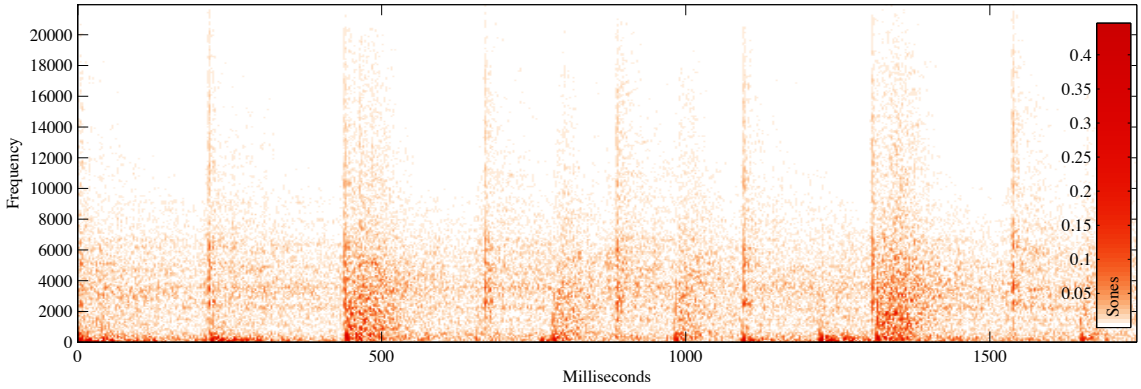
**Graph 7.3.4.4 –** *Amen* Bank Error Spectrogram




**Graph 7.3.4.5 –** *Amen* Sola Error Spectrogram

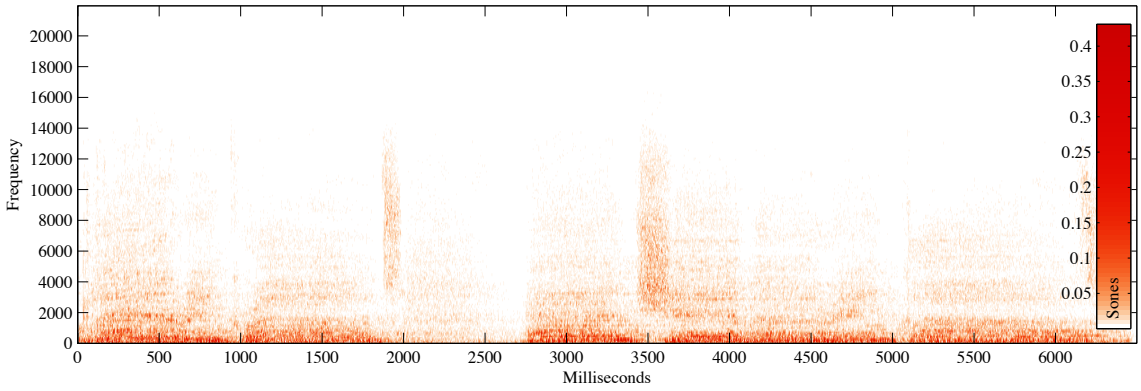



**Graph 7.3.4.6 –** *Amen* Ola Error Spectrogram

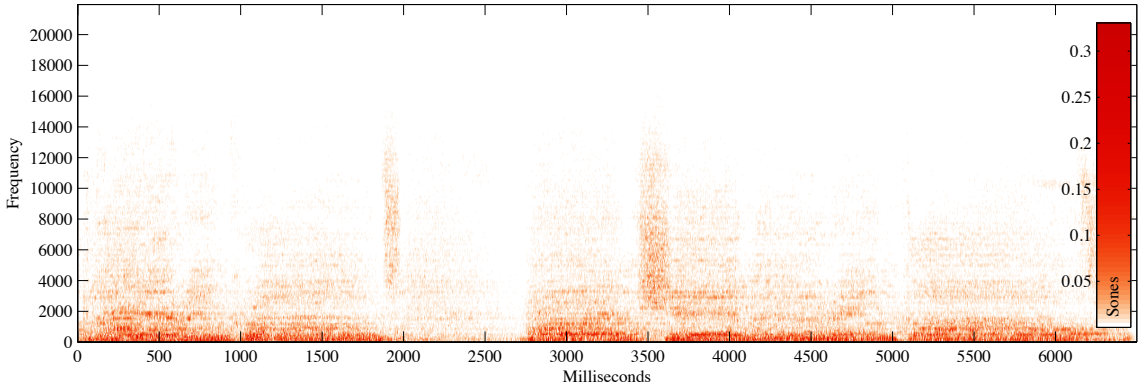


### 7.3.5 – *Autumn* Error Spectrograms

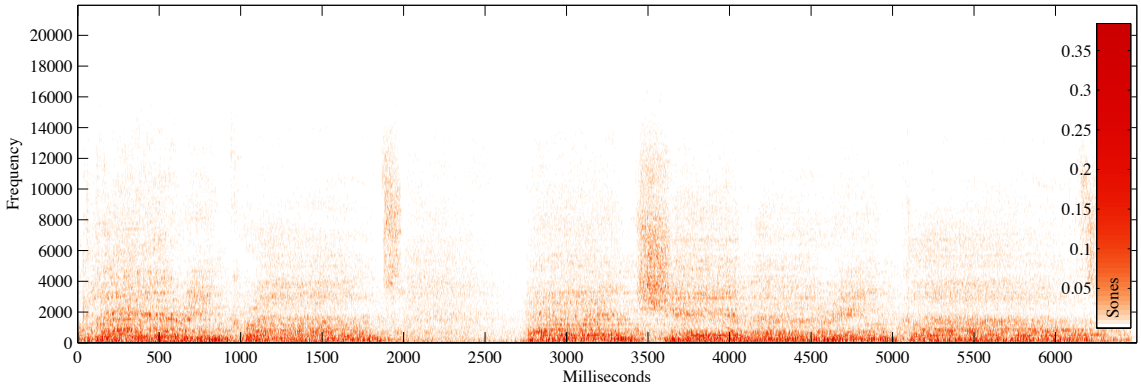
**Graph 7.3.5.1 –** *Autumn* Classic Error Spectrogram



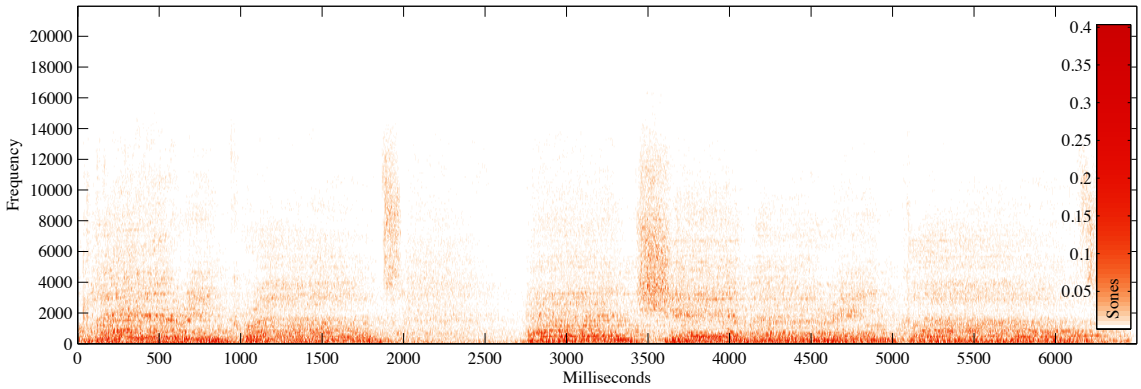

**Graph 7.3.5.2 –** *Autumn* Lock Error Spectrogram



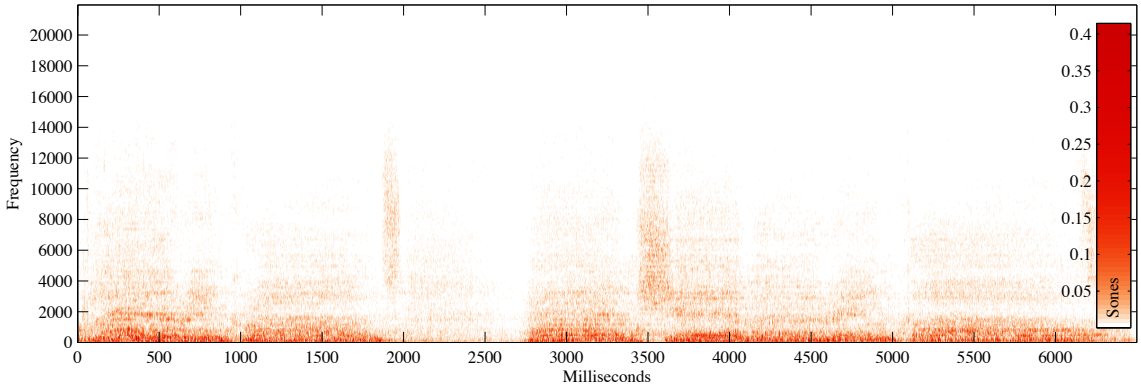

**Graph 7.3.5.3 –** *Autumn* Peak Error Spectrogram



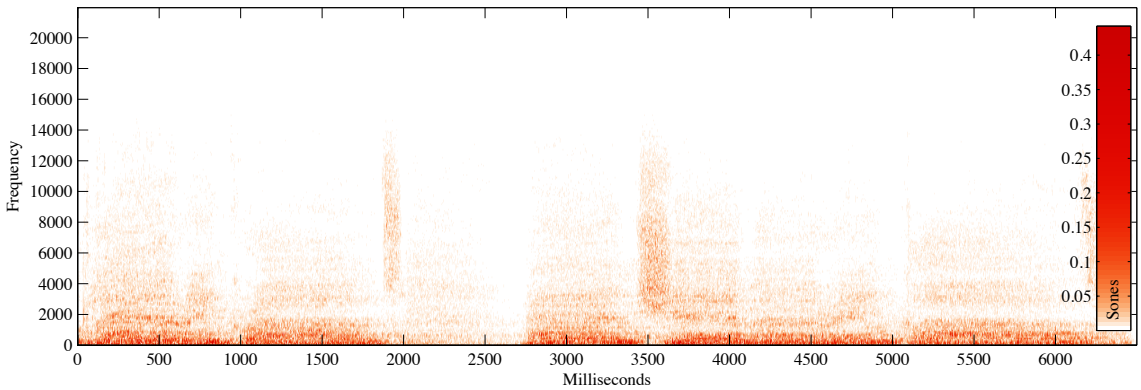
**Graph 7.3.5.4 –** *Autumn* Bank Error Spectrogram




**Graph 7.3.5.5 –** *Autumn* Sola Error Spectrogram

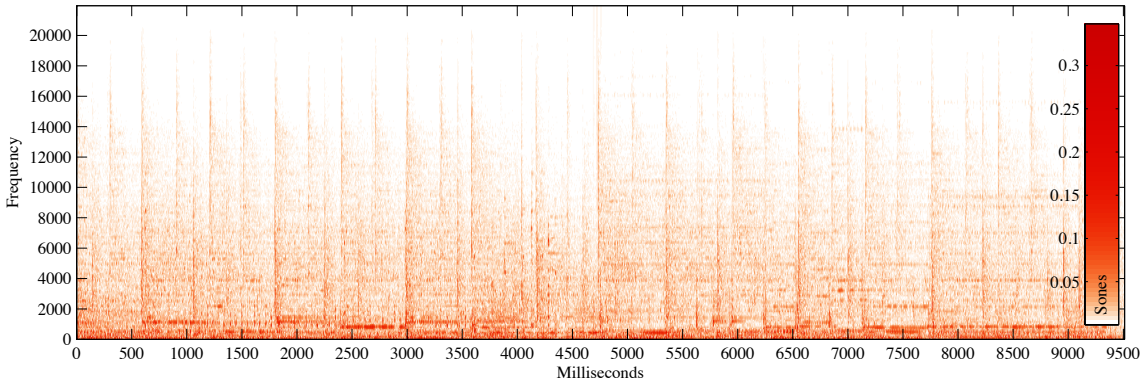



**Graph 7.3.5.6 –** *Autumn* Ola Error Spectrogram

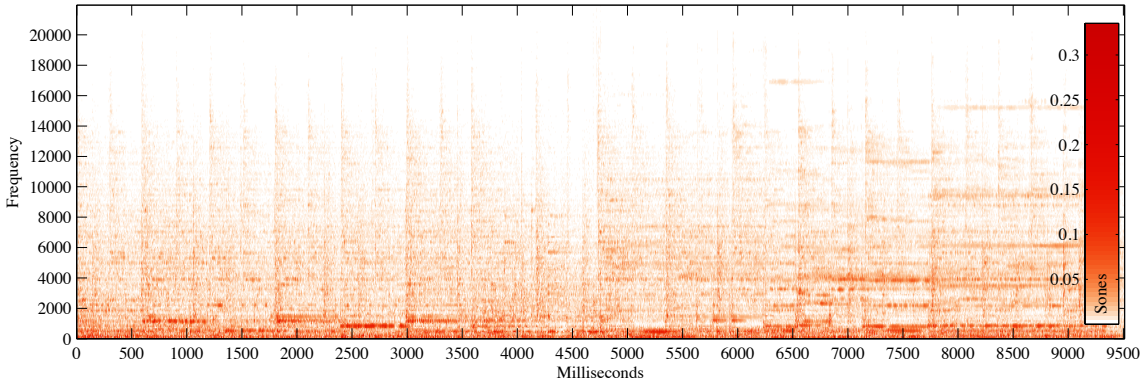


### 7.3.6 – *Peaches* Error Spectrograms

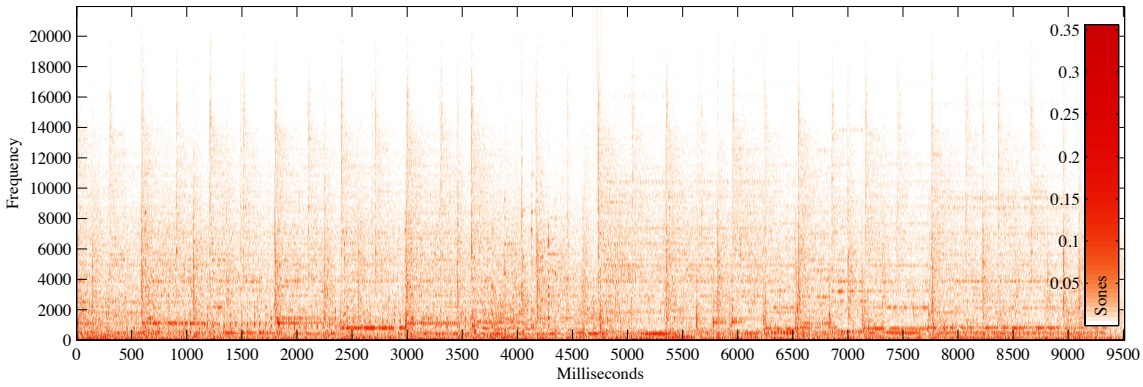
**Graph 7.3.6.1 –** *Peaches* Classic Error Spectrogram



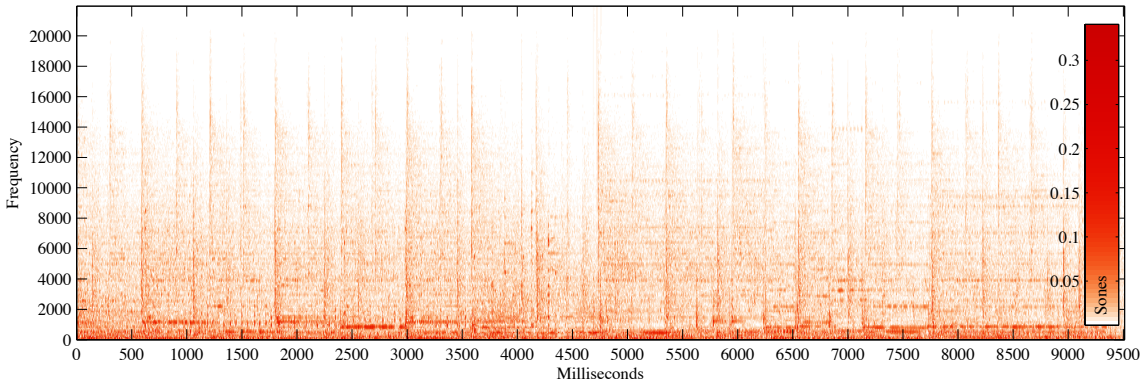

**Graph 7.3.6.2 –** *Peaches* Lock Error Spectrogram



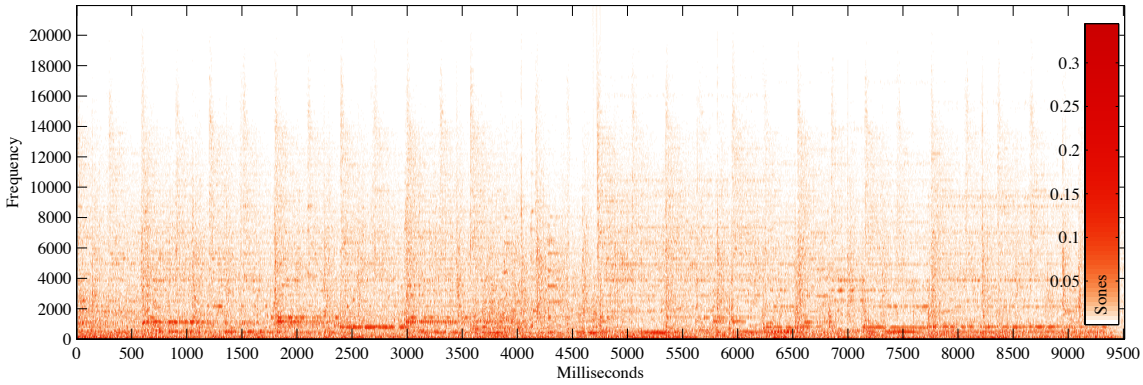

**Graph 7.3.6.3 –** *Peaches* Peak Error Spectrogram



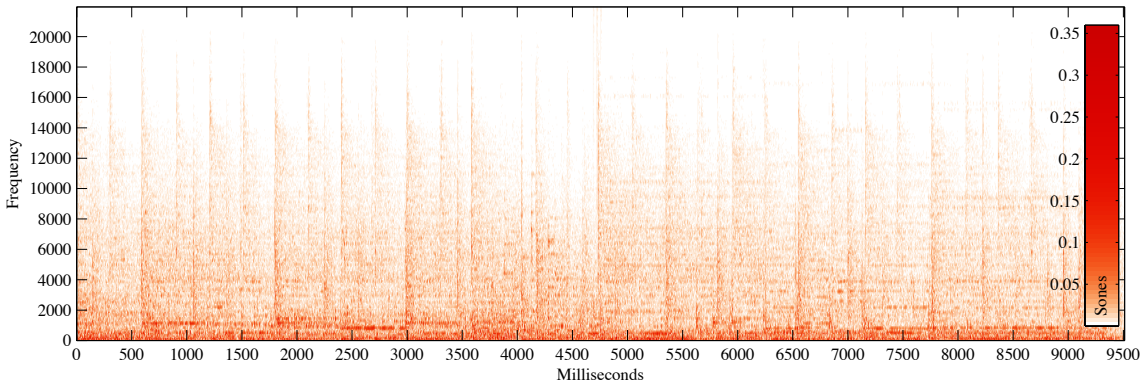
**Graph 7.3.6.4 –** *Peaches* Bank Error Spectrogram




**Graph 7.3.6.5 –** *Peaches* Sola Error Spectrogram




**Graph 7.3.6.6 –** *Peaches* Ola Error Spectrogram



# 8 – Appendix B

## 8.1 – Waveform Scripts

### 8.1.1 – Sine

```
%-------------------------------------------------------------------------
% 2 sec 1000Hz zero-padded sine tone
% .5 sec silence + 1 sec tone + .5 sec silence
%
% Cooper Baker - 2014
%-------------------------------------------------------------------------

% clean up
close all;
clear;

% Initializations
%-------------------------------------------------------------------------
sampRate  = 44100;
freq      = 1000;
zeroSec   = 0.5;
toneSec   = 1;
outFile   = 'sine.wav';
toneSamps = toneSec * sampRate;
zeroSamps = zeroSec * sampRate;

% Synthesis
%-------------------------------------------------------------------------
% generate a vector to synthesize the waveform
toneVec  = [ 0 : 1 : toneSamps - 1 ];

% generate waveform and silence
tone = sin( 2 * pi * ( freq / sampRate ) * toneVec );
zero = zeros( 1, zeroSamps );

% concatenate waveform and silence into output vector
output = [ zero( 1, 1 : zeroSamps ), tone( 1, 1 : toneSamps ), zero( 1, 1 :
zeroSamps ) ];

% Output
%-------------------------------------------------------------------------
% create comment string
commentString = sprintf( 'Frequency: %.0f\nWaveform: %s', freq, 'sine' );

% write file to disk
audiowrite( outFile, output, sampRate, 'BitsPerSample', 32, 'Artist',
'NormCoeff', 'Title', num2str( 1 ), 'Comment', commentString );

% EOF
```

### 8.1.2 – Sine Ideal

```
%-------------------------------------------------------------------------
% 4 second 1000Hz zero-padded sine tone
% 1 sec silence + 2 sec tone + 1 sec silence
%
% Cooper Baker - 2014
%-------------------------------------------------------------------------

% clean up
close all;
clear;

% Initializations
%-------------------------------------------------------------------------
sampRate  = 44100;
freq      = 1000;
zeroSec   = 1;
toneSec   = 2;
outFile   = 'sine_ideal.wav';
toneSamps = toneSec * sampRate;
zeroSamps = zeroSec * sampRate;

% Synthesis
%-------------------------------------------------------------------------

% generate a vector to synthesize the waveform
toneVec  = [ 0 : 1 : toneSamps - 1 ];

% generate waveform and silence
tone = sin( 2 * pi * ( freq / sampRate ) * toneVec );
zero = zeros( 1, zeroSamps );

% concatenate waveform and silence into output vector
output = [ zero( 1, 1 : zeroSamps ), tone( 1, 1 : toneSamps ), zero( 1, 1 :
zeroSamps ) ];

% Output
%-------------------------------------------------------------------------

% create comment string
commentString = sprintf( 'Frequency: %.0f\nWaveform: %s', freq, 'sine' );

% write file to disk
audiowrite( outFile, output, sampRate, 'BitsPerSample', 32, 'Artist',
'NormCoeff', 'Title', num2str( 1 ), 'Comment', commentString );

% EOF
```

### 8.1.3 – Square

```
%-------------------------------------------------------------------------
% 2 second 1000Hz zero-padded band-limited square tone
% .5 sec silence + 1 sec tone + .5 sec silence
%
% Cooper Baker - 2014
%-------------------------------------------------------------------------

% clean up
close all;
clear;

% Initializations
%-------------------------------------------------------------------------
sampRate  = 44100;
freq      = 1000;
zeroSec   = 0.5;
toneSec   = 1;
harmonics = 10;
harmNum   = 1;
outFile   = 'square.wav';
toneSamps = toneSec * sampRate;
zeroSamps = zeroSec * sampRate;
tone      = zeros( 1, toneSamps );

% Synthesis
%-------------------------------------------------------------------------

% generate a vector to synthesize the waveform
toneVec  = [ 0 : 1 : toneSamps - 1 ];

% generate band-limited square wave
while harmNum <= ( harmonics * 2 )

    % synthesize harmonic
    harmonic = sin( 2 * pi * ( ( freq * harmNum ) / sampRate ) * toneVec ) /
harmNum;

    % mix harmonic into tone vector
    tone = tone( 1 : toneSamps ) + harmonic( 1 : toneSamps );

    % increment harmonic number
    harmNum = harmNum + 2;
end

% generate silence
zero = zeros( 1, zeroSamps );

% concatenate waveform and silence into output vector
output = [ zero( 1, 1 : zeroSamps ), tone( 1, 1 : toneSamps ), zero( 1, 1 :
zeroSamps ) ];
```

```
% normalize between -1 and 1
output = output( 1 : end ) / max ( abs ( output ) );

% Output
%-------------------------------------------------------------------------

% create comment string
commentString = sprintf( 'Frequency: %.0f\nWaveform: %s', freq, 'square' );

% write file to disk
audiowrite( outFile, output, sampRate, 'BitsPerSample', 32, 'Artist',
'NormCoeff', 'Title', num2str( 1 ), 'Comment', commentString );

% EOF
```

### 8.1.4 – Square Ideal

```
%-------------------------------------------------------------------------
% 4 second 1000Hz zero-padded band-limited square tone
% 1 sec silence + 2 sec tone + 1 sec silence
%
% Cooper Baker - 2014
%-------------------------------------------------------------------------

% clean up
close all;
clear;

% Initializations
%-------------------------------------------------------------------------
sampRate  = 44100;
freq      = 1000;
zeroSec   = 1;
toneSec   = 2;
harmonics = 10;
harmNum   = 1;
outFile   = 'square_ideal.wav';
toneSamps = toneSec * sampRate;
zeroSamps = zeroSec * sampRate;
tone      = zeros( 1, toneSamps );

% Synthesis
%-------------------------------------------------------------------------

% generate a vector to synthesize the waveform
toneVec  = [ 0 : 1 : toneSamps - 1 ];

% generate band-limited square wave
while harmNum <= ( harmonics * 2 )
```

```
    % synthesize harmonic
    harmonic = sin( 2 * pi * ( ( freq * harmNum ) / sampRate ) * toneVec ) /
harmNum;

    % mix harmonic into tone vector
    tone = tone( 1 : toneSamps ) + harmonic( 1 : toneSamps );

    % increment harmonic number
    harmNum = harmNum + 2;
end

% generate silence
zero = zeros( 1, zeroSamps );

% concatenate waveform and silence into output vector
output = [ zero( 1, 1 : zeroSamps ), tone( 1, 1 : toneSamps ), zero( 1, 1 :
zeroSamps ) ];

% normalize between -1 and 1
output = output( 1 : end ) / max ( abs ( output ) );

% Output
%--------------------------------------------------------------------------

% create comment string
commentString = sprintf( 'Frequency: %.0f\nWaveform: %s', freq, 'square' );

% write file to disk
audiowrite( outFile, output, sampRate, 'BitsPerSample', 32, 'Artist',
'NormCoeff', 'Title', num2str( 1 ), 'Comment', commentString );

% EOF
```

### 8.1.5 – Sweep

```
%--------------------------------------------------------------------------
% 4 second 1000Hz zero-padded band-limited square tone
% 1 sec silence + 2 sec tone + 1 sec silence
%
% Cooper Baker - 2014
%--------------------------------------------------------------------------

% clean up
close all;
clear;

% Initializations
%--------------------------------------------------------------------------
sampRate  = 44100;
freq      = 1000;
zeroSec   = 1;
toneSec   = 2;
```

```
harmonics = 10;
harmNum   = 1;
outFile   = 'square_ideal.wav';
toneSamps = toneSec * sampRate;
zeroSamps = zeroSec * sampRate;
tone      = zeros( 1, toneSamps );

% Synthesis
%-------------------------------------------------------------------------

% generate a vector to synthesize the waveform
toneVec  = [ 0 : 1 : toneSamps - 1 ];

% generate band-limited square wave
while harmNum <= ( harmonics * 2 )

    % synthesize harmonic
    harmonic = sin( 2 * pi * ( ( freq * harmNum ) / sampRate ) * toneVec ) /
harmNum;

    % mix harmonic into tone vector
    tone = tone( 1 : toneSamps ) + harmonic( 1 : toneSamps );

    % increment harmonic number
    harmNum = harmNum + 2;
end

% generate silence
zero = zeros( 1, zeroSamps );

% concatenate waveform and silence into output vector
output = [ zero( 1, 1 : zeroSamps ), tone( 1, 1 : toneSamps ), zero( 1, 1 :
zeroSamps ) ];

% normalize between -1 and 1
output = output( 1 : end ) / max ( abs ( output ) );

% Output
%-------------------------------------------------------------------------

% create comment string
commentString = sprintf( 'Frequency: %.0f\nWaveform: %s', freq, 'square' );

% write file to disk
audiowrite( outFile, output, sampRate, 'BitsPerSample', 32, 'Artist',
'NormCoeff', 'Title', num2str( 1 ), 'Comment', commentString );

% EOF
```

### 8.1.6 – Sweep Ideal

```
%-------------------------------------------------------------------------
% 4 second zero-padded 1kHz sine + logarithmic sine sweep tone
% 1 sec silence + 2 sec 20Hz to 20kHz tone + 1 sec silence
%
% Cooper Baker - 2014
%-------------------------------------------------------------------------

% clean up
close all;
clear;

% Initializations
%-------------------------------------------------------------------------
sampRate  = 44100;
freq      = 1000;
zeroSec   = 1;
toneSec   = 2;
outFile   = 'sweep_ideal.wav';
toneSamps = toneSec * sampRate;
zeroSamps = zeroSec * sampRate;

% Synthesis
%-------------------------------------------------------------------------

% generate the synthesis vectors
chirpVec = [ 0 : toneSec / (toneSamps - 1) : toneSec ];
sineVec  = [ 0 : 1 : toneSamps - 1 ];

% synthesize the sweep and sine
sweep = chirp( chirpVec, 50, toneSec, 21000, 'logarithmic', 270 );
sine  = sin( 2 * pi * ( freq / sampRate ) * sineVec );

% mix sweep and sine tones
tone = sweep + sine;

% generate silence
zero = zeros( 1, zeroSamps );

% concatenate waveform and silence into output vector
output = [ zero( 1, 1 : zeroSamps ), tone( 1, 1 : toneSamps ), zero( 1, 1 :
zeroSamps ) ];

% normalize between -1 and 1
output = output( 1 : end ) / max ( abs ( output ) );

% Output
%-------------------------------------------------------------------------
```

```
% create comment string
commentString = sprintf( 'Frequency: %s\nWaveform: %s', '50Hz-21kHz', 'sine +
log sweep' );

% write file to disk
audiowrite( outFile, output, sampRate, 'BitsPerSample', 32, 'Artist',
'NormCoeff', 'Title', num2str( 1 ), 'Comment', commentString );

% EOF
```

### 8.2 – Phase Vocoder Scripts

#### 8.2.1 – FFT IFFT Classic

```
function fft_ifft_classic( filePath, fileName )
%-------------------------------------------------------------------------
% FFT to IFFT Phase Vocoder
%
% adapted from
% VX_tstretch_real_pv.m   [DAFXbook, 2nd ed., chapter 7]
%
% Cooper Baker - 2014
%-------------------------------------------------------------------------

close all;

% Settings
%-------------------------------------------------------------------------
windowSize    = 1024;
overlap       = 4;
stretchFactor = 2;
window        = hann( windowSize, 'periodic' );
tag           = 'classic';

% Initializations
%-------------------------------------------------------------------------
if any( exist( 'fileName' ) ~= 1 )
    [ fileName, filePath ] = uigetfile( '*.wav', 'Audio File' );
end

[ input, sr ] = audioread( [ filePath, fileName ] );
hopSize       = windowSize / overlap;
sampleHopSize = hopSize / stretchFactor;
input         = [ zeros( windowSize, 1 ) ; input ; zeros( windowSize - mod(
length( input ), sampleHopSize ), 1 ) ];
output        = zeros( windowSize + ceil( length( input ) * stretchFactor ), 1
);
omega         = 2 * pi * sampleHopSize * [ 0 : windowSize - 1 ]' / windowSize;
phaseOld      = zeros( windowSize, 1 );
phaseAccum    = zeros( windowSize, 1 );
sampleIndex   = 0;
```

```
frameIndex    = 0;
sampleMax     = length( input ) - windowSize;

% create progress bar dialog box
bar = waitbar( 0, '0%', 'Name', sprintf( '%s: processing %s...', mfilename,
fileName ) );

% Processing Loop
%--------------------------------------------------------------------------
while sampleIndex < sampleMax
    % copy and window the input frame
    frame = input( sampleIndex + 1 : sampleIndex + windowSize) .* window;

    % shift zero frequency component to center of spectrum
    frame = fftshift( frame );

    % perform an fft on the input frame
    spect = fft( frame );

    % cartesian to polar conversion
    mag   = abs  ( spect );
    phase = angle( spect );

    % subtract expected phase procession and compute phase delta
    phaseWrap = phase - phaseOld - omega;
    phaseWrap = mod( phaseWrap + pi, -2 * pi ) + pi;

    % add expected phase procession
    phaseDelta = omega + phaseWrap;
    phaseOld  = phase;

    % apply stretch factor to phase and compute accumulated phase
    phaseAccum = phaseAccum + phaseDelta * stretchFactor;
    phaseAccum = mod( phaseAccum + pi, -2 * pi ) + pi;

    % polar to cartesian conversion
    spect = ( mag .* exp(     * phaseAccum ) );

    % perform an ifft on the spectrum
    frame = ifft( spect );

    % discard imaginary data
    frame = real( frame );

    % shift zero frequency component to center of spectrum
    frame = fftshift( frame );

    % apply the window
    frame = frame .* window;

    % normalize the frame
    frame = frame / windowSize;
```

```
    % overlap-add frame into output buffer
    output( frameIndex + 1 : frameIndex + windowSize ) = output( frameIndex + 1
: frameIndex + windowSize ) + frame;

    % increment indices
    sampleIndex = sampleIndex + sampleHopSize;
    frameIndex  = frameIndex  + hopSize;

     % update the progress bar
    progress = ( sampleIndex / sampleMax );
    waitbar( progress, bar, sprintf( '%2.3f%%', progress * 100 ) )
end

% close progress bar dialog box
close( bar );

% Output
%--------------------------------------------------------------------------

% crop output buffer
output = output( windowSize + 1 : length( output ) );

% normalize output buffer
normCoeff = 1 / max( abs( output ) );
output    = output * normCoeff;

% plot output
timevector = linspace( 0, length( output ) / sr, length( output ) );
plot( timevector, output );

% create comment string
commentString = sprintf( 'WinSize: %.0f,\nOverlap: %.0f,\nStretch:
%.2f,\nWindow: %s', windowSize, overlap, stretchFactor, 'Hann' );

% write audio file to disk
[ a, fileName, b ] = fileparts( fileName );
outFile = sprintf( '%s.%s.wav', fileName, tag );
audiowrite( outFile, output, sr, 'BitsPerSample', 32, 'Artist', 'NormCoeff',
'Title', num2str( normCoeff ), 'Comment', commentString );

% EOF
```

### 8.2.2 – FFT IFFT Lock

```
function fft_ifft_lock( filePath, fileName )
%--------------------------------------------------------------------------
% FFT to IFFT Phase-Locked Vocoder
%
% Cooper Baker - 2014
%--------------------------------------------------------------------------

close all;
```

```
% Settings
%------------------------------------------------------------------------
windowSize    = 1024;
overlap       = 4;
stretchFactor = 2;
window        = hann( windowSize, 'periodic' );
tag           = 'lock';

% Initializations
%------------------------------------------------------------------------
if any( exist( 'fileName' ) ~= 1 )
    [ fileName, filePath ] = uigetfile( '*.wav', 'Audio File' );
end

[ input, sr ] = audioread( [ filePath, fileName ] );
hopSize       = windowSize / overlap;
sampleHopSize = hopSize / stretchFactor;
input         = [ zeros( windowSize, 1) ; input ; zeros( windowSize - mod(
length( input ), windowSize ), 1 ) ];
output        = zeros( windowSize + length( input ) * stretchFactor, 1 );
frameIndex    = 1;
frameMax      = length( output ) - windowSize;
sampleIndex   = 1;
sampleMax     = length( input ) - windowSize - hopSize;
spec          = zeros( windowSize, 1 );

% create progress bar dialog box
bar = waitbar( 0, '0%', 'Name', sprintf( '%s: processing %s...', mfilename,
fileName ) );

% Processing Loop
%------------------------------------------------------------------------
while sampleIndex < sampleMax;

    % copy and window the input frames
    frameBack  = input( sampleIndex + 1           : sampleIndex + windowSize
) .* window;
    frameFront = input( sampleIndex + 1 + hopSize : sampleIndex + windowSize +
hopSize ) .* window;

    % perform an fft on the input frame
    specBack  = fft( frameBack );
    specFront = fft( frameFront );

    % save previous spec and add salt
    specOld = spec + realmin;

    % calculate phase accumulation
    specPhase = ( conj( specBack ) .* specOld ) ./ abs( specOld );
```

```
    % calculate phase-locked spectrum
    specPhaseLock = circshift( specPhase, 1 ) + specPhase + circshift(
specPhase, -1 );

    % perform phase vocoding

    % phase lock
    spec = ( specPhaseLock .* specFront ) ./ abs( specPhaseLock + realmin );

    % no phase lock
    % spec = ( specPhase .* specFront ) ./ abs( specPhase + realmin );

    % perform an ifft on the spectrum
    frame = ifft( spec );

    % discard imaginary data
    frame = real( frame );

    % apply the window
    frame = frame .* window;

    % overlap-add the frame into the output buffer
    output( frameIndex + 1 : frameIndex + windowSize ) = output( frameIndex + 1
: frameIndex + windowSize ) + frame;

    % increment location indices
    sampleIndex = sampleIndex + sampleHopSize;
    frameIndex  = frameIndex  + hopSize;

    % update the progress bar
    progress = ( sampleIndex / sampleMax );
    waitbar( progress, bar, sprintf( '%2.3f%%', progress * 100 ) )
end

% close progress bar dialog box
close( bar );

% Output
%-------------------------------------------------------------------------

% crop output buffer
output = output( windowSize + 1 : length( output ) );

% normalize output buffer
normCoeff = 1 / max( abs( output ) );
output    = output * normCoeff;

% plot output
timevector = linspace( 0, length( output )/sr, length( output ) );
plot( timevector, output );
```

```
% create comment string
commentString = sprintf( 'WinSize: %.0f,\nOverlap: %.0f,\nStretch:
%.2f,\nWindow: %s', windowSize, overlap, stretchFactor, 'Hann' );

% write audio file to disk
[ a, fileName, b ] = fileparts( fileName );
outFile = sprintf( '%s.%s.wav', fileName, tag );
audiowrite( outFile, output, sr, 'BitsPerSample', 32, 'Artist', 'NormCoeff',
'Title', num2str( normCoeff ), 'Comment', commentString );

% EOF
```

### 8.2.3.1 – FFT IFFT Peak

```
function fft_ifft_peak( filePath, fileName )
%-------------------------------------------------------------------------
% FFT to IFFT Peak Tracking Phase Locked Vocoder
%
% adapted from:
% VX_tstretch_real_pv_phaselocked.m   [DAFXbook, 2nd ed., chapter 7]
%
% Cooper Baker - 2014
%-------------------------------------------------------------------------

close all;

% Settings
%-------------------------------------------------------------------------
windowSize    = 1024;
overlap       = 4;
stretchFactor = 2;
window        = hann( windowSize, 'periodic' );
tag           = 'peak';

% Initializations
%-------------------------------------------------------------------------
if any( exist( 'fileName' ) ~= 1 )
    [ fileName, filePath ] = uigetfile( '*.wav', 'Audio File' );
end

analysisHop   = windowSize / overlap;
synthHop      = analysisHop * stretchFactor;
[ input, sr ] = audioread( [ filePath, fileName ] );
halfWinSize   = windowSize / 2;
inputSize     = length( input );
input         = [ zeros( windowSize, 1 ) ; input ; zeros( windowSize - mod(
inputSize, analysisHop ), 1 ) ] / max( abs( input ) );
output        = zeros(windowSize + ceil( length( input ) * stretchFactor ), 1
);
omega         = 2 * pi * analysisHop * [ 0 : halfWinSize ]' / windowSize;
phi0          = zeros( halfWinSize + 1, 1 );
psi           = zeros( halfWinSize + 1, 1 );
```

```
psi2          = psi;
numPrevPeaks  = 0;
analysisIndex = 0;
analysisMax   = length( input ) - windowSize;
synthIndex    = 0;

% create progress bar dialog box
bar = waitbar( 0, '0%', 'Name', sprintf( '%s: processing %s...', mfilename,
fileName ) );

% Processing Loop
%--------------------------------------------------------------------------
while analysisIndex < analysisMax

    % copy and window the input frame
    frame = input( analysisIndex + 1 : analysisIndex + windowSize ) .* window;

    % perform fft and save relevant half of spectrum
    spec = fft( fftshift( frame ) );
    halfSpec = spec( 1 : halfWinSize + 1 );

    % cartesian to polar conversion
    mag   = abs  ( halfSpec );
    phase = angle( halfSpec );

    % find spectral peaks ( local maxima )
    peakLoc  = zeros( halfWinSize + 1, 1 );
    numPeaks = 0;

    for b = 3 : halfWinSize - 1
        if( mag( b ) > mag( b - 1 ) && mag( b ) > mag( b - 2 ) && mag( b ) >
mag( b + 1) && mag( b ) > mag( b + 2 ) )
            numPeaks = numPeaks + 1;
            peakLoc( numPeaks ) = b;
            b = b + 3;
        end
    end

    % propagate peak phases and compute spectral bin phases
    if( analysisIndex == 0 )
        psi = phase;
    elseif( numPeaks > 0 && numPrevPeaks > 0 )
        prevPeak = 1;

        for p = 1 : numPeaks

            % connect current peak to the previous closest peak
            peak2 = peakLoc( p );
            while( prevPeak < numPrevPeaks && abs( peak2 - prevPeakLoc(
prevPeak + 1 ) ) < abs( peak2 - prevPeakLoc( prevPeak ) ) )
                prevPeak = prevPeak + 1;
            end
```

```
            peak1 = prevPeakLoc( prevPeak );

            % propagate peak's phase assuming linear frequency
            % variation between connected peak1 and peak2
            % ( avgPeak is a 1-based indexing spectral bin )
            avgPeak           = ( peak1 + peak2 ) * 0.5;
            peakOmega         = 2 * pi * analysisHop * ( avgPeak - 1.0 ) /
windowSize;
            peakDeltaPhase    = peakOmega + mod( ( phase( peak2 ) - phi0( peak1
) - peakOmega ) + pi, -2 * pi ) + pi;
            peakTargetPhase   = mod( ( psi( peak1 ) + peakDeltaPhase *
stretchFactor ) + pi, -2 * pi ) + pi;
            peakPhaseRotation = mod( ( peakTargetPhase - phase( peak2 ) ) + pi,
-2 * pi ) + pi;

            % rotate phases of all bins around the current peak
            if( numPeaks == 1 )
                bin1 = 1;
                bin2 = halfWinSize + 1;
            elseif( p == 1 )
                bin1 = 1;
                bin2 = halfWinSize + 1;
            elseif( p == numPeaks )
                bin1 = round( ( peakLoc( p - 1 ) + peak2 ) * 0.5 );
                bin2 = halfWinSize + 1;
            else
                bin1 = round( ( peakLoc( p - 1 ) + peak2 ) * 0.5 ) + 1;
                bin2 = round( ( peakLoc( p + 1 ) + peak2 ) * 0.5 );
            end
                psi2( bin1 : bin2 ) = mod( ( phase( bin1 : bin2 ) +
peakPhaseRotation ) + pi, -2 * pi ) + pi;
        end
            psi = psi2;
    else
        deltaPhase = omega + mod( ( phase - phi0 - omega ) + pi, -2 * pi ) +
pi;
        psi = mod( ( psi + deltaPhase * stretchFactor ) + pi, -2 * pi ) + pi;
    end

    % polar to cartesian conversion
    spec = ( mag .* exp( 1i * psi ) );

    % reconstruct entire spectrum
    spec = [ spec( 1 : halfWinSize + 1 ) ; conj( spec( halfWinSize : -1 : 2 ) )
];

    % perform an ifft on the spectrum
    frame = ifft( spec );

    % discard imaginary data
    frame = real( frame );
```

```
    % shift zero frequency component to center of spectrum
    frame = fftshift( frame );

    % apply the window
    frame = frame .* window;

    % overlap-add the synthesized frame into the output buffer
    output( synthIndex + 1 : synthIndex + windowSize ) = output( synthIndex + 1
: synthIndex + windowSize ) + frame;

    % store values for next frame
    phi0          = phase;
    prevPeakLoc   = peakLoc;
    numPrevPeaks  = numPeaks;

    % increment analysis and synthesis indices
    analysisIndex = analysisIndex + analysisHop;
    synthIndex    = synthIndex + synthHop;

    % update the progress bar
    progress = ( analysisIndex / analysisMax );
    waitbar( progress, bar, sprintf( '%2.3f%%', progress * 100 ) );
end

% close progress bar dialog box
close( bar );

% Output
%--------------------------------------------------------------------------
% crop output buffer
output = output( windowSize + 1 : length( output ) );

% normalize output buffer
normCoeff = 1 / max( abs( output ) );
output    = output * normCoeff;

% plot output
timevector = linspace( 0, length( output )/sr, length( output ) );
plot( timevector, output );

% create comment string
commentString = sprintf( 'WinSize: %.0f,\nOverlap: %.0f,\nStretch:
%.2f,\nWindow: %s', windowSize, overlap, stretchFactor, 'Hann' );

% write audio file to disk
[ a, fileName, b ] = fileparts( fileName );
outFile = sprintf( '%s.%s.wav', fileName, tag );
audiowrite( outFile, output, sr, 'BitsPerSample', 32, 'Artist', 'NormCoeff',
'Title', num2str( normCoeff ), 'Comment', commentString );

% EOF
```

### 8.2.3.2 – FFT IFFT Peak Fixed

```
function fft_ifft_peak_fixed( filePath, fileName )
%-------------------------------------------------------------------------
% FFT to IFFT Peak Tracking Phase Locked Vocoder ( fixed )
%
% adapted from:
% VX_tstretch_real_pv_phaselocked.m   [DAFXbook, 2nd ed., chapter 7]
%
% Cooper Baker - 2014
%-------------------------------------------------------------------------

close all;

% Settings
%-------------------------------------------------------------------------
windowSize    = 1024;
overlap       = 4;
stretchFactor = 2;
window        = hann( windowSize, 'periodic' );
tag           = 'peak_fixed';

% Initializations
%-------------------------------------------------------------------------
if any( exist( 'fileName' ) ~= 1 )
    [ fileName, filePath ] = uigetfile( '*.wav', 'Audio File' );
end

analysisHop   = ( windowSize / overlap ) / stretchFactor;
synthHop      = windowSize / overlap;
[ input, sr ] = audioread( [ filePath, fileName ] );
halfWinSize   = windowSize / 2;
inputSize     = length( input );
input         = [ zeros( windowSize, 1 ) ; input ; zeros( windowSize - mod(
inputSize, analysisHop ), 1 ) ] / max( abs( input ) );
output        = zeros(windowSize + ceil( length( input ) * stretchFactor ), 1
);
omega         = 2 * pi * analysisHop * [ 0 : halfWinSize ]' / windowSize;
phi0          = zeros( halfWinSize + 1, 1 );
psi           = zeros( halfWinSize + 1, 1 );
psi2          = psi;
numPrevPeaks  = 0;
analysisIndex = 0;
analysisMax   = length( input ) - windowSize;
synthIndex    = 0;

% create progress bar dialog box
bar = waitbar( 0, '0%', 'Name', sprintf( '%s: processing %s...', mfilename,
fileName ) );

% Processing Loop
%-------------------------------------------------------------------------
```

```
while analysisIndex < analysisMax

    % copy and window the input frame
    frame = input( analysisIndex + 1 : analysisIndex + windowSize ) .* window;

    % perform fft and save relevant half of spectrum
    spec = fft( fftshift( frame ) );
    halfSpec = spec( 1 : halfWinSize + 1 );

    % cartesian to polar conversion
    mag   = abs  ( halfSpec );
    phase = angle( halfSpec );

    % find spectral peaks ( local maxima )
    peakLoc  = zeros( halfWinSize + 1, 1 );
    numPeaks = 0;

    for b = 3 : halfWinSize - 1
        if( mag( b ) > mag( b - 1 ) && mag( b ) > mag( b - 2 ) && mag( b ) >
mag( b + 1 ) && mag( b ) > mag( b + 2 ) )
            numPeaks = numPeaks + 1;
            peakLoc( numPeaks ) = b;
            b = b + 3;
        end
    end

    % propagate peak phases and compute spectral bin phases
    if( analysisIndex == 0 )
        psi = phase;
    elseif( numPeaks > 0 && numPrevPeaks > 0 )
        prevPeak = 1;

        for p = 1 : numPeaks

            % connect current peak to the previous closest peak
            peak2 = peakLoc( p );
            while( prevPeak < numPrevPeaks && abs( peak2 - prevPeakLoc(
prevPeak + 1 ) ) < abs( peak2 - prevPeakLoc( prevPeak ) ) )
                prevPeak = prevPeak + 1;
            end
            peak1 = prevPeakLoc( prevPeak );

            % propagate peak's phase assuming linear frequency
            % variation between connected peak1 and peak2
            % ( avgPeak is a 1-based indexing spectral bin )
            avgPeak           = ( peak1 + peak2 ) * 0.5;
            peakOmega         = 2 * pi * analysisHop * ( avgPeak - 1.0 ) /
windowSize;
            peakDeltaPhase    = peakOmega + mod( ( phase( peak2 ) - phi0( peak1
) - peakOmega ) + pi, -2 * pi ) + pi;
            peakTargetPhase   = mod( ( psi( peak1 ) + peakDeltaPhase *
stretchFactor ) + pi, -2 * pi ) + pi;
```

```
            peakPhaseRotation = mod( ( peakTargetPhase - phase( peak2 ) ) + pi,
-2 * pi ) + pi;

            % rotate phases of all bins around the current peak
            if( numPeaks == 1 )
                bin1 = 1;
                bin2 = halfWinSize + 1;
            elseif( p == 1 )
                bin1 = 1;
                bin2 = halfWinSize + 1;
            elseif( p == numPeaks )
                bin1 = round( ( peakLoc( p - 1 ) + peak2 ) * 0.5 );
                bin2 = halfWinSize + 1;
            else
                bin1 = round( ( peakLoc( p - 1 ) + peak2 ) * 0.5 ) + 1;
                bin2 = round( ( peakLoc( p + 1 ) + peak2 ) * 0.5 );
            end
                psi2( bin1 : bin2 ) = mod( ( phase( bin1 : bin2 ) +
peakPhaseRotation ) + pi, -2 * pi ) + pi;
        end
            psi = psi2;
    else
        deltaPhase = omega + mod( ( phase - phi0 - omega ) + pi, -2 * pi ) +
pi;
        psi = mod( ( psi + deltaPhase * stretchFactor ) + pi, -2 * pi ) + pi;
    end

    % polar to cartesian conversion
    spec = ( mag .* exp( 1i * psi ) );

    % reconstruct entire spectrum
    spec = [ spec( 1 : halfWinSize + 1 ) ; conj( spec( halfWinSize : -1 : 2 ) )
];

    % perform an ifft on the spectrum
    frame = ifft( spec );

    % discard imaginary data
    frame = real( frame );

    % shift zero frequency component to center of spectrum
    frame = fftshift( frame );

    % apply the window
    frame = frame .* window;

    % overlap-add the synthesized frame into the output buffer
    output( synthIndex + 1 : synthIndex + windowSize ) = output( synthIndex + 1
: synthIndex + windowSize ) + frame;

    % store values for next frame
    phi0          = phase;
```

```
    prevPeakLoc   = peakLoc;
    numPrevPeaks  = numPeaks;

    % increment analysis and synthesis indices
    analysisIndex = analysisIndex + analysisHop;
    synthIndex    = synthIndex + synthHop;

    % update the progress bar
    progress = ( analysisIndex / analysisMax );
    waitbar( progress, bar, sprintf( '%2.3f%%', progress * 100 ) );
end

% close progress bar dialog box
close( bar );

% Output
%--------------------------------------------------------------------------

% crop output buffer
output = output( windowSize + 1 : length( output ) );

% normalize output buffer
normCoeff = 1 / max( abs( output ) );
output    = output * normCoeff;

% plot output
timevector = linspace( 0, length( output )/sr, length( output ) );
plot( timevector, output );

% create comment string
commentString = sprintf( 'WinSize: %.0f,\nOverlap: %.0f,\nStretch:
%.2f,\nWindow: %s', windowSize, overlap, stretchFactor, 'Hann' );

% write audio file to disk
[ a, fileName, b ] = fileparts( fileName );
outFile = sprintf( '%s.%s.wav', fileName, tag );
audiowrite( outFile, output, sr, 'BitsPerSample', 32, 'Artist', 'NormCoeff',
'Title', num2str( normCoeff ), 'Comment', commentString );

% EOF
```

### 8.2.4 – FFT Bank

```
function fft_bank( filePath, fileName )
%--------------------------------------------------------------------------
% FFT to Oscillator Bank Phase Vocoder
%
% adapted from:
% VX_tstretch_bank.m   [DAFXbook, 2nd ed., chapter 7]
%
% Cooper Baker - 2014
%--------------------------------------------------------------------------
```

```
close all;

% Settings
%-------------------------------------------------------------------------
windowSize    = 1024;
overlap       = 4;
stretchFactor = 2;
window        = hann( windowSize, 'periodic' );
tag           = 'bank';

% Initializations
%-------------------------------------------------------------------------
if any( exist( 'fileName' ) ~= 1 )
    [ fileName, filePath ] = uigetfile( '*.wav', 'Audio File' );
end

[ input, sr ] = audioread( [ filePath, fileName ] );
inputHopSize  = windowSize / overlap;
synthHopSize  = inputHopSize * stretchFactor;
input         = [ zeros( windowSize, 1) ; input ; zeros( windowSize - mod(
length( input ), windowSize ), 1 ) ];
output        = zeros( windowSize + length( input ) * stretchFactor, 1 );
halfWinSize   = windowSize / 2;
inputIndex    = 0;
inputMax      = length( input ) - windowSize;
outputIndex   = 0;
magOld        = zeros( halfWinSize, 1 );
phaseOld      = zeros( halfWinSize, 1 );
phaseOut      = zeros( halfWinSize, 1 );
synthHop      = zeros( synthHopSize, 1 );
omega         = 2 * pi * inputHopSize * [ 0 : halfWinSize - 1 ]' / windowSize;

% create progress bar dialog box
bar = waitbar( 0, '0%', 'Name', sprintf( '%s: processing %s...', mfilename,
fileName ) );

% Processing Loop
%-------------------------------------------------------------------------
while inputIndex < inputMax;

    %copy and window the input frame
    inputFrame = input( inputIndex + 1 : inputIndex + windowSize ) .* window;

    % perform an fft on the input frame
    fullSpect = fft( inputFrame );

    % copy relevant half of the spectrum
    spect = fullSpect( 1 : halfWinSize );

    % cartesian to polar conversion
    mag   = abs( spect );
    phase = angle( spect );
```

```
    % calculate phase delta
    phaseDelta = (phase - phaseOld - omega);
    phaseDelta = mod( phaseDelta + pi, -2 * pi ) + pi;
    phaseDelta = phaseDelta + omega;

    % calculate increments for oscillator bank interpolation
    magInc   = ( mag - magOld ) / synthHopSize;
    phaseInc = phaseDelta / inputHopSize;

    % sample by sample oscillator bank synthesis
    for synthIndex = 1 : synthHopSize;

        % linearly interpolate magnitude and phase
        magOld   = magOld   + magInc;
        phaseOut = phaseOut + phaseInc;

        % synthesize waveform
        synthHop( synthIndex ) = magOld' * cos( phaseOut );
    end

    % save polar values for next frame
    magOld   = mag;
    phaseOld = phase;
    phaseOut = mod( phaseOut + pi, -2 * pi ) + pi;

    % normalize the frame
    synthHop = synthHop / windowSize;

    % concatenate synthesized audio to output buffer
    output( outputIndex + 1 : outputIndex + synthHopSize ) = synthHop;

    % increment location values
    inputIndex  = inputIndex + inputHopSize;
    outputIndex = outputIndex + synthHopSize;

    % update the progress bar
    progress = ( inputIndex / inputMax );
    waitbar( progress, bar, sprintf( '%2.3f%%', progress * 100 ) )
end

% close progress bar dialog box
close( bar );

% Output
%-------------------------------------------------------------------------

% crop output buffer
output = output( windowSize + 1 : length( output ) );

% normalize output buffer
normCoeff = 1 / max( abs( output ) );
output    = output * normCoeff;
```

```
% plot output
timevector = linspace( 0, length( output ) / sr, length( output ) );
plot( timevector, output );

% make comment string
commentString = sprintf( 'WinSize: %.0f,\nOverlap: %.0f,\nStretch:
%.2f,\nWindow: %s', windowSize, overlap, stretchFactor, 'Hann' );

% write audio file to disk
[ a, fileName, b ] = fileparts( fileName );
outFile = sprintf( '%s.%s.wav', fileName, tag );
audiowrite( outFile, output, sr, 'BitsPerSample', 32, 'Artist', 'NormCoeff',
'Title', num2str( normCoeff ), 'Comment', commentString );

% EOF
```

### 8.3 – Granular Scripts

#### 8.3.1 – OLA

```
function ola( filePath, fileName )
%-------------------------------------------------------------------------
% OverLap Add
%
% Cooper Baker - 2014
%-------------------------------------------------------------------------

close all;

% Settings
%-------------------------------------------------------------------------
grainSize     = 1024;
overlap       = 4;
stretchFactor = 2;
window        = hann( grainSize, 'periodic' );
tag           = 'ola';

% Initializations
%-------------------------------------------------------------------------
if any( exist( 'fileName' ) ~= 1 )
    [ fileName, filePath ] = uigetfile( '*.wav', 'Audio File' );
end

grainHopSize   = ( grainSize / overlap ) / stretchFactor;
stretchHopSize = grainHopSize;
[ input, sr ]  = audioread( [ filePath, fileName ] );
input          = [ zeros( grainSize, 1 ) ; input ; zeros( grainSize - mod(
length( input ), grainHopSize ), 1 ) ];
output         = [ zeros( length( input ) * stretchFactor + ( grainSize / 2 ),
1 ) ];
grainIndex     = 0;
```

```
grainMax       = length( input ) - grainSize;
stretchIndex   = 0;

% create progress bar dialog box
bar = waitbar( 0, '0%', 'Name', sprintf( '%s: processing %s...', mfilename,
fileName ) );

% Processing Loop
%--------------------------------------------------------------------------
while grainIndex < grainMax

    % randomize input grain location within hop to mitigate robotization
    grainRand = grainIndex + randi( grainHopSize );
    grain = input( grainRand + 1 : grainRand + grainSize ) .* window;

    % write grain to output buffer
    output( stretchIndex + 1 : stretchIndex + grainSize ) = output(
stretchIndex + 1 : stretchIndex + grainSize ) + grain;

    % increment grain indices
    grainIndex = grainIndex + grainHopSize;
    stretchIndex = stretchIndex + stretchHopSize;

    % update the progress bar
    progress = ( grainIndex / grainMax );
    waitbar( progress, bar, sprintf( '%2.3f%%', progress * 100 ) )
end

% close progress bar dialog box
close( bar );

% Output
%--------------------------------------------------------------------------
% crop output buffer
output = output( grainSize + 1 : length( output ) );

% normalize output
normCoeff = 1 / max( abs( output ) );
output    = output * normCoeff;

% make comments for file
commentString = sprintf( 'WinSize: %.0f,\nOverlap: %.0f,\nStretch:
%.2f,\nWindow: %s', grainSize, overlap, stretchFactor, 'Hann' );

% save output to file
% write audio file to disk
[ a, fileName, b ] = fileparts( fileName );
outFile = sprintf( '%s.%s.wav', fileName, tag );
audiowrite( outFile, output, sr, 'BitsPerSample', 32, 'Artist', 'NormCoeff',
'Title', num2str( normCoeff ), 'Comment', commentString );
```

```
% plot output
timevector = linspace( 0, length( output ) / sr, length( output ) );
plot( timevector, output );

% EOF
```

### 8.3.2 – SOLA

```
function sola( filePath, fileName )
%-------------------------------------------------------------------------
% Synchronous OverLap Add
%
% adapted from:
% TimeScaleSOLA.m   [DAFXbook, 2nd ed., chapter 6]
%
% Cooper Baker - 2014
%-------------------------------------------------------------------------

close all;

% Settings
%-------------------------------------------------------------------------
grainSize      = 1024;                  % 2048 - default
hopSize        = 256;                   % 256  - default
stretchFactor  = 2;                     % >= 0.25 and <= 2
tag            = 'sola';

% Initializations
%-------------------------------------------------------------------------
if any( exist( 'fileName' ) ~= 1 )
    [ fileName, filePath ] = uigetfile( '*.wav', 'Audio File' );
end

overlap        = ( hopSize * stretchFactor ) / 2;
[ input, sr ]  = audioread( [ filePath, fileName ] );
inputTranspose = input';
grainIndex     = 1;
grainMax       = ceil( length( inputTranspose ) / hopSize );
hopSizeStretch = round(hopSize*stretchFactor);
inputTranspose( grainMax * hopSize + grainSize ) = 0;
output         =  inputTranspose( 1 : grainSize );

% create progress bar dialog box
bar = waitbar( 0, '0%', 'Name', sprintf( '%s: processing %s...', mfilename,
fileName ) );

% Processing Loop
%-------------------------------------------------------------------------
while grainIndex < grainMax
```

```
    % copy the input grain
    grain = inputTranspose( grainIndex * hopSize + 1 : grainSize + grainIndex *
hopSize );

    % perform cross correlation
    crossCorrelation = xcorr( grain ( 1 : overlap ), output( 1, grainIndex *
hopSizeStretch : grainIndex * hopSizeStretch + ( overlap - 1 ) ) );

    % find maximum in cross correlate
    [ xmax( 1, grainIndex ), index( 1, grainIndex ) ] = max( crossCorrelation
);

    % generate fadeOut ramp
    fadeOut = 1 : ( -1 / ( length( output ) - ( grainIndex * hopSizeStretch - (
overlap - 1 ) + index( 1, grainIndex ) - 1 ) ) ) : 0;

    % generate fadeIn ramp
    fadeIn  = 0 : (  1 / ( length( output ) - ( grainIndex * hopSizeStretch - (
overlap - 1 ) + index( 1, grainIndex ) - 1 ) ) ) : 1;

    % fade out tail portion of previous grain
    tail = output( 1, ( grainIndex * hopSizeStretch - ( overlap - 1 ) ) +
index( 1, grainIndex ) - 1 : length( output) ) .* fadeOut;

    % fade in head portion of current grain
    head = grain( 1 : length( fadeIn ) ) .* fadeIn;

    % mix head and tail into crossfade
    crossfade = tail + head;

    % concatenate result to end of output buffer
    output = [ output( 1, 1 : grainIndex * hopSizeStretch - overlap + index( 1,
grainIndex ) - 1 ), crossfade, grain( length( fadeIn ) + 1 : grainSize ) ];

    % increment the grain index
    grainIndex = grainIndex + 1;

    % update the progress bar
    progress = ( grainIndex / grainMax );
    waitbar( progress, bar, sprintf( '%2.3f%%', progress * 100 ) )
end

% close progress bar dialog box
close( bar );

% Output
%--------------------------------------------------------------------------

% crop output buffer
output = output( 1 : length( output ) );
```

```
% normalize audio
normCoeff = 1 / max( abs( output ) );
output = output * normCoeff;

% make comments for file
commentString = sprintf( 'WinSize: %.0f,\nOverlap: %.0f,\nStretch:
%.2f,\nWindow: %s', grainSize, overlap, stretchFactor, 'Hann' );

% write audio file to disk
[ a, fileName, b ] = fileparts( fileName );
outFile = sprintf( '%s.%s.wav', fileName, tag );
audiowrite( outFile, output, sr, 'BitsPerSample', 32, 'Artist', 'NormCoeff',
'Title', num2str( normCoeff ), 'Comment', commentString );

% plot output
timevector = linspace( 0, length( output )/sr, length( output ) );
plot( timevector, output );

% EOF
```

## 8.4 – Analysis Scripts

### 8.4.1 – File Display

```
function file_disp( filePath, fileName )
%-------------------------------------------------------------------------
% Waveform and Spectrogram Displays
%
% Cooper Baker - 2014
%-------------------------------------------------------------------------

close all;

% Settings
%-------------------------------------------------------------------------
winSize = 512;
overlap = 4;
win     = chebwin( winSize, 125 );
name    = 'File Display';

% Initializations
%-------------------------------------------------------------------------
if any( exist( 'fileName' ) ~= 1 )
    [ fileName, filePath ] = uigetfile( '*.wav', 'Audio File' );
end

[ buf, sr ] = audioread( [ filePath, fileName ] );
nyqBin      = winSize / 2;
hopSize     = winSize / overlap;
samps       = length( buf );
fwdSamps    = winSize - hopSize;
```

```
seconds     = round( ( samps / sr ) * 100 ) / 100;
frames      = ( samps / winSize ) * overlap - ( overlap - 1 );
nyquist     = sr / 2;

% Analysis
%-------------------------------------------------------------------------

% spectrogram
rawGram = spectrogram( buf, win, fwdSamps, winSize, sr );

% intermediate calculations
magGram = abs( rawGram );
ampGram = ( magGram / winSize ) * overlap;

% decibel spectrogram
dbGram = 20 * log10( ampGram );

% Plots
%-------------------------------------------------------------------------
fontName   = 'Times New Roman';
fontSize   = 12;
xSpc       = 1 / 100;
ySpc       = 1 / 25;

fig = figure( 1 );
set( fig, 'Name', fileName );
set( fig, 'Position', [ 0 0 800 500 ] );
set( fig, 'defaultAxesFontName', fontName );
set( fig, 'defaultTextFontName', fontName );

% Samples
% ------------------------------------------------------------------------
set( subplot( 2, 1, 1 ), 'Position', [ 0.1, 0.5, 0.8, 0.37 ] );
plot( buf, 'LineWidth', 1, 'Color', 'Black' );
axis( [ 0 samps -1 1 ] );
set( gca, 'XTick',      [ 0 samps   ] );
set( gca, 'XTickLabel', [] );
set( gca, 'YTickLabel', [] );
ylabel( 'Amplitude' );

% Spectrogram
% ------------------------------------------------------------------------
set( subplot( 2, 1, 2 ), 'Position', [ 0.1, 0.1, 0.8, 0.37 ] );
imagesc( dbGram );
caxis( [ -120 0 ] );
axis xy;
axis( [ 0 frames 0 nyqBin ] );
set( gca, 'XTick',      [ 0 frames  ] );
set( gca, 'XTickLabel', [ 0 seconds ] );
set( gca, 'YTickLabel', [] );
xlabel( 'Seconds' );
ylabel( 'Frequency' );
```

```
% tighten up figure borders
tightfig();

xlabel( 'Seconds', 'Position', [ ( frames / 2 ) -( nyqBin * ySpc ) ] );

% Export Graphs
%-------------------------------------------------------------------------
[ a, fileName, b ] = fileparts( fileName );

hgexport( fig, [ fileName, '.eps' ] );

% EOF
```

### 8.4.2 – Average Spectrum of Sample

```
function avg_file( idlPath, idlFile, strPath, strFile )
%-------------------------------------------------------------------------
% Average Spectrum of Audio Files
%
% Cooper Baker - 2014
%-------------------------------------------------------------------------

close all;

% Settings
%-------------------------------------------------------------------------
winSize = 512;
overlap = 4;
win     = chebwin( winSize, 125 );
name    = 'Average Spectrum';

% Initializations
%-------------------------------------------------------------------------
if any( exist( 'idlFile' ) ~= 1 )
    [ idlFile, idlPath ] = uigetfile( '*.wav', 'Ideal Audio File' );
    [ strFile, strPath ] = uigetfile( '*.wav', 'Stretched Audio File' );
end

[ idlBuf, sr ] = audioread( [ idlPath, idlFile ] );
[ strBuf, sr ] = audioread( [ strPath, strFile ] );
halfWinSize    = winSize / 2;
idlSize        = length( idlBuf );
strSize        = length( strBuf );
idlMsec        = ( idlSize / sr ) * 1000;
gridMsec       = 500;
gridHops       = gridMsec / ( ( ( winSize / overlap ) / sr ) * 1000 );
hopSize        = winSize / overlap;
hopMsec        = ( hopSize / sr ) * 1000;
idlAvg         = zeros( halfWinSize, 1 );
strAvg         = zeros( halfWinSize, 1 );
idlIndex       = 1;
strIndex       = 1;
```

```
winSum         = sum( win );

% Normalization
%-------------------------------------------------------------------------

% get file metadata
idlInfo = audioinfo( [ idlPath, idlFile ] );
strInfo = audioinfo( [ strPath, strFile ] );

% get file normalization coefficient
idlFileNorm = str2num( idlInfo.Title );
strFileNorm = str2num( strInfo.Title );

% de-normalize normalized .wav file data
idlRaw = idlBuf / idlFileNorm;
strRaw = strBuf / strFileNorm;

% make hann windows
idlWin = hann( idlSize );
strWin = hann( strSize );

% window the files
idlNormWin = idlRaw .* idlWin;
strNormWin = strRaw .* strWin;

% calculate rms of windowed files
idlRms = rms( idlNormWin );
strRms = rms( strNormWin );

% calculate normalization scalar
strScale = 1 / ( strRms / idlRms );

% normalize stretch buffer to match ideal buffer
idlNorm = idlRaw;
strNorm = strRaw * strScale;

% Analysis Loops
%-------------------------------------------------------------------------

% ideal file
hop = 1;
hopMax = length( idlNorm ) - winSize;
while hop < hopMax

    % copy analysis frame from buffer
    idlFrame = idlNorm( hop : hop + winSize - 1 );

    % window the frame
    idlWin = idlFrame .* win;

    % perform fft
    idlFullSpec = fft( idlWin );
```

```
    % save relevant half of spectrum
    idlSpec = idlFullSpec( 1 : halfWinSize );

    % calculate magnitude spectrum
    idlMag = abs( idlSpec );

    % store all magnitude spectra into array
    idlArray( idlIndex, : ) = idlMag;

    % increment indices
    hop = hop + hopSize;
    idlIndex = idlIndex + 1;
end

% stretched file
hop = 1;
hopMax = length( strNorm ) - winSize;
while hop < hopMax

    % copy analysis frame from buffer
    strFrame = strNorm( hop : hop + winSize - 1 );

    % window the frame
    strWin = strFrame .* win;

    % perform fft
    strFullSpec = fft( strWin );

    % save relevant half of spectrum
    strSpec = strFullSpec( 1 : halfWinSize );

    % calculate magnitude spectrum
    strMag = abs( strSpec );

    % store all magnitude spectra into array
    strArray( strIndex, : ) = strMag;

    % increment indices
    hop = hop + hopSize;
    strIndex = strIndex + 1;
end

% calculate average spectra from all hops
idlMag = sum( idlArray ) / idlIndex;
strMag = sum( strArray ) / strIndex;

% calculate amplitude spectra
idlAmp = ( idlMag / winSum );
strAmp = ( strMag / winSum );

% calculate sones spectra
idlSones = idlAmp .^ 0.6;
```

```
strSones = strAmp .^ 0.6;

% calculate decibel spectra
idlDb  = 20 * log10( idlAmp );
strDb  = 20 * log10( strAmp );

% copy spectra to plot arrays
idlPlot = idlSones;
strPlot = strSones;

% calculate ideal plot offset
offset = 0;
offset = abs( min( [ min( idlPlot ) min( strPlot ) ] ) );

% offset plots
idlPlot = idlPlot + offset;
strPlot = strPlot + offset;

% calculate absolute difference between plots (error)
difPlot = abs( idlPlot - strPlot );

% calculate mean error value
difMean = sum( difPlot ) / ( halfWinSize );

% calculate max error value
difMax = max( difPlot );

% Plot
%--------------------------------------------------------------------------
fontName   = 'Times New Roman';
fontSize   = 12;
xSpc       = 0.103;
ySpc       = 0.64;

fig = figure( 1 );
set( fig, 'Name', name );
set( fig, 'Position', [ 0 0 800 275 ] );
set( fig, 'defaultAxesFontName', fontName );
set( fig, 'defaultTextFontName', fontName );

% spectra
plot( idlPlot, 'Color', [ 0.2 0.2 0.8 ], 'LineWidth', 6 );
hold on;
plot( strPlot, 'Color', [ 0.2 0.8 0.2 ], 'LineWidth', 2 );
hold on;
plot( difPlot, 'Color', [ 0.8 0.2 0.2 ], 'LineWidth', 3 );
hold on;

% grid and labels
grid on;
yMin = min( [ min( idlPlot ) min( strPlot ) min( difPlot ) ] );
yMax = max( [ max( idlPlot ) max( strPlot ) max( difPlot ) ] );
```

```
axis( [ 0 halfWinSize yMin yMax ]);
set( gca, 'XTick', [ 0 : ( halfWinSize / ( sr / 4000 ) ) : halfWinSize ] );
set( gca, 'XTickLabel', [ 0 : 2000 : ( sr / 2 ) ] );
xlabel( 'Frequency' );
ylabel( 'Sones' );
% title( name, 'FontSize', fontSize );

% legend
xPos = ( halfWinSize + halfWinSize * -xSpc );
yPos = yMax * ySpc;
leg = legend( 'Ideal', 'Stretch', 'Error' );
set( leg, 'Location', 'NorthEast' );
maxStr = sprintf( ' Max: %2.5f', difMax );
meanStr = sprintf( ' Avg: %2.5f', difMean );
legStr = sprintf( '  Sones Error\n%s\n%s', maxStr, meanStr );
text( xPos, yPos, legStr, 'BackgroundColor', 'w', 'EdgeColor', 'k' );

% tighen up figure borders
tightfig();

% Save Data to Files
%--------------------------------------------------------------------------

% get file name
[ a, fileName, b ] = fileparts( strFile );

% write plot to file
hgexport( fig, [ fileName, '.avg.eps' ] );

% write error data to files
csvFile = sprintf( '%s.avg.err.csv', fileName );
csvwrite( csvFile, difPlot );

% EOF
```

### 8.4.3 – Average Spectrum of Waveform

```
function avg_tone( idlPath, idlFile, strPath, strFile )
%--------------------------------------------------------------------------
% Average Spectrum of Tone Files
%
% Cooper Baker - 2014
%--------------------------------------------------------------------------

close all;

% Settings
%--------------------------------------------------------------------------
winSize = 512;
overlap = 4;
win     = chebwin( winSize, 125 );
name    = 'Average Spectrum';
```

```
% Initializations
%-------------------------------------------------------------------------
if any( exist( 'idlFile' ) ~= 1 )
    [ idlFile, idlPath ] = uigetfile( '*.wav', 'Ideal Audio File' );
    [ strFile, strPath ] = uigetfile( '*.wav', 'Stretched Audio File' );
end

[ idlBuf, sr ] = audioread( [ idlPath, idlFile ] );
[ strBuf, sr ] = audioread( [ strPath, strFile ] );
halfWinSize    = winSize / 2;
idlSize        = length( idlBuf );
strSize        = length( strBuf );
idlMsec        = ( idlSize / sr ) * 1000;
gridMsec       = 500;
gridHops       = gridMsec / ( ( ( winSize / overlap ) / sr ) * 1000 );
hopSize        = winSize / overlap;
hopMsec        = ( hopSize / sr ) * 1000;
idlAvg         = zeros( halfWinSize, 1 );
strAvg         = zeros( halfWinSize, 1 );
index          = 1;
winSum         = sum( win );

% trim or pad stretch buffer to match ideal buffer length
if( idlSize > strSize )
    strBuf = [ strBuf ; zeros( idlSize - strSize, 1 ) ];
elseif( strSize > idlSize )
    strBuf = strBuf( 1 : length( idlBuf ) );
end

% analyze only middle of buffers for 'tone' files padded with silence
hop    = round( idlSize / 3 );
hopMax = round( ( idlSize * 2 ) / 3 );

% Normalization
%-------------------------------------------------------------------------

% get file metadata
idlInfo = audioinfo( [ idlPath, idlFile ] );
strInfo = audioinfo( [ strPath, strFile ] );

% get file normalization coefficient
idlFileNorm = str2num( idlInfo.Title );
strFileNorm = str2num( strInfo.Title );

% de-normalize normalized .wav file data
idlRaw = idlBuf / idlFileNorm;
strRaw = strBuf / strFileNorm;

% make hann window
normWin = hann( round( idlSize / 2 ) );
```

```
% copy and window middle of files
idlNormWin = idlRaw( idlSize * 0.25 : idlSize * 0.75 - 1 ) .* normWin;
strNormWin = strRaw( idlSize * 0.25 : idlSize * 0.75 - 1 ) .* normWin;

% calculate rms of windowed file middles
idlRms = rms( idlNormWin );
strRms = rms( strNormWin );

% calculate normalization scalar
strScale = 1 / ( strRms / idlRms );

% scale stretch buffer to match ideal buffer
idlRaw = idlRaw;
strRaw = strRaw * strScale;

% Analysis Loop
%-------------------------------------------------------------------------
while hop < hopMax

    % copy analysis frame from buffer
    idlSamps = idlRaw( hop : hop + winSize - 1 );
    strSamps = strRaw( hop : hop + winSize - 1 );

    % window the frame
    idlWin = idlSamps .* win;
    strWin = strSamps .* win;

    % perform fft
    idlFullSpec = fft( idlWin );
    strFullSpec = fft( strWin );

    % save relevant halves of spectra
    idlSpec = idlFullSpec( 1 : halfWinSize );
    strSpec = strFullSpec( 1 : halfWinSize );

    % calculate magnitude spectra
    idlMag = abs( idlSpec );
    strMag = abs( strSpec );

    % store all magnitude spectra into array
    idlArray( index, : ) = idlMag;
    strArray( index, : ) = strMag;

    % increment indices
    hop = hop + hopSize;
    index = index + 1;
end

% calculate average spectra from all hops
idlMag = sum( idlArray ) / index;
strMag = sum( strArray ) / index;
```

```
% calculate amplitude spectra
idlAmp = ( idlMag / winSum );
strAmp = ( strMag / winSum );

% calculate sones spectra
idlSones = idlAmp .^ 0.6;
strSones = strAmp .^ 0.6;

% calculate decibel spectra
idlDb  = 20 * log10( idlAmp );
strDb  = 20 * log10( strAmp );

% copy spectra to plot arrays
idlPlot = idlSones;
strPlot = strSones;

% calculate ideal plot offset
offset = 0;
offset = abs( min( [ min( idlPlot ) min( strPlot ) ] ) );

% offset plots
idlPlot = idlPlot + offset;
strPlot = strPlot + offset;

% calculate absolute difference between plots (error)
difPlot = abs( idlPlot - strPlot );

% calculate mean error value
difMean = sum( difPlot ) / ( halfWinSize );

% calculate max error value
difMax = max( difPlot );

% Plot
%--------------------------------------------------------------------------
fontName   = 'Times New Roman';
fontSize   = 12;
xSpc       = 0.103;
ySpc       = 0.64;

fig = figure( 1 );
set( fig, 'Name', name );
set( fig, 'Position', [ 0 0 800 275 ] );
set( fig, 'defaultAxesFontName', fontName );
set( fig, 'defaultTextFontName', fontName );

% spectra
plot( idlPlot, 'Color', [ 0.2 0.2 0.8 ], 'LineWidth', 6 );
hold on;
plot( strPlot, 'Color', [ 0.2 0.8 0.2 ], 'LineWidth', 2 );
hold on;
plot( difPlot, 'Color', [ 0.8 0.2 0.2 ], 'LineWidth', 3 );
```

```
hold on;

% grid and labels
grid on
yMin = min( [ min( idlPlot ) min( strPlot ) min( difPlot ) ] );
yMax = max( [ max( idlPlot ) max( strPlot ) max( difPlot ) ] );
axis( [ 0 halfWinSize yMin yMax ]);
set( gca, 'XTick', [ 0 : ( halfWinSize / ( sr / 4000 ) ) : halfWinSize ] );
set( gca, 'XTickLabel', [ 0 : 2000 : ( sr / 2 ) ] );
xlabel( 'Frequency' );
ylabel( 'Sones' );

% legend
xPos = ( halfWinSize + halfWinSize * -xSpc );
yPos = yMax * ySpc;
leg = legend( 'Ideal', 'Stretch', 'Error' );
set( leg, 'Location', 'NorthEast' );
maxStr = sprintf( ' Max: %2.5f', difMax );
meanStr = sprintf( ' Avg: %2.5f', difMean );
legStr = sprintf( '  Sones Error\n%s\n%s', maxStr, meanStr );
text( xPos, yPos, legStr, 'BackgroundColor', 'w', 'EdgeColor', 'k' );

% tighten up figure borders
tightfig();

% Save Data to Files
%--------------------------------------------------------------------------

% get file name
[ a, fileName, b ] = fileparts( strFile );

% write plot to file
hgexport( fig, [ fileName, '.avg.eps' ] );

% write error data to files
csvFile = sprintf( '%s.avg.err.csv', fileName );
csvwrite( csvFile, difPlot );

% EOF
```

### 8.4.4 – Moving Spectral Average of Sample

```
function mov_file( idlPath, idlFile, strPath, strFile )
%--------------------------------------------------------------------------
% Moving Spectral Average of Audio File
%
% Cooper Baker - 2014
%--------------------------------------------------------------------------

close all;
```

```
% Settings
%-------------------------------------------------------------------------
winSize = 512;
overlap = 4;
win     = chebwin( winSize, 125 );
name    = 'Moving Spectral Average';

% Initializations
%-------------------------------------------------------------------------
if any( exist( 'idlFile' ) ~= 1 )
    [ idlFile, idlPath ] = uigetfile( '*.wav', 'Ideal Audio File' );
    [ strFile, strPath ] = uigetfile( '*.wav', 'Stretched Audio File' );
end

[ idlBuf, sr ] = audioread( [ idlPath, idlFile ] );
[ strBuf, sr ] = audioread( [ strPath, strFile ] );
halfWinSize    = winSize / 2;
idlSize        = length( idlBuf );
strSize        = length( strBuf );
idlMsec        = ( idlSize / sr ) * 1000;
gridMsec       = 500;
gridHops       = gridMsec / ( ( ( winSize / overlap ) / sr ) * 1000 );
hopSize        = winSize / overlap;
hopMsec        = ( hopSize / sr ) * 1000;
index          = 1;
idlIndex       = 1;
strIndex       = 1;
winSum         = sum( win );

% crop or pad stretch buffer to match ideal buffer times two
if ( idlSize * 2 ) > strSize
    strBuf = [ strBuf ; zeros( ( idlSize * 2 ) - strSize, 1 ) ];
elseif strSize > ( idlSize * 2 )
    strBuf = strBuf( 1 : idlSize * 2 );
end

strSize = length( strBuf );

% Normalization
%-------------------------------------------------------------------------

% get file metadata
idlInfo = audioinfo( [ idlPath, idlFile ] );
strInfo = audioinfo( [ strPath, strFile ] );

% get file normalization coefficient
idlFileNorm = str2num( idlInfo.Title );
strFileNorm = str2num( strInfo.Title );

% de-normalize normalized .wav file data
idlRaw = idlBuf / idlFileNorm;
strRaw = strBuf / strFileNorm;
```

```
% make hann windows
idlWin = hann( idlSize );
strWin = hann( strSize );

% window the files
idlNormWin = idlRaw .* idlWin;
strNormWin = strRaw .* strWin;

% calculate rms of windowed files
idlRms = rms( idlNormWin );
strRms = rms( strNormWin );

% calculate normalization scalar
strScale = 1 / ( strRms / idlRms );

% scale stretch buffer to match ideal buffer
idlNorm = idlRaw;
strNorm = strRaw * strScale;

% Analysis Loops
%-------------------------------------------------------------------------
% ideal file
hop    = 1;
hopMax = length( idlBuf ) - winSize;

while hop < hopMax

    % copy analysis frame from buffer
    idlSamps = idlNorm( hop : hop + winSize - 1 );

    % window the frame
    idlWin = idlSamps .* win;

    % perform fft
    idlFullSpec = fft( idlWin );

    % save relevant halves of spectra
    idlSpec = idlFullSpec( 1 : halfWinSize );

    % calculate magnitude spectra
    idlMagSpec = abs( idlSpec );

    % calculate spectral magnitude averages
    idlAvg = sum( idlMagSpec ) / halfWinSize;

    % store average values into arrays
    idlMag( idlIndex ) = idlAvg;

     % increment indices
    hop = hop + hopSize;
    idlIndex = idlIndex + 1;
end
```

```
% stretched file
hop    = 1;
hopMax = length( strBuf ) - winSize;

while hop < hopMax

    % copy analysis frame from buffer
    strSamps = strNorm( hop : hop + winSize - 1 );

    % window the frame
    strWin = strSamps .* win;

    % perform fft
    strFullSpec = fft( strWin );

    % save relevant halves of spectra
    strSpec = strFullSpec( 1 : halfWinSize );

    % calculate magnitude spectra
    strMagSpec = abs( strSpec );

    % calculate spectral magnitude averages
    strAvg = sum( strMagSpec ) / halfWinSize;

    % store average values into arrays
    strMagLong( strIndex ) = strAvg;

     % increment indices
    hop = hop + hopSize;
    strIndex = strIndex + 1;
end

% compress stretched array to match ideal array
while index < idlIndex;

    % calculate stretched array index
    strIndex = ( index * 2 - 1 );

    % average adjacent pairs of values
    strMag( index ) = ( strMagLong( strIndex ) + strMagLong( strIndex + 1 ) ) /
2;

    %increment index
    index = index + 1;
end

% calculate amplitude spectra
idlAmp = ( idlMag / winSum );
strAmp = ( strMag / winSum );

% calculate sones spectra
idlSones = idlAmp .^ 0.6;
```

```
strSones = strAmp .^ 0.6;

% calculate decibel spectra
idlDb = 20 * log10( idlAmp );
strDb = 20 * log10( strAmp );

% clip -infinity to -120 decibels
idlDb( idlDb < -120 ) = -120;
strDb( strDb < -120 ) = -120;

% copy spectra to plot arrays
idlPlot = idlSones;
strPlot = strSones;

% calculate ideal plot offset
offset = 0;
offset = abs( min( [ min( idlPlot ) min( strPlot ) ] ) );

% offset plots
idlPlot = idlPlot + offset;
strPlot = strPlot + offset;

% calculate difference between plots
difPlot = abs( idlPlot - strPlot );

% calculate mean error value
difMean = sum( difPlot ) / index;

% calculate max error value
difMax = max( difPlot );

% Plot
%------------------------------------------------------------------------
fontName   = 'Times New Roman';
fontSize   = 12;
xSpc       = 0.103;
ySpc       = 0.64;

fig = figure( 1 );
set( fig, 'Name', name );
set( fig, 'Position', [ 0 0 800 275 ] );
set( fig, 'defaultAxesFontName', fontName );
set( fig, 'defaultTextFontName', fontName );

% curves
plot( idlPlot, 'Color', [ 0.2 0.2 0.8 ], 'LineWidth', 6 );
hold on;
plot( strPlot, 'Color', [ 0.2 0.8 0.2 ], 'LineWidth', 2 );
hold on;
plot( difPlot, 'Color', [ 0.8 0.2 0.2 ], 'LineWidth', 3 );
hold on;
```

```
% grid and labels
grid on
yMin = min( [ min( idlPlot ) min( strPlot ) min( difPlot ) ] );
yMax = max( [ max( idlPlot ) max( strPlot ) max( difPlot ) ] );
axis( [ 0 idlIndex yMin yMax ]);
set( gca, 'XTick', [ 0 : gridHops : idlIndex ] );
set( gca, 'XTickLabel', [ 0 : gridMsec : idlMsec ] );
xlabel( 'Milliseconds' );
ylabel( 'Sones' );

% legend
xPos = ( idlIndex + idlIndex * -xSpc );
yPos = yMax * ySpc;
leg = legend( 'Ideal', 'Stretch', 'Error' );
set( leg, 'Location', 'NorthEast' );
maxStr = sprintf( ' Max: %2.5f', difMax );
meanStr = sprintf( ' Avg: %2.5f', difMean );
legStr = sprintf( '  Sones Error\n%s\n%s', maxStr, meanStr );
text( xPos, yPos, legStr, 'BackgroundColor', 'w', 'EdgeColor', 'k' );

% tighten up figure borders
tightfig();

% Save Data to Files
%--------------------------------------------------------------------------

% get file name
[ a, fileName, b ] = fileparts( strFile );

% export graph
hgexport( fig, [ fileName, '.mov.eps' ] );

% write error data to files
csvFile = sprintf( '%s.mov.err.csv', fileName );
csvwrite( csvFile, difPlot );

% EOF
```

### 8.4.5 – Moving Spectral Average of Waveform

```
function mov_tone( idlPath, idlFile, strPath, strFile )
%--------------------------------------------------------------------------
% Moving Spectral Average of Tone File
%
% Cooper Baker - 2014
%--------------------------------------------------------------------------

close all;

% Settings
%--------------------------------------------------------------------------
winSize = 512;
```

```
overlap = 4;
win     = chebwin( winSize, 125 );
name    = 'Moving Spectral Average';
headLoc = 0.25;
midLoc  = 0.5;
tailLoc = 0.75;
detail  = 0.02;

% Initializations
%--------------------------------------------------------------------------
if any( exist( 'idlFile' ) ~= 1 )
    [ idlFile, idlPath ] = uigetfile( '*.wav', 'Ideal Audio File' );
    [ strFile, strPath ] = uigetfile( '*.wav', 'Stretched Audio File' );
end

[ idlBuf, sr ] = audioread( [ idlPath, idlFile ] );
[ strBuf, sr ] = audioread( [ strPath, strFile ] );
idlSize        = length( idlBuf );
strSize        = length( strBuf );
idlMsec        = ( idlSize / sr ) * 1000;
rangeMsec      = detail * idlMsec;
headMsec       = ( headLoc - ( detail / 2 ) ) * idlMsec;
midMsec        = ( midLoc  - ( detail / 2 ) ) * idlMsec;
tailMsec       = ( tailLoc - ( detail / 2 ) ) * idlMsec;
headMsec       = headMsec + rangeMsec * 0.25;
midMsec        = midMsec  + rangeMsec * 0.25;
tailMsec       = tailMsec + rangeMsec * 0.25;
hopSize        = winSize / overlap;
hopMsec        = ( hopSize / sr ) * 1000;
gridMsec       = 500;
gridHops       = gridMsec / ( ( ( winSize / overlap ) / sr ) * 1000 );
halfWinSize    = winSize / 2;
hopMax         = length( idlBuf ) - winSize;
hop            = 1;
index          = 1;
winSum         = sum( win );

% crop or pad stretch buffer to match ideal buffer
if idlSize > strSize
    strBuf = [ strBuf ; zeros( idlSize - strSize, 1 ) ];
elseif strSize > idlSize
    strBuf = strBuf( 1 : length( idlBuf ) );
end

% Normalization
%--------------------------------------------------------------------------

% get file metadata
idlInfo = audioinfo( [ idlPath, idlFile ] );
strInfo = audioinfo( [ strPath, strFile ] );
```

```
% get file normalization coefficient
idlFileNorm = str2num( idlInfo.Title );
strFileNorm = str2num( strInfo.Title );

% de-normalize normalized .wav file data
idlRaw = idlBuf / idlFileNorm;
strRaw = strBuf / strFileNorm;

% make hann window
normWin = hann( round( idlSize / 2 ) );

% copy and window middle of files
idlNormWin = idlRaw( idlSize * 0.25 : idlSize * 0.75 - 1 ) .* normWin;
strNormWin = strRaw( idlSize * 0.25 : idlSize * 0.75 - 1 ) .* normWin;

% calculate rms of windowed file middles
idlRms = rms( idlNormWin );
strRms = rms( strNormWin );

% calculate normalization scalar
strScale = 1 / ( strRms / idlRms );

% scale stretch buffer to match ideal buffer
idlRaw = idlRaw;
strRaw = strRaw * strScale;

% Analysis Loop
%--------------------------------------------------------------------------
while hop < hopMax

    % copy analysis frame from buffer
    idlSamps = idlRaw( hop : hop + winSize - 1 );
    strSamps = strRaw( hop : hop + winSize - 1 );

    % window the frame
    idlWin = idlSamps .* win;
    strWin = strSamps .* win;

    % perform fft
    idlFullSpec = fft( idlWin );
    strFullSpec = fft( strWin );

    % save relevant halves of spectra
    idlSpec = idlFullSpec( 1 : halfWinSize );
    strSpec = strFullSpec( 1 : halfWinSize );

    % calculate magnitude spectra
    idlMagSpec = abs( idlSpec );
    strMagSpec = abs( strSpec );

    % calculate spectral magnitude averages
    idlAvg = sum( idlMagSpec ) / halfWinSize;
```

```
    strAvg = sum( strMagSpec ) / halfWinSize;

    % store average values into arrays
    idlMag( index ) = idlAvg;
    strMag( index ) = strAvg;

    % increment indices
    hop = hop + hopSize;
    index = index + 1;
end

% calculate amplitude spectra
idlAmp = ( idlMag / winSum );
strAmp = ( strMag / winSum );

% calculate sones spectra
idlSones = idlAmp .^ 0.6;
strSones = strAmp .^ 0.6;

% calculate decibel spectra
idlDb  = 20 * log10( idlAmp );
strDb  = 20 * log10( strAmp );

% clip -infinity to -120 decibels
idlDb( idlDb < -120 ) = -120;
strDb( strDb < -120 ) = -120;

% copy spectra to plot arrays
idlPlot = idlSones;
strPlot = strSones;

% calculate ideal plot offset
offset = 0;
offset = abs( min( [ min( idlPlot ) min( strPlot ) ] ) );

% offset plots
idlPlot = idlPlot + offset;
strPlot = strPlot + offset;

% calculate absolute difference between plots (error)
difPlot = abs( idlPlot - strPlot );

% calculate cropping values
side = round( index * detail );
head = round( index * headLoc );
mid  = round( index * midLoc );
tail = round( index * tailLoc );
range = side * 2;

% copy portions of spectra averges
idlHead = idlPlot( head - side : head + side );
idlMid  = idlPlot( mid  - side : mid  + side );
```

```
idlTail = idlPlot( tail - side : tail + side );
strHead = strPlot( head - side : head + side );
strMid  = strPlot( mid  - side : mid  + side );
strTail = strPlot( tail - side : tail + side );
difHead = difPlot( head - side : head + side );
difMid  = difPlot( mid  - side : mid  + side );
difTail = difPlot( tail - side : tail + side );

% calculate mean error value
difSig = difPlot( index * 0.25 : index * 0.75 );
difMean = sum( difSig ) / ( index * 0.5 );

% calculate max error value
difMax = max( difSig );

% Plots
%--------------------------------------------------------------------------
fontName   = 'Times New Roman';
fontSize   = 12;
xSpc       = 0.099;
ySpc       = 0.57;

fig = figure( 1 );
set( fig, 'Name', name );
set( fig, 'Position', [ 0 0 800 500 ] );
set( fig, 'defaultAxesFontName', fontName );
set( fig, 'defaultTextFontName', fontName );

% Main Plot
%--------------------------------------------------------------------------
set( subplot( 2, 1, 1 ), 'Position', [ 0.1, 0.5, 0.8, 0.3666 ] );
plot( idlPlot, 'Color', [ 0.2 0.2 0.8 ], 'LineWidth', 6 );
hold on;
plot( strPlot, 'Color', [ 0.2 0.8 0.2 ], 'LineWidth', 2 );
hold on;
plot( difPlot, 'Color', [ 0.8 0.2 0.2 ], 'LineWidth', 3 );
hold on;

% grid and labels
grid on
yMin = min( [ min( idlPlot ) min( strPlot ) min( difPlot ) ] );
yMax = max( [ max( idlPlot ) max( strPlot ) max( difPlot ) ] );
axis( [ 0 index yMin yMax ]);
set( gca, 'XTick', [ 0 : gridHops : index ] );
set( gca, 'XTickLabel', [ 0 : gridMsec : idlMsec ] );

% legend
xPos = ( index + index * -xSpc );
yPos = yMax * ySpc;
leg = legend( 'Ideal', 'Stretch', 'Error' );
set( leg, 'Location', 'NorthEast' );
maxStr = sprintf( ' Max: %2.5f', difMax );
```

```
meanStr = sprintf( ' Avg: %2.5f', difMean );
legStr = sprintf( '  Sones Error\n%s\n%s', maxStr, meanStr );
text( xPos, yPos, legStr, 'BackgroundColor', 'w', 'EdgeColor', 'k' );

% Head Plot
%-------------------------------------------------------------------------
set( subplot( 2, 3, 4 ), 'Position', [ 0.1, 0.1, 0.26, 0.3666 ] );

plot( idlHead, 'Color', [ 0.2 0.2 0.8 ], 'LineWidth', 6 );
hold on;
plot( strHead, 'Color', [ 0.2 0.8 0.2 ], 'LineWidth', 2 );
hold on;
plot( difHead, 'Color', [ 0.8 0.2 0.2 ], 'LineWidth', 3 );
hold on;

% grid and labels
grid on
yMin = min( [ min( idlPlot ) min( strPlot ) min( difPlot ) ] );
yMax = max( [ max( idlPlot ) max( strPlot ) max( difPlot ) ] );
axis( [ 0 range yMin yMax ]);
set( gca, 'XTick', [ range * 0.25 : ( range / 4 ) : range * 0.75 ] );
set( gca, 'XTickLabel', [ headMsec : rangeMsec / 4 : headMsec + rangeMsec ] );
yLab = ylabel( 'Sones' );
text( range * 0.03, yMax * 0.93, ' Head ', 'BackgroundColor', 'w', 'EdgeColor',
'k' );

% Mid Plot
%-------------------------------------------------------------------------
set( subplot( 2, 3, 5 ), 'Position', [ 0.3666, 0.1, 0.2666, 0.3666 ] );

plot( idlMid, 'Color', [ 0.2 0.2 0.8 ], 'LineWidth', 6 );
hold on;
plot( strMid, 'Color', [ 0.2 0.8 0.2 ], 'LineWidth', 2 );
hold on;
plot( difMid, 'Color', [ 0.8 0.2 0.2 ], 'LineWidth', 3 );
hold on;

% grid and labels
grid on
yMin = min( [ min( idlPlot ) min( strPlot ) min( difPlot ) ] );
yMax = max( [ max( idlPlot ) max( strPlot ) max( difPlot ) ] );
axis( [ 0 range yMin yMax ]);
set( gca, 'XTick', [ range * 0.25 : ( range / 4 ) : range * 0.75 ] );
set( gca, 'XTickLabel', [ midMsec : rangeMsec / 4 : midMsec + rangeMsec ] );
set( gca, 'YTickLabel', [] );
xlabel( 'Milliseconds' );
text( range * 0.5, yMax * 0.93, ' Middle ', 'BackgroundColor', 'w',
'EdgeColor', 'k', 'HorizontalAlignment', 'center' );

% Tail Plot
%-------------------------------------------------------------------------
set( subplot( 2, 3, 6 ), 'Position', [ 0.6399, 0.1, 0.26, 0.3666 ] );
```

```
plot( idlTail, 'Color', [ 0.2 0.2 0.8 ], 'LineWidth', 6 );
hold on;
plot( strTail, 'Color', [ 0.2 0.8 0.2 ], 'LineWidth', 2 );
hold on;
plot( difTail, 'Color', [ 0.8 0.2 0.2 ], 'LineWidth', 3 );
hold on;

% grid and labels
grid on
yMin = min( [ min( idlPlot ) min( strPlot ) min( difPlot ) ] );
yMax = max( [ max( idlPlot ) max( strPlot ) max( difPlot ) ] );
axis( [ 0 range yMin yMax ]);
set( gca, 'XTick', [ range * 0.25 : ( range / 4 ) : range * 0.75 ] );
set( gca, 'XTickLabel', [ tailMsec : rangeMsec / 4 : tailMsec + rangeMsec ] );
set( gca, 'YTickLabel', [] );
text( range * 0.97, yMax * 0.93, ' Tail ', 'BackgroundColor', 'w', 'EdgeColor',
'k', 'HorizontalAlignment', 'right' );

% tighten up figure borders
tightfig();

set( yLab, 'Position', [ range * -0.13, ( yMax * 1.05 ) ] );

% Save Data to Files
%--------------------------------------------------------------------------

% get file name
[ a, fileName, b ] = fileparts( strFile );

% export graph
hgexport( fig, [ fileName, '.mov.eps' ] );

% write error data to files
csvFile = sprintf( '%s.mov.err.csv', fileName );
csvwrite( csvFile, difPlot );

% EOF
```

### 8.4.6 – Error Spectrogram of Sample

```
function dif_file( idlPath, idlFile, strPath, strFile )
%--------------------------------------------------------------------------
% Error Spectrogram of Audio File
%
% Cooper Baker - 2014
%--------------------------------------------------------------------------

close all;

% Settings
%--------------------------------------------------------------------------
winSize = 512;
```

```
overlap = 4;
window  = chebwin( winSize, 125 );
name    = 'Error Spectrogram';

% Initializations
%-------------------------------------------------------------------------
if any( exist( 'idlFile' ) ~= 1 )
    [ idlFile, idlPath ] = uigetfile( '*.wav', 'Ideal Audio File' );
    [ strFile, strPath ] = uigetfile( '*.wav', 'Stretched Audio File' );
end

[ idlBuf, sr ] = audioread( [ idlPath, idlFile ] );
[ strBuf, sr ] = audioread( [ strPath, strFile ] );
idlSize        = length( idlBuf );
strSize        = length( strBuf );
halfWinSize    = winSize / 2;
overlapSamps   = winSize - ( winSize / overlap );
hopSize        = winSize / overlap;
hopMax         = length( idlBuf ) - winSize;
idlMsec        = ( idlSize / sr ) * 1000;
gridMsec       = 500;
gridHops       = gridMsec / ( ( ( winSize / overlap ) / sr ) * 1000 );
hopMsec        = ( hopSize / sr ) * 1000;
index          = 1;
winSum         = sum( window );

% crop or pad stretch buffer to match ideal buffer times two
if ( idlSize * 2 ) > strSize
    strBuf = [ strBuf ; zeros( ( idlSize * 2 ) - strSize, 1 ) ];
elseif strSize > ( idlSize * 2 )
    strBuf = strBuf( 1 : idlSize * 2 );
end

strSize = length( strBuf );

% Normalization
%-------------------------------------------------------------------------

% get file metadata
idlInfo = audioinfo( [ idlPath, idlFile ] );
strInfo = audioinfo( [ strPath, strFile ] );

% get file normalization coefficient
idlFileNorm = str2num( idlInfo.Title );
strFileNorm = str2num( strInfo.Title );

% de-normalize normalized .wav file data
idlRaw = idlBuf / idlFileNorm;
strRaw = strBuf / strFileNorm;

% make hann windows
idlWin = hann( idlSize );
```

```
strWin = hann( strSize );

% copy and window files
idlNormWin = idlRaw .* idlWin;
strNormWin = strRaw .* strWin;

% calculate rms of windowed file middles
idlRms = rms( idlNormWin );
strRms = rms( strNormWin );

% calculate normalization scalar
strScale = 1 / ( strRms / idlRms );

% scale stretch buffer to match ideal buffer
idlNorm = idlRaw;
strNorm = strRaw * strScale;

% Analysis
%-------------------------------------------------------------------------
idlGram     = spectrogram( idlNorm, window, overlapSamps, winSize, sr );
strGramLong = spectrogram( strNorm, window, overlapSamps, winSize, sr );

idlSize = length( idlGram ) + 1;

% compress stretched array to match ideal array
while index < idlSize;

    % calculate stretched array index
    strIndex = ( index * 2 - 1 );

    % average adjacent pairs of spectra
    strGram( :, index ) = ( strGramLong( :, strIndex ) + strGramLong( :,
strIndex + 1 ) ) ./ 2;

    %increment index
    index = index + 1;
end

% calculate magnitude spectra
idlMag = abs( idlGram );
strMag = abs( strGram );

% calculate amplitude spectra
idlAmp = ( idlMag / winSum );
strAmp = ( strMag / winSum );

% calculate sones spectra
idlSones = idlAmp .^ 0.6;
strSones = strAmp .^ 0.6;

% copy spectra to plot arrays
idlPlot = idlSones;
```

```
strPlot = strSones;

% calculate ideal plot offset
offset = 0;
offset = abs( min( [ min( idlPlot ) min( strPlot ) ] ) );

% offset plots
idlPlot = idlPlot + offset;
strPlot = strPlot + offset;

% calculate absolute difference between plots (error)
diffPlot = abs( idlPlot - strPlot );

% find maximum value of difference plot
diffMax = max( diffPlot(:) );

% Plot
%--------------------------------------------------------------------------
fontName   = 'Times New Roman';
fontSize   = 12;

fig = figure( 1 );
set( fig, 'Name', name );
set( fig, 'Position', [ 0 0 800 250 ] );
set( fig, 'defaultAxesFontName', fontName );
set( fig, 'defaultTextFontName', fontName );

colormap( [ [ 1 : -0.2/64 : 0.8 ]; ( [ 1 : -0.5/64 : 0.5 ] .^ 10 ) ; [ 1 : -
0.5/64 : 0.5 ] .^ 20 ]' );

% spectrogram
imagesc( diffPlot );
caxis( 'auto' );
axis xy;

% grid and labels
axis( [ 0 length( diffPlot ) 1 halfWinSize ] );
set( gca, 'XTick', [ 0 : gridHops : length( diffPlot ) ] );
set( gca, 'XTickLabel', [ 0 : gridMsec : idlMsec ] );
set( gca, 'YTick', [ 1 : ( halfWinSize / ( sr / 4000 ) ) : halfWinSize ] );
set( gca, 'YTickLabel', [ 0 : 2000 : ( sr / 2 ) ] );
xlabel( 'Milliseconds' );
ylabel( 'Frequency' );

% legend
bar = colorbar( 'location', 'East' );
set( get( bar, 'YLabel'), 'String', 'Sones' );
set( get( bar, 'YLabel'), 'Position', [ 0.75 ( diffMax * 0.1 ) ] );

% tighten up figure borders
tightfig();
```

```
% Save Data to Files
%-------------------------------------------------------------------------

% get file name
[ a, fileName, b ] = fileparts( strFile );

% write plot to file
hgexport( fig, [ fileName, '.spect.eps' ] );

% write error data to files
csvFile = sprintf( '%s.spect.csv', fileName );
csvwrite( csvFile, diffPlot );

% EOF
```

### 8.4.7 – Error Spectrogram of Waveform

```
function dif_tone( idlPath, idlFile, strPath, strFile )
%-------------------------------------------------------------------------
% Error Spectrogram of Tone File
%
% Cooper Baker - 2014
%-------------------------------------------------------------------------

close all;

% Settings
%-------------------------------------------------------------------------
winSize = 512;
overlap = 4;
window  = chebwin( winSize, 125 );
name    = 'Error Spectrogram';
headLoc = 0.25;
midLoc  = 0.5;
tailLoc = 0.75;
detail  = 0.01;

% Initializations
%-------------------------------------------------------------------------
if any( exist( 'idlFile' ) ~= 1 )
    [ idlFile, idlPath ] = uigetfile( '*.wav', 'Ideal Audio File' );
    [ strFile, strPath ] = uigetfile( '*.wav', 'Stretched Audio File' );
end

[ idlBuf, sr ] = audioread( [ idlPath, idlFile ] );
[ strBuf, sr ] = audioread( [ strPath, strFile ] );
idlSize        = length( idlBuf );
strSize        = length( strBuf );
halfWinSize    = winSize / 2;
overlapSamps   = winSize - ( winSize / overlap );
hopSize        = winSize / overlap;
hopMax         = length( idlBuf ) - winSize;
```

```
idlMsec         = ( idlSize / sr ) * 1000;
gridMsec        = 500;
gridHops        = gridMsec / ( ( ( winSize / overlap ) / sr ) * 1000 );
hopMsec         = ( hopSize / sr ) * 1000;
frames          = ( idlSize / winSize ) * overlap - ( overlap - 1 );
rangeMsec       = detail * idlMsec;
headMsec        = ( headLoc - ( detail / 2 ) ) * idlMsec;
midMsec         = ( midLoc  - ( detail / 2 ) ) * idlMsec;
tailMsec        = ( tailLoc - ( detail / 2 ) ) * idlMsec;
headMsec        = headMsec + rangeMsec * 0.25;
midMsec         = midMsec  + rangeMsec * 0.25;
tailMsec        = tailMsec + rangeMsec * 0.25;
winSum          = sum( window );

% trim or pad stretch buffer to match ideal buffer length
if idlSize > strSize
    strBuf = [ strBuf ; zeros( idlSize - strSize, 1 ) ];
elseif strSize > idlSize
    strBuf = strBuf( 1 : length( idlBuf ) );
end

% Normalization
%--------------------------------------------------------------------------

% get file metadata
idlInfo = audioinfo( [ idlPath, idlFile ] );
strInfo = audioinfo( [ strPath, strFile ] );

% get file normalization coefficient
idlFileNorm = str2num( idlInfo.Title );
strFileNorm = str2num( strInfo.Title );

% de-normalize normalized .wav file data
idlRaw = idlBuf / idlFileNorm;
strRaw = strBuf / strFileNorm;

% make hann window
normWin = hann( round( idlSize / 2 ) );

% copy and window middle of files
idlNormWin = idlRaw( idlSize * 0.25 : idlSize * 0.75 - 1 ) .* normWin;
strNormWin = strRaw( idlSize * 0.25 : idlSize * 0.75 - 1 ) .* normWin;

% calculate rms of windowed file middles
idlRms = rms( idlNormWin );
strRms = rms( strNormWin );

% calculate normalization scalars
strScale = 1 / ( strRms / idlRms );

% scale stretch buffer to match ideal buffer
idlNorm = idlRaw;
```

```
strNorm = strRaw * strScale;

% Analysis
%-------------------------------------------------------------------------
idlGram = spectrogram( idlNorm, window, overlapSamps, winSize, sr );
strGram = spectrogram( strNorm, window, overlapSamps, winSize, sr );

% calculate magnitude spectra
idlMag = abs( idlGram );
strMag = abs( strGram );

% calculate amplitude spectra
idlAmp = ( idlMag / winSum );
strAmp = ( strMag / winSum );

% calculate sones spectra
idlSones = idlAmp .^ 0.6;
strSones = strAmp .^ 0.6;

% copy spectra to plot arrays
idlPlot = idlSones;
strPlot = strSones;

% calculate ideal plot offset
offset = 0;
offset = abs( min( [ min( idlPlot ) min( strPlot ) ] ) );

% offset plots
idlPlot = idlPlot + offset;
strPlot = strPlot + offset;

% calculate difference between plots
diffPlot = abs( idlPlot - strPlot );

% calculate cropping values
side = round( frames * detail );
head = round( frames * headLoc );
mid  = round( frames * midLoc );
tail = round( frames * tailLoc );
range = side * 2;

% copy portions of spectrogram
diffHead = diffPlot( :, [ ( head - side ) : ( head + side ) ] );
diffMid  = diffPlot( :, [ ( mid  - side ) : ( mid  + side ) ] );
diffTail = diffPlot( :, [ ( tail - side ) : ( tail + side ) ] );

% find min and max values of difference plot
diffMax = max( diffPlot(:) );
diffMin = min( diffPlot(:) );
```

```
% Plot
%--------------------------------------------------------------------------
fontName   = 'Times New Roman';
fontSize   = 12;

fig = figure( 1 );
set( fig, 'Name', name );
set( fig, 'Position', [ 0 0 800 500 ] );
set( fig, 'defaultAxesFontName', fontName );
set( fig, 'defaultTextFontName', fontName );

colormap( [ [ 1 : -0.2/64 : 0.8 ]; ( [ 1 : -0.5/64 : 0.5 ] .^ 10 ) ; [ 1 : -
0.5/64 : 0.5 ] .^ 20 ]' );

% Main Spectrogram
%--------------------------------------------------------------------------
set( subplot( 2, 1, 1 ), 'Position', [ 0.1, 0.5, 0.8, 0.3666 ] );
imagesc( diffPlot );
caxis( 'auto' );
axis xy;

% grid and labels
axis( [ 0 length( diffPlot ) 1 halfWinSize ] );
set( gca, 'XTick', [ 0 : gridHops : length( diffPlot ) ] );
set( gca, 'XTickLabel', [ 0 : gridMsec : idlMsec ] );
set( gca, 'YTick', [ 1 : ( halfWinSize / ( sr / 4000 ) ) : halfWinSize ] );
set( gca, 'YTickLabel', [ 0 : 2000 : ( sr / 2 ) ] );

% legend
bar = colorbar( 'location', 'East' );
set( get( bar, 'YLabel'), 'String', 'Sones' );
set( get( bar, 'YLabel'), 'Position', [ 0.75 ( diffMax * 0.11 ) ] );

% Head Spectrogram
%--------------------------------------------------------------------------
set( subplot( 2, 3, 4 ), 'Position', [ 0.1, 0.1, 0.26, 0.3666 ] );
imagesc( diffHead );
caxis( [ diffMin diffMax ] );
axis xy;

set( gca, 'XTick', [ range * 0.25 : ( range / 4 ) : range * 0.75 ] );
set( gca, 'XTickLabel', [ headMsec : rangeMsec / 4 : headMsec + rangeMsec ] );
set( gca, 'YTick', [ 1 : ( halfWinSize / ( sr / 4000 ) ) : halfWinSize ] );
set( gca, 'YTickLabel', [ 0 : 2000 : ( sr / 2 ) ] );
ylabel( 'Frequency' );
yLab = ylabel( 'Frequency' );
text( range * 0.05, halfWinSize * 0.93, ' Head ', 'EdgeColor', 'k',
'BackgroundColor', 'w' );

% Mid Spectrogram
%--------------------------------------------------------------------------
set( subplot( 2, 3, 5 ), 'Position', [ 0.3666, 0.1, 0.2666, 0.3666 ] );
```

```
imagesc( diffMid );
caxis( [ diffMin diffMax ] );
axis xy;

set( gca, 'XTick', [ range * 0.25 : ( range / 4 ) : range * 0.75 ] );
set( gca, 'XTickLabel', [ midMsec : rangeMsec / 4 : midMsec + rangeMsec ] );
set( gca, 'YTick', [ 1 : ( halfWinSize / ( sr / 4000 ) ) : halfWinSize ] );
set( gca, 'YTickLabel', [] );
xlabel( 'Milliseconds' );
text( range * 0.5, halfWinSize * 0.93, ' Middle ', 'EdgeColor', 'k',
'BackgroundColor', 'w', 'HorizontalAlignment', 'center' );

% Tail Spectrogram
%-------------------------------------------------------------------------
set( subplot( 2, 3, 6 ), 'Position', [ 0.6399, 0.1, 0.26, 0.3666 ] );
imagesc( diffTail );
caxis( [ diffMin diffMax ] );
axis xy;

set( gca, 'XTick', [ range * 0.25 : ( range / 4 ) : range * 0.75 ] );
set( gca, 'XTickLabel', [ tailMsec : rangeMsec / 4 : tailMsec + rangeMsec ] );
set( gca, 'YTick', [ 1 : ( halfWinSize / ( sr / 4000 ) ) : halfWinSize ] );
set( gca, 'YTickLabel', [] );
text( range * 1.02, halfWinSize * 0.93, ' Tail ', 'EdgeColor', 'k',
'BackgroundColor', 'w', 'HorizontalAlignment', 'right' );

% tighten up figure borders
tightfig();

set( yLab, 'Position', [ range * -0.15, ( halfWinSize * 1.05 ) ] );

% Save Data to Files
%-------------------------------------------------------------------------


% get file name
[ a, fileName, b ] = fileparts( strFile );

% write plot to file
hgexport( fig, [ fileName, '.spect.eps' ] );

% write error data to files
csvFile = sprintf( '%s.spect.csv', fileName );
csvwrite( csvFile, diffPlot );

% EOF
```

### 8.4.8 – Average Spectrum Error Comparison

```
function avg_error( fileName )
%-------------------------------------------------------------------------
% Average Spectrum Error Comparison
%
% Cooper Baker - 2014
%-------------------------------------------------------------------------

close all;

name = 'Average Spectrum Error';

% set path
p = genpath( '../../analyze' );
addpath( p );

% get sample rate
fileInfo = audioinfo( [ fileName, '.classic.wav' ] );
sr = fileInfo.SampleRate;

% make file names
rawFile  = sprintf( '%s.classic.avg.err.csv' , fileName );
lockFile = sprintf( '%s.lock.avg.err.csv', fileName );
peakFile = sprintf( '%s.peak.avg.err.csv', fileName );
bankFile = sprintf( '%s.bank.avg.err.csv', fileName );
solaFile = sprintf( '%s.sola.avg.err.csv', fileName );
olaFile  = sprintf( '%s.ola.avg.err.csv' , fileName );

% load file data
rawData  = csvread( rawFile  );
lockData = csvread( lockFile );
peakData = csvread( peakFile );
bankData = csvread( bankFile );
solaData = csvread( solaFile );
olaData  = csvread( olaFile  );

% get data size
size = length( rawData );

% calculate data maximums
rawMax  = max( rawData  );
lockMax = max( lockData );
peakMax = max( peakData );
bankMax = max( bankData );
solaMax = max( solaData );
olaMax  = max( olaData  );

% calculate data averages
rawAvg  = sum( rawData  ) / size;
lockAvg = sum( lockData ) / size;
peakAvg = sum( peakData ) / size;
```

```
bankAvg = sum( bankData ) / size;
solaAvg = sum( solaData ) / size;
olaAvg  = sum( olaData  ) / size;

% find maximum maximum and minimum maximum
MaxMax = max( [ rawMax lockMax peakMax bankMax solaMax olaMax ] );
MinMax = min( [ rawMax lockMax peakMax bankMax solaMax olaMax ] );

% find maximum average and minimum average
MaxAvg = max( [ rawAvg lockAvg peakAvg bankAvg solaAvg olaAvg ] );
MinAvg = min( [ rawAvg lockAvg peakAvg bankAvg solaAvg olaAvg ] );

% graph formatting
fontName   = 'Times New Roman';
fontSize   = 12;
xTxt       = size * .99;
yTxt       = MaxMax * .9;

% set up plot window
fig = figure( 1 );
set( fig, 'Name', sprintf( '%s - %s.wav', name, fileName ) );
set( fig, 'Position', [ 0 0 800 1200 ] );
set( fig, 'defaultAxesFontName', fontName );
set( fig, 'defaultTextFontName', fontName );
set( fig, 'defaultlinelinewidth', 3 );
set( fig, 'defaultaxescolororder', [ 0.8 0.2 0.2 ] );

% plot fft ifft raw
title = 'Classic';
set( subplot( 1, 1, 1 ), 'Position', [ 0.1, 5.5/7, 0.8, 1/7.333 ] );
plot( rawData );
axis( [ 0 size 0 MaxMax ] );
grid on;
set( gca, 'XTick', [ 0 : ( size / ( sr / 4000 ) ) : size ] );
set( gca, 'XTickLabel', [] );
temp = get( gca );
yticks = get( gca, 'YTick' );
set( gca, 'YTickLabel', yticks );
legStr = sprintf( ' %s - Avg: %2.5f Max: %2.5f ', title, rawAvg, rawMax );
text( xTxt, yTxt, legStr, 'HorizontalAlignment', 'right', 'BackgroundColor',
'w', 'EdgeColor', 'k' );

% plot fft ifft phase locked
title = 'Lock';
set( subplot( 6, 1, 2 ), 'Position', [ 0.1, 4.5/7, 0.8, 1/7.333 ] );
plot( lockData );
axis( [ 0 size 0 MaxMax ] );
grid on;
set( gca, 'XTick', [ 0 : ( size / ( sr / 4000 ) ) : size ] );
set( gca, 'XTickLabel', [] );
temp = get( gca );
yticks = get( gca, 'YTick' );
```

```
set( gca, 'YTickLabel', yticks );
legStr = sprintf( ' %s - Avg: %2.5f Max: %2.5f ', title, lockAvg, lockMax );
text( xTxt, yTxt, legStr, 'HorizontalAlignment', 'right', 'BackgroundColor',
'w', 'EdgeColor', 'k' );

% plot fft ifft peak tracking
title = 'Peak';
set( subplot( 6, 1, 3 ), 'Position', [ 0.1, 3.5/7, 0.8, 1/7.333 ] );
plot( peakData );
axis( [ 0 size 0 MaxMax ] );
grid on;
set( gca, 'XTick', [ 0 : ( size / ( sr / 4000 ) ) : size ] );
set( gca, 'XTickLabel', [] );
temp = get( gca );
yticks = get( gca, 'YTick' );
set( gca, 'YTickLabel', yticks );
legStr = sprintf( ' %s - Avg: %2.5f Max: %2.5f ', title, peakAvg, peakMax );
text( xTxt, yTxt, legStr, 'HorizontalAlignment', 'right', 'BackgroundColor',
'w', 'EdgeColor', 'k' );

% plot fft oscillator bank
title = 'Bank';
set( subplot( 6, 1, 4 ), 'Position', [ 0.1, 2.5/7, 0.8, 1/7.333 ] );
plot( bankData );
axis( [ 0 size 0 MaxMax ] );
grid on;
set( gca, 'XTick', [ 0 : ( size / ( sr / 4000 ) ) : size ] );
set( gca, 'XTickLabel', [] );
temp = get( gca );
yticks = get( gca, 'YTick' );
set( gca, 'YTickLabel', yticks );
legStr = sprintf( ' %s - Avg: %2.5f Max: %2.5f ', title, bankAvg, bankMax );
text( xTxt, yTxt, legStr, 'HorizontalAlignment', 'right', 'BackgroundColor',
'w', 'EdgeColor', 'k' );

% plot synchronous overlap add
title = 'Sola';
set( subplot( 6, 1, 5 ), 'Position', [ 0.1, 1.5/7, 0.8, 1/7.333 ] );
plot( solaData );
axis( [ 0 size 0 MaxMax ] );
grid on;
set( gca, 'XTick', [ 0 : ( size / ( sr / 4000 ) ) : size ] );
set( gca, 'XTickLabel', [] );
temp = get( gca );
yticks = get( gca, 'YTick' );
set( gca, 'YTickLabel', yticks );
legStr = sprintf( ' %s - Avg: %2.5f Max: %2.5f ', title, solaAvg, solaMax );
text( xTxt, yTxt, legStr, 'HorizontalAlignment', 'right', 'BackgroundColor',
'w', 'EdgeColor', 'k' );

% plot overlap add
title = 'Ola';
```

```
set( subplot( 6, 1, 6 ), 'Position', [ 0.1, 0.5/7, 0.8, 1/7.333 ] );
plot( olaData );
axis( [ 0 size 0 MaxMax ] );
grid on;
set( gca, 'XTick', [ 0 : ( size / ( sr / 4000 ) ) : size ] );
set( gca, 'XTickLabel', [ 0 : 2000 : ( sr / 2 ) ] );
temp = get( gca );
yticks = get( gca, 'YTick' );
set( gca, 'YTickLabel', yticks );
xlabel( 'Frequency' );
ylabel( 'Sones' );
legStr = sprintf( ' %s - Avg: %2.5f Max: %2.5f ', title, olaAvg, olaMax );
text( xTxt, yTxt, legStr, 'HorizontalAlignment', 'right', 'BackgroundColor',
'w', 'EdgeColor', 'k' );

% tighten up borders
tightfig();

% move y label to middle of graphs
ylabel( 'Sones', 'Position', [ -size / 23 ( ( MaxMax * 6.1666 ) / 2 ) ] );

% write plot to file
hgexport( fig, [ fileName, '.err.avg.eps' ] );

% EOF
```

### 8.4.9 – Moving Spectral Average Error Comparison

```
function mov_error( fileName )
%-------------------------------------------------------------------------
% Moving Spectral Average Error Comparison
%
% Cooper Baker - 2014
%-------------------------------------------------------------------------

close all;

name = 'Moving Spectral Average Error';

% set path
p = genpath( '../../analyze' );
addpath( p );

% get audio file info
fileInfo = audioinfo( [ fileName, '.classic.wav' ] );
sr       = fileInfo.SampleRate;
samps    = fileInfo.TotalSamples;
winSize  = 512;
overlap  = 4;

% make file names
rawFile  = sprintf( '%s.classic.mov.err.csv' , fileName );
```

```
lockFile = sprintf( '%s.lock.mov.err.csv', fileName );
peakFile = sprintf( '%s.peak.mov.err.csv', fileName );
bankFile = sprintf( '%s.bank.mov.err.csv', fileName );
solaFile = sprintf( '%s.sola.mov.err.csv', fileName );
olaFile  = sprintf( '%s.ola.mov.err.csv' , fileName );

% load file data
rawData  = csvread( rawFile  );
lockData = csvread( lockFile );
peakData = csvread( peakFile );
bankData = csvread( bankFile );
solaData = csvread( solaFile );
olaData  = csvread( olaFile  );

% get data size
size = length( rawData );

% calculate msec grid values
msec     = ( samps / sr ) * 1000;
gridMsec = 500;
gridHops = gridMsec / ( ( ( winSize / overlap ) / sr ) * 1000 );
hopSize  = winSize / overlap;
hopMsec  = ( hopSize / sr ) * 1000;

% calculate data maximums
rawMax  = max( rawData  );
lockMax = max( lockData );
peakMax = max( peakData );
bankMax = max( bankData );
solaMax = max( solaData );
olaMax  = max( olaData  );

% calculate data averages
rawAvg  = sum( rawData  ) / size;
lockAvg = sum( lockData ) / size;
peakAvg = sum( peakData ) / size;
bankAvg = sum( bankData ) / size;
solaAvg = sum( solaData ) / size;
olaAvg  = sum( olaData  ) / size;

% find maximum maximum and minimum maximum
MaxMax = max( [ rawMax lockMax peakMax bankMax solaMax olaMax ] );
MinMax = min( [ rawMax lockMax peakMax bankMax solaMax olaMax ] );

% find maximum average and minimum average
MaxAvg = max( [ rawAvg lockAvg peakAvg bankAvg solaAvg olaAvg ] );
MinAvg = min( [ rawAvg lockAvg peakAvg bankAvg solaAvg olaAvg ] );

% graph formatting
fontName   = 'Times New Roman';
fontSize   = 12;
xTxt       = size * 0.01;
```

```
yTxt       = MaxMax * .9;

% set up plot window
fig = figure( 1 );
set( fig, 'Name', sprintf( '%s - %s.wav', name, fileName ) );
set( fig, 'Position', [ 0 0 800 1200 ] );
set( fig, 'defaultAxesFontName', fontName );
set( fig, 'defaultTextFontName', fontName );
set( fig, 'defaultlinelinewidth', 3 );
set( fig, 'defaultaxescolororder', [ 0.8 0.2 0.2 ] );

% plot fft ifft raw
title = 'Classic';
set( subplot( 1, 1, 1 ), 'Position', [ 0.1, 5.5/7, 0.8, 1/7.333 ] );
plot( rawData );
axis( [ 0 size 0 MaxMax ] );
grid on;
set( gca, 'XTick', [ 0 : gridHops : size ] );
set( gca, 'XTickLabel', [] );
legStr = sprintf( ' %s - Avg: %2.5f Max: %2.5f ', title, rawAvg, rawMax );
text( xTxt, yTxt, legStr, 'BackgroundColor', 'w', 'EdgeColor', 'k' );

% plot fft ifft phase locked
title = 'Lock';
set( subplot( 6, 1, 2 ), 'Position', [ 0.1, 4.5/7, 0.8, 1/7.333 ] );
plot( lockData );
axis( [ 0 size 0 MaxMax ] );
grid on;
set( gca, 'XTick', [ 0 : gridHops : size ] );
set( gca, 'XTickLabel', [] );
legStr = sprintf( ' %s - Avg: %2.5f Max: %2.5f ', title, lockAvg, lockMax );
text( xTxt, yTxt, legStr, 'BackgroundColor', 'w', 'EdgeColor', 'k' );

% plot fft ifft peak tracking
title = 'Peak';
set( subplot( 6, 1, 3 ), 'Position', [ 0.1, 3.5/7, 0.8, 1/7.333 ] );
plot( peakData );
axis( [ 0 size 0 MaxMax ] );
grid on;
set( gca, 'XTick', [ 0 : gridHops : size ] );
set( gca, 'XTickLabel', [] );
legStr = sprintf( ' %s - Avg: %2.5f Max: %2.5f ', title, peakAvg, peakMax );
text( xTxt, yTxt, legStr, 'BackgroundColor', 'w', 'EdgeColor', 'k' );

% plot fft oscillator bank
title = 'Bank';
set( subplot( 6, 1, 4 ), 'Position', [ 0.1, 2.5/7, 0.8, 1/7.333 ] );
plot( bankData );
axis( [ 0 size 0 MaxMax ] );
grid on;
set( gca, 'XTick', [ 0 : gridHops : size ] );
set( gca, 'XTickLabel', [] );
```

```
legStr = sprintf( ' %s - Avg: %2.5f Max: %2.5f ', title, bankAvg, bankMax );
text( xTxt, yTxt, legStr, 'BackgroundColor', 'w', 'EdgeColor', 'k' );

% plot synchronous overlap add
title = 'Sola';
set( subplot( 6, 1, 5 ), 'Position', [ 0.1, 1.5/7, 0.8, 1/7.333 ] );
plot( solaData );
axis( [ 0 size 0 MaxMax ] );
grid on;
set( gca, 'XTick', [ 0 : gridHops : size ] );
set( gca, 'XTickLabel', [] );
legStr = sprintf( ' %s - Avg: %2.5f Max: %2.5f ', title, solaAvg, solaMax );
text( xTxt, yTxt, legStr, 'BackgroundColor', 'w', 'EdgeColor', 'k' );

% plot overlap add
title = 'Ola';
set( subplot( 6, 1, 6 ), 'Position', [ 0.1, 0.5/7, 0.8, 1/7.333 ] );
plot( olaData );
axis( [ 0 size 0 MaxMax ] );
grid on;
set( gca, 'XTick', [ 0 : gridHops : size ] );
set( gca, 'XTickLabel', [ 0 : gridMsec : msec ] );
xlabel( 'Milliseconds' );
ylabel( 'Sones' );
legStr = sprintf( ' %s - Avg: %2.5f Max: %2.5f ', title, olaAvg, olaMax );
text( xTxt, yTxt, legStr, 'BackgroundColor', 'w', 'EdgeColor', 'k' );

% tighten up borders
tightfig();

% move y label to middle of graphs
ylabel( 'Sones', 'Position', [ -size / 23 ( ( MaxMax * 6.1666 ) / 2 ) ] );

% write plot to file
hgexport( fig, [ fileName, '.err.mov.eps' ] );

% EOF
```

### 8.4.10 – Moving Spectral Average Error Comparison Detail

```
function mov_error_detail( fileName )
%-------------------------------------------------------------------------
% Moving Spectral Average Error Comparision Detail
%
% Cooper Baker - 2014
%-------------------------------------------------------------------------

close all;

name = 'Moving Spectral Average Error';

% detail settings
headLoc = 0.25;
midLoc  = 0.5;
tailLoc = 0.75;
detail  = 0.02;

% set path
p = genpath( '../../analyze' );
addpath( p );
p = genpath( '../../tones' );
addpath( p );


% get audio file info
fileInfo = audioinfo( [ fileName, '_ideal.wav' ] );
sr       = fileInfo.SampleRate;
samps    = fileInfo.TotalSamples;
winSize  = 512;
overlap  = 4;

% make file names
rawFile  = sprintf( '%s.classic.mov.err.csv' , fileName );
lockFile = sprintf( '%s.lock.mov.err.csv', fileName );
peakFile = sprintf( '%s.peak.mov.err.csv', fileName );
bankFile = sprintf( '%s.bank.mov.err.csv', fileName );
solaFile = sprintf( '%s.sola.mov.err.csv', fileName );
olaFile  = sprintf( '%s.ola.mov.err.csv' , fileName );

% load file data
rawData  = csvread( rawFile  );
lockData = csvread( lockFile );
peakData = csvread( peakFile );
bankData = csvread( bankFile );
solaData = csvread( solaFile );
olaData  = csvread( olaFile  );

% get data size
size = length( rawData );
```

```
% calculate msec grid values
msec      = ( samps / sr ) * 1000;
gridMsec  = 500;
gridHops  = gridMsec / ( ( ( winSize / overlap ) / sr ) * 1000 );
hopSize   = winSize / overlap;
hopMsec   = ( hopSize / sr ) * 1000;
rangeMsec = detail * msec;
headMsec  = ( headLoc - ( detail / 2 ) ) * msec;
midMsec   = ( midLoc  - ( detail / 2 ) ) * msec;
tailMsec  = ( tailLoc - ( detail / 2 ) ) * msec;
headMsec  = headMsec + rangeMsec * 0.25;
midMsec   = midMsec  + rangeMsec * 0.25;
tailMsec  = tailMsec + rangeMsec * 0.25;

% calculate data maximums
rawMax  = max( rawData  );
lockMax = max( lockData );
peakMax = max( peakData );
bankMax = max( bankData );
solaMax = max( solaData );
olaMax  = max( olaData  );

% calculate data averages
rawAvg  = sum( rawData  ) / size;
lockAvg = sum( lockData ) / size;
peakAvg = sum( peakData ) / size;
bankAvg = sum( bankData ) / size;
solaAvg = sum( solaData ) / size;
olaAvg  = sum( olaData  ) / size;

% find maximum maximum and minimum maximum
MaxMax = max( [ rawMax lockMax peakMax bankMax solaMax olaMax ] );
MinMax = min( [ rawMax lockMax peakMax bankMax solaMax olaMax ] );

% find maximum average and minimum average
MaxAvg = max( [ rawAvg lockAvg peakAvg bankAvg solaAvg olaAvg ] );
MinAvg = min( [ rawAvg lockAvg peakAvg bankAvg solaAvg olaAvg ] );

% calculate cropping values
side = round( size * detail );
head = round( size * headLoc );
mid  = round( size * midLoc );
tail = round( size * tailLoc );
range = side * 2;

% copy detail sections of data
rawHead  = rawData ( head - side : head + side );
rawMid   = rawData ( mid  - side : mid  + side );
rawTail  = rawData ( tail - side : tail + side );
lockHead = lockData( head - side : head + side );
lockMid  = lockData( mid  - side : mid  + side );
lockTail = lockData( tail - side : tail + side );
```

```
peakHead = peakData( head - side : head + side );
peakMid  = peakData( mid  - side : mid  + side );
peakTail = peakData( tail - side : tail + side );
bankHead = bankData( head - side : head + side );
bankMid  = bankData( mid  - side : mid  + side );
bankTail = bankData( tail - side : tail + side );
solaHead = solaData( head - side : head + side );
solaMid  = solaData( mid  - side : mid  + side );
solaTail = solaData( tail - side : tail + side );
olaHead  = olaData ( head - side : head + side );
olaMid   = olaData ( mid  - side : mid  + side );
olaTail  = olaData ( tail - side : tail + side );

% graph formatting
fontName   = 'Times New Roman';
fontSize   = 12;
xTxt       = range * 0.04;
yTxt       = MaxMax * .82;

% set up plot window
fig = figure( 1 );
set( fig, 'Name', sprintf( '%s - %s.wav', name, fileName ) );
set( fig, 'Position', [ 0 0 800 1000 ] );
set( fig, 'defaultAxesFontName', fontName );
set( fig, 'defaultTextFontName', fontName );
set( fig, 'defaultlinelinewidth', 3 );
set( fig, 'defaultaxescolororder', [ 0.8 0.2 0.2 ] );

% classic
%--------------------------------------------------------------------------

% head
algo = 'Classic';
set( subplot( 6, 3, 1 ), 'Position', [ 0.1, 5.5/7, 0.26, 1/7.333 ] );
plot( rawHead );
axis( [ 0 range 0 MaxMax ] );
grid on;
set( gca, 'XTick', [ range * 0.25 : ( range / 4 ) : range * 0.75 ] );
set( gca, 'XTickLabel', [] );
legStr = sprintf( ' %s\n Avg: %2.5f \n Max: %2.5f ', algo, rawAvg, rawMax );
text( xTxt, yTxt, legStr, 'BackgroundColor', 'w', 'EdgeColor', 'k' );
title( 'Head', 'FontSize', fontSize );

% mid
set( subplot( 6, 3, 2 ), 'Position', [ 0.3666, 5.5/7, 0.26, 1/7.333 ] );
plot( rawMid );
axis( [ 0 range 0 MaxMax ] );
grid on;
set( gca, 'XTick', [ range * 0.25 : ( range / 4 ) : range * 0.75 ] );
set( gca, 'XTickLabel', [] );
set( gca, 'YTickLabel', [] );
title( 'Middle', 'FontSize', fontSize );
```

```
% tail
set( subplot( 6, 3, 3 ), 'Position', [ 0.6333, 5.5/7, 0.26, 1/7.333 ] );
plot( rawTail );
axis( [ 0 range 0 MaxMax ] );
grid on;
set( gca, 'XTick', [ range * 0.25 : ( range / 4 ) : range * 0.75 ] );
set( gca, 'XTickLabel', [] );
set( gca, 'YTickLabel', [] );
title( 'Tail', 'FontSize', fontSize );

% lock
%--------------------------------------------------------------------------

% head
algo = 'Lock';
set( subplot( 6, 3, 4 ), 'Position', [ 0.1, 4.5/7, 0.26, 1/7.333 ] );
plot( lockHead );
axis( [ 0 range 0 MaxMax ] );
grid on;
set( gca, 'XTick', [ range * 0.25 : ( range / 4 ) : range * 0.75 ] );
set( gca, 'XTickLabel', [] );
legStr = sprintf( ' %s\n Avg: %2.5f \n Max: %2.5f ', algo, lockAvg, lockMax );
text( xTxt, yTxt, legStr, 'BackgroundColor', 'w', 'EdgeColor', 'k' );

% mid
set( subplot( 6, 3, 5 ), 'Position', [ 0.3666, 4.5/7, 0.26, 1/7.333 ] );
plot( lockMid );
axis( [ 0 range 0 MaxMax ] );
grid on;
set( gca, 'XTick', [ range * 0.25 : ( range / 4 ) : range * 0.75 ] );
set( gca, 'XTickLabel', [] );
set( gca, 'YTickLabel', [] );

% tail
set( subplot( 6, 3, 6 ), 'Position', [ 0.6333, 4.5/7, 0.26, 1/7.333 ] );
plot( lockTail );
axis( [ 0 range 0 MaxMax ] );
grid on;
set( gca, 'XTick', [ range * 0.25 : ( range / 4 ) : range * 0.75 ] );
set( gca, 'XTickLabel', [] );
set( gca, 'YTickLabel', [] );

% peak
%--------------------------------------------------------------------------

% head
algo = 'Peak';
set( subplot( 6, 3, 7 ), 'Position', [ 0.1, 3.5/7, 0.26, 1/7.333 ] );
plot( peakHead );
axis( [ 0 range 0 MaxMax ] );
grid on;
set( gca, 'XTick', [ range * 0.25 : ( range / 4 ) : range * 0.75 ] );
```

```
set( gca, 'XTickLabel', [] );
legStr = sprintf( ' %s\n Avg: %2.5f \n Max: %2.5f ', algo, peakAvg, peakMax );
text( xTxt, yTxt, legStr, 'BackgroundColor', 'w', 'EdgeColor', 'k' );

% mid
set( subplot( 6, 3, 8 ), 'Position', [ 0.3666, 3.5/7, 0.26, 1/7.333 ] );
plot( peakMid );
axis( [ 0 range 0 MaxMax ] );
grid on;
set( gca, 'XTick', [ range * 0.25 : ( range / 4 ) : range * 0.75 ] );
set( gca, 'XTickLabel', [] );
set( gca, 'YTickLabel', [] );

% tail
set( subplot( 6, 3, 9 ), 'Position', [ 0.6333, 3.5/7, 0.26, 1/7.333 ] );
plot( peakTail );
axis( [ 0 range 0 MaxMax ] );
grid on;
set( gca, 'XTick', [ range * 0.25 : ( range / 4 ) : range * 0.75 ] );
set( gca, 'XTickLabel', [] );
set( gca, 'YTickLabel', [] );

% bank
%--------------------------------------------------------------------------

% head
algo = 'Bank';
set( subplot( 6, 3, 10 ), 'Position', [ 0.1, 2.5/7, 0.26, 1/7.333 ] );
plot( bankHead );
axis( [ 0 range 0 MaxMax ] );
grid on;
set( gca, 'XTick', [ range * 0.25 : ( range / 4 ) : range * 0.75 ] );
set( gca, 'XTickLabel', [] );
legStr = sprintf( ' %s\n Avg: %2.5f \n Max: %2.5f ', algo, bankAvg, bankMax );
text( xTxt, yTxt, legStr, 'BackgroundColor', 'w', 'EdgeColor', 'k' );

% mid
set( subplot( 6, 3, 11 ), 'Position', [ 0.3666, 2.5/7, 0.26, 1/7.333 ] );
plot( bankMid );
axis( [ 0 range 0 MaxMax ] );
grid on;
set( gca, 'XTick', [ range * 0.25 : ( range / 4 ) : range * 0.75 ] );
set( gca, 'XTickLabel', [] );
set( gca, 'YTickLabel', [] );

% tail
set( subplot( 6, 3, 12 ), 'Position', [ 0.6333, 2.5/7, 0.26, 1/7.333 ] );
plot( bankTail );
axis( [ 0 range 0 MaxMax ] );
grid on;
set( gca, 'XTick', [ range * 0.25 : ( range / 4 ) : range * 0.75 ] );
set( gca, 'XTickLabel', [] );
```

```
set( gca, 'YTickLabel', [] );

% sola
%-------------------------------------------------------------------------

% head
algo = 'Sola';
set( subplot( 6, 3, 13 ), 'Position', [ 0.1, 1.5/7, 0.26, 1/7.333 ] );
plot( solaHead );
axis( [ 0 range 0 MaxMax ] );
grid on;
set( gca, 'XTick', [ range * 0.25 : ( range / 4 ) : range * 0.75 ] );
set( gca, 'XTickLabel', [] );
legStr = sprintf( ' %s\n Avg: %2.5f \n Max: %2.5f ', algo, solaAvg, solaMax );
text( xTxt, yTxt, legStr, 'BackgroundColor', 'w', 'EdgeColor', 'k' );

% mid
set( subplot( 6, 3, 14 ), 'Position', [ 0.3666, 1.5/7, 0.26, 1/7.333 ] );
plot( solaMid );
axis( [ 0 range 0 MaxMax ] );
grid on;
set( gca, 'XTick', [ range * 0.25 : ( range / 4 ) : range * 0.75 ] );
set( gca, 'XTickLabel', [] );
set( gca, 'YTickLabel', [] );

% tail
set( subplot( 6, 3, 15 ), 'Position', [ 0.6333, 1.5/7, 0.26, 1/7.333 ] );
plot( solaTail );
axis( [ 0 range 0 MaxMax ] );
grid on;
set( gca, 'XTick', [ range * 0.25 : ( range / 4 ) : range * 0.75 ] );
set( gca, 'XTickLabel', [] );
set( gca, 'YTickLabel', [] );

% ola
%-------------------------------------------------------------------------

% head
algo = 'Ola';
set( subplot( 6, 3, 16 ), 'Position', [ 0.1, 0.5/7, 0.26, 1/7.333 ] );
plot( olaHead );
axis( [ 0 range 0 MaxMax ] );
grid on;
set( gca, 'XTick', [ range * 0.25 : ( range / 4 ) : range * 0.75 ] );
set( gca, 'XTickLabel', [ headMsec : rangeMsec / 4 : headMsec + rangeMsec ] );
legStr = sprintf( ' %s\n Avg: %2.5f \n Max: %2.5f ', algo, olaAvg, olaMax );
text( xTxt, yTxt, legStr, 'BackgroundColor', 'w', 'EdgeColor', 'k' );
yLab = ylabel( 'Sones' );

% mid
set( subplot( 6, 3, 17 ), 'Position', [ 0.3666, 0.5/7, 0.26, 1/7.333 ] );
plot( olaMid );
```

```
axis( [ 0 range 0 MaxMax ] );
grid on;
set( gca, 'XTick', [ range * 0.25 : ( range / 4 ) : range * 0.75 ] );
set( gca, 'XTickLabel', [ midMsec : rangeMsec / 4 : midMsec + rangeMsec ] );
set( gca, 'YTickLabel', [] );
xlabel( 'Milliseconds' );

% tail
set( subplot( 6, 3, 18 ), 'Position', [ 0.6333, 0.5/7, 0.26, 1/7.333 ] );
plot( olaTail );
axis( [ 0 range 0 MaxMax ] );
grid on;
set( gca, 'XTick', [ range * 0.25 : ( range / 4 ) : range * 0.75 ] );
set( gca, 'XTickLabel', [ tailMsec : rangeMsec / 4 : tailMsec + rangeMsec ] );
set( gca, 'YTickLabel', [] );

% tighten up graph borders
tightfig();

% move y label
set( yLab, 'Position', [ range * -0.15 ( ( MaxMax * 6.25 ) / 2 ) ] );

% write plot to file
hgexport( fig, [ fileName, '.err.mov.zoom.eps' ] );

% EOF
```

### 8.4.11 – Error Spectrogram Comparison

```
function dif_error( fileName )
%-------------------------------------------------------------------------
% Error Spectrogram Comparison
%
% Cooper Baker - 2014
%-------------------------------------------------------------------------

close all;

name = 'Error Spectrograms';

% set path
p = genpath( '../../analyze' );
addpath( p );

% get audio file info
fileInfo    = audioinfo( [ fileName, '.classic.wav' ] );
sr          = fileInfo.SampleRate;
samps       = fileInfo.TotalSamples;
winSize     = 512;
halfWinSize = winSize / 2;
overlap     = 4;
```

```
% make file names
rawFile  = sprintf( '%s.classic.spect.csv' , fileName );
lockFile = sprintf( '%s.lock.spect.csv', fileName );
peakFile = sprintf( '%s.peak.spect.csv', fileName );
bankFile = sprintf( '%s.bank.spect.csv', fileName );
solaFile = sprintf( '%s.sola.spect.csv', fileName );
olaFile  = sprintf( '%s.ola.spect.csv' , fileName );

% load file data
rawData  = csvread( rawFile  );
lockData = csvread( lockFile );
peakData = csvread( peakFile );
bankData = csvread( bankFile );
solaData = csvread( solaFile );
olaData  = csvread( olaFile  );

% get data size
size = length( rawData );

% calculate msec grid values
msec     = ( samps / sr ) * 1000;
gridMsec = 500;
gridHops = gridMsec / ( ( ( winSize / overlap ) / sr ) * 1000 );
hopSize  = winSize / overlap;
hopMsec  = ( hopSize / sr ) * 1000;

% calculate data maximums
rawMax  = max( rawData  );
lockMax = max( lockData );
peakMax = max( peakData );
bankMax = max( bankData );
solaMax = max( solaData );
olaMax  = max( olaData  );

% find maximum maximum and minimum maximum
MaxMax = max( [ rawMax lockMax peakMax bankMax solaMax olaMax ] );
MinMax = min( [ rawMax lockMax peakMax bankMax solaMax olaMax ] );

% graph formatting
fontName   = 'Times New Roman';
fontSize   = 12;
xTxt       = size * 0.015;
yTxt       = halfWinSize * 0.9;

% set up plot window
fig = figure( 1 );
set( fig, 'Name', sprintf( '%s - %s.wav', name, fileName ) );
set( fig, 'Position', [ 0 0 800 1200 ] );
set( fig, 'defaultAxesFontName', fontName );
set( fig, 'defaultTextFontName', fontName );
set( fig, 'defaultlinelinewidth', 3 );
set( fig, 'defaultaxescolororder', [ 0.8 0.2 0.2 ] );
```

```
% color mapping
colormap( [ [ 1 : -0.2/64 : 0.8 ]; ( [ 1 : -0.5/64 : 0.5 ] .^ 10 ) ; [ 1 : -
0.5/64 : 0.5 ] .^ 20 ]' );

% plot fft ifft raw
title = 'Classic';
set( subplot( 1, 1, 1 ), 'Position', [ 0.1, 5.5/7, 0.8, 1/7.333 ] );
imagesc( rawData );
axis xy;
axis( [ 0 length( rawData ) 1 halfWinSize ] );
caxis( [ 0 MaxMax ] );
set( gca, 'XTick', [ 0 : gridHops : size ] );
set( gca, 'XTickLabel', [] );
set( gca, 'YTick', [ 1 : ( halfWinSize / ( sr / 4000 ) ) : halfWinSize ] );
set( gca, 'YTickLabel', [ 0 : 2000 : ( sr / 2 ) ] );
text( xTxt, yTxt, title, 'EdgeColor', 'k', 'BackgroundColor', 'w' );
bar = colorbar( 'location', 'East' );
set( get( bar, 'YLabel'), 'String', 'Sones' );
set( get( bar, 'YLabel'), 'Position', [ 0.75 MaxMax * 0.15 ] );

% plot fft ifft phase locked
title = 'Lock';
set( subplot( 6, 1, 2 ), 'Position', [ 0.1, 4.5/7, 0.8, 1/7.333 ] );
imagesc( lockData );
axis xy;
axis( [ 0 length( rawData ) 1 halfWinSize ] );
caxis( [ 0 MaxMax ] );
set( gca, 'XTick', [ 0 : gridHops : size ] );
set( gca, 'XTickLabel', [] );
set( gca, 'YTick', [ 1 : ( halfWinSize / ( sr / 4000 ) ) : halfWinSize ] );
set( gca, 'YTickLabel', [ 0 : 2000 : ( sr / 2 ) ] );
text( xTxt, yTxt, title, 'EdgeColor', 'k', 'BackgroundColor', 'w' );
bar = colorbar( 'location', 'East' );
set( get( bar, 'YLabel'), 'String', 'Sones' );
set( get( bar, 'YLabel'), 'Position', [ 0.75 MaxMax * 0.15 ] );

% plot fft ifft peak tracking
title = 'Peak';
set( subplot( 6, 1, 3 ), 'Position', [ 0.1, 3.5/7, 0.8, 1/7.333 ] );
imagesc( peakData );
axis xy;
axis( [ 0 length( rawData ) 1 halfWinSize ] );
caxis( [ 0 MaxMax ] );
set( gca, 'XTick', [ 0 : gridHops : size ] );
set( gca, 'XTickLabel', [] );
set( gca, 'YTick', [ 1 : ( halfWinSize / ( sr / 4000 ) ) : halfWinSize ] );
set( gca, 'YTickLabel', [ 0 : 2000 : ( sr / 2 ) ] );
text( xTxt, yTxt, title, 'EdgeColor', 'k', 'BackgroundColor', 'w' );
bar = colorbar( 'location', 'East' );
set( get( bar, 'YLabel'), 'String', 'Sones' );
set( get( bar, 'YLabel'), 'Position', [ 0.75 MaxMax * 0.15 ] );
```

```
% plot fft oscillator bank
title = 'Bank';
set( subplot( 6, 1, 4 ), 'Position', [ 0.1, 2.5/7, 0.8, 1/7.333 ] );
imagesc( bankData );
axis xy;
axis( [ 0 length( rawData ) 1 halfWinSize ] );
caxis( [ 0 MaxMax ] );
set( gca, 'XTick', [ 0 : gridHops : size ] );
set( gca, 'XTickLabel', [] );
set( gca, 'YTick', [ 1 : ( halfWinSize / ( sr / 4000 ) ) : halfWinSize ] );
set( gca, 'YTickLabel', [ 0 : 2000 : ( sr / 2 ) ] );
text( xTxt, yTxt, title, 'EdgeColor', 'k', 'BackgroundColor', 'w' );
bar = colorbar( 'location', 'East' );
set( get( bar, 'YLabel'), 'String', 'Sones' );
set( get( bar, 'YLabel'), 'Position', [ 0.75 MaxMax * 0.15 ] );

% plot synchronous overlap add
title = 'Sola';
set( subplot( 6, 1, 5 ), 'Position', [ 0.1, 1.5/7, 0.8, 1/7.333 ] );
imagesc( solaData );
axis xy;
axis( [ 0 length( rawData ) 1 halfWinSize ] );
caxis( [ 0 MaxMax ] );
set( gca, 'XTick', [ 0 : gridHops : size ] );
set( gca, 'XTickLabel', [] );
set( gca, 'YTick', [ 1 : ( halfWinSize / ( sr / 4000 ) ) : halfWinSize ] );
set( gca, 'YTickLabel', [ 0 : 2000 : ( sr / 2 ) ] );
text( xTxt, yTxt, title, 'EdgeColor', 'k', 'BackgroundColor', 'w' );
bar = colorbar( 'location', 'East' );
set( get( bar, 'YLabel'), 'String', 'Sones' );
set( get( bar, 'YLabel'), 'Position', [ 0.75 MaxMax * 0.15 ] );

% plot overlap add
title = 'Ola';
set( subplot( 6, 1, 6 ), 'Position', [ 0.1, 0.5/7, 0.8, 1/7.333 ] );
imagesc( olaData );
axis xy;
axis( [ 0 length( rawData ) 1 halfWinSize ] );
caxis( [ 0 MaxMax ] );
set( gca, 'XTick', [ 0 : gridHops : size ] );
set( gca, 'XTickLabel', [ 0 : gridMsec : msec ] );
set( gca, 'YTick', [ 1 : ( halfWinSize / ( sr / 4000 ) ) : halfWinSize ] );
set( gca, 'YTickLabel', [ 0 : 2000 : ( sr / 2 ) ] );
text( xTxt, yTxt, title, 'EdgeColor', 'k', 'BackgroundColor', 'w' );
bar = colorbar( 'location', 'East' );
set( get( bar, 'YLabel'), 'String', 'Sones' );
set( get( bar, 'YLabel'), 'Position', [ 0.75 MaxMax * 0.15 ] );

xlabel( 'Milliseconds' );
ylabel( 'Frequency' );
```

```
% tighten up borders
tightfig();

% move y label to middle of graphs
ylabel( 'Frequency', 'Position', [ ( -size / 18 ) ( ( halfWinSize * 6.1666 ) /
2 ) ] );

% write plot to file
hgexport( fig, [ fileName, '.err.spect.eps' ] );

% EOF
```

### 8.4.12 – Error Spectrogram Comparison Detail

```
function dif_error_detail( fileName )
%-------------------------------------------------------------------------
% Error Spectrogram Detail Comparison
%
% Cooper Baker - 2014
%-------------------------------------------------------------------------

close all;

name = 'Error Spectrograms Detail';

% detail settings
headLoc = 0.25;
midLoc  = 0.5;
tailLoc = 0.75;
detail  = 0.01;

% set paths
p = genpath( '../../analyze' );
addpath( p );
p = genpath( '../../tones' );
addpath( p );

% get audio file info
fileInfo    = audioinfo( [ fileName, '_ideal.wav' ] );
sr          = fileInfo.SampleRate;
samps       = fileInfo.TotalSamples;
winSize     = 512;
halfWinSize = winSize / 2;
overlap     = 4;

% make file names
rawFile  = sprintf( '%s.classic.spect.csv', fileName );
lockFile = sprintf( '%s.lock.spect.csv'   , fileName );
peakFile = sprintf( '%s.peak.spect.csv'   , fileName );
bankFile = sprintf( '%s.bank.spect.csv'   , fileName );
```

```
solaFile = sprintf( '%s.sola.spect.csv'    , fileName );
olaFile  = sprintf( '%s.ola.spect.csv'     , fileName );

% load file data
rawData  = csvread( rawFile  );
lockData = csvread( lockFile );
peakData = csvread( peakFile );
bankData = csvread( bankFile );
solaData = csvread( solaFile );
olaData  = csvread( olaFile  );

% get data size
size = length( rawData );

% calculate msec grid values
msec      = ( samps / sr ) * 1000;
gridMsec  = 500;
gridHops  = gridMsec / ( ( ( winSize / overlap ) / sr ) * 1000 );
hopSize   = winSize / overlap;
hopMsec   = ( hopSize / sr ) * 1000;
% frames    = ( size / winSize ) * overlap - ( overlap - 1 );
frames = length( rawData );
rangeMsec = detail * msec;
headMsec  = ( headLoc - ( detail / 2 ) ) * msec;
midMsec   = ( midLoc  - ( detail / 2 ) ) * msec;
tailMsec  = ( tailLoc - ( detail / 2 ) ) * msec;
headMsec  = headMsec + rangeMsec * 0.25;
midMsec   = midMsec  + rangeMsec * 0.25;
tailMsec  = tailMsec + rangeMsec * 0.25;
% winSum    = sum( window );

% calculate data maximums
rawMax  = max( rawData  );
lockMax = max( lockData );
peakMax = max( peakData );
bankMax = max( bankData );
solaMax = max( solaData );
olaMax  = max( olaData  );

% find maximum maximum and minimum maximum
MaxMax = max( [ rawMax lockMax peakMax bankMax solaMax olaMax ] );
MinMax = min( [ rawMax lockMax peakMax bankMax solaMax olaMax ] );

% calculate cropping values
side = round( frames * detail );
head = round( frames * headLoc );
mid  = round( frames * midLoc );
tail = round( frames * tailLoc );
range = side * 2;

% copy detail sections of data
rawHead  = rawData ( :, ( head - side ) : ( head + side ) );
```

```
rawMid   = rawData ( :, ( mid  - side ) : ( mid  + side ) );
rawTail  = rawData ( :, ( tail - side ) : ( tail + side ) );
lockHead = lockData( :, ( head - side ) : ( head + side ) );
lockMid  = lockData( :, ( mid  - side ) : ( mid  + side ) );
lockTail = lockData( :, ( tail - side ) : ( tail + side ) );
peakHead = peakData( :, ( head - side ) : ( head + side ) );
peakMid  = peakData( :, ( mid  - side ) : ( mid  + side ) );
peakTail = peakData( :, ( tail - side ) : ( tail + side ) );
bankHead = bankData( :, ( head - side ) : ( head + side ) );
bankMid  = bankData( :, ( mid  - side ) : ( mid  + side ) );
bankTail = bankData( :, ( tail - side ) : ( tail + side ) );
solaHead = solaData( :, ( head - side ) : ( head + side ) );
solaMid  = solaData( :, ( mid  - side ) : ( mid  + side ) );
solaTail = solaData( :, ( tail - side ) : ( tail + side ) );
olaHead  = olaData ( :, ( head - side ) : ( head + side ) );
olaMid   = olaData ( :, ( mid  - side ) : ( mid  + side ) );
olaTail  = olaData ( :, ( tail - side ) : ( tail + side ) );

% graph formatting
fontName   = 'Times New Roman';
fontSize   = 12;
xTxt       = range * 0.0666;
yTxt       = halfWinSize * 0.9;

% color mapping
colormap( [ [ 1 : -0.2/64 : 0.8 ]; ( [ 1 : -0.5/64 : 0.5 ] .^ 10 ) ; [ 1 : -
0.5/64 : 0.5 ] .^ 20 ]' );

% set up plot window
fig = figure( 1 );
set( fig, 'Name', sprintf( '%s - %s.wav', name, fileName ) );
set( fig, 'Position', [ 0 0 800 1050 ] );
set( fig, 'defaultAxesFontName', fontName );
set( fig, 'defaultTextFontName', fontName );
set( fig, 'defaultlinelinewidth', 3 );
set( fig, 'defaultaxescolororder', [ 0.8 0.2 0.2 ] );

% classic
%--------------------------------------------------------------------------

% head
algo = 'Classic';
set( subplot( 6, 3, 1 ), 'Position', [ 0.1, 5.5/7, 0.26, 1/7.333 ] );
imagesc( rawHead );
axis xy;
caxis( [ 0 MaxMax ] );
set( gca, 'XTick', [ range * 0.25 : ( range / 4 ) : range * 0.75 ] );
set( gca, 'XTickLabel', [] );
set( gca, 'YTick', [ 1 : ( halfWinSize / ( sr / 4000 ) ) : halfWinSize ] );
set( gca, 'YTickLabel', [ 0 : 2000 : ( sr / 2 ) ] );
text( xTxt, yTxt, algo, 'EdgeColor', 'k', 'BackgroundColor', 'w' );
title( 'Head', 'FontSize', fontSize );
```

```
%middle
set( subplot( 6, 3, 2 ), 'Position', [ 0.3666, 5.5/7, 0.26, 1/7.333 ] );
imagesc( rawMid );
axis xy;
caxis( [ 0 MaxMax ] );
set( gca, 'XTick', [ range * 0.25 : ( range / 4 ) : range * 0.75 ] );
set( gca, 'XTickLabel', [] );
set( gca, 'YTick', [ 1 : ( halfWinSize / ( sr / 4000 ) ) : halfWinSize ] );
set( gca, 'YTickLabel', [] );
title( 'Middle', 'FontSize', fontSize );

%tail
set( subplot( 6, 3, 3 ), 'Position', [ 0.6333, 5.5/7, 0.26, 1/7.333 ] );
imagesc( rawTail );
axis xy;
caxis( [ 0 MaxMax ] );
set( gca, 'XTick', [ range * 0.25 : ( range / 4 ) : range * 0.75 ] );
set( gca, 'XTickLabel', [] );
set( gca, 'YTick', [ 1 : ( halfWinSize / ( sr / 4000 ) ) : halfWinSize ] );
set( gca, 'YTickLabel', [] );
bar = colorbar( 'location', 'East' );
set( get( bar, 'YLabel'), 'String', 'Sones' );
set( get( bar, 'YLabel'), 'Position', [ 0.75 MaxMax * 0.15 ] );
title( 'Tail', 'FontSize', fontSize );

% phase locked
%--------------------------------------------------------------------------

% head
algo = 'Lock';
set( subplot( 6, 3, 4 ), 'Position', [ 0.1, 4.5/7, 0.26, 1/7.333 ] );
imagesc( lockHead );
axis xy;
caxis( [ 0 MaxMax ] );
set( gca, 'XTick', [ range * 0.25 : ( range / 4 ) : range * 0.75 ] );
set( gca, 'XTickLabel', [] );
set( gca, 'YTick', [ 1 : ( halfWinSize / ( sr / 4000 ) ) : halfWinSize ] );
set( gca, 'YTickLabel', [ 0 : 2000 : ( sr / 2 ) ] );
text( xTxt, yTxt, algo, 'EdgeColor', 'k', 'BackgroundColor', 'w' );

%middle
set( subplot( 6, 3, 5 ), 'Position', [ 0.3666, 4.5/7, 0.26, 1/7.333 ] );
imagesc( lockMid );
axis xy;
caxis( [ 0 MaxMax ] );
set( gca, 'XTick', [ range * 0.25 : ( range / 4 ) : range * 0.75 ] );
set( gca, 'XTickLabel', [] );
set( gca, 'YTick', [ 1 : ( halfWinSize / ( sr / 4000 ) ) : halfWinSize ] );
set( gca, 'YTickLabel', [] );
```

```
%tail
set( subplot( 6, 3, 6 ), 'Position', [ 0.6333, 4.5/7, 0.26, 1/7.333 ] );
imagesc( lockTail );
axis xy;
caxis( [ 0 MaxMax ] );
set( gca, 'XTick', [ range * 0.25 : ( range / 4 ) : range * 0.75 ] );
set( gca, 'XTickLabel', [] );
set( gca, 'YTick', [ 1 : ( halfWinSize / ( sr / 4000 ) ) : halfWinSize ] );
set( gca, 'YTickLabel', [] );
bar = colorbar( 'location', 'East' );
set( get( bar, 'YLabel'), 'String', 'Sones' );
set( get( bar, 'YLabel'), 'Position', [ 0.75 MaxMax * 0.15 ] );

% peak tracking
%--------------------------------------------------------------------------

% head
algo = 'Peak';
set( subplot( 6, 3, 7 ), 'Position', [ 0.1, 3.5/7, 0.26, 1/7.333 ] );
imagesc( peakHead );
axis xy;
caxis( [ 0 MaxMax ] );
set( gca, 'XTick', [ range * 0.25 : ( range / 4 ) : range * 0.75 ] );
set( gca, 'XTickLabel', [] );
set( gca, 'YTick', [ 1 : ( halfWinSize / ( sr / 4000 ) ) : halfWinSize ] );
set( gca, 'YTickLabel', [ 0 : 2000 : ( sr / 2 ) ] );
text( xTxt, yTxt, algo, 'EdgeColor', 'k', 'BackgroundColor', 'w'  );

%middle
set( subplot( 6, 3, 8 ), 'Position', [ 0.3666, 3.5/7, 0.26, 1/7.333 ] );
imagesc( peakMid );
axis xy;
caxis( [ 0 MaxMax ] );
set( gca, 'XTick', [ range * 0.25 : ( range / 4 ) : range * 0.75 ] );
set( gca, 'XTickLabel', [] );
set( gca, 'YTick', [ 1 : ( halfWinSize / ( sr / 4000 ) ) : halfWinSize ] );
set( gca, 'YTickLabel', [] );

%tail
set( subplot( 6, 3, 9 ), 'Position', [ 0.6333, 3.5/7, 0.26, 1/7.333 ] );
imagesc( peakTail );
axis xy;
caxis( [ 0 MaxMax ] );
set( gca, 'XTick', [ range * 0.25 : ( range / 4 ) : range * 0.75 ] );
set( gca, 'XTickLabel', [] );
set( gca, 'YTick', [ 1 : ( halfWinSize / ( sr / 4000 ) ) : halfWinSize ] );
set( gca, 'YTickLabel', [] );
bar = colorbar( 'location', 'East' );
set( get( bar, 'YLabel'), 'String', 'Sones' );
set( get( bar, 'YLabel'), 'Position', [ 0.75 MaxMax * 0.15 ] );
```

```
% oscillator bank
%-------------------------------------------------------------------------

% head
algo = 'Bank';
set( subplot( 6, 3, 10 ), 'Position', [ 0.1, 2.5/7, 0.26, 1/7.333 ] );
imagesc( bankHead );
axis xy;
caxis( [ 0 MaxMax ] );
set( gca, 'XTick', [ range * 0.25 : ( range / 4 ) : range * 0.75 ] );
set( gca, 'XTickLabel', [] );
set( gca, 'YTick', [ 1 : ( halfWinSize / ( sr / 4000 ) ) : halfWinSize ] );
set( gca, 'YTickLabel', [ 0 : 2000 : ( sr / 2 ) ] );
text( xTxt, yTxt, algo, 'EdgeColor', 'k', 'BackgroundColor', 'w'  );

%middle
set( subplot( 6, 3, 11 ), 'Position', [ 0.3666, 2.5/7, 0.26, 1/7.333 ] );
imagesc( bankMid );
axis xy;
caxis( [ 0 MaxMax ] );
set( gca, 'XTick', [ range * 0.25 : ( range / 4 ) : range * 0.75 ] );
set( gca, 'XTickLabel', [] );
set( gca, 'YTick', [ 1 : ( halfWinSize / ( sr / 4000 ) ) : halfWinSize ] );
set( gca, 'YTickLabel', [] );

%tail
set( subplot( 6, 3, 12 ), 'Position', [ 0.6333, 2.5/7, 0.26, 1/7.333 ] );
imagesc( bankTail );
axis xy;
caxis( [ 0 MaxMax ] );
set( gca, 'XTick', [ range * 0.25 : ( range / 4 ) : range * 0.75 ] );
set( gca, 'XTickLabel', [] );
set( gca, 'YTick', [ 1 : ( halfWinSize / ( sr / 4000 ) ) : halfWinSize ] );
set( gca, 'YTickLabel', [] );
bar = colorbar( 'location', 'East' );
set( get( bar, 'YLabel'), 'String', 'Sones' );
set( get( bar, 'YLabel'), 'Position', [ 0.75 MaxMax * 0.15 ] );

% sola
%-------------------------------------------------------------------------

% head
algo = 'Sola';
set( subplot( 6, 3, 13 ), 'Position', [ 0.1, 1.5/7, 0.26, 1/7.333 ] );
imagesc( solaHead );
axis xy;
caxis( [ 0 MaxMax ] );
set( gca, 'XTick', [ range * 0.25 : ( range / 4 ) : range * 0.75 ] );
set( gca, 'XTickLabel', [] );
set( gca, 'YTick', [ 1 : ( halfWinSize / ( sr / 4000 ) ) : halfWinSize ] );
set( gca, 'YTickLabel', [ 0 : 2000 : ( sr / 2 ) ] );
text( xTxt, yTxt, algo, 'EdgeColor', 'k', 'BackgroundColor', 'w'  );
```

```
%middle
set( subplot( 6, 3, 14 ), 'Position', [ 0.3666, 1.5/7, 0.26, 1/7.333 ] );
imagesc( solaMid );
axis xy;
caxis( [ 0 MaxMax ] );
set( gca, 'XTick', [ range * 0.25 : ( range / 4 ) : range * 0.75 ] );
set( gca, 'XTickLabel', [] );
set( gca, 'YTick', [ 1 : ( halfWinSize / ( sr / 4000 ) ) : halfWinSize ] );
set( gca, 'YTickLabel', [] );

%tail
set( subplot( 6, 3, 15 ), 'Position', [ 0.6333, 1.5/7, 0.26, 1/7.333 ] );
imagesc( solaTail );
axis xy;
caxis( [ 0 MaxMax ] );
set( gca, 'XTick', [ range * 0.25 : ( range / 4 ) : range * 0.75 ] );
set( gca, 'XTickLabel', [] );
set( gca, 'YTick', [ 1 : ( halfWinSize / ( sr / 4000 ) ) : halfWinSize ] );
set( gca, 'YTickLabel', [] );
bar = colorbar( 'location', 'East' );
set( get( bar, 'YLabel'), 'String', 'Sones' );
set( get( bar, 'YLabel'), 'Position', [ 0.75 MaxMax * 0.15 ] );

% ola
%-------------------------------------------------------------------------

% head
algo = 'Ola';
set( subplot( 6, 3, 16 ), 'Position', [ 0.1, 0.5/7, 0.26, 1/7.333 ] );
imagesc( olaHead );
axis xy;
caxis( [ 0 MaxMax ] );
set( gca, 'XTick', [ range * 0.25 : ( range / 4 ) : range * 0.75 ] );
set( gca, 'XTickLabel', [ headMsec : rangeMsec / 4 : headMsec + rangeMsec ] );
set( gca, 'YTick', [ 1 : ( halfWinSize / ( sr / 4000 ) ) : halfWinSize ] );
set( gca, 'YTickLabel', [ 0 : 2000 : ( sr / 2 ) ] );
text( xTxt, yTxt, algo, 'EdgeColor', 'k', 'BackgroundColor', 'w'  );
yLab = ylabel( 'Frequency' );

%middle
set( subplot( 6, 3, 17 ), 'Position', [ 0.3666, 0.5/7, 0.26, 1/7.333 ] );
imagesc( olaMid );
axis xy;
caxis( [ 0 MaxMax ] );
set( gca, 'XTick', [ range * 0.25 : ( range / 4 ) : range * 0.75 ] );
set( gca, 'XTickLabel', [ midMsec : rangeMsec / 4 : midMsec + rangeMsec ] );
set( gca, 'YTick', [ 1 : ( halfWinSize / ( sr / 4000 ) ) : halfWinSize ] );
set( gca, 'YTickLabel', [] );
xlabel( 'Milliseconds' );
```

```
%tail
set( subplot( 6, 3, 18 ), 'Position', [ 0.6333, 0.5/7, 0.26, 1/7.333 ] );
imagesc( olaTail );
axis xy;
caxis( [ 0 MaxMax ] );
set( gca, 'XTick', [ range * 0.25 : ( range / 4 ) : range * 0.75 ] );
set( gca, 'XTickLabel', [ tailMsec : rangeMsec / 4 : tailMsec + rangeMsec ] );
set( gca, 'YTick', [ 1 : ( halfWinSize / ( sr / 4000 ) ) : halfWinSize ] );
set( gca, 'YTickLabel', [] );
bar = colorbar( 'location', 'East' );
set( get( bar, 'YLabel'), 'String', 'Sones' );
set( get( bar, 'YLabel'), 'Position', [ 0.75 MaxMax * 0.15 ] );

% tighten up borders
tightfig();

% move y label to middle of graphs
set( yLab, 'Position', [ (-range / 6 ) ( ( halfWinSize * 6.25 ) / 2 ) ] );

% write plot to file
hgexport( fig, [ fileName, '.err.spect.zoom.eps' ] );

% EOF
```

### 8.4.13 – Average Spectrum Error Summary

```
function avg_summary()
%-------------------------------------------------------------------------
% Average Spectrum Error Summary
%
% Cooper Baker - 2014
%-------------------------------------------------------------------------

close all;

name = 'Average Spectrum Summary';

% set path
p = genpath( '../../analyze' );
addpath( p );

% algorithm names for plot labels
algos = { 'Classic', 'Lock', 'Peak', 'Bank', 'Sola', 'Ola' };

% file names for plot labels
fileAname = 'Sine';
fileBname = 'Square';
fileCname = 'Sweep';
fileDname = 'Amen';
fileEname = 'Autumn';
fileFname = 'Peaches';
```

```
% file names for csvread
fileA = 'sine';
fileB = 'square';
fileC = 'sweep';
fileD = 'amen';
fileE = 'autumn';
fileF = 'peaches';

% read the error plot data
rawAdata = csvread( [ fileA, '.classic.avg.err.csv' ] );
rawBdata = csvread( [ fileB, '.classic.avg.err.csv' ] );
rawCdata = csvread( [ fileC, '.classic.avg.err.csv' ] );
rawDdata = csvread( [ fileD, '.classic.avg.err.csv' ] );
rawEdata = csvread( [ fileE, '.classic.avg.err.csv' ] );
rawFdata = csvread( [ fileF, '.classic.avg.err.csv' ] );

lockAdata = csvread( [ fileA, '.lock.avg.err.csv' ] );
lockBdata = csvread( [ fileB, '.lock.avg.err.csv' ] );
lockCdata = csvread( [ fileC, '.lock.avg.err.csv' ] );
lockDdata = csvread( [ fileD, '.lock.avg.err.csv' ] );
lockEdata = csvread( [ fileE, '.lock.avg.err.csv' ] );
lockFdata = csvread( [ fileF, '.lock.avg.err.csv' ] );

peakAdata = csvread( [ fileA, '.peak.avg.err.csv' ] );
peakBdata = csvread( [ fileB, '.peak.avg.err.csv' ] );
peakCdata = csvread( [ fileC, '.peak.avg.err.csv' ] );
peakDdata = csvread( [ fileD, '.peak.avg.err.csv' ] );
peakEdata = csvread( [ fileE, '.peak.avg.err.csv' ] );
peakFdata = csvread( [ fileF, '.peak.avg.err.csv' ] );

bankAdata = csvread( [ fileA, '.bank.avg.err.csv' ] );
bankBdata = csvread( [ fileB, '.bank.avg.err.csv' ] );
bankCdata = csvread( [ fileC, '.bank.avg.err.csv' ] );
bankDdata = csvread( [ fileD, '.bank.avg.err.csv' ] );
bankEdata = csvread( [ fileE, '.bank.avg.err.csv' ] );
bankFdata = csvread( [ fileF, '.bank.avg.err.csv' ] );

solaAdata = csvread( [ fileA, '.sola.avg.err.csv' ] );
solaBdata = csvread( [ fileB, '.sola.avg.err.csv' ] );
solaCdata = csvread( [ fileC, '.sola.avg.err.csv' ] );
solaDdata = csvread( [ fileD, '.sola.avg.err.csv' ] );
solaEdata = csvread( [ fileE, '.sola.avg.err.csv' ] );
solaFdata = csvread( [ fileF, '.sola.avg.err.csv' ] );

olaAdata = csvread( [ fileA, '.ola.avg.err.csv' ] );
olaBdata = csvread( [ fileB, '.ola.avg.err.csv' ] );
olaCdata = csvread( [ fileC, '.ola.avg.err.csv' ] );
olaDdata = csvread( [ fileD, '.ola.avg.err.csv' ] );
olaEdata = csvread( [ fileE, '.ola.avg.err.csv' ] );
olaFdata = csvread( [ fileF, '.ola.avg.err.csv' ] );

% find max values
```

```
rawAmax = max( rawAdata );
rawBmax = max( rawBdata );
rawCmax = max( rawCdata );
rawDmax = max( rawDdata );
rawEmax = max( rawEdata );
rawFmax = max( rawFdata );

lockAmax = max( lockAdata );
lockBmax = max( lockBdata );
lockCmax = max( lockCdata );
lockDmax = max( lockDdata );
lockEmax = max( lockEdata );
lockFmax = max( lockFdata );

lockAavg = sum( lockAdata ) / length( lockAdata );
lockBavg = sum( lockBdata ) / length( lockBdata );
lockCavg = sum( lockCdata ) / length( lockCdata );
lockDavg = sum( lockDdata ) / length( lockDdata );
lockEavg = sum( lockEdata ) / length( lockEdata );
lockFavg = sum( lockFdata ) / length( lockFdata );

peakAmax = max( peakAdata );
peakBmax = max( peakBdata );
peakCmax = max( peakCdata );
peakDmax = max( peakDdata );
peakEmax = max( peakEdata );
peakFmax = max( peakFdata );

bankAmax = max( bankAdata );
bankBmax = max( bankBdata );
bankCmax = max( bankCdata );
bankDmax = max( bankDdata );
bankEmax = max( bankEdata );
bankFmax = max( bankFdata );

solaAmax = max( solaAdata );
solaBmax = max( solaBdata );
solaCmax = max( solaCdata );
solaDmax = max( solaDdata );
solaEmax = max( solaEdata );
solaFmax = max( solaFdata );

olaAmax = max( olaAdata );
olaBmax = max( olaBdata );
olaCmax = max( olaCdata );
olaDmax = max( olaDdata );
olaEmax = max( olaEdata );
olaFmax = max( olaFdata );

% calculate averages
rawAavg = sum( rawAdata ) / length( rawAdata );
rawBavg = sum( rawBdata ) / length( rawBdata );
```

```
rawCavg = sum( rawCdata ) / length( rawCdata );
rawDavg = sum( rawDdata ) / length( rawDdata );
rawEavg = sum( rawEdata ) / length( rawEdata );
rawFavg = sum( rawFdata ) / length( rawFdata );

peakAavg = sum( peakAdata ) / length( peakAdata );
peakBavg = sum( peakBdata ) / length( peakBdata );
peakCavg = sum( peakCdata ) / length( peakCdata );
peakDavg = sum( peakDdata ) / length( peakDdata );
peakEavg = sum( peakEdata ) / length( peakEdata );
peakFavg = sum( peakFdata ) / length( peakFdata );

bankAavg = sum( bankAdata ) / length( bankAdata );
bankBavg = sum( bankBdata ) / length( bankBdata );
bankCavg = sum( bankCdata ) / length( bankCdata );
bankDavg = sum( bankDdata ) / length( bankDdata );
bankEavg = sum( bankEdata ) / length( bankEdata );
bankFavg = sum( bankFdata ) / length( bankFdata );

solaAavg = sum( solaAdata ) / length( solaAdata );
solaBavg = sum( solaBdata ) / length( solaBdata );
solaCavg = sum( solaCdata ) / length( solaCdata );
solaDavg = sum( solaDdata ) / length( solaDdata );
solaEavg = sum( solaEdata ) / length( solaEdata );
solaFavg = sum( solaFdata ) / length( solaFdata );

olaAavg = sum( olaAdata ) / length( olaAdata );
olaBavg = sum( olaBdata ) / length( olaBdata );
olaCavg = sum( olaCdata ) / length( olaCdata );
olaDavg = sum( olaDdata ) / length( olaDdata );
olaEavg = sum( olaEdata ) / length( olaEdata );
olaFavg = sum( olaFdata ) / length( olaFdata );

% find max of each audio file
maxAmax = max( [ rawAmax lockAmax peakAmax bankAmax solaAmax olaAmax ] );
maxBmax = max( [ rawBmax lockBmax peakBmax bankBmax solaBmax olaBmax ] );
maxCmax = max( [ rawCmax lockCmax peakCmax bankCmax solaCmax olaCmax ] );
maxDmax = max( [ rawDmax lockDmax peakDmax bankDmax solaDmax olaDmax ] );
maxEmax = max( [ rawEmax lockEmax peakEmax bankEmax solaEmax olaEmax ] );
maxFmax = max( [ rawFmax lockFmax peakFmax bankFmax solaFmax olaFmax ] );

% create matricies of max values for plots
maxA = [ rawAmax lockAmax peakAmax bankAmax solaAmax olaAmax ];
maxB = [ rawBmax lockBmax peakBmax bankBmax solaBmax olaBmax ];
maxC = [ rawCmax lockCmax peakCmax bankCmax solaCmax olaCmax ];
maxD = [ rawDmax lockDmax peakDmax bankDmax solaDmax olaDmax ];
maxE = [ rawEmax lockEmax peakEmax bankEmax solaEmax olaEmax ];
maxF = [ rawFmax lockFmax peakFmax bankFmax solaFmax olaFmax ];

% create matrices of avg values for plots
avgA = [ rawAavg lockAavg peakAavg bankAavg solaAavg olaAavg ];
avgB = [ rawBavg lockBavg peakBavg bankBavg solaBavg olaBavg ];
```

```
avgC = [ rawCavg lockCavg peakCavg bankCavg solaCavg olaCavg ];
avgD = [ rawDavg lockDavg peakDavg bankDavg solaDavg olaDavg ];
avgE = [ rawEavg lockEavg peakEavg bankEavg solaEavg olaEavg ];
avgF = [ rawFavg lockFavg peakFavg bankFavg solaFavg olaFavg ];

% Plot
%-------------------------------------------------------------------------

% graph formatting
fontName   = 'Times New Roman';
fontSize   = 12;
xTxt = 0.125;
yTxt = 1.2;
yOff = 0.13;

% set up plot window
fig = figure( 1 );
set( fig, 'Name', sprintf( '%s', name ) );
set( fig, 'Position', [ 0 0 800 1000 ] );
set( fig, 'defaultAxesFontName', fontName );
set( fig, 'defaultTextFontName', fontName );
set( fig, 'defaultaxescolororder', [ 0.8 0.2 0.2 ] );

% fileA
%-------------------------------------------------------------------------
plotL = avgA;
plotR = maxA;
fileName = fileAname;
sp = subplot( 6, 2, 1 );
bar( plotL, 'FaceColor', [ 0.8 0.2 0.2 ] );
set( sp, 'Position', [ 0.1, 5.5/7, 0.39, 1/7.333 ] );
axis( [ 0 7 0 ( maxAmax * 1.3 ) ] );
for( i = 1 : 6 )
    text( i, plotL( i ) + maxAmax * yOff, num2str( plotL( i ), '%2.5f' ),
'HorizontalAlignment', 'center', 'BackgroundColor', 'w', 'EdgeColor', 'k',
'FontSize', 9 );
end
set( gca, 'TickLength', [ 0 0 ] );
set( gca, 'YGrid', 'on', 'XGrid', 'off' );
set( gca, 'XTickLabel', [] );
temp = get( gca );
yticks = get( gca, 'YTick' );
set( gca, 'YTickLabel', yticks );
text( xTxt, yTxt * maxAmax, [ ' ' fileName ' ' ], 'BackgroundColor', 'w',
'EdgeColor', 'k' );
title( 'Average', 'FontSize', fontSize );

sp = subplot( 6, 2, 2 );
bar( plotR, 'FaceColor', [ 0.8 0.2 0.2 ] );
set( sp, 'Position', [ 0.5, 5.5/7, 0.39, 1/7.333 ] );
axis( [ 0 7 0 ( maxAmax * 1.3 ) ] );
for( i = 1 : 6 )
```

```
    text( i, plotR( i ) + maxAmax * yOff, num2str( plotR( i ), '%2.5f' ),
'HorizontalAlignment', 'center', 'BackgroundColor', 'w', 'EdgeColor', 'k',
'FontSize', 9 );
end
set( gca, 'TickLength', [ 0 0 ] );
set( gca, 'YGrid', 'on', 'XGrid', 'off' );
set( gca, 'XTickLabel', [] );
set( gca, 'YTickLabel', [] );
title( 'Maximum', 'FontSize', fontSize );

% fileB
%-------------------------------------------------------------------------
plotL = avgB;
plotR = maxB;
fileName = fileBname;
sp = subplot( 6, 2, 3 );
bar( plotL, 'FaceColor', [ 0.8 0.2 0.2 ] );
set( sp, 'Position', [ 0.1, 4.5/7, 0.39, 1/7.333 ] );
axis( [ 0 7 0 ( maxBmax * 1.3 ) ] );
for( i = 1 : 6 )
    text( i, plotL( i ) + maxBmax * yOff, num2str( plotL( i ), '%2.5f' ),
'HorizontalAlignment', 'center', 'BackgroundColor', 'w', 'EdgeColor', 'k',
'FontSize', 9 );
end
set( gca, 'TickLength', [ 0 0 ] );
set( gca, 'YGrid', 'on', 'XGrid', 'off' );
set( gca, 'XTickLabel', [] );
temp = get( gca );
yticks = get( gca, 'YTick' );
set( gca, 'YTickLabel', yticks );
text( xTxt, yTxt * maxBmax, [ ' ' fileName ' ' ], 'BackgroundColor', 'w',
'EdgeColor', 'k' );

sp = subplot( 6, 2, 4 );
bar( plotR, 'FaceColor', [ 0.8 0.2 0.2 ] );
set( sp, 'Position', [ 0.5, 4.5/7, 0.39, 1/7.333 ] );
axis( [ 0 7 0 ( maxBmax * 1.3 ) ] );
for( i = 1 : 6 )
    text( i, plotR( i ) + maxBmax * yOff, num2str( plotR( i ), '%2.5f' ),
'HorizontalAlignment', 'center', 'BackgroundColor', 'w', 'EdgeColor', 'k',
'FontSize', 9 );
end
set( gca, 'TickLength', [ 0 0 ] );
set( gca, 'YGrid', 'on', 'XGrid', 'off' );
set( gca, 'XTickLabel', [] );
set( gca, 'YTickLabel', [] );

% fileC
%-------------------------------------------------------------------------
plotL = avgC;
plotR = maxC;
fileName = fileCname;
```

```
sp = subplot( 6, 2, 5 );
bar( plotL, 'FaceColor', [ 0.8 0.2 0.2 ] );
set( sp, 'Position', [ 0.1, 3.5/7, 0.39, 1/7.333 ] );
axis( [ 0 7 0 ( maxCmax * 1.3 ) ] );
for( i = 1 : 6 )
    text( i, plotL( i ) + maxCmax * yOff, num2str( plotL( i ), '%2.5f' ),
'HorizontalAlignment', 'center', 'BackgroundColor', 'w', 'EdgeColor', 'k',
'FontSize', 9 );
end
set( gca, 'TickLength', [ 0 0 ] );
set( gca, 'YGrid', 'on', 'XGrid', 'off' );
set( gca, 'XTickLabel', [] );
temp = get( gca );
yticks = get( gca, 'YTick' );
set( gca, 'YTickLabel', yticks );
text( xTxt, yTxt * maxCmax, [ ' ' fileName ' ' ], 'BackgroundColor', 'w',
'EdgeColor', 'k' );

sp = subplot( 6, 2, 6 );
bar( plotR, 'FaceColor', [ 0.8 0.2 0.2 ] );
set( sp, 'Position', [ 0.5, 3.5/7, 0.39, 1/7.333 ] );
axis( [ 0 7 0 ( maxCmax * 1.3 ) ] );
for( i = 1 : 6 )
    text( i, plotR( i ) + maxCmax * yOff, num2str( plotR( i ), '%2.5f' ),
'HorizontalAlignment', 'center', 'BackgroundColor', 'w', 'EdgeColor', 'k',
'FontSize', 9 );
end
set( gca, 'TickLength', [ 0 0 ] );
set( gca, 'YGrid', 'on', 'XGrid', 'off' );
set( gca, 'XTickLabel', [] );
set( gca, 'YTickLabel', [] );

% fileD
%-------------------------------------------------------------------------
plotL = avgD;
plotR = maxD;
fileName = fileDname;
sp = subplot( 6, 2, 7 );
bar( plotL, 'FaceColor', [ 0.8 0.2 0.2 ] );
set( sp, 'Position', [ 0.1, 2.5/7, 0.39, 1/7.333 ] );
axis( [ 0 7 0 ( maxDmax * 1.3 ) ] );
for( i = 1 : 6 )
    text( i, plotL( i ) + maxDmax * yOff, num2str( plotL( i ), '%2.5f' ),
'HorizontalAlignment', 'center', 'BackgroundColor', 'w', 'EdgeColor', 'k',
'FontSize', 9 );
end
set( gca, 'TickLength', [ 0 0 ] );
set( gca, 'YGrid', 'on', 'XGrid', 'off' );
set( gca, 'XTickLabel', [] );
temp = get( gca );
yticks = get( gca, 'YTick' );
set( gca, 'YTickLabel', yticks );
```

```
text( xTxt, yTxt * maxDmax, [ ' ' fileName ' ' ], 'BackgroundColor', 'w',
'EdgeColor', 'k' );

sp = subplot( 6, 2, 8 );
bar( plotR, 'FaceColor', [ 0.8 0.2 0.2 ] );
set( sp, 'Position', [ 0.5, 2.5/7, 0.39, 1/7.333 ] );
axis( [ 0 7 0 ( maxDmax * 1.3 ) ] );
for( i = 1 : 6 )
    text( i, plotR( i ) + maxDmax * yOff, num2str( plotR( i ), '%2.5f' ),
'HorizontalAlignment', 'center', 'BackgroundColor', 'w', 'EdgeColor', 'k',
'FontSize', 9 );
end
set( gca, 'TickLength', [ 0 0 ] );
set( gca, 'YGrid', 'on', 'XGrid', 'off' );
set( gca, 'XTickLabel', [] );
set( gca, 'YTickLabel', [] );

% fileE
%--------------------------------------------------------------------------
plotL = avgE;
plotR = maxE;
fileName = fileEname;
sp = subplot( 6, 2, 9 );
bar( plotL, 'FaceColor', [ 0.8 0.2 0.2 ] );
set( sp, 'Position', [ 0.1, 1.5/7, 0.39, 1/7.333 ] );
axis( [ 0 7 0 ( maxEmax * 1.3 ) ] );
for( i = 1 : 6 )
    text( i, plotL( i ) + maxEmax * yOff, num2str( plotL( i ), '%2.5f' ),
'HorizontalAlignment', 'center', 'BackgroundColor', 'w', 'EdgeColor', 'k',
'FontSize', 9 );
end
set( gca, 'TickLength', [ 0 0 ] );
set( gca, 'YGrid', 'on', 'XGrid', 'off' );
set( gca, 'XTickLabel', [] );
temp = get( gca );
yticks = get( gca, 'YTick' );
set( gca, 'YTickLabel', yticks );
text( xTxt, yTxt * maxEmax, [ ' ' fileName ' ' ], 'BackgroundColor', 'w',
'EdgeColor', 'k' );

sp = subplot( 6, 2, 10 );
bar( plotR, 'FaceColor', [ 0.8 0.2 0.2 ] );
set( sp, 'Position', [ 0.5, 1.5/7, 0.39, 1/7.333 ] );
axis( [ 0 7 0 ( maxEmax * 1.3 ) ] );
for( i = 1 : 6 )
    text( i, plotR( i ) + maxEmax * yOff, num2str( plotR( i ), '%2.5f' ),
'HorizontalAlignment', 'center', 'BackgroundColor', 'w', 'EdgeColor', 'k',
'FontSize', 9 );
end
set( gca, 'TickLength', [ 0 0 ] );
set( gca, 'YGrid', 'on', 'XGrid', 'off' );
set( gca, 'XTickLabel', [] );
```

```
set( gca, 'YTickLabel', [] );

% fileF
%-------------------------------------------------------------------------
plotL = avgF;
plotR = maxF;
fileName = fileFname;
sp = subplot( 6, 2, 11 );
bar( plotL, 'FaceColor', [ 0.8 0.2 0.2 ] );
set( sp, 'Position', [ 0.1, 0.5/7, 0.39, 1/7.333 ] );
axis( [ 0 7 0 ( maxFmax * 1.3 ) ] );
for( i = 1 : 6 )
    text( i, plotL( i ) + maxFmax * yOff, num2str( plotL( i ), '%2.5f' ),
'HorizontalAlignment', 'center', 'BackgroundColor', 'w', 'EdgeColor', 'k',
'FontSize', 9 );
end
set( gca, 'TickLength', [ 0 0 ] );
set( gca, 'YGrid', 'on', 'XGrid', 'off' );
set( gca, 'XTickLabel', algos );
temp = get( gca );
yticks = get( gca, 'YTick' );
set( gca, 'YTickLabel', yticks );
yLab = ylabel( 'Sones' );
text( xTxt, yTxt * maxFmax, [ ' ' fileName ' ' ], 'BackgroundColor', 'w',
'EdgeColor', 'k' );
sp = subplot( 6, 2, 12 );
bar( plotR, 'FaceColor', [ 0.8 0.2 0.2 ] );
set( sp, 'Position', [ 0.5, 0.5/7, 0.39, 1/7.333 ] );
axis( [ 0 7 0 ( maxFmax * 1.3 ) ] );
for( i = 1 : 6 )
    text( i, plotR( i ) + maxFmax * yOff, num2str( plotR( i ), '%2.5f' ),
'HorizontalAlignment', 'center', 'BackgroundColor', 'w', 'EdgeColor', 'k',
'FontSize', 9 );
end
set( gca, 'TickLength', [ 0 0 ] );
set( gca, 'XTickLabel', algos );
set( gca, 'YTickLabel', [] );
set( gca, 'YGrid', 'on', 'XGrid', 'off' );
xLab = xlabel( 'Algorithms' );

% tighten up figure borders
tightfig();

% move x and y labels
set( xLab, 'Position', [ -0.1 ( maxFmax * -0.16 ) ] );
set( yLab, 'Position', [ -0.625 ( ( maxFmax * 1.3 * 6.25 ) / 2 ) ] );

% write plot to file
hgexport( fig, [ 'summary.avg.eps' ] );

% EOF
```

### 8.4.14 – Moving Spectral Average Error Summary

```
function mov_summary()
%-------------------------------------------------------------------------
% Moving Spectral Average Summary Of Average And Maximum Error
%
% Cooper Baker - 2014
%-------------------------------------------------------------------------

close all;

name = 'Moving Spectral Average Summary';

% set path
p = genpath( '../../analyze' );
addpath( p );

% algorithm names for plot labels
algos = { 'Classic', 'Lock', 'Peak', 'Bank', 'Sola', 'Ola' };

% file names for plot labels
fileAname = 'Sine';
fileBname = 'Square';
fileCname = 'Sweep';
fileDname = 'Amen';
fileEname = 'Autumn';
fileFname = 'Peaches';

% file names for csvread
fileA = 'sine';
fileB = 'square';
fileC = 'sweep';
fileD = 'amen';
fileE = 'autumn';
fileF = 'peaches';

% read the error plot data
rawAdata = csvread( [ fileA, '.classic.mov.err.csv' ] );
rawBdata = csvread( [ fileB, '.classic.mov.err.csv' ] );
rawCdata = csvread( [ fileC, '.classic.mov.err.csv' ] );
rawDdata = csvread( [ fileD, '.classic.mov.err.csv' ] );
rawEdata = csvread( [ fileE, '.classic.mov.err.csv' ] );
rawFdata = csvread( [ fileF, '.classic.mov.err.csv' ] );

lockAdata = csvread( [ fileA, '.lock.mov.err.csv' ] );
lockBdata = csvread( [ fileB, '.lock.mov.err.csv' ] );
lockCdata = csvread( [ fileC, '.lock.mov.err.csv' ] );
lockDdata = csvread( [ fileD, '.lock.mov.err.csv' ] );
lockEdata = csvread( [ fileE, '.lock.mov.err.csv' ] );
lockFdata = csvread( [ fileF, '.lock.mov.err.csv' ] );

peakAdata = csvread( [ fileA, '.peak.mov.err.csv' ] );
```

```
peakBdata = csvread( [ fileB, '.peak.mov.err.csv' ] );
peakCdata = csvread( [ fileC, '.peak.mov.err.csv' ] );
peakDdata = csvread( [ fileD, '.peak.mov.err.csv' ] );
peakEdata = csvread( [ fileE, '.peak.mov.err.csv' ] );
peakFdata = csvread( [ fileF, '.peak.mov.err.csv' ] );

bankAdata = csvread( [ fileA, '.bank.mov.err.csv' ] );
bankBdata = csvread( [ fileB, '.bank.mov.err.csv' ] );
bankCdata = csvread( [ fileC, '.bank.mov.err.csv' ] );
bankDdata = csvread( [ fileD, '.bank.mov.err.csv' ] );
bankEdata = csvread( [ fileE, '.bank.mov.err.csv' ] );
bankFdata = csvread( [ fileF, '.bank.mov.err.csv' ] );

solaAdata = csvread( [ fileA, '.sola.mov.err.csv' ] );
solaBdata = csvread( [ fileB, '.sola.mov.err.csv' ] );
solaCdata = csvread( [ fileC, '.sola.mov.err.csv' ] );
solaDdata = csvread( [ fileD, '.sola.mov.err.csv' ] );
solaEdata = csvread( [ fileE, '.sola.mov.err.csv' ] );
solaFdata = csvread( [ fileF, '.sola.mov.err.csv' ] );

olaAdata = csvread( [ fileA, '.ola.mov.err.csv' ] );
olaBdata = csvread( [ fileB, '.ola.mov.err.csv' ] );
olaCdata = csvread( [ fileC, '.ola.mov.err.csv' ] );
olaDdata = csvread( [ fileD, '.ola.mov.err.csv' ] );
olaEdata = csvread( [ fileE, '.ola.mov.err.csv' ] );
olaFdata = csvread( [ fileF, '.ola.mov.err.csv' ] );

% find max values
rawAmax = max( rawAdata );
rawBmax = max( rawBdata );
rawCmax = max( rawCdata );
rawDmax = max( rawDdata );
rawEmax = max( rawEdata );
rawFmax = max( rawFdata );

lockAmax = max( lockAdata );
lockBmax = max( lockBdata );
lockCmax = max( lockCdata );
lockDmax = max( lockDdata );
lockEmax = max( lockEdata );
lockFmax = max( lockFdata );

lockAavg = sum( lockAdata ) / length( lockAdata );
lockBavg = sum( lockBdata ) / length( lockBdata );
lockCavg = sum( lockCdata ) / length( lockCdata );
lockDavg = sum( lockDdata ) / length( lockDdata );
lockEavg = sum( lockEdata ) / length( lockEdata );
lockFavg = sum( lockFdata ) / length( lockFdata );

peakAmax = max( peakAdata );
peakBmax = max( peakBdata );
peakCmax = max( peakCdata );
```

```
peakDmax = max( peakDdata );
peakEmax = max( peakEdata );
peakFmax = max( peakFdata );

bankAmax = max( bankAdata );
bankBmax = max( bankBdata );
bankCmax = max( bankCdata );
bankDmax = max( bankDdata );
bankEmax = max( bankEdata );
bankFmax = max( bankFdata );

solaAmax = max( solaAdata );
solaBmax = max( solaBdata );
solaCmax = max( solaCdata );
solaDmax = max( solaDdata );
solaEmax = max( solaEdata );
solaFmax = max( solaFdata );

olaAmax = max( olaAdata );
olaBmax = max( olaBdata );
olaCmax = max( olaCdata );
olaDmax = max( olaDdata );
olaEmax = max( olaEdata );
olaFmax = max( olaFdata );

% calculate averages
rawAavg = sum( rawAdata ) / length( rawAdata );
rawBavg = sum( rawBdata ) / length( rawBdata );
rawCavg = sum( rawCdata ) / length( rawCdata );
rawDavg = sum( rawDdata ) / length( rawDdata );
rawEavg = sum( rawEdata ) / length( rawEdata );
rawFavg = sum( rawFdata ) / length( rawFdata );

peakAavg = sum( peakAdata ) / length( peakAdata );
peakBavg = sum( peakBdata ) / length( peakBdata );
peakCavg = sum( peakCdata ) / length( peakCdata );
peakDavg = sum( peakDdata ) / length( peakDdata );
peakEavg = sum( peakEdata ) / length( peakEdata );
peakFavg = sum( peakFdata ) / length( peakFdata );

bankAavg = sum( bankAdata ) / length( bankAdata );
bankBavg = sum( bankBdata ) / length( bankBdata );
bankCavg = sum( bankCdata ) / length( bankCdata );
bankDavg = sum( bankDdata ) / length( bankDdata );
bankEavg = sum( bankEdata ) / length( bankEdata );
bankFavg = sum( bankFdata ) / length( bankFdata );

solaAavg = sum( solaAdata ) / length( solaAdata );
solaBavg = sum( solaBdata ) / length( solaBdata );
solaCavg = sum( solaCdata ) / length( solaCdata );
solaDavg = sum( solaDdata ) / length( solaDdata );
solaEavg = sum( solaEdata ) / length( solaEdata );
```

```
solaFavg = sum( solaFdata ) / length( solaFdata );

olaAavg = sum( olaAdata ) / length( olaAdata );
olaBavg = sum( olaBdata ) / length( olaBdata );
olaCavg = sum( olaCdata ) / length( olaCdata );
olaDavg = sum( olaDdata ) / length( olaDdata );
olaEavg = sum( olaEdata ) / length( olaEdata );
olaFavg = sum( olaFdata ) / length( olaFdata );

% find max of each audio file
maxAmax = max( [ rawAmax lockAmax peakAmax bankAmax solaAmax olaAmax ] );
maxBmax = max( [ rawBmax lockBmax peakBmax bankBmax solaBmax olaBmax ] );
maxCmax = max( [ rawCmax lockCmax peakCmax bankCmax solaCmax olaCmax ] );
maxDmax = max( [ rawDmax lockDmax peakDmax bankDmax solaDmax olaDmax ] );
maxEmax = max( [ rawEmax lockEmax peakEmax bankEmax solaEmax olaEmax ] );
maxFmax = max( [ rawFmax lockFmax peakFmax bankFmax solaFmax olaFmax ] );

% create matricies of max values for plots
maxA = [ rawAmax lockAmax peakAmax bankAmax solaAmax olaAmax ];
maxB = [ rawBmax lockBmax peakBmax bankBmax solaBmax olaBmax ];
maxC = [ rawCmax lockCmax peakCmax bankCmax solaCmax olaCmax ];
maxD = [ rawDmax lockDmax peakDmax bankDmax solaDmax olaDmax ];
maxE = [ rawEmax lockEmax peakEmax bankEmax solaEmax olaEmax ];
maxF = [ rawFmax lockFmax peakFmax bankFmax solaFmax olaFmax ];

% create matrices of avg values for plots
avgA = [ rawAavg lockAavg peakAavg bankAavg solaAavg olaAavg ];
avgB = [ rawBavg lockBavg peakBavg bankBavg solaBavg olaBavg ];
avgC = [ rawCavg lockCavg peakCavg bankCavg solaCavg olaCavg ];
avgD = [ rawDavg lockDavg peakDavg bankDavg solaDavg olaDavg ];
avgE = [ rawEavg lockEavg peakEavg bankEavg solaEavg olaEavg ];
avgF = [ rawFavg lockFavg peakFavg bankFavg solaFavg olaFavg ];

% Plot
%--------------------------------------------------------------------------

% graph formatting
fontName   = 'Times New Roman';
fontSize   = 12;
xTxt = 0.125;
yTxt = 1.2;
yOff = 0.13;

% set up plot window
fig = figure( 1 );
set( fig, 'Name', sprintf( '%s', name ) );
set( fig, 'Position', [ 0 0 800 1000 ] );
set( fig, 'defaultAxesFontName', fontName );
set( fig, 'defaultTextFontName', fontName );
% set( fig, 'defaultlinelinewidth', 3 );
set( fig, 'defaultaxescolororder', [ 0.8 0.2 0.2 ] );
```

```
% fileA
%-------------------------------------------------------------------------
plotL = avgA;
plotR = maxA;
fileName = fileAname;
sp = subplot( 6, 2, 1 );
bar( plotL, 'FaceColor', [ 0.8 0.2 0.2 ] );
set( sp, 'Position', [ 0.1, 5.5/7, 0.39, 1/7.333 ] );
axis( [ 0 7 0 ( maxAmax * 1.3 ) ] );
for( i = 1 : 6 )
    text( i, plotL( i ) + maxAmax * yOff, num2str( plotL( i ), '%2.5f' ),
'HorizontalAlignment', 'center', 'BackgroundColor', 'w', 'EdgeColor', 'k',
'FontSize', 9 );
end
set( gca, 'TickLength', [ 0 0 ] );
set( gca, 'YGrid', 'on', 'XGrid', 'off' );
set( gca, 'XTickLabel', [] );
temp = get( gca );
yticks = get( gca, 'YTick' );
set( gca, 'YTickLabel', yticks );
text( xTxt, yTxt * maxAmax, [ ' ' fileName ' ' ], 'BackgroundColor', 'w',
'EdgeColor', 'k' );
title( 'Average', 'FontSize', fontSize );

sp = subplot( 6, 2, 2 );
bar( plotR, 'FaceColor', [ 0.8 0.2 0.2 ] );
set( sp, 'Position', [ 0.5, 5.5/7, 0.39, 1/7.333 ] );
axis( [ 0 7 0 ( maxAmax * 1.3 ) ] );
for( i = 1 : 6 )
    text( i, plotR( i ) + maxAmax * yOff, num2str( plotR( i ), '%2.5f' ),
'HorizontalAlignment', 'center', 'BackgroundColor', 'w', 'EdgeColor', 'k',
'FontSize', 9 );
end
set( gca, 'TickLength', [ 0 0 ] );
set( gca, 'YGrid', 'on', 'XGrid', 'off' );
set( gca, 'XTickLabel', [] );
set( gca, 'YTickLabel', [] );
title( 'Maximum', 'FontSize', fontSize );

% fileB
%-------------------------------------------------------------------------
plotL = avgB;
plotR = maxB;
fileName = fileBname;
sp = subplot( 6, 2, 3 );
bar( plotL, 'FaceColor', [ 0.8 0.2 0.2 ] );
set( sp, 'Position', [ 0.1, 4.5/7, 0.39, 1/7.333 ] );
axis( [ 0 7 0 ( maxBmax * 1.3 ) ] );
for( i = 1 : 6 )
    text( i, plotL( i ) + maxBmax * yOff, num2str( plotL( i ), '%2.5f' ),
'HorizontalAlignment', 'center', 'BackgroundColor', 'w', 'EdgeColor', 'k',
'FontSize', 9 );
```

```
end
set( gca, 'TickLength', [ 0 0 ] );
set( gca, 'YGrid', 'on', 'XGrid', 'off' );
set( gca, 'XTickLabel', [] );
temp = get( gca );
yticks = get( gca, 'YTick' );
set( gca, 'YTickLabel', yticks );
text( xTxt, yTxt * maxBmax, [ ' ' fileName ' ' ], 'BackgroundColor', 'w',
'EdgeColor', 'k' );

sp = subplot( 6, 2, 4 );
bar( plotR, 'FaceColor', [ 0.8 0.2 0.2 ] );
set( sp, 'Position', [ 0.5, 4.5/7, 0.39, 1/7.333 ] );
axis( [ 0 7 0 ( maxBmax * 1.3 ) ] );
for( i = 1 : 6 )
    text( i, plotR( i ) + maxBmax * yOff, num2str( plotR( i ), '%2.5f' ),
'HorizontalAlignment', 'center', 'BackgroundColor', 'w', 'EdgeColor', 'k',
'FontSize', 9 );
end
set( gca, 'TickLength', [ 0 0 ] );
set( gca, 'YGrid', 'on', 'XGrid', 'off' );
set( gca, 'XTickLabel', [] );
set( gca, 'YTickLabel', [] );

% fileC
%-------------------------------------------------------------------------
plotL = avgC;
plotR = maxC;
fileName = fileCname;
sp = subplot( 6, 2, 5 );
bar( plotL, 'FaceColor', [ 0.8 0.2 0.2 ] );
set( sp, 'Position', [ 0.1, 3.5/7, 0.39, 1/7.333 ] );
axis( [ 0 7 0 ( maxCmax * 1.3 ) ] );
for( i = 1 : 6 )
    text( i, plotL( i ) + maxCmax * yOff, num2str( plotL( i ), '%2.5f' ),
'HorizontalAlignment', 'center', 'BackgroundColor', 'w', 'EdgeColor', 'k',
'FontSize', 9 );
end
set( gca, 'TickLength', [ 0 0 ] );
set( gca, 'YGrid', 'on', 'XGrid', 'off' );
set( gca, 'XTickLabel', [] );
temp = get( gca );
yticks = get( gca, 'YTick' );
set( gca, 'YTickLabel', yticks );
text( xTxt, yTxt * maxCmax, [ ' ' fileName ' ' ], 'BackgroundColor', 'w',
'EdgeColor', 'k' );

sp = subplot( 6, 2, 6 );
bar( plotR, 'FaceColor', [ 0.8 0.2 0.2 ] );
set( sp, 'Position', [ 0.5, 3.5/7, 0.39, 1/7.333 ] );
axis( [ 0 7 0 ( maxCmax * 1.3 ) ] );
for( i = 1 : 6 )
```

```
    text( i, plotR( i ) + maxCmax * yOff, num2str( plotR( i ), '%2.5f' ),
'HorizontalAlignment', 'center', 'BackgroundColor', 'w', 'EdgeColor', 'k',
'FontSize', 9 );
end
set( gca, 'TickLength', [ 0 0 ] );
set( gca, 'YGrid', 'on', 'XGrid', 'off' );
set( gca, 'XTickLabel', [] );
set( gca, 'YTickLabel', [] );

% fileD
%-------------------------------------------------------------------------
plotL = avgD;
plotR = maxD;
fileName = fileDname;
sp = subplot( 6, 2, 7 );
bar( plotL, 'FaceColor', [ 0.8 0.2 0.2 ] );
set( sp, 'Position', [ 0.1, 2.5/7, 0.39, 1/7.333 ] );
axis( [ 0 7 0 ( maxDmax * 1.3 ) ] );
for( i = 1 : 6 )
    text( i, plotL( i ) + maxDmax * yOff, num2str( plotL( i ), '%2.5f' ),
'HorizontalAlignment', 'center', 'BackgroundColor', 'w', 'EdgeColor', 'k',
'FontSize', 9 );
end
set( gca, 'TickLength', [ 0 0 ] );
set( gca, 'YGrid', 'on', 'XGrid', 'off' );
set( gca, 'XTickLabel', [] );
temp = get( gca );
yticks = get( gca, 'YTick' );
set( gca, 'YTickLabel', yticks );
text( xTxt, yTxt * maxDmax, [ ' ' fileName ' ' ], 'BackgroundColor', 'w',
'EdgeColor', 'k' );

sp = subplot( 6, 2, 8 );
bar( plotR, 'FaceColor', [ 0.8 0.2 0.2 ] );
set( sp, 'Position', [ 0.5, 2.5/7, 0.39, 1/7.333 ] );
axis( [ 0 7 0 ( maxDmax * 1.3 ) ] );
for( i = 1 : 6 )
    text( i, plotR( i ) + maxDmax * yOff, num2str( plotR( i ), '%2.5f' ),
'HorizontalAlignment', 'center', 'BackgroundColor', 'w', 'EdgeColor', 'k',
'FontSize', 9 );
end
set( gca, 'TickLength', [ 0 0 ] );
set( gca, 'YGrid', 'on', 'XGrid', 'off' );
set( gca, 'XTickLabel', [] );
set( gca, 'YTickLabel', [] );

% fileE
%-------------------------------------------------------------------------
plotL = avgE;
plotR = maxE;
fileName = fileEname;
sp = subplot( 6, 2, 9 );
```

```
bar( plotL, 'FaceColor', [ 0.8 0.2 0.2 ] );
set( sp, 'Position', [ 0.1, 1.5/7, 0.39, 1/7.333 ] );
axis( [ 0 7 0 ( maxEmax * 1.3 ) ] );
for( i = 1 : 6 )
    text( i, plotL( i ) + maxEmax * yOff, num2str( plotL( i ), '%2.5f' ),
'HorizontalAlignment', 'center', 'BackgroundColor', 'w', 'EdgeColor', 'k',
'FontSize', 9 );
end
set( gca, 'TickLength', [ 0 0 ] );
set( gca, 'YGrid', 'on', 'XGrid', 'off' );
set( gca, 'XTickLabel', [] );
temp = get( gca );
yticks = get( gca, 'YTick' );
set( gca, 'YTickLabel', yticks );
text( xTxt, yTxt * maxEmax, [ ' ' fileName ' ' ], 'BackgroundColor', 'w',
'EdgeColor', 'k' );

sp = subplot( 6, 2, 10 );
bar( plotR, 'FaceColor', [ 0.8 0.2 0.2 ] );
set( sp, 'Position', [ 0.5, 1.5/7, 0.39, 1/7.333 ] );
axis( [ 0 7 0 ( maxEmax * 1.3 ) ] );
for( i = 1 : 6 )
    text( i, plotR( i ) + maxEmax * yOff, num2str( plotR( i ), '%2.5f' ),
'HorizontalAlignment', 'center', 'BackgroundColor', 'w', 'EdgeColor', 'k',
'FontSize', 9 );
end
set( gca, 'TickLength', [ 0 0 ] );
set( gca, 'YGrid', 'on', 'XGrid', 'off' );
set( gca, 'XTickLabel', [] );
set( gca, 'YTickLabel', [] );

% fileF
%-------------------------------------------------------------------------
plotL = avgF;
plotR = maxF;
fileName = fileFname;
sp = subplot( 6, 2, 11 );
bar( plotL, 'FaceColor', [ 0.8 0.2 0.2 ] );
set( sp, 'Position', [ 0.1, 0.5/7, 0.39, 1/7.333 ] );
axis( [ 0 7 0 ( maxFmax * 1.3 ) ] );
for( i = 1 : 6 )
    text( i, plotL( i ) + maxFmax * yOff, num2str( plotL( i ), '%2.5f' ),
'HorizontalAlignment', 'center', 'BackgroundColor', 'w', 'EdgeColor', 'k',
'FontSize', 9 );
end
set( gca, 'TickLength', [ 0 0 ] );
set( gca, 'YGrid', 'on', 'XGrid', 'off' );
set( gca, 'XTickLabel', algos );
yLab = ylabel( 'Sones' );
temp = get( gca );
yticks = get( gca, 'YTick' );
set( gca, 'YTickLabel', yticks );
```

```
text( xTxt, yTxt * maxFmax, [ ' ' fileName ' ' ], 'BackgroundColor', 'w',
'EdgeColor', 'k' );

sp = subplot( 6, 2, 12 );
bar( plotR, 'FaceColor', [ 0.8 0.2 0.2 ] );
set( sp, 'Position', [ 0.5, 0.5/7, 0.39, 1/7.333 ] );
axis( [ 0 7 0 ( maxFmax * 1.3 ) ] );
for( i = 1 : 6 )
    text( i, plotR( i ) + maxFmax * yOff, num2str( plotR( i ), '%2.5f' ),
'HorizontalAlignment', 'center', 'BackgroundColor', 'w', 'EdgeColor', 'k',
'FontSize', 9 );
end
set( gca, 'TickLength', [ 0 0 ] );
set( gca, 'XTickLabel', algos );
set( gca, 'YTickLabel', [] );
set( gca, 'YGrid', 'on', 'XGrid', 'off' );
xLab = xlabel( 'Algorithms' );

% tighen up figure borders
tightfig();

% move x and y labels
set( xLab, 'Position', [ -0.1 ( maxFmax * -0.16 ) ] );
set( yLab, 'Position', [ -0.625 ( ( maxFmax * 1.3 * 6.25 ) / 2 ) ] );

% write plot to file
hgexport( fig, [ 'summary.mov.eps' ] );

% EOF
```

### 8.4.15 – Coordinating Script

```
%-------------------------------------------------------------------------
% Coordinating Script - Stretch, Analyze, and Graph
%
% Cooper Baker - 2014
%-------------------------------------------------------------------------

% set up paths
%-------------------------------------------------------------------------
restoredefaultpath;

p = genpath( '../analyze' );
addpath( p );
p = genpath( '../audio' );
addpath( p );
p = genpath( '../graph' );
addpath( p );
p = genpath( '../grain' );
addpath( p );
p = genpath( '../pvoc' );
addpath( p );
```

```
p = genpath( '../tones' );
addpath( p );

% 0/1 = off/on
stretchAudio = 0;
makeFileDisp = 0;
makeGraphs   = 0;
compareError = 1;

% create stretched audio files
%-------------------------------------------------------------------------
if( stretchAudio )

    [ filePath, fileName ] = a_getFile( 'amen.wav' );
    fft_bank        ( filePath, fileName );
    fft_ifft_classic( filePath, fileName );
    fft_ifft_lock   ( filePath, fileName );
    fft_ifft_peak   ( filePath, fileName );
    ola             ( filePath, fileName );
    sola            ( filePath, fileName );

    [ filePath, fileName ] = a_getFile( 'autumn.wav' );
    fft_bank        ( filePath, fileName );
    fft_ifft_classic( filePath, fileName );
    fft_ifft_lock   ( filePath, fileName );
    fft_ifft_peak   ( filePath, fileName );
    ola             ( filePath, fileName );
    sola            ( filePath, fileName );

    [ filePath, fileName ] = a_getFile( 'peaches.wav' );
    fft_bank        ( filePath, fileName );
    fft_ifft_classic( filePath, fileName );
    fft_ifft_lock   ( filePath, fileName );
    fft_ifft_peak   ( filePath, fileName );
    ola             ( filePath, fileName );
    sola            ( filePath, fileName );

    [ filePath, fileName ] = a_getFile( 'sine.wav' );
    fft_bank        ( filePath, fileName );
    fft_ifft_classic( filePath, fileName );
    fft_ifft_lock   ( filePath, fileName );
    fft_ifft_peak   ( filePath, fileName );
    ola             ( filePath, fileName );
    sola            ( filePath, fileName );

    [ filePath, fileName ] = a_getFile( 'sweep.wav' );
    fft_bank        ( filePath, fileName );
    fft_ifft_classic( filePath, fileName );
    fft_ifft_lock   ( filePath, fileName );
    fft_ifft_peak   ( filePath, fileName );
    ola             ( filePath, fileName );
    sola            ( filePath, fileName );
```

```
    [ filePath, fileName ] = a_getFile( 'square.wav' );
    fft_bank        ( filePath, fileName );
    fft_ifft_classic( filePath, fileName );
    fft_ifft_lock   ( filePath, fileName );
    fft_ifft_peak   ( filePath, fileName );
    ola             ( filePath, fileName );
    sola            ( filePath, fileName );

end

% generate input file displays
%-------------------------------------------------------------------------
if( makeFileDisp )
    [ filePath, fileName ] = a_getFile( 'amen.wav' );
    file_disp( filePath, fileName );

    [ filePath, fileName ] = a_getFile( 'autumn.wav' );
    file_disp( filePath, fileName );

    [ filePath, fileName ] = a_getFile( 'peaches.wav' );
    file_disp( filePath, fileName );

    [ filePath, fileName ] = a_getFile( 'sine.wav' );
    file_disp( filePath, fileName );

    [ filePath, fileName ] = a_getFile( 'sweep.wav' );
    file_disp( filePath, fileName );

    [ filePath, fileName ] = a_getFile( 'square.wav' );
    file_disp( filePath, fileName );
end

% perform analysis on stretched files and generate graphs
%-------------------------------------------------------------------------
if( makeGraphs )
    % amen.wav
    [ idlPath, idlName ] = a_getFile( 'amen.wav' );
    [ strPath, strName ] = a_getFile( 'amen.classic.wav' );
    avg_file( idlPath, idlName, strPath, strName );
    mov_file( idlPath, idlName, strPath, strName );
    dif_file( idlPath, idlName, strPath, strName );

    [ strPath, strName ] = a_getFile( 'amen.lock.wav' );
    avg_file( idlPath, idlName, strPath, strName );
    mov_file( idlPath, idlName, strPath, strName );
    dif_file( idlPath, idlName, strPath, strName );

    [ strPath, strName ] = a_getFile( 'amen.peak.wav' );
    avg_file( idlPath, idlName, strPath, strName );
    mov_file( idlPath, idlName, strPath, strName );
    dif_file( idlPath, idlName, strPath, strName );
```

```
[ strPath, strName ] = a_getFile( 'amen.bank.wav' );
avg_file( idlPath, idlName, strPath, strName );
mov_file( idlPath, idlName, strPath, strName );
dif_file( idlPath, idlName, strPath, strName );

[ strPath, strName ] = a_getFile( 'amen.ola.wav' );
avg_file( idlPath, idlName, strPath, strName );
mov_file( idlPath, idlName, strPath, strName );
dif_file( idlPath, idlName, strPath, strName );

[ strPath, strName ] = a_getFile( 'amen.sola.wav' );
avg_file( idlPath, idlName, strPath, strName );
mov_file( idlPath, idlName, strPath, strName );
dif_file( idlPath, idlName, strPath, strName );

% autumn.wav
[ idlPath, idlName ] = a_getFile( 'autumn.wav' );
[ strPath, strName ] = a_getFile( 'autumn.classic.wav' );
avg_file( idlPath, idlName, strPath, strName );
mov_file( idlPath, idlName, strPath, strName );
dif_file( idlPath, idlName, strPath, strName );

[ strPath, strName ] = a_getFile( 'autumn.lock.wav' );
avg_file( idlPath, idlName, strPath, strName );
mov_file( idlPath, idlName, strPath, strName );
dif_file( idlPath, idlName, strPath, strName );

[ strPath, strName ] = a_getFile( 'autumn.peak.wav' );
avg_file( idlPath, idlName, strPath, strName );
mov_file( idlPath, idlName, strPath, strName );
dif_file( idlPath, idlName, strPath, strName );

[ strPath, strName ] = a_getFile( 'autumn.bank.wav' );
avg_file( idlPath, idlName, strPath, strName );
mov_file( idlPath, idlName, strPath, strName );
dif_file( idlPath, idlName, strPath, strName );

[ strPath, strName ] = a_getFile( 'autumn.ola.wav' );
avg_file( idlPath, idlName, strPath, strName );
mov_file( idlPath, idlName, strPath, strName );
dif_file( idlPath, idlName, strPath, strName );

[ strPath, strName ] = a_getFile( 'autumn.sola.wav' );
avg_file( idlPath, idlName, strPath, strName );
mov_file( idlPath, idlName, strPath, strName );
dif_file( idlPath, idlName, strPath, strName );

% peaches.wav
[ idlPath, idlName ] = a_getFile( 'peaches.wav' );
[ strPath, strName ] = a_getFile( 'peaches.classic.wav' );
avg_file( idlPath, idlName, strPath, strName );
```

```
mov_file( idlPath, idlName, strPath, strName );
dif_file( idlPath, idlName, strPath, strName );

[ strPath, strName ] = a_getFile( 'peaches.lock.wav' );
avg_file( idlPath, idlName, strPath, strName );
mov_file( idlPath, idlName, strPath, strName );
dif_file( idlPath, idlName, strPath, strName );

[ strPath, strName ] = a_getFile( 'peaches.peak.wav' );
avg_file( idlPath, idlName, strPath, strName );
mov_file( idlPath, idlName, strPath, strName );
dif_file( idlPath, idlName, strPath, strName );

[ strPath, strName ] = a_getFile( 'peaches.bank.wav' );
avg_file( idlPath, idlName, strPath, strName );
mov_file( idlPath, idlName, strPath, strName );
dif_file( idlPath, idlName, strPath, strName );

[ strPath, strName ] = a_getFile( 'peaches.ola.wav' );
avg_file( idlPath, idlName, strPath, strName );
mov_file( idlPath, idlName, strPath, strName );
dif_file( idlPath, idlName, strPath, strName );

[ strPath, strName ] = a_getFile( 'peaches.sola.wav' );
avg_file( idlPath, idlName, strPath, strName );
mov_file( idlPath, idlName, strPath, strName );
dif_file( idlPath, idlName, strPath, strName );

% sine_ideal.wav
[ idlPath, idlName ] = a_getFile( 'sine_ideal.wav' );
[ strPath, strName ] = a_getFile( 'sine.classic.wav' );
avg_tone( idlPath, idlName, strPath, strName );
mov_tone( idlPath, idlName, strPath, strName );
dif_tone( idlPath, idlName, strPath, strName );

[ strPath, strName ] = a_getFile( 'sine.lock.wav' );
avg_tone( idlPath, idlName, strPath, strName );
mov_tone( idlPath, idlName, strPath, strName );
dif_tone( idlPath, idlName, strPath, strName );

[ strPath, strName ] = a_getFile( 'sine.peak.wav' );
avg_tone( idlPath, idlName, strPath, strName );
mov_tone( idlPath, idlName, strPath, strName );
dif_tone( idlPath, idlName, strPath, strName );

[ strPath, strName ] = a_getFile( 'sine.bank.wav' );
avg_tone( idlPath, idlName, strPath, strName );
mov_tone( idlPath, idlName, strPath, strName );
dif_tone( idlPath, idlName, strPath, strName );

[ strPath, strName ] = a_getFile( 'sine.ola.wav' );
avg_tone( idlPath, idlName, strPath, strName );
```

```
mov_tone( idlPath, idlName, strPath, strName );
dif_tone( idlPath, idlName, strPath, strName );

[ strPath, strName ] = a_getFile( 'sine.sola.wav' );
avg_tone( idlPath, idlName, strPath, strName );
mov_tone( idlPath, idlName, strPath, strName );
dif_tone( idlPath, idlName, strPath, strName );

% square_ideal.wav
[ idlPath, idlName ] = a_getFile( 'square_ideal.wav' );
[ strPath, strName ] = a_getFile( 'square.classic.wav' );
avg_tone( idlPath, idlName, strPath, strName );
mov_tone( idlPath, idlName, strPath, strName );
dif_tone( idlPath, idlName, strPath, strName );

[ strPath, strName ] = a_getFile( 'square.lock.wav' );
avg_tone( idlPath, idlName, strPath, strName );
mov_tone( idlPath, idlName, strPath, strName );
dif_tone( idlPath, idlName, strPath, strName );

[ strPath, strName ] = a_getFile( 'square.peak.wav' );
avg_tone( idlPath, idlName, strPath, strName );
mov_tone( idlPath, idlName, strPath, strName );
dif_tone( idlPath, idlName, strPath, strName );

[ strPath, strName ] = a_getFile( 'square.bank.wav' );
avg_tone( idlPath, idlName, strPath, strName );
mov_tone( idlPath, idlName, strPath, strName );
dif_tone( idlPath, idlName, strPath, strName );

[ strPath, strName ] = a_getFile( 'square.ola.wav' );
avg_tone( idlPath, idlName, strPath, strName );
mov_tone( idlPath, idlName, strPath, strName );
dif_tone( idlPath, idlName, strPath, strName );

[ strPath, strName ] = a_getFile( 'square.sola.wav' );
avg_tone( idlPath, idlName, strPath, strName );
mov_tone( idlPath, idlName, strPath, strName );
dif_tone( idlPath, idlName, strPath, strName );

% sweep_ideal.wav
[ idlPath, idlName ] = a_getFile( 'sweep_ideal.wav' );
[ strPath, strName ] = a_getFile( 'sweep.classic.wav' );
avg_tone( idlPath, idlName, strPath, strName );
mov_tone( idlPath, idlName, strPath, strName );
dif_tone( idlPath, idlName, strPath, strName );

[ strPath, strName ] = a_getFile( 'sweep.lock.wav' );
avg_tone( idlPath, idlName, strPath, strName );
mov_tone( idlPath, idlName, strPath, strName );
dif_tone( idlPath, idlName, strPath, strName );
```

```
    [ strPath, strName ] = a_getFile( 'sweep.peak.wav' );
    avg_tone( idlPath, idlName, strPath, strName );
    mov_tone( idlPath, idlName, strPath, strName );
    dif_tone( idlPath, idlName, strPath, strName );

    [ strPath, strName ] = a_getFile( 'sweep.bank.wav' );
    avg_tone( idlPath, idlName, strPath, strName );
    mov_tone( idlPath, idlName, strPath, strName );
    dif_tone( idlPath, idlName, strPath, strName );

    [ strPath, strName ] = a_getFile( 'sweep.ola.wav' );
    avg_tone( idlPath, idlName, strPath, strName );
    mov_tone( idlPath, idlName, strPath, strName );
    dif_tone( idlPath, idlName, strPath, strName );

    [ strPath, strName ] = a_getFile( 'sweep.sola.wav' );
    avg_tone( idlPath, idlName, strPath, strName );
    mov_tone( idlPath, idlName, strPath, strName );
    dif_tone( idlPath, idlName, strPath, strName );
end

if( compareError )
    % compare average spectrum error
    avg_error( 'amen' );
    avg_error( 'autumn' );
    avg_error( 'peaches' );
    avg_error( 'sine' );
    avg_error( 'sweep' );
    avg_error( 'square' );

    % compare moving spectral average error
    mov_error( 'amen' );
    mov_error( 'autumn' );
    mov_error( 'peaches' );
    mov_error( 'sine' );
    mov_error( 'sweep' );
    mov_error( 'square' );

    % compare zoomed moving spectral average error
    mov_error_detail( 'sine' );
    mov_error_detail( 'square' );
    mov_error_detail( 'sweep' );

    % compare error spectrograms
    dif_error( 'amen' );
    dif_error( 'autumn' );
    dif_error( 'peaches' );
    dif_error( 'sine' );
    dif_error( 'sweep' );
    dif_error( 'square' );
```

```
    % compare zoomed error spectrograms
    dif_error_detail( 'sine' );
    dif_error_detail( 'sweep' );
    dif_error_detail( 'square' );

    % generate summary charts
    avg_summary();
    mov_summary();
end

% EOF
```

## 9 – References


[Bak13] Baker, Cooper, and Tom Erbe. “Pd Spectral Toolkit.” *Proceedings of the International Computer Music Conference* (2013): 410-413.

[Bra00] Bracewell, Ronald Newbold. *The Fourier Transform and its Applications*. 3rd ed. New York: McGraw-Hill, 1986.

[Dep91] De Poli, Giovanni, Aldo Piccialli, and Curtis Roads. *Representations of Musical Signals*. MIT press, 1991.

[Dod97] Dodge, Charles, and Thomas A. Jerse. *Computer Music: Synthesis, Composition, and Performance*. Macmillan Library Reference, 1997.

[Dol86] Dolson, Mark. "The Phase Vocoder: A Tutorial." *Computer Music Journal* (1986): 14-27.

[Fla66] Flanagan, James Loton, and R. M. Golden. "Phase Vocoder." *Bell System Technical Journal* 45.9 (1966): 1493-1509.

[Jaf87a] Jaffe, David A. "Spectrum Analysis Tutorial, Part 1: The Discrete Fourier Transform." *Computer Music Journal* (1987): 9-24.

[Jaf87b] Jaffe, David A. "Spectrum Analysis Tutorial, Part 2: Properties and Applications of the Discrete Fourier Transform." *Computer Music Journal* (1987): 17-35.

[Lar97] Laroche, Jean, and Mark Dolson. "Phase-Vocoder: About This Phasiness Business." *Applications of Signal Processing to Audio and Acoustics, 1997. 1997 IEEE ASSP Workshop on*. IEEE, 1997.

[Lar99] Laroche, Jean, and Mark Dolson. "Improved Phase Vocoder Time-Scale Modification of Audio." *Speech and Audio Processing, IEEE Transactions on* 7.3 (1999): 323-332.

[Loy06] Loy, Gareth. *Musimathics: The Mathematical Foundations of Music*. Vol. 1. MIT Press, 2011.

[Mat14] MathWorks, Inc. *MATLAB: The Language of Technical Computing. Desktop Tools and Development Environment, Version R2014a*. Computer Software. MathWorks, 2014.

[Mak86] Makhoul, John, and Amro El-Jaroudi. "Time-Scale Modification In Medium to Low Rate Speech Coding." *Acoustics, Speech, and Signal Processing, IEEE International Conference on ICASSP'86.*. Vol. 11. IEEE, 1986.

[Moo90] Moore, F. Richard. *Elements of Computer Music*. Prentice-Hall, Inc., 1990.

[Par10] Park, Tae Hong. *Introduction to Digital Signal Processing: Computer Musically Speaking*. World Scientific, 2010.

[Pre07] Press, William H., Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery. *Numerical Recipes: The Art of Scientific Computing*. 3rd ed. Cambridge University Press, 2007.

[Puc95] Puckette, Miller. "Phase-Locked Vocoder." *Applications of Signal Processing to Audio and Acoustics, 1995., IEEE ASSP Workshop on*. IEEE, 1995.

[Puc07] Puckette, Miller. *The Theory and Technique of Electronic Music*. World Scientific, 2007.

[Puc14] Puckette, Miller. *Pure Data, Version 0.46-2*. Computer Software. 2014.

[Roa96] Curtis Roads. *The Computer Music Tutorial*. MIT press, 1996.

[Ros02] Rossing, Thomas D., F. Richard Moore, and Paul A. Wheeler. *The Science of Sound*. 3rd ed. Reading, MA: Addison-Wesley, 1990.

[Rou85] Roucos, Salim, and Alexander Wilgus. "High Quality Time-Scale Modification for Speech." *Acoustics, Speech, and Signal Processing, IEEE International Conference on ICASSP'85.*. Vol. 10. IEEE, 1985.

[Sin75] The Singers Unlimited. “Autumn In New York.” *A Capella II*. MPS Records. 1975. Vinyl Record.

[Ste96] Steiglitz, Kenneth. *A Digital Signal Processing Primer with Applications to Digital Audio and Computer Music*. Addison-Wesley, 1996.

[Str85] Strawn, John. *Digital audio signal processing: an anthology*. W. Kaufmann, 1985.

[Win69] The Winstons. “Amen Brother.” *Color Him Father*. Metromedia Records. 1969. Vinyl Record.

[Zap69] Frank Zappa. “Peaches en Regalia.” *Hot Rats*. Bizarre Records. 1969. Vinyl Record.

[Zöl08a] Zölzer, Udo. *DAFX: Digital Audio Effects*. John Wiley & Sons, Ltd., 2008.

[Zöl08b] Zölzer, Udo. *Digital Audio Signal* Processing. 2nd ed. John Wiley & Sons, Ltd., 2008.

[Zöl11] Zölzer, Udo. *DAFX: Digital Audio Effects*. 2nd ed. John Wiley & Sons, Ltd., 2011.

[Eof] “End-of-File.” *Cplusplus.com.* Code Sites Ltd. 4 April 1999. Web. 23 March 2015.