Stand number ________

Maths Behind Music

An investigation into the simulation of musical instrument timbres by Short Time Discrete Fourier Analysis.

Sarah Kate Sweeney

Scoil Mhuire Gan Smál

Blarney

County Cork

Judges' notes

Table of Contents

Contents

Acronyms

CIT	Cork Institute of Technology
CSM	Cork School of Music
CSS	Cascading Style Sheets
DFT	Discrete Fourier Transform
DSP	Digital Signal Processing
iDFT	inverse Discrete Fourier Transform
FFT	Fast (discrete) Fourier Transform
HTML	Hyper Text Markup Language
JSON	JavaScript Object Notation
MJON	Music JavaScript Object Notation
MML	Music Markup Language
PCM	Pulse Code Modulation
STDFT	Short Time Discrete Fourier Transform
XML	Extensible Markup anguage

Table of Figures

1. Meet Sarah

Hi! My name is Sarah Kate Sweeney. My favourite school subjects are music, maths and science. I study piano and musicianship at the CIT School of Music. For three years now, I attend and mentor at CoderDojo where I learn Python, Minecraft hacking, Hyper Text Markup Language (HTML), JavaScript, Cascading Style Sheets (CSS) and JSON. In 2015 I represented my school or dojo in eight science fairs.

I really love this project, Maths behind Music, because it combines all my favourite hobbies; music, maths, science and coding!

Figure 1 Recording notes at CSM

2. Summary

2.1 Objective

The primary objective of this project is to understand why different musical instruments sound different even when playing the same note. The secondary objective is to use these findings to compactly encode musical scores and to build a software synthesiser to render such scores.

2.2 Methodology

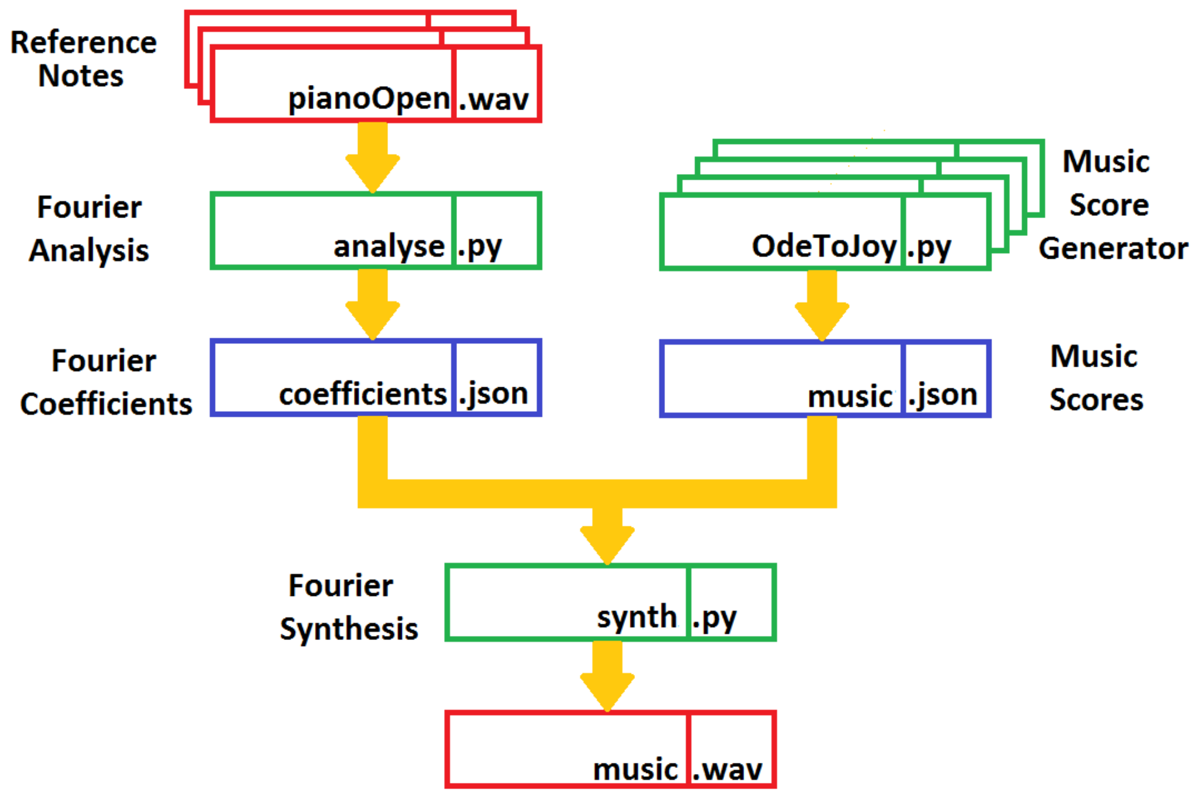


Figure 2 Methodology Flowchart

I recorded 285 reference notes on a piano (stringed percussion), ukulele (stringed plucked), guitar (string strummed) and recorder (woodwind) into .wav files (Appendix A11).

I wrote a Python program (analyse.py) that performed a short time discrete Fourier transform (STDFT) on these reference notes and extracted the amplitudes of the fundamental and 19 harmonics in each window of 100ms (Appendix A9). These amplitude coefficients were stored in JSON files.

I developed a JSON format for musical scores. I developed a Python program (odetojoy.py) that produced a JSON file of OdeToJoy.json by Beethoven in this format.

I wrote a Python program (synth.py) to generate musical output using the Fourier coefficients created by analyse.py and OdeToJoy.json musical score (Appendix A10).

2.3 Results Summary

The penultimate result for this project is the synthesis of "OdeToJoy" by Beethoven. You can listen to this piece of synthesised music at the following URL…

http://coderdojo.cix.ie/BTYSTE2016

This piece of music uses four different instrument for each of the four motifs and uses a closed lid piano for the left hand accompaniment. This piece of music demonstrates the achievement of the project objectives.

There are a number of other less important results outputs. These can also be reached from hyperlinks via the above hyperlink.

- In August 2015, I used the recording studios in the Cork School of Music (CSM) to record 285 reference notes for detailed analysis.
- I developed a suite of eight Python experiments / techniques that were useful in analysis and synthesis. Chapter 5 of this report outlines these experiments in detail.
- I developed a working analysis program (analyse.py) that outputs the harmonic amplitudes in JavaScript Object Notation (JSON).
- I developed a synthesis program to interpret the JSON and output a .wav file.
- I have created a JSON document that encodes 'Ode to Joy' by Beethoven using the MJSON format I designed.
- I have a website containing some of the .wav files I produced (Ref 001).

2.4 Application

My project is looking at digital real sound synthesis, replacing sampling with mathematical simulation. This allows interaction between the synthesiser and say a game application. It is likely that this form of synthesis will become more important with time in both the music and movie industries.

Interactive synthesis of sounds (musical and non-musical) has applications in the game market. The game market is now larger than either the movie or music industries.

3. Introduction

Music is my number one interest. My main instrument is piano but I also play recorder, guitar and ukulele. As well as playing, I love to study musicianship and music theory. I have attended CoderDojo for three years and this has also given me an interest in technology and coding. My school, Scoil Mhuire Gan Smál, has a long tradition of doing science projects and Mr. Foley has given me great encouragement and help.

I plan to have a musical career. However, in the twenty first century it seems that music production and technology are colliding. Modern laptops that run software such as “Logic” or “FL Studios” have more capability than existed in recording studios a mere twenty years ago.

My brother, Cian, is a musician and recently completed an MA in sound production at the University of Westminster. In December 2014, he published a paper that outlines the mathematical basis for musical scales. His paper started me thinking on the mathematical basis of music. All of his work is based on simple multiplication and I was delighted to find that I could understand all of what he has written (Ref 002).

My Dad is an electronics engineer and he has studied a lot of complicated mathematics used in signal analysis. He is helping me to understand some of this. For this project, I use Fourier analysis and this requires doing Fourier transforms from the time domain to the frequency domain. Initially I couldn’t do this type of mathematics but he explained the ideas qualitatively. Across the summer I began to understand the maths. Correlation, the underlying digital signal processing (DSP) concept, can be understood by imagining a mechanical Fourier transform made from the harp inside a piano. I now have discrete Fourier transform (DFT), inverse discrete Fourier transform (iDFT) software and a simplified synthesis program.

Thankfully I inherited my Mom’s musical ear. My Dad is tone deaf!

I feel that this project has helped me to understand the link between technology and music. What is more, this is only the first stage. I plan to continue this work for the rest of my secondary education in parallel with my piano and musicianship classes. It is my intention to learn as much as I can about the mathematics of digital signal processing, sound and music. After I leave school, I hope to obtain a doctorate in this area of music and technology.

As both a coder and a musician, the idea of being able to program music really excites me. Music JSON, or MJSON, could be used in instrumentals, backing tracks, and is ideal for games. In such an application, the tempo or the instruments timbre could be changed within the game. It can also be used as a form of musical documentation. A page of MJSON is be very small when compared to a sound file, and would therefore be ideal for transmission to a mobile device. MJSON is formatted in JSON rather than XML. People have coded musical cores in XML previously but I couldn’t find examples of coding music in JSON.

I am hoping to continue this work for the rest of my secondary education with a view to understanding how music can be faithfully recreated mathematically and develop a program that would convert backwards and forwards between MJSON and other musical score formats such as LilyPond and MIDI. The work to date will form the basis for this. I would also like to expand my work to include the synthesis of sounds that are not musical. This has applications normally implemented by ‘Foley Artists’ in movies and radio.

4. Background Research

4.1 The Sine Wave

Sound is a longitudinal pressure wave moving through air. The name for the shape of this pressure wave is a sine wave. Sine is defined in trigonometry as the ratio of the opposite over the hypotenuse. I calculated the sine of a number of angles from zero to 720 degrees (0 to 4 Pi radians) in a spreadsheet and plotted the graph. My Dad explained the relationship to trigonometry, to me, using the unit circle drawing below. The associated Excel table below shows the sine function being calculated and these values are plotted in the Excel chart. For example sin 30^0 is 0.5.

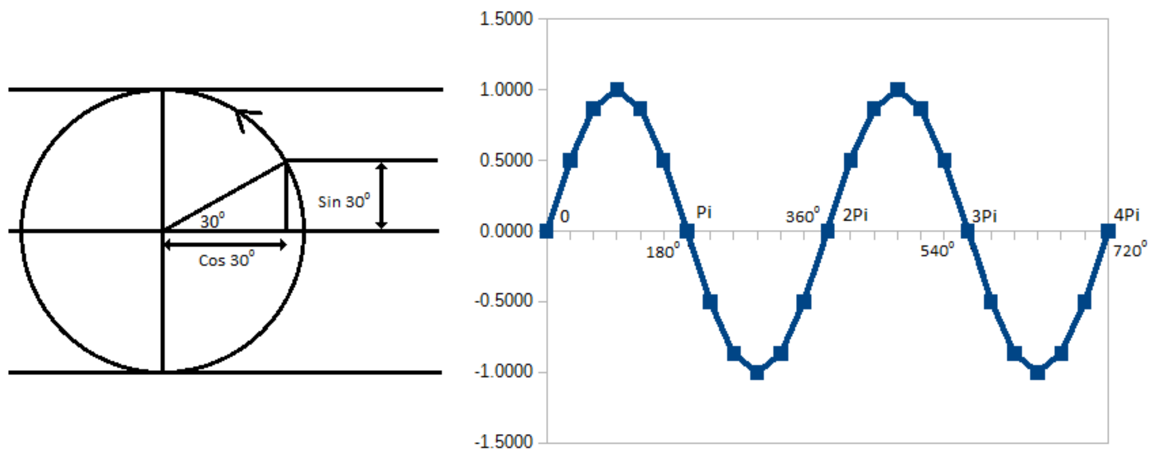


Figure 3 Sine wave and the unit circle

Degrees	Radians	Sin
0	0	0
30	0.523598776	0.5
60	1.047197551	0.866
90	1.570796327	1
120	2.094395102	0.866
150	2.617993878	0.5
180	3.141592654	0
210	3.665191429	-0.5
240	4.188790205	-0.866
270	4.71238898	-1
300	5.235987756	-0.866
330	5.759586532	-0.5
360	6.283185307	0

Degrees	Radians	Sin
360	6.283185307	0
390	6.806784083	0.5
420	7.330382858	0.866
450	7.853981634	1
480	8.37758041	0.866
510	8.901179185	0.5
540	9.424777961	0
570	9.948376736	-0.5
600	10.47197551	-0.866
630	10.99557429	-1
660	11.51917306	-0.866
690	12.04277184	-0.5
720	12.56637061	0

In the Python programs developed for this project and in the spreadsheet above I needed to use radians. 180^0 = Pi radians.

Because a lot of my research related to Fourier series. I needed to understand the sine wave before I could understand Fourier series.

My Dad claims that all engineering problems fall into two categories: to make things oscillate correctly and to stop things oscillating. This project is about making things oscillate correctly. The Tacoma Bridge is an infamous example of something oscillating that shouldn't (Ref 008).

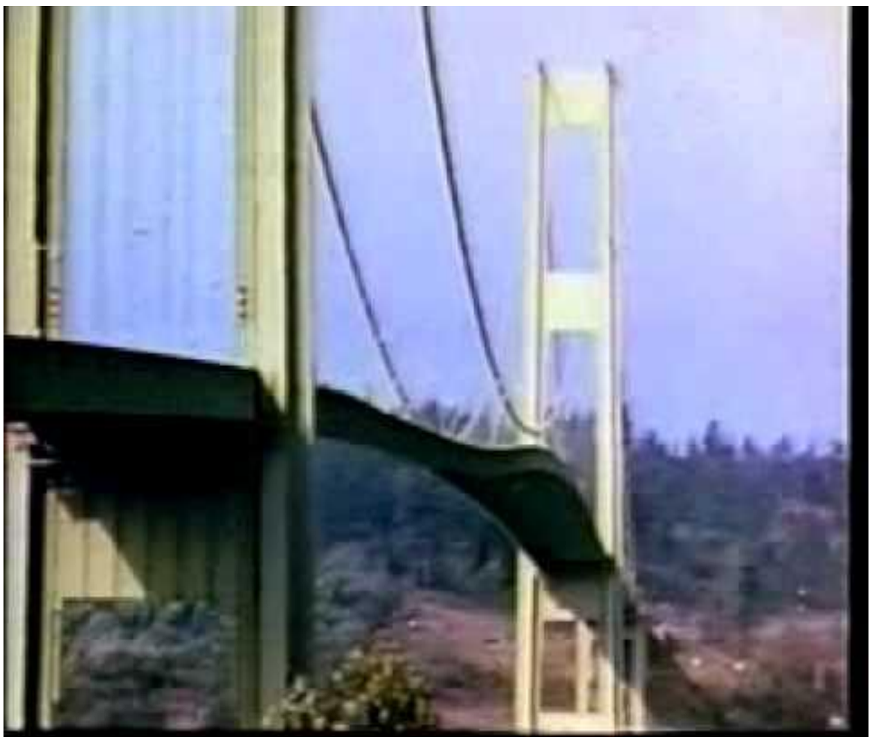


Figure 4 Tacoma Bridge

This bridge had a natural resonant frequency of 0.2Hz. A certain steady velocity of wind had a similar effect on the bridge, as wind on the reed of a woodwind instrument. Therefore, the bridge began to oscillate at its resonant frequency. Its amplitude kept increasing until the bridge collapsed. This took about an hour.

4.2 Digital Signal Processing (DSP)

DSP is how computers process signals. DSP is processing analog signals (e.g. a vibrating air molecule) as a series of numbers, what we call a digital signal.

4.2.1 Pulse Code Modulation

Pulse Code Modulation (PCM) is a standard technique for representing sound signals digitally (Ref 003, page 2). The following diagram taken from electronicshub.org explains the concept of PCM visually.

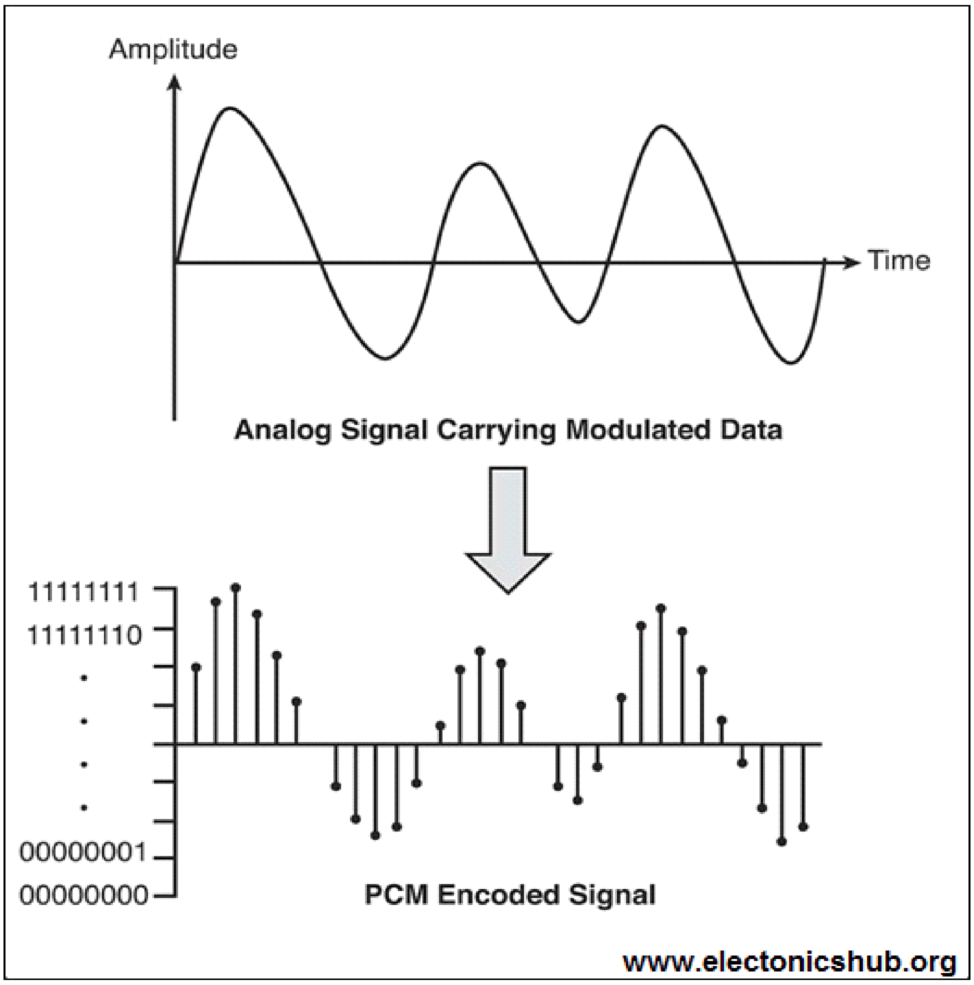


Figure 5 Pulse Code Modulation

First, the analog signal must be sampled in such a way that the shape of the samples is approximately the same shape as the original waveform. These sample points can then be given values according to their vertical position in the waveform. These values can be stored digitally in a computer.

The Wav file format is the most common non compressed digital storage file format and is the format I use in this project. Standard Wav uses a sample rate of 44,100Hz and a vertical resolution of 16 bits (-32,768 to +32,867) on two stereo channels.

4.2.2 Nyquist-Shannon Sampling Theorem

The Nyquist-Shannon Sampling Theorem is a vital part of DSP. It states that the sample rate has to be greater than double the highest frequency component in a signal. In the following diagram I illustrate the consequence of not following this rule.

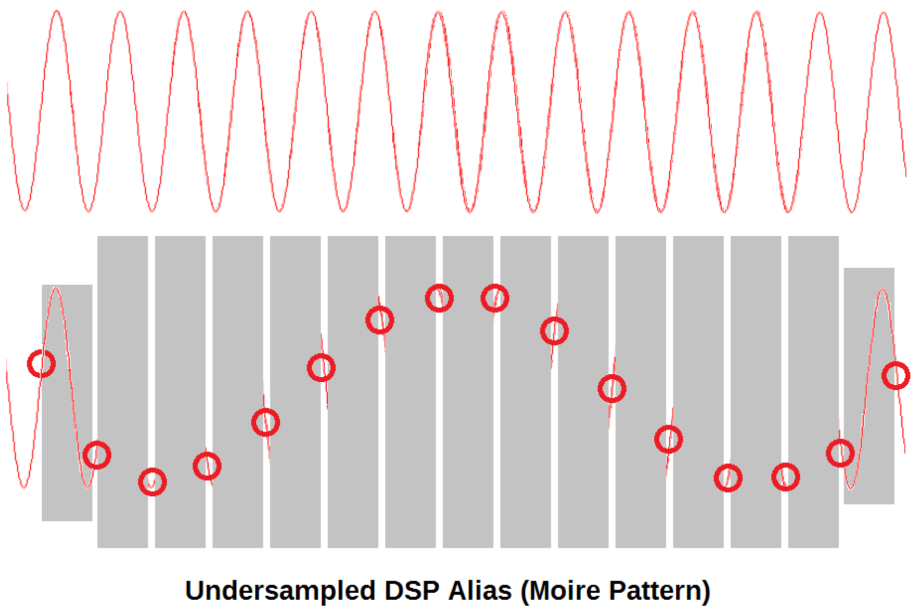


Figure 6 Nyquist Shannon Sampling Theorem

Using a Moiré Pattern, which is a visual form of alias, this shows what happens if the sampling rate is too low. As the samples are too sparse, they create a lower frequency, or an alias. Nyquist Shannon must be taken into account if this is to be avoided.

4.3 Fourier Analysis

4.3.1 Understanding Harmonics and the Fourier series

For the first phase of my project, I did not know any of the maths behind Fourier Analysis, but I had a good understanding of the concept. From June 2015 I started to seriously study Fourier analysis mathematically.

Before SciFest CIT, my Dad showed me was how to play a note on a piano without pressing the key by using resonance. I remember being really surprised by that. This proved that one note contained the frequency of another. If you hold a note until it gets quiet and then strongly but briefly press the note one octave down, you will hear the note that is down being played. We repeated this on a swing. He showed me that even small pushes at any integer multiple of the natural frequency of the swing caused the amplitude to grow but irregular pushes caused nothing except annoying wobbles.

I watched the YouTube videos of Anna-Maria Hefele (Ref 004). Anna-Maria is an overtone singer. This means she is able to create cavities using her mouth which match the resonant frequency of the harmonic and amplifies it. This gives an illusion of singing two notes at a time. The two notes must be integer harmonics of each other. Anna-Maria and I are doing the same thing; the difference being that I use instruments, she uses her voice. Please note that on her videos what she refers to as an overtone I call a harmonic.

During the summer, my Dad and I imagined how we could make a physical Fourier transform machine by cutting the front off our piano and attaching a microphone to each string. If we tuned the strings with 200Hz intervals we would have a mechanical Fourier transform that would have 88 linear buckets from 200 Hz to 17.6kHz.

4.3.2 Discrete Fourier Transform

When I was in SciFest CIT, I had a good qualitative understanding of how Fourier Analysis works, but I knew very little about the background mathematics. Across the summer of 2015, I worked to gain a quantitative understanding of Fourier Analysis. The following is the formula used to perform a DFT:

$$\underset{(k=0,N-1)}{F(k)} = \sum_{n=0}^{N-1} f(n)\mathrm{Cos}(2\pi nk/N) - j\, f(n)\mathrm{Sin}(2\pi nk/N)$$

The above formula translates into the snippet of Python code below.

- The outer 'for loop' computes each F(k)
- The inner 'for loop' performs the sigma by successively adding terms to Ftemp.
- f(n) contains the recording we want to analyse.
- N is the total number of samples in the signal f(n)
- F(k) ends up containing the complex Fourier coefficients (each F(k) is called a bucket).

```python
F = []
N = len(f)
for k in range(0, N):
    Ftemp = 0
    for n in range(0, N):
        Ftemp += (f[n] * math.cos(2*n*math.pi*k/N)) -
        (1j*f[n] * math.sin(2*n*math.pi*k/N))
    F.append(Ftemp)
```

An understanding of correlation is required to understand Fourier Transforms, section 4.3.4 gives a detailed explanation of correlation.

Please note that even though I wrote my own Fourier Transform, it ran too slowly to be useful for the large volumes of data I needed to process. In the Analysis software I wrote I use the Numpy Standard Python library. This implements the faster Cooley Tuckey method which I have not studied.

4.3.3 Inverse Discrete Fourier Transform

The following is the formula used to perform an iDFT.

$$\underset{(n=0,N-1)}{f(n)} = \frac{1}{N}\sum_{k=0}^{N-1} a_k \mathrm{Cos}(2\pi nk/N) - b_k \mathrm{Sin}(2\pi nk/N)$$

The above formula translates into the snippet of Python code below.

- The outer ‘for loop’ computes each f(n)
- The inner ‘for loop’ performs the sigma by successively adding terms to ftemp
- F(k) contains the complex Fourier coefficients, each F(k) is called a bucket
- f(n) contains the time domain signal we want to synthesise

```
    f = []
    N = len(F)
    for n in range(0, N):
        ftemp = 0
        for k in range(0, N):
            ftemp += (F[k].real * math.cos(2*n*math.pi*k/N)) -
(F[k].imag * math.sin(2*n*math.pi*k/N))
        f.append(round(ftemp/N))
```

4.3.4 Simplified Synthesis Function

Fourier analysis applies to all signals, not just audio. Since I am using only audio, I can make three simplifications to the DFT :

1. a0 is the offset component. In audio, this always equals 0, so for my purposes it can be ignored.

2. According to the Nyquist-Shannon sampling theorem, components from N/2 to N – 1 are all aliases and should be ignored if filtered correctly.

3. Because phase shift is inaudible a general amplitude coefficient, A_n can be computed as follows for each harmonic:

$$A_n = \sqrt{a_n^2 + b_n^2}$$

Bringing these three points together yields the following simplified Fourier Series for audio:

$$\underset{(n=0,N-1)}{f(n)} = \frac{1}{N}\sum_{k=0}^{N-1} A_k \operatorname{Sin}(2\pi nk/N)$$

The Fourier Series is in effect an iDFT.

4.3.4 Correlation

4.3.4.1 How It Works

Fourier series says that any signal is made up of a fundamental and integer multiple harmonics. A Fourier Transform is a technique for finding the amplitudes of these harmonics.

Correlation is the backbone of the Fourier Transform. It is what Fourier uses to find out what frequencies are contained in a signal, and their amplitudes. Fourier Series tells us what the frequencies are and we use correlation to find how much of each is in the signal.

The following chart shows a square wave (the signal we want to analyse) and four sinusoids (sin wt, cos wt, sin 2wt, cos 2wt) that we are going to correlate against. Correlation finds the 'degree of match' between the signal and the comparison sinusoids.

The amazing thing is that correlation (degree of match) is calculated by simply adding together each value of the signal multiplied by the value of the correlating signal at the same time.

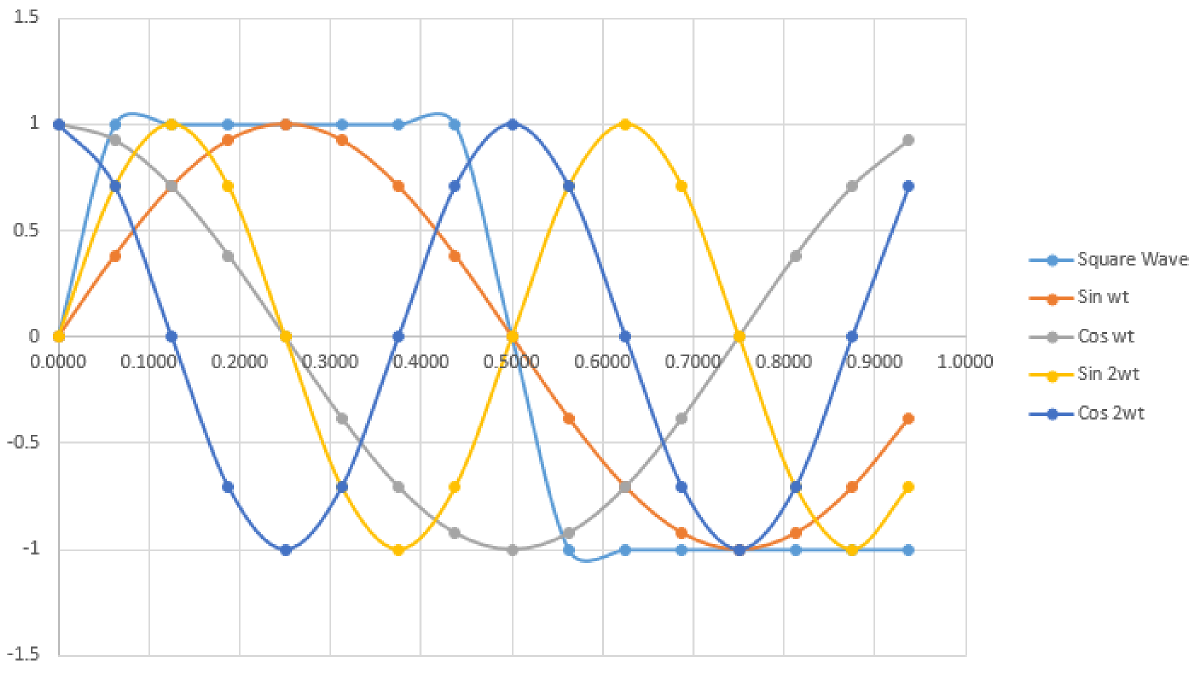


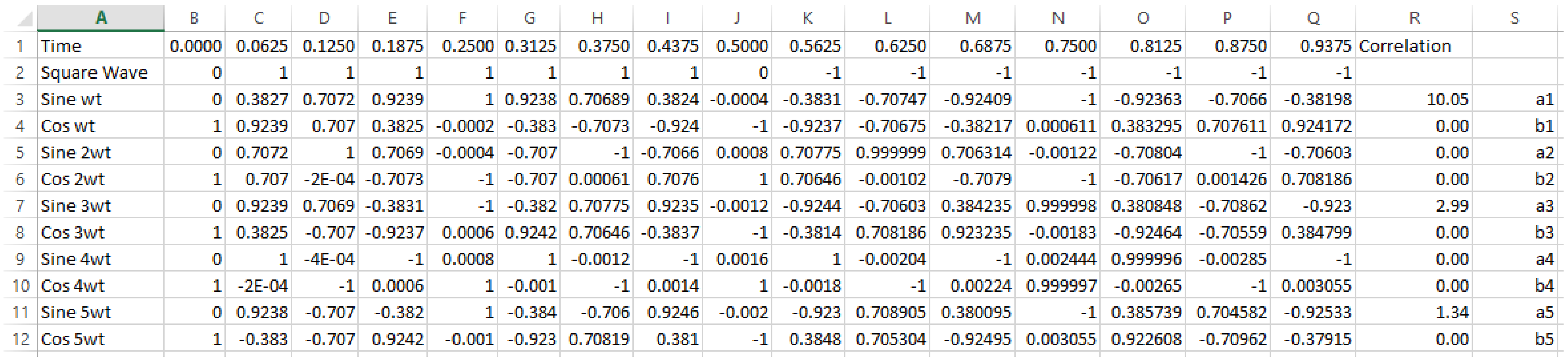
Figure 7 Correlation

The graph above was created in Excel from data calculated in Excel in the table on the next page.

Sin wt correlates very well with the square wave and Cos wt does not. The first term of the Fourier Series (the fundamental) of the waveform is 10.05 + j0.

There is no correlation between either Sin 2wt or Cos 2wt and the signal so the second term is 0 + j0.

Similarly the third term is 2.99 + j0.



	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S
1	Time	0.0000	0.0625	0.1250	0.1875	0.2500	0.3125	0.3750	0.4375	0.5000	0.5625	0.6250	0.6875	0.7500	0.8125	0.8750	0.9375	Correlation	
2	Square Wave	0	1	1	1	1	1	1	1	0	-1	-1	-1	-1	-1	-1	-1		
3	Sine wt	0	0.3827	0.7072	0.9239	1	0.9238	0.70689	0.3824	-0.0004	-0.3831	-0.70747	-0.92409	-1	-0.92363	-0.7066	-0.38198	10.05	a1
4	Cos wt	1	0.9239	0.707	0.3825	-0.0002	-0.383	-0.7073	-0.924	-1	-0.9237	-0.70675	-0.38217	0.000611	0.383295	0.707611	0.924172	0.00	b1
5	Sine 2wt	0	0.7072	1	0.7069	-0.0004	-0.707	-1	-0.7066	0.0008	0.70775	0.999999	0.706314	-0.00122	-0.70804	-1	-0.70603	0.00	a2
6	Cos 2wt	1	0.707	-2E-04	-0.7073	-1	-0.707	0.00061	0.7076	1	0.70646	-0.00102	-0.7079	-1	-0.70617	0.001426	0.708186	0.00	b2
7	Sine 3wt	0	0.9239	0.7069	-0.3831	-1	-0.382	0.70775	0.9235	-0.0012	-0.9244	-0.70603	0.384235	0.999998	0.380848	-0.70862	-0.923	2.99	a3
8	Cos 3wt	1	0.3825	-0.707	-0.9237	0.0006	0.9242	0.70646	-0.3837	-1	-0.3814	0.708186	0.923235	-0.00183	-0.92464	-0.70559	0.384799	0.00	b3
9	Sine 4wt	0	1	-4E-04	-1	0.0008	1	-0.0012	-1	0.0016	1	-0.00204	-1	0.002444	0.999996	-0.00285	-1	0.00	a4
10	Cos 4wt	1	-2E-04	-1	0.0006	1	-0.001	-1	0.0014	1	-0.0018	-1	0.00224	0.999997	-0.00265	-1	0.003055	0.00	b4
11	Sine 5wt	0	0.9238	-0.707	-0.382	1	-0.384	-0.706	0.9246	-0.002	-0.923	0.708905	0.380095	-1	0.385739	0.704582	-0.92533	1.34	a5
12	Cos 5wt	1	-0.383	-0.707	0.9242	-0.001	-0.923	0.70819	0.381	-1	0.3848	0.705304	-0.92495	0.003055	0.922608	-0.70962	-0.37915	0.00	b5

Figure 8 Fourier transform explaining correlation

4.4 Musical Scales

I am very interested in music theory. This section outlines the ideas I learned during my studies, especially from Ref 002 and Ref 004.

An octave consists of twelve equal semitones, in modern equal temperament tuning. There are tuning schemes other than equal temperament. These are used in traditional music, sometimes jazz music, and in specific cultures. It is often stated that pentatonic scales(Ref 009) are more natural that equal temperament. Since the early 1700s, all formal western music has been in equal temperament scales. The big advantage of equal temperament is that if a musician meets a singer that requires a key change there is no need to retune the instrument as all keys are interchangeable in equal temperament. This is vital for church organs and keyboard instruments in general.

Equal temperament can be represented by the following formula where X can be solved to 1.0594630944. This number X represents the difference in frequency between any two adjacent keys on a piano regardless of whether they are black or white.

$X^{12} = 2$

I used the spreadsheet below to iteratively find the value of X. I kept adjusting X until A6 was exactly 1,760Hz. The value I got for X was 1.0594630944.

X to the power of the note	Note	Frequency(Hz)
1	A	880
1.0594830944	A#	932.32752303
1.1224620483	B	987.76660249
1.189207115	C	1046.5022612
1.2599210499	C#	1108.7305239
1.3348398541	D	1174.6590716
1.4142135623	D#	1244.5079348
1.4983070768	E	1318.5102276
1.587401519	F	1396.9129256
1.6817928304	F#	1479.9776907
1.7817974361	G	1567.9817438
1.8877486252	G#	1661.1287902
2	A	1760

As you can see, the first note is exactly half of the last note. This is called an octave. All scales use octaves that double in frequency. Because X is a constant (semitone) between two notes, we call this a chromatic scale. Not all scales are chromatic. For example pentatonic scales are not chromatic.

4.5 Chords and Circle of Fifths

Consonant chords are based on integer ratios. The simplest such ratio is the perfect fifth (also called the 'power chord' in rock music) which represents a ratio of 1.5. The difficulty is that no integer power of 1.5 (fifths) and integer power of 2 (octaves) coincide. It is impossible to have a chromatic scale that has a perfect octave and a perfect fifth.

$1.5^{12} = 129.746$

Notice in the table above that E forms an almost perfect fifth with A (the ratio being 1.4983070768 instead of 1.5). In a piano, this approximation to a fifth is equivalent to seven semitones. Octaves are perfectly tuned and fifths are approximately tuned to fit in with octaves. The following equality shows that in seven octaves you get twelve fifths by using the ratio 1.4983070768.

$2^{7} = 128 = 1.498307076812^{12}$

The reason there are twelve semitones in an octave is because the 'Circle of Fifths' rotates twelve times before returning to the starting note. On each rotation it lands on a different note in the octave.

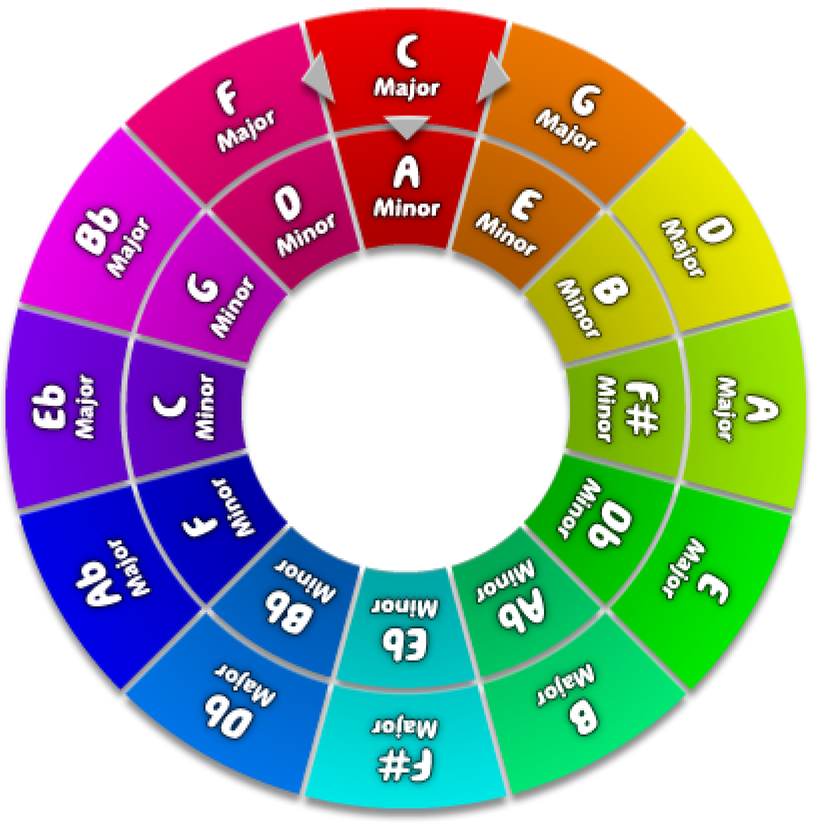


Figure 9 Circle of Fifths

Every music theory student learns about the Circle of Fifths but few understand the mathematical basis beneath it.

When equal temperament tuning was developed, the instrument tuners were faced with a choice: the perfect fifth, or the perfect octave? They chose the perfect octave, but in doing a 'perfect' fifth is not actually perfect. A true perfect fifth is 1.5 times the original note, but in equal temperament it is only 1.4983070768. This is the reason many musicians are distressed when they hear that the ‘perfect’ fifth is, ironically, imperfect.

Occasionally arpeggio singers abandon the perfect octave in favour of the perfect fifth.

Most pentatonic scales do have true perfect fifths and perfect octaves but they are not chromatic.

4.7 Claude Shannon and Information Theory

In the late 1940s Claude Shannon developed what he referred to as “Information Theory”. This branch of mathematics/physics laid the basis for long distance communications and the digital world we take for granted today. His research covered areas such as noise and data loss in analog and digital systems as well as data compression and encryption. I tried to understand some of his ideas and to interpret my project using those ideas. (Ref 003)

The pressure of air at a point in space and time can take any value. All of the analysis done in this project is performed by a digital computer. The microphone in my computer is an analog device. It turns the sound pressure wave into an electronic analog. The computer digitises the sound pressure waves, turning them into numbers. After processing the computer turns them back into analog signals which are turned back into pressure waves to enable the sound to be heard.

In my project I am using wav files for storing and processing digital signals. Wav files are based on the CD format developed around 1980. This format allows for 16 bit amplitude resolution and a sampling rate of 44,100 samples per second. According to Shannon's information theory this sample rate is capable of being used to reproduce frequencies up to 22,050Hz which is just above human ability to hear. The 16 bit amplitude allows for +- 32,767 which again exceeds human ability to distinguish sound levels.

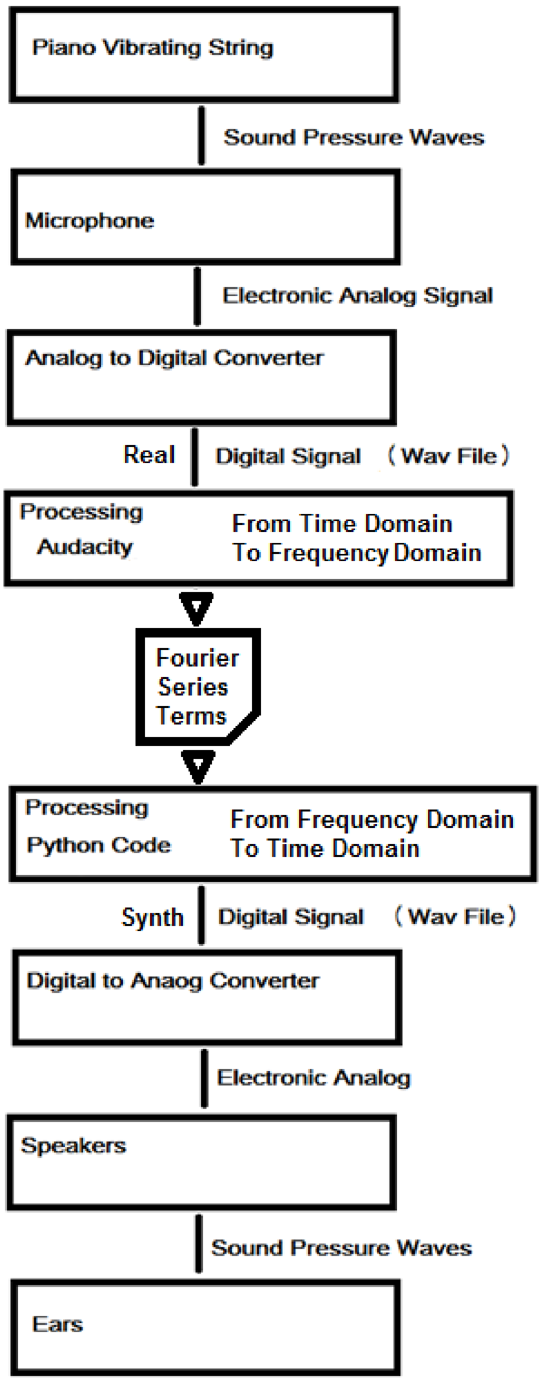


Figure 10 DSP Process

Wav files record amplitude linearly and not logarithmically (i.e. not in dBs).

For my initial experimentation I used the microphone and A/D converter in my laptop. When I went to record the tune in the second part of my project I used an external professional microphone and audio amplifier and the quality was far superior. The signal to noise ratio was 15dB better with the professional recording equipment.

4.8 Synthesis

There are four phases so far in the evolution of digital synthesis. Music (and sound) electronic synthesis is entering the fourth phase of its evolution (Ref 003).

Phase 1: Analog Modular Synthesis

Initially synthesis was based on analog modular systems such as the Moog Synthesisers of the 1970s. Although Analog Synthesisers are no longer in commercial production the ideas are incorporated in modern digital synthesisers products like 'Pure Data'.

Phase 2: Digital Synthesisers

In the 1980s, electronic bands (such as Kraftwerk) added pure digital synthesis. These instruments produced the electronic sounds of that era. No attempt was made to simulate traditional instruments.

Phase 3: Sampling

Mike Oldfield's Tubular Bells in the early 1980s is a masterpiece of digital synthesis but it also started to add samples of real instruments. Today, synthesisers such as "Garage Band" and its big brother 'Logic' use sampling to synthesise real instruments. They even use sampling to recreate the sounds of synthesisers that were originally digital.

Phase 4: Interactive Digital Real Sound Synthesis

My project is looking at digital real sound synthesis, replacing sampling with mathematical simulation. This allows interaction between the synthesiser and say a game application. It is likely that this form of synthesis will become more important with time in both the music and movie industries.

5. Experiment Programs

Before I could take the project to a level beyond what I had done for SciFest@CIT, I needed to understand the maths behind Fourier Analysis and I needed to develop my coding skills. I spent most of the summer reading about the topic and practicing my Python coding. I did this work in the form of eight experiments. This chapter outlines those experiments.

5.1 Python and Complex Numbers

See appendix A.1 for code.

Experiment one deals with how Python handles complex numbers and proves that complex multiplication yields a rotation.

An angle theta (θ in the Greek alphabet) is assigned a value of pi over 6 radians, or 30 degrees. The program is the asked to find the sin and cos of theta. The variable 'a' is then declared cos of theta and j times the sin of theta. This is a complex number. When multiplying complex numbers, the rule is to graph this on an argand diagram, then add the angles and multiply the lengths. This results in an angle of 2 pi over 6 radians, or 60 degrees. This is a larger angle, proving that complex multiplication yeilds a rotation.

5.2 FFT Analysis of Square Wave

See reference A.2 for code.

Experiment two uses numpy, an import from Python, to perform an FFT. It proves that a0 is always equal to 0 for a function centered on the x axis.

It first is given a signal of [1, 1, 1, 1, 1, 1, 1, 1, -1, -1, -1, -1, -1, -1, -1, -1] (a square wave), performs an FFT and then calculates the absolute value of these results and plots them in a graph.

The reason it calculates the absolute value of the FFT output is due to the fact that an FFT yields a complex number. The real component represents the frequency amplitude, and the imaginary component represents the offset. However, it is unnecessary to account for offset in audio as the human ear is not sensitive to it.

Second, it is given a signal = [2, 2, 2, 2, 2, 2, 2, 2, 0, 0, 0, 0, 0, 0, 0, 0] (another square wave) and performs an FFT on the signal, calculates the absolute values and plots it in a graph as it did the first.

If you look at the results, you will notice that all coefficients in the entire first column equal 0. These are what I call the 'a0' coefficients. They are what determines the offset of a signal. If the signal is centered on the x axis in the graph, a0 will always equal 0.

5.3 Synthesis of a Square Wave

See appendix A.3 for code.

Experiment three is the first experiment to create an audible .wav file. It's main purpose however, is to prove that phase shift is inaudible to the human ear. It does this by creating two almost identical square waves. One has phase shift and one does not. In the waveform graphs drawn using matplotlib, the two look completely different. However, when the .wav files are played they sound identical. This indirectly proves that the human ear is Fourier Transform.

5.4 Reading a wav file

See Appendix A.4 for code.

This tool does what is says on the tin. It imports an existing wav file, analyses it, and prints out the first five stereo values.

5.5 Creating a Spectrogram

See Appendix A.5 for code.

A spectrogram is a visual representation of how a frequency spectrum changes over time. I use them for sound, but they can be used for any signal. This tool creates and displays a spectrogram using the Python library matplotlib. This code generates a spectrogram of a single piano note.

A Spectrogram is like a sliced pan, where each slice is a spectrum for that window.

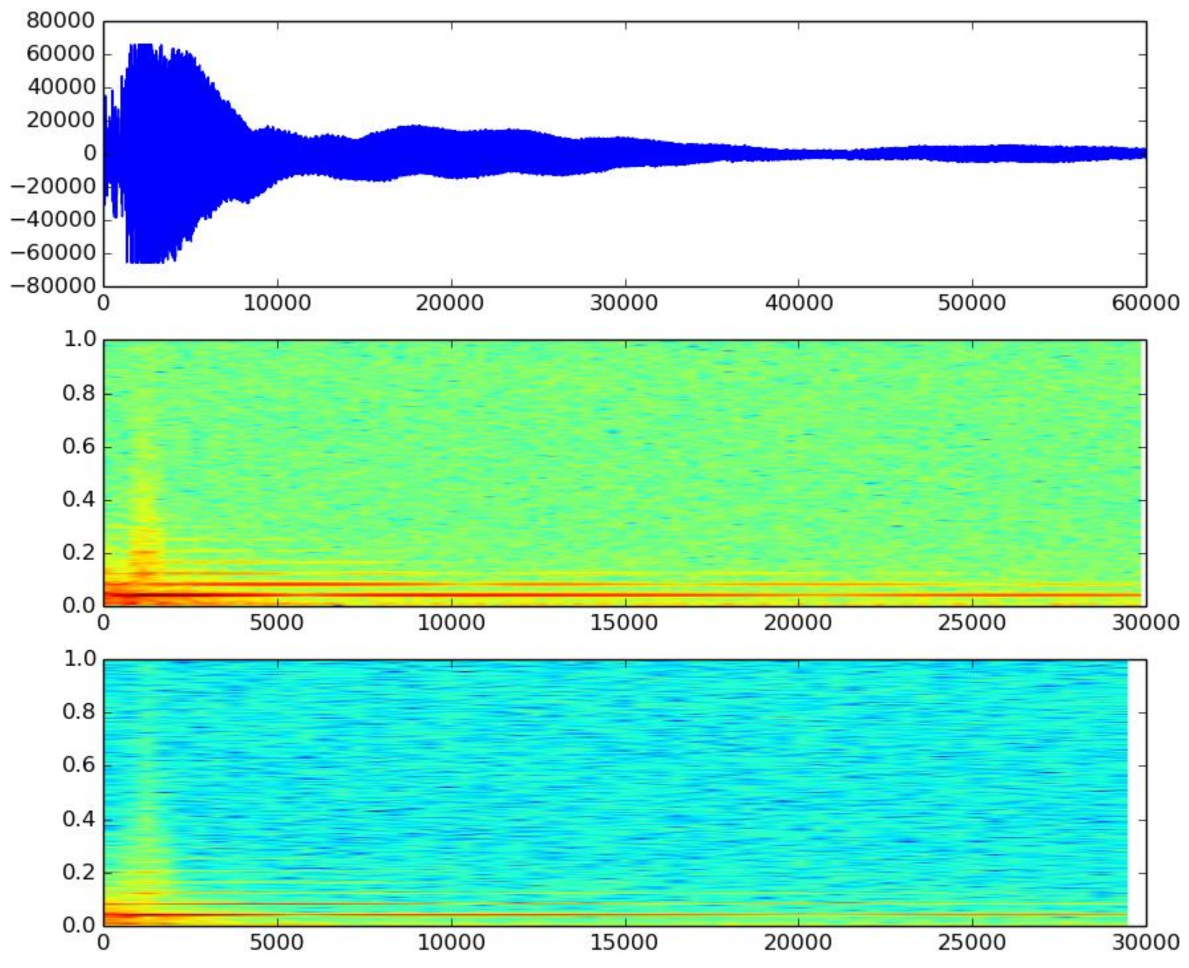


Figure 11 Spectrogram

5.6 Dictionary into a JSON object

See Appendix A.6 for code.

This experiment puts a dictionary into a JSON object then reads it back into a dictionary. I needed a way to store both amplitude coefficients and musical scores on my computer. Within a Python program this data can be stored as a dictionary. A JSON object is similar to a Python dictionary but it has the advantage that it can be stored as a file on a computer hard drive for use by another computer.

5.7 Outputting Data to a .wav file

See Appendix A.7 for code.

This experiments creates a functional wav file. This program outputs a square wave to a way file.

5.8 DFT and iDFT

See Appendix A.8 for code.

The purpose of this experiment was to prove that for any signal (even, odd, or neither even nor odd) a DFT followed by an iDFT would yield the original signal. This proved that the transforms are reversible.

This code was developed from first principles and the fft functionality in numpy was not used.

6. Experimental Methods

6.1 Recording Reference Notes

For this third phase of the project, I was allowed the use of a music studio in the Cork School of Music with a Steinway piano and professional grade recording equipment. I brought a ukulele, guitar and recorder. I recorded multiple notes per instrument to ensure that at there will be 'good' notes for analysis among the odd 'dud' ones.

In total I now have a library of 288 reference notes.

6.1.1 Piano

I recorded the piano notes on a grand piano. I noticed that when I played with the lid closed, it sounded different than with the lid open. I was curious to know how the harmonics changed from open to closed lid, and so recorded both.

I recorded every note on the keyboard for both (88 notes * 2 lid positions).

6.1.2 Guitar

There are six strings on a guitar: Low E, A, D, G, B and high E. I recorded one octave (12 semitones) per string.

6.1.3 Ukulele

There are four strings on a ukulele: A, C, E and G. I recorded 8 notes per string.

6.1.4 Recorder

I recorded a C major scale on the recorder consisting of 8 notes.

6.2 Analysing Reference Notes

See Appendix A.9 for code.

I have developed a program, called analyse.py, to perform a short time discrete Fourier transform (STDFT) analysis on my recorded reference musical notes. An FFT performed on a signal divided into windows is called an STDFT. The algorithm of analyse.py is as follows:

- Loops through instruments.
- Loops through notes.
- Loops through windows.
- Performs a FFT on each window.
- Finds the absolute value of the amplitude coefficients.
- Saves the coefficients to a JSON file.

6.3 Synthesising Music

I have developed a program, called synth.py, which uses the coefficients generated in analyse.py to create orchestral multi instrument music from a JSON file. The algorithm of synth.py is as follows:

- Reads the JSON score for the song.
- Creates two empty lists that will contain the left and right stereo channels. Make these 10ms longer than the last note in the song to allow for slap elimination.
- Loop on each note.
- Loop on each window.
- Calculate window_start and window_end for slap elimination.
- Loop on each sample.
- Loop on each harmonic.
- Calculate sigma A_n * sine (2 * pi * f * t) and store for that sample.
- All loops end here.
- Add some left to right and vice versa for headphone usage.
- Save the lists to the wav file.

6.4 Slap Removal

One of the main problems I faced with synthesis was one that I called ‘Slap’. This was an annoying click between windows. The following drawing shows a comparison between a window transition at 2.6 seconds on the Ode To Joy music.

In the upper signal the wave transitions smoothly from window to the next. In the lower signal there is a sudden transition and hence a slap in the waveform that is clearly audible.

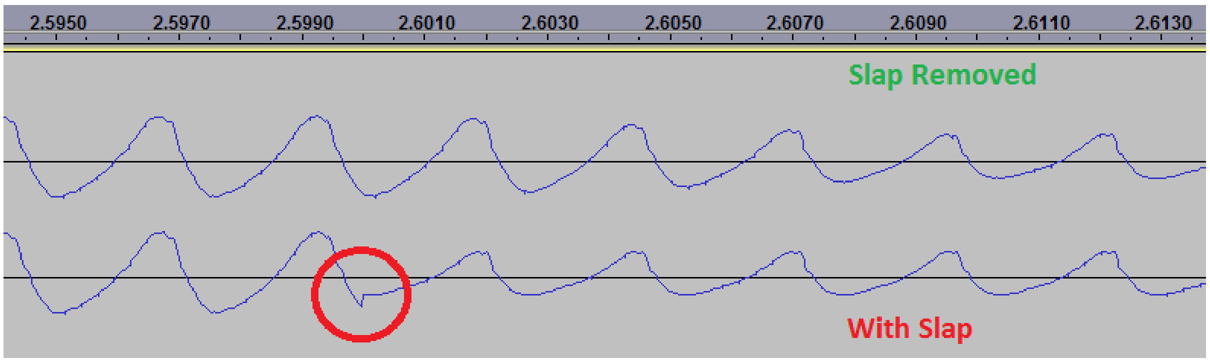


Figure 12 Example of Slap

Slap occurs when a window ends in a different place to where the next begins. The result is what looks like a vertical line. This line requires an infinite amount of energy for the speaker to play it, so it creates a ‘slap’ in the speaker.

I have eliminated slap by making the amplitude coefficients ‘ramp up’ for 10ms at the start of every window, continue on for the 100ms window, and overlap into the next window for 10ms to ‘ramp down’, while the next note is ramping up.

The code to fix this is…

```
# If the window is starting then ramp in over 10ms
if sample < (start_window + 441):
       signal *= ((sample - start_window) / float(441))

# If the window is ending then ramp out over 10ms
if sample > (end_window - 441):
        signal *= ((end_window - sample) / float(441))
```

Note: 441 samples = 10ms

The following diagram explains slap and its solution.

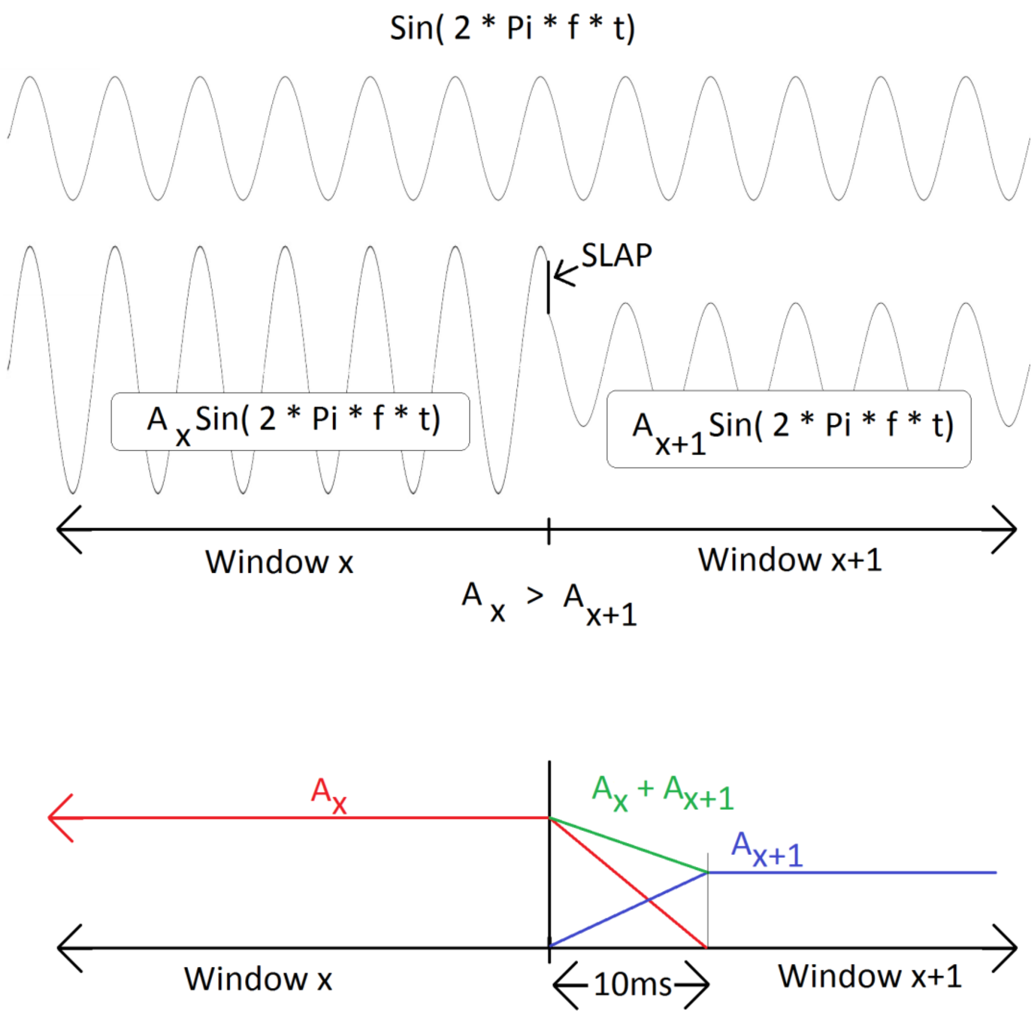


Figure 13 Slap Elimination

The red line added to the blue line, during the window transition, gives the green line. The green line is a smooth transition between windows.

7. Results

The sound file results and software code of this project are available at this URL.

http://coderdojo.cix.ie/BTYSTE2016

7.1 Phase 1

In the first phase of this project I analysed A5 on three different musical instruments. I then synthesised those notes and achieved a reasonable likeness.

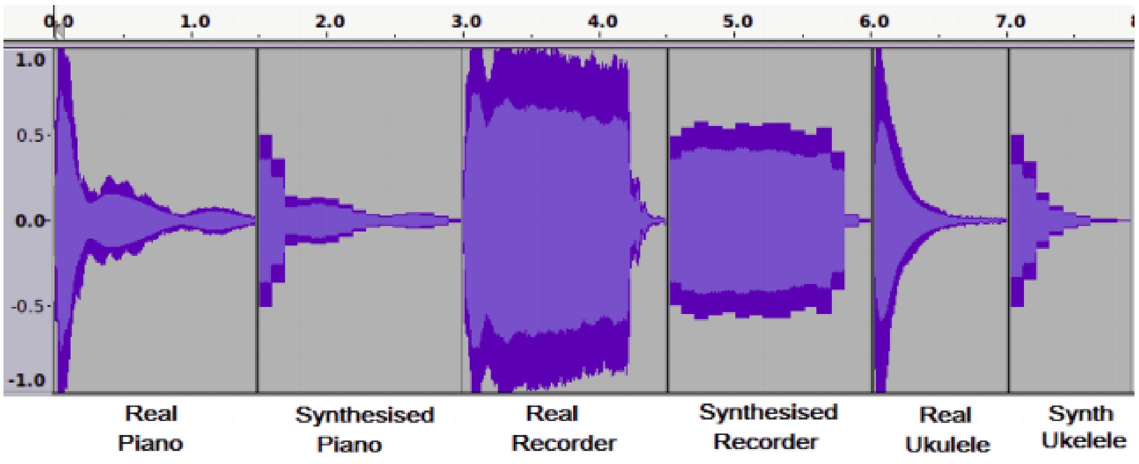


Figure 14 Synthesised Notes

I also synthesised a 32 note tune using extrapolated harmonics. This tune is an original composition designed to use every note in an octave, at least once.

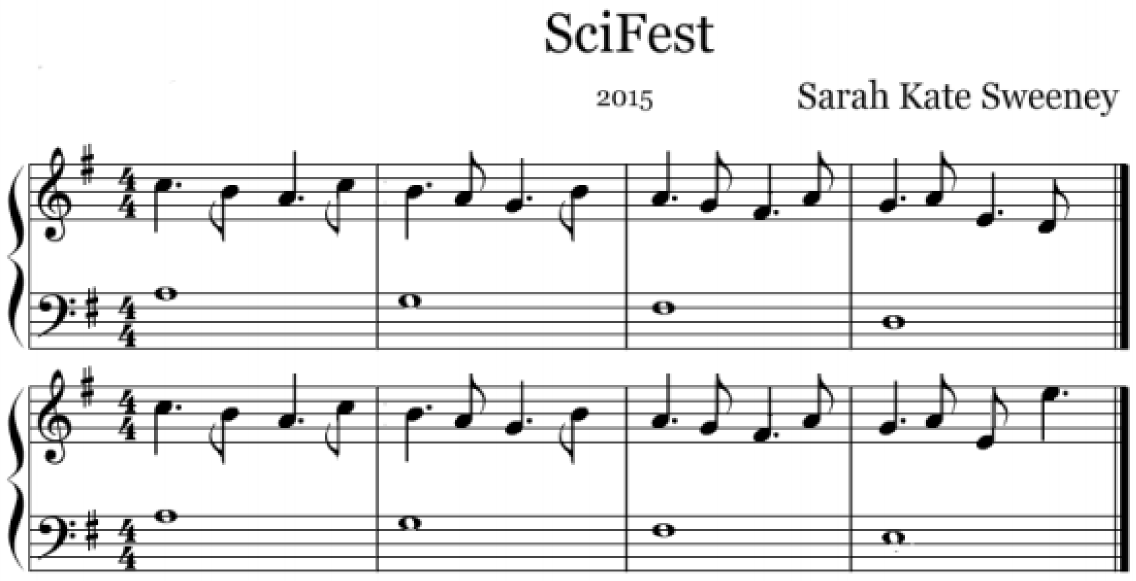


Figure 15 Synthesised Tune

There were a number of problems that needed to be addresses with the results from phase 1.

1 There were 16 harmonics x 2 seconds x 10 windows per note. This required 320 coefficients to be collected per note or 960 coefficients in total for three notes. This took at least two full days of tedious work.
2. The harmonic coefficients were collected from Audacity and were of low resolution.
3. There was a 'slap' audible between windows.
4. Because a ready-made piece of software, Audacity, was used to do the analysis I did not have a deep understanding of the mathematical process behind the analysis.

Clearly these problems needed to be addressed in future work.

7.2 Phase 2

The output of Phase 2 is a series of eight experimental programs that supplied the toolset necessary to deal with the deficiencies identified in Phase 1. These programs are described in Chapter 5 of this report.

7.3 Phase 3

For the third phase of this project, I have written my own software for analysis and synthesis. I used these programs to synthesize Beethoven's 'Ode To Joy'. I chose 'Ode To Joy' because it has four distinct motifs, one for each instrument, it is well known and has a simple melody. The piano open has the first motif, the guitar has the second, the recorder has the third and the ukulele has the fourth. The piano closed plays the accompaniment all the way through.

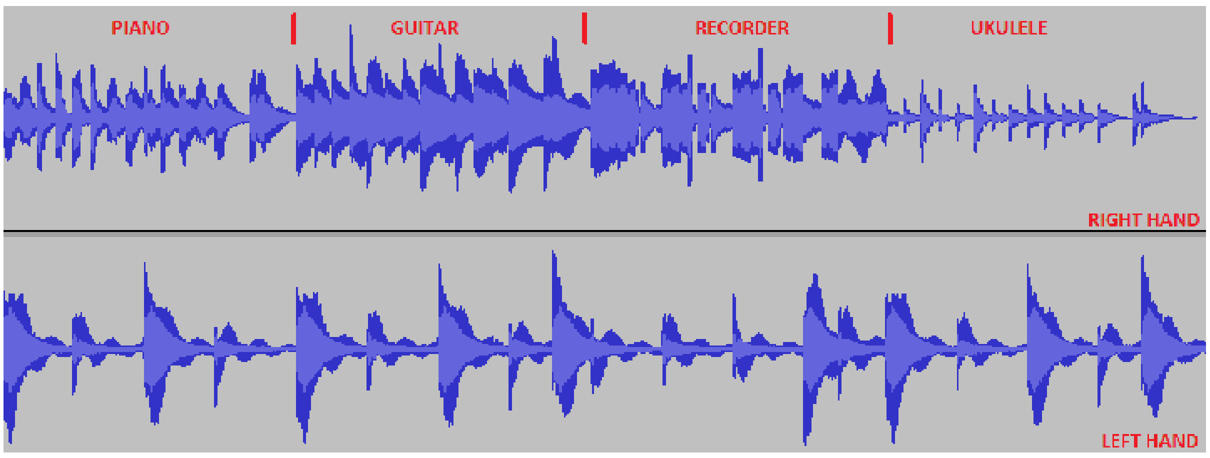


Figure 16 Ode To Joy

8. Conclusions

8.2 General Conclusions

From this project, I conclude that it is possible to reproduce musical instrument timbre using Short Time Discrete Fourier Analysis. This means that music can be stored and transmitted in files that are perhaps 1,000 smaller that the equivalent .wav file. More importantly the music can be modified by the device playing the software. For example a user could pick the instruments or the software could change the tempo at a critical part of a game.

8.2 What I Learned

During this project I have learned a lot about many different things. Here are a few of them:

- How timbre works; from both a musical and scientific perspective.
- Mathematics; geometry and complex numbers mainly.
- Python programming, JSON and web development in HTML.
- The physics of sound.
- Sine waves and Fourier analysis.
- Information Theory including noise and analog and digital signals.
- Musical theory, especially in scales and fifths.
- Report writing.
- Presentation skills.

This project has taught me a lot about music from a scientific perspective, one I would not have seen before. I now understand why instruments sound the way they do, and why a perfect fifth is not perfect. I have also improved my coding skills in Python. Many of the life lessons I learned cannot be taught in a classroom. I strongly believe that this project has improved not only my science aptitude, but I am also a better coder, musician and mathematician as a result.

8.2 Further Work

8.2.1 Overview

I have decided that I am going to continue to study the subject matter of technology and music for the rest of my secondary education. This project is phase 2 of that study. The eventual objective is to develop a coding language called Music Markup Language (MML) that one could code music in the same way one can easily code a website in HTML. This is useful for instrumentals or backing tracks, and very useful for games as one could change the music tempo or timbre in sync with the game action. All of this would be imbedded into the code for the game. An MML file would be tiny compared to the size of an mp3 or a wav file that it could generate. This would be very useful to reduce the bandwidth required for playing online games.

8.2.2 Developing MJSON

At present, this project belongs in the Chemical, Physical and Mathematical Sciences in most Irish science fairs. However, when I begin to develop MML, it will switch to the Technology category. Sometime in the near future, I plan to develop MML from the ground up.

9. Acknowledgements

I would like to thank following people:

- Mr. Foley, my science teacher and Mr. Richard O Shea, a former BTYS winner for helping me prepare for BTYSTE 2016.
- Mr. Hugh McCarthy, for allowing me to use the recording studio at Cork School of Music and for lending me his books.
- My parents, Karen and Jerry. My Dad spent weeks teaching me Python and Fourier Analysis across the summer and helped me hugely with programming.
- My brother, Cian for all of the advice and my sister Bláthnaid, who helped set up my stand at SciFest.

10. References

(Ref 001) A webpage containing all of the sound files I produced.

http://coderdojo.cix.ie/BTYSTE2016

(Ref 002) My brother, Cian Sweeney's research paper that was the starting point for my project.

http://coderdojo.cix.ie/SciFest2015/ResearchPaper_CianSweeney_w1517148.pdf

(Ref 003) Cook, Perry R. (2002) "Real sound synthesis for interactive applications". A K Peters LTD.

(Ref 004) Anna-Maria Hefele overtone singing.

https://www.youtube.com/watch?v=UHTF1-IhuC0&feature=youtu.be

(Ref 005) Duffin, Ross W. (2008)"How equal temperament ruined harmony (and why you should care)".

(Ref 005) Levitin, Daniel J. (2007) "This is your brain on music". Penguin Group.

(Ref 006) Angus, James, Howard, David M. (2007) "Acoustics and phychoacoustics". Penguin Group.

(Ref 007) A series of YouTube videos explaining Information Theory.

https://www.youtube.com/watch?v=p0ASFxKS9sg&feature=youtu.be

(Ref 008) Tacoma Oscillating Bridge

https://www.youtube.com/watch?v=3mclp9QmCGs&feature=youtu.be

(Ref 009) Bobby McFerrin: Power of the Pentatonic Scale

https://www.youtube.com/watch?v=fsO53ydK-yA&feature=youtu.be

Appendices

A.1 Experiment001- Python and Complex Numbers

```
############################################################
# Experiment 1:              Revision Date: Sun, 5 Jul 2015
# How Python handles Complex Numbers.
# Proving that Complex multiplication yields a rotation.
# Sarah Kate Sweeney
############################################################
import math

theta = math.pi/6
sin_theta = math.sin(theta)
cos_theta = math.cos(theta)
print theta, cos_theta, sin_theta

a = cos_theta + 1j*sin_theta
print a

a *= a
print a

#########################################################
# Results
# 0.523598775598 0.866025403784 0.5
# (0.866025403784+0.5j)
# (0.5+0.866025403784j)
#########################################################
# Since the Cos of 30 degrees = Sin of 60 degrees
# and the Sin of 30 degrees = Cos of 60 degrees
# this proves that a squared is a rotation.
#########################################################
```

A.2 Experiment002- FFT Analysis of a square wave

```
#################################################################
# Experiment 2:              Revision Date: Sun 25 Oct 2015
# Using numpy to do an FFT.
# Proving that a0 is 0 for a function centered  on x axis.
# Sarah Kate Sweeney
##################################################################

import numpy.fft as np
from pylab import plot, show, subplot, specgram

signal = [1, 1, 1, 1, 1, 1, 1, 1, -1, -1, -1, -1, -1, -1, -1, -1]
F = np.fft(signal)
print F
print
print abs(F)

subplot(411)
plot(signal)
subplot(412)
plot(abs(F))

signal = [2, 2, 2, 2, 2, 2, 2, 2, 0, 0, 0, 0, 0, 0, 0, 0]
F = np.fft(signal)
print F
print abs(F)

subplot(413)
plot(signal)
subplot(414)
plot(abs(F))
show()

############################################################################
#
#   Results:
#
############################################################################
#  The Complex Fourier Coefficients of a square Wave from 1 to -1
#
############################################################################
# [ 0. +0.j       2.-10.05467898j    0. +0.j       2. -2.99321153j
#   0. +0.j       2. -1.33635728j    0. +0.j       2. -0.39782473j
#   0. +0.j       2. +0.39782473j    0. +0.j       2. +1.33635728j
#   0. +0.j       2. +2.99321153j    0. +0.j       2.+10.05467898j]

############################################################################
#  The Absolute Value Fourier Coefficients of a square Wave from 1 to -1
#
############################################################################
# [ 0.            10.25166179        0.            3.59990489
#   0.             2.40537955        0.            2.03918232
#   0.             2.03918232        0.            2.40537955
#   0.             3.59990489        0.           10.25166179]

############################################################################
#  The Complex Fourier Coefficients of a square Wave from 2 to 0
#
########################################################################
# [16. +0.j         2.-10.05467898j   0. +0.j           2. -2.99321153j
#   0. +0.j         2. -1.33635728j   0. +0.j           2. -0.39782473j
#   0. +0.j         2. +0.39782473j   0. +0.j           2. +1.33635728j
#   0. +0.j         2. +2.99321153j   0. +0.j           2.+10.05467898j]

##########################################################################
#  The Absolute Value Fourier Coefficients of a square Wave from 2 to 0
#
#########################################################################
# [16.          10.25166179         0.           3.59990489
#   0.           2.40537955         0.           2.03918232
#   0.           2.03918232         0.           2.40537955
#   0.           3.59990489         0.          10.25166179]
##########################################################################
```

A.3 Experiment003- Synthesis of a square wave

```
########################################################################
# Experiment 3:                      Revision Date: Sun 25 Oct 2015
# Create audible .wav files.
# Proving that phase shift is inaudible
# Sarah Kate Sweeney
########################################################################
import math
from pylab import plot, show, subplot, specgram
import wave
import struct

f = 440                                 # Frequency A4
w = 2.0 * math.pi * f                   # Angular Velocity
a1 = math.sqrt(1.0/(2.0 * 1.0 * 1.0))   # Fundamental real
b1 = a1                                 # Fundamental imaginary pi/4
r1 = abs(a1 + 1j * b1)
a3 = math.sqrt(1.0/(2.0 * 3.0 * 3.0))   # Third harmonic real
b3 = a3                                 # Third harmonic imaginary pi/4
r3 = abs(a3 + 1j * b3)
a5 = math.sqrt(1.0/(2.0 * 5.0 * 5.0))   # Fifth harmonic real
b5 = a5                                 # Fifth harmonic imaginary pi/4
r5 = abs(a5 + 1j * b5)
a7 = math.sqrt(1.0/(2.0 * 7.0 * 7.0))   # Seventh harmonic real
b7 = a7                                 # Seventh harmonic imaginary pi/4
r7 = abs(a7 + 1j * b7)

X = []
Xshort = []
Y = []
Yshort = []

# Open wav file
music_output = wave.open('experiment003.wav', 'w')
music_output.setparams((2, 2, 44100, 0, 'NONE', 'not compressed'))

# For loop to add up squiggly wave
for sample in range(0, 44100):
    t = sample/44100.00
    y = (a1*math.sin(w*t) + b1*math.cos(w*t)) + (a3*math.sin(3*w*t) +
b3*math.cos(3*w*t)) + (a5*math.sin(5*w*t) + \
        b5*math.cos(5*w*t)) + (a7*math.sin(7*w*t) + b7*math.cos(7*w*t))
    X.append(t)
    Y.append(y)
    if sample < 500:
        Xshort.append(t)
        Yshort.append(y)
    y=int(y*10000)
    #Process wav file
    packed_value = struct.pack('h', y)
    music_output.writeframes(packed_value)
    music_output.writeframes(packed_value)

subplot(211)
plot(Xshort,Yshort)
```

```
X = []
Xshort = []
Y = []
Yshort = []

for sample in range (0,44100):
    packed_value = struct.pack('h', 0)
    music_output.writeframes(packed_value)
    music_output.writeframes(packed_value)

# For loop to add up square wave
for sample in range(0, 44100):
    t = sample/44100.00
    y = (r1*math.sin(w*t)) + (r3*math.sin(3*w*t)) + (r5*math.sin(5*w*t)) +
(r7*math.sin(7*w*t))
    X.append(t)
    Y.append(y)
    if sample < 500:
        Xshort.append(t)
        Yshort.append(y)
    y=int(y*10000)
    #Process wav file
    packed_value = struct.pack('h', y)
    music_output.writeframes(packed_value)
    music_output.writeframes(packed_value)

music_output.close()

# Plot and display graph
subplot(212)
plot(Xshort, Yshort)
show()
```

A.4 Experiment004- Reading a .wav file

```
########################################################################
# Experiment 4:                         Revision Date: Sun 25 Oct 2015
# Reading a wav file
# Sarah Kate Sweeney
########################################################################

import scipy.io.wavfile as wavfile

rate,data=wavfile.read('recorder.wav')

print data[:5]
print rate

############################################################
# Results
############################################################
# The first 5 stereo values from the wav file
############################################################
# [[    0     0]
# [3731 3731]
# [6493 6493]
# [8400 8400]
# [9997 9997]]
############################################################
# Sample Rate
############################################################
# 44100
############################################################
```

A.5 Experiment005- Creating a spectrogram

```
######################################################################
# Experiment 5:                          Revision Date: Sun 25 Oct 2015
# Creating a spectrogram.
# To compare a real and synthesised note.
# Sarah Kate Sweeney
######################################################################
from scipy.io.wavfile import read, write
from pylab import plot, show, subplot, specgram

rate, datastereo = read('results.wav')    # reading

#  data = [0] * (len(datastereo))
data = [0] * 60000

for n in range (60000):
    data[n] = datastereo[n, 0] * 2


subplot(311)
plot(range(len(data)), data)
#  NFFT is the number of data points used in each block for the FFT
#  and noverlap is the number of points of overlap between blocks
subplot(312)
specgram(data, NFFT=512, noverlap=0)
subplot(313)
specgram(data, NFFT=1024, noverlap=0)

show()
```

A.6 Experiment006- JSON object

```
######################################################################
# Experiment 006            Date 2nd Dec 2015
#
# Puts a dict into a JSON object then reads it back into a dict
#
######################################################################

import json

before_dict = {'noteA': {0: 22000, 1: 16000, 2: 12000},
               'noteB': {0: 20000, 1: 14000, 2: 11000}}
print before_dict

json_data = json.dumps(before_dict)
print json_data

after_dict = json.loads(json_data)
print after_dict

print after_dict['noteB']
print after_dict['noteB']['1']

###########################################################################
# {'noteA': {0: 22000, 1: 16000, 2: 12000}, 'noteB': {0: 20000, 1: 14000,
2: 11000}}
# {"noteA": {"0": 22000, "1": 16000, "2": 12000}, "noteB": {"0": 20000,
"1": 14000, "2": 11000}}
# {u'noteA': {u'1': 16000, u'0': 22000, u'2': 12000}, u'noteB': {u'1':
14000, u'0': 20000, u'2': 11000}}
# {u'1': 14000, u'0': 20000, u'2': 11000}
# 14000
##########################################################################
```

A.7 Experiment007- Outputting Data to a .wav file

```
###########################################
## Experiment 7          November 12th 2015
## Export WAV file
###########################################

import wave
import struct

# Open wav file
music_output = wave.open('experiment007.wav', 'w')
music_output.setparams((2, 2, 44100, 0, 'NONE', 'not compressed'))

# Frequency = 44,100 / 24 = 1.8375 kHz
signal = [10000, 10000, 10000, 10000, 10000, 10000, 10000, 10000, 10000,
10000, 10000, 10000,
     -10000, -10000, -10000, -10000, -10000, -10000, -10000, -10000, -
10000, -10000, -10000, -10000]

# 10,000 cycles = 5.44 seconds
signal *= 10000

#   Process wav file
for i in range(0, len(signal)):
    packed_value = struct.pack('h', signal[i])
    music_output.writeframes(packed_value)
    music_output.writeframes(packed_value)

music_output.close()
```

A.8 Experiment008- DFT and IDFT

```
###################################################################
# Experiment 008                   Revision Date: Sun 25 Oct 2015
# Perform an FT
# Sarah Kate Sweeney
###################################################################

import math
import numpy.fft


def dft(f_in):
    F_in = []
    N = len(f_in)
    for k in range(0, N):
        Ftemp = 0
        for n in range(0, N):
            Ftemp += (f_in[n] * math.cos(2*n*math.pi*k/N)) - (1j*f_in[n] *
math.sin(2*n*math.pi*k/N))
        F_in.append(Ftemp)
    return F_in


def idft(F_in):
    f_in = []
    N = len(F_in)
    for n in range(0, N):
        ftemp = 0
        for k in range(0, N):
            ftemp += (F_in[k].real * math.cos(2*n*math.pi*k/N)) -
(F_in[k].imag * math.sin(2*n*math.pi*k/N))
        f_in.append(round(ftemp/N))
    return f_in


f = [0, 5, -2, 4, 3, 8, 3, 3, 0, -3, -3, -3, -3, -4, -5, 2, 7]
F = dft(f)
nF = numpy.fft.fft(f)

print len(f), f
print len(F), F
print len(nF), nF

f = idft(F)

print len(f), f
```

```
##########################################################################
#   Results:
# 17 [0, 5, -2, 4, 3, 8, 3, 3, 0, -3, -3, -3, -3, -4, -5, 2, 7]
# 17 [(12+0j), (12.232264386303726-30.687369918596566j),
#     (3.405212375849918+14.87833871961477j),
(12.851192352366573+11.058421190976881j),
#     (0.46007610144805644+7.715389988173088j), (-7.575664743143951-
0.993300929005895j),
#     (-8.189744000238928-0.6935734094248822j), (-4.2088892346013544-
9.822734337956419j),
#     (-14.974447237984048-7.290448671589815j), (-
14.974447237984048+7.290448671589817j),
#     (-4.20888923460146+9.822734337956376j), (-
8.189744000238864+0.693573409424725j),
#     (-7.575664743143901+0.993300929005847j), (0.46007610144800215-
7.715389988173072j),
#     (12.851192352366615-11.058421190976851j), (3.4052123758500445-
14.878338719614746j),
#     (12.232264386303658+30.687369918596595j)]
# 17 [ 12.00000000 +0.j          12.23226439-30.68736992j
#   3.40521238+14.87833872j  12.85119235+11.05842119j
#   0.46007610 +7.71538999j  -7.57566474 -0.99330093j
#  -8.18974400 -0.69357341j  -4.20888923 -9.82273434j
# -14.97444724 -7.29044867j -14.97444724 +7.29044867j
#  -4.20888923 +9.82273434j  -8.18974400 +0.69357341j
#  -7.57566474 +0.99330093j   0.46007610 -7.71538999j
#  12.85119235-11.05842119j   3.40521238-14.87833872j
#  12.23226439+30.68736992j]
# 17 [0.0, 5.0, -2.0, 4.0, 3.0, 8.0, 3.0, 3.0, 0.0, -3.0, -3.0, -3.0, -3.0,
-4.0, -5.0, 2.0, 7.0]
##########################################################################
```

A.9 Analysis Program

```
################################################################
# Analyse:                  Revision Date:  Sun 20 Dec 2015
# Analysing the imported signal.
################################################################

import scipy.io.wavfile as wavfile
import numpy.fft as np
import json

samples = 441000    # Samples per note
notes = 88          # Number of notes to collect
windows = 100       # 100 Windows in the sample
resolution = 10     # Gap between buckets in Hz
harmonics = 20      # Number of harmonics to collect


#################################################################
# harmonic_calculate   (Harmonic Calculator
# Returns the freq of a particular harmonic for a particular note
#################################################################

# Returns frequency for 88 (0 to 87) notes and  20 ( fundamental and 19 )
harmonics
# The resolution is the separation between buckets 100ms window = 10Hz
separation
# The frequency is rounded to the nearest bucket
# Def is a function that can be called later.

def frequency_rounded(note, harmonic):
    fundamental = 27.5 * 1.0594630944 ** note
    frequency = fundamental * (harmonic+1)
    answer = resolution * int((frequency + (resolution/2))/resolution)
    return answer


def note_analyse():
    for w in range(0, windows):

        # First get the Fourier transform of the window.
        F = np.fft(data[n * 441000 + w * 4410: (n * 441000 + (w + 1) *
4410)])

        # Remove the complex numbers by getting the absolute value.
        Fabs = abs(F)
        # Now get the amplitude of the fundamental and harmonics
        hardata = {}
        for h in range(0, harmonics):
            bucketf = frequency_rounded(n + start_offset, h)
            if bucketf > 18000:
                  # use amplitude 0 for Shannon Nyquist cutoff
                hardata["harmonic{0}".format(h)] = 0
            else:
                amplitude = Fabs[bucketf/resolution]

                # Allow for tuning variability by looking for local maximum
within 2% of the frequency
                delta = int((bucketf * 0.02)/10)
                for d in range(-delta, delta):
                    if Fabs[d + bucketf/resolution] > amplitude:
                        amplitude = Fabs[d + bucketf/resolution]

                hardata["harmonic{0}".format(h)] = amplitude
        windata["window{0}".format(w)] = hardata

# End of function declarations, program starts here.
# Analyse each instrument separately within the loop

instruments = [
    ["../ReferenceNotes/pianoOpen.wav", 88, 0,
"../Coefficients/pianoOpen.json"],
    ["../ReferenceNotes/pianoClosed.wav", 88, 0,
"../Coefficients/pianoClosed.json"],
    ["../ReferenceNotes/guitarA.wav", 13, 23,
"../Coefficients/guitarA.json"],
    ["../ReferenceNotes/guitarB.wav", 12, 25,
"../Coefficients/guitarB.json"],
    ["../ReferenceNotes/guitarD.wav", 12, 13,
"../Coefficients/guitarD.json"],
    ["../ReferenceNotes/guitarG.wav", 12, 45,
"../Coefficients/guitarG.json"],
    ["../ReferenceNotes/guitarHighE.wav", 12, 54,
"../Coefficients/guitarHighE.json"],
    ["../ReferenceNotes/guitarLowE.wav", 12, 30,
"../Coefficients/guitarLowE.json"],
    ["../ReferenceNotes/recorder.wav", 13, 50,
"../Coefficients/recorder.json"],
    ["../ReferenceNotes/ukulele.wav", 23, 45,
"../Coefficients/ukulele.json"]
    ]
```

```
for i in range(len(instruments)):

    # gather the variables
    instrument = instruments[i]
    notedata = {}
    input_wav = instrument[0]
    note_count = instrument[1]
    start_offset = instrument[2]
    output_json = instrument[3]

    # read the Reference Notes
    rate, data = wavfile.read(instrument[0])
    for n in range(1, note_count + 1):
        windata = {}
        note_analyse()
        notedata["note{0}".format(n + start_offset)] = windata

    # save the Fourier Coefficients to json file
    with open(output_json, 'w') as outfile:
        json.dump(notedata, outfile)
```

A.10 Synthesis Program

```
###################################################################
# Synth :                    Revision Date:  Sat 22 Dec 2015
# Synthesising Music.
###################################################################

import json
import wave
import math
import struct

harmonics = 20

harmonic_amplitudes = [[0 for harmonic in range(harmonics)] for window in
range(100)]


def get_harmonic_amplitudes():
    if instrument == 'pianoOpen':
        data = open('../Coefficients/pianoOpen.json', 'r')
    elif instrument == "pianoClosed":
        data = open('../Coefficients/pianoClosed.json', 'r')
    elif instrument == "guitarA":
        data = open('../Coefficients/guitarA.json', 'r')
    elif instrument == "guitarB":
        data = open('../Coefficients/guitarB.json', 'r')
    elif instrument == "guitarD":
        data = open('../Coefficients/guitarD.json', 'r')
    elif instrument == "guitarG":
        data = open('../Coefficients/guitarG.json', 'r')
    elif instrument == "guitarHighE":
        data = open('../Coefficients/guitarHighE.json', 'r')
    elif instrument == "guitarLowE":
        data = open('../Coefficients/guitarLowE.json', 'r')
    elif instrument == "recorder":
        data = open('../Coefficients/recorder.json', 'r')
    elif instrument == "ukulele":
        data = open('../Coefficients/ukulele.json', 'r')
    else:
        data = open('../Coefficients/pianoOpen.json', 'r')

    json_data = data.read()
    instrument_data = json.loads(json_data)
    nte_data = instrument_data["note{0}".format(note)]
    for win in range(0, 100):
        for har in range(0, harmonics):
            win_data = nte_data["window{0}".format(win)]
            har_data = win_data["harmonic{0}".format(har)]
            harmonic_amplitudes[win][har] = har_data

    return harmonic_amplitudes


# Read the song to be synthesised
song_file = open('../SheetMusic/music.json', 'r')
song_json = song_file.read()
song_score = json.loads(song_json)

# Find the length of the song
note_count, song_length = 0, 0
for each_note in song_score:
    print each_note, note_count, song_length
    note_data = song_score["note{0}".format(note_count)]
    if note_data["end"] > song_length:
        song_length = note_data["end"]
    note_count += 1

print note_count, "notes    ", song_length / 100, "seconds"

# Set all samples in the wav file to zero
song_left = []
song_right = []
for i in range((song_length * 441) + 441):  # Extra 10ms for slap tail
    song_left.append(0)
    song_right.append(0)

# Process notes one by one

for n in range(0, note_count):
    note_data = song_score["note{0}".format(n)]
    note = note_data["note"]
    instrument = note_data["instrument"]
    loudness = note_data["loudness"]
    start_note = note_data["start"] * 441
    end_note = note_data["end"] * 441
    channel = note_data["channel"]

    get_harmonic_amplitudes()

    note_end = 0
    for w in range(0, int((end_note - start_note)/4410)):
        print "Note ", n, "Window  ", w

        start_window = start_note + w * 4410            # Used to create
tail to elimenate slap
        end_window = start_note + (w + 1) * 4410 + 441  # Used to create
tail to elimenate slap

        for h in range(0, harmonics):
            frequency = int((27.5 * 1.0594630944 ** note)  * (h + 1))
            f2pi = frequency * 2 * math.pi              # Omega = 2 * Pi *
f

            for sample in range(start_window, end_window):
                t = sample / float(44100)
                signal = int(math.sin(f2pi * t) * loudness *
harmonic_amplitudes[w][h])

                # If the window is starting then ramp in over 10ms
                if sample < (start_window + 441):
                    signal *= ((sample - start_window) / float(441))

                # If the window is ending then ramp out over 10ms
                if sample > (end_window - 441):
                    signal *= ((end_window - sample) / float(441))

                if channel == "left":
                    song_left[sample] += signal
                else:
                    song_right[sample] += signal


# Find the scaling factor to prevent clipping

absolute_max = 0
for i in range(0, song_length * 441):
    if abs(song_left[i]) > absolute_max:
        absolute_max = abs(song_left[i])
    if abs(song_right[i]) > absolute_max:
        absolute_max = abs(song_right[i])

scale = absolute_max / 28000          # wav = 2**16 = +- 2**15 = -32,768 to
+ 32,767
print "Absolute Maximum ", absolute_max, "  Scale  ", scale


# Save the wav file to disk

music_output = wave.open('../SynthMusic/music.wav', 'w')
music_output.setparams((2, 2, 44100, 0, 'NONE', 'not compressed'))

for i in range(0, song_length * 441):

    # Mix a bit of the left with right and visa versa for headphone usage
    value_left = int(song_left[i] / scale) + int((song_right[i] / scale) *
0.05)
    value_right = int(song_right[i] / scale) + int((song_left[i] / scale) *
0.05)

    packed_value_right = struct.pack('h', value_right)
    packed_value_left = struct.pack('h', value_left)
    music_output.writeframes(packed_value_right)
    music_output.writeframes(packed_value_left)

music_output.close()
```

A.11 Reference Notes

Using a high quality microphone in an acoustic studio at Cork School of Music, I recorded 285 reference notes for harmonic analysis using a discrete Fourier Transform.

Instrument	Note Count
Piano Lid Open	88
Piano Lid Closed	88
Guitar	78
Ukulele	23
Recorder	8
Total	285

Figure 17 Reference Note Summary

#	Note	Freq	PiOp	PiCl	GuLE	GuA	GuB	GuD	GuG	GuHE	Ukul	Recor
0	A1	27.5	Yes	Yes								
1	As1/Bf1	29.135	Yes	Yes								
2	B1	30.868	Yes	Yes								
3	C1	32.703	Yes	Yes								
4	Cs1/Df1	34.648	Yes	Yes								
5	D1	36.708	Yes	Yes								
6	Ds1/Ef1	38.891	Yes	Yes								
7	E1	41.203	Yes	Yes								
8	F1	43.654	Yes	Yes								
9	Fs1/Gf1	46.249	Yes	Yes								
10	G1	48.999	Yes	Yes								
11	Gs1/Af2	51.913	Yes	Yes								
12	A2	55	Yes	Yes								
13	As2/Bf2	58.27	Yes	Yes								
14	B2	61.735	Yes	Yes								
15	C2	65.406	Yes	Yes								
16	Cs2/Df2	69.296	Yes	Yes								
17	D2	73.416	Yes	Yes								
18	Ds2/Ef2	77.782	Yes	Yes								
19	E2	82.407	Yes	Yes								
20	F2	87.307	Yes	Yes								
21	Fs2/Gf2	92.499	Yes	Yes								
22	G2	97.999	Yes	Yes								
23	Gs2/Af3	103.83	Yes	Yes								
24	A3	110	Yes	Yes		Yes						
25	As3/Bf3	116.54	Yes	Yes		Yes						
26	B3	123.47	Yes	Yes		Yes						
27	C3	130.81	Yes	Yes		Yes						
28	Cs3/Df3	138.59	Yes	Yes		Yes						
29	D3	146.83	Yes	Yes		Yes						
30	Ds3/Ef3	155.56	Yes	Yes		Yes						
31	E3	164.81	Yes	Yes	Yes	Yes						
32	F3	174.61	Yes	Yes	Yes	Yes						
33	Fs3/Gf3	185	Yes	Yes	Yes	Yes						
34	G3	196	Yes	Yes	Yes	Yes						

35	Gs3/Af4	207.65	Yes	Yes	Yes	Yes						
36	A4	220	Yes	Yes	Yes	Yes						
37	As4/Bf4	233.08	Yes	Yes	Yes							
38	B4	246.94	Yes	Yes	Yes		Yes					
39	C4	261.63	Yes	Yes	Yes		Yes					
40	Cs4/Df4	277.18	Yes	Yes	Yes		Yes					
41	D4	293.67	Yes	Yes	Yes		Yes	Yes				
42	Ds4/Ef4	311.13	Yes	Yes	Yes		Yes	Yes				
43	E4	329.63	Yes	Yes	Yes		Yes	Yes				
44	F4	349.23	Yes	Yes			Yes	Yes				
45	Fs4/Gf4	369.99	Yes	Yes			Yes	Yes				
46	G4	392.00	Yes	Yes			Yes	Yes	Yes			
47	Gs4/Af5	415.3	Yes	Yes			Yes	Yes	Yes		Yes	
48	A5	440	Yes	Yes			Yes	Yes	Yes		Yes	
49	As5/Bf5	466.16	Yes	Yes			Yes	Yes	Yes		Yes	
50	B5	493.88	Yes	Yes			Yes	Yes	Yes		Yes	
51	C5	523.25	Yes	Yes				Yes	Yes		Yes	Yes
52	Cs5/Df5	554.37	Yes	Yes				Yes	Yes		Yes	Blank
53	D5	587.33	Yes	Yes				Yes	Yes		Yes	Yes
54	Ds5/Ef5	622.25	Yes	Yes					Yes		Yes	Blank
55	E5	659.26	Yes	Yes					Yes	Yes	Yes	Yes
56	F5	698.46	Yes	Yes					Yes	Yes	Yes	Yes
57	Fs5/Gf5	739.99	Yes	Yes					Yes	Yes	Yes	Blank
58	G5	783.99	Yes	Yes					Yes	Yes	Yes	Yes
59	Gs5/Af6	830.61	Yes	Yes						Yes	Yes	Blank
60	A6	880	Yes	Yes						Yes	Yes	Yes
61	As6/Bf6	932.33	Yes	Yes						Yes	Yes	Blank
62	B6	987.77	Yes	Yes						Yes	Yes	Yes
63	C6	1046.5	Yes	Yes						Yes	Yes	Yes
64	Cs6/Df6	1108.7	Yes	Yes						Yes	Yes	
65	D6	1174.7	Yes	Yes						Yes	Yes	
66	Ds6/Ef6	1244.5	Yes	Yes						Yes	Yes	
67	E6	1318.5	Yes	Yes						Yes	Yes	
68	F6	1396.9	Yes	Yes							Yes	
69	Fs6/Gf6	1480	Yes	Yes							Yes	
70	G6	1568	Yes	Yes								

71	Gs6/Af7	1661.2	Yes	Yes								
72	A7	1760	Yes	Yes								
73	As7/Bf7	1864.7	Yes	Yes								
74	B7	1975.5	Yes	Yes								
75	C7	2093	Yes	Yes								
76	Cs7/Df7	2217.5	Yes	Yes								
77	D7	2349.3	Yes	Yes								
78	Ds7/Ef7	2489	Yes	Yes								
79	E7	2637	Yes	Yes								
80	F7	2793	Yes	Yes								
81	Fs7/Gf7	2960	Yes	Yes								
82	G7	3136	Yes	Yes								
83	Gs7/Af8	3324.4	Yes	Yes								
84	A8	3520	Yes	Yes								
85	As8/Bf8	3729.3	Yes	Yes								
86	B8	3951.1	Yes	Yes								
87	C8	4186	Yes	Yes								

Figure 18 Reference Note Detail