SONIC IMAGERY: A VIEW OF MUSIC VIA MATHEMATICAL COMPUTER SCIENCE AND SIGNAL PROCESSING

by

SHANNON STEINMETZ

Bachelor of Science, MSU, 1999

A thesis submitted to the

Faculty of the Graduate School of the

University of Colorado in partial fulfillment

of the requirements for the degree of

Master of Integrated Sciences

Integrated Sciences

2016

This thesis for the Master of Integrated Sciences degree by

Shannon Steinmetz

has been approved for the

Integrated Sciences Program

by

Ellen Gethner, Chair

Gita Alaghaband

Varis Carey

April 18, 2016

Steinmetz, Shannon (MIS, Integrated Sciences)

Sonic Imagery: A View of Music via Mathematical Computer Science and Signal Processing

Thesis directed by Associate Professor Ellen Gethner

ABSTRACT

For centuries humans have strived to visualize. From cave paintings to modern artworks, we are beings of beauty and expression. A condition known as Synesthesia provides some with the ability to see sound as it occurs. We propose a mathematical computer science and software foundation capable of transforming a sound *a priori* into a visual representation. We explore and exploit techniques in signal processing, Fourier analysis, group theory and music theory and attach this work to a psychological foundation for coloring and visuals. We propose a new theorem for tone detection, a parallelized FFT and provide an algorithm for chord detection. We provide an extensible software architecture and implementation and compile the results of a small survey.

The form and content of this abstract are approved. I recommend its publication.

Approved: Ellen Gethner

DEDICATION

This work is dedicated to my loving family Diane, Kerlin, Brandie, Lathan, Olive and Wesley Steinmetz, Harry Lordeno, Syrina, KJ, Javier and my little buddies Paco and Rambo who sat hours on end being ignored while I tapped away at my computer and scribbled on my white board. I would also like to dedicate this work to my great friends Charly Randall and Brian Parker who gave me confidence, inspiration and ideas throughout my life as well as Mike Iiams, without whom, I'd never have been given the opportunities that lead me down this path.

ACKNOWLEDGMENT

I would like to thank the University of Colorado and all the wonderful folks that provided us opportunities to share and grow our research. This work could not have been possible without great mentors. I would like to thank Dr. Ellen Gethner for her ideas, inspiration and for being the single greatest contributor to my academic experience. I would like to thank Dr. Martin Huber who showed me patience and understanding during tough times and offered me this incredible opportunity. Without these wonderful scholars I literally would not have made it. I would also like to thank Jason Fisk for encouraging me to grow and Jim Muller, Dr. Bob Lindeman, Doc Stoner, Scott Cambell, Dr. Blane Johnson and Jeff Caulder for being role models and mentors for so many years.

TABLE OF CONTENTS

TABLES

Table

FIGURES

Figure

1. Introduction

For centuries humans have strived to visualize. From cave paintings to modern artworks, we are beings of beauty and expression. Within the very nature of our language is the underlying desire to express what we feel in terms of pictographic imagery. The English language is ladled with terms such as "let me see," or "see what I mean," and rarely do we give the underlying meaning of terms a second thought. When given new information in the classroom we often desire a picture of the concept to solidify understanding. When we hear the words *"c squared equals a squared plus b squared"* they portray little intuition but when shown a right triangle something clicks. There is often a chasm between representation and intuition constantly being filled by new technology and ideas. It is within this chasm we begin our climb.

Our thesis is inspired by the idea of *Synesthesia*, which is defined as the cross modality of senses [37, 47] and we aim to devise a mathematical computer science capable of transforming the physical shape of a sound into an intuitive representation, agnostic of culture or background. For example, imagine a musician with an instrument connected to his or her computer and as the musician plays s/he sees amazing patterns, shapes and colors "congruent" to the harmony in real-time that represents the actual "mood" of the melody. Similarly, one may select a song from an mp3, mp4, or .wav file and play the music into an application capable of rendering sonorities[1] as they emerge. Our research leverages Fourier Analysis, Signal Processing Detection/Characterization, Computer Graphics/Animation, Group Theory, Musical Geometry, Music Theory and Psychology. This task is daunting, it requires not only a profound understanding of a number of advanced scientific disciplines but the ability to integrate several research areas into a cohesive model involving theoretical, subjective and experimental methodologies.

[1] A term in music theory to describe a collection of pitches.

In order to provide a rigorous thesis and still be able to maintain a level of creativity we address three major fronts: *a)* the construction of a mathematical model and computer algorithms *b)* the substantiation and derivation of a philosophy involving the human perception of music *c)* the aesthetics of computer generated *Art*. Building from the works of Dmitri Tymoczko [42,43], Cytowic and Wood [36,37], Stephen W. Smith [38], James Cooly and John Tukey [17], Bello, De Poli, Oppenheim [6,22,30], Michal Levy [24] and Gethner, Steinmetz & Verbeke [9] to name a few. Our thesis takes one small step toward the derivation of a model (mathematics, algorithms, software and artistic creativity) capable of transforming the physical "shape" of a sound into imagery divorced from cultural subjectivity.

2. Inspiration

2.1 Previous Work

Since the dawn of the electronic era, mathematicians, physicists, computer scientists and electrical engineers have been attacking the seemingly unsolvable problem of blindly characterizing a time series [1]. Whether we are parsing a doppler RADAR system, human speech or music we leverage much of the same mathematics and techniques. Our endeavour hinges on the ability to extract notes from a time series of raw energy impulses. In the 1970's MIT's Alan Opppenheim pioneered some of first techniques in speech transcription and signals analysis. In 1977 the University of Michigan's Martin Piszczalski and Bernard Galler implemented one of the first computer algorithms to transcribe monophonic tones. Later many experts such as Juan Pablo Bello and Giovanni De Poli added various methodologies to improve transcription of monophonic and polyphonic instruments, high frequency detection, peak detection, isolation and so on. The field of *polyphonic music transcription* serves as a guide to deriving a mathematical model and methodology. A tremendous amount of work has been done in the area of automatic transcription but sadly, there is no magic equation and the various approaches come down to their individual trade offs [6, 18, 21, 22, 28–30, 34, 38].

Unfortunately, (or fortunately depending on how you look at it) we must switch focus rapidly in our research because we draw from so many disciplines at once. We turn our attention now to the inspiration of musician and author *Michal Levy* [24] who suffers herself from a condition known as Synesthesia. In her beautiful, procedurally generated animations one can see choreographed imagery that mirrors the tempo and flow of a song. Michal Levy constructed several animations to include the title *"Giant Steps"* designed to intuitively externalize her condition. Another famous contributor to music visuals is the composer and computer scientist *Stephen Malinowski*

[1] A series of energy impulses extracted from an analog signal (covered in Section 2.2).

who in the 1980's constructed a simplistic but effective visualizer which leverages encoded MIDI information to create injective animations. Strangely enough Malinowski was inspired by his experiments with animated graphical scores in 1974 after taking (LSD) and listening to Bach [26]. This is not only interesting but substantial because according to Psychological research (LSD) may induce synesthesia [36].

2.1.1 A Little Music Please?

It is well known that music is underpinned by a geometric structure [1, 42, 43]. For our research it is important to digest a small amount of music theory, especially when it comes to jargon. Engineers and scientists are known for the compulsion to name everything and musicians, as it turns out, are no different. One of the most commonly used terms is the term *interval*. An interval is nothing more than the distance between any two notes and should be no stranger to mathematicians as its meaning is consistent in music theory. However, musicians created confusing labels for each of the non-negative integers up to and including 12. A scientist could go insane trying to mnemonically associate the labels since the numeric value in the label has little to do with the actual interval. Table 2.1 describes the intervals and their names. Notice that seventh is actually a step of 10 or 11, and sixth is a step of 8 or 9. Wow!

Anyway, we must be aware of this nomenclature as it is vital to understanding much of the psychological research and music theory research regarding tonality. We turn now to the fundamental inspiration for our thesis, the work of the great music theorist and mathematician Dimitri Tymzcko. To be fair to Dr. Tymzcko, in his own words he states "I am not a mathematician, [43]" however his work in music geometry contradicts such a claim. Tymoczko's work is the glue that holds our suspicions in place. In the book "A Geometry of Music", Tymzcko describes a scientific model for the behavior of music. His claims, many substantiated and some not, strongly suggests an objective characterization of melody and harmony,

Table 2.1: Music Intervals [25, 43]

Step	Name
0	unison
1	minor second
2	major second
3	minor third
4	major third
5	prefect fourth
6	diminished fifth
7	perfect fifth
8	minor sixth
9	major sixth
10	minor seventh
11	major seventh
12	octave

which boils down to five fundamental features: *Conjunct Melodic Motion*, *Acoustic Consonance*, *Harmonic Consistency*, *Limited Macroharmony* and *Centricity*. Each of these terms is sophisticated and difficult to understand without a background in music theory. We will attempt to define these terms intuitively while hopefully doing justice to Tymzcko's work. **Conjunct Melodic Motion** means that some harmony does not differ in its interval between notes by too large of an amount. This is substantiated by the fact that changes in frequency, which are too small, are undetectable and changes in frequency, which are too large are offensive [43]. **Acoustic Consonance** is the term used to describe that consonant harmonies are preferred to dissonant harmonies and usually appear at "stable" points in the song. **Harmonic Consistency** is perhaps the most important to us as this suggest a sequence of tones whose geometric structure is similar within some frame of a sound. **Limited Macroharmony** can be thought of as the external distance of a passage of music before a noticeable transition occurs. **Centricity** should be comfortable to most mathematicians and engineers as it describes an inertial reference frame or

centroid to the sequence of music. Applying the law of large numbers to music theory, one can almost envision centricity as the expected value of the music. Figure 2.1 provides one interpretation of the five features, excluding consonance. Each frame represents a macroharmony and within each frame are two chords (represented by red dots). Notice that the second chord in both frames are a linear combination of the first (scaled equally).

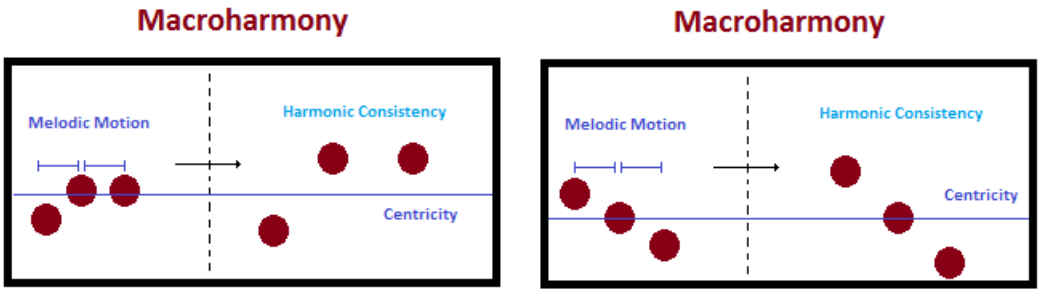

Figure 2.1: The Five Features of Music

We will leverage the ideas of Tymoczko in Section 4.4 where we attempt to garner a sense of behavior from a sequence of characterized tones in a time series. For now we continue onward by discussing the different models for music characterization. It is important to note that transcription is the primary source of our mathematical model but we are not attempting to transcribe music here. To bound the scope of this thesis we will be less concerned with the specific instrument, timbre or music notation than we are with the raw tones and chords present in any given second of a time series. Expanding upon both time fidelity and instrument identification may be part of future work. There are several models that are used when attacking the problem of music transcription, of them the most popular have been *a) Bottom Up* where the data flow goes from raw time to characterized notes *b) Top Down* where one begins with *a posteriori* knowledge of the underlying signal *c) Connectionst*, which acts like a human brain (or neural network) divided into cells each a primitive unit

that processes in parallel and attempts to detect links *d) Blackboard Systems*, which is a very popular and sophisticated system that allows for a scheduled opportunistic environment that has forward-esque flow of logic with a feedback mechanism based upon new information [22] [21]. It is the *Bottom Up* approach we chose to leverage, largely due to the fact that we intend to process raw sound *a priori*, which is to say, independent of *a posteriori* knowledge. It is at this time we transition our discussion toward an area of study dedicated to the brain and perception; we speak of course of Psychology and though it is a small portion of our thesis, it is of major influence.

2.2 Signal Basics

Our research depends upon the behavior of a sound wave. Sound waves travel through the atmosphere and generally range from 25Hz to 25Khz [6, 38] or in other words 25 cycles per second to 25 thousand cycles per second. As we age the range of human hearing decreases because our ear fibers become brittle over time and can no longer sense changes at such a high rate [38]. The human ear perceives sound by the changes in pressure generated by the frequency on both the up and down cycle of the wave [38, 42]. The speed at which that pressure changes *(e.g., the frequency)* is the way in which a brain interprets information as sound. The faster the change *(higher frequency)* the higher the pitch and vice versa. The relationship so described provides a conduit to decomposing the raw information into its basic parts and in turn algorithmically interpreting and processing information about sound. Most users generally listen to music in the form of a Compact Disc, MP3 Player, or from a television or other stereo source; all of these systems use an encoding scheme called ***PCM***. *PCM* stands for *P*ulse *C*ode *M*odulation and is the preferred means of transmitting and storing digital sound information electronically [2, 39]. Figure 2.2 illustrates a simple time domain signal.

If one observes the red line as a measurement of how intense a sound is recorded over a period of time (going from left to right on our graph) then one can gain a good

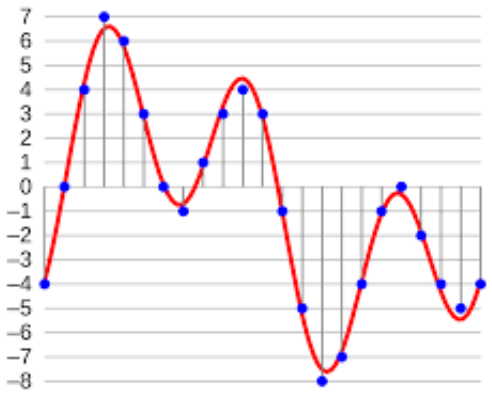

Figure 2.2: (PCM) Time Series Graph

idea of how a sound wave is received. In Figure 2.2 the small blue dots are discrete points identified along the curve, in this case a 3 Hz analog wave. These *Sample* points are where a microprocessor system, such as an analog to digital converter, would measure the sound wave height and store it for use. The number of times sound is sampled determines how accurately the digital copy represents the real sound. The *Shannon-Nyquist Theorem* [38] states that to accurately represent a signal in digital form one must sample at least two times the maximum frequency. A common sample rate found in mp3 files is 44.1Khz. Since normal human hearing tends to run between 20Hz and 20Khz [38] this makes for a good sample rate because by the Nyquist rule we have $1/2 \cdot 44100 \approx 22Khz$ being the maximum audible frequency. Such an encoding provides us with a well defined discretization of an analog sound wave. We now have enough information to break down the original sound.

2.3 Synesthesia

The psychological condition known as *synesthesia* involves what is known as a cross modality of senses [36], where the so-called *Synesthete* experiences a (usually) involuntary overlap between some combination of hearing, sight and taste. For example one may quite literally taste color, smell sound or more importantly *see sound*.

We do not offer a rigorous study in Psychology, moreover we intend to leverage elements of research, particularity in the visualization of sound, as a road map toward what may be a more scientific approach to visualizing music. Some compelling results lend credence to our thesis that there exists a universal interpretation of sound. The core of our conjecture is the idea that one can see the physical "shape" of music and experience visuals in a culturally neutral fashion. The research, as it turns out, seems to support such an idea. The work of Cytowic and Wood in the 1970's suggests a relationship between synesthetes and so-called *normals* (those without synesthesia) [36, 37]. Their research demonstrates a likely connection between synesthetes and normals, which suggests that colors are intuitively associated with certain sounds. Zellner et al. suggests that a version called *weak synesthesia* is experienced by most people as opposed to *strong synesthesia* only experienced by Synesthetes [16]. In addition to color, Synesthetes typically visualize a shape, which is referred to as a *photism* [37]. The term photism is used often in the psychological circles and refers to the geometries and color seen by synesthetes when hearing a particular frequency or melody. A photism is defined as "a hallucinated patch of light" [3]. Amongst the research of synesthetes, a photism is used to describe the stimuli when presented with a tone, chord or complex melody.

A note about the author

The concept of photism can be difficult to explain. When I was young I used to experience a stunningly choreographed array of color and shapes whenever I would hear any of my favorite music. Most commonly I would experience a vortex of spinning gradients that changed in brightness and hue synchronized to the tempo. Sometimes these vortices would change homomorphically into other polytopes. The experience was involuntary and intense for several years but began to slowly fade as I aged. I had not been convinced I had experienced Synesthesia until I read the works of Carol Bergfeld Mills et al. [5] where the exact conditions I experienced were reported by others. At the start of our research I could not help but feel unsurprised when learning which colors would commonly be chosen to represent various frequencies. I was equally unsurprised when presented with evidence of which shapes were most commonly selected. Now, I understand why. Although, I no longer possess an involuntary response, I do often purposefully visualize similar photisms when hearing a pleasing macroharmony.

As it turns out the idea of visualizing sound is not new, in the 1930's Otto Ortmann mapped out several charts attempting to define ranges and tones and their corresponding color leveraging the work of Erasmus Darwin (1790) and Isaac Newton (1704) who both predicted the existence of a pitch color scale [32,36]. We focus much of our research on the association of tone and frequency that we focus much of our research. *Colored hearing* as it is called, is believed to be the most common [36] and most often presents when played some tone that lasts for more than 3 seconds [5]. Throughout much of the published works on colored hearing there is a common theme, that is to say, without prior knowledge a substantial portion of synesthetes experi-

ence a common scale of color ranging from 26% to 82% concurrence and photisms whose size and scale s as high as 98% concurrence between synesthetes [5, 8, 16, 31]. Conveniently, the generalized experience in color and frequency can be heuristically mapped to a simple algorithm. From the works of Konstantina Orlandatou [31] we have the ranges of 0 - 50Hz as mostly black, 50 - 700Hz as mostly white, and 700Hz to 3Khz is mostly yellow. Orlandatou also notes that there is a clear association between pure tone and singular coloring and vice versa. We see from Lawrence Marks that there is a quantifiable relationship between amplitude and dimension of a photism [27]. It has also been observed that noise is generally achromatic [2] [5, 31]. We also see in multiple studies [16, 31] that pure tones are often seen as yellow and red whereas a sawtooth tone is seen as green or brown. This can be associated with the harmonic steps in a melody. Finally, we see that there is a *mood* associated with some coloring, which may present a challenge if not for the work of Dimitri Tymoczko (Section 2.1) wherein the overall behavior of a macroharmony can be decoded with the use of some music geometry. It is from these works of Psychology and Synesthesia that we derive the following tables, which will act as our algorithm guide going forward. The contents of Tables 2.2 and 2.3 have been constructed by consolidating the works of [5, 8, 16, 19, 27, 31, 32, 36, 37]. These mappings represent direct psychological experiments with our own interpolations and statistical averaging of Synesthete responses.

[2] Being without color or black/white.

Table 2.2: (Synesthesia) Note Color Association

Base Frequency Map $N(a)$

Note	RGB Color
C/C#	blue
D/D#	red
E	yellow
F/F#	brown
G/G#	green
A/A#	green
B	black

Table 2.3: (Synesthesia Tone Color Mapping)

Additive Frequency Map F(a)

Frequency Range	Note	Color
0 - 50 (Hz)	All	black
50 - 700 (Hz)	All	white
700 - 22 (KHz)	All	yellow

Pattern Map P(a)

Frequency Pattern	Color Effect
Harmonic Stair Step	green

2.3.1 A Colored Hearing Theorem

We will now derive a few theorems allowing us to extract a numeric color value from a frequency.

Theorem 1 (The Roswell Theorem)**.** *There exists a proportionate mapping such that an increase in noise takes any color toward the gray color scale.*

Proof: Recall the linear interpolation equation $(1-t)P_1+tP_2$ with P_1, P_2 vectors in $\mathbb{R}^m$. The value of t ranges from 0 to 1 being a percentage of the total distance between vectors P_1 and P_2. Let P_1 be an R^3 vector of the form (r, g, b) where the values r, g, b range from 0 to 255 and $P_2 = (128, 128, 128)$ (the gray color). Let $\boldsymbol{\sigma}$ represent the total noise level of a signal. If we compute $t = \frac{|\sigma|}{max(|\sigma|)}$ then we have a ratio of 0 to 1 over the range of the noise. If we substitute t back into the interpolation equation $(1-t)P_1 + t(128, 128, 128)$ we have a linear interpolation, which transitions any color toward gray as noise increases. ■

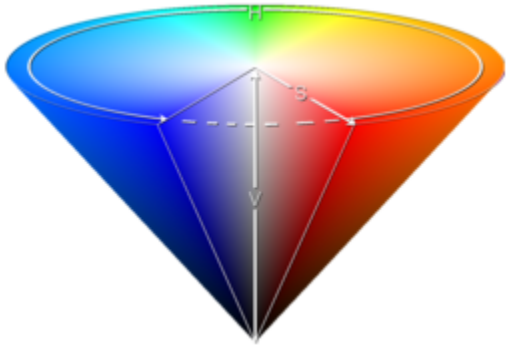

Figure 2.3: Hue, Saturation, Brightness Color Scale [23]

Theorem 2 (The Stripes Theorem)**.** *Define the binary operators* $\diamond$ *and* $\circ$ *to be* additive color[3] *and* color intensity *operations, respectively. Any musical frequency* $\boldsymbol{a}$ *can be mapped to a color consistent with "Colored Hearing" using the equation*

$$\begin{aligned} \boldsymbol{(r, g, b)} &= (1-t)\left[H(a) \circ (N(a) \diamond F(a))\right] + t(128, 128, 128) \\ & \qquad\qquad where \\ t &= \frac{|\sigma|}{max(|\sigma|)} \end{aligned} \tag{2.1}$$

and $\boldsymbol{\sigma}$ is the noise level.

Proof: Let $H(a)$ be a mapping of the fundamental frequency to an HSV[4] color from Table 2.2 and $F(a)$ be a mapping from any frequency range to an HSV color (h, s, v) in Table 2.3. When we mix colors with

$$(h, s, v) \diamond (j, t, w) = ((h+j)/2 \ (mod\, 360), (s+t)/2 \ (mod\, 1+\epsilon), (v+w)/2 \ (mod\, 1+\epsilon)) \tag{2.2}$$

according to the surface of the cone in Figure 2.3, then scale the result of (2.2) according to harmonic $k = H(a)$ such that

$$k \circ (h, s, v) = (h, (1-k)/max(k), k/max(k)). \tag{2.3}$$

The result of 2.3 is a color combination of the observed note-color and general frequency-color matching that is brighter for higher frequencies and darker for lower. Assuming that $\circ$ maps to an RGB[5] value, we plug the result of (2.3) into Theorem 1 and the output is a color that simulates a Synesthetes *Colored Hearing* response. ■

Figures 2.4,2.5 and 2.6 illustrate color samples using the "Stripes Theorem", which were generated over the frequency range 16 to 22.05 (Khz). In Figure 2.6 we see a darkened version of the blended colors over the ranges of 16 to 31 (Hz). We

[4]HSV or HSB is a hue, saturation, brightness color scale where $0 \leq saturation, brightness \leq 1$ and $0 \leq hue \leq 360$ [15].

[5]RGB is a red, green, blue color scale used commonly in computer graphics where $0 \leq r, g, b \leq 255$ [35].

Figure 2.4: Tigger Stripes, 16 to 31 (Hz) (No Noise)

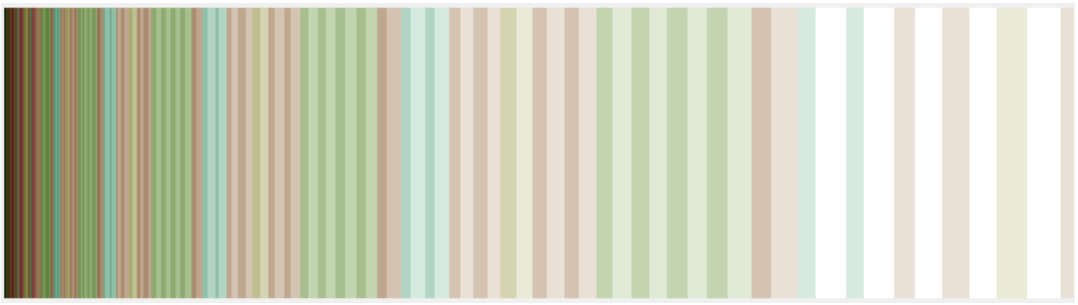

Figure 2.5: Tigger Stripes, 16 to 22.05 (Khz) (No Noise)

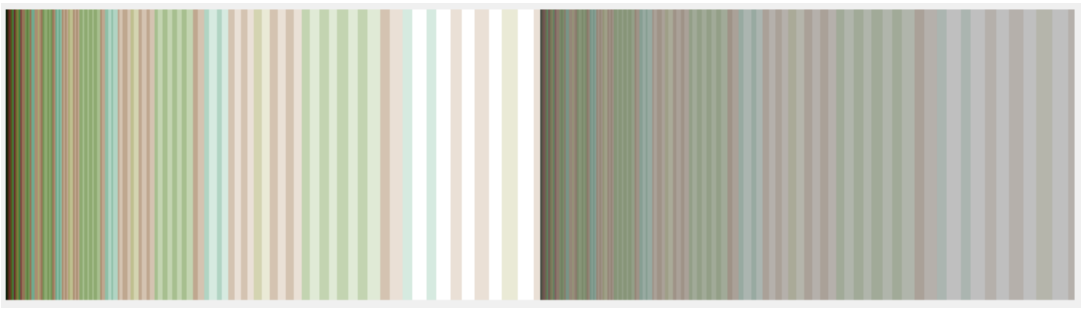

Figure 2.6: Tigger Stripes, 16 to 22.05 (Khz), No Noise (left), 50% Noise (right)

then generate colors for $\approx$ 22 (Khz) shown in Figure 2.5 where we notice a clear transition from darker to lighter tones, which is consistent with our proof. Finally, in Figure 2.6 we add 50% noise to the frequency spectrum (right side), which causes a clear transition toward the gray scale consistent with Theorem 1.

3. Proof of Concept

3.1 Discovery and Approach

It requires so many technical elements to solve the problem of visualizing sound, so much so that the question of how to begin poses a significant challenge. We must contend with the logistics of parsing and interpreting PCM, reading from various devices, the means with which we can display graphical information and all of the structures and utilities necessary to calculate and render. As with any journey, we must take a first step and what better step to take than a simple end-to-end prototype that can read a sound file and generate some sort of mapped graphical imagery using only the time domain information. This prototype, or proof of concept if you will, allows us to learn a few things about our data. We construct a sound processing framework in the Microsoft Windows environment using C# .Net. This language was chosen primarily for it's high performance capability and flexible syntax which allows us to leverage operator overloading to more easily handle mathematical structures. We are also able to utilize the raw *struct* syntax that provides on stack memory allocation as opposed to dynamic memory that provides significant performance decrease when dealing with random access. The *SoundBus Framework*, as we call it, is a processing framework that utilizes an animation plug-in system where each animation plug-in acts as an interface that can receive both sound data messages and requests to render their current content. This allows us to experiment with different algorithms without losing any previous work. We employ the *Microsoft XNA* as a graphics application program interface *API* that allows us to speak to the graphics processing unit (GPU). The *NAudio* sound processing package that acts an API connecting our system to the sound input device, frees us from having to implement a device driver or decompression algorithm. Our implementation is capable of seamlessly processing raw pulse data from a raw MP3, .wav file, or direct microphone input. At first, we intended on providing a one-to-one mapping of impulse to graphical element. There

are several challenges when dealing with an attempt to synchronize the visualization of sound and imagery. At 44K impulses per second, a 100Hz refresh and drawing one image per frame the backlog grows arithmetically as $\beta(t) = (44100)t + (-10000)t$. Even if we increase to 100 images per frame after 10 seconds we have a backlog of $\beta(10) = 341000$ images to be drawn. As a consequence we can never keep up with the sound that is playing in real-time without creating tremendous clutter on screen or simply rendering an image the user never actually sees. Thus, we abandon the one-to-one rasterization[1] but not the one-to-one calculation. Our framework provides a stream of sound information to an interface designed to process individual samples at a time. Simultaneously there exists another interface mechanism that is called on a 30FPS interval to refresh the computer display. In order to keep the different streams synchronized we employ a timer system that calculates the current temporal backlog and flushes data to the graphics calculation. The processing implementation system then chews off individual data blocks and continuously incorporates the data into a set of running parameters for our geometric figures. At any given time, the interface is asked to render itself in its *current state*. The end result is that we receive a fluid animation that generally mirrors the pace of the sound and neither gets too far behind, nor too far ahead if reading from a sound file.

3.1.1 A Time Domain Experiment

The images shown in the upcoming results section are created using the following approach: we primarily take advantage of statistical characteristics of a sound wave in the time domain. The implementation receives the PCM over time and parameterizes a set of simple dihedral geometric figures whose edges are drawn in stages over time based upon initial parameters. We begin with the set $\mathbf{X} = \{x \mid x \in \mathbb{Z}, -2^k \leq x \leq 2^k\}$ that describes the amplitude data. To minimize clutter we limit the total number of animations on screen at any time to $n \in \mathbb{Z}^+$. In our time sampling we deal with

[1] A term in computer graphics that describes converting memory elements to screen pixels [35].

hundreds of thousands of samples in just a few seconds. We must limit the elements generated so as to not overburden the graphics system and our CPU. Experimentally, we chose a hard limit of 200 items which we shall adjust later as needed. Let $\mathbf{S} = \{s_k \,|\, 0 < k < n\}$ represent our parameterized animation elements such that $s_k = (\theta, \phi, \alpha, \vec{P}, r, m, v)$ with $0 < \theta \leq \alpha \leq \phi < 2x\pi$ with θ, ϕ being the starting and ending angle, α the current angle of rotation, $\vec{P} \in \mathbb{Z}^3$ the centroid, and r, m are the radius and color respectively (the color here is the integer form of a bitwise combined RGB value). Finally, v represents the step that determines the number of vertices in the geometric figure. We then define a set of mappings $f_p : \mathbf{X} \rightarrow \mathbf{S}$ that map time parameters to an animation element, $f_d : \mathbf{S} \rightarrow \mathbb{R}^3$, that maps an animation element to the display (a $2 \times 2 \times 2$ bounded region in $\mathbb{R}^3$) and $f_n : \mathbb{R} \rightarrow \mathbb{Z}^+$ where $f_n(x) = -1 + |x/\max(x)| \cdot 2$, that normalizes data to screen bounds. Table 3.1 contains the initialization parameters that maps elements in $\mathbf{X}$ to elements in $\mathbf{S}$.

As a new amplitude is received, an initial state that represents the signal at that time is constructed by way of the parameters defined in Table 3.1. As mentioned in Section 3.1 there are two key stages consisting of a **paint interval** and **time step**. When a paint interval occurs the elements in $\mathbf{S}$ are rendered as a curve extrapolated from the set of rotations $R = \{i \,|\, i \in \mathbb{Z}^+, i_0 = \theta, i_{k+1} = i_k + (\phi - \theta)/60, i \leq \alpha\}$. The shape is then mapped to the display with a simple linear transformation $f_d(s_k, i) = (r\,cos(i) + P(s_k)_x, r\,sin(i) + P(s_k)_y, P(s_k)_z)$; this essentially connects the vertices of some partially, or fully formed regular polygon on screen over time. Depending on the current rotational perspective we also paint a *disc* at the centroid of a geometric figure whose size is determined by $r - (\alpha - \theta)/r\phi$ where $r, \phi \neq 0$ and that produces a visual *singularity* type effect. Simultaneously, at each time step we increment the current angle α of each element s_k by $\alpha = \alpha + v$. An animation reaches its life's end when $\alpha \geq \phi$ at which time it is purged. The size, color and vertex count of an animation element is a direct representation of the shape of the pulse waveform at

Table 3.1: Time Domain Initial Parameterizations

Parameter	**Value**	Description
x		Current amplitude
x_{k-1}		Previous amplitude
σ		Signal to Noise Ratio
g		Gain
θ	$\lvert x/255 * 2\pi \rvert$	Starting angle
ϕ	$\theta + \lvert x_{k-1}/255 * 2\pi \rvert$	Ending angle
α	θ	Current angle
r	$x/(max(x_k) * 2)$	Radius
v	$(\phi - \theta)/30$	Rotational velocity
$ColorRed$	$f_n(x)\ (mod\, 255)$	RGB Red Value
$ColorGreen$	$x_{k-1}\ (mod\, 255)$	RGB Green Value
$ColorBlue$	$\sigma * 255\ (mod\, 255)$	RGB Blue Value
P_x	$f_n(rand() + 2g - 1)$	Centroid X
P_y	$f_n(rand() + 2g - 1)$	Centroid Y
P_z	$f_n(rand() + 4g - 1)$	Centroid Z

the time it is created. As an additional visual element we also set a gradient tone for the background based upon the current signal strength where $Background\, RGB = (0, 0, f_n((E[X]_k - E[X]_{k-1})/E[X]_k))\, (mod\, 128)$ with $E[X]$ being the expected value. Note that our display rotates the entire view matrix about the y-axis (assuming y points north) very slowly in a counterclockwise direction. The rotation angle is associated with an average of a subset consisting of recent amplitudes in ratio to the maximum.

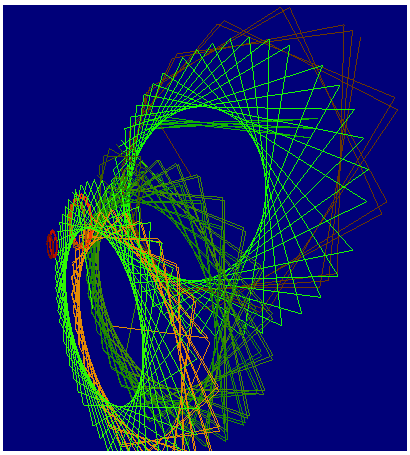

Figure 3.1: Example Set of Parameterized Geometric Figures

3.1.2 Initial Results

Our application was run against a handful of music files which, in this case came from symphony music downloaded from the internet. To use the application one simply selects the input source, in this case an MP3 file, and then presses play.

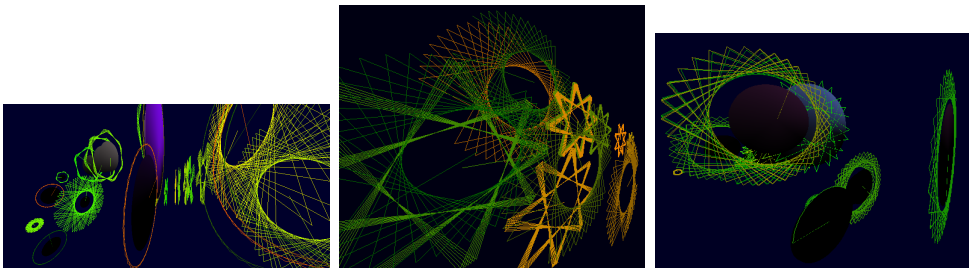

Figure 3.2: Symphony - Sound Dogs

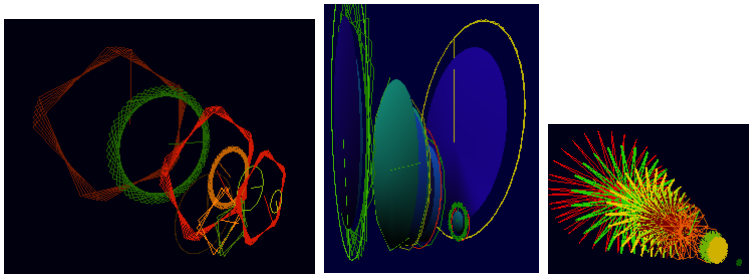

Figure 3.3: Symphony (Beethoven 12'th Symphony,Beethoven Violin Sonata,Schubert's Moment Musical)

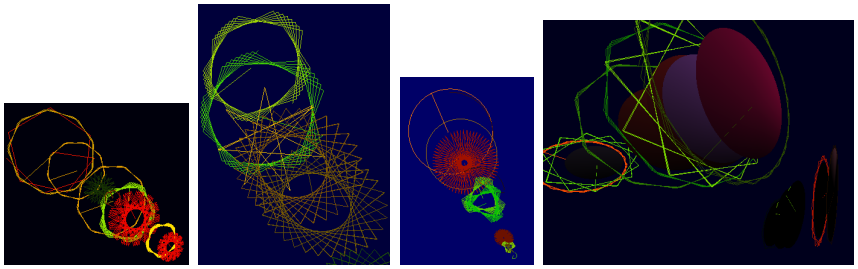

Figure 3.4: Techno Electronica - (Termite Serenity,Nao Tokui,She nebula,Termite Neurology)

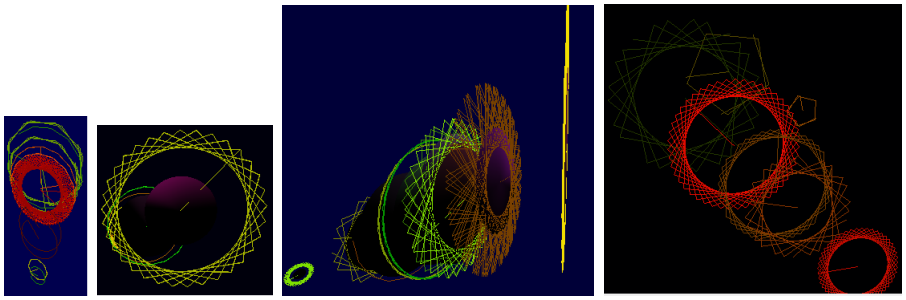

Figure 3.5: Various Composers - (Debussay Clair de Lune,Mozart Eine Kleine,Mozart Sonata,Chopin Etude)

Figures 3.2,3.3,3.4 and 3.5 illustrate screen captures taken from our application while playing the specified music. If one observes the images, particularly Schubert's Sonata from Figure 3.3 one can see the formation of a conic structure composed of successive geometric figures. We conjecture we are seeing the physical shape of the waveform over some duration. This behavior manifests itself throughout most songs. Though mathematically we have shown that our visualizations are in fact a direct result of the *shape* and *behavior* of the raw time series it is very difficult to qualify that we are seeing any behavior that mirrors the underlying tonality or melody. The imagery is captivating and interspersed with brief moments of melodic mimicry and synchronization but it is not a sufficient demonstration of our thesis. We did however accomplish the initial goal of constructing a software framework for moving forward. Portions of the processing source code can be found in appendices A.0.10, A.0.11 and A.0.12.

3.1.3 Approach

It is time to leverage our research and continue trying to extract what makes the music *behave* the way it does. Thus, we may attempt to incorporate such behavior into our visuals. In order to do this, we devise a plan for our overall model illustrated as a road map in Figure 3.6. The plan in Figure 3.6 allows us to handle each stage of the transformation with the level of rigor we deem necessary or possible within the scope of this thesis. We offer a modular approach to the processing pipeline insofar as we break our algorithm into several parts. Each stage yields a clear output that acts as input to the next stage. We do this with the knowledge that we can both improve each stage independently and add tuning and parameter adjustment for known shortcomings. We do not abandon a cohesive mathematical model across modules, nor do we assume that inputs/outputs are mutually exclusive. We merely strive to break apart the challenges and allow independent research and improvement. The idea being, when one component of the algorithm pipeline improves, others do

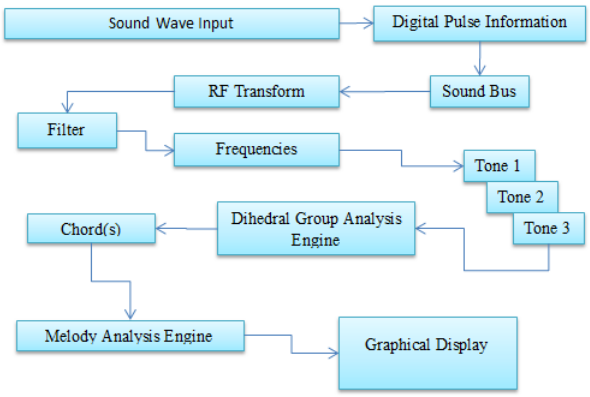

Figure 3.6: Research approach

as well. Garbage in, garbage out and vice versa, so to speak. The real beauty is that we can find partial solutions to an algorithm and still move on to the next algorithm. This is very important because there are no perfect solutions with detection and characterization, discussed in Sections 4.1 and 4.2.

3.1.4 Animation Time Budget

As noted earlier, we must be able to process data in realtime and this must be done quickly enough so as to not to lag too far behind the music. We propose an initial *time budget* consisting of 1.5 seconds from time sample to visualization. The time budget acts as a guide on how to tune performance and accuracy of an algorithm. For example, we may sacrifice accuracy in a calculation that takes minutes and is done rarely but is very slow. The concept of the time budget is not new; the Microsoft XNA graphics environment provides a game clock where paths in the code can make choices to skip actions on the current cycle, or double up if lots of time is available. Our time budget, shown in Table 3.2, is a guesstimate based upon experimentation from Section 3.1 and research regarding the Fourier transform. The *time budget* is a soft requirement that allows us to think of the entire processing pipeline as a cohesive function so we do not lose sight of where we are in time. It is easy to drift off when focusing on one area and forget that it is a small aggregate of a larger calculation.

Ultimately, we expect to see large deviations from our initial guess but we must begin somewhere.

Table 3.2: Animation Time Budget

Step	Allotted Time (milliseconds)
Frequency Detection	600
Note Characterization	100
Chord Characterization	200
Geometry Analysis	100
Melody Analysis	100
Graphics Processing/Rendering	200
Total	1.3 Seconds

4. Research and Development

4.1 Fourier Analysis and Frequency Detection

To detect, isolate and characterize the contents of a music signal we must be able extract the tones from a time series. This implies that we require frequency information from the time data. The basis for this assumption is from a simple concept proposed by Bello [22] that implies we must be able to determine two major properties of any time distribution. First, we identify the *significant* frequencies within the sample distribution, specifically those associated with musical tones. Second, we identify the event time of each frequency. Bello proposes three key values *pitch, onset, and duration* which, we will utilize implicitly later on. For the time being, we constrain our analysis to a one second interval and use the Discrete Time Fourier Transform *(DTFT)* [6, 38] to extract frequency information from a time series. We choose our time interval to be one second which, is based upon Synesthesia research stating that sub-second intervals of tone are unlikely to invoke a response and longer intervals do not change the response [33]. This also decreases our mathematical complexity and implementation complexity. The model for extracting tones that includes all frequencies of the form

$$f_n(a) = a \cdot (2)^n \text{ where } a \in \mathbb{R}^+,\ n \in \mathbb{Z}^+ \tag{4.1}$$

is every value of n that produces a harmonic of the fundamental frequency a. A list of all the fundamental frequencies is shown in Table 4.1. We refer to the fundamental for a particular note as f_0 but we must introduce a few new forms of notation. The following is more computer science than mathematical and is how we will reference a "named note" as a function $f_0('A') = 27.5, f_1('A\#') = 58.28$ and so on. We will also use the more common notation for a harmonic frequency commonly found in music texts which is $< Note >< Harmonic >$, (eg: $A4$ which implies $27.5 \cdot 2^4$) [1].

[1] $A_4 = f_4('A')$ is the fourth harmonic of the set of frequencies $f_k('A') = \{\, 440 \cdot (2)^{k/12} \mid -48 \leq k \leq 39 \,\}$ [22] [38].

Table 4.1: Fundamental Frequencies [7, 38]

Note	Frequency (Hz)	Wavelength
C_0	16.35	2109.89
$C\#_0$	17.32	1991.47
D_0	18.35	1879.69
$D\#_0$	19.45	1774.20
E_0	20.60	1674.62
F_0	21.83	1580.63
$F\#_0$	23.12	1491.91
G_0	24.50	1408.18
$G\#_0$	25.96	1329.14
A_0	27.50	1254.55
$A\#_0$	29.14	1184.13
B_0	30.87	1117.67

Definition 1 (Music Fundamental Set)**.** *Let F be the set of fundamental frequencies where $F = \{f_0('C'), f_0('C\#'), ..., f_0('B')\}$ as shown in Table 4.1.*

Definition 2 (Musical Harmonic Set)**.** *Let M be the set of all musical harmonic frequencies where $M = \{f_0('C') \cdot 2^k, f_0('C\#') \cdot 2^k, ..., f_0('B') \cdot 2^k, \forall\, k \geq 0\}$.*

Using our definitions[2] and knowing the specific ranges we are targeting, we can design our analysis in such a way as to extract a specific subset of the time domain. We will be dealing with unmodulated data *(ie: there is no carrier wave or shift keying)* and assume a maximum sample rate of 44,100 Hz [2] [38].

4.1.1 Understanding the DFT

Mathematically speaking, what is a wave? To answer this we must first examine the cosine function. Back in Figure 2.2 we saw a 3 (Hz) wave. If we inspect the cosine function over some period of time t you have the mapping $f : \mathbb{R} \to \mathbb{R}$ where $x = f(t) = a\,cos(t)$ with $\{x| -1 \leq x \leq 1\}$ and the peak and trough of the function are a maximum and minimum of a, illustrated in Figure 4.1 with $a = 1$. Remember the cosine function rises and falls symmetrically over the abscissa and peak to peak measurements are congruent. Now examine Figure 4.2, which illustrates a very simple example of how a series of cosine values being added can produce a new waveform.

Suppose that instead of $f(t) = cos(t)$ you had $t = 2k\pi n$ or $f(n) = cos(2k\pi n)$ where k is some constant *(the frequency)* and n is some real number that iterates over all possible values of the function for that desired frequency. For the purposes of Figure 4.2, k would be equal to unity thus yielding $x = cos(2 \cdot 1 \cdot \pi)$. Interestingly enough we have added three waves together, each wave being a single cycle/frequency of amplitudes 2,3,5 respectively. In general 2,3 and 5 could be measurements of voltage, power or some other ratio of change between two values such as Decibels

[2]To assist the reader we try to maintain consistency and definition mnemonic. Notice that F is the fundamental frequency set and M is all musical frequencies. Whenever possible, we stick with consistent variable names for frequency, iterants, sets, etc...

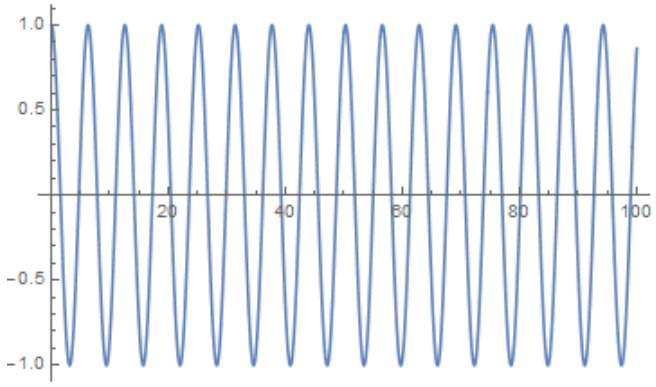

Figure 4.1: Cosine Function $x = cos(t)$

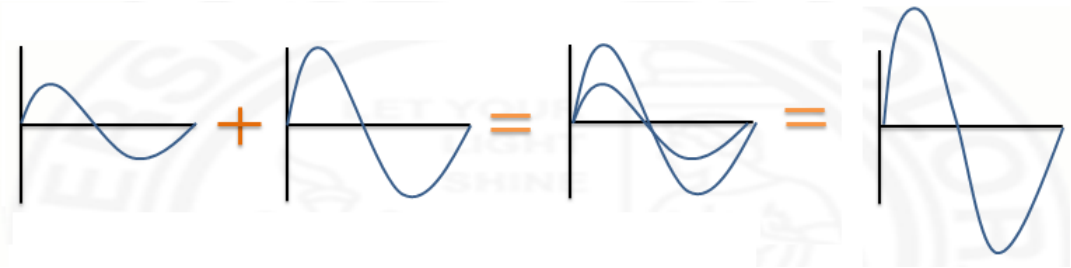

Figure 4.2: $x = 2\,cos(2\pi t) + 3\,cos(2\pi t) = 5\,cos(2\pi t)$

where $a = 10\,log_{10}(x/\delta x)$ [38]. For the moment we ignore intensity and assume our power levels are at unit. Figure 4.3 is a pictographic representation of a random wave. Note that this wave is not accurate in terms of its structure but for intuition only. This image depicts how two waves of different frequencies are able to be added together to form a new wave just as in the previous example. Using Figure 4.2 as a starting point one might see that we can reverse engineer the *presence* of any original wave within another. How do we do this? By dividing out each frequency we perceive to be *present* in the original time series as illustrated in Figure 4.4.

Thus, take the set $\mathbf{X} = \{cos(2\pi n) + cos(4\pi n),\, 0 \leq n \leq \infty\}$. How would we determine if a sinusoid of frequency 2 (Hz) lives within this wave? Essentially, we would want to ask the wave at every point how much the function $f(n) = cos(2 \cdot 2\pi n)$

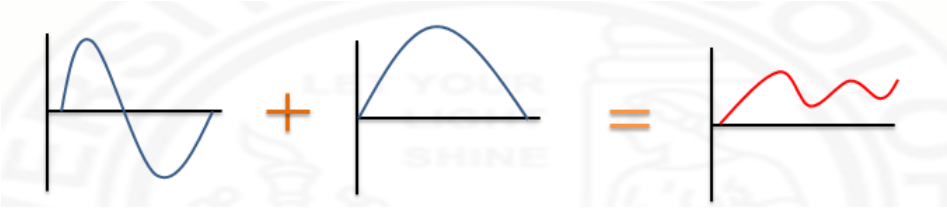

Figure 4.3: Random Waveform of More Than One Frequency Note: Not an accurate graph

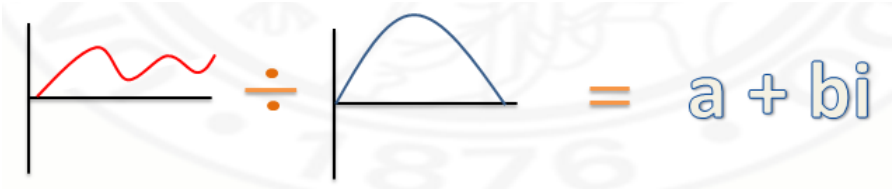

Figure 4.4: Dividing Frequencies, $a + bi \in \mathbb{C}$

matches in what amounts to a cross correlation between the time series and the complex function. Remember, the complex function iterates over all possible values of a wave at a particular frequency so comparing it to the original time series $\mathbf{X}$ at the same intervals results in a correlation response. However, we do not subtract, we divide by our search wave at each interval and sum the results. This generates what is in effect a correlation value whose magnitude is a measure of the presence of our desired frequency. For example if we were looking for 2 (Hz) signals in the original wave we could perform the following

$$Presence_{(2Hz)} = \left| \sum_{all\, n} \frac{\mathbf{X}_n}{cos(2 \cdot 2\pi n)} \right| \tag{4.2}$$

Recall Euler's formula $e^{ix} = cos(x) + i\, sin(x)$ where i is the imaginary unit. Take note that if we were to divide by the cosine only we'd have our original Equation 4.2. However Figure 4.5 depicts the complex plane and the behavior of the Eulier's forumula. The function $Z = Re^{-i2k\pi n}$ traces a curve in the complex plane, which results in a perfect circle about the origin of radius R. Within the complex plane itself, it is

not apparent how k affects the wave thus we extend along a 4th axis with $k = f_0 t$. We then use an orthogonal projection onto some affine plane parallel with the imaginary axis. One can see that the image under the projection is the 2D sinusoidal waveform similar to Equation 4.2, the difference being that now we have encoded phase information into the result of our division.

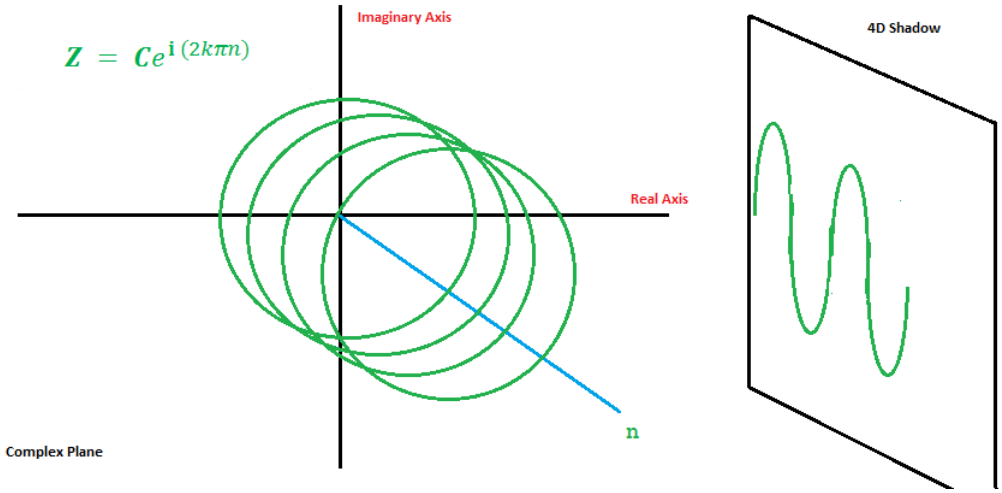

Figure 4.5: Geometry of Complex Frequency, C = Constant Amplitude/Radius, k = Frequency, n = Real valued coefficient.

From this model it follows naturally to substitute complex division for our real division, and the resulting mapping is known as the Discrete Fourier Transform or *(DFT)*. Then

$$\overline{X_k} = \sum_{n=0}^{N-1} X_n \, e^{-i \frac{2\pi k n}{N}} \tag{4.3}$$

where $\overline{X_k}$ is a set of complex numbers whose magnitudes signify the *presence* of a frequency k in the original signal. Additional references for the various flavors of the DFT and FFT can be found in [4, 6, 11, 17, 38, 40, 41].

4.1.2 A Parallelized Fourier Transform

The Fast Fourier Transform is predicated on a number theoretic approach to factorizing the DFT into smaller pieces. In general this allows us to improve the computational complexity by extracting factors from an inner summand and performing those multiplications a single time thus reducing our flops from $O(N^2)$ to $2Nlog_2\,N$ [17]. Although there are several libraries that have full implementations of the FFT we attempt to derive our own mathematical model here. In part, allowing us to explore the behavior of the computation, but also we assume we may need to fine tune the computation addressing specifics of our detection algorithm. To maximize performance on modern hardware we take advantage of this idea on two fronts: *a)* by removing factors from the inner summand and *b)* creating a computation that can be parallelized. When dealing with the Fourier transform it is common to define a constant $W_N = e^{-i\frac{2\pi}{N}}$ which, allows us to deal with a compact form of the DFT as $\overline{X_k} = \sum_{n=0}^{N-1} X_n\, W_N^{nk}$. Taking ideas from the work by James Cooley and John Tukey [17] and others [4, 10, 41] we derive a simpler version of the Fast Fourier Transform (FFT) in Theorem 3.

Theorem 3 (A Parallelizable Fast Fourier Transform)**.**

$$\begin{aligned} f_b = f(a, N_1, N_2, k) &= \sum_{a=0}^{N_2-1} X_{a+bN_2} W_N^{ak} \\ \overline{X_k} &= \frac{1}{N} \sum_{b=0}^{N_1-1} f_b \cdot W_N^{bN_2k} \end{aligned} \tag{4.4}$$

Proof: Let $N, N_1, N_2 \in \mathbb{Z}$. We obtain N_1, N_2 by factoring N such that $N = N_1 \cdot N_2$. Using the n, nth roots of unity W_N^{nk} let $n = a + b\,N_2$ with integers a, b. By the division algorithm, we have mappings in $a = \{0, 1, 2, ..., N_2 - 1\}$ and $b =$

$\{0, 1, 2, ..., N_1 - 1\}$. The key is to notice that a is cyclic in N_2 and b is cyclic in N_1 [17]. This is essentially the same as a two dimensional expansion of the single dimensioned value n. Observe, when we break apart the DFT using our two dimensional index scheme we have

$$\overline{X_k} = \sum_{b=0}^{N_1-1} \sum_{a=0}^{N_2-1} X_{a+bN_2} W_N^{(a+bN_2)k}.$$

If we apply some basic algebra and factor we end up with
$W_N^{(a+bN_2)k} = W_N^{ak} W_N^{bN_2k}$ which, ultimately yeilds

$$\overline{X_k} = \sum_{b=0}^{N_1-1} \sum_{a=0}^{N_2-1} X_{a+bN_2} W_N^{ak} W_N^{bN_2k}.$$

Notice, the only changes in the inner summand are a and k which, means we can extract a function

$$f_b = f(a, N_1, N_2, k) = \sum_{a=0}^{N_2-1} X_{a+bN_2} W_N^{ak} \tag{4.5}$$

If we plug our function back in, and normalize the result by the total sample count $1/N$ we get

$$\overline{X_k} = \frac{1}{N} \sum_{b=0}^{N_1-1} f_b \cdot W_N^{bN_2k} \tag{4.6}$$

■

Our derived FFT is approximately $7N^2 + 7N_1$ with this factorization and is computationally more complex than the standard DFT. If we let $e^{-i(2\pi/Nx_1x_2)}$ approximate 6 flops, assuming Euler's Equation $e^{ix} = cos(x) + i\, sin(x)$ [48]. The original DFT ranges over N elements and N frequencies that implies $(6+1)N \cdot N = 7N^2$. Observe that Equation 4.5 is approximately $6+1$ flops ranging over N_2 and $6+1$ flops ranging over the outer summand N_1 which yields $N_1(N_2 7) + 7N_1$ from the outer summand. Given N search frequencies we have $N(N_1(7N_2)) + 7N_1 \approx 7N^2 + 7N_1$ total flops.

Although the increase in flops seems to be a computational loss we have an an overall gain as we have extracted a memory contiguous inner summand with no external dependencies. This calculation allows us to provide equal sized contiguous blocks of RAM to independent threads (providing we factor evenly). Contiguous blocks of independent data per thread minimizes cache contention and decreases overall overhead associated with locking and context switching [20]. Algorithm 1 approaches the parallelization of Equation 4.6 by creating a process/thread to execute N_2 multiplications and additions each and returning the partial summand that we then add to a synchronized accumulation variable.

Algorithm 1 (Parallel Fourier Transform)**.**

Shared searchRf $\leftarrow$ {Search Frequencies}, X $\leftarrow$ {PCM}, c $\leftarrow$ 0

Private N $\leftarrow$ Length(X), $\{N_1,N_2\} \leftarrow$ Factor(N)

For i $\leftarrow$ 0 To Length(searchRf)

c $\leftarrow$ 0

k $\leftarrow$ searchRf[i]

For a $\leftarrow$ 0 To $N_1 - 1$

Parallel

Private t $\leftarrow$ 0

Consume c **into** t

Produce c $\leftarrow$ t + $f_b * e^{-i\frac{2\pi a N_2 k}{N}}$

End Parallel

Next

Barrier

$X[\lfloor k \rfloor] \leftarrow c * 1/N$

Next

4.1.3 A Synthetic Test

Before we empirically analyze the performance, we will verify our parallelized fourier model will correctly extract a frequency distribution. First, we construct a signal *synthesizer* utility that generates an artificial signal at the desired frequency values $F = \{f_1, f_2, \cdots, f_k\}$. Then, using Equation 4.1 and Equation 4.7 we can

interlace those frequencies into a time series. Given $a \in \mathbb{R}$, $k, n \in \mathbb{Z}$

$$X_n = \sum_{all\ k} a * cos(\frac{2\pi F_k * n}{N}) \tag{4.7}$$

where a is the amplitude, n is current sample and N is the sample rate. Figure 4.6 shows our target signal, fabricated with only a single fundamental $f_0('A')$ and 12 harmonics. The source code for the Fourier transform and Synthesizer can be found in Appendix A.0.15 and A.0.14.

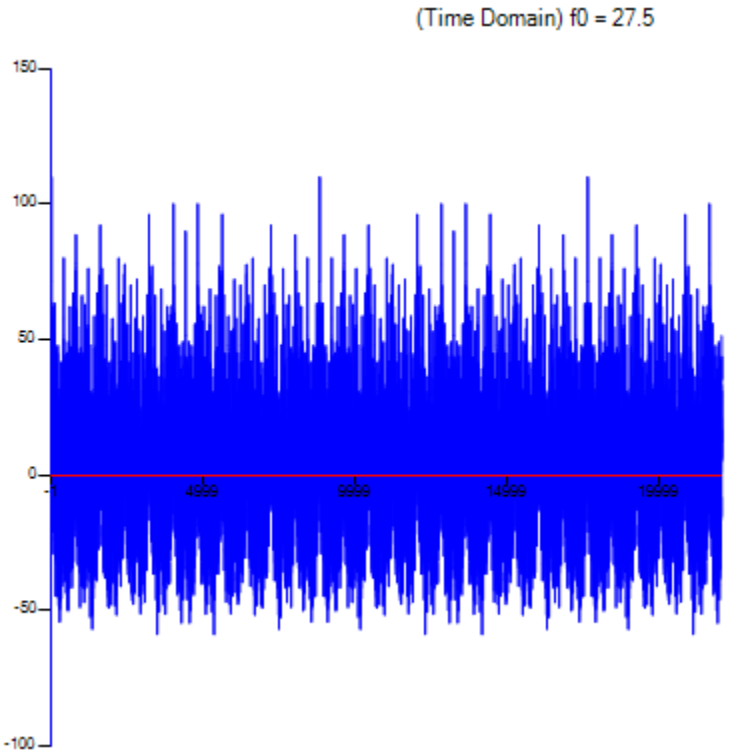

Figure 4.6: Synthesized Signal at $27.5 \cdot 2^k$ $\{0 \leq k \leq 11\}$

We compare our new Fourier model in Equation 4.6 with the standard DFT using the signal in Figure 4.6 to determine if the results are equivalent. Figure 4.7 shows amplitude measurements for each known frequency are the same between the DFT

and the new model. Confident that our algorithms are producing the same peaks we now contrast their overall performance. We executed each algorithm 5 times and averaged their speed using a high precision software timer. The results are shown in Table 4.2.

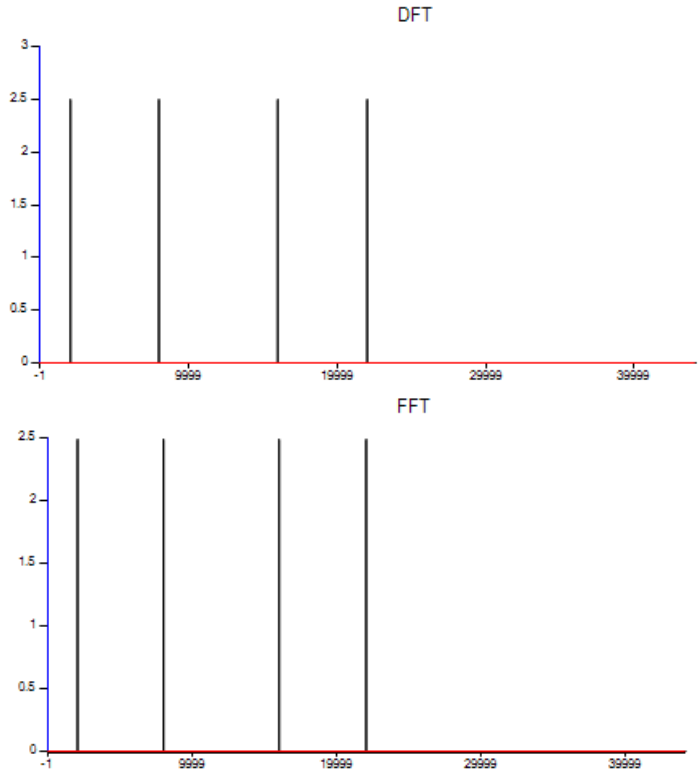

Figure 4.7: Comparison of Standard DFT to Parallelized FFT

Table 4.2: DFT vs FFT Performance

Algorithm	Time (ms)
Parallelized FFT	72.2
Standard DFT	192.4

Table 4.2 was computed with $f_0('A')$ and 12 Harmonics on an *Intel 3.5Ghz i7* with 16GB SDRAM on Windows 7 Professional 64bit. We compute the overall performance with $Speedup = \frac{T_1}{T_p}$ [20], which yields $192.4/72.2 \approx 2.66$ thus our algorithm is $\approx 166\%$

faster than the standard DFT. This will be helpful when trying to extract a large number of frequencies in a timely fashion.

Before further analysis, we must include another calculation in our Fourier model in order to deal with issues that arise during the sub-sampling of a time distribution. We apply a technique known as *windowing*, which minimizes the effect called *spectral leakage* [6,22,30]. The leakage is in response to the transform being applied to a time series in partial chunks, in our case 1 second intervals. When we sample a portion of a time series it has been observed that high frequency aliasing may appear at the seams of the sample space [6, 30]. To compensate, the window function allows us to partially repair this leakage by smoothing the transition. This technique takes many possible forms and we have chosen the *Hanning or Hann* window. Figure 4.8 illustrates the graph of a *Hanning* window, which is essentially the haversine function. Although there are many different types of windowing functions this one is known to be effective in musical transcription [6, 30]. Equation 4.8 shows the calculation adjusted for our factorization and when we incorporate $\omega(a, b)$ into Equation 4.5 we have Equation 4.9.

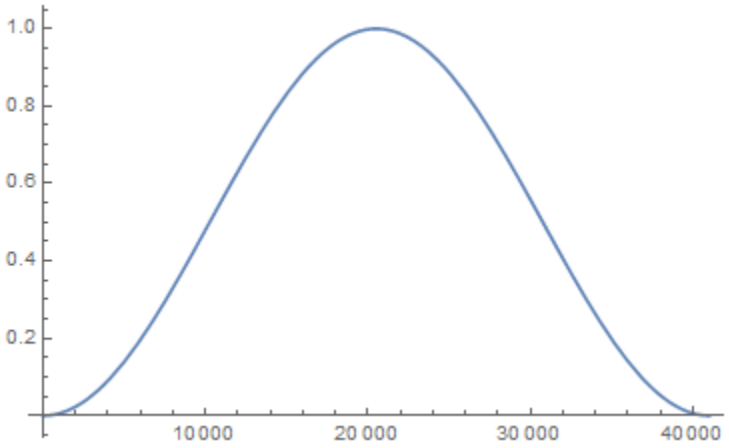

Figure 4.8: Hanning Window

$$\omega(a,b) = \frac{1}{2}\left(1 - cos\left(\frac{2\pi(a+bN_2)}{N-1}\right)\right) \tag{4.8}$$

$$f_{\omega,a} = \sum_{a=0}^{N_2-1}\left(\frac{1}{2}\left(1 - cos\left(\frac{2\pi(a+bN_2)}{N-1}\right)\right)\right) X_{a+bN_2} W_N^{ak}$$
$$\overline{X_k} = \sum_{b=0}^{N_1-1} f_{\omega,b} \cdot W_N^{bN_2k} \tag{4.9}$$

It follows that we must determine if our algorithm will be effective against the *full spectrum* of tones $f_0('C') \cdot 2^k$ through $f_0('B') \cdot 2^k$. We include the *Hanning* window in all the following calculations. In order to verify accuracy and performance we ran Equation 4.9 against a set of synthetic signals, which included a minimum of 12 harmonic steps for every frequency up to and including the Nyquest. The results are shown in Table 4.3.

We are approaching a key threshold in our frequency detection performance. Recall from Section 3.1.4 that our time budget allows for 500 ms transform time. Instead of jumping to a very complicated musical score, we will continue to incrementally "complexify" our signal and evaluate our analysis as we go. You can see another signal pattern (Figure 4.9) and the resultant FFT values for the specified frequencies in Table 4.4.

An interesting result is that the DFT and FFT now differ slightly in their real and imaginary components. Initially we assumed this was caused by numerical precision issues in our algorithm but as it turns out we were correct in our assumption. In Section 4.2 we discovered certain frequency ranges were not being detected properly and this was due to the fact that we had mixed the frequency search variable type between integer and floating point causing the $e^{-i\frac{2\pi((int)n)k}{N}}$ computation to return rounded results. After correcting the problem the output is exact between the DFT and our parallelized transform.

Table 4.3: FFT Performance Extracting the Full Spectrum of Tones

Sample Rate	Num Freq's	Ave Time (ms)
8000	96	156.2
16000	108	219.2
22050	120	422.6
32000	120	452
37800	132	569.2
44056	132	624
44100	132	640.4
47250	132	696
48000	132	698.2
50000	132	719.8
50400	132	724.4
88200	144	1334

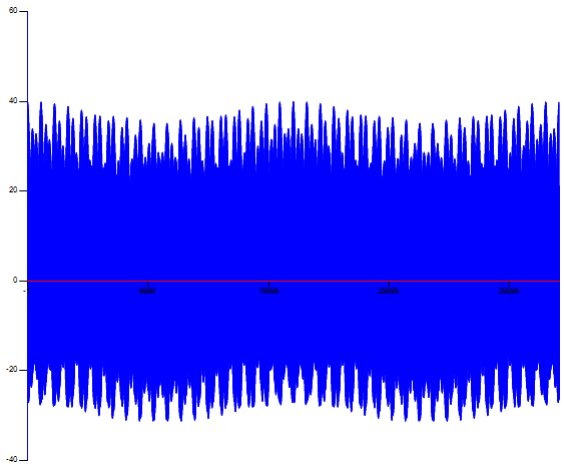

Figure 4.9: Synthesized Signal {2049,8000,16000,22031} (Hz) Over 1 Second

4.2 Tone Detection and Characterization

Table 4.4: FFT Versus DFT Accuracy

Rf (Hz)	DFT 2-norm	FFT 2-norm
2049	2.500003	2.488039
8000	2.499994	2.488039
16000	2.499994	2.488036
22031	2.49998	2.488338

With our raw frequency detection technique in place we must determine an efficient and accurate way of characterizing the musical signal in terms of its musical notes *(eg: A, A#, C, etc.)*. We do not expect to find a perfect technique, especially when it comes to complex scores, noise and/or percussion instruments generating anti-tonal sound. We also have to contend with discrete samples of a continuous wave. It is understood that we will encounter frequency aliasing, frequency loss and detection ambiguity [6, 22, 29, 30, 34, 38]. The following detection algorithm began by borrowing a technique proposed by Jehan [18] which leverages a scaled, homogeneous, mean squared error of the spectral graph. During our experimentation we we able to simplify the response detection for our purposes of only extracting dominant tones.

4.2.1 A Detector Predicate

Provided with an amplitude in relation to a particular frequency we must be able to estimate the musical note, however there is an additional challenge. Recall the DFT requires us to impose the mathematical *floor* of real values, thus accommodating situations where a frequency such as 27.5 (Hz) exists in the original signal but our analysis produces a peak at 27 (Hz) or potentially 26 through 28 (Hz) or some similar combination. This could be troublesome going forward however all our frequencies differ by at least .5 (Hz), which means we can truncate each frequency to the nearest integer without too much trouble. Given a complex valued frequency distribution X and $a \in \mathbb{C}$ we can detect a frequency peak using Equation 4.10.

$$S(a) = \frac{|a|}{\max(|\overline{X}|)}$$

$$D(a) = \begin{cases} True & S(a) > t \\ False & S(a) < t \end{cases} \tag{4.10}$$

The strength of a signal is defined by the expected value of the distribution as opposed to *noise*, which is defined to be the standard deviation [38]. This concept is generally used to detect a signal within the noise floor [6, 38]. We approach it somewhat differently here because our methodology is to only go after frequencies that we want. We do not transform the entire space of frequencies from 1 to $1/2 \cdot \text{Nyquist}$ thus our impulse response is constrained to the domain of the ≈ 144 frequencies we care about. We then match the impulse response of the known frequencies against the maximum impulse of known frequencies; we then trigger a boolean response of *True* whenever a frequency exceeds the average signal strength by some tolerance $0 \leq t \leq 1$, which can be tuned later.

4.2.2 Musical Note Characterization

We require another tool in our toolbox for determining which note is played after a successful detection. After some experimentation we observe that Equation 4.1 "bounces" the fundamental frequency over a step function toward its harmonic. From this observation we construct the following theorem.

Theorem 4 (The "Tigger" Theorem)**.** *Every element in M can be mapped to a distinct integer of the form*

$$T(f) = \lfloor 100 \cdot \left(\frac{f}{2^{\lfloor log_2(f) \rfloor}} - 1 \right) \rfloor \tag{4.11}$$

Proof: Given the equation

$$t = \frac{f}{2^{\lfloor log_2(f) \rfloor}}. \tag{4.12}$$

Since $\frac{f}{2^{\lfloor log_2(f) \rfloor}} = \frac{f}{2^{k+c}}$ for some integer c it implies $k + c = \lfloor log_2(f) \rfloor$, which implies (4.12) maps any frequency to its fundamental divided by another c factors of 2. If we compute the values for each fundamental frequency we get $\boldsymbol{T} = \{1.021875, 1.0825, 1.146875, 1.215625, 1.2875, 1.364375, 1.445, 1.53125, 1.6225, 1.71875, 1.82125, 1.929375\}$, ordering the set ascending by the corresponding fundamental. When we subtract 1 from each element we get $\{.021875, .0825, .146875, .215625, .2875, .364375, .445, .53125, .6225, .71875, .82125, .929375\}$. When we multiply by 100 and take the mathematical floor we have the set $\boldsymbol{Q} = \{2, 8, 14, 21, 28, 36, 44, 53, 62, 71, 82, 92\}$. By inspection every element in $\boldsymbol{Q}$ is unique therefore (4.12) maps every element in M to a distinct integer. ■

Definition 3 (Tigger Harmonic Set)**.** *Given the Tigger Theorem mapping $T : M \rightarrow \mathbb{Z}$, define $\boldsymbol{\tau}$ to be the set of all Tigger Harmonic values such that $\boldsymbol{\tau} = \{T(m),\ m \in M\}$.*

Now that we have a detection and characterization technique we devise an algorithm, which allows us to process any audio input stream and guess the notes within that stream. We construct our algorithm as follows, Equation 4.6 extracts a portion of a frequency distribution of a musical time series, Equation 4.10 allows us to detect the presence of a frequency and Theorem 4 allows us to map that detected frequency to a musical note. The entire approach is defined in Algorithm 2 and Listing 4.1 and

4.2 shows the main structures and methods used by the algorithm.

Algorithm 2 (Note Characterizer)**.**

Private X ← { Sample Inputs }, SearchRf ← { f_1, f_2, ... }, DetectedNotes ← ()

Private N ← Length(SearchRf), i ← 0

X ← FFT(X, SearchRf)

For k ← 0 To N-1

    If D($X[k]$) **Then**

        DetectedNotes[i] ← T(k)

        $i \leftarrow i + 1$

    End If

Next

Return DetectedNotes

Listing 4.1: Characterization Structures

```
// Immutable structure that holds info about
// a fundamental tone.
struct MusicalNote {
    // Note enumeration value
    public eMusicNote myNote;
    // The fundamental Rf for this note
    public float myRf;
    // The tigger mapped value.
    public float myTiggerRf;
    }
// Mutable structure that holds information
// about an observable tone.
struct MusicalDetect {
    // The Starting second the note was seen.
    public long myTimeOn;
    // The observed frequency
    public float myRf;
    // The observed amplitude
    public float myAmp;
    // The note detected.
    public MusicalNote myNote;
    // The duration the tone was played in milliseconds
    public float myDuration;
    // The harmonic of the fundamental in myNote.
    public int myHarmonic;
    }
```

Listing 4.2: Characterization Methods

```
// Determines the fundamental note from any frequency.
//  f := The frequency for which to guess the note.
MusicalNote Characterize(float f);

// Determines which notes exist within the given time
// series starting from offset.
//              X := Set of time series samples (PCM)
//    sampleRate := The samples per second in X
//        offset := Where in X to begin guessing.
MusicalDetect[] GuessNotes(Z2[] X, int sampleRate, int offset);
```

It is wise at this point to experiment with the algorithm and determine its effectiveness. Figure 4.10 depicts the time plot of a more complicated (albeit slightly unrealistic) signal generated with the synthesizer. The signal is composed of three tones *A, C, G#* at values of (27.5,1000,2),(17.32, 5000,1), (25.96,7000,2) frequency, power and time (seconds) respectively. We execute Algorithm 2 on this signal and demonstrate the results in Table 4.5.

Table 4.5: Initial Note Guessing Results

Time	Found	Expected	Amp
0	A	A	248.0868
1	A	A	248.0868
2	C	C	1242.433
3	G	G#	1736.93
4	G	G#	1741.317

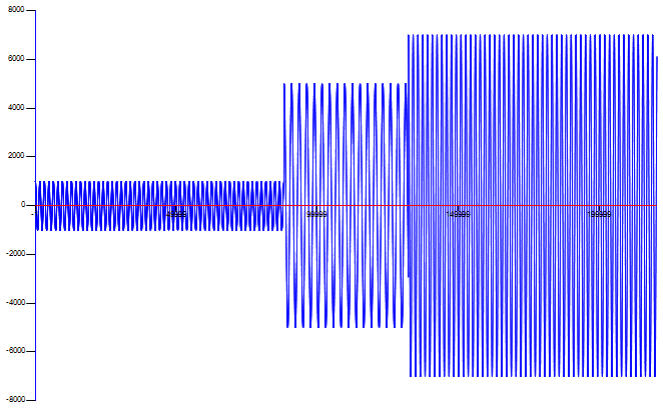

Figure 4.10: Synthesized Tones @ 44.1Khz, {A,C,G#} Over 5 Seconds

The results of our detection and characterization are promising, however there are obviously not exact. Recall that the FFT must work with discrete frequency values thus providing a challenge to the accuracy of the Tigger Theorem. Figure 4.11 illustrates a comparison of the rational valued harmonics versus the discrete harmonics in the 44,100 Nyquist range. The vertical axis is the image under the Tigger mapping and the horizontal axis is the frequency (denoted by the corresponding musical note).

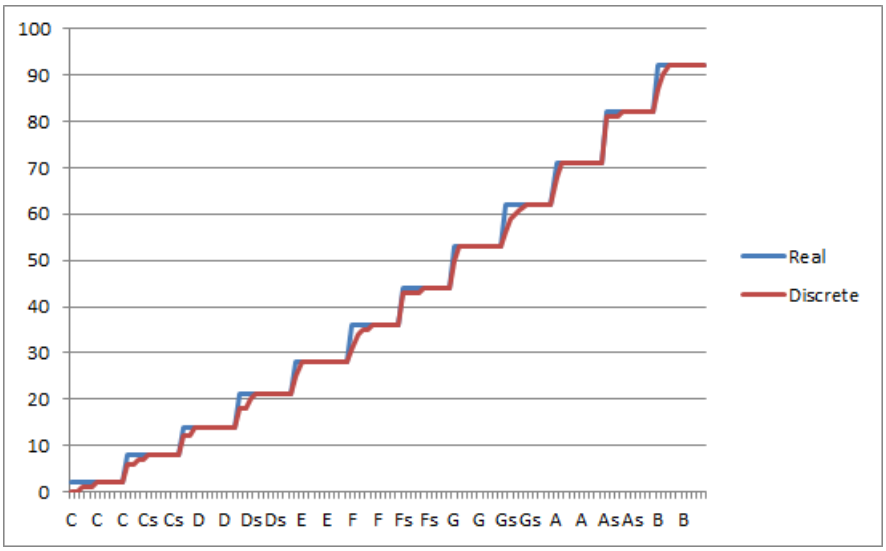

Figure 4.11: Comparison of Real vs Discrete *Tigger Theorem* (Note vs $T(a)$)

The results are surprisingly similar, however we notice clear deviations that are most prominent at the fundamentals. We shall attempt to remedy this error by adding a new characterization step. We require a new set of integer values directly mapped to the fundamental frequencies and another set directly mapped to the harmonics.

Definition 4 (Fundamental Integer Set)**.** *Define F_I to be the set of integers bijective to F such that $F_I = \{\lfloor F \rfloor\}$.*

Definition 5 (Harmonic Integer Set)**.** *Define M_I to be the set of integers bijective to M such that $M_I = \{\lfloor M \rfloor\}$.*

Theorem 5 (The Discrete "Tigger" Theorem)**.**
Any fundamental frequency can be mapped to a unique integer.

Proof: We compute by exhaustion the set $F_I = \{16,17,18,19,20,21,23,24,25,27,29,30\}$. By observation all elements of F_I are unique. ■

Theorems 4 and 5 allow us to derive a slightly more accurate approach in the numerical environment. Given any music frequency value $a \in M_I$ we attempt our characterization by determining if the frequency is an element of F_I and using the corresponding note if a match is found. If no match is found in the fundamentals we compute the "Tigger Harmonic" and search for a match in the "Tigger Harmonic Set" $\boldsymbol{\tau}$ so as to minimize $|T(a) - \boldsymbol{\tau}_k|$. We execute this new technique on the same signal and display the results in Table 4.6. The results are perfectly accurate for our very simple signal. The source code for the note detection/characterization can be found in Appendix A.0.7 in the function **GuessNotes()**.

Table 4.6: Final Note Guessing Results

Time	Found	Expected	Amp
0	A	A	248.0868
1	A	A	248.0868
2	C	C	1242.433
3	G#	G#	1736.93
4	G#	G#	1741.317

4.2.3 Rigorous Characterization Analysis

We have successfully fine tuned our model to perfectly detect the sequence $A, C, G\#$ so we proceed to perform more rigorous analysis. We start by synthesizing a sound that appends every harmonic for every note into a time series and execute Algorithm 2 on the time series. Figure 4.12 clearly illustrates the results.

The algorithm appears to break down in the upper harmonics, which has been observed by bello [22] and others. A high frequency detection method is supplied by Bello, which could possibly be applied at a later date. For the time being, the 11'th and 12'th harmonics remain $\approx 45\%$ accurate while the rest of the spectrum is 100% accurate. As a final test we synthesize signals, which consist of *random tones* at *random harmonics* for *random intervals*, creating a more complex sound of approximately 5 - 14 seconds. Figure 4.13 illustrates one example of the random sound used in this analysis. We keep track of the *ground truth* information and use it to measure the expected output against the guessing algorithm. At first we generate only a few signals and run them through the detector. The output can be seen in Table 4.7.

Table 4.7: Accuracy of Random Signals

Time	Found	Expected	Amp
0	G	G	983.947
1	G	G	983.947
2	F#	F#	2487.615
3	A#	A#	963.4857
4	G#	G#	2223.319
5	G#	G#	2223.319
6	F	F	1707.491
7	D	D	1086.08
8	G	G	1142.743
9	G	G	2419.794
10	D	D	2255.06
11	B	B	2432.167
12	B	B	2432.167

Time	Found	Expected	Amp
0	F	F	1927.708
1	F	F	1927.708
2	E	E	1222.492
3	E	E	1222.492

Time	Found	Expected	Amp
0	F	F	2219.488
1	C	C	2048.204
2	C#	C#	2557.707
3	G	B	10.61324
4	A	A	2507.143
5	A	A	2507.143
6	C	C	1920.38
7	G	G	2351.047
8	G	G	2351.047
9	Undetected	F#	5470.793
10	Undetected	F#	5470.793

Time	Found	Expected	Amp
0	A#	A#	1157.51
1	F#	F#	972.2834
2	D#	D#	564.4082
3	A#	A#	513.2103
4	C#	C#	1556.932

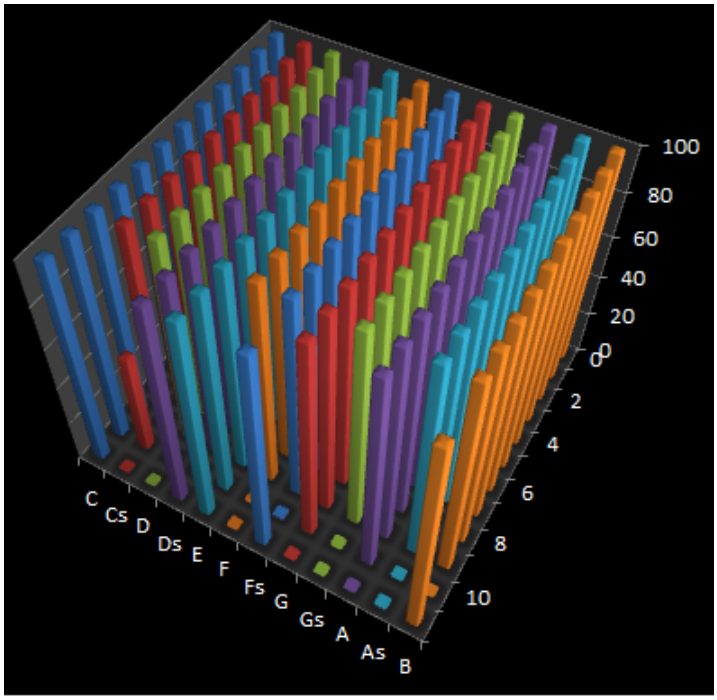

Figure 4.12: Accuracy Plot (Note vs Harmonic vs Percent Accuracy (0 - 100%))

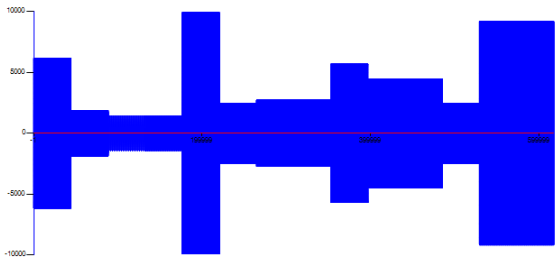

Figure 4.13: Synthesized Random Sounds @ 44.1Khz

As expected, the guesses are fairly accurate but we we need more information to determine how accurate, so we proceed to automate a very rigorous test over hundreds of random signals. We tally the results into three areas, Figure 4.14 shows the success as a percentage of accuracy over time, Table 4.8 demonstrates the overall performance and Table 4.9 shows us the tones and ranges that were undetected.

The execution time is well below what was expected since $\approx$ 8 seconds of time only takes $\approx$ 1.6 seconds to guess the tones. This leaves us a substantial amount of

Table 4.8: Detection & Characterization Results (Initial Metrics)

Total Runs	Ave Sample Time	Ave Guess Time	Overall Accuracy
200	8.435 (s)	1675.49 (ms)	86.60344%

Table 4.9: Detection & Characterization Results (Undetected)

Undetected Note	Times	Frequency	Amplitude
C	17	35471.36 - 35471.36 (Hz)	2557.409 - 9612.205
C	41	28160 - 56320 (Hz)	1008.208 - 10325.92
C	17	26583.04 - 53166.08 (Hz)	1367.82 - 10405.38
C	22	22353.92 - 44707.84 (Hz)	1348.063 - 10926.15
C	21	23674.88 - 47349.76 (Hz)	1836.552 - 10948.71
C	30	31610.88 - 63221.76 (Hz)	1796.666 - 10658.98
C	10	33484.8 - 33484.8 (Hz)	1145.468 - 7371.489
C	29	25088 - 50176 (Hz)	1211.848 - 10403.89
C	24	29839.36 - 59678.72 (Hz)	1711.161 - 10979.81
C	9	37580.8 - 37580.8 (Hz)	3221.58 - 10981.38
C	10	42188.8 - 42188.8 (Hz)	1913.405 - 10745.18
C	8	39833.6 - 39833.6 (Hz)	1283.639 - 10054.6

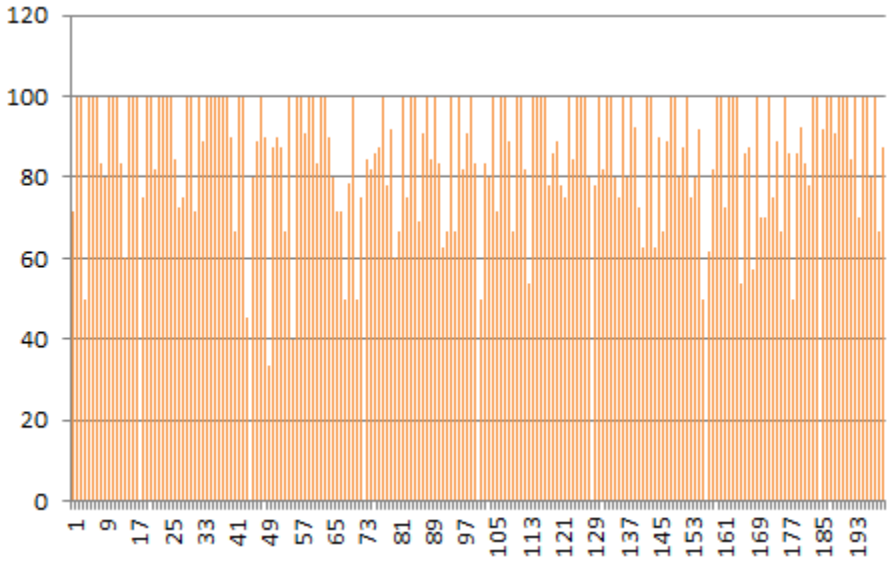

Figure 4.14: Detection & Characterization Results (Run vs % Accuracy)

processing slack for melody analysis. However, the accuracy of $\approx 86\%$ is fairly good but does not seem to follow with our 100% accuracy in Table 4.5. After careful observation of Table 4.9 we discover a phenomenon. Notice that all undetected instances of C are above the Nyquist frequency of 22.050 (Khz). Recall we are sampling at 44.1 (Khz), which means that 22.050 (Khz) is the maximum detectable frequency. As it turns out our tone generator has a very simple but significant defect. At octaves above 12 it creates frequencies above the Nyquist range. We correct the issue by limiting the synthesized frequencies to the Nyquist range and re-run our analysis. The results are excellent. Table 4.10 and Figure 4.15 shows that we are now at 100% accuracy running $\approx 500\%$ times faster than *real time*. As previously stated, no algorithm will be perfect, thus we expect to see a margin of error when processing real music, but for the time being we are confident enough to move forward.

Table 4.10: Detection & Characterization Corrected Results (Final Metrics)

Total Runs	Ave Sample Time	Ave Guess Time	Overall Accuracy
200	8.855 (s)	1814.64 (ms)	100%

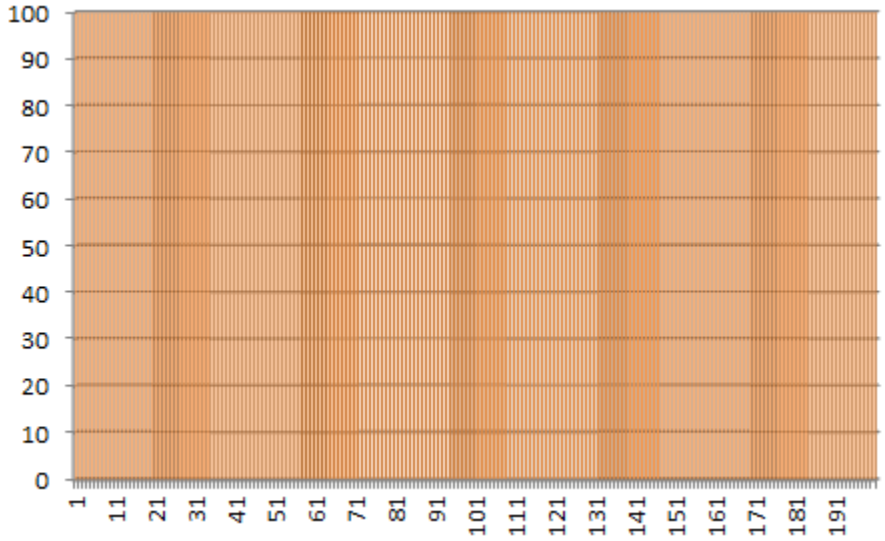

Figure 4.15: Detection & Characterization Results Corrected (Run vs % Accuracy)

4.3 Chord Detection and Characterization

A musical chord can be difficult to clearly define, especially when we diverge from western music and include the idea of inharmonic or dissonant tones. Informally, a chord is a set of notes played at the same time but to limit confusion and enhance scientific rigor we adopt Tymoczko's algebraic definition of a *musical object* that is "an ordered sequence of pitches [44]" whose elements are vertices of an element in the D_{12} group[3]. To precisely detect and characterize a chord is very difficult. Our tone characterization provides us with a set of musical notes but we must determine the starting time and duration of groups of notes which form a chord. This is especially

[3]The dihedral group on 12 vertices. Discussed in detail in Section 4.4.

tricky when dealing with sub-band[4] onset of multiple tones combined with low latency. Remember from Section 2.1 that a synesthetic response is triggered by a tone being played for at least 3 seconds. We wish to elicit this same response in normals and we conjecture that a preferred response in normals is more likely when mirroring the stimuli of synesthetes. The reason being that all humans share common neurological components in auditory and visual cognition and most people experience some kind of synesthesia [16, 37, 38]. Nevertheless, at this point one would prefer a sub-second knowledge of tone onset to aid in chord identification. For example, suppose we had a two second interval and our characterization algorithm produces the notes {A,C,G#} and {A,D,E}. We don't know at what times within the second any of the tones started only they exist somewhere within the second. Nothing prohibits an *'A'* note from playing for 25% of the first second and 50% of the next second and our detection would not specify which portion of the second it began or ended. We shall add a high fidelity time model to our list of future work and press onward with our 1 second fidelity.

4.3.1 A Chord Detection Algorithm

In order to identify a chord we employ a straightforward technique but first we must discuss some terminology. The term *residual* as used in this section, will refer to any unison tone not immediately paired with an existing chord. We use the term *minimum duration* to represent the minimum time (milliseconds) a note is played. Figure(s) 4.16, 4.18, 4.21, 4.19, and 4.17 show the Use Cases[5] for our algorithm. It should be noted that our algorithm is agnostic to time fidelity so future work will employ the same algorithm even if we adjust sub-band frequency extraction. We break our cases into *New Notes*, *Match*, *Refresh Chord*, *Residuals*, *Chord Expiration*

[4]We use this term in this context to refer to space in the sampling less than the current Nyquist sample rate.

[5]In Software Engineering a Use Case defines the concept of operation of an "actor" and the interaction with a particular interface, machine or domain.

Use Case (New Notes:)

- *Assumption:* None.

1 A set of new notes are characterized.

2 If no ***Match:*** is found ***Residuals:***.

3 If ***Match:*** is found ***Refresh Chord:***.

3 Increment step time to the maximum of all notes-onset time.

4 Run ***Chord Expiration:***.

Figure 4.16: Use Case (New Note)

and *Chord Kill*. Each case is a sequential portion of the grander algorithm with *New Note* being the entry point and *Match* being a utility for comparison of notes. We use the class structure in Listing 4.3 to manage the chord construction. We call this class the *ChordDetector* and it acts as a state machine that behaves according to the algorithms described by Use Cases 4.16, 4.18, 4.21, 4.19, and 4.17. The chord detector class accepts one to many tones in the form of a *MusicalDetect* array *(Listing 4.1)*. *ChordDetector* makes no assumptions about the incoming tones and attempts to merge items by time and duration into appropriate chords or unison tones. The source code for the chord detector can be found in Appendix A.0.9, and A.0.8.

Use Case (Match:)

- *Assumption:* Let n be any existing note and m be the incoming note, $T(\cdots)$ is the time onset and $D(\cdots)$ is the last known duration of a note.

    1 If $f_0('n') == f_0('m')$ and if $f_k('n') == f_k('m')$ for some k and if $T(n) \leq T(m) \leq T(n) + D(n)$ we have a match.

    2 Otherwise no match.

Figure 4.17: Use Case (Match)

Use Case (Refresh Chord:)

- *Assumption* Matching note found.

    1 If a matching note is found in an existing chord and increment duration by *minimum duration.*

    2 Else if a note with the same onset time is found add note to chord.

Figure 4.18: Use Case (Refresh)

Use Case (Residuals:)

- *Assumption:* No matches to the note.

  1 Start a new chord.

  2 Add note to chord.

Figure 4.19: Use Case (Residual)

Use Case (Chord Expiration:)

- *Assumption:* Let n be any existing note and $T(\cdots)$ is the time onset and $D(\cdots)$ is the last known duration of a note. *Step Time* is a counter that holds each interval of *minimum duration* that has passed.

  1 If $T(n) + D(n) <$ (Step Time) for any note, remove that note.

  2 If all notes expired *Kill Chord:*.

Figure 4.20: Use Case (Expiration)

Use Case (Kill Chord:)

- *Assumption:* All notes expired in chord.

  1 Remove the chord from the dictionary.

Figure 4.21: Use Case (Kill)

Listing 4.3: Chord Detector Class

```
// A chord structure that holds notes played at the same time
class Chord  {
    // List of notes in this chord.
    List<MusicalDetect> myNotes;
    }
// Manages a set of chords detected in real time.
class ChordDetector {
    // The current time step in units of myMinDuration.
    // incremented on each call to New Notes.
    private long myTimeStep = 0;
    // The list of currently held chords
    private Chord myChord;
    // Add new notes to be detected and added to a chord
    // or create new chords.
    public void NewNotes(MusicalDetect[] newNotes);
    // Request all currently held chord structures.
    public Chord GetChord();
    }
```

Fortunately, it is easier to verify this algorithm because the data space is smaller and the algorithm is in fact deterministic. Table 4.11 demonstrates the input series of notes manually constructed to try to fool our algorithm. Notice that from time 1 to time 2 our algorithm is not fooled and properly removes tones $A2$ and $G\#3$. It also compresses $B4$ into the same note and properly disposes of $C2$ from time 2 to time 3. If we experience issues later in our implementation we will revisit a more rigorous

Table 4.11: Chord Detector Basic Test

Time	0	1	2	3	4
Notes	C2 A2 G#3	C2 A2 G#3	C2 B4 B4	B4 A4 G#4	B4 A4 G#4
Expected	{C2,A2,G#3}	{C2,A2,G#3}	{C2, B4}	{B4,A4,G#4}	{B4,A4,G#4}
Output	{C2,A2,G#3}	{C2,A2,G#3}	{C2, B4}	{B4,A4,G#4}	{B4,A4,G#4}

test of the chord characterizer.

4.4 Melody Analysis

In this section we explore the analysis of the music as a cohesive set of chord progressions wherein a chord may consist of a single note. We treat each chord as a geometric *musical object* so before we begin the melody analysis using the five features of music in Section 2.1. Our melody mapping is based upon the notion that music has some kind of center; musical objects that move in small increments and are consonant are more pleasing and that similarly of chord structures are expected to appear often.

We must quickly mention the algebraic groups, specifically the *dihedral* group. The **dihedral** *group is the group of symmetries of a regular polygon in n vertices [12].* To bound the scope of our thesis we do not offer a rigorous education in group theory but we do offer a simple intuitive explanation of the dihedral group. Suppose you have a planar graph with four vertices. You embed the vertices in the 2D cartesian plane such that all vertices are symmetric and equidistant from (0,0). If we arbitrarily label each vertex as in Figure 4.22 we have the dihedral four group often referred to as D_4. The first image of Figure 4.22 shows position 0, which we deem the identity. The second image is a clockwise rotation about the origin of 90 degrees. If we continue to superimpose the vertices in every permutation and label each configuration, those labels combined with a binary operation form the group. It is called an algebraic

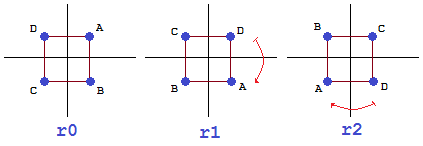

Figure 4.22: D_4 Group Example

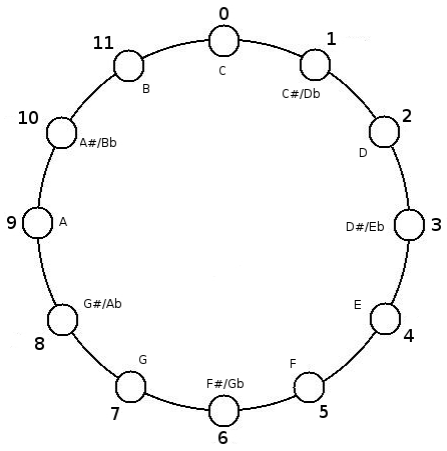

Figure 4.23: D_{12} Pitch Class

"group" because it can be shown to possess closure[6], associativity, inverse and identity of the binary operation and its elements [13]. In the case of the symmetric group we use the $\circ$ operator to denote the binary operation. For example $r0 \circ r1 = r1 = r1 \circ r0$, giving us the identity in $r0$. Similarly, $r1 \circ r1 = r2$ and $r1 \circ r1^3 = r1 \circ r1^{-1} = r0$. Because we return to the identity element $r1^3$ is the inverse of $r1$. This can be thought of as taking the value of $90\,\text{deg} + (90\,\text{deg})^3 \equiv 0\ (mod\ 360)$. One final way to envision a group inverse and identity is using the familiar real numbers. Notice that the inverse of the value 2 is $2 \cdot \frac{1}{2} = 2 \cdot 2^{-1} = 1$ whereas the identity of 2 is $2 \cdot 1 = 1 \cdot 2 = 2$. The algebraic dihedral group simply uses a new fancy label for familiar concepts such as inverses and identities. It is within these group structures that we determine the behavior of the music.

We analyze the melody according to the *Pitch class*, a term in music theory referring to the D_{12} group with musical notes shown in Figure 4.23 [45] *(throughout this section we will interchange the terms* pitch class *and* D_{12}*)*. Using the five tenets of music we construct a pattern analysis designed to capture the flow of the melody. Our *Chord Detector* can extract enough information to assign a value of the D_{12} group

[6]Closure of a group implies that the binary operation on two elements results in another element within the same group.

to each note of a chord, which forms a subgroup of the group. There are many changes to a subgraph that can occur but two key changes are known as distance preserving functions *transposition* and *inversion*. Transposition and inversion are analogous to the geometric operations of translation and reflection [46]. The smaller the translation value the harder it is to detect the change [46] and vice versa. Inversion, according to Tymoczko, has a *similarity* property wherein inversely related chords sound similar. We will use these ideas to help guide our mappings.

4.4.1 A Generalized Parameterized Visualizer

Much like our proof of concept in Section 3.1 we want to produce similar output on screen but using more rigorous musical information. Remember from Table 3.1 that the initial implementation is directly linked to the time series. In order to facilitate the use of the five features of tonality we must break the time series association, refactor our code and generate a robust and re-useable animation widget. Constructing various animation components can be very time consuming and complicated. In future work we intend to experiment with a myriad of visual behaviors but for our current scope we will re-use the current work. This means we want to be able to plug in a new animation without having to re-wire visual calculations or depend on visuals tied closely to the type of data. We created a class structure and interface designed to help us de-couple the renderings from the analysis logic. To that end, we define a new space.

Definition 6 (Value Space)**.** *A* value space *is a vector in* $\mathbb{Q}^m$ *whose elements are values ranging from 0 to 1, ordered by the importance of the value from lowest to highest.*

Definition 7 (Color Space)**.** *A* color space *is a set of integer RGB values ordered from lowest to highest importance.*

Definition 8 (Focal Space)**.** *A* focal space *is vector in* $\mathbb{Q}^m$ *whose elements are values from 0 to 1, suggesting a reference point or centroid of activity somewhere within the* value space*.*

Definitions 6,7, and 8 allow us to discuss some geometry with very specific values, without having to know specifically what that geometry will be. When we combine all these spaces together we have our *visual space*, which can be used to parameterize the drawing or animation engine.

Definition 9 (Visual Space)**.** *A* visual space *is an object that contains a value space, focal space and color-space along with*

Spectacle -	*A value from 0 to 1, with 1 being more "fancy" and 0 being dull.*
Maximum elements -	*A value that specifies a hard limit on the number of generated elements.*

We create a new interface called *IVisualizer* meant to be implemented by an animation engine that draws things on screen. We need a way to tell the visualizer how to draw according to our music, so we provide a class structure called *VisualSpace* that conforms to our definitions. The visual space provides intuitive information that

Table 4.12: Visual Space Axioms

Axiom	Description
a	All values are considered to be homogeneous to one another and all data going forward.
b	Whenever possible values should range from 0 to 1.
c	Unless not supplied, all supplied colors are to be used and derived colors are to be gradients of supplied colors.

a *"knowledgable"* algorithm can compute ahead of time and offer the animator. The *visual space* decouples the animation implementation from engineering units or the type of data. There is an implicit agreement between the visual space and animation engine, which includes the tenets of Table 4.12.

As we progress we may add more parameters to the visual space but for now we move on to the melody and how we intend to extract useable values that can be fed the visual space. We attempt to describe the heuristics without getting too far into the source code, but the full melody analysis source can be found in Appendix A.0.13. The source code for the *IVisualizer* and our initial animation engine (derived from the time domain animation) can be found in the Appendix A.0.19, A.0.17 and A.0.18.

4.4.2 Mmmmmm, The Musical Melody Mathematical Modularity Movement Manager

If you are not chuckling you may not be ready to digest this approach. The technique we propose requires us to leverage everything we've considered up to this point. We approach transformation of melody using Tymoczko's *five properties* as a guide and exploit transposition and inversion to help create reasonable values. From Section 4.4.1 we need to parameterize the visual space in a way that makes sense. Unfortunately we must bombard the reader with several more definitions and theorems; however each definition is a critical component in calculating the visual space.

Theorem 6 (Angry Tigger Theorem)**.** *Given a chord C with N tones each having frequency $\boldsymbol{a}$, we can approximate a value of consonance ranging from 0 to 1*

$$x = \frac{1}{N^2 - N} \sum_{j=0}^{N-1} \sum_{i=0}^{N-1} |a_i - a_j|$$
$$\chi = \begin{cases} 1 & \text{if } \epsilon_1 < x > \epsilon_2 \\ \frac{2x-\epsilon_1}{2\epsilon_2-\epsilon_1} & otherwise \end{cases} \tag{4.13}$$

with 1 being "most" consonant.

Proof: It has been observed that two notes being played, which are very close together in frequency but not exact causes a dissonant tone [38, 43]. Given the frequencies of a chord $\{a_1, a_2, \cdots, a_N\}$ we compute the sum of $|a_i - a_j|$ for all $i, j \leq N$. We then take the average by multiplying $\frac{1}{N^2-N}$ subtracting N in the denominator to exclude items compared with themselves (which are necessarily zero) and assign this value to x. Let ϵ_1 be the difference in frequency between the A and $A\#$ notes, and ϵ_2 be the difference between the A and B notes. Let $P_0 = \frac{\epsilon_1}{2}$ and $P_1 = \epsilon_2$ such that $x = (1-t)P_0 + tP_1$. Note: we divide ϵ_2 by 2 to ensure we have a value that is very close, but not exactly equal. If x is less than ϵ_1 or x is greater than ϵ_2 we deem it consonant and return 1, otherwise we have a linear interpolation between P_0 and P_1 as a function of t thus $x = (1-t)\frac{\epsilon_1}{2} + t\epsilon_2 \Rightarrow t = \frac{x-\epsilon_1/2}{\epsilon_2-\epsilon_1/2} = \frac{2x-\epsilon_1}{2\epsilon_2-\epsilon_1}$, which yields t as a function of x. Since t is a value from 0 to 1 where 0 is equal to $\frac{\epsilon_1}{2}$ and 1 is equal to ϵ_2, we have a measure of consonance from 0 to 1. ■

Definition 10 (Magnitude of a Chord)**.** *Given chord* $C = \{c_1, c_2, \cdots, c_N\}$ *where* n *is a named note, the magnitude of a chord is defined to be the average frequency scaled by the average harmonic value*

$$\mu = \frac{1}{N^2} \sum_{j=1}^{N} \sum_{k=1}^{N} f_0(c_k) \cdot H(c_j) \tag{4.14}$$

Definition 11 (Centricity of a Chord)**.** *The centricity of a chord* $C = \{c_1, c_2, \cdots, c_N\}$ *where* c *is a named note, is defined as the origin about which a chord is centered with regards to both the harmonic value and fundamental frequency.*

The Harmonic Centricity

$$\theta_H = \frac{1}{N} \sum_{k=1}^{N} H(c_k) \tag{4.15}$$

The Fundamental Centricity

$$\theta_F = \frac{1}{N} \sum_{k=1}^{N} f_0(c_k) \tag{4.16}$$

Definition 11 describes centricity with respect to a single chord. We must also maintain the overall centricity of melody across all chords as they progress with respect to both harmonic value and pitch. We will refer to these values as Θ_H and Θ_F. We have the ability to measure things about a chord, but we need the ability to analyze change, this means additive change in regards to octave and pitch class. Technically speaking, the computer system does not support group operations of D_{12} since they are non-numeric/symbolic. To solve this we assign a unique integer value to each vertex as shown in Figure 4.23. This yields the $(\mathbb{Z}_{12}, +)$ group, which is isomorphic to the cyclic subgroup $< r_1 >= \{r_0, r_1, r_2, \cdots, r_{11}\}$, which is all the rotations of $(D_{12}, +)$. We leverage the fact that any finite cyclic group of order n is isomorphic

to $(\mathbb{Z}_n, +)$ [14], thus we can deal with translations of our notes as positive integers modulo 12.

Definition 12 (Change of a Chord)**.** *The* change of a chord *is a value from 0 to 1 where 1 means a large change in tonal structure. Let C, D be two chords whose elements are from $\mathbb{Z}_{12}$ sorted in ascending order. Given $m = max(|C|, |D|)$ where $|\cdot|$ is the order of the set, we define $X = \{C, 0_1, \cdots, 0_k, D, 0_1, \cdots, 0_l\}$ where $k \leq |C| - m$ and $l \leq |D| - m$, as the joined notes of C and D whose order is divisible by 2 with zero fill respectively. Let $Y = \{\, |X_t - X_{t+m}| \,\}$ with $0 \leq t < m$ be the differences between respective notes of C and D, which works because we have zero filled for uneven chords. If σ is the standard deviation and E the expected value, we compute the* change of a chord *as*

$$\Delta = 1 - \frac{|\sigma(Y) - E(Y)|}{E(Y)} \qquad E(Y) \neq 0 \tag{4.17}$$

Definition 13 (Spectacle of a Chord)**.** *Given successive chords C and D, whose tones are mapped to $\mathbb{Z}_{12}$ the* spectacle of a chord *is a measure from 0 to 1 where*

$$S_{k+1} = \frac{1}{N}\left[(N-1) \cdot S_k + \frac{|C_k - D_k|}{144}\right] \tag{4.18}$$

Although mathematically straightforward, Definition 12 can be tricky to envision. We are taking the differences between notes in two chords to determine to what extent they have changed uniformly. If the differences between the notes of any two chords are uniform *(ie: a transposition)* we have a value of 1 and something less than 1 otherwise. When we combine everything we've discussed so far we can begin to see the formation of the *view space of a chord*. Based upon the previous definitions and theorems, we propose computations allowing us to transform a series of chords and

subsequently their behaviors, into the visual space. We call this "Tigger's Roar".

Definition 14 (Tigger's Roar)**.** *Given a series of chords processed in time order the* view space of a chord *is defined to be*

Values:

$(v_1, v_2, v_3) = (\mu, \mu + \Delta, \theta_H)$

Focus:

$(c_1, c_2, c_3) = (0, 0, \Delta)$

Colors:

$(RGB_1, RGB_2, \cdots, RGB_k) =$ The Stripes Theorem

Spectacle:

Definition 13

Elements:

$E = 30 + (1 - \chi) * 50 + Spectacle * 50$

The notation in Definition 14 may be confusing. It is assumed all calculations are done with respect to the appropriate centricity and the most recent chord in the sequence. Definition 14 is really an amalgamation of our work combining the properties of music, Synesthetes response, Fourier analysis, note and chord detection and changes to melody over time. It is our final calculation and there is not much at this point that can proven mathematically.Figure 4.24 illustrates the visual space computed using Definition 14 against 100 seconds of symphony music. It appears we have, at least in part, achieved our goal of a view space that represents the changes of the melody. As desired the *primary focus* and *primary value* demonstrate a strong correlation, which means we have captured harmonic consistency. It would also seem we have addressed centricity in the *focus space* because the secondary and tertiary foci are stable and consistent, which follows the theory of conjunct melodic motion.

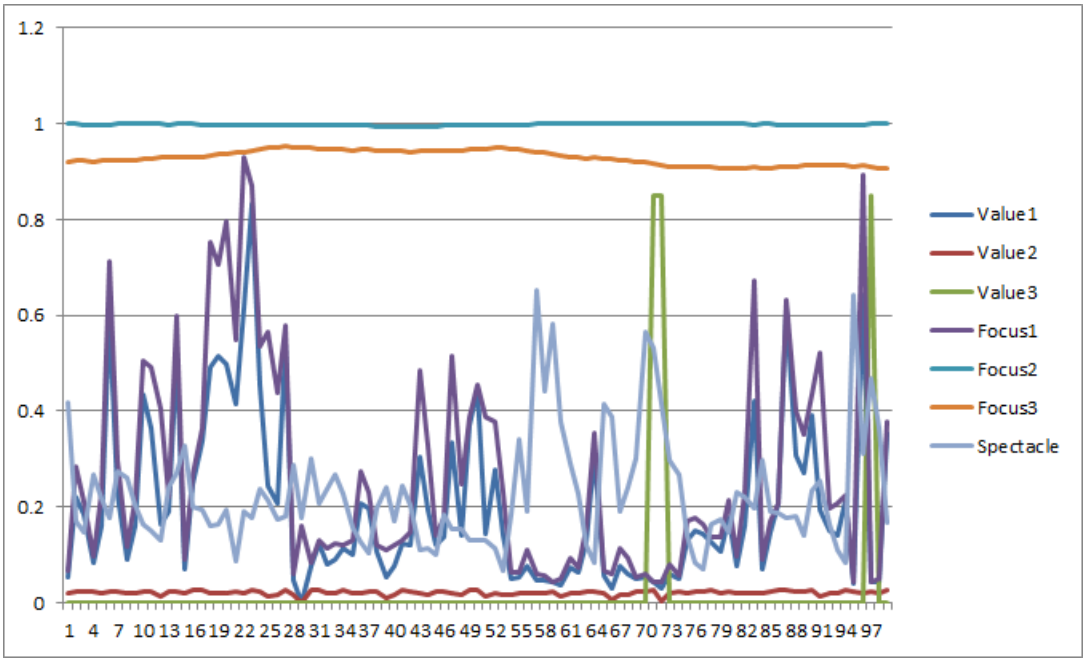

Figure 4.24: Visual Space Example

It is encouraging to see that the third element of the *value space* spikes only a few times. Recall that this is the chord magnitude scaled by a dissonance factor so we would not expect to see this value grow often. Unfortunately, it is impossible to truly deduce how the animation engine will behave from this graph.

5. Results

5.1 Experimentation

At this point we can only speculate as to how well our data will reflect the music. We must press onward and integrate the view space algorithm into our original time domain engine and visually examine the results. Figures 5.1 and 5.2 illustrate output using the melody-parameterized view space from Section 4.4. We will leave a more rigorous study and discussion in the various animation techniques for future work; for the time being one can see our current implementation in Appendix A.0.17. The imagery is stunning and we observe consistent connections between tone change and coloring as well as a coarse association between the geometry and melody. Unfortunately, it is impossible to capture the behavior of a fluid animation sequence in a document.

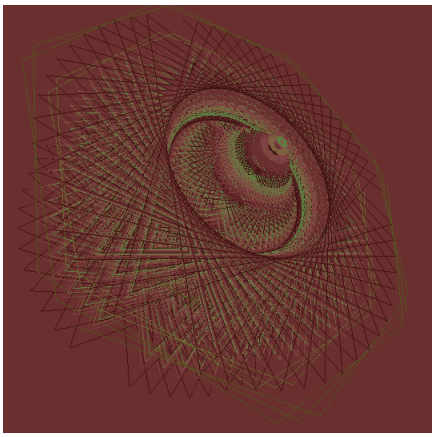

Figure 5.1: Beethoven Minuet in G

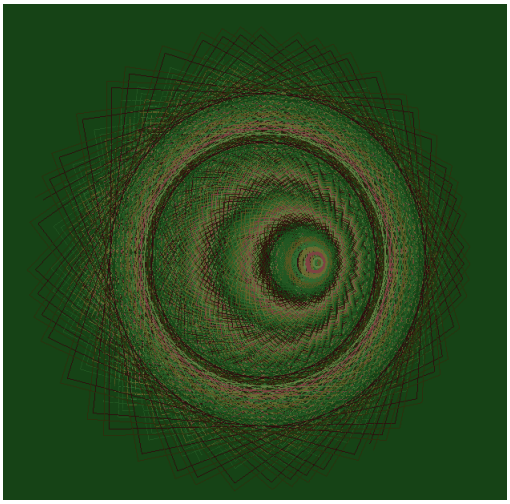

Figure 5.2: Techno Electronica (She Nebula)

Our thesis addresses the construction of a processing framework capable of detecting frequencies, characterizing tones and generating a parameterized geometry from the music. At this point we have only scratched the surface in terms of the ability to visualize melody but our prototype demonstrates interesting behavior. For the time

being we would like feedback on our progress so we devise a survey and supply it to a small number of individuals. We created an application and framework that plays several songs to a listener. Each song is followed by a questionnaire whose results are tallied and emailed to us upon completion. The questions are based upon a weighted system where we define either positive or negative results from the response. The areas of discovery are as follows; a) neutralize any bias toward the music genre b) neutralize a bias toward the aesthetics c) heavily weight "positive" associations of "mood" and synchronization. To that end we devised the questions and answers in Table 5.1.

Table 5.1: Questions and Answers

Importance	Question
2	Do you like this type of music?
3	Were the visuals aesthetically pleasing?
8	How much do you feel the visuals matched the "genre" of the song?
8	Were the visuals in sync with the music?
10	Do you feel the visuals captured the "mood" of the music?
10	Did you see and hear enough to answer questions?
5	Did the visuals keep your attention?

Weight	Answer
1	Unsure
5	Strongly Agree
4	Agree
3	Disagree
2	Strongly Disagree

To calculate the score per module, where each module is associated to one song, we let $Q = \{q_1, q_2, ..., q_k\}$ be the importance of each question based on Table 5.1, $A = \{a_1, a_2, \cdots, a_k\}$ be the answers to each question. We compute the *best* possible answer based upon the following; the subject strongly dislikes the music type; strongly dislikes the visuals; strongly believes the visuals matched the type of music; strongly found the visuals to be synchronized with the music; strongly believes the visuals captured the mood; strongly feels they saw enough to make a decision and was strongly focused on the experiment, which yields

Definition 15 (Best Score)**.** $\boldsymbol{B} = \{2, 2, 5, 5, 5, 5, 5\}$

This implies that the *worst* possible answer to be based upon; the subject strongly likes the music type; strongly likes the visuals; strongly believes the visuals did not match the type of music; strongly believes visuals to be out of synch; strongly believes the visuals did not capture the mood; strongly believes they did not see enough to make a decision and strongly believes they were distracted, which yields

Definition 16 (Worst Score)**.** $\boldsymbol{W} = \{5, 5, 2, 2, 2, 2, 2\}$

Equation 5.1 calculates the score by creating a weighted sum of the best answer corresponding to the points for that answer.

$$\begin{aligned} &\text{Let an inverse score be} \\ &s_i = \frac{5 - a_i}{5} q_i \\ &\text{Let a direct score be} \\ &t_i = \frac{a_i}{5} q_i \qquad (5.1) \\ &\text{Then we compute} \\ \text{score}(\boldsymbol{S}) &= s_1 + s_2 + t_3 + t_4 + t_5 + t_6 + t_7 \\ \text{grade} &= \frac{score(\boldsymbol{S})}{score(\boldsymbol{B})} \end{aligned}$$

Since the best possible grade is $\text{score}(\boldsymbol{B}) = 45$ we divide the users score from Equation 5.1 by 45 and get a "grade" for a song. The source code for processing the survey results can be found in A.0.20. In future work this foundation allows us to continue our research as the implementation grows more sophisticated.

5.2 Survey Results

The survey consisted of 11 people ranging from age 9 to 75 with both male and female participants. Figure 5.3 shows that Techno Electronica and Symphony music scored the highest. This is not at all surprising being that both were commonly used during development. The averages in Figure 5.4 seem low, but are much better than

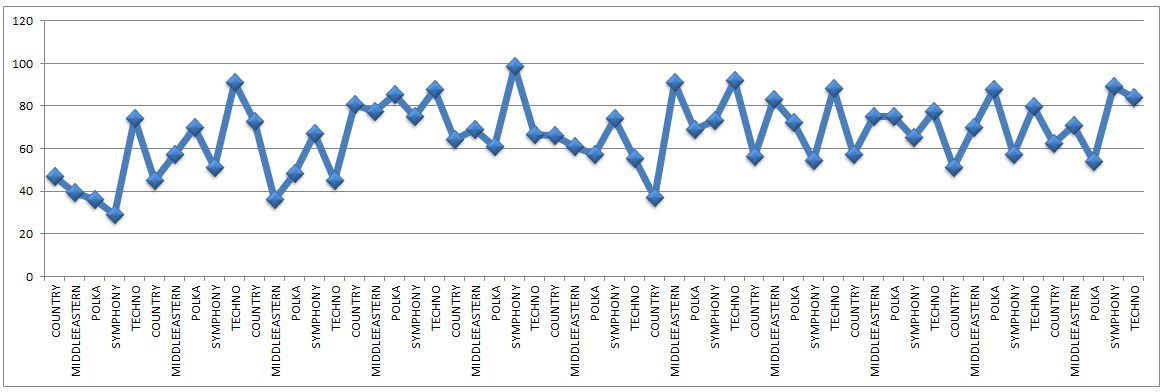

Figure 5.3: Survey Results by User (Music Genre vs Grade %)

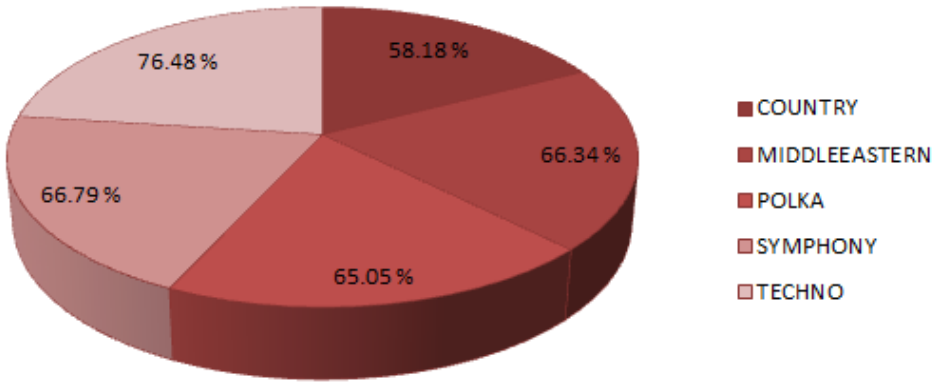

Figure 5.4: Survey Results, Average Grade by Genre

we had expected. When we compare the scores in Figure 5.4 with Figure 5.5 the dramatic shift within the same genre underlines the challenge we face with regard to subjectivity. It would be premature to draw any solid conclusions from this experiment, however we can state that we have a quantifiable measure of success and clearer understanding of external perception.

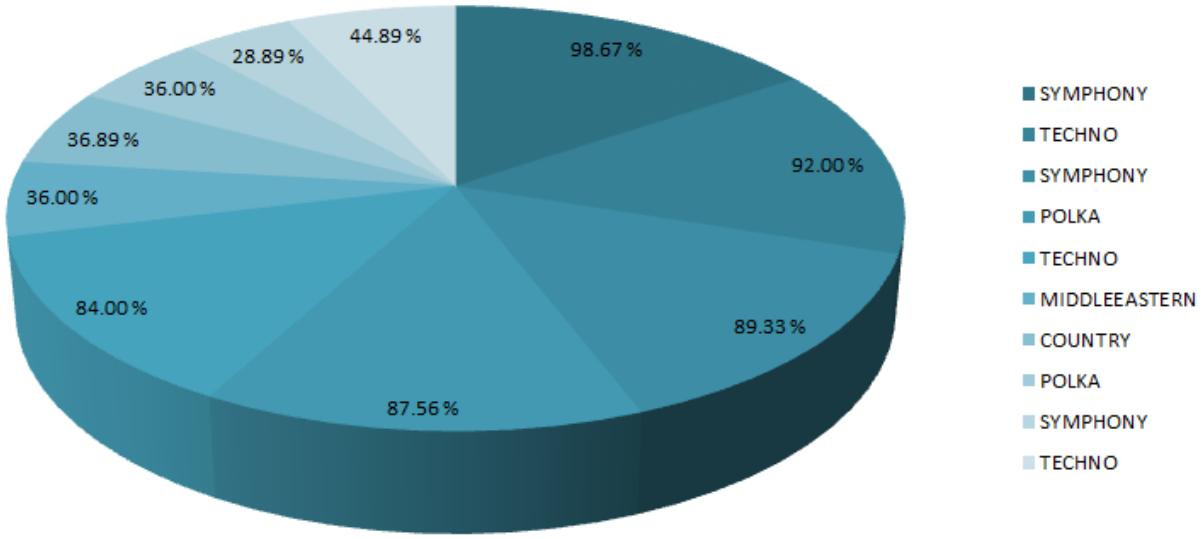

Figure 5.5: Survey Results, Top and Bottom 5 Scores

5.3 Conclusions and Future Work

As we have mentioned, time fidelity poses a real challenge in our ability to accurately model sound. Not only do we intend to evaluate our detection accuracy with realistic data, we intend to increase the time fidelity and extract sub-second peaks to improve characterization. Because we have observed that the Fourier transform is sensitive to the sample size, we must research adjusting all calculations. It follows that we will need a more effective detection; we may possibly use the technique proposed by Jehan [18]. We would also like to consolidate the entire model (end-to-end) in a nicely compacted algebraic factorization; the idea is to find a closed-form computation, which gets us closer to being able to prove an isomorphism. The issue of melody is very complex and still unclear how important it will be in our effort, but we shall continue to research this. We intend to continue focusing on the geometry of music and on mathematical representations of melody. Although we have captured a

good amount thus far, we want to continue researching what it means to be mathematically uplifting or depressing in terms of a pitch class operation. We also intend to expand upon our initial work with transposition and inversion and unlock a more sophisticated relationship within the musical chord progressions. Most importantly, now that we have a solid framework to expand upon, we plan on experimenting with many forms of geometry and animation to include fractals, fluid dynamics and other visuals that may be more conducive to intuitive representation.

5.3.1 Tangential Applications

There are many directions we can take this research and many of them are practical. One of the most uplifting, is the ability to allow a hearing impaired person to "see" what their environment sounds like. Not only for entertainment purposes but for safety and comprehension. Suppose that a deaf person has a dog and that person has never heard the animals voice. Our research may allow them to associate unique patterns of imagery with the mood or "timbre" of the animal allowing them to discern anger, happiness, concern or what have you. This may also be extended to include common sounds around the household. Imagine that a microwave "ding" goes off, or the oven timer or the washer/dryer. More importantly, imagine that someone breaks open the front door or a window or is yelling for help. A large panel on various walls of the house, or even a hand held device could alert the user to the sounds of the environment. With the proper mathematics the images would be consistent and therefore distinctly identifiable. You could say it's like braille for the eyes!

5.3.2 The Lawnmower Filter

During our analysis we identified a few mathematical techniques, which warrant further research and may prove useful going forward. The first technique involves instant filtering of unwanted frequency information using the matrix expansion of an exponential series. Recall the well known series $e^x = 1 + x + \frac{x^2}{2!} \cdots$, $cos(x) =$

$1-x^2+\frac{x^4}{4!}\cdots$ and $sin(x) = x-\frac{x^3}{3!}\cdots$. Recall Euler's Equation $e^{ix} = cos(x)+i\,sin(x)$. If we replace x with a matrix $\mathbf{A}$ and insert it into the exponential series we have $e^A = I + A + \frac{A^2}{2!}\cdots$. Using Euler's Equation we have $e^{i\mathbf{A}} = cos(\mathbf{A}) + i\,sin(\mathbf{A})$ and we can solve for the cosine and sine using the series expansion and some convergence technique. Now suppose that $\mathbf{A}$ is a diagonal matrix whose elements A_{ii} are of the form $\{-2\pi k_1, -2\pi k_2, \cdots\}$. If we substitute the matrix form of the exponential function and multiply the exponent with the imaginary unit we have a simultaneous Fourier transform that checks for multiple frequencies at once $Y = \sum_{n=0}^{N-1} X_n \cdot e^{i\mathbf{A}\cdot n}$. The risk here is that our response to multiple frequencies is bound together and likely to be inaccurate with respect to the current sample. Lets reverse our thinking from detection to filtering. We have the n'th roots of unity but not necessarily a valid coefficient. If we select some magnitude ρ such that

$$X_k = \sum_{n=0}^{N-1} X_n - \rho \cdot ||e^{i\mathbf{A}\cdot n}||_p \tag{5.2}$$

we may just be able to "delete" unwanted noise or other frequencies from the signal enough to see some desired pattern.

5.3.3 An Instrument Fingerprint

One topic we have not yet breached in our research is the ability to identify specific instruments. It is unclear how much this will affect our overall goal but some of the Synesthesia research demonstrates specific colors and shapes chosen based upon musical instrument [32]. We propose one possible technique to identify the signature, or fingerprint if you will, of an instrument. Our approach is quite simple, we anchor a value at the first point in the function where the slope is positive. We draw a curve (line) to each successive point from the anchor making little triangles and compute the angle off the x-axis for each triangle. We do this until the slope goes negative. If we add the results together for some portion of the entire wave, we have a value for each positively monotonic segment of the function. Figure 5.6 illustrates a crude

depiction of the geometry. The hope is that each instrument can theoretically be modeled by some function $f(x)$ whose derivative identifies a unique signature to that instrument.

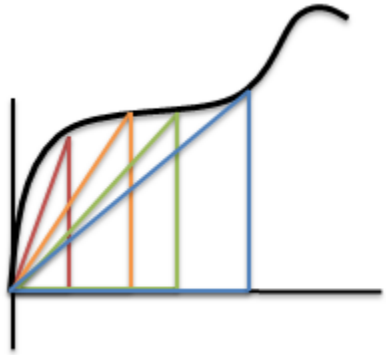

Figure 5.6: Fingerprint Technique

One possible solution might be to integrate over each segment of positive slope and add all the angles together. Let $\boldsymbol{\nu}(x) = \begin{cases} 0 & f'(x) < 0 \\ 1 & f'(x) > 0 \end{cases}$ then the function

$$\varphi = \sum \left[\int_{\nu(x)\neq 0}^{\nu(x)=0} tan^{-1}(\frac{|f(x)|}{|x - x_0|})dx \right]. \tag{5.3}$$

Equation 5.3 provides a continuous model of this approach that aggregates the front side slope of some series of impulses over a desired region of the wave. We simply derive a discrete version of this function, combine it with Equation 5.2 to reduce noise and voila! a unique value that describes the instrument. If we take the expected value over several samples and create a range of tolerance we have a min/max bounds for detecting the instrument.

5.3.4 Conclusions

Although we made significant progress, our research is far from over. We have learned a number of things about the nature of processing sound and the challenges of precision and timeliness. We successfully melded the neurological response of *colored hearing* with the computer systems ability to generate imagery. We demonstrated a mathematical mapping from sound to pictograph and successfully implemented a flexible framework capable of processing any sound in real time. We have demonstrated a substantial improvement in capturing the melody in the frequency domain versus the time domain. We have proposed and implemented a unique technique for musical frequency detection and note characterization. We have exposed our implementation to human subjects and seen encouraging results. All in all, our progress opens the door to a larger world and lays the foundation for more possibilities in extending human sensory perception.

REFERENCES

[1] Thomas M. Fiore Alissa S. Crans and Ramon Satyendra. Musical actions of dihedral groups. *The Mathematical Association of America*, June - July, 2009.

[2] MIDI Manufacturers Association. History of midi. `http:/www.midi.org/aboutmidi/tut_history.php`, 2015.

[3] Jan Dirk Blom. *A Dictionary of Hallucinations*, page 73. Springer Science+Business Media, 2010.

[4] C. Sidney Burrus. Index mappings for multidimensional formulation of the DFT and convolution. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, ASSP-25(3):239–241, 1977.

[5] Glenda K Larcombe Carol Bergfeld Mills, Edith Howell Boteler. Seeing things in my head: A synesthete's images for music and notes. *Perception, volume 32*, pages 1359 – 1376, 2003.

[6] Aldo Piccialli Giovanni De Poli Curtis Roads, Stephen Travis Pope. *Musical Signal Processing*. Swets & Zeitlinger B.V, Lisse, Netherlands, 1997.

[7] Michigan Tech Department of Physics. Musical signal frequencies. `http://www.phy.mtu.edu/~suits/notefreqs.html`, 2015.

[8] Zohar Eitan and Inbar Rothschild. How music touches: Musical parameters and listeners audio-tactile metaphorical mappings. *Psychology of Music 39(4)*, pages 449–467, 2010.

[9] Shannon Steinmetz Ellen Gethner and Joseph Verbeke. A view of music. In Douglas McKenna Kelly Delp, Craig S. Kaplan and Reza Sarhangi, editors, *Proceedings of Bridges 2015: Mathematics, Music, Art, Architecture, Culture*, pages 289–294, Phoenix, Arizona, 2015. Tessellations Publishing. Available online at `http://archive.bridgesmathart.org/2015/bridges2015-289.html`.

[10] Jai Sam Kim Nicola Veneziani Giovanni Aloisio, G.C Fox. A concurrent implementation of the prime factor algorithm on hypercube. *IEEE Transactions on Signal Processing*, 39(1), 1991.

[11] I.J. Good. The interaction algorithm and practical fourier analysis. *Journal of the Royal Statistical Society. Series B*, 20(2):361–372, 1958.

[12] Thomas W. Hungerford. *Abstract Algebra An Introduction*, page 176. Brooks/-Cole, 2014.

[13] Thomas W. Hungerford. *Abstract Algebra An Introduction*, pages 169–179. Brooks/Cole, 2014.

[14] Thomas W. Hungerford. Abstract algebra an introduction. page 219. Brooks/-Cole, 2014.

[15] Johannes Itten. The art of color. page 34. Wiley & Sons INC, 1973.

[16] Debra A. Zellner J. Michael Barbiere, Ana Vidal. The color of music:correspondence through emotion. *Empirical Studies Of The Arts, Vol. 25(2)*, pages 193–208, 2007.

[17] John W. Tukey James W. Cooley. An algorithm for the machine calculation of complex fourier series. *Mathematics of Computation*, 19(90):397–301, Apr. 1965.

[18] Tristan Jehan. Musical signal parameter estimation. Master's thesis, IFSIC, Université de Rennes, France, and Center for New Music and Audio Technologies (CNMAT), University of California, Berkeley, USA, 1997.

[19] George H. Joblove and Donald Greenberg. Color spaces for computer graphics. *SIGGRAPH 5'th Annual*, pages 20–25, 1978.

[20] Harry F. Jordan and Gita Alaghband. Fundamentals of parallel processing. page 160. Pearson Education, 2003.

[21] Giuliano Monti Juan Pablo Bello and Mark Sandler. An implementation of automatic transcription of monophonic music with a blackboard system. *Iris Signals ans Systems Conference (ISSC 2000)*, 2000.

[22] Giuliano Monti Juan Pablo Bello and Mark Sandler. Techniques for automatic music transcription. 2000.

[23] Guillaume Leparmentier. Manipulating colors in .net. `http://www.codeproject.com/Articles/19045/Manipulating-colors-in-NET-Part`, 2016.

[24] Michal Levy. Giant steps. `http://www.michalevy.com/giant-steps/index.html`, 2015.

[25] David Lewin. Generalized musical intervals and transformations. pages 9–11. Oxford University Press, 1 edition, 2011.

[26] Stephen Malinowski. The music animation machine. `http://www.kunstderfuge.com/theory/malinowski.htm`, 2016.

[27] Lawrence E. Marks. On associations of light and sound, the mediationof brightness, pitch and loudness. *The American Journal of Psychology, Vol. 87, No 1/2*, pages 173–188, Nov 2016.

[28] Paul Masri and Andrew Bateman. Improved modeling of attack transients in music analysis-resynthesis. pages 100–103, 1996.

[29] James A. Moorer. On the transcription of musical sound. *Computer Music Journal*, 1(4):32–38, 1977.

[30] Alan V. Oppenheim. Speech spectographs using the fast fourier transform. *IEEE Spectrum*, 7(8):57–62, Aug 1970.

[31] Konstantina Orlandatou. Sound characteristics which affect attributes of the synaesthetic visual experience. *Musicae Scientiae, Vol. 19(4)*, page 389401, 2015.

[32] Otto Ortmann. Theories of synesthesia in the light of a case of color-hearing. *Human Biology*, 5(2):155–211, May 1933.

[33] Otto Ortmann. Theroies of synesthesia in the light of a case of color-hearing. *Human Biology*, 5(2):176, May 1933.

[34] Martin Piszczalski and Bernard A. Galler. Automatic music transcription. *Computer Music Journal*, 1(4):24–31, Nov. 1977.

[35] Cornel Pokorny. *Computer Graphics An Object-Oriented Approach To The Art And Science*. Franklin, Beedel and Associates Incorporated, 1 edition, 1994.

[36] Frank B. Wood Richard E. Cytowic. Synesthesia i. a review of major theories and thier brain basis. pages 36–49, 1982.

[37] Frank B. Wood Richard E. Cytowic. Synesthesia ii. psychophysical relations in the synesthesia of geometrically shaped taste and colored hearing. *Ninth Annual Meeting of the International Neuropsychological Society,Brain and Cognition*, pages 36–49, 1982.

[38] Stephen W. Smith. *The Scientists and Engineer's Guide to Digital Signal Processing*. California Technical Publishing, 1997.

[39] Praveen Sripada. Mp3 decoder in theory and practice. Master's thesis, Blekinge Institude of Technology, March 2006.

[40] Clive Temperton. Implementation of a self-sorting in-place prime factor FFT algorithm. *Journal of Computational Physics*, 58(4):283–299, 1985.

[41] Clive Temperton. A generalized prime factor FFT algorithm for $n = 2^p 3^q 5^r$. *SIAM*, 13(3):676–686, May 1992.

[42] Dimitri Tymoczko. The geometry of muscial chords. *Science*, 313(0036-8075):72, July 2006.

[43] Dimitri Tymoczko. *A Geometry of Music*. Oxford University Press, 1 edition, 2011.

[44] Dimitri Tymoczko. A geometry of music. pages 35–36. Oxford University Press, 1 edition, 2011.

[45] Dimitri Tymoczko. A geometry of music. pages 28–32. Oxford University Press, 1 edition, 2011.

[46] Dimitri Tymoczko. A geometry of music. pages 33–34. Oxford University Press, 1 edition, 2011.

[47] C. van Campen. *The Hidden Sense: Synesthesia in Art and Science*. Leonardo (Series) (Cambridge, Mass.). MIT Press, 2008.

[48] Dennis G. Zill and Patrick D. Shanahan. *A First Cource in Complex Analysis with Applications*, page 35. Jones and Bartlett, 2009.

APPENDIX A. Source Code

The entire application framework involves several thousand lines of source code including everything from U/I components to logging utilities. We offer a subset of the specific classes and algorithms most pertinent to our thesis.

A.0.5 Musical Detect Class

```
  using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace Toolkit.Media.DSP {

  /// <summary>
  /// Provides a musical detection class that defines the
  /// detection of some musical note and all its details.
  /// </summary>
  public struct MusicalDetect {

    public static readonly MusicalDetect Empty = new MusicalDetect(MusicalNote.Empty,0f,0f,0);

    #region Private
    private long myTimeOn;
    private long myDuration;
    private float myRf;
    private float myAmp;
    private MusicalNote myNote;
    private int myHarmonic;
    #endregion

    /// <summary>
    /// Create the detection of a note.
    /// </summary>
    /// <param name="note">The note found</param>
    /// <param name="rf">The frequency it was found</param>
    /// <param name="amp">The intensity</param>
    /// <param name="timeOn">The first epoch since start of play it was detected in milliseconds</
        param>
    public MusicalDetect(MusicalNote note, float rf, float amp, long timeOn) {
      myNote = note;
      myRf = rf;
      myAmp = amp;
      myTimeOn = timeOn;
      myHarmonic = (int)Math.Ceiling(Math.Log(rf/note.Fundamental,2));
      myDuration = 0;
      }

    /// <summary>
    /// Create the detection of a note.
    /// </summary>
    /// <param name="note">The note found</param>
    /// <param name="rf">The frequency it was found</param>
    /// <param name="amp">The intensity</param>
    /// <param name="timeOn">The first epoch since start of play it was detected in milliseconds</
        param>
    /// <param name="duration">The duration this note lasts in milliseconds.</param>
    public MusicalDetect(MusicalNote note, float rf, float amp, long timeOn, long duration) :
        this(note,rf,amp,timeOn) {
      myDuration = duration;
      }

    #region Properties

    /// <summary>
    /// Get the first observed time of this detection from epoch since
    /// stream start in milliseconds.
    /// </summary>
    public long TimeOn { get { return myTimeOn; } set { myTimeOn = value; } }

    /// <summary>
    /// Get the duration of this note in milliseconds.
    /// </summary>
    public long Duration { get { return myDuration; } set { myDuration = value; } }

    /// <summary>
    /// Get the original sampled rf value.
    /// </summary>
    public float Rf { get { return myRf; } }

    /// <summary>
    /// Get the impulse value.
```

```
    /// </summary>
    public float Amp { get { return myAmp; } }

    /// <summary>
    /// Get the note this matches.
    /// </summary>
    public MusicalNote Note { get { return myNote; }}

    /// <summary>
    /// Get/Set the optional harmonic if known.
    /// </summary>
    public int Harmonic { get { return myHarmonic; } set { myHarmonic = value; }}

    #endregion

    /// <summary>
    /// Is this a non detect.
    /// </summary>
    public bool IsEmpty {
      get { return myNote.IsUnknown && myTimeOn ==0 && myRf == 0 && myAmp == 0; }
      }

    #region Object Overrides

    public override bool Equals(object obj) {
      MusicalDetect det = (MusicalDetect)obj;
      return myTimeOn == det.myTimeOn &&
             myRf == det.myRf &&
             myAmp == det.myAmp &&
             myNote.Equals(det.myNote);
      }

    public override int GetHashCode() {
      return ToString().GetHashCode();
      }

    public override string ToString() {
      return "(" + myNote.Note + "" + Harmonic + ":" + Rf  + ") " + "On:" + myTimeOn + ", Amp:" +
          myAmp;
      }


    #endregion

    }

}
```

A.0.6 Musical Note Class

```
  using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace Toolkit.Media.DSP {

  /// <summary>
  /// Immutable structure that holds information about a musical note.
  /// </summary>
  public struct MusicalNote {

    #region Properties
    public static readonly MusicalNote Empty = new MusicalNote(eMusicNote.Unknown,0,0);
    private eMusicNote myNote;
    private float myRf;
    private float myTiggerHarmonic;
    #endregion

    public MusicalNote(MusicalNote inOther) : this(inOther.myNote,inOther.myTiggerHarmonic,inOther
        .myRf) {}

    /// <summary>
    /// Create the musical note.
    /// </summary>
    /// <param name="inNote">The note enumeration</param>
    /// <param name="inTiggerRf">The tigger mapping</param>
    /// <param name="inFundamental">The corresponding fundamental frequency</param>
    /// <param name="inRf">The measured frequency (harmonic)</param>
    public MusicalNote(eMusicNote inNote, float inTiggerRf, float inRf) {
      myNote = inNote;
      myRf = inRf;
      myTiggerHarmonic = inTiggerRf;
      }

    #region Properties

    /// <summary>
    /// Get the raw note name
    /// </summary>
    public eMusicNote Note { get { return myNote;  } }

    /// <summary>
    /// Get the fundamental frequency for this note.
    /// </summary>
    public float Fundamental { get { return myRf; }}

    /// <summary>
    /// Get the calculated Tigger Rf
    /// </summary>
    public float TiggerHarmonic { get { return myTiggerHarmonic; } }

    #endregion

    #region Utility and Transforms

    /// <summary>
    /// Generate step harmonics above this tone.
    /// </summary>
    public MusicalDetect Harmonic(int step) {
      float newRf = (float)(myRf*Math.Pow(2.0,(float)step));
      MusicalDetect md = new MusicalDetect(this,newRf,1,0);
      md.Harmonic = step;
      return md;
      }

    #endregion

    #region Object Overrides
    public bool IsUnknown { get { return myNote == eMusicNote.Unknown && myRf == 0; } }

    public override string ToString() {
      return "(" + myNote + "␣:" + myRf + ":" + myTiggerHarmonic + ")";
      }
```

```
  public override bool Equals(object obj) {
    MusicalNote mn = (MusicalNote)obj;
    return myNote == mn.myNote &&
                myRf == mn.myRf &&
                myTiggerHarmonic == mn.myTiggerHarmonic;
    }

  public override int GetHashCode() {
    return ToString().GetHashCode();
    }

  #endregion
  }

}
```

A.0.7 Music Utility Class

```
  using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

using System.Drawing;
using Core.Drawing;
using Core.Mathematics;
using Core.Mathematics.Algebra;
using Core.Mathematics.Stats;
using Core.Mathematics.Analysis;
using System.Diagnostics;


namespace Toolkit.Media.DSP {

  /// <summary>
  /// Provides a table of fundimental frequencies for musical
  /// tones.
  ///
  /// </summary>
  public class Music {
    private static readonly Random ourRandom = new Random();

    #region Performance Testing Constants

    public static bool UseWeakDetect = false;
    /// <summary>
    /// Always has the runtime (ms) of the most recent call to
    /// Guess.
    /// </summary>
    public static long GuessTime = 0;

    #endregion

    #region Private Constants
    /// <summary>
    /// Map of unique integers to musical notes.
    /// </summary>
    private static readonly Dictionary<int,MusicalNote> ourTiggerMap;
    /// <summary>
    /// Map of integer fundamental frequencies to the note.
    /// </summary>
    private static readonly Dictionary<int,MusicalNote> ourIntegerFund;

    /// <summary>
    /// Base color value to a tigger value.
    /// </summary>
    private static readonly Dictionary<int,int> ourTiggerColorMap;

    /// <summary>
    /// Consonance delta. (Difference between C/C#)
    /// </summary>
    private static readonly float DISONANCE_1 = 100f;
    private static readonly float DISONANCE_2 = 150f;

    #endregion

    #region Private Static Util

    /// <summary>
    /// Create the full harmonic set for the given sample rate.
    /// </summary>
    private static float[] CreateHarmonic(int sampleRate) {

      // Create all fundamental and harmonics
      List<float> rfs = new List<float>();
      int max = (int)Math.Ceiling(Math.Log((sampleRate/2.0f/Music.Min),2.0f));

      // For each Rf generate each harmonic.
      for(int r=0;r<Music.Notes.Length;r++) {
        for(int k=0;k<max;k++) {
          float rf = (float)(Music.Notes[r].Fundamental*Math.Pow(2.0f,k));

          if(rf < (sampleRate/2.0f))
```

```
            rfs.Add((float)(Music.Notes[r].Fundamental*Math.Pow(2.0f,k)));
          }
        }

    return rfs.ToArray();
    }

#endregion

#region Static Constructor
static Music() {
    ourTiggerMap = new Dictionary<int,MusicalNote>();
    for(int i=0;i<Notes.Length;i++) ourTiggerMap.Add(TiggerHarmonic(Notes[i].Fundamental),Notes[
        i]);

    // Create fundamentals
    Fundamentals = new float[Notes.Length];
    for(int i=0;i<Notes.Length;i++) Fundamentals[i] = Notes[i].Fundamental;

    // Create harmonics.
    Harmonics = new Dictionary<int,float[]>();
    for(int i=0;i<KnownSampleRates.Rates.Length;i++) {
      float[] vals = CreateHarmonic((int)KnownSampleRates.Rates[i]);
      Harmonics.Add((int)KnownSampleRates.Rates[i],vals);
      }

    // Create integer fundamentals.
    ourIntegerFund = new Dictionary<int,MusicalNote>();
    for(int i=0;i<Notes.Length;i++)
      ourIntegerFund.Add((int)Notes[i].Fundamental,Notes[i]);

    // Create tigger color map
    ourTiggerColorMap = new Dictionary<int,int>();
    ourTiggerColorMap.Add(TiggerHarmonic(C0),Color.Blue.ToArgb());
    ourTiggerColorMap.Add(TiggerHarmonic(Cs0),Color.Blue.ToArgb());
    ourTiggerColorMap.Add(TiggerHarmonic(D0),Color.Red.ToArgb());
    ourTiggerColorMap.Add(TiggerHarmonic(Ds0),Color.Red.ToArgb());
    ourTiggerColorMap.Add(TiggerHarmonic(E0),Color.Yellow.ToArgb());
    ourTiggerColorMap.Add(TiggerHarmonic(F0),Color.Brown.ToArgb());
    ourTiggerColorMap.Add(TiggerHarmonic(Fs0),Color.Brown.ToArgb());
    ourTiggerColorMap.Add(TiggerHarmonic(G0),Color.Green.ToArgb());
    ourTiggerColorMap.Add(TiggerHarmonic(Gs0),Color.Green.ToArgb());
    ourTiggerColorMap.Add(TiggerHarmonic(A0),Color.Green.ToArgb());
    ourTiggerColorMap.Add(TiggerHarmonic(As0),Color.Green.ToArgb());
    ourTiggerColorMap.Add(TiggerHarmonic(B0),Color.Black.ToArgb());
    }

#endregion

#region Public Constants

/// <summary>
/// Maximum Fundimental Rf in the table.
/// </summary>
public const float Max = B0;

/// <summary>
/// Minimum Fundimental Rf in the table.
/// </summary>
public const float Min = C0;

public static readonly MusicalNote C0n = new MusicalNote(eMusicNote.C,Music.TiggerHarmonic(C0)
    ,C0);
public static readonly MusicalNote Cs0n = new MusicalNote(eMusicNote.Cs,Music.TiggerHarmonic(
    Cs0),Cs0);
public static readonly MusicalNote D0n = new MusicalNote(eMusicNote.D,Music.TiggerHarmonic(D0)
    ,D0);
public static readonly MusicalNote Ds0n = new MusicalNote(eMusicNote.Ds,Music.TiggerHarmonic(
    Ds0),Ds0);
public static readonly MusicalNote E0n = new MusicalNote(eMusicNote.E,Music.TiggerHarmonic(E0)
    ,E0);
public static readonly MusicalNote F0n = new MusicalNote(eMusicNote.F,Music.TiggerHarmonic(F0)
    ,F0);
public static readonly MusicalNote Fs0n = new MusicalNote(eMusicNote.Fs,Music.TiggerHarmonic(
    Fs0),Fs0);
public static readonly MusicalNote G0n = new MusicalNote(eMusicNote.G,Music.TiggerHarmonic(G0)
    ,G0);
public static readonly MusicalNote Gs0n = new MusicalNote(eMusicNote.Gs,Music.TiggerHarmonic(
    Gs0),Gs0);
public static readonly MusicalNote A0n = new MusicalNote(eMusicNote.A,Music.TiggerHarmonic(A0)
    ,A0);
public static readonly MusicalNote As0n = new MusicalNote(eMusicNote.As,Music.TiggerHarmonic(
    As0),As0);
```

```
    public static readonly MusicalNote B0n = new MusicalNote(eMusicNote.B,Music.TiggerHarmonic(B0)
        ,B0);

    /// <summary>
    /// List of all Rf's for musical tones starting with
    /// C -> B.
    /// </summary>
    public static readonly MusicalNote[] Notes = {
                            C0n,
                            Cs0n,
                            D0n,
                            Ds0n,
                            E0n,
                            F0n,
                            Fs0n,
                            G0n,
                            Gs0n,
                            A0n,
                            As0n,
                            B0n,
};

    /// <summary>
    /// Has the list of fundimental frequenices in order
    /// from C to B.
    /// </summary>
    public static readonly float[] Fundamentals;

    /// <summary>
    /// A map that contains a mapping from the sample rate to float arrays each of which are the
    /// set of fundamentals and 11 harmonics in order.
    ///
    /// Key := The sample rate.
    /// Value := The frequencies.
    /// </summary>
    public static readonly Dictionary<int,float[]> Harmonics;

    #endregion

    #region Characterization Methods

    /// <summary>
    /// Execute the Tigger harmonic theorem which maps a frequency to a unique integer.
    /// </summary>
    public static int TiggerHarmonic(float f) {
      return (int)(100.0f*(f/Math.Pow(2,Math.Floor(Math.Log(f,2))) - 1.0f));
      }

    /// <summary>
    /// Use the Tigger harmonc to determine what note corresponds to the given frequency.
    /// </summary>
    /// <param name="inF"></param>
    /// <returns></returns>
    public static MusicalNote Characterize(float f) {
      int th = TiggerHarmonic(f);
      if(ourTiggerMap.ContainsKey(th)) return ourTiggerMap[th];

      if(!UseWeakDetect) {
        if(ourIntegerFund.ContainsKey((int)f))
          return ourIntegerFund[(int)f];
        }

      float sm = float.MaxValue;
      int smI = -1;

      for(int i=0;i<Notes.Length;i++) {
        float t = Math.Abs(Notes[i].TiggerHarmonic - th);
        if(t < sm) {
          sm = t;
          smI = i;
          }
        }

      return Notes[smI];
      }

    /// <summary>
    /// Determine the presence of which notes exist in the given input stream of
    /// samples until end of stream is reached.
    /// </summary>
    /// <param name="X">The sequence of sample values.</param>
    /// <param name="offset">The starting offset to execute in X</param>
    /// <param name="sampleRate">The sample rate</param>
```

```
/// <param name="stats">The current running statistics or null to compute.</param>
public static MusicalDetect[] GuessNotes(Z2[] X, int offset, int sampleRate) {
  // Add the sample rate if we don't already know about it.
  if(!Harmonics.ContainsKey(sampleRate)) Harmonics.Add(sampleRate,CreateHarmonic(sampleRate));

  List<MusicalDetect> notes = new List<MusicalDetect>();
  float N = sampleRate;
  float[] freqs = Harmonics[sampleRate];
  int i = offset;
  float max = 0;

  while(i < X.Length) {
    FourierTransform.RfFFT(X,sampleRate,i,freqs,eWindowType.Hanning);

    // Compute max{X} and noise
    max = 0;
    for(int m=0;m<sampleRate;m++) {
      float aa = X[i+m].B.Magnitude;
      if(aa > max)
        max = aa;
      }

    // Find all values whose peak to signal is large enough.
    for(int k=0;k<sampleRate;k++) {
      float m = X[i + k].B.Magnitude;
      float d = m/max;

      if(d > SoundParameters.RF_DETECT_FULL_AMP) {
        MusicalDetect det = new MusicalDetect(Characterize(k),k,m,(long)((((double)i)/((double
            )sampleRate))*1000.0d));
        // Statically added for now.
        det.Duration = 1000;
        notes.Add(det);
        }
      }

    i += sampleRate;
    }

  return notes.ToArray();
  }

/// <summary>
/// Compute the consonance factor.
/// Returns a value from 0 to 1 with 1 being the most consonant tone.
/// </summary>
/// <returns>A value from 0 to 1 with 1 being completely consonant.</returns>
public static float Consonance(Chord ch) {
  if(ch.Count == 1) return 1;

  float aveDelta = 0;
  for(int i=0;i<ch.Count;i++) {
    for(int j=0;j<ch.Count;j++)
      aveDelta += Math.Abs(ch[i].Rf - ch[j].Rf);
    }
  aveDelta = Math.Abs(aveDelta/((float)(ch.Count*ch.Count-ch.Count)));

  if(aveDelta > DISONANCE_2) return 1;

  return (float)(2.0f*aveDelta - DISONANCE_1)/(2.0f*DISONANCE_2 - DISONANCE_1);
  }

/// <summary>
/// Get the Z12 element for a specific note.
/// </summary>
public static int Z12(eMusicNote note) {

  switch(note) {

    case eMusicNote.C:
      return 0;
    case eMusicNote.Cs:
      return 1;
    case eMusicNote.D:
      return 2;
    case eMusicNote.Ds:
      return 3;
    case eMusicNote.E:
      return 4;
    case eMusicNote.F:
      return 5;
    case eMusicNote.Fs:
      return 6;
```

```
    case eMusicNote.G:
      return 7;
    case eMusicNote.Gs:
      return 8;
    case eMusicNote.A:
      return  9;
    case eMusicNote.As:
      return 10;
    case eMusicNote.B:
      return 11;
    }

  return -1;
  }

/// <summary>
/// Convert a Z12 element to a pitch scale element.
/// </summary>
public static eMusicNote PitchScale(int ps) {

  switch(ps) {

    case 0:
      return eMusicNote.C;
    case 1:
      return eMusicNote.Cs;
    case 2:
      return eMusicNote.D;
    case 3:
      return eMusicNote.Ds;
    case 4:
      return eMusicNote.E;
    case 5:
      return eMusicNote.F;
    case 6:
      return eMusicNote.Fs;
    case 7:
      return eMusicNote.G;
    case 8:
      return eMusicNote.Gs;
    case  9:
      return eMusicNote.A;
    case 10:
      return eMusicNote.As;
    case 11:
      return eMusicNote.B;
    }

  return eMusicNote.Unknown;
  }

/// <summary>
/// Determine a value from 0 to 1 that is how much a chord changes in tonal structure, meaning
/// 1 is a big change.
/// </summary>
/// <remarks>A value from 0 to 1.</remarks>
public static float ChangeOfChord(Chord c1, Chord c2) {
  int m = Math.Max(c1.Count,c2.Count);

  // Preparation. (This could be done in one loop, but I'm tired).
  // nest both arrays in one array
  float[] X = new float[m*2];
  for(int i=0;i<c1.Count;i++) X[i] = Z12(c1[i].Note.Note);
  for(int i=0;i<(c1.Count-m);i++) X[c1.Count + i] = 0;
  for(int i=0;i<c2.Count;i++) X[m + i] = Z12(c2[i].Note.Note);
  for(int i=0;i<(c2.Count-m);i++) X[m + i] = 0;

  // Calculation |x_t - x_{t+m}|
  float[] Y = new float[m];
  for(int t=0;t<m;t++) Y[t] = Math.Abs(X[t]-X[t+m]);

  // Compute std.
  float exp = ExtraStats.Mean(Y);
  float std = ExtraStats.StdDev(Y);
  if(exp == 0) return 0;

  return 1f - Math.Abs(std-exp)/exp;
  }

/// <summary>
/// Compute the magnitude of a chord.
/// </summary>
public static float Magnitude(Chord c1) {
```

```
  float f = 0;
  float N = c1.Count;

  for(int i=0;i<c1.Count;i++) {
    for(int j=0;j<c1.Count;j++)
      f += c1[i].Rf*(c1[i].Harmonic+1);
    }

  return (1.0f/(N*N))*f;
  }

/// <summary>
/// Compute the harmonic centricity theta_H of a chord.
/// </summary>
public static float HarmonicCentricity(Chord c) {
  float t= 0;
  float N = c.Count;
  for(int i=0;i<c.Count;i++) t += (c[i].Harmonic+1);
  return t/N;
  }

/// <summary>
/// Compute the fundamental centricity theta_F of a chord.
/// </summary>
public static float FundamentalCentricity(Chord c) {
  float t= 0;
  float N = c.Count;
  for(int i=0;i<c.Count;i++) t += c[i].Note.Fundamental;
  return t/N;
  }

#endregion

#region Musical Tone Rf Values

public const float C0 = 16.35f;
public const float Cs0 =  17.32f;
public const float D0 = 18.35f;
public const float Ds0 = 19.45f;
public const float E0 = 20.60f;
public const float F0 = 21.83f;
public const float Fs0 = 23.12f;
public const float G0 = 24.50f;
public const float Gs0 = 25.96f;
public const float A0 = 27.50f;
public const float As0 = 29.14f;
public const float B0 = 30.87f;

#endregion

#region Tone Generation

/// <summary>
/// Retrive the musical note for the given enumeration at
/// the fundamental.
/// </summary>
public static MusicalNote GetNote(eMusicNote n) {
  for(int i=0;i<Notes.Length;i++) {
    if(Notes[i].Note == n)
      return Notes[i];
    }

  return MusicalNote.Empty;
  }

/// <summary>
/// Get harmonics for all notes
/// </summary>
/// <param name="s">The harmonic value of the notes to retrieve.</param>
/// <returns>A list of all notes at the given harmonic</returns>
public static MusicalDetect[] GetHarmonics(int s) {
  List<MusicalDetect> l = new List<MusicalDetect>();
  for(int i=0;i<Notes.Length;i++) l.Add(Notes[i].Harmonic(s));
  return l.ToArray();
  }

/// <summary>
/// Generate a random sound of the given duration in seconds.
/// </summary>
public static RandomSound RandomSound(int dur,int sampleRate, int floorAmp) {
  Synthesizer st = new Synthesizer(sampleRate);
  List<MusicalDetect> notes = new List<MusicalDetect>();
```

```
      int i = 0;
      float Nyquist = ((float)sampleRate)/2.0f;

      while(i < dur) {
        int step = 1 + ourRandom.Next(2);
        MusicalNote note = Notes[ourRandom.Next(Notes.Length)];
        int harm = ourRandom.Next(12);
        float freq = ((float)(note.Fundamental*Math.Pow(2.0f,harm--)));
        while(freq > Nyquist)
          freq = ((float)(note.Fundamental*Math.Pow(2.0f,--harm)));

        float amp = (float)(floorAmp + ourRandom.NextDouble()*10000.0f);

        for(int k=0;k<step;k++) {
          MusicalDetect md = new MusicalDetect(note,
                                               freq,
                                               amp,
                                               (int)((i+k)*1000.0f));
          st.Add(md.Rf,md.Amp,1);
          notes.Add(md);
          }

        i += step;
        }

      return new RandomSound(sampleRate, st.ToArray(),notes.ToArray());
      }

    /// <summary>
    /// Utility method to generate a bogus chord from Z12 vales.
    /// </summary>
    /// <returns>A chord of fundamental tones</returns>
    public static Chord MakeChord(int[] values) {
      Chord ch = new Chord();

      for(int i=0;i<values.Length;i++) {
        MusicalNote n = Music.GetNote(Music.PitchScale(values[i]));
        ch.Add(new MusicalDetect(n,n.Fundamental,10,0));
        }

      return ch;
      }


    #endregion

    #region Color Mapping

    /// <summary>
    /// Provides a color mapped from the given frequency value.
    /// </summary>
    /// <param name="inTiggerH">The tigger harmonic value</param>
    /// <param name="noise">The noise sample, a value from 0 to 1 with 1 being max noise</param>
    public static Color TiggerStripes(MusicalDetect inTone, float noise) {
      Color Na = Color.FromArgb(ourTiggerColorMap[(int)inTone.Note.TiggerHarmonic]);
      float t = noise;
      float Ha = ((((float)inTone.Harmonic)+1f)/12.0f) % 1.00001f;
      Color Fa;
      Color g = Color.FromArgb(128,128,128);

      if(inTone.Rf <= 50)
        Fa = Color.Black;
      else if(inTone.Rf >= 50 && inTone.Rf <= 700)
        Fa = Color.White;
      else
        Fa = Color.Yellow;

      // N(a) \diamond F(a)
      //--------------------
      //          H(a)
      // However, we use the intensity as the harmonic in HSV
      Color left = Fa.Add(Na);
      left = ExtraColor.HSVtoRGB(left.GetHue(),(1f-Ha),Ha);

      return left.Mul(1.0f-t).AddRGB(g.Mul(t));
      }

    #endregion

    }

#region Additional Structures/Classes
```

```
/// <summary>
/// A musical note.
/// </summary>
public enum eMusicNote {
  C = 0,
  Cs = 1,
  D = 2,
  Ds = 3,
  E = 4,
  F = 5,
  Fs = 6,
  G = 7,
  Gs = 8,
  A = 9,
  As = 10,
  B = 11,
  Unknown = 100,
  }

/// <summary>
/// Utilty class that contains a set of PCM and the corresponding notes.
/// </summary>
public struct RandomSound {
  public Z2[] Samples;
  public MusicalDetect[] Notes;
  public int SampleRate;

  /// <summary>
  /// Create a random sound.
  /// </summary>
  public RandomSound(int sampleRate, Z2[] samples, MusicalDetect[] det) {
    Samples = samples;
    Notes = det;
    SampleRate = sampleRate;
    }

  }

#endregion
}
```

A.0.8 Chord Class

```
  using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace Toolkit.Media.DSP {

  /// <summary>
  /// Provides a structure that manages a set of notes defining some
  /// musical chord.
  /// </summary>
  public class Chord : List<MusicalDetect> {

    public static int OWNS = -2;
    public static int NONE = -1;

    #region Constructor
    public Chord() {}
    #endregion

    #region Utility Methods

    /// <summary>
    /// Ask if the given musical detect matches something in this chord.
    /// Match Criteria:
    ///
    /// 1 Same note, same harmonic
    /// 2 Same time duration.
    /// </summary>
    /// <returns>
    /// Chord.NONE (No Match)
    /// Chord.OWNS (Belongs to chord, but no direct match)
    /// index (The index of matched note)
    /// The matched detection index or -1</returns>
    public int BelongsTo(MusicalDetect md) {

      // First see if matching note.
      for(int i=0;i<Count;i++) {
        MusicalDetect tst = this[i];

        if(tst.Note.Note == md.Note.Note &&
           tst.Harmonic == md.Harmonic)
          return i;
        }

      // Now see if times overlap.
      for(int i=0;i<Count;i++) {
        MusicalDetect tst = this[i];

        if(md.TimeOn >= tst.TimeOn &&
           md.TimeOn <= tst.TimeOn+tst.Duration)
          return OWNS;
        }

      return NONE;
      }

    /// <summary>
    /// Returns this chord with all notes raised by
    /// the given harmonic.
    /// </summary>
    public Chord GetHarmonic(int h) {

      Chord ch = new Chord();
      for(int i=0;i<Count;i++) {
        MusicalDetect cur = this[i];
        MusicalNote mn = Music.GetNote(cur.Note.Note);
        float newRf = (float)(mn.Fundamental*Math.Pow(2.0f,cur.Harmonic+h));
        ch.Add(new MusicalDetect(mn,newRf,cur.Amp,cur.TimeOn));
        }

      return ch;
      }

    #endregion
```

```
  #region Properties
  #endregion

  public override string ToString() {
    StringBuilder sb= new StringBuilder();
    sb.Append("{");
    for(int i=0;i<Count;i++) {
      if(i != 0) sb.Append(",");
      sb.Append("(" + this[i].TimeOn + "-" + this[i].Duration + ")" + this[i].Note.Note + "" +
           this[i].Harmonic);
      }
    sb.Append("}");
    return sb.ToString();
    }
  }

}
```

A.0.9 Chord Detection Class

```
  using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace Toolkit.Media.DSP {

  /// <summary>
  /// Provides a chord detection system that keeps track of a chord's structure over
  /// time as new notes are added.  Although there is only one chord supplied we
  /// provide an array of chords for expansion into later implementations.
  /// </summary>
  public class ChordDetector {

    #region Private

    // The current time step in units of myMinDuration.
    // incremented on each call to New Notes.
    private long myTimeStep = 0;
    private List<Chord> myChords = new List<Chord>();
    private int myRecent = -1;
    #endregion

    #region Algorithm

    /// <summary>
    /// Assumption: Let n be any existin gnote and T is the time onset and D
    /// is the last known duration of a note. Step Time is a counter that holds each
    /// interval of minimum duration that has passed.
    ///  1 If T(n) + D(n) &lt Step Time for any note, remove that note.
    ///  2 If all notes expired Kill Chord:.
    /// </summary>
    private void Expiration() {

      for(int i=0;i<myChords.Count;i++) {
        Chord c = myChords[i];

        for(int j=0;j<c.Count;j++) {
          MusicalDetect md = c[j];

          // Check to see if
          long expireTime = md.TimeOn + md.Duration;

          if(expireTime <= myTimeStep) {
            c.RemoveAt(j);
            j--;
            }
          }

        if(c.Count == 0) {
          myChords.RemoveAt(i);
          i--;
          }
        }

      }

    #endregion

    #region Constructors

    /// <summary>
    /// Create the chord detection.
    /// </summary>
    /// <param name="sampleRate">The sample rate of the original data</param>
    /// <param name="minimumDuration">The minimum duration of a chord's life in milliseconds </
        param>
    public ChordDetector() {}

    #endregion

    #region Utility

    /// <summary>
    /// Add a new note to the chord system.
```

```
/// </summary>
public void NewNotes(MusicalDetect[] newNotes) {

  // Refresh all new notes that already exist in chords.
  for(int i=0;i<newNotes.Length;i++) {
    MusicalDetect match = newNotes[i];
    newNotes[i] = match;

    bool found = false;

    // Refresh:
    for(int j=0;j<myChords.Count && !found;j++) {
      Chord ch = myChords[j];
      int foundIdx = ch.BelongsTo(match);

      // If the note belongs to the chord.
      if(foundIdx != Chord.NONE) {

        // This note belongs to the chord in time but that note isn't there.
        if(foundIdx == Chord.OWNS) {
          ch.Add(match);
          myRecent = j;
          found=true;
          }
        // The note is in the chord already.
        else {
          MusicalDetect md = ch[foundIdx];
          md.Duration += match.Duration;
          ch[foundIdx] = md;
          myRecent = j;
          found=true;
          }
        }
      }

    // Residuals:
    if(!found) {
      Chord ch = new Chord();
      ch.Add(match);
      myChords.Add(ch);
      myRecent = myChords.Count-1;
      }
    }

  Expiration();

  for(int i=0;i<newNotes.Length;i++) {
    if(newNotes[i].TimeOn > myTimeStep) myTimeStep = newNotes[i].TimeOn;
    }
  }

/// <summary>
/// Get all currently held chords.
/// </summary>
public Chord[] GetChords() { return myChords.ToArray(); }

/// <summary>
/// Remove all chords.
/// </summary>
public void Clear() {
  myChords.Clear();
  myRecent = -1;
  }

#endregion

#region Properties

/// <summary>
/// Get the most recently seen chord.
/// </summary>
/// <value>null if no chord exists.</value>
public Chord Latest {
  get {
    if(myRecent == -1) return null;
    return myChords[myRecent];
    }
  }

#endregion

public override string ToString() {
  StringBuilder sb = new StringBuilder();
```

```
        for(int i=0;i<myChords.Count;i++) sb.Append(myChords[i].ToString() + "\r\n");
        return sb.ToString();
        }

    }

}
```

A.0.10 Sound Processing Bus Class

```
  using System;
using System.IO;
using System.Threading;
using System.Collections.Generic;
using System.Linq;
using System.Text;

using Core.IO;
using Core.Collections;
using Core.Forms;
using Core.Util;
using Core.Mathematics.Algebra;

using NAudio.Wave;
using Toolkit.Media.DSP;
using System.Diagnostics;

namespace Toolkit.Media.DSP {

  /// <summary>
  /// Identifies the current state of the sound bus
  /// </summary>
  public enum eSoundBusPhase {

    /// <summary>
    /// The bus played the entire stream or
    /// something was killed, or an error occured.
    /// </summary>
    PlayEnded,

    /// <summary>
    /// The sound bus has received a PCM stream and is starting play
    /// </summary>
    PlayStarted,

    /// <summary>
    /// The sound bus has received a request to pause play.
    /// </summary>
    PlayPaused,

    }

  //~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
  /// <summary>
  /// Delegate notified of the current sound bus phase
  /// </summary>
  /// <param name="phase"></param>
  //~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
  public delegate void dSoundBusPhase(eSoundBusPhase phase);

  //~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
  /// <summary>
  /// Delegate that is notified of clock ticks as the soundbus processes data.
  /// </summary>
  /// <param name="tineNs">The current processing time in Nanoseconds</param>
  /// <param name="step">The definition of this tick step</param>
  //~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
  public delegate void dSoundBusTick(DSPStatistics timeNs, eTickStep step);

  //~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
  /// <summary>
  /// Delegate notified of each read sample.
  /// </summary>
  //~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
  public delegate void dSampleNotice(Z2[] inSamp);

  //~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
  /// <summary>
  /// Provides a foundation utilty class that can read input files or
  /// from a microphone and play's to the speaker as well as
  /// forwarding raw data to processors for any purpose.
  ///
  /// Note:  This class is threaded.
  /// </summary>
  //~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

```
public class SoundBus {

  #region Private Members

  private bool myPlayForever = false;                                    // If true then we suck in PCM
       forever (a microphone source or something)
  private ePlayerState myState = Core.Forms.ePlayerState.None;
  private DSPStatistics myTimeStats = new DSPStatistics();
  private Object myMutex = new Object();
  private SoundBusPluginQueue myPlugins = new SoundBusPluginQueue();
  private Thread myPlayThread;
  private Stream myInputPCMStream;
  private PCMInfo myInfo;

  // Used for reading a sample 1 at a time.
  private byte[] myWorkspaceBuffer;
  // Provider of wave data to sound card
  private BufferedWaveProvider myProvider = null;
  // Total number of bytes read from input PCM stream.
  private double myProcessedBytes = 0;
  // Thread use only for stopping.
  private bool myThIsPlaying = false;

  #endregion

  #region Event Methods

  private void FirePhaseChange(eSoundBusPhase inPh) {
    if(PhaseChanged != null)
      PhaseChanged(inPh);
    }

  private void FireProgressChange(String inMsg, int min, int max, int val) {
    if(ProgressChanged != null)
      ProgressChanged(inMsg,min,max,val);
    }

  private void FireTick(DSPStatistics timeNs, eTickStep step) {
    try {
      if(Tick != null) Tick(timeNs,step);
      }
    catch(Exception ex) { Log.Error(ex); }
    }

  private void FireSampleRead(Z2[] samp) {
    if(SampleRead != null)
      SampleRead(samp);
    }

  #endregion

  #region Threads

  //~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
  /// <summary>
  /// This will attempt to buffer a good number of samples for playing.
  /// If we are a wave file input (myPlayForever=false) then
  /// buffer enough data to the card to match up. Otherwise just grab
  /// one sample at a time.
  /// </summary>
  /// <param name="input">How many seconds to buffer.</param>
  /// <returns>true if something to process</returns>
  //~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
  private bool BufferSeconds(Stream inStream, ref Z2[] outSamples, int sec) {
    int neededSamples = (int)(myInfo.SampleRate*sec);

    // Re-shape temp buffer if needed.
    if(myWorkspaceBuffer == null || myWorkspaceBuffer.Length < neededSamples)
      myWorkspaceBuffer = new byte[(int)(myInfo.SampleSize*myInfo.SampleRate)];

    Array.Clear(outSamples,0,outSamples.Length);
    Array.Clear(myWorkspaceBuffer,0,myWorkspaceBuffer.Length);

    int r = 0;
    int dr = 0;
    int offset = 0;
    int bytesToRead = (int)(myInfo.SampleRate*myInfo.SampleSize);
    int total = 0;

    if(myWorkspaceBuffer.Length < bytesToRead) throw new ArgumentException("Not enough buffer
         space in 'inSpace'");
    if(outSamples.Length < myInfo.SampleRate*sec) throw new ArgumentException("Not enough buffer
         space in 'inTo'");
```

```
  // Read n seconds of data.
  for(int i=0;i<sec;i++) {

    // Read the amount we need
    while(r < bytesToRead) {
      dr = inStream.Read(myWorkspaceBuffer,r, bytesToRead-r);
      if(dr == 0) break;
      r += dr;
      }

    if(r > 0) {
      myProvider.AddSamples(myWorkspaceBuffer,0,r);
      MediaUtils.CreateSamples(ref outSamples,myWorkspaceBuffer,0,(int)myInfo.SampleRate,
          offset,myInfo);
      myTimeStats.AddA(outSamples,0, (int)(r / myInfo.SampleSize));
      myProcessedBytes += r;
      }

    total += r;
    r = 0;
    offset += (int)myInfo.SampleRate;
    }

  return total > 0;
  }

//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
/// <summary>
/// The main thread that plays the current audio stream.
/// </summary>
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
private void PlayCorrectlyThread() {
  myState = ePlayerState.Playing;
  WaveOut wavePlayer = null;

  try {
    GameTime gt = new GameTime();
    myThIsPlaying = true;
    wavePlayer = new WaveOut();
    ((WaveOut)wavePlayer).DesiredLatency = (int)SoundParameters.LATENCY;
    myState = Core.Forms.ePlayerState.Playing;
    FirePhaseChange(eSoundBusPhase.PlayStarted);
    myProvider = new BufferedWaveProvider(myInfo.ToWaveFormat());
    wavePlayer.Init(myProvider);
    wavePlayer.Play();

    // Add padding to match block alignment.
    myTimeStats.Clear();
    myTimeStats.ResetInterval = (int)myInfo.SampleRate;

    // Initialize all the plugins with the wave information.
    for(int k=0;k<myPlugins.Length;k++) myPlugins[k].Initialize(this, myTimeStats,myInfo);

    bool dataAvail = false;
    double myTotalBytes = (myPlayForever)?double.MaxValue:myInputPCMStream.Length;
    FireProgressChange("Processing Signal ...",0,0,0);
    DateTime nextStatus = DateTime.UtcNow.Add(new TimeSpan(0,0,3));
    int bufferedTimeS = 1;
    float targetPaintInterval = (1f/60f)*1000f;
    float paintInterval = targetPaintInterval;
    float paintCnt = 0;

    // Do 1 second buffer.
    Z2[] _1Secondbuffer = new Z2[(int)myInfo.SampleRate];

    while(true) {

      if (myState == Core.Forms.ePlayerState.Paused) {
        while(myState == Core.Forms.ePlayerState.Paused) Thread.Sleep(50);
        }

      if (myState == Core.Forms.ePlayerState.Stopped) break;

      // We have real-time mic mode and playing from file mode
      // This is for playing from a file mode.  We need to keep the sound card in sync.
      if(!myPlayForever) {

        if(myProvider.BufferedDuration < new TimeSpan(0,0,0,1,300))
          dataAvail = BufferSeconds(myInputPCMStream,ref _1Secondbuffer,bufferedTimeS);

        // No data so stop.
        if(!dataAvail && myProvider.BufferedBytes == 0) break;
```

```
        }
      else
        dataAvail = BufferSeconds(myInputPCMStream, ref _1Secondbuffer, bufferedTimeS);

      // Update game time.
      gt.UpGame(1000);

      // Did we read anything new.
      if(dataAvail) {
        myPlugins.Add(_1Secondbuffer);
        FireSampleRead(_1Secondbuffer);
        }

      // Wait for real time to catch up.
      while(myProvider.BufferedDuration > new TimeSpan(0,0,0,0,300)) {
        FireTick(myTimeStats, eTickStep.FPS30_Tick);
        paintCnt++;
        Thread.Sleep((int)paintInterval);
        }

      // Status/progress update.
      if(DateTime.UtcNow > nextStatus) {
        int p = (int)((myProcessedBytes/myTotalBytes)*100.0d);
        FireProgressChange("Processing Signal ...",0,100,p);

        paintInterval = paintInterval*(paintCnt/180f);
        if(paintInterval > targetPaintInterval)
          paintInterval = targetPaintInterval;
        else if(paintInterval == 0) paintInterval = 1f;

        nextStatus = DateTime.UtcNow.Add(new TimeSpan(0,0,3));
        }

      while(gt.Schedule() > 0) {
        FireTick(myTimeStats, eTickStep.FPS30_Tick);
        paintCnt++;
        Thread.Sleep(1);
        }

      gt.Clear();
      }

    }
  catch(ThreadInterruptedException) {}
  catch(ThreadAbortException) {}
  catch(IOException ex) {
    Log.Status(ex.ToString());
    if(ProgressChanged != null) ProgressChanged("PCM Stream ended " + ex.Message,0,0,0);
    }
  catch(Exception ex) {
    Log.Error(ex);
    if(ProgressChanged != null) ProgressChanged("Yipes! Something went wrong " + ex.Message
        ,0,0,0);
    }
  finally {
    myState = ePlayerState.Stopped;

    try {
      if(myInputPCMStream != null) myInputPCMStream.Close();
      if(wavePlayer != null) wavePlayer.Dispose();
      myInputPCMStream = null;
      myWorkspaceBuffer = null;
      myProcessedBytes = 0;
      myProvider = null;
      myPlugins.ClearPlugins();
      }
    catch(Exception) {}
    myThIsPlaying = false;

    try {
      if(PhaseChanged != null) PhaseChanged(eSoundBusPhase.PlayEnded);
      }
    catch(Exception ex) {
      Log.Error(ex);
      }
    }

  }

#endregion

#region Constructors
```

```
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
/// <summary>
/// Destructor
/// </summary>
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~SoundBus() {
  Dispose();
  }

//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
/// <summary>
/// Create a sound bus.
/// </summary>
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
public SoundBus() { }

#endregion

#region Plugin Actions

//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
/// <summary>
/// Add a bus plugin to the sound bus for messaging.
/// </summary>
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
public void Add(ISoundBusPlugin plugin) {
  if(plugin == null) throw new NullReferenceException("Can't add a null plugin");
  if(myState == ePlayerState.Playing) throw new SystemException("Can't add a plugin while the
      bus is playing");

  myPlugins.Add(plugin);
  }

//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
/// <summary>
/// Query for the plugin by the specified name.
/// </summary>
/// <param name="name">The human readable name of this plugin</param>
/// <returns>null if not found</returns>
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
public ISoundBusPlugin FindPlugin(String name) {

  lock(myMutex) {
    for(int i=0;i<myPlugins.Length;i++) {
      if(myPlugins[i].PluginName != null && myPlugins[i].PluginName == name)
        return myPlugins[i];
      }
    }

  return null;
  }

#endregion

#region Playback

//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
/// <summary>
/// Begin playing of the given audio file.
/// Supported .wav, .mp3
/// </summary>
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
public void Play(FilePath inFile) {
  PCMStreamSource src = new PCMStreamSource(new FilePath(inFile));
  Stream str = src.CreateStream();
  Play(src.Format,str);
  }

//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
/// <summary>
/// Begin the playing of the given audio stream.
/// </summary>
/// <param name="info">The PCM content of the given stream</param>
/// <param name="pcmAudioStream">The stream to read PCM data from</param>
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
public void Play(PCMInfo info, Stream pcmAudioStream) {

  try {
    Log.Status("Starting Play Thread ...");

    // Try to determine the stream source
    if(pcmAudioStream is MicrophoneStream)
```

```
      myPlayForever = true;
    else
      myPlayForever = false;

    ExtraThread.Kill(ref myPlayThread);

    myInputPCMStream = pcmAudioStream;
    myInfo = info;

    myPlayThread = new Thread(PlayCorrectlyThread);
    myPlayThread.IsBackground=true;
    myPlayThread.Name = "Playback Thread";
    myPlayThread.Start();
    }
  catch(ThreadAbortException) {
    Log.Status("Play thread aborted ...");
    ExtraThread.Kill(ref myPlayThread);
    }
  catch(ThreadInterruptedException) {
    Log.Status("Play thread interrupted ...");
    ExtraThread.Kill(ref myPlayThread);
    }
  catch(Exception ex) { Log.Error(ex); }
  }

//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
/// <summary>
/// Pause playback
/// </summary>
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
public void Pause() {
  myState = Core.Forms.ePlayerState.Paused;
  if(PhaseChanged != null) PhaseChanged(eSoundBusPhase.PlayPaused);
  }

//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
/// <summary>
/// Stop playback
/// </summary>
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
public void Stop() {
  myState = ePlayerState.Stopped;
  while(myThIsPlaying) Thread.Yield();
  }

//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
/// <summary>
/// Resume playback.
/// </summary>
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
public void Resume() {
  myState = Core.Forms.ePlayerState.Playing;
  if(PhaseChanged != null) PhaseChanged(eSoundBusPhase.PlayStarted);
  }

#endregion

#region Cleanup

//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
/// <summary>
/// Clear all plugins and ready for another run.
/// </summary>
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
public void Clear() {

  lock(myMutex) {
    ExtraThread.Kill(ref myPlayThread);
    myWorkspaceBuffer = null;
    myProcessedBytes = 0;
    myProvider = null;
    myInputPCMStream = null;
    }
  }

//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
/// <summary>
/// Dispose the Bus
/// </summary>
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
public void Dispose() {

  try {
```

```
      Stop();
      }
    catch(Exception ex) {}

    }

#endregion

#region Properties

//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
/// <summary>
/// Request access to the plugins installed in the sound bus.
/// (Never modify this array)
/// </summary>
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
public ISoundBusPlugin[] Plugins {
  get {
    return myPlugins.ToArray();
    }
  }

//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
/// <summary>
/// Get the statistics being gathered on each read item.
/// </summary>
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
public DSPStatistics TimeStats { get { return myTimeStats; } }

//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
/// <summary>
/// Get the stream encoding information.
/// </summary>
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
public PCMInfo StreamInfo { get { return myInfo; }}

//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
/// <summary>
/// Request the current status of the sound bus.
/// </summary>
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
public ePlayerState Status { get { return myState; }}

/// <summary>
/// Get the sound bus plugin queue that processes data and sends to plugins.
/// </summary>
public SoundBusPluginQueue Queue { get { return myPlugins; }}

#endregion

#region Events

//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
/// <summary>
/// Event notified of the current state.
/// </summary>
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
public event dSoundBusPhase PhaseChanged;

//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
/// <summary>
/// Event that is notified of progress.
/// </summary>
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
public event dProgressInfo ProgressChanged;

//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
/// <summary>
/// Event that is notified of tick steps along the way during processing
/// of data.
/// </summary>
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
public event dSoundBusTick Tick;

//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
/// <summary>
/// This event is for non ISoundBusPlugins that wish to be notified
/// whenever a sample is read without all the other functions.
/// </summary>
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
public event dSampleNotice SampleRead;

#endregion
```

```
    }

}
```

A.0.11 Media Utilities Class

```
  using System;
using System.IO;
using System.Collections.Generic;
using System.Linq;
using System.Text;

using Core.IO;
using Core.Util;
using Core.Forms;
using Core.Mathematics.Algebra;
using NAudio.Wave;

namespace Toolkit.Media.DSP {

  //~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
  /// <summary>
  /// Various utilities for dealing with sound/video/images.
  /// </summary>
  //~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
  public abstract class MediaUtils {

    #region Private Utility

    private static List<FilePath> ourCreatedTempFiles = new List<FilePath>();
    private static bool ourIsCleaned = false;

    #endregion

    #region Calculations

    //~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    /// <summary>
    /// Get the number of bytes in a sample chunk
    /// </summary>
    /// <param name="bps">Bits per sample</param>
    /// <param name="channels">Number of channels</param>
    /// <param name="blockAlign">Block alignment</param>
    /// <returns>The number of bytes in a sample</returns>
    //~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    public static int GetSampleSize(int bps, int channels, int blockAlign) {
      return new PCMInfo(bps,channels,0,blockAlign).SampleSize;
      }

    #endregion

    #region I/O

    //~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    /// <summary>
    /// Given a buffer of raw PCM data read samples for a specific channel.
    /// </summary>
    /// <param name="numSamps">The total number of samples to read</param>
    /// <param name="outRec">Receives by reference the read sample data.</param>
    /// <param name="buf">The buffer to read from</param>
    /// <param name="offset">The starting offset to read from buf</param>
    /// <param name="info">The stream infor</param>
    /// <param name="size">The number of samples to read</param>
    /// <param name="channel">The channel to read from</param>
    /// <returns>An array of the samples read.</returns>
    //~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    public static void CreateSamples(ref Z2[] outRec, byte[] buf, int channel, int numSamps, int
        offset, PCMInfo info) {
      if(outRec.Length < numSamps) throw new ArgumentException("Sample buffer is not large enough!
          ");

      for(int i=0;i<numSamps;i++) {
        PCMSample s = CreateSample(buf,offset,info);

        switch(channel) {
          case 0:
            outRec[i].A = s.Amp1;
            break;
          case 1:
            outRec[i].A = s.Amp2;
            break;
```

```
      case 2:
        outRec[i].A = s.Amp3;
        break;
      case 3:
        outRec[i].A = s.Amp4;
        break;
      case 4:
        outRec[i].A = s.Amp5;
        break;
      }

    offset += info.SampleSize;
    }

  }

//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
/// <summary>
/// Given a complex valued impulse in the time domain create a PCM sample.
/// </summary>
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
public static byte[] CreatePCM(Z s, PCMInfo info) {
  int b = (info.BitsPerSample/8);            // Bytes per sample
  byte[] a = new byte[info.SampleSize*info.Channels];

  for(int i=0;i<info.Channels;i++) {
    if(b == 1) {
      byte b1 = (byte)((int)s.Magnitude);
      a[i*info.SampleSize] = b1;
      }
    else if(b == 2) {
      byte[] bytes = BitConverter.GetBytes((int)s.Magnitude);
      a[i*info.SampleSize] = bytes[1];
      a[i*info.SampleSize + 1] = bytes[0];
      }
    }

  return a;
  }

//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
/// <summary>
/// Create a PCM sample from a buffer that contains a PCM encoded sample starting at
/// the given offset.
/// </summary>
/// <param name="buf">The buffer to read from</param>
/// <param name="info">Info about the PCM encoding</param>
/// <returns>The sample read</returns>
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
public static PCMSample CreateSample(byte[] buf, int offset, PCMInfo info) {
  int b = (info.BitsPerSample/8);        // Bytes per sample

  PCMSample pc = new PCMSample();

  for (int i=0;i<info.Channels;i++) {
    int k = offset + b*i;
    int valread = 0;

    if(b == 2) {
      int val = buf[k+1];
      val = val<<8;
      val = val | buf[k];
      valread = (short)val;
      if(valread > Int16.MaxValue) valread = Int16.MaxValue;
      else if(valread < -Int16.MaxValue) valread = -Int16.MaxValue;
      }
    else if(b == 1) {
      valread = (short)buf[k];
      if(valread > Int16.MaxValue) valread = Int16.MaxValue;
      else if(valread < -Int16.MaxValue) valread = -Int16.MaxValue;
      }

    // Depending on which channel we've read save the value off.
    switch(i) {
      case 0:
        pc.Amp1 = valread;
        break;
      case 1:
        pc.Amp2 = valread;
        break;
      case 2:
        pc.Amp3 = valread;
        break;
```

```
      case 3:
        pc.Amp4 = valread;
        break;
      case 4:
        pc.Amp5 = valread;
        break;
      }

    }

  return pc;
  }

#endregion

#region Generation Utilities

//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
/// <summary>
/// Fabricate a PCM stream of data that is probably just noise but
/// PCM formatted as data chunks
/// </summary>
/// <param name="bps">Bits per second</param>
/// <param name="sampleRate">Sample rate</param>
/// <param name="blockAlign">Block alignment</param>
/// <param name="channels">Channels</param>
/// <param name="numSamples">The number of samples to generate</param>
/// <returns>The raw data array of data chunks</returns>
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
public static byte[] CreateRandomPCM(PCMInfo pi, int numSamples) {
  int bps = pi.BitsPerSample;
  float sampleRate = pi.SampleRate;
  int blockAlign = pi.BlockAlign;
  int channels = pi.Channels;

  if((bps/8)*channels > blockAlign) throw new ArgumentException("Block alignment must be
      greater than or equal to bits per sample and channels");

  // Add padding to match block alignment.
  int sampleSize = channels*(bps/8) +
                   blockAlign - channels*(bps/8);

  byte[] buf = new byte[sampleSize];
  MemoryStream ms = new MemoryStream();

  for(int i=0;i<numSamples;i++) {
    Random r = new Random();

    for(int j=0;j<buf.Length;j++) {
      if(r.Next(100) > 50)
        buf[j] = (byte)r.Next(char.MaxValue/2);
      else
        buf[j] = (byte)-r.Next(char.MaxValue/2);
      }

    ms.Write(buf,0,buf.Length);
    }

  return ms.ToArray();
}

//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
/// <summary>
/// Create a sample stream assuming the given data is
/// a) Single Channel
/// b) 16 bits per sample
/// c) 40.1 Khz Sample Rate
/// </summary>
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
public static PCMSamplerMemoryStream CreateSingleChanne16BPS(int[] channel1) {
  PCMInfo pi = new PCMInfo(16,1,40100,2);
  PCMSamplerMemoryStream ss = new PCMSamplerMemoryStream(pi);

  for(int i=0;i<channel1.Length;i++)
    ss.Write(new PCMSample(channel1[i]));

  return ss;
  }

//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
/// <summary>
/// Create a sample stream assuming the given data is
/// a) Dual Channel
```

```
/// b) 16 bits per sample
/// c) 40.1 Khz Sample Rate
/// </summary>
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
public static PCMSamplerMemoryStream CreateDualChannel6BPS(int[] channel1, int[] channel2) {
  PCMInfo pi = new PCMInfo(16,2,40100,4);
  PCMSamplerMemoryStream ss = new PCMSamplerMemoryStream(pi);

  for(int i=0;i<channel1.Length;i++)
    ss.Write(new PCMSample(channel1[i],channel2[i]));

  return ss;
  }


//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
/// <summary>
/// Create a sample stream assuming the given data is
/// a) Single Channel
/// b) 8 bits per sample
/// c) 40.1 Khz Sample Rate
/// </summary>
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
public static PCMSamplerMemoryStream CreateSingleChannel8BPS(int[] channel1) {
  PCMInfo pi = new PCMInfo(8,1,40100,1);
  PCMSamplerMemoryStream ss = new PCMSamplerMemoryStream(pi);

  for(int i=0;i<channel1.Length;i++)
    ss.Write(new PCMSample(channel1[i]));

  return ss;
  }

//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
/// <summary>
/// Create a sample stream assuming the given data is
/// a) Dual Channel
/// b) 8 bits per sample
/// c) 40.1 Khz Sample Rate
/// </summary>
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
public static PCMSamplerMemoryStream CreateDualChanne8BPS(int[] channel1,int[] channel2) {
  PCMInfo pi = new PCMInfo(8,2,40100,2);
  PCMSamplerMemoryStream ss = new PCMSamplerMemoryStream(pi);

  for(int i=0;i<channel1.Length;i++)
    ss.Write(new PCMSample(channel1[i],channel2[i]));

  return ss;
  }

#endregion

#region Stream Creation

//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
/// <summary>
/// Given a stream source data object this creates the stream and
/// returns the stream and populates the format information in the
/// given source object.
/// </summary>
/// <returns>null if unable to find the stream location of a PCM stream</returns>
/// <param name="source">The source information for the stream desired</param>
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
public static Stream CreateStream(PCMStreamSource source) {

  try {

    // Input from a selected file.
    switch(source.SourceType) {

      #region File Source
      case ePCMSourceType.WaveFile:

        if(!source.FilePath.Exists) {
          StatusControl.Instance.StatusBlue = "You must select a file to read from!";
          return null;
          }

        StatusControl.Instance.StatusBlue = "Loading file ...";

        WaveFileReader input = null;
```

```
            if(source.FilePath.Extension.ToLower() == "mp3") {
              Mp3FileReader mpr = new Mp3FileReader(source.FilePath.ToString());
              StatusControl.Instance.StatusBlue = "Decoding file ...";
              FilePath myCreatedFile = FilePath.CreateTempFile();
              lock(ourCreatedTempFiles) {
                ourCreatedTempFiles.Add(myCreatedFile);
                }
              WaveFileWriter.CreateWaveFile(myCreatedFile.ToString(),mpr);
              mpr.Close();
              input = new WaveFileReader(myCreatedFile.ToString());
              }
            else
              input = new WaveFileReader(source.FilePath.ToString());

            source.Format = new PCMInfo(input.WaveFormat);
            return input;
          #endregion

          #region Micropohone

          case ePCMSourceType.Microphone:
            MicrophoneStream ms = new MicrophoneStream(source.Microphone);
            ms.Start();
            source.Format = ms.OutputInfo;
            return ms;

          #endregion

          }

        }
      catch(Exception ex) {
        StatusControl.Instance.StatusRed = ex.Message;
        Log.Error(ex);
        }

      return null;
      }

    #endregion

    //~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    /// <summary>
    /// Should be called on app shutdown to cleanup media artifacts and system
    /// resources.
    /// </summary>
    //~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    public static void Shutdown()  {

      try {

        if(!ourIsCleaned) {
          ourIsCleaned=true;

          lock(ourCreatedTempFiles) {
            foreach(FilePath fp in ourCreatedTempFiles) fp.Delete();
            }

          }
        }
      catch(Exception ex) {}
      }
    }

}
```

A.0.12 PCM Info Class

```
  using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

using NAudio.Wave;

namespace Toolkit.Media.DSP {

  //~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
  /// <summary>
  /// Stucture that holds Pulse Code Modulated data
  /// </summary>
  //~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
  public struct PCMInfo {

    #region Constant

    public static readonly PCMInfo Empty = new PCMInfo(-1,-1,-1,-1);

    #endregion

    #region Members

    /// <summary>
    /// Bits per sample
    /// </summary>
    public int BitsPerSample;

    /// <summary>
    /// Number of channels
    /// </summary>
    public int Channels;

    /// <summary>
    /// The sample rate in (Hz)
    /// </summary>
    public float SampleRate;

    /// <summary>
    /// The Maximum size of a sample chunk including each channel.
    /// Block alignment states what boundry the bytes should end on when
    /// encompassing each bits per sample and each channel being of the bits
    /// per sample.
    ///
    /// <example>
    ///                     Block Alignment (2 bytes)
    ///                           |
    ///                           v
    /// ┌──────────┬──────────┬──────────┬──────────┐
    /// |  Byte 0  | Byte 1   |  Byte 2  |   Byte 3    |
    /// ├──────────┴──────────┼──────────┴──────────┤
    /// |       Time 0        |         Time 1          |
    /// ├──────────┬──────────┼──────────┬──────────┤
    /// |  Ch 1    |   Ch 2   |    Ch 1  |   Ch 2      |
    /// └──────────┴──────────┴──────────┴──────────┘
    ///
    /// (Where Ch is the channel)
    ///
    /// </example>
    /// </summary>
    public int BlockAlign;

    #endregion

    #region Constructors

    //~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    /// <summary>
    /// Make a copy
    /// </summary>
    //~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    public PCMInfo(PCMInfo other) {
      BitsPerSample = other.BitsPerSample;
      Channels = other.Channels;
```

```
    SampleRate = other.SampleRate;
    BlockAlign = other.BlockAlign;
    }

  //~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
  /// <summary>
  /// Create a PCM Info from a wave format
  /// </summary>
  //~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
  public PCMInfo(WaveFormat fmt) {
    BitsPerSample = fmt.BitsPerSample;
    Channels = fmt.Channels;
    SampleRate = fmt.SampleRate;
    BlockAlign = fmt.BlockAlign;
    }

  //~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
  /// <summary>
  /// Create a PCMInfo from the values
  /// </summary>
  /// <param name="bps">Bits per sample</param>
  /// <param name="channels">Channels</param>
  /// <param name="sampleRate">Sample rate (Hz)</param>
  /// <param name="blockAlign">Block alignment</param>
  //~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
  public PCMInfo(int bps, int channels, float sampleRate, int blockAlign) {
    BitsPerSample = bps;
    Channels = channels;
    SampleRate = sampleRate;
    BlockAlign = blockAlign;
    }

  #endregion

  #region Utility Methods

  //~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
  /// <summary>
  /// Caluclate the sample size of a chunk (how many bytes per a single data sample)
  /// </summary>
  //~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
  public int SampleSize {
    get {
      return Channels*(BitsPerSample/8) + BlockAlign - Channels*(BitsPerSample/8);
      }
    }

  //~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
  /// <summary>
  /// Convert this to a wave format structure.
  /// </summary>
  //~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
  public WaveFormat ToWaveFormat() {
    return new PCMWaveFormat(this);
    }

  #endregion

  #region Internal Classes

  /// <summary>
  /// Utility class that alllows us to convert from a PCMInfo class to a
  /// WaveFormat class.
  /// </summary>
  public class PCMWaveFormat : WaveFormat {
    private PCMInfo myInfo;

    public PCMWaveFormat(PCMInfo info) : base((int)info.SampleRate,(int)info.BitsPerSample,info.
        Channels) {
      base.blockAlign = (short)info.BlockAlign;
      }

    }

  #endregion

  #region Proeprties

  //~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
  /// <summary>
  /// Ask if this is an empy instance of the PCMinfo structure
  /// </summary>
  //~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

```
    public bool IsEmpty {
      get {
        return BitsPerSample == -1 &&
               Channels == -1 &&
               SampleRate == -1 &&
               BlockAlign == -1;
      }
    }

  #endregion

  #region Object Overloads

  public override bool Equals(object obj) {
    PCMInfo other = (PCMInfo)obj;

    return BitsPerSample == other.BitsPerSample &&
           BlockAlign == other.BlockAlign &&
           Channels == other.Channels &&
           SampleRate == other.SampleRate;
    }

  public override int GetHashCode() {
    return ToString().GetHashCode();
    }

  public override string ToString() {
    return (SampleRate/1000f) + "kHz Ch: " + Channels + " BPS: "+ BitsPerSample;
    }

  #endregion

  #region Operator Overloads

  public static bool operator==(PCMInfo a, PCMInfo b) {
    return a.Equals(b);
    }

  public static bool operator!=(PCMInfo a, PCMInfo b) {
    return !a.Equals(b);
    }

  #endregion
  }

}
```

A.0.13 Melody Analysis

```
  using System;
using System.Drawing;
using System.Collections.Generic;
using System.Linq;
using System.Text;

using Core.Mathematics;
using Core.Mathematics.Stats;
using Core.Mathematics.Linear;
using Core.Mathematics.Algebra;

using Toolkit.Media.UI;

namespace Toolkit.Media.DSP {

  /// <summary>
  /// Provides a melody analysis system which computes a visual space
  /// from musical input.
  /// </summary>
  public class MelodyAnalysis {

    #region Private Members
    private PCMInfo myInfo;
    private Chord myLastCh;
    private ChordDetector myChordDetector = new ChordDetector();
    private RunningStatistics myStats;
    private float myMaxNoise = 0;

    #region Running Values

    // Min/Max values.
    private float myMaxa = -float.MaxValue;
    private float myMaxb = -float.MaxValue;
    private float myMaxc = -float.MaxValue;
    private float myMina = float.MaxValue;
    private float myMinb = float.MaxValue;
    private float myMinc = float.MaxValue;

    private double mySpectacle = 0;
    private double myN = 0;              // Total number of chords seen.
    // Centriciy stuff.
    private double myThetaH = 0;
    private double myThetaF = 0;

    #endregion

    #endregion

    #region Private Members

    /// <summary>
    /// Update the minimum and maximum range values.
    /// </summary>
    private void UpdateMinMax(float v1, float v2, float v3,
                              float f1,float f2, float f3) {

      myMina = Math.Min(myMina,v1);
      myMinb = Math.Min(myMinb,v2);
      myMinc = Math.Min(myMinc,v3);

      myMaxa = Math.Max(myMaxa,v1);
      myMaxb = Math.Max(myMaxb,v2);
      myMaxc = Math.Max(myMaxc,v3);

      myMina = Math.Min(myMina,f1);
      myMinb = Math.Min(myMinb,f2);
      myMinc = Math.Min(myMinc,f3);

      myMaxa = Math.Max(myMaxa,f1);
      myMaxb = Math.Max(myMaxb,f2);
      myMaxc = Math.Max(myMaxc,f3);
      }

    #endregion
```

```
#region Constructors

/// <summary>
/// Create the melody analysis with the time series statistics and the stream information.
/// </summary>
public MelodyAnalysis(RunningStatistics stats, PCMInfo info) {
  myInfo = info; myStats = stats;
  myMaxNoise = (float)(Math.Pow(2.0f,(float)info.BitsPerSample)/2f);
  }

private float myRunCDiff = 0;

/// <summary>
/// A value from [0,1] that describes "how much" the
/// system is dramatically changing chords over time.
/// </summary>
/// <param name="ch"></param>
/// <param name="ch2"></param>
/// <returns></returns>
private float CDiff(Chord ch, Chord ch2) {

  Comparison<MusicalDetect> s = delegate(MusicalDetect a, MusicalDetect b) {
    if(a.Note.Fundamental < b.Note.Fundamental)
      return -1;
    else if(a.Note.Fundamental > b.Note.Fundamental)
      return 1;

    return 0;
    };

  ch.Sort(s);
  ch2.Sort(s);

  Chord shorter;
  Chord longer;
  if(ch.Count > ch2.Count) {
    longer = ch;
    shorter = ch2;
    }
  else {
    longer = ch2;
    shorter = ch;
    }

  float sum = 0;
  for(int i=0;i<shorter.Count;i++) sum += (float)Math.Abs( (Music.Z12(ch[i].Note.Note)+1f) - (
      Music.Z12(ch2[i].Note.Note)+1f));
  for(int i=shorter.Count;i<longer.Count;i++) sum += Music.Z12(longer[i].Note.Note) + 1f;

  myRunCDiff = (float)(((myRunCDiff*myN) + sum/144f)/(myN+1f));
  return myRunCDiff;
  }

#endregion

/// <summary>
/// Provide a 1 second set of time samples.
/// </summary>
/// <param name="outBG">Receives the background that should be used</param>
/// <param name="X">The samples to process.</param>
public VisualSpace[] Analyze(Z2[] X) {
  MusicalDetect[] md = Music.GuessNotes(X,0,(int)myInfo.SampleRate);
  for(int i=0;i<md.Length;i++)  md[i].TimeOn = (int)myN;

  myN++;

  myChordDetector.NewNotes(md);

  Chord ch = myChordDetector.Latest;
  myChordDetector.Clear();

  // Calculate base values.
  float sigma = (float)(myStats.StdDeviation/(2.0f*Math.Pow(2f,myInfo.BitsPerSample)));
  float thetaF = Music.FundamentalCentricity(ch);
  float thetaH = Music.HarmonicCentricity(ch);
  float chi = Music.Consonance(ch);
  float mu = Music.Magnitude(ch);
  float delta = 0;
  float ba = 0;
  int bg = 0;

  int[] color = new int[ch.Count];
  for(int i=0;i<ch.Count;i++) {
```

```
      color[i] = Music.TiggerStripes(ch[i],sigma).ToArgb();

      if(Math.Abs(ch[i].Amp) > ba) {
        ba = Math.Abs(ch[i].Amp);
        bg = color[i];
        }
      }

    if(sigma > 1) sigma = 1f;
    if(sigma < 0) sigma = 0f;

    delta = 0;
    float cdiff = 0;
    // Compute spectacle.
    if(myLastCh != null) {
      delta = Music.ChangeOfChord(myLastCh,ch);
      cdiff = CDiff(ch,myLastCh);
      mySpectacle = delta + CDiff(ch,myLastCh);
      }

    float va = mu;
    float vb = mu + delta;
    float vc = thetaH; //mu + cdiff*(1-chi)*mu;
    float fa = 0;
    float fb = 0;
    float fc = 2f - cdiff - (float)mySpectacle;
    int elements = 30 + (int)((1-chi)*50) + (int)mySpectacle*50;

    UpdateMinMax(va,vb,vc,fa,fb,fc);

    float aa = (myMaxa - myMina);
    float bb = (myMaxb - myMinb);
    float cc = (myMaxc - myMinc);
    if(aa == 0) aa = 1;if(bb == 0) bb = 1;if(cc == 0) cc = 1;

    va = (va-myMina)/aa;
    vb = (vb-myMinb)/bb;
    vc = (vc-myMinc)/cc;

    myLastCh = ch;
    myThetaF = (myThetaF*(myN-1) + thetaF)/myN;
    myThetaH = (myThetaH*(myN-1) + thetaH)/myN;

    VisualSpace vs =new VisualSpace(elements,
                            new float[] {va,vb,vc},
                            new float[] {fa,fb,fc},
                            color,
                            new int[] {bg},
                            (float)mySpectacle);

    return new VisualSpace[] { vs };
    }

  }

}
```

A.0.14 Fourier Transform Class

```
     using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Threading;

using Core.Mathematics.Linear;
using Core.Mathematics.Stats;
using Core.Mathematics.Algebra;
using Core.Mathematics.NumberTheory;

namespace Core.Mathematics.Analysis {

  /// <summary>
  /// Provides a class that takes in a set of sampled time domain
  /// PCM and converts to frequency domain.
  /// </summary>
  public class FourierTransform {

    /// <summary>
    /// Perform the inner summand from equation (4)
    /// </summary>
    /// <param name="a">The current sub-index a, of (a + bN1)</param>
    /// <param name="offset">The offset into X to work relative to.</param>
    /// <param name="k">The base frequency we are after (Hz)</param>
    /// <param name="N">The total sample size</param>
    /// <param name="N1">The first factor N1*N2 = N</param>
    /// <param name="N2">The second factor N1*N2 = N</param>
    /// <param name="N1">Total N</param>
    /// <returns>The complex inner summand</returns>
    private static Z f_b(Z2[] X, int offset, float k, float b, float N1, float N2, float N,
        eWindowType useWindow) {
      Z t = 0;
      float w = 0;

      for(int a=0;a<N2;a++) {
        int idx = (int)(a+b*N2);

        // Hanning window if specified.
        // (1/2)*(1 - Cos((2 Pi*x)/(N - 1))
        switch(useWindow) {
          case eWindowType.Hanning:
            w = (float)(0.5f*(1.0f-Math.Cos((2.0f*Math.PI*Math.Floor(a+b*N2)))/N));
            break;
          default:
            w = 1;
            break;
          }

        t += X[offset + idx].A*Z.W(N,a*k)*w;
        }

      return t;
      }

    /// <summary>
    /// Given an array of real amplitude values perform the DFT assuming a sample rate
    /// of X.Length and a single sample.
    /// </summary>
    /// <param name="inRFs"></param>
    /// <param name="X"></param>
    public static void DFT(Z2[] X, float[] inRFs, eWindowType useWindow) {
      DFT(X,X.Length,0,inRFs,useWindow);
      }

    /// <summary>
    /// Given a complex array of raw input samples assumed to have (sample rate values) this
        performs
    /// a DFT on the requested Rf values.
    /// </summary>
    /// <param name="X">The sample space X[0].A is time domain, X[1].B is Rf </param>
    /// <param name="rfs">The Rfs to extract.</param>
    public static void DFT (Z2[] X, int sampleRate, int offset, float[] rfs, eWindowType useWindow
        ) {
```

```
  if((X.Length % sampleRate) != 0) throw new ArgumentException("The given data space is not
      congruent with the sample size " + sampleRate);

  float N = sampleRate;
  float _1OverN = 1.0f/N;
  Z c = 0;
  float w;

  for (int k=0;k<rfs.Length;k++) {
    c = 0;

    for(float n=0;n<N;n++) {
      switch(useWindow) {
        case eWindowType.Hanning:
          w = (float)(.5f*(1.0f-Math.Cos((2.0f*Math.PI*n)/N)));
          break;
        default:
          w = 1;
          break;
        }

      c += X[offset + (int)n].A*w*Z.W(N,n*rfs[k]);
      }

    X[offset + (int)rfs[k]].B = c*_1OverN;
    }
  }

/// <summary>
/// Perform an FFT on only the given targeted RF values assuming a single sample in
/// X whose samplerate is the size of the given array.
/// </summary>
/// <param name="inRFs">The frequencies to attack</param>
/// <param name="X">The structure consisting of 1 to many samples.</param>
/// <param name="useWindow">If true then a Hanning window is applied to the transform.</param>
public static void RfFFT(Z2[] X, float[] inRFs, eWindowType useWindow)  {
  RfFFT(X,X.Length,0,inRFs,useWindow);
  }

/// <summary>
/// Perform an FFT on the given array X using the Z2.A real values
/// as inputs.
/// </summary>
/// <param name="inRFs">The frequencies to extract</param>
/// <param name="offset">The offset in the X array to work relative to</param>
/// <param name="samplerate">The samplerate of the data</param>
/// <param name="X">The source amplitude array.</param>
public static void RfFFT(Z2[] X, int samplerate, int offset, float[] inRFs, eWindowType
    useWindow)  {
  int f1, f2;
  float N1,N2;
  int N = samplerate;
  float _1OverN = (1.0f/N);
  Z c = 0;

  // Factor and turn the factors to floats
  ExtraNumberTheory.Factor(samplerate,out f1, out f2);
  N1 = f1; N2 = f2;

  Object sync = new Object();
  Dictionary<int,int> vals = new Dictionary<int,int>();

  for (int i=0;i<inRFs.Length;i++) {
    float k = inRFs[i];

    c = 0;
    Z t = 0;

    // Non-parallelized for testing.
    //for(int b=0;b<N1;b++)
    //  c += f_b(X,offset,k,b,N1,N2,N,useWindow)*Z.W(N,b*N2*k);

    // Parallelize each inner summand.
    Parallel.For<Z>(0, (int)(N1-1),
                    () => 0,
                    (b,loopState,local2) => {
                      local2 += f_b(X,offset,k,b,N1,N2,N,useWindow)*Z.W(N,b*N2*k);
                      return local2;
                      },

                     (outputValue) => {
                       lock(sync) c += outputValue;
                       });
```

```
        X[offset + (int)k].B = c*_1OverN;
        }

      }

    }

/// <summary>
/// Specifies what kind of windowing technique to be used.
/// </summary>
public enum eWindowType {
    Hanning,
    None
    }

}
```

A.0.15 Synthesizer Class

```
  using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

using Core.Mathematics.Algebra;

namespace Core.Mathematics.Analysis {

  /// <summary>
  /// Provides utilities for generating different discrete waveforms as a series of
  /// amplitude samples.
  /// </summary>
  public class Synthesizer : List<Z2> {

    #region Private
    private int mySampleRate;
    #endregion

    #region Constructors

    /// <summary>
    /// Create the synthesizer.
    /// </summary>
    public Synthesizer(int sampleRate) { mySampleRate = sampleRate; }

    #endregion

    #region Utility

    /// <summary>
    /// Add the given frequency at the specified amplitude for the given duration in seconds.
    /// </summary>
    public void Add(float rf, float amp, int dur) {
      Z2[] sig = SynthesizeSignal(mySampleRate,new float[] {rf},dur,eDurationType.Seconds,new
          float[] {amp});
      AddRange(sig);
      }

    /// <summary>
    /// Add the given frequency to the signal starting at the given time.
    /// </summary>
    /// <param name="rf">The frequency to add</param>
    /// <param name="amp">The power level</param>
    /// <param name="dur">The duration of the frequency in seconds </param>
    /// <param name="start">The starting time offset in seconds</param>
    public void Add(float rf, float amp, int dur, int start) {
      Add(rf,amp,dur,eDurationType.Seconds,start,eDurationType.Seconds);
      }

    /// <summary>
    /// Add the given frequency to the signal starting at the given
    /// time.
    /// </summary>
    /// <param name="rf">The frequency to add</param>
    /// <param name="amp">The power level</param>
    /// <param name="dur">The duration of the frequency, units specified by inDurType </param>
    /// <param name="start">The starting time (offset) to begin adding</param>
    /// <param name="inDurUnits">How to interprest the duration value.</param>
    /// <param name="inStartUnits">What units are the start value to be considered</param>
    public void Add(float rf, float amp, int dur, eDurationType inDurUnits, int start,
        eDurationType inStartUnits) {
      Z2[] X = SynthesizeSignal(mySampleRate,new float[] { rf}, dur, inDurUnits, new float[] { amp
          });

      if(inStartUnits == eDurationType.Seconds) start = start*mySampleRate;

      // Add silence until we reach start (if not enough space allocated yet)
      while(Count < start+1) Add(new Z2(Z.Zero,Z.Zero));

      for(int i=0;i<X.Length;i++) {
        int offset = i + start;

        if(offset == Count) {
          Add(X[i]);
```

```
      }
    else {
      Z2 zt = this[offset];
      zt.A = this[offset].A + X[i].A;
      zt.B = this[offset].B + X[i].B;
      this[offset] = zt;
      }
    }

  }

/// <summary>
/// Add noise to the signal from the given starting point (sample item) to
/// for the given duration.
/// </summary>
/// <param name="start">The starting sample to begin adding noise</param>
/// <param name="dur">The duration of time to add nose (in seconds)</param>
/// <param name="noiseFloor">The maximum amplitude in the noise</param>
/// <param name="inDurUnits">The units to interpret the duration value in</param>
/// <param name="inStartUnits">The units to interpret the start offset in</param>
public void AddNoise(int start, eDurationType inStartUnits, int dur, eDurationType inDurUnits,
      float noiseFloor) {
  Random r = new Random((int)(DateTime.UtcNow.Ticks << 32));
  if(inStartUnits == eDurationType.Seconds) start = start*mySampleRate;

  if(inDurUnits == eDurationType.Seconds) dur = mySampleRate*dur;

  // Add silence until we reach start (if not enough space allocated yet)
  while(Count < start+1) Add(new Z2(Z.Zero,Z.Zero));

  for(int i=0;i<dur;i++) {
    int offset = i + start;

    if(offset == Count) {
      Z z = (float)(r.NextDouble()*noiseFloor);
      Add(new Z2(z,Z.Zero));
      }
    else {
      Z2 zt = this[offset];
      zt.A = (float)(this[offset].A.A + r.NextDouble()*noiseFloor);
      this[offset] = zt;
      }
    }

  }

#endregion

#region Properties

/// <summary>
/// Request the current total duration (in seconds) of data.
/// </summary>
public int Duration {
  get {
    return (int)Math.Ceiling(((double)Count)/mySampleRate);
    }
  }

public int SampleRate { get { return mySampleRate; }}

#endregion

#region Utility Methods

/// <summary>
/// Generate a time series of N samples per second that contains
/// the given array of frequencies embedded across the entire sample at the given volume.
/// </summary>
/// <param name="amp">The amplitudes to apply to each frequency respectively.</param>
/// <param name="rfs">The frequency list to be embedded in the entire sample</param>
/// <param name="sampleRate">The number of samples to provide</param>
/// <param name="duration">The duration the signal should last</param>
/// <param name="inTyp">How to interpret the durtion value's units</param>
public static Z2[] SynthesizeSignal(int sampleRate, float[] rfs, int duration, eDurationType
    inTyp, float[] amp) {
  int dur = (inTyp == eDurationType.Seconds)?sampleRate*duration:duration;
  Z2[] td = new Z2[dur];

  double _2PI = (float)(Math.PI*2.0d);
  float N = (float)sampleRate;
  float s = 0;
```

```
      for(int n=0;n<dur;n++) {
        s = 0;

        // Add each of the frequencies in.
        for(int j=0;j<rfs.Length;j++) s += (float)(amp[j]*Math.Cos((-2PI*rfs[j]*((float)n))/N));
        td[n].A = s;
        }

      return td;
      }

    /// <summary>
    /// Generate a time series of N samples per second that contains
    /// the given frequency embedded across the entire sample time at the given volume
    /// </summary>
    /// <param name="amp">The amplitude to apply to each frequency</param>
    /// <param name="rf">The frequency to be embedded in the entire sample</param>
    /// <param name="sampleRate">The number of samples to provide</param>
    /// <param name="duration">The duration of the signal in seconds.</param>
    public static Z2[] SynthesizeSignal(int sampleRate,float rf, int duration, float amp) {
      return SynthesizeSignal(sampleRate,new float[] {rf}, duration,eDurationType.Seconds, new
          float[] {amp});
      }

    /// <summary>
    /// Generate a time series of N samples per second that contains
    /// the given frequencies embedded across the entire sample time at the given volume.
    /// </summary>
    /// <param name="amp">The amplitude to apply to each frequency</param>
    /// <param name="rfs">The frequencies to be embedded in the entire sample</param>
    /// <param name="sampleRate">The number of samples to provide</param>
    /// <param name="duration">The duration of the signal in seconds.</param>
    public static Z2[] SynthesizeSignal(int sampleRate,float[] rfs,int duration, float amp) {
      float[] amps = new float[rfs.Length];
      for(int i=0;i<rfs.Length;i++) amps[i] = amp;
      return SynthesizeSignal(sampleRate,rfs,duration,eDurationType.Seconds,amps);
      }

    #endregion

    }

  /// <summary>
  /// Utilty enum used by the synthesize method.
  /// </summary>
  public enum eDurationType {
    /// <summary>
    /// The duration should last the given number of seconds.
    /// </summary>
    Seconds,
    /// <summary>
    /// The duration should last the given number of sample values.
    /// </summary>
    Samples
    }

}
```

A.0.16 Complex Number Class

```
  using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace Core.Mathematics.Algebra {

  //~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
  /// <summary>
  /// Provides the container for a complex number with a real
  /// and imaginary portion.
  /// </summary>
  //~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
  public struct Z : IField2 {

    #region Constants
    private const float _2PI = (float)(Math.PI*2d);
    #endregion

    //~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    /// <summary>
    /// The "Empty" value for this structure
    /// </summary>
    //~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    public static readonly Z Empty = new Z(float.MinValue,float.MinValue);

    //~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    /// <summary>
    /// The zero element.
    /// </summary>
    //~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    public static readonly Z Zero = new Z(0,0);

    #region Constructors

    //~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    /// <summary>
    /// Create a complex number
    /// </summary>
    //~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    public Z(float a, float b) {
      A = a;
      B = b;
      }

    //~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    /// <summary>
    /// Create a unit length complex number from the exponential form being
    /// Z = e^{iarg(z)} where |z| = 1.
    /// </summary>
    /// <param name="argz">The argument/angle which can be from 0 to N</param>
    //~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    public Z(float argz) {
      A = (float)Math.Cos(argz);
      B = (float)Math.Sin(argz);
      }

    #endregion

    #region Fields

    //~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    /// <summary>
    /// The real portion of the complex number
    /// </summary>
    //~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    public float A;

    //~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    /// <summary>
    /// The imaginary portion of the complex number
    /// </summary>
    //~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    public float B;
```

```
#endregion

#region IField2 Interface

public float X1 {
  get {
    return A;
    }
  set {
    A = value;
    }
  }

public float X2 {
  get {
    return B;
    }
  set {
    B = value;
    }
  }

#endregion

#region Properties

//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
/// <summary>
/// Ask if this is the empty structure
/// </summary>
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
public bool IsEmpty { get { return A == Empty.A && B == Empty.B; }}

//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
/// <summary>
/// Get the magnitude of this complex number.
/// </summary>
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
public float Magnitude {
  get { return (float)Math.Sqrt(A*A + B*B); }
  }

//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
/// <summary>
/// Get the argument of this complex number from 0 - 2PI
/// (This is NOT the principal argument)
/// </summary>
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
public float Arg {

  get {
    // 90  where tangent is undefined
    if(A == 0 && B > 0) return (float)(Math.PI/2.0f);

    // 270 where tangent is undefined.
    if(A == 0 && B < 0) return (float)(3.0f*Math.PI/2.0f);

    return (float)((Math.Atan2(B,A)  + _2PI) % _2PI);
    }

  }

//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
/// <summary>
/// Get the complex conjugate.
/// </summary>
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
public Z Conjugate {
  get { return new Z(A,-B); }
  }

#endregion

#region Utiltities

//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
/// <summary>
/// Assuming this complex number is in exponential form
/// raise it to the given power
/// </summary>
/// <param name="n">The power to raise the complex number to</param>
/// <returns>The new number</returns>
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

```
public Z e(float n) {
  return e((float)Math.Pow(Magnitude,n),Arg*n);
  }

//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
/// <summary>
/// Create a unit length exponential form of a complex number with the given argument.
/// </summary>
/// <param name="arg">The argument</param>
/// <param name="mag">The magnitude</param>
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
public static Z e(float mag, float arg) {
  return new Z(arg)*mag;
  }

//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
/// <summary>
/// Create an W_n which is e^{-i2PI/N*(arg)}.
/// </summary>
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
public static Z W(float N, float arg) {
  return new Z((float)(-Math.PI*2.0d/N*arg));
  }

#endregion

#region Operator Overloads

//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
/// <summary>
/// Divide a complex number by a real
/// </summary>
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
public static Z operator/(Z a, double d) {
  return new Z((float)(a.A/d),(float)(a.B/d));
  }

//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
/// <summary>
/// Divide two numbers
/// </summary>
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
public static Z operator/(Z a, Z b) {
  // Do a/b.

  Z c = b.Conjugate;
  double m = (b*b.Conjugate).A;
  return (a*c)/m;
  }

//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
/// <summary>
/// Assignment of a real value.
/// </summary>
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
public static implicit operator Z(double x) {
  return new Z((float)x,(float)0);
  }

//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
/// <summary>
/// Multiply by a constant.
/// </summary>
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
public static Z operator*(Z c1, double c) {
  return new Z((float)(c1.A*c),(float)(c1.B*c));
  }

//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
/// <summary>
/// Multiply by a constant.
/// </summary>
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
public static Z operator*(double c,Z c1) {
  return new Z((float)(c1.A*c),(float)(c1.B*c));
  }

//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
/// <summary>
/// Difference between to complex numbers c1 - c2.
/// </summary>
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
public static Z operator-(Z c1, Z c2) {
```

```
  return new Z(c1.A-c2.A,c1.B-c2.B);
  }

//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
/// <summary>
/// The multiplication operation
/// </summary>
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
public static Z operator*(Z c1, Z c2) {
  float a = c1.A;
  float b = c1.B;
  float c = c2.A;
  float d = c2.B;
  return new Z(a*c-b*d, a*d+b*c);
  }

//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
/// <summary>
/// The addition operator.
/// </summary>
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
public static Z operator+(Z c1, Z c2) {
  return new Z(c1.A+c2.A,c1.B+c2.B);
  }

//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
/// <summary>
/// The equality.
/// </summary>
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
public static bool operator==(Z c1, Z c2) {
  if(ReferenceEquals(c1,c2)) return true;
  else if( ((Object)c1) == null || ((Object)c2) == null) return false;
  return c1.A == c2.A && c1.B == c2.B;
  }

//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
/// <summary>
/// Real number equality.
/// </summary>
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
public static bool operator==(Z c1, double v) {
  return c1.A == v && c1.B == 0;
  }

//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
/// <summary>
/// Real number equality.
/// </summary>
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
public static bool operator!=(Z c1, double v) {
  return c1.B != 0 || c1.A != v;
  }

//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
/// <summary>
/// The in equality.
/// </summary>
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
public static bool operator!=(Z c1, Z c2) {
  return !(c1 == c2);
  }

#endregion

//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
/// <summary>
/// Object override  for equality.
/// </summary>
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
public override bool  Equals(object obj) {
  return this == ((Z)obj);
  }

public override string ToString() {
  if(B == 0)
    return "" + A;
  else {
    if(B < 0)
      return "" + A + " - i" +  Math.Abs(B);
    else
      return "" + A + " + i" +  B;
    }
```

```
    }

  public override int GetHashCode() {
    return ToString().GetHashCode();
    }

  }

}
```

A.0.17 Vortex Visual

```
  using System;
using System.Drawing;
using System.Threading;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using Microsoft.Xna.Framework;

using Core.Drawing;
using Core.Util;
using Core.Mathematics;
using Core.Mathematics.Geometry;
using Core.Mathematics.Algebra;

using Toolkit.Media.DSP;
using Toolkit.Xna;
using Toolkit.Media.DSP.Filters;
using Toolkit.Media.UI;

using Rectangle=System.Drawing.Rectangle;
using Color=System.Drawing.Color;
using Core.Mathematics.Stats;


namespace Apps.PictographReader.Plugins.Visuals {

  /// <summary>
  /// Provides an animation visual that creates a spinning vortex based upon the
  /// input parameters.
  /// </summary>
  public class VortexVisual : IVisualizer {

    #region Static Members
    private static float _2PI = (float)(Math.PI*2f);
    #endregion

    #region Private Members

    private bool myEnabled = true;

    private Random ourRandom = new Random((int)EpochTime.Now);
    // Vector that defines rotation angles (in radians) about each axis per each render action.
    protected List<VortexPlusData> mySpinnies = new List<VortexPlusData>();
    private Z myDFT = new Z(0,0);
    private VisualSpace myLastVS = VisualSpace.Empty;
    #endregion

    #region Private Utility

    /// <summary>
    /// Plot an arc centered about x,y,z with starting, ending and current angles of the given
        radius
    /// </summary>
    private void PlotArc(IGraphicsCanvas g, float x, float y, float z, float start, float end,
        float cur, float rad, int color) {
      float x2,y2;

      // Calulate the current radial line and draw it.
      x2 = (float)(rad*Math.Cos(cur));
      y2 = (float)(rad*Math.Sin(cur));

      float incSz = (float)(end - start)/60f;
      float i = start;
      float lX = (float)(rad*Math.Cos(i));
      float lY = (float)(rad*Math.Sin(i));

      for(;i<=cur;i+=incSz) {
        x2 = (float)(rad*Math.Cos(i));
        y2 = (float)(rad*Math.Sin(i));
        g.DrawLine(lX + x,lY + y,z,x2 + x,y2 + y,z,color);
        lX = x2;
        lY = y2;
        }
      }
```

```
/// <summary>
/// Plot the line and increment it.
/// Returns true when its done.
/// </summary>
private void Plot(IGraphicsCanvas g) {

  try {
    XnaGraphicsCanvas graphics = (XnaGraphicsCanvas)g;
    graphics.Projection = Toolkit.Xna.eXnaCanvasPerspective.Perspective;

    if(!Inc()) return;

    lock(mySpinnies) {

      foreach(VortexPlusData data in mySpinnies) {

        float br = (data.mySize - (data.myAngle-data.mySAngle)/data.myEAngle*data.mySize);

        if(br > 0 && data.Spectacle > .4f)
          graphics.FillCircle(data.myX,data.myY,data.myZ,br,data.Color);

        // Plot an arc
        int cnt = (int)(1f + data.MaxElements*data.Spectacle);
        float rinc = data.mySize/((float)cnt);
        float r = data.mySize;
        float z = data.myZ;
        int c = 0;

        for(int i=0;i<cnt;i++) {
          PlotArc(graphics,data.myX,data.myY,z,data.mySAngle,data.myEAngle,data.myAngle,r,data
              .myColors[c++ % data.myColors.Length]);
          r -= rinc;
          z -= rinc;
          }
        }

      } // End lock

    }
  catch(Exception ex) { Log.Error(ex); }

  }

/// <summary>
/// Increment the animations to the next phase.
/// </summary>
private bool Inc() {

  lock(mySpinnies) {
    bool draw = false;

    for(int i=0;i<mySpinnies.Count;i++) {

      if(mySpinnies[i].IsDone) {
        mySpinnies.RemoveAt(i);
        i--;
        }
      else {
        mySpinnies[i] = mySpinnies[i].Inc();
        draw=true;
        }
      }

    return draw;
    }
  }

#endregion

#region Constructors

/// <summary>
/// Create a vortex visual renderer.
/// </summary>
public VortexVisual () {}

#endregion

#region Utility Methods

/// <summary>
/// Reset the vortex.
/// </summary>
```

```
public void Clear() {
  mySpinnies.Clear();
  }

/// <summary>
/// Add a sample to be processed.
/// </summary>
public void Update(VisualSpace vs) {
  if(!myEnabled) return;

  if(!myLastVS.IsEmpty) {

    lock(mySpinnies) {

      // Starting angle
      float pStart = (float)vs.Values[0]*_2PI;

      // Ending angle
      float pEnd = (float)(pStart + vs.Values[1]*_2PI*100f + vs.Spectacle*_2PI*100);

      // Radius
      float radius = (vs.Values[0])*1.5f;

      // Setup center to be some growth away from focus based upon the pri/sec/third values.
      float specRange = .25f;

      float y = -2.0f + (vs.Focus[0]*2f - specRange) + specRange*vs.Spectacle + (float)
          ourRandom.NextDouble();
      float x = -2.0f + (vs.Focus[1]*2f - specRange) + specRange*vs.Spectacle + (float)
          ourRandom.NextDouble();
      float z = -2.0f + (vs.Focus[2]*2f)  + (float)ourRandom.NextDouble();

      // Velocity
      float velocity = (float)((pEnd-pStart)/(80 - (30.0f*vs.Spectacle)));

      VortexPlusData sd = new VortexPlusData (pStart,pEnd,velocity,x,y,z,radius,vs.
          ForegroundColors,vs.Spectacle,vs.MaxElements);

      if(mySpinnies.Count < 30)
        mySpinnies.Add(sd);
      }
    }

  myLastVS = vs;
  }

public void PerformDrawing(IGraphicsCanvas canvas) {
  if(!myEnabled) return;
  Plot(canvas);
  }

#endregion

#region Properties

/// <summary>
/// Get the type of graphics system this expects.
/// </summary>
public eGraphicsSupport GraphicsSupport {
  get { return eGraphicsSupport.Xna; }
  }

#endregion

}

#region Utility Classes

public struct VortexPlusData {


  private int myCIdx;       // Index into color to use.
  public float mySAngle;    // Starting angle postion
  public float myEAngle;    // Ending angle position
  public float myAngle;     // Current rotational position
  public float myRotateVelocity; // Angle increment
  public float myX;         // Tranlsation
  public float myY;         // Tranlsation
  public float myZ;         // Translation
  public float mySize;      // Length of line.
  public int[] myColors;
  public float Spectacle;
  public int MaxElements;
```

```
    private bool myFirstPassDone;

    /// <summary>
    /// Is this done rotating
    /// </summary>
    public bool IsDone {

      get {

        if(float.IsNaN(myAngle) || float.IsNaN(myEAngle))
          return true;

        if(!myFirstPassDone)
          return false;
        else
          return (myAngle <= mySAngle);
        }
      }

    /// <summary>
    /// Get the color to use on the current leg of the spinny guy.
    /// </summary>
    public int Color {
      get {
        return myColors[myCIdx];
        }
      }

    /// <summary>
    /// Rotatate to the next position
    /// </summary>
    public VortexPlusData Inc() {
      if(!myFirstPassDone)
        myAngle += myRotateVelocity;
      else
        myAngle -= myRotateVelocity;

      myCIdx = (myCIdx + 1) % myColors.Length;
      if(myAngle >= myEAngle && !myFirstPassDone) myFirstPassDone = true;
      return this;
      }

    /// <summary>
    /// Create a spinny data object.
    /// </summary>
    /// <param name="color">RGB color</param>
    /// <param name="cx">Center X of spinner</param>
    /// <param name="cy">Center Y of spinner</param>
    /// <param name="endAngle">Ending angle in radians</param>
    /// <param name="rad">Radius of spinner</param>
    /// <param name="startAngle">Starting angle of spinner in radians</param>
    /// <param name="velocity">Velicty of the spinners rotation in radians</param>
    public VortexPlusData(float startAngle,
                          float endAngle,
                          float velocity,
                          float cx,
                          float cy,
                          float cz,
                          float rad,
                          int[] color,
                          float spectacle,
                          int elements) {
      mySAngle = startAngle;
      myEAngle = endAngle;
      myAngle = mySAngle;
      myRotateVelocity = velocity;
      myX = cx;
      myY = cy;
      myZ = cz;
      mySize = rad;
      myColors = color;
      myCIdx = 0;
      Spectacle = spectacle;
      MaxElements = elements;
      myFirstPassDone = false;
      }

    }

  #endregion
}
```

A.0.18 Visualizer Interface

```
  using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

using Core.Mathematics.Algebra;
using Core.Mathematics;
using Core.Drawing;

using Toolkit.Media.DSP;
using Toolkit.Media;
using Toolkit.Media.UI;

namespace Apps.PictographReader.Plugins.Visuals {

  /// <summary>
  /// Provides an interface connected to some implementation that generates an
  /// animation or visual drawing based upon a parameter space.
  ///
  /// The underlying implementation is thought to have no "brains" and only draws what it is told.
  /// </summary>
  public interface IVisualizer : IGraphicsEnabled {

    /// <summary>
    /// Remove/Cleanup everything for another run.
    /// </summary>
    void Clear();

    /// <summary>
    /// Construct the visual based upon the paramter space.
    /// </summary>
    void Update(VisualSpace inSpace);

    }

  }
```

A.0.19 Visualizer Space

```
  using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

using Core.Util;

namespace Toolkit.Media.UI {

  /// <summary>
  /// Provides a structure that describes an animation sprite as a set of generic
  /// parameters to be interpreted by some graphic renderer.
  /// </summary>
  public struct VisualSpace {

    #region Public Constants
    public static readonly VisualSpace Empty = new VisualSpace(-1,null,null,null,null,-1);
    #endregion

    #region Properties

    /// <summary>
    /// A threshold value from 0 to 1 where 1 means
    /// absolutely spectauclar visualization and zero means
    /// nothing special.
    /// </summary>
    public float Spectacle;

    /// <summary>
    /// Maximum number of elements to be created.
    /// Designed to limit the underlying animator.
    /// </summary>
    public int MaxElements;

    /// <summary>
    /// A three space value list to be interpreted by the visualizer.
    ///
    /// The values can be anything but commonly interpreted as
    /// primary,
    /// secondary,
    /// tiertiery
    /// respectively.
    /// </summary>
    public float[] Values;

    /// <summary>
    /// Provides the centroid element that describes some focal region of the
    /// animation.
    /// </summary>
    public float[] Focus;

    /// <summary>
    /// The primary color RGBA value to be used.
    /// </summary>
    public int[] ForegroundColors;

    /// <summary>
    /// A list of color values to be used in background renderings.
    /// </summary>
    public int[] BackgroundColors;

    public float Mag;

    #endregion

    #region Constructors

    /// <summary>
    /// Create the visualizer space.
    /// </summary>
    public VisualSpace(int inMaxElements,
                       float[] inValue,
                       float[] inFocus,
                       int[] inColors,
                       int[] inBGColors,
```

```
                        float inSpectacle) {
      MaxElements = inMaxElements;
      Values = inValue;
      Focus = inFocus;
      ForegroundColors = inColors;
      BackgroundColors = inBGColors;
      Spectacle = inSpectacle;
      Mag = 0;
      }

    #endregion

    public override string ToString() {
      StringBuilder sb = new StringBuilder();

      sb.Append(ExtraString.ToString(Values));
      sb.Append(",");
      sb.Append(ExtraString.ToString(Focus));
      sb.Append(",");

      sb.Append(Spectacle);
      return sb.ToString();
      }

    #region State Properties

    /// <summary>
    /// Ask if this visual space is an empty space.
    /// </summary>
    public bool IsEmpty {
      get { return MaxElements == -1 && Spectacle == -1 && ForegroundColors == null && Values ==
          null && Focus == null; }
      }
    #endregion
    }
}
```

A.0.20 Survey Results

```
  using System;
using System.IO;
using System.Collections.Generic;
using System.Linq;
using System.Text;

using Core.Mathematics.Linear;
using Core.Util;
using Core.IO;

namespace ThesisAnalysis {

  /// <summary>
  /// A class that parses an email response and computes
  /// the score of a taken survey.
  /// </summary>
  public class SurveyResults {

    #region Private Members
    // Scores for each survey (1 per person)
    private List<SurveyScore> myScores = new List<SurveyScore>();
    private int myFail = 0;
    #endregion

    #region Private Util

    /// <summary>
    /// Score computation that is an inverse score.
    /// </summary>
    private float s(float a) { return (5f-a)/5f; }

    /// <summary>
    /// Score computation that is a direct score.
    /// </summary>
    private float t(float a) { return a/5f; }

    /// <summary>
    /// Compute the score given a set of answers.
    /// </summary>
    private float ComputeScore(float[] answers) {
      float[] mappings = {0,5,4,0,2,1};
      float[] importance = {2,3,8,8,10,10,5};

      // Map answers to values.
      for(int i=0;i<answers.Length;i++) answers[i] = mappings[(int)answers[i]];

      return s(answers[0])*importance[0] +
             s(answers[1])*importance[1] +
             t(answers[2])*importance[2] +
             t(answers[3])*importance[3] +
             t(answers[4])*importance[4] +
             t(answers[5])*importance[5] +
             t(answers[6])*importance[6];
      }

    #endregion

    #region Constructors

    /// <summary>
    /// Create a survey results processor by loading every file in the given directory
    /// with extension .txt assuming each file is an email response from the survey.
    /// </summary>
    public SurveyResults(FilePath inDir) {
      foreach(FilePath fp in inDir.GetFiles(".txt")) Add(fp.Text);
      }

    #endregion

    #region Utility

    /// <summary>
    /// Add the results of one email to the processor.
    /// </summary>
    /// <param name="inMsg">The email message</param>
```

```
public void Add(String inMsg) {

  try {
    StringReader sr = new StringReader(inMsg);
    String line;
    List<String> results = new List<String>();
    int run = 0;
    String user = "";

    // Read each line and look for the results (Ignore debug)
    while((line = sr.ReadLine()) != null) {
      if(line.Trim() == "") continue;
      else if(line.Contains("Run:")) {
        run = int.Parse(line.Substring(line.IndexOf(":")+1).Trim());
        continue;
        }
      else if(line.Contains("TimeStamp:")) {
        user = line.Substring(line.IndexOf(":")+1).Trim();
        continue;
        }

      // Found results section, read til the log section.
      if(line.Contains("RESULTS")) {
        while((line = sr.ReadLine()) != null && !line.Contains("LOG")) {
          if(line.Trim() == "") continue;
          results.Add(line);
          }
        }
      }

    SurveyScore ss = new SurveyScore();

    float bestScore = ComputeScore(new float[] {5,5,1,1,1,1,1});

    // Compute the score for each.
    for(int i=0;i<results.Count;i++) {
      String s = results[i];
      String genre = ExtraString.SubString(s,0,s.IndexOf(":")).Trim().ToUpper();
      s = ExtraString.SubString(s,s.IndexOf(":")+1,s.Length);
      String[] opts = s.Split(new char[] {','},StringSplitOptions.RemoveEmptyEntries);

      float[] v = new float[opts.Length];
      for(int j=0;j<opts.Length;j++) v[j] = int.Parse(opts[j]);
      ss.Add(new QuestionScore(run,user,genre,(float)(((ComputeScore(v)/bestScore)*100f) )));
      }

    myScores.Add(ss);
    }
  catch(Exception ex) {
    Log.Error(ex);
    myFail++;
    }
  }

/// <summary>
/// Create CSV from the score.
/// </summary>
public String ToRawCSV() {
  StringBuilder sb = new StringBuilder();
  float f = 0;
  float fcnt = 0;

  sb.Append("User, Grade %\r\n");
  for(int i=0;i<myScores.Count;i++) {
    foreach(QuestionScore qs in myScores[i])  {
      sb.Append(qs.Genre.ToUpper() + "," + qs.Score + "\r\n");
      f += qs.Score;
      fcnt++;
      }
    }

  sb.Append("\r\n\r\n\r\n");
  TitleSummer ts = new TitleSummer();
  sb.Append("Genre, Grade Ave %\r\n");
  for(int i=0;i<myScores.Count;i++) {
    foreach(QuestionScore q in myScores[i])
      ts.Add(q.Genre,q.Score);
    }
  foreach(String key in ts.Titles) sb.Append(key + "," + ts.GetScore(key) + "\r\n");

  sb.Append("\r\n\r\n\r\n");
  TopperLower tl = new TopperLower(true,5);
  sb.Append("Top 5 Scores, Grade %\r\n");
```

```
      for(int i=0;i<myScores.Count;i++) tl.Add(myScores[i]);
      foreach(QuestionScore s in tl) sb.Append(s.Genre + "," + s.Score + "\r\n");

      sb.Append("\r\n\r\n\r\n");
      tl = new TopperLower(false,5);
      sb.Append("Bottom 5 Scores, Grade %\r\n");
      for(int i=0;i<myScores.Count;i++) tl.Add(myScores[i]);
      foreach(QuestionScore s in tl) sb.Append(s.Genre + "," + s.Score + "\r\n");

      Console.WriteLine("Overall Grade " + (f/fcnt));

      return sb.ToString();
      }

    /// <summary>
    /// Get the scores.
    /// </summary>
    public SurveyScore[] Scores() {
      return myScores.ToArray();
      }

    #endregion

    #region Properties

    /// <summary>
    /// Get the number of failed surveys.
    /// </summary>
    public int Failed { get  { return myFail; }}

    /// <summary>
    /// Get the total number of surveys taken.
    /// </summary>
    public int Count { get { return myScores.Count; } }
    #endregion

    }

  #region Internal Classes

  /// <summary>
  /// Class that manages the score to a full survey.
  /// </summary>
  public class SurveyScore : List<QuestionScore> {

    public SurveyScore() {}

    /// <summary>
    /// Get the best score.
    /// </summary>
    public QuestionScore Best {
      get {
        QuestionScore b = null;

        foreach(QuestionScore qs in this) {
          if(b == null || qs.Score > b.Score)
            b = qs;
          }

        return b;
        }
      }

    /// <summary>
    /// Get the worst score.
    /// </summary>
    public QuestionScore Worst {

      get {
        QuestionScore b = null;

        foreach(QuestionScore qs in this) {
          if(b == null || qs.Score < b.Score)
            b = qs;
          }

        return b;
        }
      }

    /// <summary>
    /// Compute the overall score
    /// </summary>
```

```
  public float Score {
    get {
      double v = 0;
      foreach(QuestionScore qs in this) v += qs.Score;
      return (float)(v/((float)Count));
      }
    }
  }

/// <summary>
/// Class that manages a score to a single question.
/// </summary>
public class QuestionScore {
  public int Run;
  public String User;
  public float Score;
  public String Genre;

  public QuestionScore(int run, String user, String genre, float score)  {
    Run = run;
    User = user;
    Genre = genre;
    Score = score;
    }

  public override string ToString() {
    return Genre + ":_" + Score;
    }

  }

public class TitleSummer {

  // Values.
  private Dictionary<String, float > myD = new Dictionary<string,float>();

  // Conunts.
  private Dictionary<String, float > myC = new Dictionary<string,float>();

  /// <summary>
  /// Put or get without having to worry if the item already exists.
  /// </summary>
  public void Add(String key, float v) {

    if(myD.ContainsKey(key)) {
      float f = myD[key];
      f += v;
      myD[key] = f;

      float cnt = myC[key];
      cnt ++;
      myC[key] = cnt;
      }
    else {
      myD.Add(key,v);
      myC.Add(key,1);
      }
    }

  /// <summary>
  /// Get all the titles of this summation.
  /// </summary>
  public String[] Titles {
    get { return myD.Keys.ToArray(); }
    }

  /// <summary>
  /// Get the score for the given title.
  /// </summary>
  public float GetScore(String title) {
    float s = myD[title];
    return s/myC[title];
    }
  }


/// <summary>
/// Holds top/bottom N scores.
/// </summary>
public class TopperLower : List<QuestionScore> {
  private bool myTop = true;
  private int myCnt = 0;
```

```
  public ToppperLower(bool doTop, int cnt) {
    myTop = doTop;
    myCnt = cnt;
    }

  public void Add(SurveyScore ss) {

    foreach(QuestionScore q in ss) {

      if(Count < myCnt) {
        base.Add(q);
        continue;
        }

      // Find and replace low/high score.
      for(int i=0;i<Count;i++) {

        if(myTop) {
          if(this[i].Score < q.Score) {
            this[i] = q;
            break;
            }
          }
        else if(this[i].Score > q.Score) {
          this[i] = q;
          break;
          }

        }
      }
    }

  }

#endregion


}
```