Audio Fingerprinting

Review of audio fingerprinting algorithms and looking into a multi-faceted approach to fingerprint generation

Nick Palmer A763896

Supervisor: Professor James L Alty

May 11

This Project involves researching, developing and implementing an algorithm to generate, store and search for audio fingerprints. After initial research into audio fingerprinting showed that almost all current audio fingerprinting methods base the fingerprint purely on frequency/time data I decided to develop a method taking other musical features, such as the key, tempo and time signature, into account with a view to increase search speed and decrease the number of false positive matches. Testing how this compared to a standard fingerprint revealed only minor improvement in search times and while results contained less false positives with the augmented fingerprint, they also returned no matches more frequently. Reasons for which are explained however and future improvement should be possible.

Contents

1 Introduction 3

1.1 What I'm trying to do 3

1.2 Needs & Uses of Audio Fingerprinting 3

1.3 Current Methods 4

1.4 How I plan to improve 4

2 Literature Review 5

3 Design 10

3.1 What I'm trying to do 10

3.2 Criteria for success 11

3.3 Environment 11

4 Experimentation 12

4.1 Reading Audio 12

4.2 Spectrograms 13

4.3 Reading in wave files 16

4.4 Graphical User Interface 17

4.5 Databases 17

4.6 Fingerprinting 17

4.7 Other Factors 18

5 Detailed Design 20

5.1 Development Choices 20

5.2 UML 21

5.3 HCI Considerations 25

5.4 Database Design 27

5.5 Third Party Libraries 28

5.6 Development Process 29

6 Implementation 30

6.1 Audio Readers 30

6.2 Waveform Window 30
6.3 Spectrogram Window 31
6.4 Main window 33
6.5 Basic Fingerprinting 34
6.6 Basic Fingerprint Storing & Searching 34
6.7 Augmented Fingerprint Storing & Searching 35
6.8 Database Handling 36
7 Testing 37
7.1 Results 37
8 Conclusions 39
8.1 Looking to the future 40
Bibliography 41
Appendices 43
Appendix A - Database SQL 43
Appendix B – User Manual 44
Appendix C – Gantt Chart 46
Appendix D – Source Code Listing 47

1 Introduction

Audio fingerprinting is the term used to describe methodologies used to process audio data to generate a representational hash, known as the audio fingerprint, based on the perceived sound rather than the data the audio is stored as. Doing so requires reading of audio files, taking them from their often compressed formats, transforming them back into a stream of time-based waveform amplitude data, transforming that into a series of chunks containing frequency data, finding defining features and generating the fingerprint based on their values in frequency, time and relation to each other.

There are currently a handful of well-known and robust audio fingerprinting algorithms developed by companies for a variety of uses. Most current algorithms are built for speed and only tend to use features found through spectral analysis of the audio to generate their fingerprints.

1.1 What I'm trying to do

For this project I aim to implement an audio fingerprinting algorithm in the style of many currently designed methods and then develop an alternative version that takes into other aspects of the music which I could add to, or "augment", the fingerprint that might aide recognition accuracy and speed.

The resulting software I produce should be able to process an audio file and, using either a basic or augmented method, generate a fingerprint that can either be stored in a database for later retrieval or searched for in a database attempting to find matched values.

1.2 Needs & Uses of Audio Fingerprinting

Audio fingerprinting has many applications in both the private business and public domains. The most popular use is through software developed by companies like Shazam, http://www.shazam.com/, and SoundHound, http://www.soundhound.com/, who both offer mobile solutions for audio fingerprinting which can be used through a mobile phone for identifying music overhead in a public place such as a car or a café.

Gracenote, http://www.gracenote.com/, is another company who also offer mobile software for audio identification but also has software available for digital media players, like iTunes and Winamp so that music can be properly named, searched, grouped and identified. Gracenote can also be used for voice recognition and is currently in use with Ford's SYNC in-car communications system (GRACENOTE, 2011).

Last.fm, http://www.last.fm/, uses audio fingerprinting to try and correct inappropriate music metadata when it "scrobbles" (the term last.fm has for when it makes a record of a user listening to a track) (LAST.FM, 2007).

Uses for audio fingerprinting extend beyond the general public managing music libraries and wanting to identify songs however. Audio fingerprinting can be used by composers and artists to try and identify parts of songs and riffs that an artist or composer wishes to check isn't already copyrighted. Police and forensic investigators can, and do, use audio fingerprinting to identify stolen music on the computers of those suspected of copyright infringement. If a track of unreleased music is leaked to the public, music producers can use audio fingerprinting to identify the exact revision of the track to help identify when and who leaked the track. Similar to the police identifying files on a suspect's computer, YouTube, http://www.youtube.com/, operate a system called Content ID (YOUTUBE, 2008) which in turn implements an audio fingerprinting system developed by Audible Magic, http://www.audiblemagic.com/. Content ID helps copyright owners identify videos on YouTube that might use either video or audio which they own. Content owners can then either monetise the uses through advertising, track the viewing metrics or remove offending videos.

You can also fingerprint various phonetic sounds, then by fingerprinting a stream of voice audio, do rudimentary voice recognition. The same can be done for individual voice samples to identify the speaker.

1.3 Current Methods

Most current methods tend to be patented with few technical details, if developed in a country that allows software patents, or company specific trade secrets. However, many academic papers exist on the topic of identifying audio through audio fingerprinting and some companies, like Shazam, have also produced white papers giving an overview of the algorithms they have created.

1.4 How I plan to improve

I plan to implement a fairly rudimentary fingerprinting algorithm. Once this algorithm is working I intend to experiment by adding in other music feature analysis, such as tempo or musical key, which I hope will narrow down the search field and reduce false positive matches.

2 Literature Review

In the past few years there has been a significant interest taken in generating new ways to index and search the new forms of media that many people now have on their personal computers. As people now often have hundreds to thousands of individual multimedia files. Indexing and searching through them to be able to identify all these files is a challenge which is why multimedia fingerprinting was created, to give users more humanistic search controls and to improve indexing.

Common software based methods to identify files focus on the raw data of the file, often hashing the bytes with an algorithm such as MD5. While this created a mostly unique hash for each individual file, it doesn't take into account the human perception of multimedia. Two photos might be visually almost indistinguishable, but if they have any slight difference in hue of any pixel, or are stored in different formats, then the MD5 hashes generated will not identify the visual similarity, only the data. Which in that case would be different in part or wholly, which would make for two non-comparable MD5 hashes.

Instead, using multimedia fingerprinting, the algorithm is based upon the perception of the data by the user. In the case of audio fingerprinting it is often accomplished by analysing the frequencies present in the audio and what time they appear in the file or in relation to each other. This is similar to a hashing function, but allows means that though the physical data for two files might be wholly different, the fingerprint should be identical, or at least comparable by some method, if both files are perceived by the user as being the same or similar.

In a paper by Jaap Haitsma and Ton Keller entitled "A Highly Robust Audio Fingerprinting System" (HAITSMA, Jaap and Kalker, Ton, 2002) they list the main parameters for an audio fingerprinting system as robustness, how well the algorithm can identify audio with severe degradation and distortion; Reliability, how often is the correct song incorrectly identified, known as a false positive; Fingerprint size, how much storage is required for the fingerprint? Fast searching and storage relies on small fingerprint size; Granularity, how long must a piece of audio be before it can be correctly identified and finally Search speed and scalability, how long does it take to find matches and how would this be affected by the addition of more fingerprints into the database.

To come up with a good audio fingerprint the algorithm strives to be robust, reliable, have small output, require only a small sample of the audio and be, and keep being, fast to search.

There are challenges to generating a fingerprint however. In the lecture slides from Columbia Universities Electrical Engineering course written by Dan Ellis (ELLIS, Dan, 2011) he identifies that channel noise, noise added from the speakers/microphones; Recognising short fragments being the goal; Having a large, data wise, reference item to search for and false positives are all problems to overcome when designing an audio fingerprinting solution.

The most common and evidently the most robust current algorithms to generate a fingerprint based on audio rely on the idea of landmarks in the audio. The algorithm Shazam, http://www.shazam.com/, a widely used audio fingerprinting solution for mobile devices, uses is described in a paper by Avery Li-Chun Wang who works for Shazam. The paper, entitled “An Industrial Strength Audio Search Algorithm” (WANG, Avery Li-Chun, 2003) details how they transform the audio data from time domain, where the stream of bytes is storing the amplitude of the audio wave, to the frequency domain, where the stream is compartmentalised into chunks of data and then converted so that the individual frequencies present in the chunk are identified, and use the frequency magnitudes and time data to compute the hash as seen in Figure 1 overleaf.

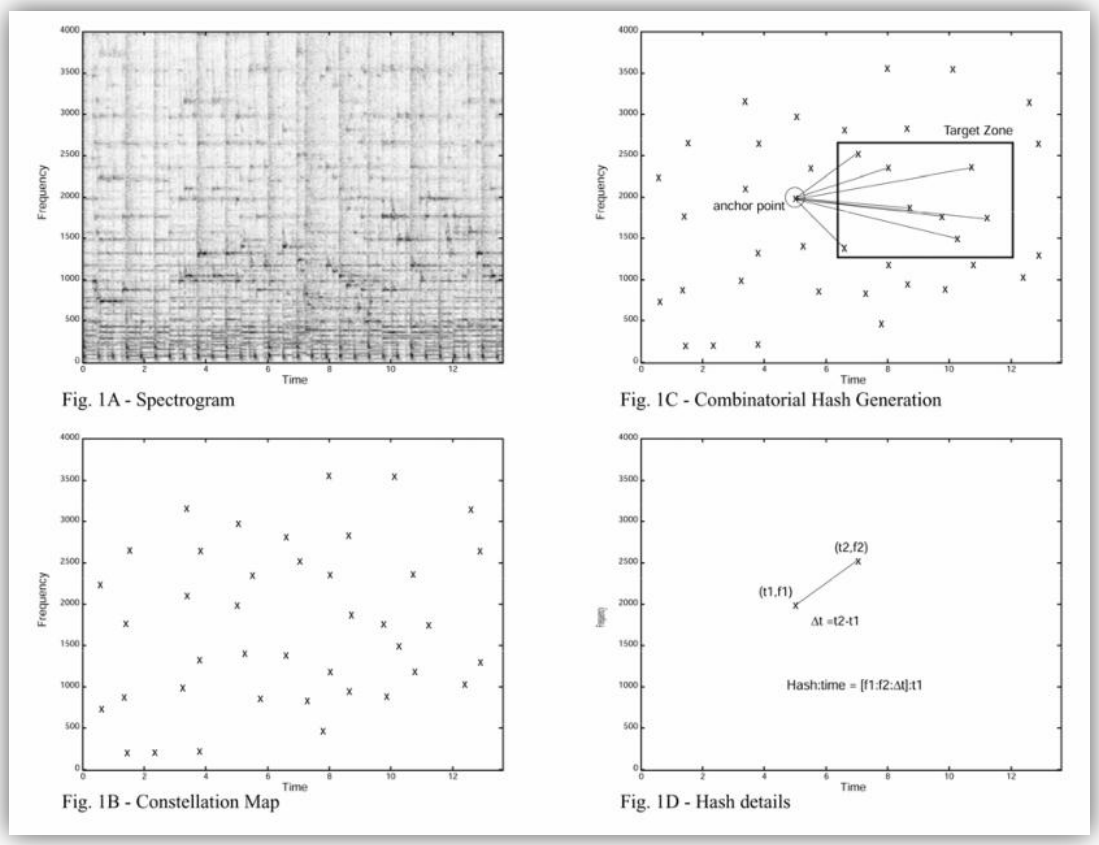


Figure 1 - Image taken from (WANG, Avery Li-Chun, 2003)

In Fig. 1A they have converted the data into frequency domain where you can see high magnitudes of certain frequencies standing out as darker areas on the graph. In Fig. 1B the algorithm has chosen the most important features from the graph where importance is defined as a time-frequency point which has a higher magnitude than all points around it in a given area. Whereas you could store the data from Fig. 1B in a database and search based on that searching for matches would not be as efficient as it could be. Instead Shazam take points, shown as an anchor point in Fig. 1C, and then generates the hash based on the relation of the anchor point to other points in the "Constellation Map" as shown in Fig. 1D.

Once Shazam has their database of hashes a search can be conducted by finding correlations between a sample and database series of hash values.

By doing this Shazam have created an algorithm that is fast enough to be processed on a mobile phone, only sending the hash data to Shazams servers to identification. Their method of finding important time/frequency features also takes into account the distortion and low quality recording that mobile devices will add to the sample.

Shazam is not the only documented audio fingerprinting algorithm. Music Brainz, http://musicbrainz.org/, is a community service that has publicly available data on music files and services to manage and update metadata for music based on fingerprinting. By fingerprinting unknown music files on your computer with the software Music Brainz supplies a user can find out the details of that piece of music.

The algorithm Music Brainz currently uses is called PUID which is unfortunately closed source and not well documented. However, their previous fingerprinting solution is open source and well documented.

The Future Proof Fingerprint, as it was called (evidently not all that future proof as it's since been succeeded by the PUID technology), is proposed as a solution on a page available on the Music Brainz website (MUSIC BRAINZ, 2006) and a function description giving details on a proposed solution for the proposal is also listed (SCHMIDT, Geoff, 2005).

The Future Proof Fingerprint operates in much the same way as Shazam, only it defines a "codebook" beforehand which feature points from frequency domain data can be mapped to, instead of Shazams method of finding relations between feature points.

By doing this the Future Proof Fingerprint algorithm generates what is essentially a highly compressed representation of the audio. By doing this for several overlapping samples of the audio and "jittering" the sample window around when you search you should be able to reliably find matches despite timing distortions. Meaning a 10 second sample should be simply found as a match to 10 seconds in the middle of a 2 hour recording. Also, the data format for the fingerprints is kept open, allowing future modification without needing to re-construct the entire database.

Every fingerprinting algorithm I came across during my research was purely based on frequency domain data being hashed using various algorithms and searched for. To get the data from time domain to frequency domain a transform is done on the data called the Discrete Fourier Transform (DFT). The most widely used algorithm to perform DFT is called a Fast Fourier Transform (FFT) which is an implementation to compute the DFT and its inverse. FFT is a way to compute the same result as doing a DFT, only in less time but at the expense of approximation error. The error introduced however can be made arbitrarily small by increasing the computations performed.

FFT is not the only method that can be used to convert the time domain data into the non-time-based frequency domain where frequency values only are returned. You can use

Wavelets, "*mathematical functions that cut up data into different frequency components, and then study each component with a resolution matched to its scale"* (IEEE, 1995). While wavelet transforms might have offered better results for searching the spectrogram images for feature points, as wavelet transforms after often used in computer vision fields, I could only find one major publication that used wavelet transforms for audio fingerprinting.

Content Fingerprinting Using Wavelets, a paper by Shumeet Baluja and Michele Covell from Google (BALUJA, Shumeet and Covell, Michele, 2006), outlines how they use computer vision techniques to create compact fingerprints that can be matched efficiently. While the paper is fairly detailed I had never come across wavelet transforms before this, and the lack of other fingerprinting methods using it meant that I ruled out using wavelet transforms for my project.

3 Design

To increase the accuracy of audio fingerprinting there are two core ideas. The first idea is to take a bigger sample or extract more “unique” audio perception feature data from each sample but a bigger sample size, or attempting to extract more information from a normal sized sample, might add distortion which over all pulls accuracy down and therefore the opposite of the intended effect. Taking more data from a sample also increases the likelihood for false positive matches as while each larger sample would have more capacity for variation, and therefore uniqueness, it also has more chance of matching another sample. With a larger sample size you’re increasing the length of audio that you’re attempting to recognise, but only generating one little hash per sample and therefore any variation is averaged out and lessened to the extent that quick variations can be lost where previously they would have been hashed and matched.

The second idea to increase fingerprinting accuracy would be to take into account more than just pitch and time based audio perception when computing the hash and searching for matches. Instead you can add more unique factors in to the search which, hopefully, cut down on search times while also reducing the chances of false positives. While two pieces of audio might have similar fingerprints due to noise interference, they might still be distinguishable by analysis to find the key the music was written in or the tempo. At the very least, by restricting fingerprint searches to within a set distance from another factor, such as tempo, you cut down the list of searchable items and reduce search times.

3.1 What I’m trying to do

I am going to attempt to implement a fairly rudimentary audio fingerprinting system, run it through a series of tests, described in further detail in the testing section of this report, to get a baseline set of results to compare to. Then by adding in other factors to my fingerprint definition and search functions, tempo, loudness or key for example, I can run the tests again to see if there is any statistically significant improvement in search time, accuracy or reduction of false positives.

I’m not going to attempt to develop, or even implement, just a robust fingerprinting solution that could compete with the commercial solutions currently available as it would take a lot longer to research, implement and test a system comparable to solutions such as Shazam or Gracenote.

3.2 Criteria for success

To test my software and determine whether I have been successful in attaining my goal of reducing false-positive matches and increasing search speed I will store fingerprints of music of various genres using lossless compression so as to get a perfect master copy of the music with which I can compare. Degrading the music files with compression (such as if the file was converted to MP3) where frequency range is cut, adding noise at various volumes to the sample to make finding musical features less reliable with code and therefore good to check for matching purposes. With the degraded file I can then fingerprint them using both methods and compare the search times and results. Hopefully showing a statistically relevant improvement for the augmented fingerprint in terms of reduced search time and reduced miss-matches and false positives.

3.3 Environment

To develop the software for this project I decided to use Java as a programming language, detailed reasons outlined in the detailed design section later in this report. I also developed on my standard PC primarily; this does however have a Solid State Drive which helped with database insert/search times, and a 4 core processor which helps to improve the speeds of multithreaded aspects of the system. While testing the system on Apples Mac OS X operating system I did run into problems with larger audio samples as Mac OS X limits java instances to only a small amount of the available memory. This error could be averted by increasing the memory limit with the java parameter –Xmx1024m.

4 Experimentation

Once I had the overall goal for my project I came up with some feasibility tests and benchmarks to check that the implementation was possible in the given timeframe, was possible to be developed by me and that my development methods were suitable.

Having never developed software for raw audio this complex and low level before personally, I decided the first few benchmarks should be done to find out how Java, my chosen language, handles audio data and how it could be manipulated and processed.

4.1 Reading Audio

Therefore first benchmark I did was to read in audio data from the microphone. While it could be argued that reading audio from a file would have been a better test, I was aiming to see how Java handled audio, not how file formats are handled, so I went with the simplest method to start with. It didn't take me long to program a simple java class to set up and read from the microphone requiring only minimal searching and reading of the appropriate java documentation. To test that this class worked correctly I made it record 10 seconds of audio from the default audio line in device, in raw format 44,100 samples per second. With the resulting 441,000 samples I wrote them to standard out, copied them manually and pasted the raw audio data into Microsoft Excel, using the line graph functionality to generate the current waveform diagram.

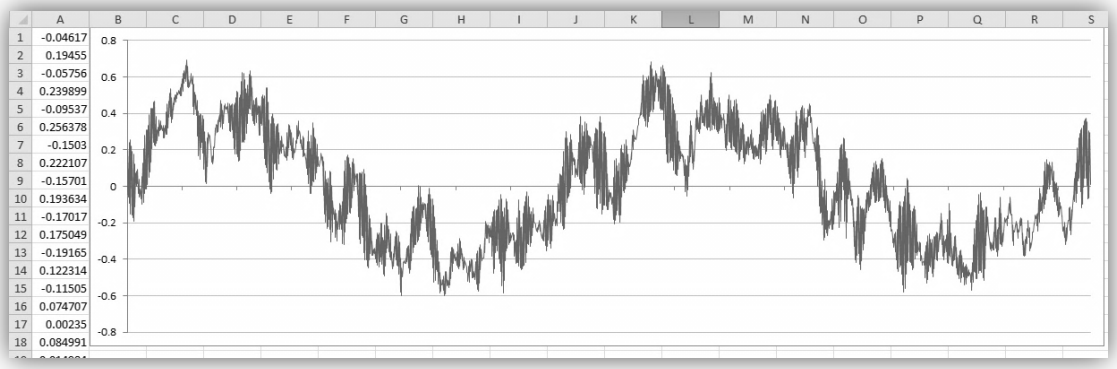


Figure 2 – Line graph of output drawn using Microsoft Excel

From the graph that Excel, shown in Figure 2, I determined that my code to read in the raw audio data was working correctly.

4.2 Spectrograms

Once I saw that I was correctly reading audio data form the microphone the next experiment I did was to figure out the basis of all signal processing: FFT and spectrograms. When reading data in from the microphone the data isn't all that useful in its current format, in the time domain, storing just the amplitude of a signal against time. Instead, for pretty much any signal processing based system, you need to find out what frequencies are present in the signal that you're processing. This is vital to audio fingerprinting and so was a very important part of feasibility testing.

While I could have written my own objects and methods to handle converting time domain data into frequency domain data using the most common method, Discreet Fourier Transform, I decided it would be far simpler, more reliable and a more effective use of my time to instead use an existing class. The best solution I found written for Java was from Princeton Universities Intro to Computer Science resources on data analysis (SEDGEWICK, Robert and Wayne, Kevin, 2011).

Once data has been passed through the FFT function it can be drawn in a similar way to drawing a waveform. Only instead of showing the amplitude of a wave over time, it showed the magnitudes of various frequencies for chunks of the time domain data.

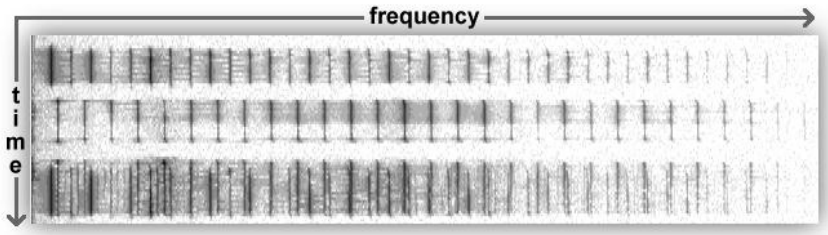


Figure 3 – Spectrogram of harmonica tones

The early example seen in Figure 3 above (flipped 90 ° clockwise for space considerations) shows, in a linear non-logarithmic, scale the spectrogram being drawn as I played one note on a harmonica then a higher pitched note and finally both notes together.

This spectrogram was the first indication I had that my code was working as expected, and it looked similar to the spectrograms presented in (WANG, Avery Li-Chun, 2003) which I included in the literature review as Figure 1.

To check the spectrograms further I first tried some real music which has distinctive tones, Rusko – Woo Boost! From the album O.M.G!, to see if actual audio showed up as a recognisable image.

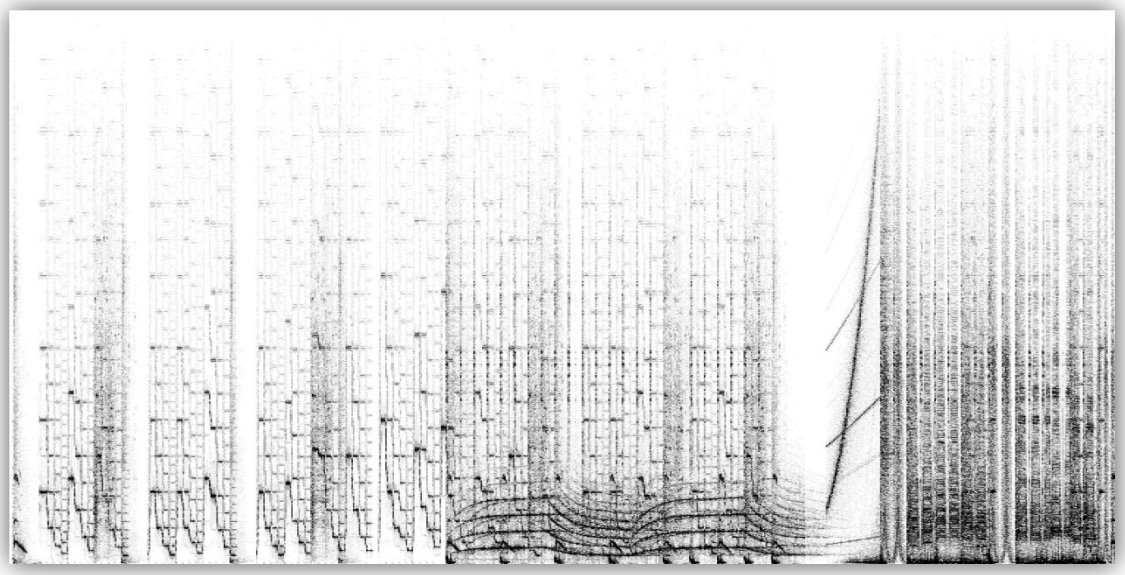


Figure 4 – Rusko – Woo Boost! Spectrogram

As you can see in Figure 4 above, this track has several, sometimes overlapping, frequencies that change pitch and peak in intensity at times. This showed me that the fingerprinting solution whereby I calculate key points from the spectrogram would indeed be possible as it's plainly visible how the music is made up of the frequencies and how they alter over time.

To test the accuracy of the spectrogram code I developed I decided to run some music through where the artist has purposefully included frequencies that when drawn in a spectrogram would show up as images. This is an aspect of stenography and not really within the scope of this project, but it would reassure me that the correct magnitudes for frequencies were being rendered.

Aphex Twin, an electronic musician and composer, released an album titled WindowLicker which includes on the second track a spectrogram image of his own face. And a track by Venetian Snares called Look, from the album titled Songs about my cats, which includes images of the musician's cats.

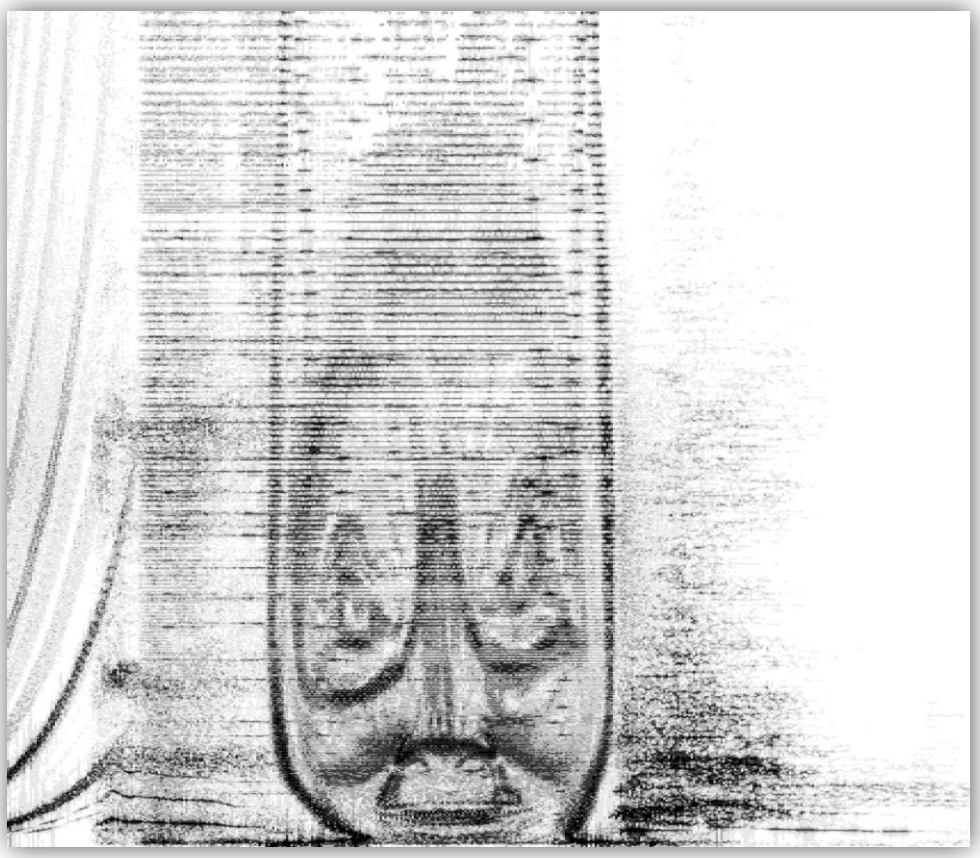


Figure 5 – Aphex Twin Spectrogram

While you can make out the face of Aphex twin in Figure 5 above, it appears distorted. The distortion is due to my spectrogram rendering algorithm using a linear scale, and the software Aphex Twin used to encode the image of his face into audio using a logarithmic frequency scale.

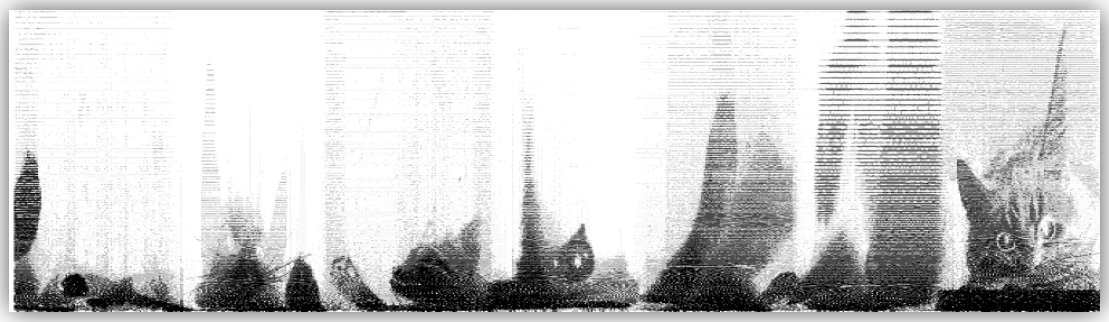


Figure 6 - Venetian Snares Spectrogram

There is some slight distortion on Figure 6, above, for the track by Venetian Snares but it's much less noticeable. The cats are easily visible in the first two parts, and the last part shows

up very well. The middle two parts are less obvious to make out because of my linear frequency scale.

While it seems the FFT operation and spectrogram rendering code all work as I expected, there are some inherent flaws. Due to harmonics in audio you can get reflections being shown on the upper portions of the spectrogram. However the reflections of the sound are only half as bright magnitude wise, so they shouldn't throw off my feature finding algorithms. Also, due to mp3 compression, most music doesn't extend into the upper half of the human hearing spectrum, and so can be ignored by setting an upper frequency limit later on when finding feature points.

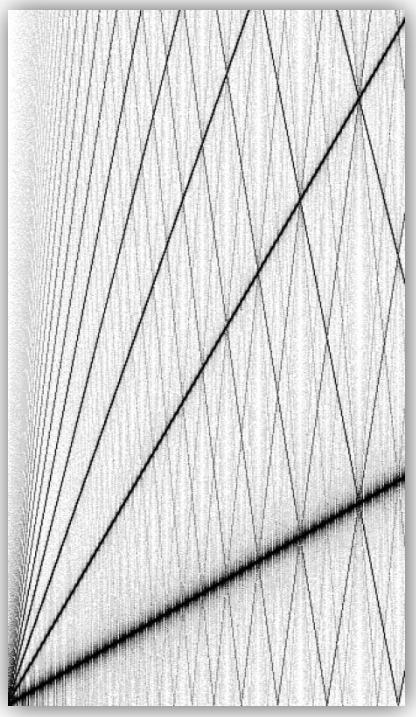


Figure 7 - Square Wave Sweep Spectrogram

As FFT is designed around sine waves, and square waves can be defined using sine waves with the Fourier series, when rendering a spectrogram of a square wave that sweeps from 50-25000Hz the harmonics of sine waves that can represent the square wave are very evident, as seen in the spectrogram image Figure 7.

4.3 Reading in wave files

Reading audio data from the microphone was a relatively simple task, but it would not suit the use of the final software that I hoped to develop. Instead I wished to generate various versions of a file with varying levels of degradation to use as my test data. While I could have played these files through speakers and into the microphone it would add an extra, uncontrollable, level of distortion to the test data. Instead I deemed it a good idea to write code that could instead read the audio data direct from audio files.

I originally intended to support various audio file formats, as then I could later easily compare fingerprints generated from an audio file saved in "lossy" formats, like MP3 where compression destroys some of the original signal, and lossless formats, like FLAC, which store the whole signal. Java makes this task fairly difficult, and MP3 being a proprietary format means that there is little support beyond playing the audio. Instead I struck upon the java sound API which could, supposedly handle files in raw wave format. However, after a few days of un-recognisable waveforms and errors I decided to pursue another avenue. A

brief search for alternative ways to read in wave files resulted in a class developed by Dr. Andrew Greensted as a simple, working, alternative (GREENSTED, Dr. Andrew, 2010). This worked straight away, though is heavily modified and cut down in my implementation as many methods, such as writing, were not used. I also changed it to extend an interface audio reader class so, should I add support for other audio file formats, it will be simple to extend the rest of the software.

4.4 Graphical User Interface

My only previous use of Java was while on placement where I worked on a large servlet system that had no user interface other than through a web connection. I also chose to use the Eclipse IDE, which doesn't come with any inbuilt, standard GUI development tools. While I could have written the code to generate the GUI manually, as a first time Java GUI developer I decided it would be easier to have instant visual feedback and some visual toolkit to help me design and manage the interface. A search on the internet returned Jigloo (CLOUDGARDEN, 2010) as the best plugin for Eclipse for SWING, the API java has to provide graphical user interfaces for java programs.

4.5 Databases

My only previous experience with Java and databases was between Java and Oracle while I was on placement. While Oracle would be a valid option for the database system I could use to store the fingerprints I decided that the overhead required for an Oracle database system was too large. For this system I ended up using SQLite, with detailed design considerations behind this being explain in the detailed design chapter later in this report.

SQLite doesn't have an official binding for Java, but thankfully someone had implemented a cross platform java API which I was able to use (ALM WORKS, 2010).

To test the library, java and SQLite's performance I wrote a test class in java to create a database with one table in, insert rows with words and then select back the rows printing the words to standard out. This worked first time and was relatively simple to accomplish with the library. Based on this test I found the library suited my needs and I could carry on with development.

4.6 Fingerprinting

To check that I would be able to develop an algorithm in java that would generate a fingerprint I decided to test how well I could find the feature points in the audio and see how easy it was to compare these to a database of other points.

To do this experiment I duplicated the spectrogram rendering class and modified it to display the feature points for later comparison. To identify the feature points I went with a simple banded high score system where the frequency domain data is split into groups, or bands, of frequencies. Each band is looped through to find the frequency with the highest magnitude. This high magnitude frequency is deemed notable and becomes my feature point.

While a simpler method than the more robust Shazam system where it checks around the feature point to help eliminate noise throwing off the recognition, the banded high score method worked ok and got roughly similar results.

To test if the feature points were comparable after distortion I saved two images of the spectrograms showing the feature points. With post processing these images could be combined using an image editing tool to remove all but the feature points, and overlay them in such a way as to check accuracy. An example is given in Figure 8 where red pixels are the original file, blue are the degraded file and green is where they overlap exactly. Ideally it should be nothing but green pixels, but you can see how the noise has changed where feature points are found in bands where there was not a major feature before.

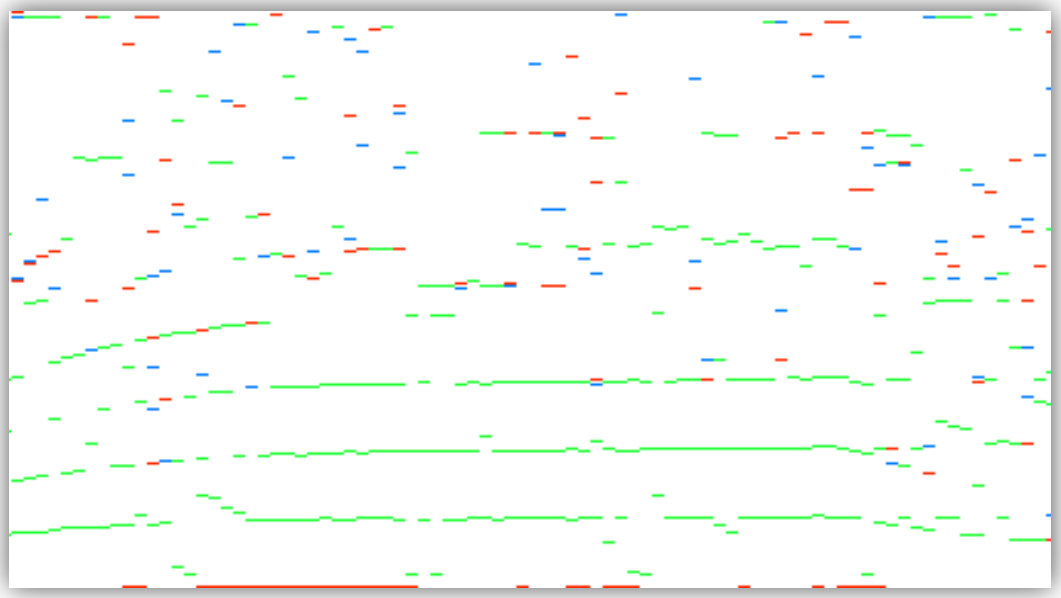


Figure 8 – Comparing feature points

4.7 Other Factors

As well as a method to fingerprint the audio I also needed the optional fingerprinting method whereby I would discern other factors from the audio sample. Figuring out tempo, key and loudness are outside of the scope of this report so instead of developing it myself I

used a well-known, reliable third party API from The Echo Nest (http://the.echonest.com/) which I can access through a java library (http://code.google.com/p/jen-api/).

"The Echo Nest is a music intelligence company that powers smarter music applications for a wide range of customers -- including MTV, The BBC, MOG, Thumbplay, Warner Music Group and a community of about 7,000 independent app developers." (THE ECHO NEST)

I first heard of The Echo Nest when I attended a weekend developer event in London, called the London Music Hack day, http://london.musichackday.org/2010/, where developers get together and for 24 solid hours work on developing small projects that have something to do with music. When I attended in September 2010 The Echo Nest CEO and some developers had flown in especially for the event from their offices in Boston, USA. They gave a talk describing and demonstrating some of the features of their API, which I noted down at the time as possibly being of use for this project.

After registering for a developer account with The Echo Nest, and downloading the java library, I ran a few files through their system to check what extra data could be gathered and to see how accurate it was.

While the results take quite a while to return, it gives me information on the time signature, tempo, key, loudness as shown in Figure 9 below.

```
send-err-> http://developer.echonest.com/api/v4/track/profile?md5=d47cc300523097582c1e83fdc0600e18&api_key=VZK0SNBGSEUD84U8S
recv-err-> {"response":{"status":{"message":"The Identifier specified does not exist: d47cc300523097582c1e83fdc0600e18","code":5,"ve
Waiting
Waiting
Waiting
Waiting
 ====== TRVUQZP12F971DE09C =======
Title   : WONDERBOY
audio   : null
foreign : null
Analysis: https://echonest-analysis.s3.amazonaws.com:443/TR/TRVUQZP12F971DE09C/3/full.json?Signature=IDkUd7FRf3A0nTz%2FhCtdnENYGDA%3
Artist  : Tenacious D
MD5     : null
Duration: 30.07111
Key     : 7
Loudness: -7.66
Mode    : 1
Preview : null
Release : null
Status  : COMPLETE
Tempo   : 100.189
Time Sig: 4
```

Figure 9 - Echo Nest Analysis

I can then use that extra metadata that The Echo Nest has computed to augment my search for matches, hopefully getting me the result I'm after.

5 Detailed Design

5.1 Development Choices

To develop all of the software for this project I decided to use the Java programming language coupled with the Eclipse IDE.

When considering what language to develop in for this project I came up with a list of languages that I already had some knowledge and previous use in and ranked them according to my familiarity with them, suitability for digital signal processing and availability of third party libraries and general support. From this the top 5 languages were:

1. Python
2. Java
3. C++
4. Ruby
5. C

From this list I disregarded Python because while there are many libraries to facilitate audio processing, is cross platform, I've developed MIDI software in python before at the London Music Hackday 2010, it's not fast enough to process large raw audio files to compute fingerprints in a reasonable time. While it can be compiled down to native code this isn't totally reliable and isn't really a solution to the problem. Also, a lot of the libraries dealing with Python and audio data are written in C, so using Python was mostly pointless aside from some of the high level support it offers.

C++ has a lot of support and libraries available, and could be developed in for cross platform. However I am not experienced enough to commit to developing this project in it.

Ruby is one of my favourite languages, and does have support for audio processing. But it falls into the same pitfalls and Python with speed and pointless high level features.

C seemed like it could be the better choice due to low level support which would help make for optimised audio data handling and processing. However, some of the higher level processing to be done, such as image rendering and Graphical User Interfaces, are not so simple to do. While I already know C fairly well, with the time constraints of the project and the scope of development I wanted to undertake, it did not seem like a better choice over Java.

Java has a lot of support and libraries available, runs around the same speed as C, gives you easy access to low level constructs where needed and also handles high level code, like image rendering and Graphical User Interfaces, with ease. While Java isn't my strongest language with barely a year of use and almost entirely self-taught, it came out as the best choice to develop this project in.

Once I had decided to use Java I looked at the IDE's available to support my development. I had only used JDeveloper (http://www.oracle.com/technetwork/developer-tools/jdev/overview/index.html), Eclipse (http://www.eclipse.org/) and NetBeans (http://www.netbeans.org/) before. While I could have tried out a new IDE, or gone low level and just used a text editor and the command line to compile and debug my code, I decided to go with Eclipse as I find it has all the tools I needed for the project while not being slowed down with other features, like servlet containers and oracle database support, which wasn't needed for this project.

Speaking of databases, my project would also require a database of some form so that I could store and search for fingerprints and track details. The options that sprang to mind were MySQL (http://www.mysql.com/), Oracle (http://www.oracle.com/us/products/database/index.html), PostgreSQL (http://www.postgresql.org/) or SQLite (http://www.sqlite.org/). MySQL and PostgreSQL both have similar features and are both suitable. Oracle might be free for this use, but requires a fairly complex setup. While Oracle has the most features, it has nothing that this project requires that another database system can fulfil and it requires the overhead of running the oracle server. SQLite has enough features for use in this system. Between MySQL, PostgreSQL and SQLite, SQLite stood out as the best choice due it its small size, efficiency, available libraries for java and documentation. As this system doesn't require a server/client setup, a capacity for high volumes of connections and queries, a large dataset or high concurrency then SQLite is fine. MySQL and PostgreSQL would also suit the job, but require a lot more processing power and are made for purposes more complex than my uses.

5.2 UML

A common method of modelling software during the design stage is to design UML diagrams. For my system there is a relatively simple use case diagram as the software is fairly compartmentalised and doesn't link to anything external besides the user and The

Echo Nest. Figure 10, is a use case diagram showing an outline overview of the systems functionality.

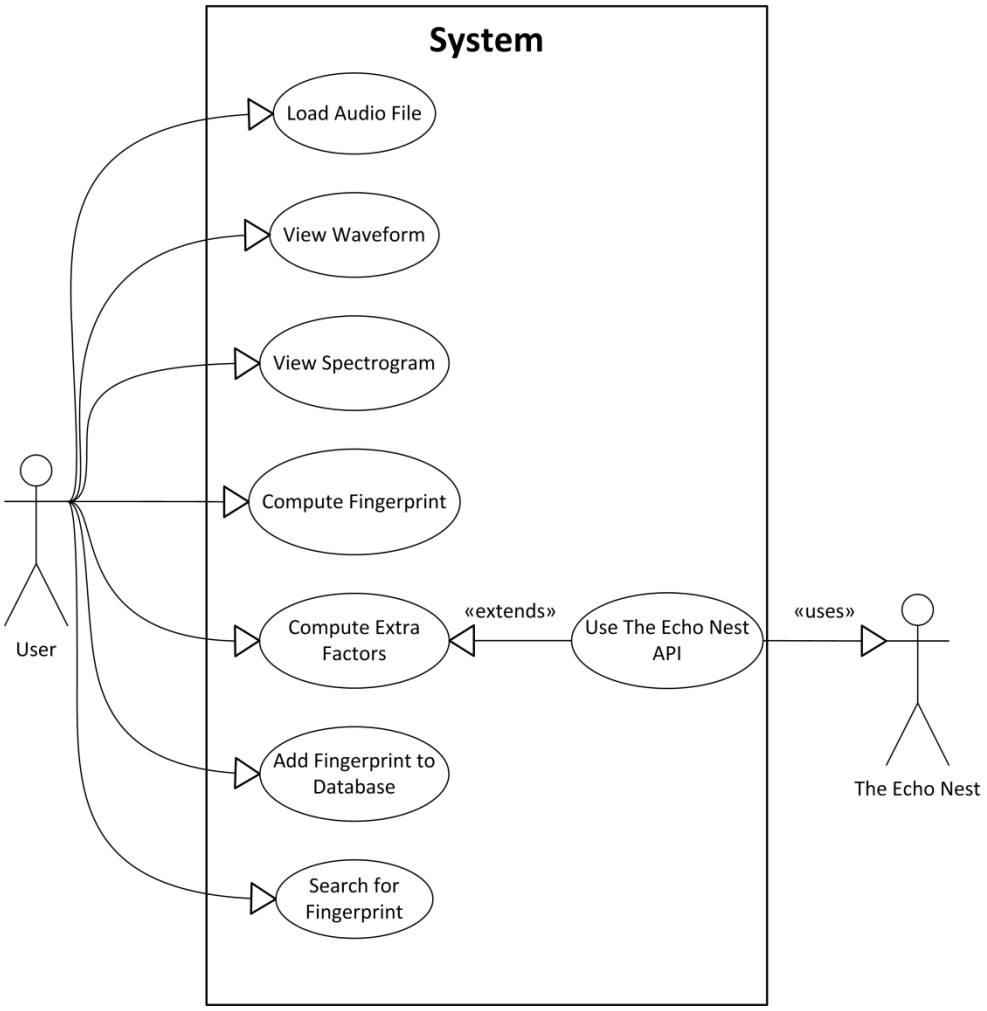


Figure 10 – UML – Use Case

UML also has a type of diagram to help developers design how processes interact with each other, a sequence diagram. I designed three of these to help me refine the sequence of operations that occur when a user fingerprints a file, searches for it in the database or stores it in the database.

Figure 11 on the next page is the sequence diagram outlining the fingerprinting process for both normal and augmented fingerprints.

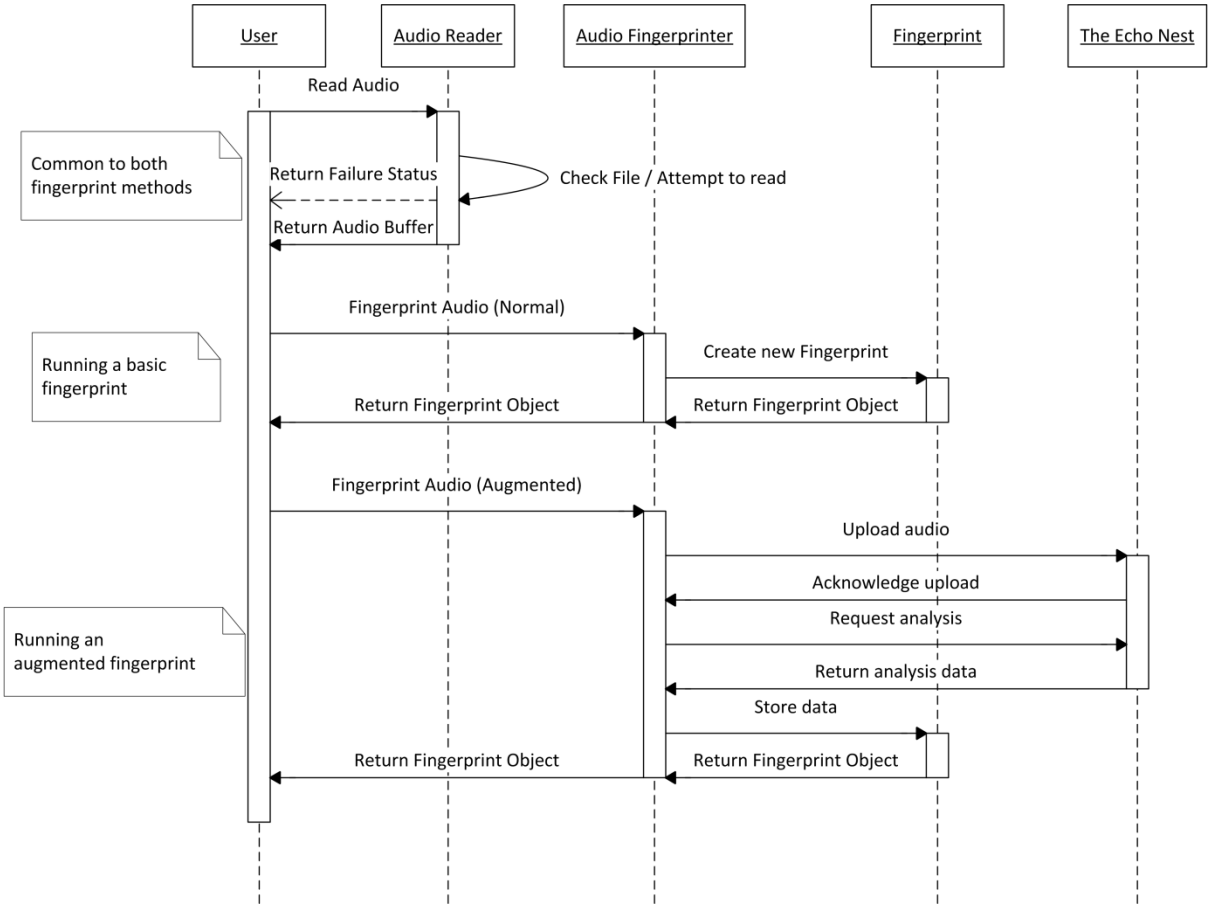


Figure 11 - UML - Fingerprinting

The next diagram, Figure 12, outlines the sequence of operations to store a fingerprint. This is common for both basic and augmented fingerprints as the extra data for the augmented fingerprint is stored along with the regular track information.

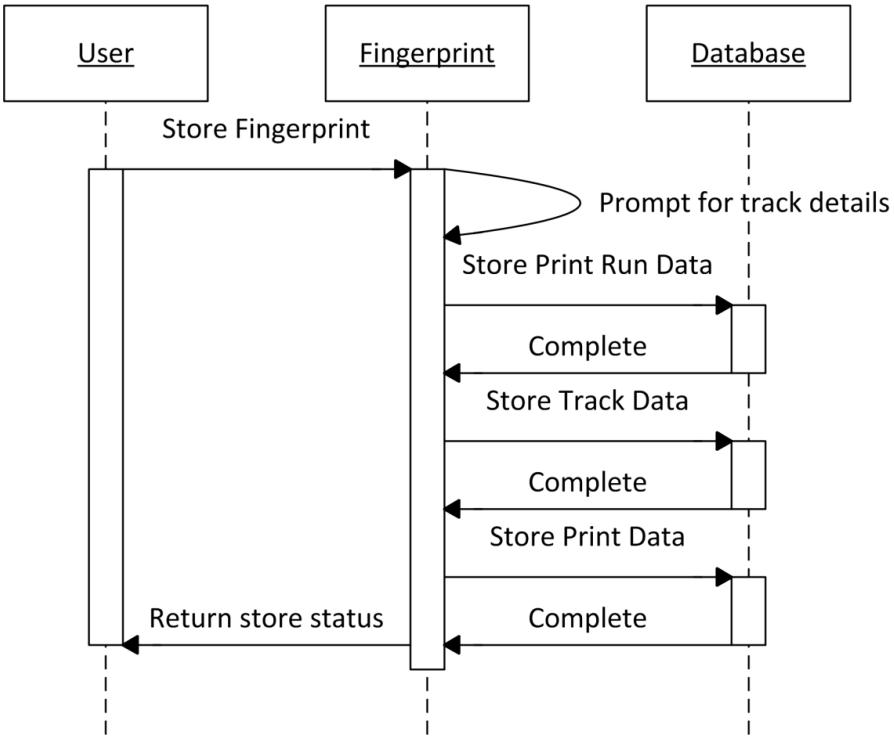


Figure 12 - UML - Store Fingerprint

The next sequence diagram, Figure 13, shows an overview of searching for matches, assuming the fingerprint object was already created by following through the fingerprint process.

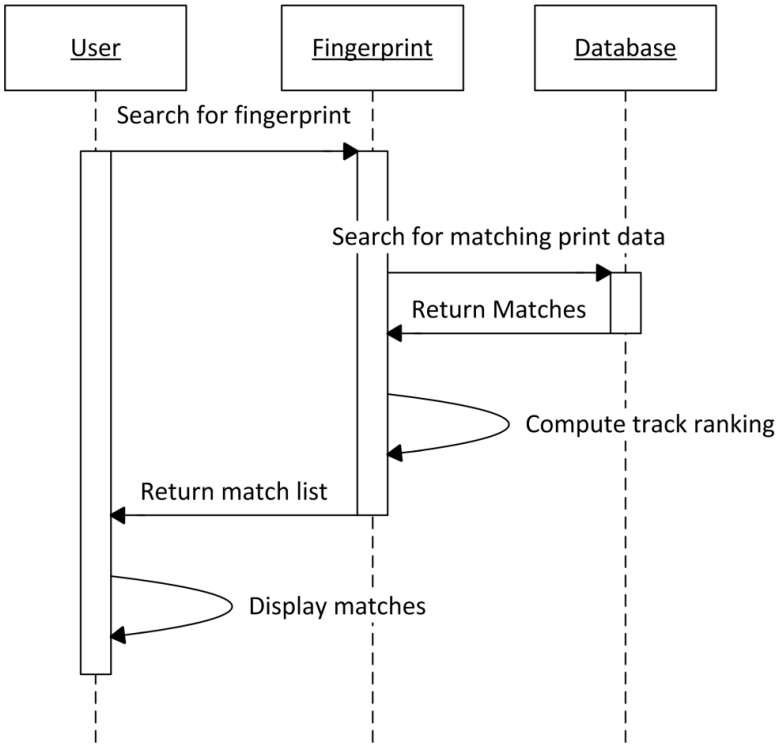


Figure 13 - Searching Fingerprints

I also made a class diagram Figure 14, outlining the classes, methods and members of each class and how they relate to each other. The relationships are all calling about the WindowMain class, which creates instances of many of the other classes, but the main focus is around the Fingerprint interface which the fingerprint classes implement, the fingerprinter classes generate through and the main window stores.

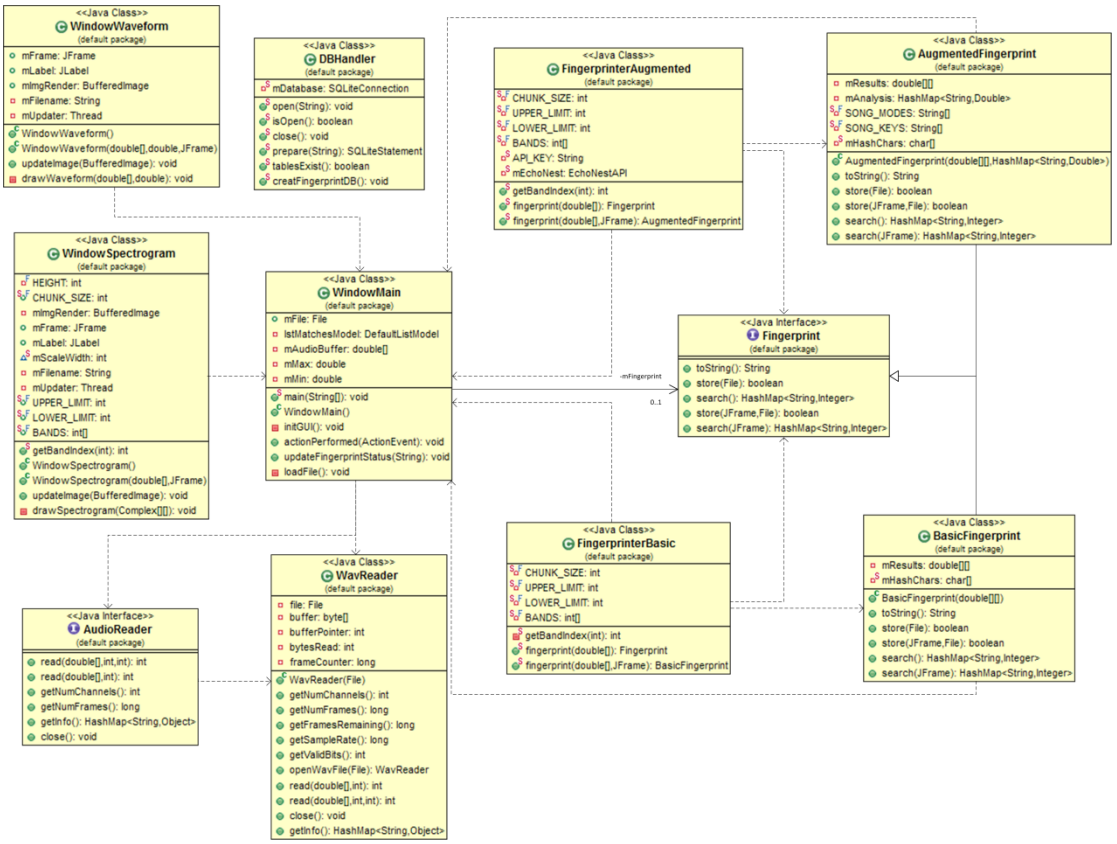


Figure 14 - Class Diagram

5.3 HCI Considerations

Part of the software I plan to develop is an interface through which people can see the fingerprint and results simply and also have options for viewing waveforms and spectrograms. The interface wouldn't have to display much data, but I still had to think about the form and control design from a HCI standpoint.

With HCI there are 8 "Golden Rules" of interface design: (SHNEIDERMAN, Ben, 1998)

1. Consistency
2. Shortcuts
3. Informative feedback
4. Design dialogs to give closure
5. Error prevention and simple error handling

6. Permit easy reversal of actions
7. Support internal locus of control
8. Reduce short term memory load

For my simple interface consistency was simple to achieve. Making sure that whenever a user loaded a new file it would wipe any previous actions performed on previously loaded files, static menus and labels and a static theme meant that the user interface was always kept consistent with what the user would expect.

Due to the few controls on offer, shortcuts were simple to implement. By adding keyboard shortcuts, like Ctrl+O to load a file, it makes it easier and faster for frequent users to accomplish the task at hand.

Informative feedback will be displayed wherever needed. When doing some tasks, like rendering a spectrogram or adding fingerprint data to the database, it can take a longer time than the user may expect. To display the current status and inform the user of the underlying process through the window title bar or a label on a form, the user will be able to have feedback that their action is being worked on.

Dialogs should yield closure by either alerting the user via alert box when an action is complete or by updating the interface as it works. In the case of the visual audio representation windows, waveform and spectrogram drawing, I can update the display as it renders.

Error prevention and handling will be done throughout. Making use of javas try/catch structure I can catch any errors that do arise, attempt to salvage the current function and alert the user to the problem.

As the only thing to be changed by the user is the database when a fingerprint is stored in it, there are no features that might require reversal. While it could be argued that I should make it easy for a user to remove the fingerprint data from the database, I didn't feel it was necessary for this project.

Locus of control remains solely in the users hands for this system. The only time control is taken away is when the interface is busy using all processing power to compute operations on the audio data. Otherwise the user has full control.

All information that might be required by a user for this system will be displayed on the main window. That way items like the file location that is currently being fingerprinted or the fingerprint for that file are displayed and not required to be remembered.

Based on these decisions the following is a prototype user interface design for the main window as seen in Figures 15 and 16. I decided the waveform and spectrogram windows didn't need any design from a HCI standpoint as there was no controls, just display.

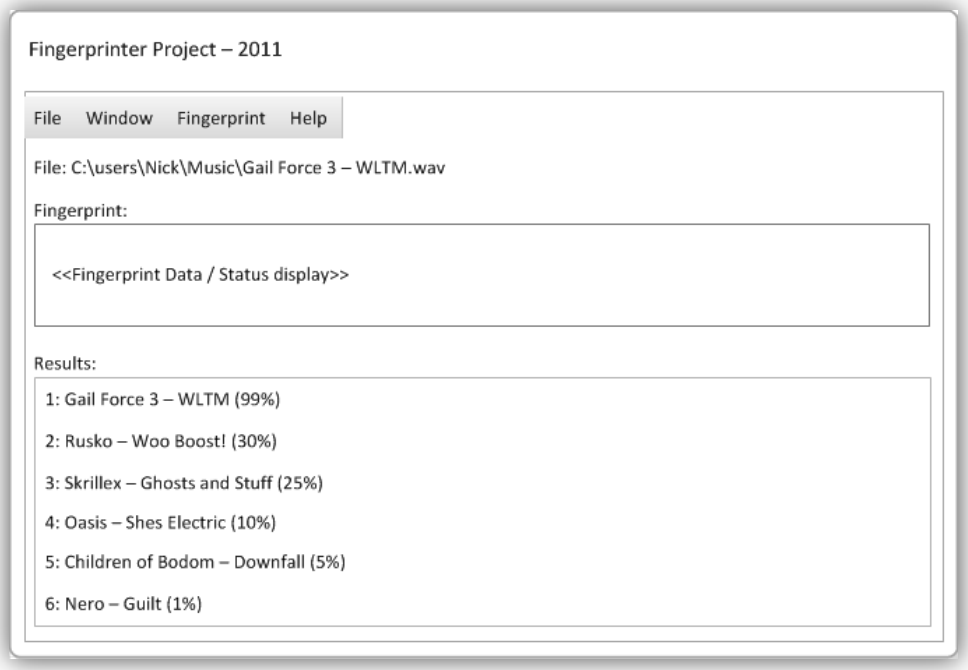


Figure 15 - HCI - Main Window

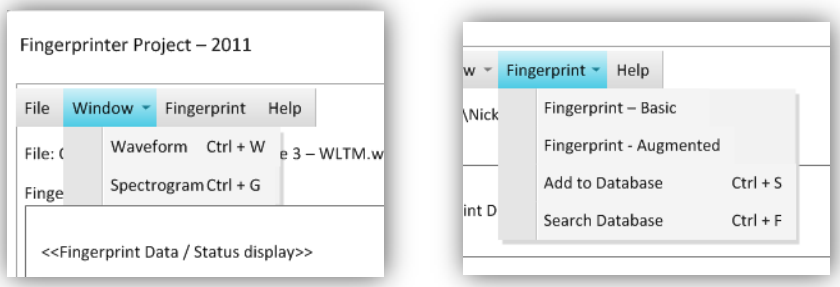


Figure 16 - HCI - Menus

5.4 Database Design

The database design for this project was relatively simple. All I need to store was a set of data for the fingerprint itself and to associate that data to a track where I could get details

about the original piece of music. I also decided to add in a table to give information about the fingerprint, such as the file path it was originally made from and the time it was created.

For the fingerprinting method I decided to implement, the fingerprint would be stored as the maximum frequency in a range of frequency bands. So the print_data table stores the values for each band, the chunk number it was taken from so I can search for successive parts of the fingerprint, the track id so I can relate that table to the tracks table and the print_run_id so I can relate it to the fingerprint run.

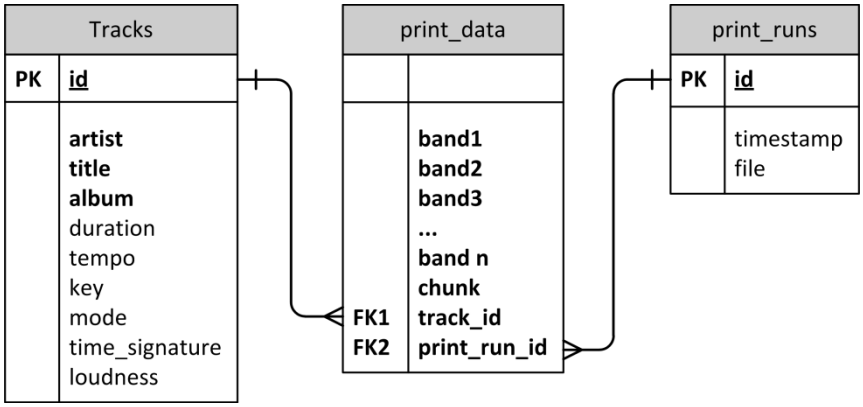


Figure 17 - Entity Relationship Diagram

Figure 17 is an Entity Relationship Diagram I designed. The SQL source to create the database based on this diagram is listed in Appendix A at the end of this report.

5.5 Third Party Libraries

To analyse the audio samples and computer the other factors needed for my augmented fingerprint I had two options. One was to develop the analysis code myself, which would have alone been enough to warrant one or several final year project reports. This left me with the other option of using a third party library.

There aren't many libraries publicly available for this kind of audio analysis and the choices available that suited this project were The Echo Nest (THE ECHO NEST), or VAMP (QUEEN MARY, UNIVERSITY OF LONDON, 2007). While VAMP is a little more technical and has specific audio analysis plugins, it's only got wrappers for C++ and Python, which ruled it out for my use in java unless I created a java library to wrap it. The Echo Nest, while having less of a focus on audio analysis, does offer some analysis features through the use of their server. There was also a java library that let me use their API. Therefore, The Echo Nest was the third party audio analysis service I went with.

5.6 Development Process

To develop the software for this project I looked at various development processes to find out if any of them would be suitable. Almost all of the standard development methodologies, such as waterfall or v-process, are designed with large teams of developers, user-based requirements and deployment procedures in mind. My project had no user requirements to capture however, aside from my own objectives. I was also the sole developer, so I didn't need to plan ahead to make sure people weren't waiting on my code to be finished or checked in to a source code repository. Therefore most of the generic model of the Software Development Life Cycle; Requirements Capture, Design, Build, Test and Implementation, did not apply to me.

Instead of going for the simple, but often error prone, route of the Build & Fix model I instead opted to develop using the Extreme Programming method. Extreme Programming was, as the creator Kent Beck said in his book "Extreme Programming Explained: Embrace Change", "*developed to address the specific needs of software development conducted by small teams in the face of vague and changing requirements*" (BECK, Kent, 1995). This suited my needs as while the design is done and the requirements known, as I progress through the project I anticipate many changes to the requirements and will need a development method that lets me keep agile.

In an article written by Microsoft they highlight that "*In Extreme Programming, rather than designing whole of the system at the start of the project, the preliminary design work is reduced to solving the simple tasks that have already been identified*" (MICROSOFT CORPORATION, 2005). They also note that Extreme Programming should be used when the developers are doing small, incremental releases with a very test driven development process, are working with new technologies and have a small team size. All of which suit me and this project.

6 Implementation

To develop the software I split up the work into several key tasks, basing my decisions on the results of my experimentation earlier and following the development principals laid out by the Extreme Programming development model which calls for small, incremental releases that start by building the core of the system and expanding it to meet requirements as it is developed.

6.1 Audio Readers

The first task was to develop a robust system for reading the audio files in. While I only wanted to implement the reading of wav files for this project, I knew that I might have time to expand the system later to support other formats. With expansion and future work in mind I decided to create an interface called AudioReader. This interface defines the common methods that would be required to read any audio format, such as reading into a buffer, getting information about the file and closing it. As developing a class to read in wav file data wasn't in the scope of the project, and the code java provided didn't work reliably, I used a third party class (GREENSTED, Dr. Andrew, 2010) which was modified to implement the interface I defined. I cut out a lot of the overloaded methods in the third part class which I didn't require, and modified those that were left to fit with my interfaces requirements. I also took the exception class that the wav reader class used and made it generic so that any future audio reader classes could throw exceptions in the same way and be caught by the code that used it.

Once the audio reading was implemented and tested by making it return the amplitude data to standard out, the next task was to re-implement the waveform and spectrogram display. While these displays are not required for the operation of the system they provide very useful information about the workings of the system and are a great view when debugging the fingerprinting methods later.

6.2 Waveform Window

The waveform window is called from the main window, being passed, by reference, the buffer holding the audio data and the maximum amplitude of all the samples. While I should, arguably, be passing just the file reference and doing a buffered read inside the waveform drawing routine to cut down on long term memory use, it was simpler to do it this way for the small audio samples I am dealing with.

The waveform window then constructs a new window and calls the draw routine. The draw routine sets up the Graphics2D object which is used to draw to a buffered image, which is put inside a JLabel in the JFrame window. The draw routine then, after finding out how many samples need to be taken in per vertical line of window space, starts drawing. For every vertical 1 pixel line in the image a window of amplitude samples are taken and lines drawn between them. While the loops are running to draw the data onto the buffered image object, a new thread was instantiated which calls the update method every 50ms. The update method puts the buffered image in the JLabel, clears the JFrame and then puts the JLabel in that and then forces a redraw of the JFrame. This makes for a real-time display as the waveform is drawn. Figure 18 is an example of the waveform window displaying the waveform for the first 30 seconds of *Rusko – Woo Boost!*.

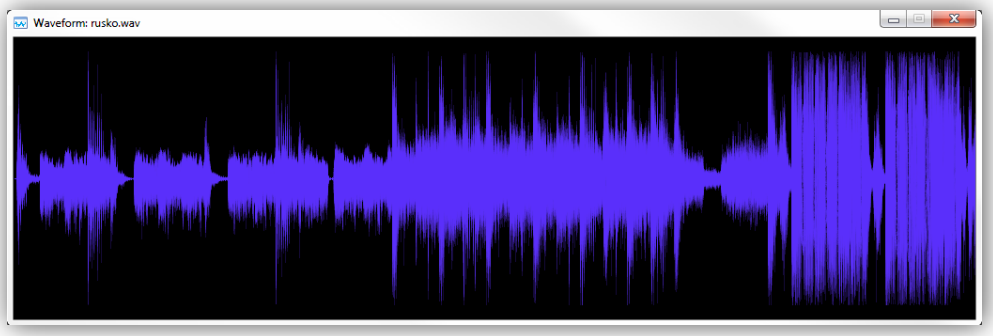


Figure 18 - Waveform Window

6.3 Spectrogram Window

The spectrogram window operates in much the same way, only before calling the draw function it converts the time domain amplitude data into frequency domain frequency data. To do this it fills a buffer of complex number objects, the complex class coming from (SEDGEWICK, Robert and Wayne, Kevin, 2011), which can then be run through the FFT function, the FFT class also coming from (SEDGEWICK, Robert and Wayne, Kevin, 2011). With the chunks of data now in the frequency domain, the 2dimensional array of frequency data is passed to the draw function. The draw method sets up the Graphics2D object again, but this time loops through each chunk of processed data. For each chunk it runs through each frequency to find out the feature points which the fingerprinting classes will later be based around. Then it draws a 5x1 pixel rectangle for each frequency group in the chunk onto the buffered image, using either full red as the colour for a feature point, a dark green for a frequency band boundary or a blue saturation relative to the magnitude of the frequency. Figure 19 is the spectrogram window for the first 30 seconds of *Rusko – Woo Boost!*.

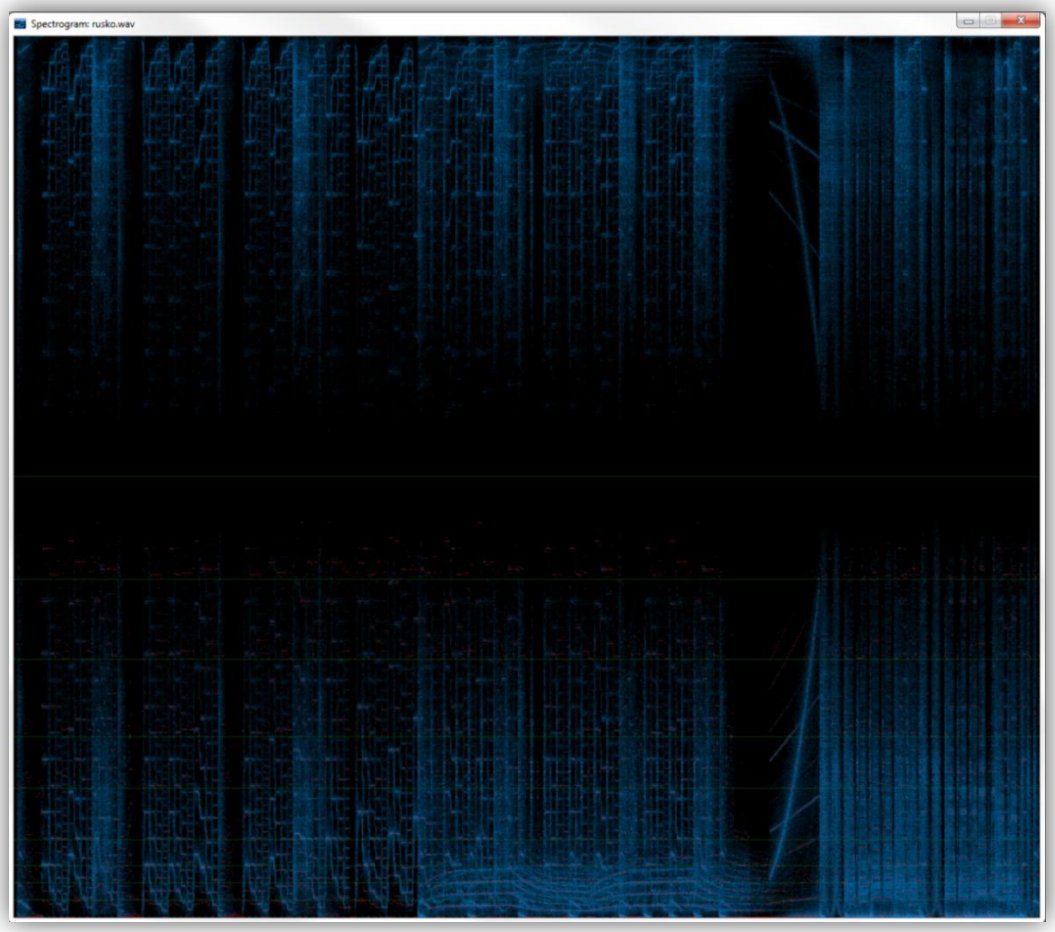


Figure 19 - Spectrogram Window

A zoomed in section from the above spectrogram, showing the frequencies from the audio in detail as well as highlighting the feature points in red is shown below in Figure 20.

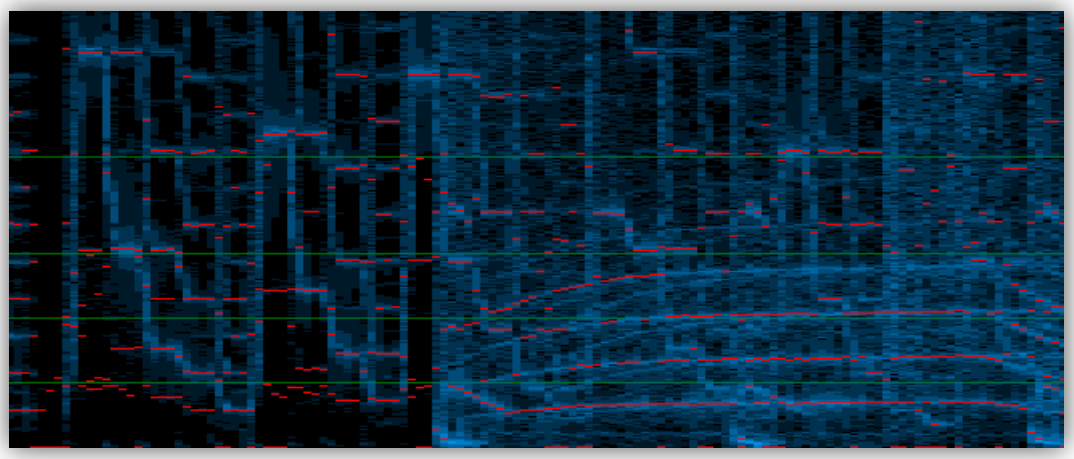


Figure 20 - Zoomed In Spectrogram

6.4 Main window

Now that I had some visual feedback of the audio data I created the main window. Based on the diagrams I designed after considering the impact of HCI on design laid out in the detailed design chapter earlier in this report I developed a simple window with a menu layout for controls, shortcut keys and with plenty of area for feedback to be displayed. I designed the interface using a GUI builder plugin for the Eclipse IDE (CLOUDGARDEN, 2010). I also used and modified icons for the interface (JAMES, Mark, 2005) (TANGO DESKTOP PROJECT, 2009). A screenshot of the main window is shown in Figure 21 below, along with examples of the menu content.

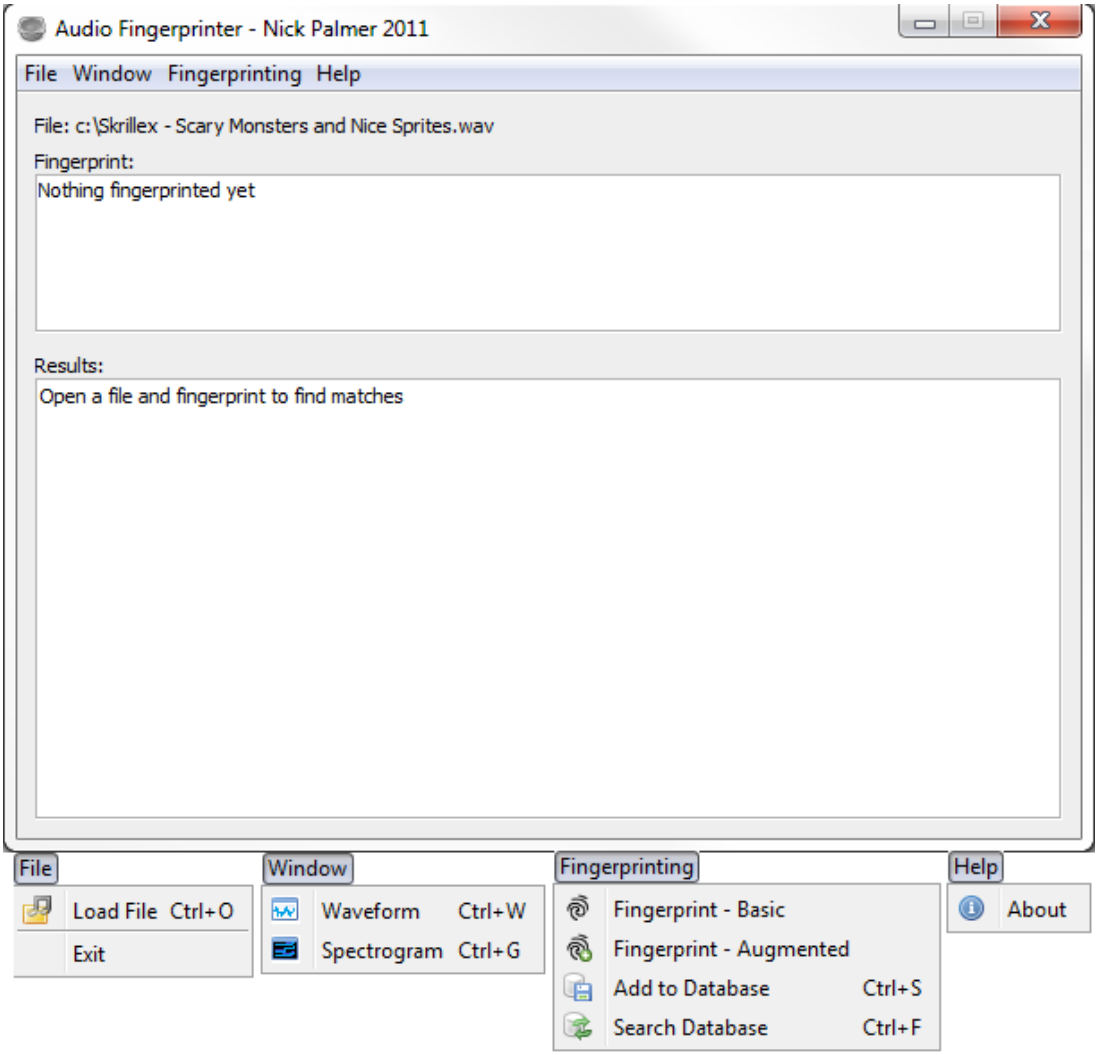


Figure 21 - Main window and menus

6.5 Basic Fingerprinting

Now I had all the interface work developed the next task to do was the initial fingerprinting algorithm. To do this I wasn't striving for the perfect fingerprinting algorithm, I was just after a fairly robust and simple method that could identify audio. To do this I referenced my research from the literature review and saw that all methods found feature points and based their fingerprints around those. The simplest method to find feature points was, after splitting the audio buffer into chunks and running them through the FFT code, split the chunks up into bands of frequencies. By having several bands of frequency I could get standard, and hopefully repeatable, features found on samples. For each band I find the frequency with the highest magnitude and take that as my feature point. While this can be susceptible to error when noise occurs on a previously empty band I added in a lower magnitude limit to try and reduce this problem.

With the feature points stored as frequency values per band for each chunk I could have used the Shazam method of using a feature point as an anchor point and calculating vectors between it and other feature points, but instead I opted to just store the frequency values per band for each chunk. This is inefficient database wise as it means storing multiple values per chunk. It is simpler, however, and let me see exactly what was happening as the fingerprint was created. It also makes search algorithms slightly simpler, though it will mean slower searches as there's more data to search. To implement this in code I decided to create a class, FingerprinterBasic.java with a static method to generate the fingerprint as there was no need to consider concurrency as the system can only fingerprint one file at a time. The fingerprint method splits the audio buffer into chunks, computes the FFT and finds the feature points for the bands which can then be added to an array of results. This array is then passed to the constructor of the BasicFingerprint.java class. The BasicFingerprint class implements a common interface for fingerprint methods which the basic and augmented fingerprint classes share. It stores the results for a fingerprint and contains methods to display the fingerprint as a string, search for the fingerprint in the database or store the fingerprint in the database.

6.6 Basic Fingerprint Storing & Searching

The store function in the BasicFingerprint class can then be called on the fingerprint instance which gets stored in the main window after fingerprinting. To store a fingerprint the user is prompted for the track details; provided a user enters the details and doesn't click cancel the function then creates a new row in the print_runs table to store the file location that is

being fingerprinted along with the time. Then it stores a new row in the track table to store the track artist, title and album name. Once the track is set up it can start looping through every chunk of result data and store the values for each band in a column to be searched for later. Provided no SQL errors are thrown and the user provided the track details, a true value is returned by the function to let the main window know the store was successful.

The search function in the BasicFingerprint class can also be called once a fingerprint has been generated. The search method implemented is very crude currently, but works surprisingly well considering. It loops through each chunk of fingerprint data, looking through all rows in the print_data table that have values within certain limits of each band. If any chunk matches one from the database it retrieves the track_id of the matched row and increments a value in a score map. The score map is then later iterated through to get the track details and populate a hash map which it can return to the main window for display.

6.7 Augmented Fingerprint Storing & Searching

Now that I had it fingerprinting files I duplicated the FingerprinterBasic and BasicFingerprint classes, renamed basic to augmented and added the code to augment the fingerprint. To do this I added a block of code to the end of the fingerprint method in FingerprinterAugmented.java which created a new Echo Nest API instance and uploads the track to The Echo Nest servers for analysis. The Echo Nest API also hashes the file with a normal MD5 hash to check if the file needs uploading. If the file has already been uploaded based on the MD5 hash it uses previous analysis results to save upload time. Once the track has been uploaded, or identified as having been uploaded previously, a new TrackAnalysis instance is created which holds all the values I'm using for augmenting the audio fingerprint.

The TrackAnalysis instance is then used to create a HashMap which is then passed into the constructor of AugmentedFingerprint alongside the normal fingerprint results so that AugmentedFingerprint can later store and search using the values The Echo Nest has derived. I stored the values in a HashMap here instead of passing through a reference to the TrackAnalysis object because calls to get analysis values on the TrackAnalysis object are going to the Echo Nest servers each time. By caching the value on the time-unimportant fingerprint generate stage I can save time later when searching for the fingerprint.

AugmentedFingerprint.java operates in much the same way as BasicFingerprint.java did as they both implement the same Fingerprint interface. The augmented fingerprint class,

however, now inserts all the data it has about the track when it stores the row in the tracks table.

When searching for a fingerprint it also checks that the augmented values in the tracks table associated to the current print_data row are within range of the current fingerprints values if The Echo Nest returns a high enough confidence value in its analysis. If they are it operates a score system in the same way as the basic fingerprint search method and results in a hashmap of track information to score being returned for the main window to display.

6.8 Database Handling

There is also a DBHandler.java file included in the project. This handles the opening, creation and reference to the SQLite database. The class is all static and the open function is called when the main window is instantiated. This served as a common place for database access which the fingerprint classes call into when preparing statements while storing/searching.

7 Testing

To test that the software I develop meets my goals I will conduct two sets of tests. One to confirm my basic audio fingerprinter can reliably identify audio based upon musical perception features and a second set of testing to see if there is any effect, hopefully positive, on using an augmented fingerprint to search with.

To test the software can reliably identify audio based upon musical perception features I have created a series of files. I've taken audio of various types: Classical music, Dupstep, Metal, Pop, Chiptune and Jazz. Each piece of audio has been obtained in the highest quality I could find, often FLAC (Free Lossless Audio Codec), and converted it to wav format. As the wav file format is lossless I don't lose any data due to compression and the test remains valid. After fingerprinting each high quality version of the files and storing them in my database I can then set about finding the limits of recognition. I will degrade the file by adding various volumes of static noise or filtering out certain frequencies to simulate various versions of the same file that many people may have. By comparing the fingerprint of the degraded the file to that of the master copies I can check to see how robust and reliable my fingerprinting software is. As well as recording the amount of matching chunks I will also record any false positive matches or files that come close to being a false positive.

To test the effect of adding in other music factors, such as key, loudness or tempo into my fingerprint I will compare the same files and check if there is any change in time spent searching for matches and the number of false positive results / non matches.

7.1 Results

First, to set the baseline of the project and have something to compare my test results to I ran all my test files through the basic fingerprinter and searched for matches after only inserting the original, high quality fingerprints into the database. The results are shown in Figure 22.

	Basic Fingerprint												
	Chiptune		Classical		Dubstep		Jazz		Metal		Rock		
	Time	Top Match	Time	Top Match	Time	Top Match	Time	Top Match	Time	Top Match	Time	Top Match	
Clean	2794	7261	2616	32934	2432	2234	2491	12235	2925	3949	2776	4637	
Static 0.2	2461	2152	2642	MISSING	2248	-179	2503	MISSING	2425	2135	2456	MISSING	
Static 0.4	3064	1407	2499	MISSING	2267	-109	2626	MISSING	2466	1618	2504	MISSING	
Static 0.6	2848	869	2415	MISSING	2398	-292	2444	MISSING	2476	1222	2327	MISSING	
Static 0.8	2583	759	2396	MISSING	2397	-250	2327	MISSING	2433	955	2316	MISSING	
HPF 200Hz	2498	7308	3139	31305	2256	2124	2557	10977	2362	3357	2416	4259	
HPF 400Hz	2419	6760	2715	28947	2224	2071	2413	8666	2362	2718	2435	3933	
HPF 600Hz	2476	6114	2578	26146	2267	2007	2381	6791	2417	2395	2466	3566	
HPF 800Hz	2245	5715	2616	23865	2283	1916	2576	5839	2399	2201	2344	3215	
HPF 1000Hz	2513	5327	3295	22391	2192	1790	2349	5316	2395	2076	2463	3163	Overall Avg.
AVG	2590		2691		2296		2467		2466		2450		2493

Figure 22 - Basic Fingerprint Results

The times given are milliseconds taken from the user clicking “Search Database” until it returned a result. For testing purposes each search was done 3 times per search and the time result was averaged. The Top Match column shows how many chunks matched when doing a search. This number can be bigger the original file match if certain chunks end up matching more than one chunk of the track in the database.

Results marked as missing are results where the fingerprinter did return matches but none of them were the right match. Results in the negative are from where matches were found, but the top match was incorrect, the difference between the intended result and the top result is the negative value shown.

From the above table you can identify that times were roughly consistent across all searches and only the quiet sample audio, such as the jazz, classical and rock samples, were completely unable to match after noise has been added. The dubstep & noise tests came out negative as each time the fingerprinting system incorrectly identified the sample as the chiptune sample, only by a small margin however. This isn’t too big of a surprise as both tracks used in this test do have similar frequencies used at the start but the chiptune sample uses an 8 bit music effect which sounds like a rough, distorted, low bass which shows on the spectrogram as bands of white noise following the soprano notes.

To see if augmenting the fingerprint with the values of key; e.g. C, G, F-Sharp; tempo and time signature improved the accuracy or improved search speed I re-ran the tests with the same samples and the same original sample fingerprints in the database. The results came out as shown in Figure 23.

	Augmented Fingerprint												
	Chiptune		Classical		Dubstep		Jazz		Metal		Rock		
	Time	Top Match	Time	Top Match	Time	Top Match	Time	Top Match	Time	Top Match	Time	Top Match	
Clean	2568	7261	1709	32934	2496	2234	2239	12235	2241	3949	2257	4637	
Static 0.2	2636	2152	2568	MISSING	MISSING	MISSING	MISSING	MISSING	2363	2135	2434	MISSING	
Static 0.4	2208	1407	MISSING	MISSING	2087	-109	2166	MISSING	2332	1618	2291	MISSING	
Static 0.6	2304	869	2213	MISSING	2174	-292	2056	MISSING	2305	1222	2318	MISSING	
Static 0.8	2287	-115	2174	MISSING	2225	MISSING	2157	MISSING	2354	955	2257	MISSING	
HPF 200Hz	2156	7308	MISSING	MISSING	2061	2124	2188	10977	2140	3357	2564	4259	
HPF 400Hz	2090	6760	MISSING	MISSING	2050	2071	2173	8666	2157	2718	2386	MISSING	
HPF 600Hz	2151	6114	2221	26146	2028	2007	2111	6791	2941	2395	2310	MISSING	
HPF 800Hz	2179	5715	2196	23865	2029	1916	2050	5839	3031	2201	2430	MISSING	
HPF 1000Hz	2097	5327	2211	22391	2083	1790	2159	5316	2071	2076	2215	MISSING	Overall Avg.
AVG	2268		2185		2137		2144		2394		2346		2246

Figure 23 - Augmented Fingerprint Results

As you can see at first glance over the table there are a lot more missing values. This seems to contradict my idea that the augmented values would improve the matching accuracy. There are reasons for this, however.

The large number of missing values is a result of the inaccuracy of the analysis that The Echo Nest does coupled with the strictness of my search algorithm. While The Echo Nest does return a confidence value stating how accurate the result is likely to be, with noise introduced or a high pass filter the calculation of important values, such as key, can be wildly misinterpreted. This misinterpretation of the values I'm augmenting the fingerprint with then eliminates the correct tracks from being matched leading to missing match values as my search algorithm accepts augmenting values with a 50% confidence. It does mean however that although the correct tracks are missing from the results list, the false positives are cut down too though it isn't really measurable with a small sample size.

However, it does seem that my hopes for the augmented fingerprint being faster to search for were correct. Although the average overall time is only about 0.02 seconds faster on average, it was a consistent average. Also, the small size of the database doesn't reflect how well the augmented fingerprint method would scale in comparison to the basic fingerprint in terms of search times. However that hypothesis is difficult to test accurately with this system as SQLite is not designed to handle large data sets well.

8 Conclusions

From the results I cannot conclusively determine whether my augmented fingerprint is better or worse when compared to a standard fingerprinting method. The difference between search times is negligible and at such small values can easily be affected by other system processes while testing. While the augmented fingerprint method didn't return so many false positive results, it often returned no matches, which makes it difficult to assess if it has provided an improvement or if it is worse.

Provided, with a database that was designed for larger data sets, the search time decreased when more fingerprints had been added to the database I would conclude that the augmented fingerprint was indeed an improvement. But currently, for this implementation at least, it is not feasible to test.

Personally I would say that the reduction in false positives, even at the expense of extra no-matches, is an improvement. While the system reports no match, no matches to me is more accurate than misrepresenting a fingerprint as another as it's erring on the side of caution with regards to fingerprinting accuracy. However, for a unanimous improvement in accuracy

I was looking for a reduction in matches for false positives but not a reduction on the actual file.

8.1 Looking to the future

If I were to restart the project I would make several changes to my design and to my test plan.

To the design I'd change from using The Echo Nest for my audio analysis and either learn how to, and implement, the analysis features I need and build them into the project or create a java library that wrapped the VAMP plugin system so I could get more accurate analysis and hopefully reduce the occurrences of no-matches in the testing phase.

I'd also try to make time to add the features I felt could have improved the usability of the system, such as support for other file formats and other fingerprinting methods. It would have been nice to support more file formats because it would have saved time creating test data and would have let me fingerprint all my music much faster.

If I added other fingerprinting methods into the system, such as a proper Shazam-style fingerprinter or the use of a third party library like the one Gracenote provides I could test my system against actual commercial systems. If I added my augmented style searching to filter the results from a commercial system it might also be interesting to test if there was any reduction in false positives.

Looking to the future I would dedicate more time to how I stored and searched for matches with my fingerprint implementation. While it did fingerprint, store and search it was inefficient in storing all the data for each band and the search algorithm could have been more relaxed so as not to exclude some matches, while being stricter in what it defined as a match so as not to match the same chunk many times.

Bibliography

ALM WORKS. 2010. *sqlite4java - Minimalistic high-performance Java wrapper for SQLite*. [online]. [Accessed Feb 2011]. Available from World Wide Web: <http://code.google.com/p/sqlite4java/>

BALUJA, Shumeet and Michele COVELL. 2006. Content Fingerprinting Using Wavelets. *In*: *Visual Media Production, 2006. CVMP 2006. 3rd European Conference on*. London, pp.198 - 207.

BECK, Kent. 1995. *Extreme Programming Explained: Embrace Change*. Addison-Wesley Professional.

CLOUDGARDEN. 2010. *Jigloo SWT/Swing GUI Builder for Eclipse and WebSphere*. [online]. [Accessed January 2011]. Available from World Wide Web: <http://www.cloudgarden.com/jigloo/>

ELLIS, Dan. 2011. *E4896 - outline*. [online]. [Accessed 2011]. Available from World Wide Web: <http://www.ee.columbia.edu/~dpwe/e4896/lectures/E4896-L13.pdf>

GRACENOTE. 2011. *Gracenote | Ford SYNC*. [online]. [Accessed 2011]. Available from World Wide Web: <http://www.gracenote.com/casestudies/ford_sync/>

GREENSTED, Dr. Andrew. 2010. *Java Wav File IO*. [online]. [Accessed December 2010]. Available from World Wide Web: <http://www.labbookpages.co.uk/audio/javaWavFiles.html>

HAITSMA, Jaap and Ton KALKER. 2002. A Highly Robust Audio Fingerprinting System. *In*: *Int Symp Music Info Retrieval, Vol. 32*. Ircam - Centre Pompidou, pp.107-115.

IEEE. 1995. *IEEE Computational Science and Engineering*. IEEE Computer Society.

JAMES, Mark. 2005. *Silk Icons*. [online]. [Accessed January 2011]. Available from World Wide Web: <http://www.famfamfam.com/lab/icons/silk/>

LAST.FM. 2007. *Audio Fingerprinting for Clean Metadata*. [online]. [Accessed 2011]. Available from World Wide Web: <http://blog.last.fm/2007/08/29/audio-fingerprinting-for-clean-metadata>

MICROSOFT CORPORATION. 2005. *Testing Methodologies*. [online]. [Accessed 2011]. Available from World Wide Web: <http://msdn.microsoft.com/en-us/library/ff649520.aspx>

MUSIC BRAINZ. 2006. *Future Proof Fingerprint*. [online]. [Accessed December 2010]. Available from World Wide Web: <http://musicbrainz.org/doc/Future_Proof_Fingerprint>

QUEEN MARY, UNIVERSITY OF LONDON. 2007. *Vamp Plugins*. [online]. [Accessed 2010]. Available from World Wide Web: <http://vamp-plugins.org/>

SCHMIDT, Geoff. 2005. *Future Proof Fingerprint Function*. [online]. [Accessed December 2010]. Available from World Wide Web: <http://musicbrainz.org/doc/Future_Proof_Fingerprint_Function>

SEDGEWICK, Robert and Kevin WAYNE. 2011. *Complex.java*. [online]. [Accessed December 2010]. Available from World Wide Web: <http://introcs.cs.princeton.edu/97data/Complex.java.html>

SEDGEWICK, Robert and Kevin WAYNE. 2011. *Data Analysis*. [online]. [Accessed December 2010]. Available from World Wide Web: <http://introcs.cs.princeton.edu/97data/>

SEDGEWICK, Robert and Kevin WAYNE. 2011. *FFT.java*. [online]. [Accessed March 2011]. Available from World Wide Web: <http://introcs.cs.princeton.edu/97data/FFT.java.html>

SHNEIDERMAN, Ben. 1998. *Designing the user interface: Strategies for effective human-computer interaction*. Reading, MA: Addison-Wesley.

TANGO DESKTOP PROJECT. 2009. *Tango Desktop Project*. [online]. [Accessed January 2011]. Available from World Wide Web: <http://tango.freedesktop.org/Tango_Desktop_Project>

THE ECHO NEST. *About Us*. [online]. [Accessed 2011]. Available from World Wide Web: <http://the.echonest.com/company/>

WANG, Avery Li-Chun. 2003. *An Industrial-Strength Audio Search Algorithm*. [online]. [Accessed 2010]. Available from World Wide Web: <http://www.ee.columbia.edu/~dpwe/papers/Wang03-shazam.pdf>

YOUTUBE. 2008. *Copyright Overview*. [online]. [Accessed 2011]. Available from World Wide Web: <http://www.youtube.com/t/content_management>

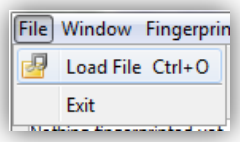
Appendices

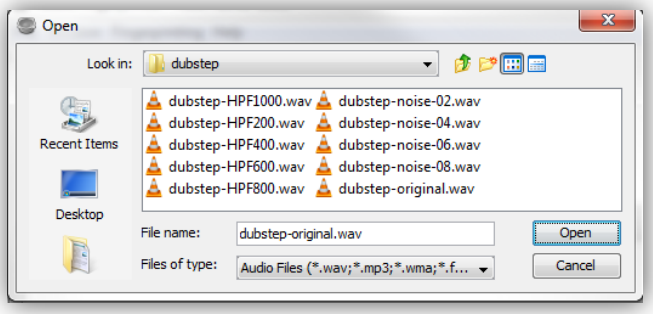
Appendix A - Database SQL

```
CREATE TABLE IF NOT EXISTS tracks (
  id INTEGER PRIMARY KEY,
  artist VARCHAR(128),
  title VARCHAR(128),
  album VARCHAR(128),
  duration NUMERIC,
  tempo NUMERIC,
  key NUMERIC,
  mode NUMERIC,
  time_signature NUMERIC,
  loudness NUMERIC
);

CREATE TABLE IF NOT EXISTS print_runs (
  id INTEGER PRIMARY KEY,
  augmented NUMERIC,
  timestamp VARCHAR(128),
  filename VARCHAR(128)
);

CREATE TABLE IF NOT EXISTS print_data (
  chunk INTEGER,
  band1 INTEGER,
  band2 INTEGER,
  band3 INTEGER,
  band4 INTEGER,
  band5 INTEGER,
  band6 INTEGER,
  band7 INTEGER,
  band8 INTEGER,
  band9 INTEGER,
  track_id INTEGER,
  print_run_id INTEGER
);
```

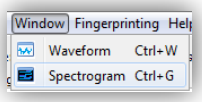
Appendix B – User Manual

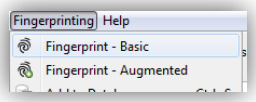
To fingerprint, either basic or augmented, an audio file click File -> Load File from the menu, or press Ctrl+O to get the file chooser. Choose the file you wish to fingerprint and click open.



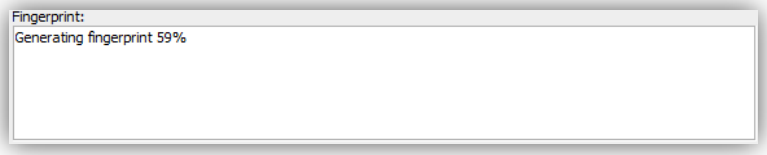


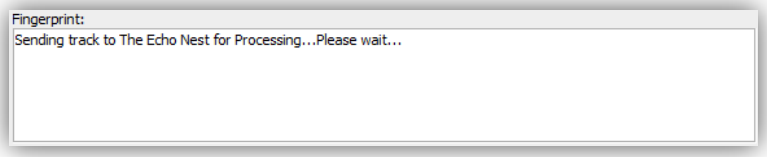
Once the file is loaded you can look at the waveform or spectrogram of the audio by selecting the value from the Window menu or by pressing Ctrl+W or Ctrl+G repectively, or generate a basic or augmented fingerprint using the options in the Fingerprint menu.

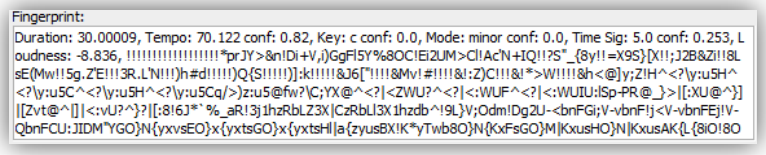




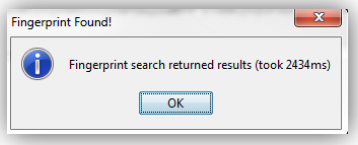
Once a fingerprint has been generated a text representation of it will show up in the box labelled Fingerprint on the main form. You can then either add the fingerprint to the database or search for it in the database using Ctrl+S or Ctrl+F respectively.

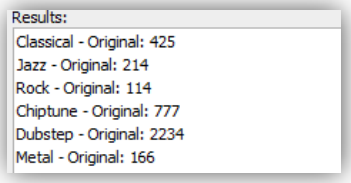




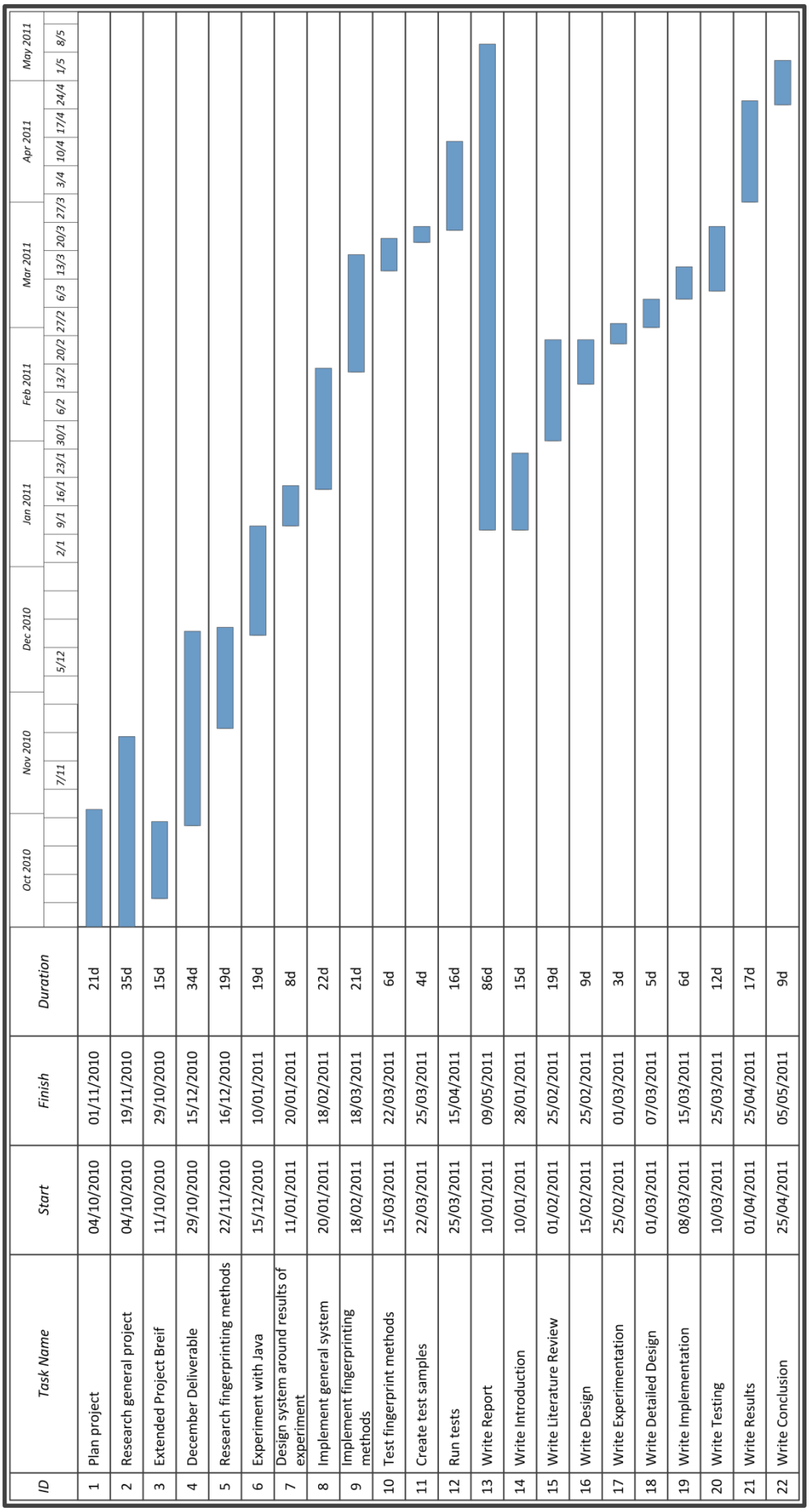


After a search has finished results are displayed in the list on the main window labelled Results.



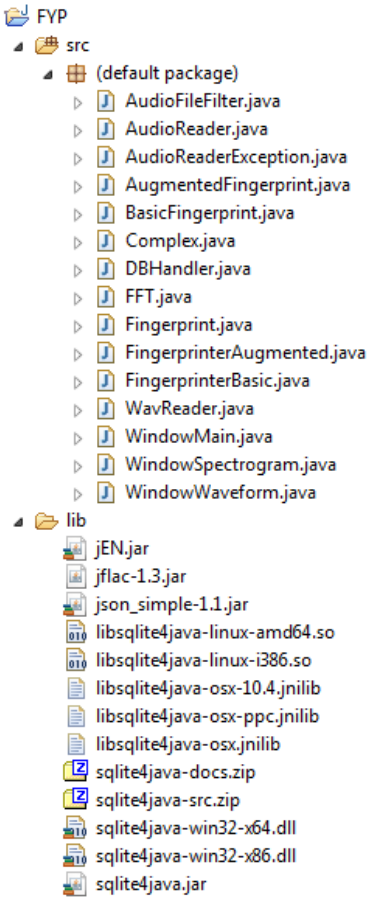


Appendix C – Gantt Chart



ID	Task Name	Start	Finish	Duration
1	Plan project	04/10/2010	01/11/2010	21d
2	Research general project	04/10/2010	19/11/2010	35d
3	Extended Project Breif	11/10/2010	29/10/2010	15d
4	December Deliverable	29/10/2010	15/12/2010	34d
5	Research fingerprinting methods	22/11/2010	16/12/2010	19d
6	Experiment with Java	15/12/2010	10/01/2011	19d
7	Design system around results of experiment	11/01/2011	20/01/2011	8d
8	Implement general system	20/01/2011	18/02/2011	22d
9	Implement fingerprinting methods	18/02/2011	18/03/2011	21d
10	Test fingerprint methods	15/03/2011	22/03/2011	6d
11	Create test samples	22/03/2011	25/03/2011	4d
12	Run tests	25/03/2011	15/04/2011	16d
13	Write Report	10/01/2011	09/05/2011	86d
14	Write Introduction	10/01/2011	28/01/2011	15d
15	Write Literature Review	01/02/2011	25/02/2011	19d
16	Write Design	15/02/2011	25/02/2011	9d
17	Write Experimentation	25/02/2011	01/03/2011	3d
18	Write Detailed Design	01/03/2011	07/03/2011	5d
19	Write Implementation	08/03/2011	15/03/2011	6d
20	Write Testing	10/03/2011	25/03/2011	12d
21	Write Results	01/04/2011	25/04/2011	17d
22	Write Conclusion	25/04/2011	05/05/2011	9d

Appendix D – Source Code Listing



Source code files included in the project are listed to the left.

Along with the source files are a set of libraries that must be linked in to the project for it to compile. For the origins of the libraries, look to the third party library section of the detailed design and the bibliography.

All source code and libraries are included on the CD provided with this report. A compiled version is also included in the form of a runnable jar. If running on Apple Mac OS X, see the end of the Environment section in the design earlier in this report for a command line option to increase Javas default memory limits.