2010

Perceptual audio classification using principal component analysis

Zak Burka

Perceptual Audio Classification Using Principal Component Analysis

Zak Burka

A Thesis Submitted in Partial Fulfillment of the Requirements for the Degree of Master of Science in Computer Science

Department of Computer Science
Golisano College of Computing and Information Sciences
Rochester Institute of Technology

Approved May 6, 2010

Abstract

The development of robust algorithms for the recognition and classification of sensory data is one of the central topics in the area of intelligent systems and computational vision research. In order to build better intelligent systems capable of processing environmental data accurately, current research is focusing on algorithms which try to model the types of processing that occur naturally in the human brain. In the domain of computer vision, these approaches to classification are being applied to areas such as facial recognition, object detection, motion tracking, and others.

This project investigates the extension of these types of perceptual classification techniques to the realm of acoustic data. As part of this effort, an algorithm for audio fingerprinting using principal component analysis for feature extraction and classification was developed and tested. The results of these experiments demonstrate the feasibility of such a system, and suggestions for future implementation enhancements are examined and proposed.

Approved By:

Dr. Roger Gaborski
Director, Laboratory of Intelligent Systems, Department of Computer Science
Thesis Committee Chair

Dr. Joe Geigel
Associate Professor, Department of Computer Science
Thesis Committee Reader

Dr. Peter G. Anderson
Professor Emeritus, Department of Computer Science
Thesis Committee Observer

Contents

List of Figures

1 Introduction

As the quantity and availability of digital multimedia content continues to grow worldwide, organizing and classifying this digital information efficiently becomes increasingly more important. As the size of this digital cache increases through online digital media services such as Flickr, YouTube, and Lala, it becomes ever more imperative to effectively and efficiently categorize this data appropriately in order to extract relevant metadata to aid in both search and categorization. The sheer size of this "multimedia mass" and it's rate of growth make it impossible to rely entirely on user-aided classification and metadata tagging strategies. Thus, efficient algorithms and intelligent systems capable of the automatic categorization of digital data are becoming more and more necessary.

In the domain of intelligent systems and computer vision, the effective classification of digital data is one focus area of current research. Algorithms for object detection, object recognition, scene understanding and motion tracking are continually being developed and improved. These techniques have applications to the domain of autonomous mobile platforms, surveillance, ground processing of field-acquired sensor data, and the automatic categorization of unordered digital databases.

As research and development of intelligent systems continues, the ability to classify auditory data from an environment in addition to visual data is an important new dimension which can assist in scene understanding. Similarly, certain applications of intelligent systems might require the need to effectively process and react to acoustic events in the environment, making it necessary for the development of robust audio classification techniques.

The area of audio classification is diverse, with various applications such as speech recognition, voice detection, language and sex identification, musical identification and genre classification. Algorithms for these various problems are very application specific, but they each apply to the broad class of acoustic classification techniques.

The focus of this research is on an acoustic classification technique known as audio fingerprinting. In general, audio fingerprinting refers to the method of classifying some audio signal from the environment against a previously heard version of that signal. The environmental sampled signal is usually expected to be distorted in some way that was not present in training, so robustness to distortions is a core metric with which audio fingerprinting systems are measured against. In addition, other important properties of audio fingerprinting systems include robustness to alignment variations between the sampled test signal and the training signal, database lookup speed, and accuracy.

Presented below is an algorithm for audio fingerprinting which uses principal component analysis for feature extraction and classification. A general framework for audio fingerprinting systems is discussed in Section 2.1, some implementation examples of audio fingerprinting systems are given in Section 2.2, and the algorithm developed in this effort is proposed and discussed in Section 3. In Section 4, some experiments are performed on the algorithm, and a detailed discussion of the results is given in Section 5. Finally, future work and enhancements are detailed in Section 5.3.

2 Overview

In order to measure the quality of an audio fingerprinting system, it is important to first develop a formal understanding of what such a system entails, as well as a concept of some core parameters and terminology. In general, audio fingerprinting is an acoustical classification technique which attempts to categorize and identify audio samples via their temporal characteristics in order to extract meaningful metadata. This information can be used not just for content identification purposes, but also for copyright protection, digital watermarking, integrity verification, ect...

While “acoustic classification” refers to any method of organizing acoustic data, the term “audio fingerprinting” refers to a method of classifying a digital sample of an audio signal (the “fingerprint”) by matching it to a “master signal” in a database containing the sampled fingerprint. The core concept of the algorithm involves extracting temporal features of the data, such as frequency and time deltas, in order to reduce the dimensionality of the problem so that the input can be classified in this reduced space more efficiently.

The advantage of audio fingerprinting over other techniques for acoustical data classification comes from the fact that the temporal data itself is used as the classification input as opposed to the digital representation of the data from which the sound is produced. Because this technique uses the acoustical signal itself, it is thus robust to varying compression formats and different digital distribution and storage methods. This characteristic of audio fingerprinting systems make it well suited to process audio signals from any variety of binary sources; for example mp3s, CDs, AM/FM radio, or vinyl. It is this content-based classification attribute which makes audio fingerprinting said to be “perceptual” in nature, as the audio signal is processed in a way similar to how the signal is processed in the human ear and interpreted in the auditory cortex.

Audio fingerprinting works by matching *exact* fingerprint samples to their corresponding entries in a database. This is notably different to other types of intelligent classification techniques which attempt to relate test data to *similar* training data. As such, this technique is extremely sensitive to pitch and phase deviations from the master signal, as well as live performances or other types of re-interpretations of the master data record in the training database. However, the techniques used in audio fingerprinting make it very resilient to background noise and other signal degenerates that are not apparent in the original master signal. Because of this, it is expected that a version of the master signal, though distorted, is used for classification testing.

Because the fingerprint sample needs to be classified against a set of training data in an efficient manner, and because this data set can be very large, this problem also contains a substantial database component to it. The fingerprint sample features are typically used to calculate a “perceptual hash” value, and this hash value is then used to efficiently lookup the matching master record in the database. When categorizing large records of audio data, it is important to have the capability to lookup test records in an efficient manner.

2.1 General Audio Fingerprinting Principles

While each audio fingerprinting system can operate on different principles and algorithms, Cano *et. al.* propose a general framework for all audio classification systems to which most systems apply. Each fingerprinting system, they argue, is composed of a few fundamental requirements for both the generation and analysis of the fingerprint as well as the fingerprint matching algorithm. They are as follows [1]:

- High discriminatory power of the fingerprint
- Distortion invariant
- Compactness
- Computational cost

The authors present a high level generic model [1] to meet these requirements which consists of both a “fingerprint extraction” section and a “fingerprint matching” section. The fingerprint extraction section contains both a front-end [2] as well as a modeling component. The fingerprint matching section contains the corresponding database search functionality as well as the classification portion which attempts to match the input sample fingerprint with the closest sample from the database.

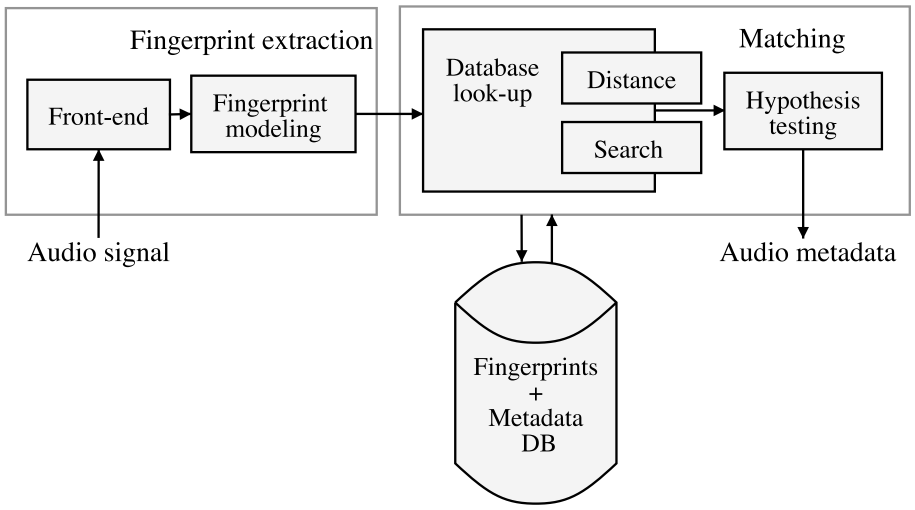

Figure 1: High Level Model of Generic Fingerprinting Framework[1]

The fingerprint generation algorithm is generically presented as being composed of five subsections: *Preprocessing*, *Framing & Overlap*, *Transformation*, *Feature Extraction*, and *Post-Processing*2.

2.1.1 Preprocessing Phase

The preprocessing phase consists of generating the digital representation of the analog sound signal into some common, raw form. This can be done via recording a live stream from a record, CD, mp3, or some other type of representation. This can also include some simple scaling or filtering to ensure a common intensity for all sampled signals. Generally, the raw data will be sampled down to a lower rate which the rest of the algorithm will operate on[†]. This down-sampling

[†] In the algorithm developed here, the raw audio was down-sampled to 11025Hz

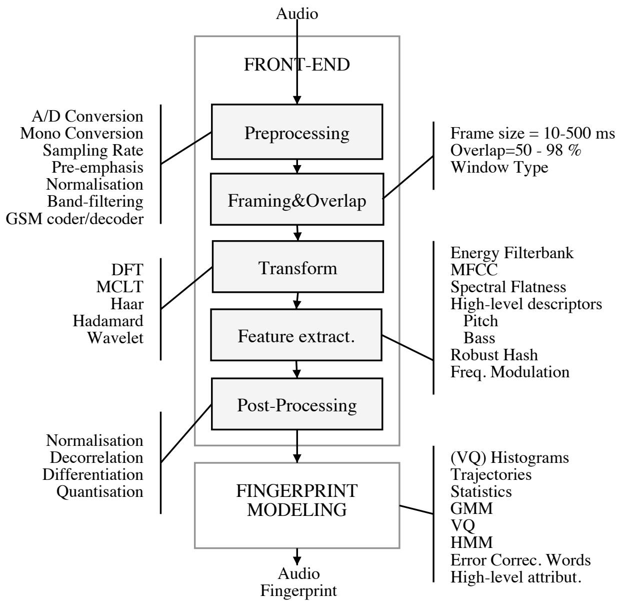

Figure 2: Fingerprint Calculation Model[1]

is performed in order to reduce the size of the data to be processed, while still retaining much of the important temporal characteristics of the signal. The data can also be converted to mono here if the input is from a multichannel source, and the bit-depth of the signal might also be reduced as well.

It is important to note that this pre-processing is applied in both the training and testing phases. Because some filtering and sub-sampling or interpolation could be performed on the signal, it is important that upon exit from this processing block, the signals in both training and test have similar characteristics (sampling rate, bit-depth, number of channels, etc...).

2.1.2 Framing & Overlap Phase

The framing and overlap phase involves breaking the sound sample up into many different clips with sufficient overlap for analysis. These overlapping clips from the input signal are referred to as frames. A diagram showing how overlapping frames are taken from an input audio signal is shown in [3].

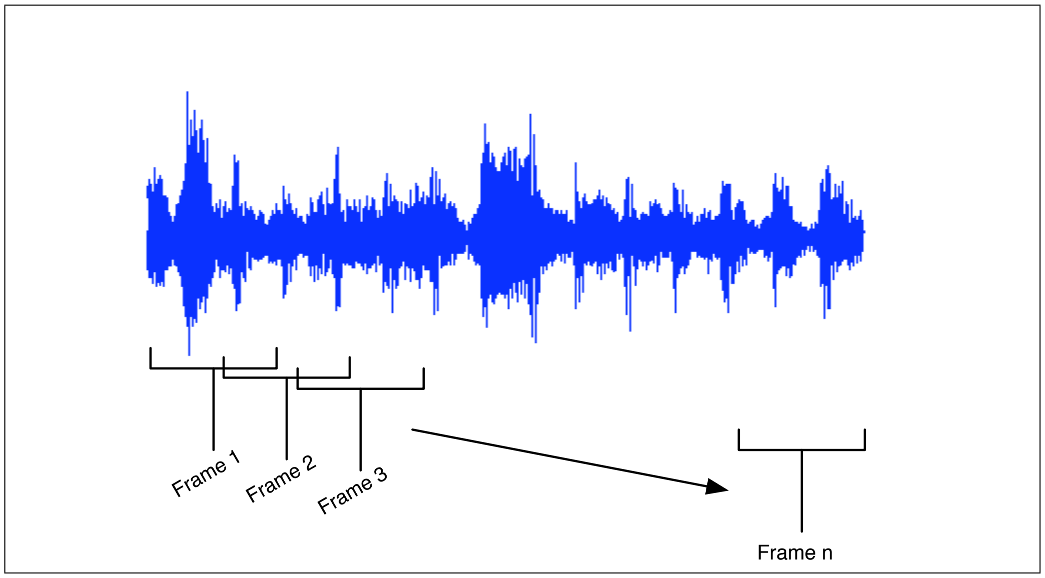

Figure 3: Example of splitting audio into overlapping frames

It is important that there is sufficient overlap between the frames to ensure that the analyzed sample signal can be matched to the reference master signal in the database. The master signal will have already been characterized with the same fingerprinting scheme, and will have a matching framing and sample rate. It is this overlap of the frames which allows the fingerprint signal to be sampled at any point in the clip and be matched to that corresponding point in the master signal. From this point forward in the algorithm, all subsequent phases are applied then to *each* frame individually from the fingerprint source.

2.1.3 Transformation Phase

In the transformation phase, various linear transformations are applied to the input audio signal in order to convert it to a domain which will ease feature extraction. Typically, this involves performing some sort of operation to decompose the signal into its frequency or power spectrum components. Various transformations can be used; some common ones are the Fast Fourier Transform (FFT) and the Discrete Cosine Transform (DCT).

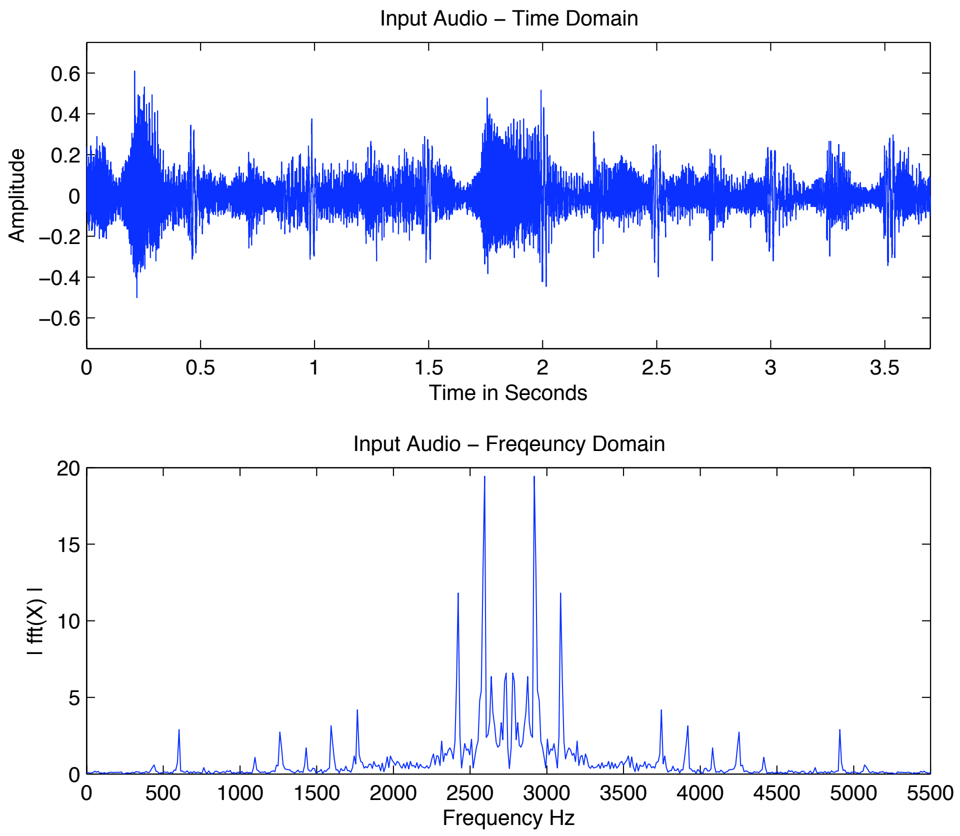

Figure 4: Comparison between time and frequency domain of an audio signal

An example of an input audio signal in both the time and frequency domain is shown in [4]. In this example, an input clip of audio of about 3.5 seconds in duration was captured and down sampled to 11025Hz and converted to mono. The temporal signal is displayed, and a plot of the output of the signal following an FFT operation is displayed showing the various frequency components contained within the signal[†].

Because audio signals are time variant, transforming them into a frequency domain allows for statistical-based features to be extracted from them such as pitch, tone, and power. These types of characteristics are useful when analyzing audio because they can provide a metric with which to quantify the traits of the signal against. It is in this domain that the various features that are used for classification in an audio fingerprinting system are extracted from.

The purpose of the transformation phase is to decompose the fingerprint sample in order to generate statistical information about the signal's characteristics (such as frequency and power spectrum information). While FFT-type transformations are typical, other transformations such as the Discrete Wavelet Transform (DWT) or Singular Value Decomposition (SVD) can also be used to feed into the feature extraction block.

2.1.4 Feature Extraction Phase

In the feature extraction phase, certain characteristics from the audio frame following transformation are obtained. The specific features that are sampled from the frame vary between different systems, but in general, these features are used to reduce the dimensionality for classification. Some examples of features that can be used include measuring the spectral flatness, critical-band sections, band energy, loudness, and bandwidth.

[†] NOTE - For this plot, the a 1024 length FFT is taken and shifted so that the zero-frequency component is located at the center of the plot.

This phase in the processing chain of the algorithm is very important, as the characteristics taken here are used by the classifier to determine signal matches. This process reduces the dimensionality of the data both in terms of data size, and also in terms of the classification space which is used to determine membership of a particular class.

Both of these qualities are important in an audio fingerprinting system. A reduced data size allows the classifier algorithm to be more efficient, especially when the classification involves searching a large database for matches to previously registered audio signals. Also, a reduced feature space for classification allows the algorithm to be more accurate, as the features extracted are expected to be robust to various types of distortions as well as highly discriminate.

2.1.5 Post Processing Phase

The post-processing phase is responsible for developing the feature vectors to be used in the modeling block. In this step, the extracted features can be averaged, normalized, quantized, ect... The processing performed in this phase is dependent on the algorithm used in the modeling block, and is responsible for improving the accuracy of the classification process by making the input data more uniform.

The modeling block is then responsible for packaging up the various processed features into a distinguishable form for either retrieval from the database in the case of classification, or to be added to the database in the case of training. It is important that in this step the features are sufficiently reduced to allow for efficient database access. In implementation, usually the data in this stage is reduced to some hash value that can be easily queried by the target database.

2.1.6 Recognition Phase

The recognition phase of the algorithm is responsible for receiving a fingerprint from a test signal, and determining if there is a match contained in the previously trained database. The fingerprints for test are generated via the same process as in training, and it is expected that at this stage there is an efficient way to calculate similarity between the testing and training frames.

There are various ways to calculate frame similarity, and each fingerprinting system can use a different approach, but in general it is done by gathering the hashed value record for the associated input keys, performing some kind of distance measurement between them, and using this data within a reasonable threshold to produce a best guess of the matching entry.

2.2 Audio Fingerprinting Systems

While intelligent systems research is continually interested in exploring various new approaches and techniques to improve audio fingerprinting algorithms, there are a few implementations of fingerprinting systems currently available. These can vary from embedded types of systems that work on portable electronic devices such as cell phones and PDAs and identify songs recorded from the environment on the device's microphone[1], to desktop media player software which can automatically populate the currently playing song's metadata via perceptual identification[2]. There are even open source versions of audio fingerprinting software, distributed under the GPL license, complete with publicly maintained databases of fingerprinted songs[3].

[1] `http://www.shazam.com/`

[2] `http://research.microsoft.com/en-us/um/people/cburges/rare.htm`

[3] `http://musicbrainz.org/`

Two commercial implementations - “Shazam” by Shazam Entertainment Ltd. and Microsoft’s RARE engine, are explored in detail below.

2.2.1 Shazam

One popular application for audio fingerprinting technology has been consumer electronics. Portable devices, such as cell phones, mp3 players, or portable computers, are being used as mobile classification devices which are able to recognize songs in the environment and display the associated metadata to the users. One such example of this software is “Shazam”, which is developed by Shazam Entertainment Ltd., and is available on a number of different mobile platforms including the iPhone, Blackberry, and Android based devices. In their published white paper, Wang *et. al.* describe the algorithm used by the Shazam service in detail[4].

The crux of any audio fingerprinting algorithm lies in the feature extraction phase of the processing chain. In the Shazam algorithm, the frame samples are used to generate spectrograms relating the temporal signal to its frequency and intensity values. An example spectrogram plot for an audio sample is displayed in [5].

From the spectrogram, the Shazam algorithm uses thresholding to extract various peak intensity values from the plot. The peaks chosen are of relatively high energy compared to their neighbors, and are thus less susceptible to background noises and interference. The term “constellation map” is used in the literature to refer to the extracted peak value spectrogram plot. An example constellation map is displayed in [6].

Once the peak values have been extracted from the spectrogram, the points are paired together with neighboring points to create a “hash-point pair”. The hash value is thus a combination of the location of both points in the pair, as well as the time delta between them. These point pairs are used to reduce the amount of time

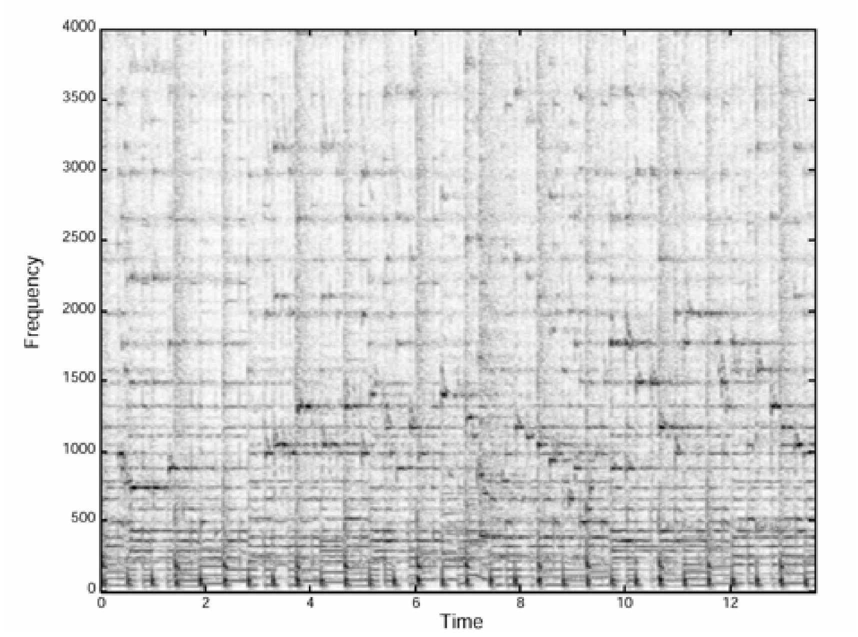

Figure 5: Spectrogram plot of audio sample[4]

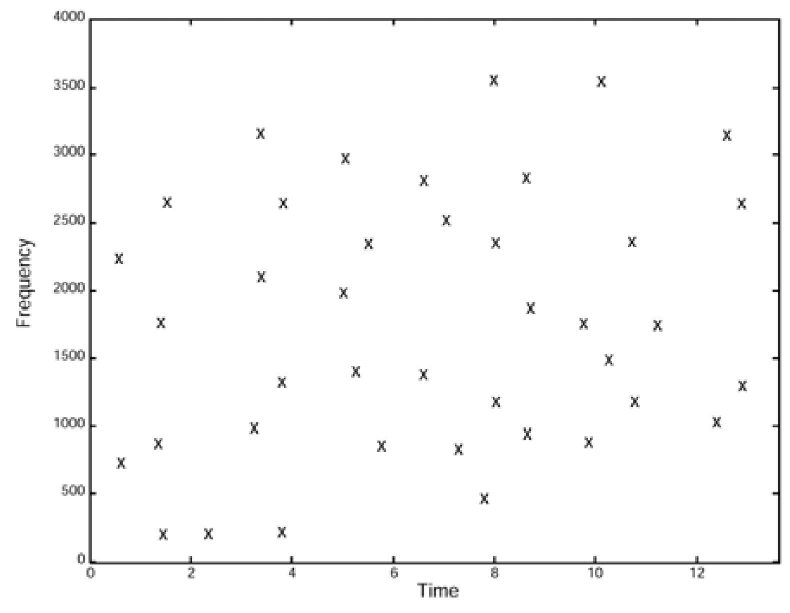

Figure 6: Constellation map of audio sample[4]

that the database lookup takes, as well as to to provide another dimension to the hash value used and also to reduce the amount of data that a single hash requires - the authors cite that a single hash value for a point pair from a constellation map containing 1024 frequency bins can be stored in a 32bit unsigned integer. An example of the point pairing algorithm in the constellation space is displayed in[7].

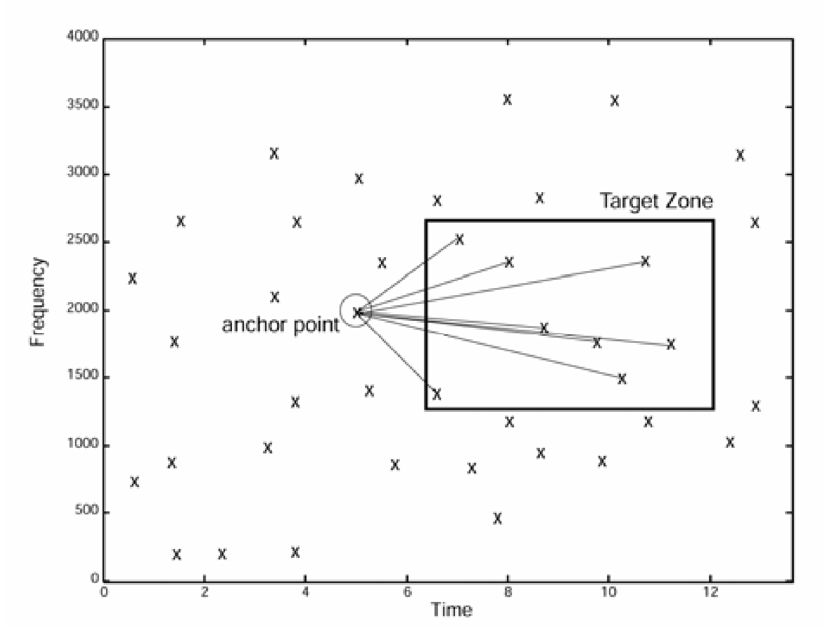

Figure 7: Point Pairing in Constellation Map[4]

Once the point-pair hashes have been computed for each of the samples in the fingerprint, the hash values are then packaged up along with their time offsets, and sent to the database for classification. The algorithm then searches the database for the matching hash values, and, for each hit within some reasonable threshold, it keeps track of the song to which it applies as well as the relative location in the song. (Again, this same algorithm was used to classify each song in the database already.)

After the database lookup operation, the algorithm then has a list of peak intensity pair hits, the songs to which these frequency hits are associated, and the relative location in time that each of these frequency hits occurred. Then, for each song present in the hash value hits, a scatterplot is made of the input sample relative times vs. the database song relative times for the associated point-pair locations. An example of this plot is shown in [8].

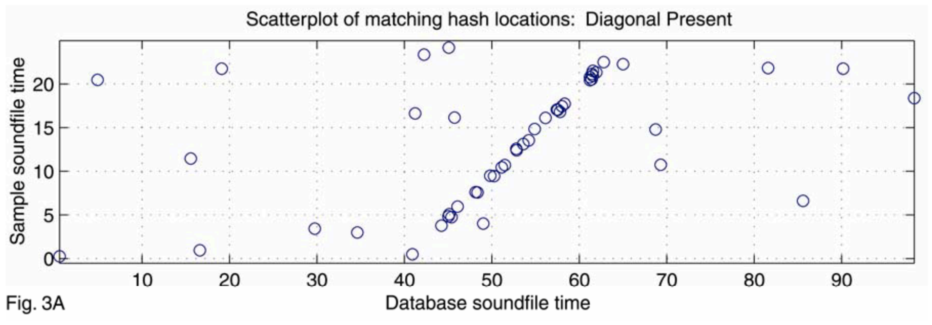

Figure 8: Scatterplot of Sample Relative Time Hits vs. Master Relative Time Hits[4]

The problem of verifying whether or not the sample song is a match for the master song is thus reduced to finding the presence of a strong diagonal line in one of the scatterplots. The authors perform this step in $Nlog(N)$ time by defining t_k as the time value in the sample fingerprint for the frequency match and t'_k as the corresponding point time value in the master database and for each (t'_k, t_k) pair computing $\delta t_k = t'_k - t_k$. Because the offset between the sample and master time value differences should be constant for matching songs, a histogram is then taken of the δt_k data and used to see if a peak occurs indicating that many values have a similar offset. If a peak within some predefined threshold is present, the song is said to be a match[†].

The Shazam algorithm has been shown to be robust to environmental noises as well as resistant to false positives. Because of the way that point-pairs are used in combination with relative time offset for hash indexing, the authors cite a false

[†] Note - the run time for this algorithm is asymptotically upper bounded by an $Nlog(N)$ sort which is done on the deltas prior to performing the histogram

positive rate in the order of 0.01-0.1% and the ability to correctly match corresponding songs when only around 1-2% of the sample hash tokens survive in the sample.

2.2.2 RARE

Another implementation example of an audio fingerprinting system can be found in Microsoft's RARE engine, which powers the fingerprinting capability built into Windows Media Player (WMP). This technology is used to automatically generate metadata from audio content played within the software in order to retrieve information such as song name, artist name, and album art. The RARE engine is a combination of a number of different algorithms and technologies, however the main classifier algorithm was developed by Burges *et. al.* at Microsoft Research Labs, and is referred to as Distortion Discriminant Analysis (DDA)[3].

The DDA approach to fingerprinting, while similar to the Shazam algorithm in many ways, is very different in the feature extraction block of the algorithm. Input audio signals are still pre-processed, down-sampled, split into frames, and transformed via a linear operation, however the features that are used for classification are extracted using multiple layers of Principal Component Analysis (PCA)[†]. PCA is a data analysis technique which attempts to "automatically" extract the most meaningful features from a multi-dimensional data set. An in depth look at PCA and its application to audio data is described in detail in 3.2.5.

An attractive feature of using an information theory based approach such as PCA over traditional segmentation techniques such as frequency and energy spectrum analysis, comes from the possibility of finding more robust "intrinsic" features of the audio signal that outperform classification from features extracted using stan-

[†] The DDA algorithm actually uses "oriented" PCA (OPCA) on the audio data. This is only slightly different to standard PCA, and involves taking into account the signal-to-noise variance in the feature projections. More information can be found in [3]

dard heuristics. The desire is that the application of PCA can automatically capture the most meaningful mathematical properties of an audio signal in the same way that it does in other applications of data analysis, such as PCA applications to imaging systems.

In PCA based applications, the principal features for a data set are calculated from some number of different measurements (or samples) of the data being classified. These features can have an arbitrary number of dimensions, but many different measurements of the sample must be taken in order to derive the principal components across the entire set. For example, in imaging applications multiple images of the same sample, for instance multiple different images of a person's face, can be used to define the class for which to calculate the principal components of. In audio fingerprinting however, deriving multiple samples from some input audio must be performed artificially as the only source of input to train with is the sole audio clip.

In the DDA algorithm, multiple artificial distortions are mathematically applied to the incoming signal in order to train the system. It is desired to use distortions that might be typical, though not identical, of the types of distortions that might be contained in test signal when sampled from the environment. Some of the distortions used are a 3/1 compressor above 30dB, a compander, a spline boost between 1.2KHz and 5KHz, and a spline notch filter among others. In all, nine different distortions are applied, and thus nine different distorted version of the input audio signal are used for training.

The DDA algorithm works by first converting the input audio signal to 11025Hz mono, and splitting the signal into overlapping frames of 4096 samples. This amounts to a frame size which spans approximately 372ms in time. These frames are then each decomposed into the frequency domain via a lapped transformation[†],

[†] The DDA algorithm (and the algorithm proposed in section 3), use the Modulated Complex Lapped Transformation (MCLT) for frequency decomposition. The MCLT algorithm is described in detail in section 3.2.3

and are then further pre-processed to remove equalization and volume variances as well as frequency ranges that are inaudible to the human ear. This is done for each of the distorted versions of the training frame, and this final collection of pre-processed frames is then fed to the PCA-based feature extractor for classification.

The feature extraction portion of the algorithm works by performing PCA on the pre-processed frame in layers. An example of the multi-layered approach of DDA is shown in 9. In the first layer, each frame is projected on to a 64 dimensional space, where the values are represented by those projections with the 64 highest eigenvalues. This is done individually for 32 consecutive frames, and each frame's 64 projections are retained. These projections are then concatenated together, and another layer of PCA is performed to reduce the set to another 64 projections. These final 64 dimensions are thus the fingerprint which is calculated for both training and testing. By performing PCA in layers, the final fingerprint thus spans a much wider temporal region than could be computationally feasible if calculated directly, and results in converting 6.1 seconds of audio into a fingerprint of 64 numbers.

The DDA algorithm also contains a unique approach to misalignment compensation. Alignment robustness refers to the resiliency of an algorithm to variations in the positional alignment of a test frame to that of the matching signal frame in the training database. This can occur when an audio signal is sampled from the environment where the position of the start of the first frame is unlikely to be in exact alignment with the start of that same frame in the training set. The DDA algorithm accounts for alignment variations during the training portion of the algorithm by shifting the training signal forward and back by 1/4 frame and treating it as another distortion. This way, the principal components of the frame that make up the fingerprint are actually "smeared" temporally about a small region of time. This effectively allows the algorithm to compensate for variations in testing alignment, by making the lookup operation more resilient to slight shifts in time for a given frame. Large shifts in time, on the other hand, are compensated for by the use of

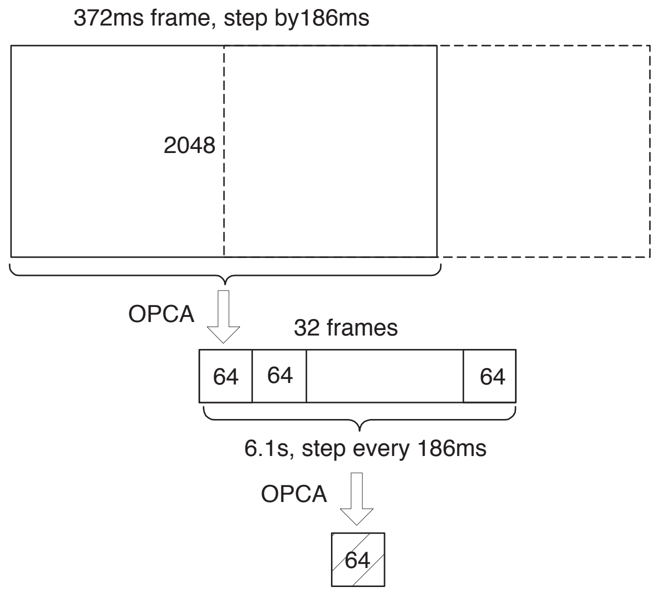

Figure 9: Application of Multi-Layered PCA in DDA Algorithm[6]

overlapping the frames by 1/2 frame length.

In the testing phase, the test clip is run through the same multiple layers of PCA in order to derive the 64 dimensional "test" fingerprint. This fingerprint value is then used to search for a match in the training database by calculating the Euclidean between it and the fingerprints stored during training. If this distance is within some threshold, the algorithm will report that the song had found a match, and can then return the appropriate metadata for the song in question.

The authors of the DDA algorithm report results showing the application of OPCA outperforming both PCA and Bark averaging when calculating the Euclidean distance between a training signal and a testing signal. They also report results from full system testing over many days of audio classification showing a false positive rate of 1.5×10^{-8} per test clip[3]. They also note the algorithm's reduced sensitivity to phase shift variations over other, purely heuristic based models.

3 Algorithm

In the previous sections, an overview of general audio fingerprinting principals were reviewed and some commercial implementations discussed. In the following sections, the algorithm that was developed for this project is proposed and analyzed.

The goal for this development effort was to explore the implementation of an audio fingerprinting system which uses PCA-based approaches for feature extraction and classification. This algorithm builds on previous efforts that were completed as part of an Independent Study with Dr. Gaborski in Winter quarter of 2009. In that initial investigation, various principals of audio fingerprinting systems were explored, and an initial implementation of a fingerprinting algorithm was developed. Frameworks and code for dealing with audio data and fingerprinting systems were developed, and a rudimentary classifier was implemented and analyzed.

The main focus of this effort was the development of an original classifier algorithm that could quickly and efficiently produce accurate results for fingerprint analysis. Also, various pre-processing techniques were utilized to improve the accuracy of the algorithm, and optimizations were performed to improve the run time. These techniques and their investigation are described in detail in section 3.3. The results from various experiments are discussed in section 4.

3.1 Algorithm Overview

The algorithm developed for this project generally follows the model for fingerprinting systems proposed by Cano *et. al.* and discussed in section 2.1. An overview of the processing chain is shown in figure [10].

Similar to other fingerprinting system, this algorithm contains both training and

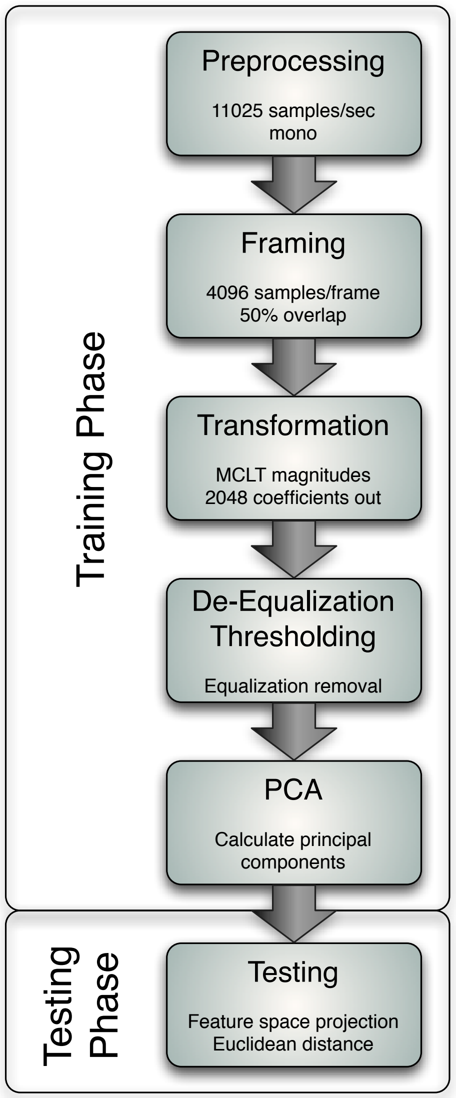

Figure 10: Processing Chain

test phases. In both phases of the algorithm, the input audio is run through each block, with the exception that in training, after the PCA block, the training data is added to the database. Only in the testing block does the test signal get projected onto the feature space of the training data and the Euclidean distance measured for classification. Following, each processing block in the classification chain is analyzed.

3.2 Training

3.2.1 Preprocessing

In order to generate the fingerprint database in the training phase of the algorithm, the audio signals (songs) first need to be uniformly converted to some common values. As such, each song is first down-sampled to 11025Hz and converted to mono. There are many ways that this conversion may be performed, however for the purposes of these experiments, the BSD `afconvert` tool from the Mac OS X tool suite was used along with a Python script to generate the appropriate raw files from a given input directory of mp3s.

Following this subsampling conversion, the Adobe Audition tool suite is used to apply given distortions to each master song, and to generate a new distorted version of the song for each given distortion. The distortions used in these experiments were: a Compander, a De-Esser filter, an Expander, a High Frequency Hiss cutter, a Notch filter @ -6dB from 430Hz-3.4kHz, an “Old Radio” filter, a “Slow Drums” filter, a Generic High Pass filter, and a spline boost filter from 1.2kHz-5kHz.

Scripts were used to interface with the Audition tools and to allow the batch processing of the given input songs. Following this step, there are 9 distorted versions of each song, each with a sampling rate of 11025Hz, and 1 “master” version of the song which was generated from the mp3 and used to generate the distortions. These 10 songs are then used as input to the framing block of the processing chain.

3.2.2 Framing

In order to compensate for alignment variations, the input distorted songs are split up into overlapping "frames". Each frame is composed of 4096 samples, which corresponds to a temporal frame length of approximately $372ms$. These frames are generated at $\frac{1}{2}$ frame length increments, allowing each frame to be overlapped with both the preceding and following frame, and guaranteeing that each temporal region of the audio signal is contained within two separate frames. Note, however, that this is with the exception of the first $\frac{1}{2}$ of the first frame and the last $\frac{1}{2}$ of the last frame. This will result in a frame count for a given input audio signal of

$$2 \times \frac{NumberOfSamples}{FrameSize} - 1 \tag{3.1}$$

For a given frame duration, each of the distorted versions of the input signal from the same frame region is then input to the transformation block.

3.2.3 Transformation

Following framing, the audio data is next fed into the transformation block. In order to ease the classification of the data, it is desirable to extract various features from the input temporal signal in order to both reduce the dimensionality as well as to provide a more meaningful data set for which to classify against. One particularly useful feature space to work in for audio signals is the frequency domain.

There are various transformations and filters that can be used to extract frequency information from time domain signals, however one of the most common classes of these transformations are those based on the fast Fourier transform (FFT). The transform used in these experiments is one such FFT based algorithm, and is presented by Malvar in [8], and is called the “Modulated Complex Lapped Transform” (MCLT).

The MCLT is a type of *lapped* transform, meaning that it is optimized for applications to data sets that are “sub-windows” from some larger continuous set of data. It is actually an extension of the “Modulated Lapped Transform” (MLT), which is a type of lapped transform that utilizes iterative discrete Fourier transform (DFT) filter banks to perform frequency decomposition[9]. The MLT, because of it’s use of DCT filter banks, produces real coefficients laking any phase information[8]. The MCLT is a simple extension of the MLT, which adds in phase information via a discrete sine transformation (DST) calculation in addition to the DCT calculation from the MLT, which thus allows for the production of complex coefficients[†].

In [9], a length-M MCLT of a length-$2M$ signal block $x(n)$ is given as

$$X(k) = \sum_{n=0}^{2M-1} x(n)p(n,k) \tag{3.2}$$

where k is the frequency index, n is the time index, and $p(n,k)$ are the basis functions $p(n,k) = p_c(n,k) - jp_s(n,k)$ where $p_c(n,k)$ and $p_s(n,k)$ are defined as

[†] While the MCLT does produce phase information in the form of complex coefficients, these are discarded by the classification algorithm as only the magnitudes are used.

$$p_c(n,k) = \sqrt{\frac{2}{M}}\, h(n) cos \left[\left(n + \frac{M+1}{2} \right) \left(k + \frac{1}{2} \right) \frac{\pi}{M} \right] \tag{3.3}$$

$$p_s(n,k) = \sqrt{\frac{2}{M}}\, h(n) sin \left[\left(n + \frac{M+1}{2} \right) \left(k + \frac{1}{2} \right) \frac{\pi}{M} \right] \tag{3.4}$$

where $h(n)$ is the windowing function given as

$$h(n) = -sin \left[\left(n + \frac{1}{2} \right) \frac{\pi}{2M} \right] \tag{3.5}$$

Above, $p_c(n,k)$ produces the cosine components of the transformation, and are taken directly from the MLT, and $p_s(n,k)$ produce the sine components. The MCLT coefficients are then produced via recombination of the output components from the two basis function calculations, which will produce complex coefficients of length M when the input signal is length $2M$.

In [9], the author provides an efficient algorithm for the "Fast" MCLT (FMCLT) calculation, and provides implementation examples in the form of Matlab code for both the FMCLT and the inverse FMCLT algorithms. The MCLT algorithm was chosen for these experiments based on similar preprocessing for audio fingerprinting done by Burges *et. al.* in the DDA algorithm[3], and the author's implementation examples in [9] were used to calculate the MCLTs for the frames of data used in the algorithms described below. An example of an MCLT applied to an input audio clip is shown in figure [11].

After the application of the MCLT on the 4096 sample sized input frame of data, 2048 complex coefficients are returned. The absolute value is taken from these to

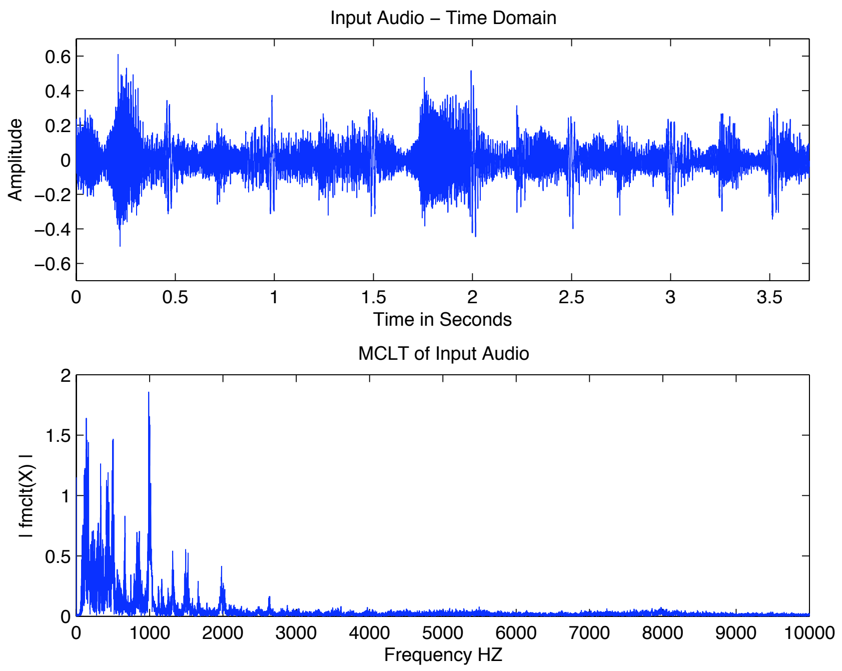

Figure 11: Comparison between input and MCLT of an audio signal

produce the magnitude[†], and the log is taken of the absolute coefficients. These frame coefficients are then fed into the de-equalization thresholding block.

3.2.4 De-equalization Thresholding

In the de-equalization thresholding block, the frequency spectrum of the various frames is again preprocessed in order to remove any distortions due to frequency or volume adjustment in the original signal. The inspiration for this approach comes from [3].

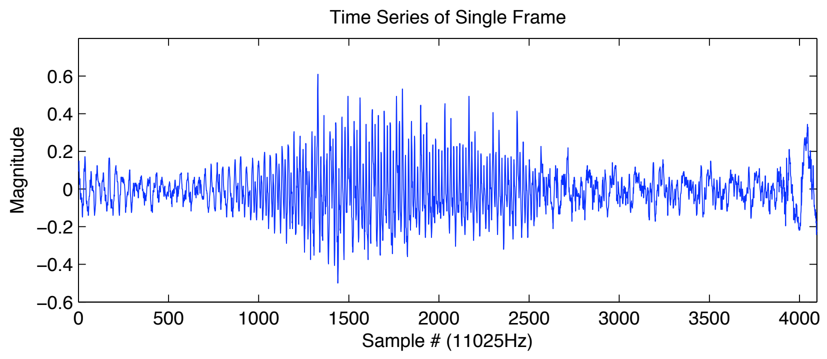

Figure 12: Time series representation of a single frame

Because audio signals, especially those sampled from the environment, can vary in their equalization profile, it is desirable to remove such variations for the purposes of classification. This step can be thought of as a type of “normalization” technique that is applied to the frequency domain. The idea here is to remove any equalization trends from a sample test frame by essentially applying a low pass filter to the log spectrum of the frame.

In this step, each frame of data is again operated on individually. Coming from the transformation block, each frame is now the log modulus of the computed

[†] *The complex modulus is calculated as* $|z| = \sqrt{x^2 + y^2}$ *where* z *is a complex number of form* $z = x + iy$

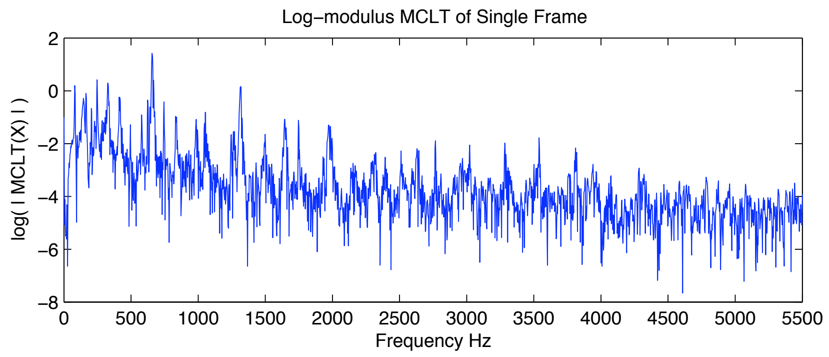

Figure 13: Log spectrum of frame from MCLT

MCLT coefficients. A Discrete Cosine Transformation (DCT) is applied to the log-modulus spectrum, producing 2048 DCT coefficients.

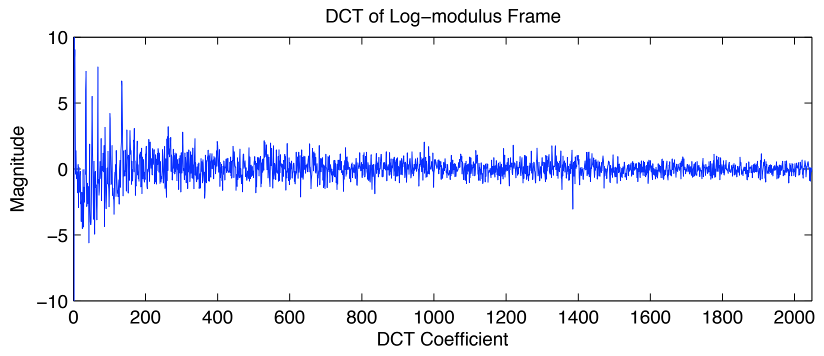

Figure 14: DCT applied to MCLT frame

Of these 2048 coefficients, the first 6 are scaled linearly from 1 to 0, with the remaining 2042 all set to 0, and stored in a vector, A. An inverse DCT is calculated from A, resulting in an approximation to the curve of the log spectrum data. This approximation is due to the "capacitive" effect of the low pass filter on the high frequency changes in the log spectrum.

Finally, a component wise difference between the log spectrum data and the approximation curve is calculated, producing a level, de-equalized, log spectrum frame.

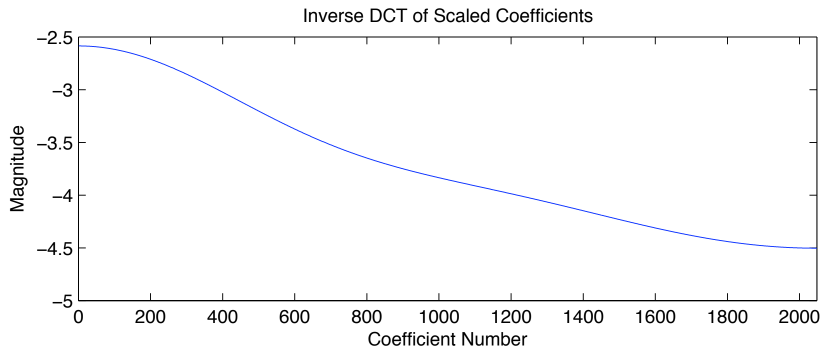

Figure 15: Approximation curve from inverse DCT of scaled coefficients

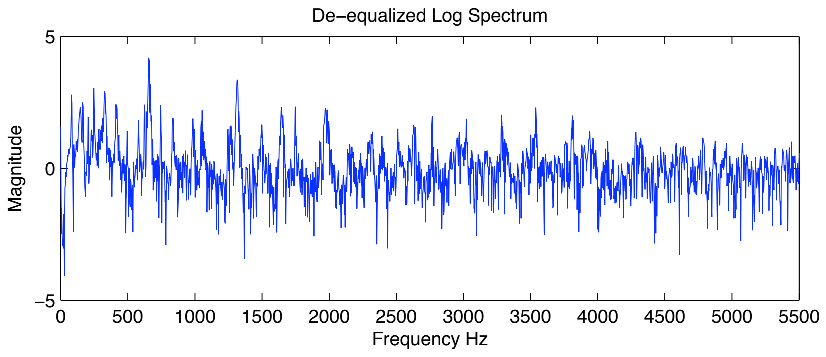

Figure 16: Final de-equalized log spectrum frame

This process is done for each distorted frame of the same frame dimension, which are then organized into an $M \times N$ matrix where M is the processed coefficients of a frame, and N is the number of distorted frames. This frame matrix is then passed to the PCA block.

3.2.5 PCA

In order to reduce the dimensionality of the feature space produced from the MCLT algorithm applied to the audio frames in the previous section, principal component analysis (PCA) is used. PCA is a data analysis technique that can be used in order to extract the "most important" features from a large data set. There are few different algorithms for calculating the principal components of a data set, however the one used here in these experiments is based on eigenvalue decomposition.

The traditional approach to calculating the principal components of some data set involves calculating the eigenvectors from the covariance matrix of the mean subtracted data set. So, if the data set X is organized as an $M \times N$ matrix, where M is the number of features (dimensions) organized in rows and N is the number of samples organized in columns, then the mean vector u can be calculated from each dimension as

$$u_m = \frac{1}{N} \sum_{n=1}^{N} X_{mn} \tag{3.6}$$

The mean subtracted data set is then generated as $B = X - u$ and the covariance matrix of the mean subtracted set is calculated by

$$C = \frac{1}{N-1} B B^{\tau} \tag{3.7}$$

where B^{τ} represents the transpose of B.

Finally, the eigenvectors V and eigenvalues λ (represented as a diagonal matrix), are calculated directly from the covariance matrix by solving the generalized eigenvector problem for

$$C \cdot V = \lambda V \tag{3.8}$$

This is a computationally expensive operation, and is typically performed via a data analysis software package. Following this step, the projection of the original data back onto the eigenvectors produces the "principal components" of the data set, with M number of principal components.

One problem with this algorithm when applied to the audio frames used in these experiments, however, lies in the size of the covariance matrix. Because the feature vector used is 2048 dimensions, the covariance matrix, when computed as above using the generalized eigenvector algorithm, is a 2048 $\times$ 2048 matrix. Computing the eigenvectors and eigenvalues of this large a matrix is very computationally expensive, and much too slow to be used practically for these audio classification experiments.

Instead, a modified algorithm, proposed by Turk and Pentland[2] in their paper presenting their "eigenface" approach to facial recognition and classification is substituted. In this algorithm, the fact that computationally infeasible data sizes for eigenvalue decomposition are used is compensated for by restricting the algorithm to only calculate the N top eigenvectors (those with the highest corresponding eigenvalue), from the given $M \times N$ matrix. This results in an $N \times N$ covariance matrix (as opposed to $M \times M$), and also in a much less expensive algorithm at the cost of loosing $M - N$ principal components. Note that it is thus important to have a sufficient number of data samples (N) for a particular application to ensure that enough meaningful principal components are generated for classification.

In the eigenface algorithm, the input data are the pixel values from pre-segmented and aligned facial images, composed of a many different face samples. The algorithm is given by taking a set of M centered training faces, and representing them in a $N \times M$ matrix where each column is one training image represented in vector form as $\Gamma_1, \Gamma_2, \Gamma_3, ...\Gamma_M$[†]. From this data matrix, the mean column vector Ψ is calculated by

$$\Psi = \frac{1}{M} \sum_{n=1}^{M} \Gamma_n \tag{3.9}$$

and the mean subtracted column vectors, Φ_n, is given by

$$\Phi_i = \Gamma_i - \Psi \tag{3.10}$$

The set of these mean subtracted column vectors is defined by A where $A = [\Phi_1 \ \Phi_2 \ ... \ \Phi_M]$ and can be used to calculate the covariance matrix of the mean subtracted vectors by

[†] Note that in the Turk and Pentland algorithm, the symbols for the matrix dimensions are swapped ($N \times M$ vs. $M \times N$). M in the eigenface algorithm corresponds to the number of faces, while N in the general eigenvector algorithm corresponds to the number of data samples used.

$$C = \frac{1}{M} \sum_{n=1}^{M} \Phi_n \Phi_n^T \tag{3.11}$$

$$C = AA^T \tag{3.12}$$

This results in a covariance matrix size of $N^2 \times N^2$. Instead, the covariance matrix of size $M \times M$ is computed by

$$C = A^T A \tag{3.13}$$

from which the eigenvectors are then computed. This results in only calculating the M best eigenvectors (those with the largest eigenvalues) and significantly reduces the computations necessary.

The eigenvectors u_i that are calculated from this covariance matrix thus make up the M-dimensional basis vectors of the feature space. The original mean subtracted data matrix A is then projected back onto the eigenvectors in order to determine the set of weights by

$$\Omega = u_i^T A \tag{3.14}$$

where $\Omega_i = [w_1\, w_1 \ldots w_M]$ and represents the weight vector associated with each training sample's contribution to the corresponding eigenvector dimension. These "weights" are the reduced principal components for the input data.

For each $2048 \times N$ frame of data, these eigenvalues are computed using the number of distorted frames as the N sample dimensions, and the principal components of the normalized eigenvectors are calculated. The eigenvectors, mean feature vectors, and projected weight vectors are stored in a database for this frame along with an ID signifying the song that this clip was generated from. The generation of this database is considered the "training phase" of the algorithm, and following completion the database is then used in the testing phase to identify an unknown clip.

3.3 Testing

Thus far, each of the processing blocks discussed apply to the training portion of the algorithm. Upon completion of the training phase, there exists a database containing the feature space representation of each frame, as well as the principal components of the projected frames which are used to test against.

In the testing phase, some unknown audio clip is submitted to the system for classification. This test clip is then passed through the same preprocessing, framing, transformation, and de-equalization thresholding blocks as in the training phase. Following these steps, there exists a collection of overlapping test frames, each transformed into MCLT magnitude coefficients, de-equalized, and normalized.

3.3.1 Calculating Closest Training Frame

Next, this collection of test frames is compared to a song in the database in an attempt to find a match. The test frame features Γ_{test}, organized as a column vector, are mean subtracted using the training frame's mean features Ψ. These are then projected onto the training eigenvectors u_{train} in order to determine the testing weights Ω_{test}

$$\Omega_{test} = u_{train}^T(\Gamma_{test} - \Psi_{train}) \tag{3.15}$$

where Ω_{test} is the testing frame's weight in terms of the contribution from the distorted training frame's eigenvectors.

Now that the testing weight Ω_{test} is known (via calculation), and training weights Ω_{train} are known (from training), the min Euclidean distance between Ω_{test} and each of the M weights in Ω_{train} is calculated by

$$\epsilon_{min} = min(\|\Omega_{test} - \Omega_{train}\|^2) \tag{3.16}$$

where ϵ_{min} is the minimum distance between the test weight and each of the training weights.

This process of calculating the minimum Euclidean distance between a training frame and a testing frame is performed for each of the training frames for a given song. The minimum distance is retained, and at the end of this calculation it is known which training frame number had the closest distance to this testing frame. The algorithm then goes through each of the remaining testing frames incrementally, each time calculating the training frame number with the minimum distance for this frame.

3.3.2 Classification

After each closest frame is calculated, the offset position in the training frame database for the closest frame is compared with the relative offset position of the test frame. If this position lines up linearly with one of the previously calculated closest frames, a counter is incremented indicating that these two frames are equidistant apart in both training and test, and thus aligned. If a training frame offset is calculated that is less than the last training frame offset, the counter is reset to zero, and the algorithm continues testing the frames.

Once three aligned matches are found, effectively showing a strong linear alignment between the testing and training clip, the classifier returns the ID for this clip as a match. If three aligned matches are not found, the classifier returns that this training song did not match the testing clip, and the algorithm continues looking through the other clips in the database for a strong match.

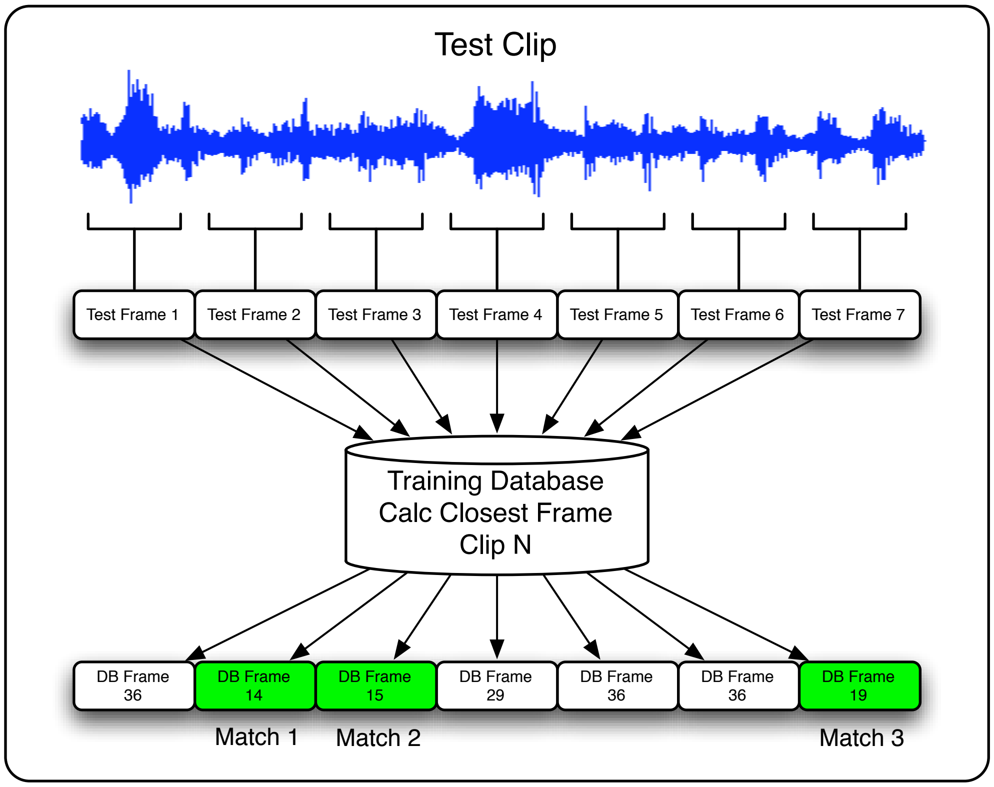

Figure 17: Example of positively classifying a test clip

Thus, a positive clip identification occurs when three testing frames show a strong correlation to three training frames that all have the same relative offsets between them. The algorithm continues to go through the database of training songs, testing each one until it receives a positive match. If, at the end of this procedure, the algorithm cannot find a positive match for a test clip, the classifier marks this song as “unknown”.

Note here that there is no thresholding used, apart from use of three consecutive aligned matches indicating a positive audio identification. When testing against an incorrect song, each testing frame still has a minimum distance value associated with it with respect to the training song. This technique relies on the proper *matching of alignments* of test frames with training frames for identification. Because it is highly unlikely that a series of testing clip frames will align linearly with a different audio clip’s training frames, the risk of false positives is greatly reduced.

3.3.3 Alignment Variation

One of the strengths of this approach to classification lies in the fact that the algorithm can identify not only when it has received a strong match, but also when it failed to produce any match. In this case, other techniques can be used in order to determine if the test clip does in fact have a training match, or if this test clip is simply not part of the database.

One scenario that could lead to the case of missing a matching database song might occur if the testing frame is severely misaligned with the database frames from training. This is a common occurrence in implementation, as the framing of a test clip taken from the environment (for instance, when recorded from a live source) is completely asynchronous to the framing procedure that took place during training - there’s no guarantee when the test clip will start relative to training.

However, because this system is efficient in determining if no strong match exists, such alignment variations can be compensated for by slightly altering the start of

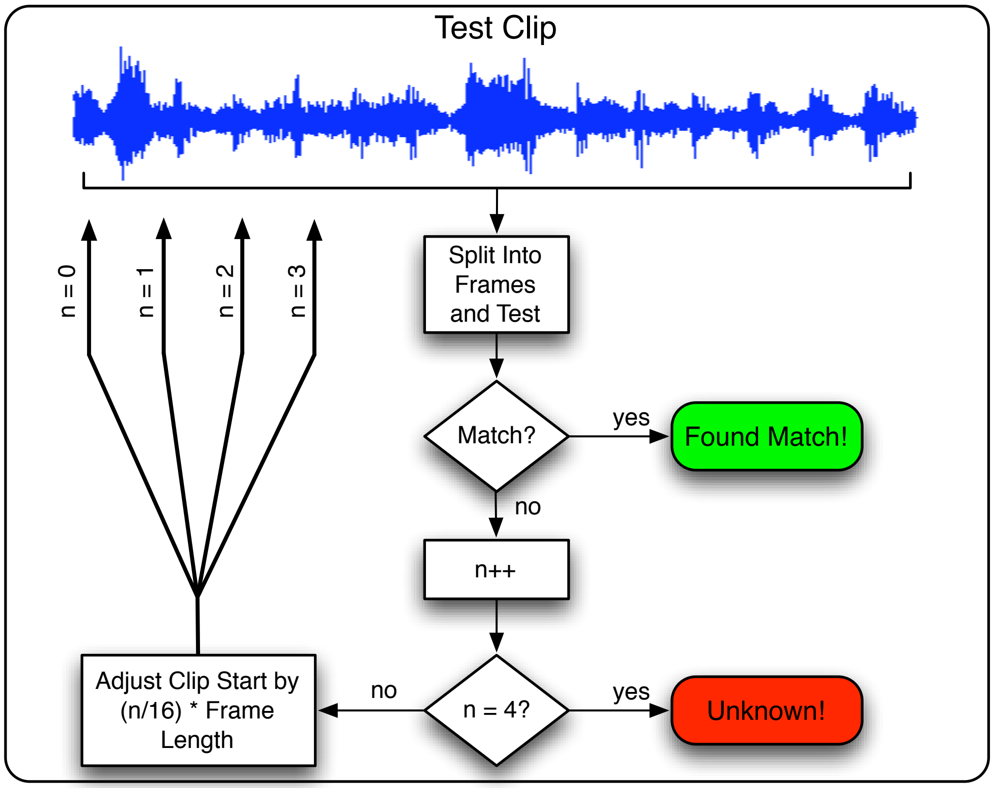

Figure 18: Flowchart for alignment compensation algorithm

the test clip's position and retesting. Because frames are generated every 4096 samples from the input signal, and because the frames are overlapped by $1/2$ frame length, the worst case scenario for a test frame misalignment occurs when the test frame is $1/4$ frame out of phase with training.

To compensate for this, when the classifier returns no strong match for a given test clip, the test clip is simply resampled with the starting position increased by $1/16$ frame length, and retested. This process continues, with the algorithm increasing the start frame position by $1/16$ frame length until it either finds a match, or it reaches the worst case alignment of $1/4$ frame length. If the algorithm does not find a strong match after shifting the alignment by $1/4$ frame, then this test clip is finally classified as "unknown", and no further testing is performed.

4 Results

Below, a number of different experiments are performed in order to test the various aspects of the fingerprinting system. For each of these experiments, a database of 50 songs is used spanning many different musical genres including jazz, blues, rock, heavy metal and hip-hop[†]. The training database used in these experiments is generated from the 9 distortions discussed in 3.2.1, except where otherwise noted.

4.0.4 Experiment #1

In this experiment, a single testing frame from each song is tested against the entire database. The testing frames used are a distorted version of the song generated via Audition using a distortion that was not present in training. For each given test frame, the distance from that frame is calculated for each training frame in the entire database, and the resulting closest frame found is used as the best guess. This test is performed using perfectly aligned testing frames, slightly misaligned (5%) frames, and severely misaligned frames (100%, or 1/4 frame size).

Table 1: Results of single frame test

	Aligned	Slight Misalignment	Severe Misalignment
Correctly Guessed	40	40	35
Incorrectly Guessed	10	10	15
Unknown	N/A	N/A	N/A
Accuracy	**80%**	**80%**	**70%**

[†] A complete list of the songs used is given in appendix table A.2

From this experiment, it is observed that the simple method of testing frames individually from the entire database can itself yield reasonable results. It can also be seen that this method is resistant to slight alignment variations, as the experiment with just a 5% alignment variation has the same resulting accuracy as the perfectly aligned test. For the severely misaligned frame however, the accuracy is negatively affected as expected.

It is worth noting that in these experiments, because the entire database needs to be read in order to calculate the closest frame, the run time of this algorithm increases exponentially as the size of the database grows. Also, it is not possible using this method to test a single song from the database for a match, as the frame's distance is is relative to the entire training database.

Another problem with this method for classification lies in the algorithm's inability to distinguish between when there exists a match in the database and when the song is not present in the database. Because this method tests the distance between each training frame and uses the closest frame as a guess with no thresholding, any time that a test is performed with a clip that is not part of the training database, the algorithm's guess will be incorrect.

4.0.5 Experiment #2

In this experiment, the same training database as in experiment #1 is used, however 3 frames of a test clip are used to test against. For each frame, the closest song from the training database is calculated, as in experiment # 1, and the song with the most number of matches is given as the best guess. This test is again performed using perfectly aligned testing frames, slightly misaligned (5%) frames, and severely misaligned frames (100%, or 1/4 frame size).

In this experiment, the overall accuracy is increased for each test with respect to the single frame experiment. This accuracy however, comes at the cost of lookup

Table 2: Results of 3 frame test

	Aligned	Slight Misalignment	Severe Misalignment
Correctly Guessed	43	43	38
Incorrectly Guessed	7	7	12
Unknown	N/A	N/A	N/A
Accuracy	**86%**	**86%**	**76%**

speed as for each test fingerprint, 3 different frames must be tested.

While it can be seen that using more training frames can improve accuracy, because this approach needs to search through the entire database for each testing frame, the algorithm still scales poorly to large data sets. This technique also still has the same drawbacks as in experiment # 1, including the inability to distinguish when a fingerprint is not in the database, however it is 3x slower.

4.0.6 Experiment #3

In this experiment, the algorithm that is described in section 3 is used, with the exception that no compensation for alignment variations is performed here. The testing clip used for each song is again distorted using a distortion that is not present in training. The test fingerprint used for each song is 6 seconds in duration, and taken from a point in the middle of the song. This test is again performed using perfectly aligned testing frames, slightly misaligned (5%) frames, and severely misaligned frames (100%, or 1/4 frame size).

Table 3: Full test on untrained distortion - no alignment compensation

	Aligned	Slight Misalignment	Severe Misalignment
Correctly Guessed	50	50	41
Incorrectly Guessed	0	0	0
Unknown	0	0	9
Accuracy	**100%**	**100%**	**82%**

In this experiment, the new classifier algorithm which was developed for this project is used to test the fingerprints. It can be seen here that this algorithm produces much more accurate results than that used in the prior experiments. Further, because this algorithm is capable of recognizing weak matches (as described in 3.3.2), there are no false positives, and instead the algorithm can classify unrecognized clips as “unknown”.

In even the severely misaligned tests, the algorithm outperforms the previous experiments, with each incorrect match being marked accordingly as opposed to producing a false positive. Again, no alignment variation compensation was performed here, these results are the raw values, returned by the first pass of the algorithm.

4.0.7 Experiment #4

In this experiment, the same testing and training data from experiment #3 is again used, however this time misalignment compensation (as discussed in section 3.3.3) is additionally performed. This test is again performed using perfectly aligned testing frames, slightly misaligned (5%) frames, and severely misaligned frames (100%, or 1/4 frame size).

Table 4: Full test on untrained distortion using alignment compensation

	Aligned	Slight Misalignment	Severe Misalignment
Correctly Guessed	50	50	50
Incorrectly Guessed	0	0	0
Unknown	0	0	0
Accuracy	**100%**	**100%**	**100%**

In this experiment, the effectiveness of the alignment variation compensation portion of the algorithm is demonstrated. Each of the unknown test clips from the previous experiment in the severely misaligned test are correctly recognized. In these cases, the algorithm correctly recognized when it did not have a strong match, and proceeded to shift the input test clip appropriately, in increments of 1/16 frame size samples, until a strong match was encountered. In this test, all of the fingerprints are correctly identified.

4.0.8 Experiment #5

In this experiment, live recordings of the various songs are used in order to test the algorithm's ability to classify the audio in an actual live environment. In order to perform this experiment, Matlab scripts were generated to iterate through the collection of testing songs, play a 10 second sample from a location near the center of the song using the system's speakers, and record the sample using the system's built-in microphone[†]. In this test, the database used was trained with all 9 distortions, and alignment variation enabled.

Table 5: Full algorithm on live recorded songs

Correctly Guessed	46
Incorrectly Guessed	0
Unknown	4
Accuracy	**92%**

This experiment shows that the algorithm had an accuracy of 92% when tested from live recorded samples of the various test clips. Due to the live nature of the recordings, the test clips were by default likely to be misaligned with training, and thus requiring the use of the alignment variation technique discussed in 3.3.3.

In the test, it was found that 40 test clips were found without any alignment variation needed, 6 clips required alignment variation compensation to find a match, and 4 were unrecognized and are thus considered false negatives. There were no

[†] This experiment was run on an Apple Macbook Pro. Full specifications for the test system are given in 5.2

instances of false positives in this experiment.

Some of the error in this experiment, leading to the generation of the 4 false negative cases, can be accounted for by considering the types of distortions used. The 9 distortions used during the training phase of the algorithm are meant to to maximize the variance of the feature space such that the algorithm is robust to tests on audio that is distorted in ways not present during training. Ideally, these training distortions should be similar to the types of distortions that the algorithm is expected to experience in a live environment.

Because the training distortions were generated by hand using the Adobe Audition tool suite, it is likely that the filters used were not entirely representative of the types of distortion found in this experiment from the live environment. Because the audio was being recorded live, as it passes from transducer to transducer (in this case from speakers to microphone), each stage adds its own distortion in the form of noise, including the ambient environmental background noise and room acoustics.

In sections 5.3.1 and 5.3.3, different enhancements are proposed in order to improve the classification results in live environmental testing. These include generating mathematical models of the types of distortions used in training, as well as a further investigation into which distortions maximize the classification accuracy.

In general however, this experiment shows that using a simple assortment of training distortions is still effective in providing the discriminatory power necessary to achieve reasonable results. When tested from the live environment, representing a completely different class of distortion from training, the algorithm still had the ability to correctly identify 92% of the test clips, and produce no false positives.

4.0.9 Experiment #6

In this experiment, the same training database is used as well as the same untrained distorted test songs, however in this test more fingerprints are generated for each input song. For each song in the database of 50 trained songs, a 3 second test fingerprint is generated every 3 seconds for a full minute of audio, resulting in 20 test fingerprints per song and 1000 fingerprints total. Because this 3 second mark is not aligned on a frame boundary, this experiment again tests the algorithm's robustness to alignment variations.

Table 6: Full test on multiple fingerprints per song

Correctly Guessed	943
Incorrectly Guessed	0
Unknown	57
Accuracy	**94.3%**

It can be seen here that the algorithm, when tested with every possible fingerprint combination, is reasonable accurate, classifying correctly 94.3% of the time and producing no false positives. It is important to note here that the size of the fingerprints used in this experiment are 3 seconds in duration as opposed to the 6 second fingerprints used in experiments #3 & #4[†]

For the unrecognized clips, it was found that these mostly appear in dynamically "sparse" regions of songs, where few notes are being played and there are large temporal gaps in the audio. These were found to typically match to other dynamically sparse regions from the training songs. This, combined with the fact that only 3 seconds of audio is used to test with, help to explain where these inaccuracies come from. In typical fingerprinting systems, fingerprints are used that are longer in duration, typically 6-10 seconds.

[†] This is due to the large size of the training database, and the desire to extract as many fingerprints from the test clip as possible while keeping them large enough to provide reasonable classification results.

4.0.10 Experiment #7

In this experiment, the algorithm's robustness to false positives is tested. Because the algorithm stops testing once a match has been identified, it is possible for it to not test every possible combination of test fingerprint to training song. In this test, 3 second fingerprints are again generated every 3 seconds from the test song and classified against the database. Here however, each test clip is only tested against *other* songs in the database - i.e. the song is explicitly not tested against the matching song in training. In this way, every possible combination of test fingerprint to different training song is tested.

Table 7: False positive testing results

True Negative	995
False positive	5
Accuracy	**99.5%**

These results show that the algorithm is indeed robust to false positives. This is due to the fact that it is highly unlikely that 3 frames from both testing and training will exhibit linear alignment with each other. Because this test was excluding the correct fingerprint from the training database for each classification operation, the desired outcome for each test was an "Unknown" clip (or a True Negative), signifying that the algorithm could not find the test clip in the database. Any instance of an incorrectly guessed clip indicates a false positive.

In the case where false positives were detected, it was again observed that the regions of both the testing and training songs that were matched were dynamically sparse with little variance. This can again be partially attributed to the fact that only 3 seconds of test audio were used for the fingerprint as clips with a longer duration would be expected to have more variance. The algorithm would likely see improved resistance to these cases of false positives if further restrictions were used on the test side to test the clip for an appropriate amount of musical variation prior to classification.

5 Discussion

5.1 Results Analysis

The algorithm developed here is shown to be capable of correctly identifying various types of song clips from a stored database of trained songs. The algorithm is robust to classification errors when tested with audio clips that are distorted in ways that are not present in training, including those types of distortions that come from live recordings in an environmental setting. It is also highly resistant to false positive misclassifications.

One interesting feature of this algorithm which factors into it's classification strength is the fact that it not only uses the perceptual audio features from frames for recognition, but also uses these frames relationship to each other in time to aid in accuracy. In this way, the algorithm really operates in a 2 dimensional space on the input data - using the PCA projections to classify frames of test data to training data, and using time offsets to relate these frames to each other. When compared to other algorithms which use PCA for classification, such as the Eigenface algorithm, the notion of using this extra dimension to improve the results is unique[†].

Another feature of this algorithm is the fact that it lends itself nicely to parallelization techniques if used in a real-world implementation. Because each song can be classified independently to the rest of the database, this algorithm is well suited to distributed computing environments as well as high performance computing techniques. Distributed systems, symmetric multiprocessors, and general purpose CPUs (GPCPUs) including GPUs could all be utilized to improve the speed and accuracy of the algorithm without any fundamental redesign of the algorithm's classifier.

[†] While images are indeed 2 dimensional, in PCA-based image recognition algorithms images are reduced to single dimensional column vectors for classification, where each pixel is used as a feature.

The speed of the algorithm is reasonably fast and capable of determining both matches and unknown test clips quickly. Because the classifier works on songs from the database one at a time, the speed with which it classifies a single song is not affected by the size of the database. As the database grows, the run time scales linearly as each new song is added. In the worst case scenario, where an unknown fingerprint is tested against a song from the database and no match is found, the algorithm takes less than 2 seconds to read the training data from disk into memory and test the entire song. Using the Matlab profiler, it was observed that a significant portion of this time was spent reading the song from disk. Some possible further optimizations addressing this issue are discussed in 5.3.1.

There were two different types of errors in classification: *false positives* - where the algorithm incorrectly guessed a song, and *false negatives*, where the algorithm could not correctly identify a song from the database when it was present. While the false positive tests obtained generally good results (0.5% per clip), this number could still likely be reduced further via the use of musical variance pre-testing and longer duration fingerprints, as well as the use of some further preprocessing techniques (discussed in 5.3.4).

In the case of false negatives, the algorithm showed a 94.3% accuracy. This means that 5.7% of the time, the algorithm could not correctly identify a given test song when the corresponding training song existed in the database. In the majority of cases, it was observed that this occurred in areas of the audio that were dynamically sparse and lacked musical variance. This issue could again be largely mitigated with the addition of some preprocessing testing which “screens” fingerprints prior to classification to ensure that they are a good candidate. Also, the use of some of the other preprocessing techniques proposed in 5.3.4 could also decrease the frequency of this issue. Given the scope of these experiments, these numbers are still reasonable, and show this approach’s promise as a robust audio fingerprinting system.

5.2 Implementation Notes

For this project, the algorithms were implemented in Matlab, and all experiments were run on on a Macbook Pro with a 2.16GHz Intel Core Duo processor and 2GB of memory. Python scripts were used in combination with the Unix tool `afconvert` in order to generate the down-sampled raw audio files for training. The Adobe Audition tool was used to generate the distorted versions of the raw training songs, which were then processed in Matlab in order to build the database. The Matlab scripts *macwavplay.m* & *macwavrecord.m* were developed in order to both play and record audio from the environment, as the Matlab commands `wavplay` & `wavrecord` are not implemented for the Mac OS X version of Matlab.

Many optimizations were performed in order to improve the overall performance of the algorithm in both training and testing phase, and also to reduce the database size. It was found that the explicit use of single precision numbers not only reduced the data sizes by half, but also improved the speed of classification. This small change sped up the lookup time by almost a factor of 2.

Another optimization performed was the method by which the weight distance calculations between the testing weight vector and the training weight matrices were calculated. As opposed to looping over each individual column and calculating the distance, it was observed to be much faster in practice to produce an equivalently sized matrix from the test weight vector and to perform all of the distance calculations in a single matrix operation. Matlab's `repmat` command was found to be slow at the type of vector manipulation necessary for this experiment, so a further increase in performance was achieved by again using matrix manipulations to multiply the test vectors with logical matrices of appropriate size in order to produce the necessary test matrices.

Another further increase in performance was produced in this step by abandoning the use of the Matlab `norm` command to calculate the Euclidean distance. This command was found to be slow, partially due to the fact that there was no way run it on each mean subtracted column individually without using loops. Thus, matrix operations were again used in order to calculate the distances over each column in a single operation, and produce the minimum distance from the entire set. This optimization was responsible for an additional 2x increase in speed.

5.3 Further Enhancements

5.3.1 Database Size

One problem with the approach of training using predetermined distortions and retaining the eigenvectors of training clips for testing classification is the fact that this greatly increases the size of the database. For example, for a given frame of audio data containing 4096 samples, if 9 distortions are used to compute the principal components and then stored, the eigenvectors end up being a 2048 x 9 matrix. Because the 2048 x 1 mean subtracted feature vector is also retained, as well as the 9 x 9 projected weight matrix, this data ends up ballooning to over 10 times the size of the original frame. Since frames are calculated and stored at 1/2 frame size overlapping intervals, this ends up increasing the size by another factor of 2. Thus, for a 1 minute song clip sampled at 11025Hz, the size of the input song is approximately 2.5MB. The size of the training data generated from this song is approximately 25.5MB. For a database of just 50 of these 1 minute song clips, after training the size of the database is roughly 1.29GB.

With data sizes this big, it makes it difficult to efficiently process incoming test clips in a reasonable run time. Also, because the data sizes are so large, they must be stored on disk as there would typically not be enough room to store this data in memory - especially as the database size increases.

One optimization that would greatly reduce the size of the data would be to change the algorithm to not rely on retaining the eigenvectors from training for testing classification. The actual classification portion of the algorithm relies exclusively on measuring the Euclidean distance between the testing weight vector and the training weight vectors in order to find a minimum. The eigenvectors from training are only used for projection of the test frame onto the feature space of the training data in order to derive the testing weight in terms of the training frame's basis vectors for the distance measurement.

To see how this might be reduced, it needs to first be understood what the training eigenvectors actually represent, and how they are used in order to derive the test weights. Because the training principal components are calculated from the individual transformed feature coefficients across each of the distortions used, the feaatures are essentially "smeared" across this high dimensional space in order to allow for a higher discriminatory power during testing projection. This can intuitively be thought of as these distortions adding some slight amount of "fuzz" around each of the features in this high dimensional space. During test then, a test feature is projected onto this same space in order to produce the testing weights in terms of the training feature space components. In the case of a match, these test feature projections are ideally made closer to the training features because of the increase in feature size that the distortions add.

In these experiments, the distortions are modeled in Adobe Audition and applied to each song only in training. If instead, these various distortions were modeled mathematically, they could be applied not only to the training songs, but also to the testing songs as part of the preprocessing phase. The benefit of this approach would be that, in both training and test, there would be a mathematical way of determining the principal components of the associated frames, and the testing side would not have to rely on the training feature metadata (eigenvectors and mean separated feature vectors) in order to calculate the weights. In this way, the only data that would need to be retained for each frame would be the 9 x 9 calculated weight vectors. In the test phase then, the same process could be applied to the testing data, and a 9 x 9 matrix of test weights could be used to classify against training. If implemented successfully, this could reduce the size of the database by more than 250 times. A data reduction of this magnitude would reduce the training database size from about 25.5MB down to about 100kb for 1 minute of training audio.

One drawback of this approach would be that, because the testing weights would not be calculated in terms of the training basis vectors, it is likely that this would effect the accuracy of the distance calculations. There is a danger that by deriving these values independently, that the discriminatory power of the lookup operation could be affected such that it's ability to distinguish the testing clips from the training database could decrease. In order to determine the effects of this, and how the other stages of the algorithm would need to be adjusted to accommodate this change, further investigation is needed.

5.3.2 Database Speed

Another database enhancement to this algorithm would be the implementation of an actual relational database to store the training weights in. In the experiments performed here, the database used was simply a collection of data structures that were stored on disk and loaded from file as needed. With a proper database implementation, this process could be managed better and requests for data could be processed much quicker.

There would also likely be benefits from the investigation and experimentation into how the data in the database is organized and ultimately hashed. In [7], the authors discuss a technique for database organization to increase the query speed when dealing with high dimensional data. The technique involves the use of data redundancy in combination with bit vector indexing in order to reduce both the access time as well as the distance measurement operation time. In essence, advance techniques such as this move the distance calculation from being explicitly applied on the retrieved data to being automatically calculated as part of the hashing operation into the database.

In combination with the database size reduction techniques described above, the application of these database approaches could increase the speed and real-world feasibility of this fingerprinting system.

5.3.3 Distortions

The ability of this algorithm to classify songs that are likely distorted by the recording environment, comes from the use of canned distortions in the training phase to approximate the types of coloration that songs might posses during test. The list of distortions used were given in 3.2.1.

Some further investigation could be performed into characterizing the distortions in order to maximize their effectiveness in implementation. For example, various environmental recordings could be analyzed in order to determine mathematical models for the types of distortions that might typically be found in various real world testing environments. Another option could be the investigation into the use of different noise models or the application various probability density functions in the training phase in order to approximate the distortions necessary for the high dimensional discrimination needed for successful classification.

The use of the 9 different types of distortions used in these experiments show that reasonable results can be obtained from using these simple filters. These distortions, and the method of their application, could likely be analyzed and adjusted in order to maximize the classification accuracy - especially in the case of live environmental recordings.

5.3.4 Additional Preprocessing

In the algorithm developed here, the application of the de-equalization thresholding phase (described in 3.2.4), greatly increased the accuracy of the classifier. Additional preprocessing techniques could be examined to improve this even more.

In [6], the authors discuss the use of additional preprocessing techniques which they use to remove distortions from the audio frames that are inaudible to the human ear. This is by generating a frequency-dependent perceptual filter, and applying it to the de-equalization thresholding output data prior to PCA classification. Again, further investigation into these techniques could yield improved accuracy.

5.3.5 Lookup Classification Thresholding

In the experiments, the case of a false negative would occur when the testing phase could not find an existing matching song in the database. Some techniques to minimize the effects of this were proposed such as testing for some level of minimum musical variance prior to classification, as well as the use of longer duration fingerprints.

In the experiments, it was observed that in case of false negatives, there would often be curious results from the classifier such as numerous database matches for a given testing frame to the same training frame. In these cases, it is apparent that the algorithm is calculating erroneous values by the frequency with which it continues to match a test clip to the same frame in training. This case most frequently occurs in the areas of the song which are again very dynamically sparse and lack musical variance, and confuses the classifier by "spamming" the closest-frame calculations with multiple instances of the same training clip. When this occurs, it is difficult for the classifier to find 3 frames that are in linear alignment between training and test.

Investigation into the post-processing of the classifier would likely result in additional thresholding approaches to improve the accuracy. Simple techniques, such as restricting the algorithm's search from database frames which demonstrate this effect of continual appearance in the testing results, could greatly improve results. Again, further investigation is required in order to determine the feasibility of this approach.

5.3.6 Search Techniques

One key advantage of this algorithm lies in it's ability to determine a song's classification with respect to some training clip by only looking at that training clip itself as opposed to that clip in relation to other clips from the database. In terms of graph theory, this effectively results in the ability to perform a depth-first search (DFS) as opposed to a breadth-first search (BFS) for classification, and eases the requirements on the amount of data that must be loaded into memory in order to perform classification. If however, the data size were to be drastically reduced (via techniques such as those proposed in 5.3.1), much more training data would be able to be loaded into memory, and a hybrid approach of DFS and BFS could be utilized in order to both increase the accuracy of the lookup as well as the speed.

This would amount to the ability to search the entire database for the closest matching test clip as opposed to linearly progressing through each song individually. This could also greatly speed up the classification time of the algorithm, as it would be possible to effectively operate across the entire database as the same time.

One downside of this approach however, would be that the run time of the algorithm would no longer scale linearly with an increase in database size. It would require further investigation to determine if the benefits of the classification speed increase would outweigh the cost of the algorithm's ability to scale to larger data sets.

6 Conclusion

The development of effective algorithms for the recognition and classification of digital data is an important area of research in the domains of intelligent systems and computer vision. The goal of this research is the development of systems capable of advanced situational understanding, with the ability to process data from the environment in a manner similar to the processing capability of the human brain. There are various algorithms and techniques for this perceptual understanding of digital data that are used in the realm of computer vision for scene understanding, and their application to the domain of auditory data is both a natural and interesting extension.

In this project, a system for audio fingerprinting which uses a PCA-based approach for feature extraction and classification was developed, implemented, and tested. A number of different testing scenarios were investigated and analyzed, and the system was shown to have a high discriminatory ability. In experiments including live environmental recordings as well as worst-case monte carlo test of simulated distortions, the algorithm proved to be over 90% effective at identifying matches to a stored training database. This approach was also shown to be robust to alignment variations between training and test, as well as to the appearance of false positive classification errors. These errors were analyzed as well as some of the other system characteristics, and future work was proposed to address some of these needs.

References

[1] Pedro Cano, Eloi Batlle, Ton Kalker, Jaap Haitsma., *A Review of Audio Fingerprinting*, Journal of VLSI Signal Processing 41, pp. 271-284, 2005.

[2] Turk, M. A. and Pentland, A. P., *Face recognition Using Eigenfaces*. IEEE Computer Society Conference on Computer Vision and Pattern Recognition, 1991. pp. 586-591, 1991.

[3] Christopher J.C. Burges, John C. Platt and Soumya Jana., *Distortion Discriminant Analysis for Audio Fingerprinting* IEEE Transactions on Speech and Audio Processing Vol. 11, No. 3, pp. 165-174, 2003.

[4] Avery Li-Chun Wang., *An Industrial-Strength Audio Search Algorithm* Shazam Entertainment, Ltd., 2003.

[5] Avery Li-Chun Wang., *The Shazam Music Recognition Service* Communications Of The ACM vol. 49, 2006.

[6] Christopher J.C. Burges, John C. Platt and Soumya Jana., *Extracting Noise-Robust Features from Audio Data* Microsoft Research 2002.

[7] Jonathan Goldstein , John C. Platt and Christopher J. C. Burges ., *Indexing High Dimensional Rectangles for Fast Multimedia Identification* Microsoft Research 2003.

[8] Henrique Malvar., *A Modulated Complex Lapped Transform and its Applications to Audio Processing*, Microsoft Research, 1999.

[9] Henrique Malvar., *Fast Algorithm for the Modulated Complex Lapped Transform*, Microsoft Research, 2005.

A Appendix

A.1 Matlab Code

getSongFingerPrints.m

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% getSongFingerPrints(songs, frame_size) -
%
% This function calculates the fingerprints for the songs passed in. The
% songs are ordered as a matrix where the columns are song clips of various
% distortions. The frame sized passed in is the frame length that will be
% used, in overlapping sequences, to build each fingerprint.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function printStruct = getSongFingerPrints(songs, frame_size)

num_samps = size(songs, 1);
num_songs = size(songs, 2);

% Start frame at first non zero element
frame_start = find(songs, 1, 'first');

% Preallocate matrices
frame_matrix = zeros(frame_size/2, num_songs);

% Init first layer matrix
frame_cnt = 1;

% Go through song in increments of frames. Shift window by 1/2 frame for
% each iteration and use projections to build first layer feature matrix.
while (frame_start + frame_size - 1) ≤ num_samps

    % Calc frame end
    frame_end = frame_start + frame_size - 1;

    % Go through each song clip passed in
    for j = 1:num_songs

        % Calculate MCLT on this frame for this song
        frame_matrix(:,j) = fmclt(songs(frame_start:frame_end,j));
        frame_matrix(:,j) = log(abs(frame_matrix(:,j)));

        % Apply perceptual preprocessing and de-equalization
        % First, take DCT of log space of frame
        frame_matrix(:,j) = dct(frame_matrix(:,j));
        % Scale the first 6 coefficients from 1-0. Set the rest to 0.
        A = frame_matrix(:,j)' .* [linspace(1,0,6) zeros(1,(2048-6))];
        % Take the inverse DCT of the frame subracted from the coefficient
        % data. This is an approximation of the curve of the log space.
        frame_matrix(:,j) = idct(frame_matrix(:,j) - A');
        % Normalize
        frame_matrix(:,j) = frame_matrix(:,j)/norm(frame_matrix(:,j));
    end

    % Run PCA on this collection of clips
    if (sum(isinf(frame_matrix(:))) == 0)

        [eigvecs, eigvals, mean_feats, weights] = runPCA(frame_matrix);

        % Build final fingerprint matrix. Each num_songs number of values
            will
        % be the fingerprint for a song. Build an array of structures, one
            for
        % each fingerprint, and store the song ID in the struct. This should
        % make it easier navigate later.
        printStruct(frame_cnt).eigvecs = eigvecs;
        printStruct(frame_cnt).mean_feats = mean_feats;
        printStruct(frame_cnt).weights = weights;
        frame_cnt = frame_cnt + 1;
    end

    frame_start = frame_start + frame_size/2;
end
```

runPCA.m

```
% This function runs the Eigenface PCA algorithm on the clips passed in.
function [eigvecs, eigvals, mean_feats, weights] = runPCA(clips)

num_samps = size(clips, 1);
num_songs = size(clips, 2);

% Normalize
for i = 1:num_songs
    mu = mean(clips(:,i));
    stdev = std(clips(:,i));
    clips(:,i) = (clips(:,i) - mu)/stdev;
end

% Calculate the mean feature values
mean_feats = mean(clips,2);

% Subtract mean features from clips
clips = clips - repmat(mean_feats, 1, num_songs);

% Calculate clip covariance using the Turk and Pentland trick
clip_cov = clips'*clips;

% Get the eigenvectors in the proper decending order
[eigvecs, eigvals] = eig(clip_cov);
eigvecs = fliplr(eigvecs);

% Get the feature space projection (other Turk and Pentland trick)
eigvecs = clips * eigvecs;

% Normalize eigenvectors
for i = 1:num_songs
    eigvecs(:,i) = eigvecs(:,i)/(norm(eigvecs(:,i)));
end

% Get the weights by projecting clips back over eigenvectors
weights = eigvecs'*clips;

% Set everything to single precision
eigvecs = single(eigvecs);
mean_feats = single(mean_feats);
weights = single(weights);
```

recordSongsEnv.m

```

% This script plays and records songs live from the system it is run on.
% This script is used in order to aquire environmental data to test with.

% Record live samples of each song
num_songs = 50;
song_path = '/Users/zak/School Stuff/Independent Study/audio_data/
    ordered_orig/';
out_path = '/Users/zak/School Stuff/Independent Study/audio_data/
    live_50_third/';
```

```

length = 10;
start_time = 30;
Fs = 11025;

listing = dir(sprintf('%s*.mp3',song_path));

for i = 1:num_songs
    disp(sprintf('Shhhh! Recording song %i!',i));

    % Play this song as a backround process
    disp(sprintf('\tIssuing play command'));
    command = sprintf('afplay "%s%i.mp3" -s %i %i &',song_path,i,start_time,
        start_time+length+1);
    system(command);

    % Give some time for the system to play (airfoil)
    pause(1)

    % Record this song
    disp(sprintf('\tStarting to record'));
    clip = macwavrecord(length*Fs,Fs);
    disp(sprintf('\tStoping recording'));

    % Store this song to disk
    out_file = sprintf('%sliveclip%i.wav',out_path,i);
    wavwrite(clip, Fs, out_file);

    % Give some time for the system to stop
    pause(1)
end
```

macwavplay.m

```

% This function plays an input wav signal on a Mac
function macwavplay(y, Fs)

% Save input wav to a temporary file
wavwrite(y, Fs, 'tmp.wav');

% Play using afplay
[status, result] = system(['afplay ', 'tmp.wav']);

```

```
% Remove the temp file
%system('rm -f tmp.wav');

return
```

macwavrecord.m

```
% This script records audio data on a Mac
function data = macwavrecord(n, Fs)

% Open a handle to audio device
r = audiorecorder(Fs,16,1);

% Start recording
record(r);

% Wait for n/Fs samples + some wiggle room
pause((n/Fs) + 0.5);

% Stop recording
stop(r);

% Get the recorded data in wav form
tmp = getaudiodata(r, 'double');

% Truncate data to exact desired output size, and fill output
data = tmp(1:n);
```

fimclt.m

```
function y = fimclt(X)

% FIMCLT - Compute IMCLT of a vector via double-length FFT
%
% H. Malvar, September 2001  --  (c) 1998-2001 Microsoft Corp.
%
% Syntax:  y = fimclt(X)
%
% Input:   X : complex-valued MCLT coefficients, M subbands
%
% Output:  y : real-valued output vector of length 2*M

% in Matlab, by default j = sqrt(-1)

% NOTE - Taken from 'Fast Algorithm for the Modulated Complex Lapped
% Transform' by Henrique S. Malvar, January 2005

% determine # of subbands, M
M = length(X);

% allocate vector Y
Y = zeros(2*M,1);

% compute modulation function
k = [1:M-1]';
c = W(8,2*k+1) .* W(4*M,k);

% map X into Y
Y(2:M) = (1/4) * conj(c) .* (X(1:M-1) - j * X(2:M));

% determine first and last Y values
Y(1)   =   sqrt(1/8) * (real(X(1)) + imag(X(1)));
Y(M+1) = - sqrt(1/8) * (real(X(M)) + imag(X(M)));

% complete vector Y via conjugate symmetry property for the
% FFT of a real vector (not needed if the inverse FFT
% routine is a "real FFT", which should take only as input
% only M+1 coefficients)
Y(M+2:2*M) = conj(Y(M:-1:2));

% inverse normalized FFT to compute the output vector
% output of ifft should have zero imaginary part; but
% by calling real(.) we remove the small rounding noise
% that's present in the imaginary part
y = real(ifft(sqrt(2*M) * Y));

return;


% Local function: complex exponential

function w = W(M,r)

w = exp(-j*2*pi*r/M);

return;
```

fmclt.m

```
function X = fmclt(x)

% FMCLT - Compute MCLT of a vector via double-length FFT
%
% H. Malvar, September 2001  --  (c) 1998-2001 Microsoft Corp.
%
% Syntax:  X = fmclt(x)
%
% Input:   x : real-valued input vector of length 2*M
%
% Output:  X : complex-valued MCLT coefficients, M subbands

% in Matlab, by default j = sqrt(-1)

% NOTE - Taken from 'Fast Algorithm for the Modulated Complex Lapped
% Transform' by Henrique S. Malvar, January 2005

% determine # of subbands, M
L = length(x);
M = L/2;

% normalized FFT of input
U = sqrt(1/(2*M)) * fft(x);

% compute modulation function
k = [0:M]';
c = W(8,2*k+1) .* W(4*M,k);

% modulate U into V
V = c .* U(1:M+1);

% compute MCLT coefficients
X = j * V(1:M) + V(2:M+1);

return;


% Local function: complex exponential

function w = W(M,r)

w = exp(-j*2*pi*r/M);

return;
```

test_multiple_prints_per_song.m

```

% This script breaks the input test clips up into multiple fingerprints in
% order to test a larger number of cases.

% System constants
num_songs = 50;
frame_size = 4096;
samps_per_sec = 11025;
train_start_sec = 5;
train_end_sec = 65;
num_prints_per_song = 20;
print_length = (train_end_sec-train_start_sec)/num_prints_per_song;
total_right = 0;
total_wrong = 0;
total_unknown = 0;
maxcorrectdist = 0;
minwrongdist = 99999;

% Go through the songs
for i = 1:num_songs

    % Test against the genhighpass distortion
    song_name = [int2str(i) '_genhighpass'];

    % Read the song in
    [test_song, Fs, nbits] = wavread(song_name);

    % Go through the fingerprints in the song
    for j = 0:num_prints_per_song - 1

        % Calc print start and end times
        print_start = (train_start_sec + (j * print_length)) * samps_per_sec;
        print_end = (print_start + (print_length * samps_per_sec));

        % Try to find a match - full algorithm
        test_clip = test_song(print_start:print_end);
        [matched_song, cnts]  = findClosestClipFull(db_path, test_clip,
            frame_size,i);

        % Test for misalignement
        if (cnts ≠ 3)

            % Initially save this as the best guess
            best_match = matched_song;
            best_cnts = cnts;

            disp(sprintf('Clip %i returned weak guess (%i counts, songID = %s), shifting and retesting...', i,cnts, matched_song));

            for h = (frame_size/16):(frame_size/16):(frame_size/4)

                disp(sprintf('\tShifting by %i and retesting...',h));
                [matched_song, cnts]  = findClosestClipFull(db_path, test_clip(h:end), frame_size,i);

                if (cnts == 3)
                    disp(sprintf('\tFound a strong answer (3 counts, songID = %s)!',matched_song));

                    break;
                elseif (cnts == 2)

                    % If this is the second time we've guessed this song with 2
                    % counts, consider this the "guess" and stop retrying
                    if ((strcmp(matched_song,best_match) == 1) && (best_cnts == 2))
                        disp(sprintf('\tFound a second instance of the same weak answer. Using this as our guess.'))
                        break;
                    end

                    disp(sprintf('\tFound a weak answer (2 counts, songID = %s). Save and keep trying...',matched_song));
                    best_match = matched_song;
                    best_cnts = cnts;
                end

                if (h == (frame_size/4))
                    matched_song = best_match;
                    cnts = best_cnts;
                end
            end
        end

        disp(sprintf('Song with max hits for song %i (clip %i) is: %s (%i counts)', i, j, matched_song, cnts))

        if (strcmp(matched_song,'N/A') == 1)
            total_unknown = total_unknown + 1;
        elseif ((str2num(strrep(matched_song,'song','')) == i))
```

```
            total_right = total_right + 1;
        else
            total_wrong = total_wrong + 1;
        end
    end
end

disp(sprintf('\nTotal results: %i correct guesses, %i incorrect guesses, %i
    unknown, %f%% accuracy (correct vs. incorrect)', total_right, total_wrong
    , total_unknown, ((total_right)/(total_right+total_wrong))*100));
```

build_database.m

```

% This script reads in each of the songs from the test set, grouped by
% distortion, and builds the database of fingerprints. The structure
% generated here is used for fingerprint lookup and classification.

% This script builds the training database by reading in the various songs,
% generating the song fingerprints, and storing the fingerprint structures
% to file.

% Define system constants
num_songs = 50;
num_dists = 9;
frame_size = 4096;

% Database paths
%db_path = '/Users/zak/School Stuff/MSProject/audio_data/db/
    traindb_full_9dists.mat';
%db_path = '/Users/zak/School Stuff/MSProject/audio_data/db/traindb_small.mat
    ';
%db_path = '/Users/zak/School Stuff/MSProject/audio_data/db/
    traindb_small_dist.mat';
db_path = '/Users/zak/School Stuff/MSProject/audio_data/db/
    traindb_full_8dists.mat';

% Distortion names
dist_names(1).string = 'compander';
dist_names(2).string = 'deesser';
dist_names(3).string = 'expander';
dist_names(4).string = 'hisscutr';
dist_names(5).string = 'notch';
dist_names(6).string = 'slowdrums';
dist_names(7).string = 'splineboost';
dist_names(8).string = 'oldradio';
dist_names(9).string = 'genhighpass';

% Start and end positions in clips to train
start_pos = 5*11025;
end_pos = start_pos + 65*11025;

% Check if we're overwriting the database - learned this the hard way
if (exist(db_path) == 2)

    warn = input('Database already exists! Do you want to overwrite? Y/N [N]:
         ', 's');
    if isempty(warn)
        warn = 'N';
    end
    if (strcmpi(warn, 'Y')) ≠ 1
        disp(sprintf('Aborting script!'));
        return;
    end
end

% Create the database - store creation time
dbdate = fix(clock);
save(db_path, 'dbdate');
save(db_path, 'num_songs', '-append');
save(db_path, 'num_dists', '-append');
save(db_path, 'frame_size', '-append');

% Go through each song
for i = 1:num_songs

    disp(sprintf('Reading in song %i',i));

    train_clip = [];

    for j = 1:num_dists

        % Get string of distorted song name
        song_name = [int2str(i) '_' dist_names(j).string];

        % Read in song
        [song, Fs, nbits] = wavread(song_name);

        % Cat distorted versions of the clips together for PCA
        train_clip = [train_clip song(start_pos:end_pos)];
    end

    disp(sprintf('Training song %i',i));
    songStruct = getSongFingerPrints(train_clip, frame_size);
    songID = ['song' int2str(i)];
    eval(sprintf('%s = songStruct;',songID));

    % Store song in db on disk
    save(db_path, songID, '-append');
    clear('songStruct');
    clear(songID);
end
```

run_tests.m

```

% This script tests a fingerprint from each song against the training
% database in an attempt to find a match. This algorithm accounts for
% alignment variations by shifting weak matches by increments of 1/16 frame
% sizes and retesing in order to try to find a better match.

% System constants
num_songs = 50;
frame_size = 4096;
total_right = 0;
total_wrong = 0;
total_unknown = 0;
maxcorrectdist = 0;
minwrongdist = 99999;

% Song clip positions
%start_pos = (33*11025);
%start_pos = (5*11025)+(4096*4);
%start_pos = (88*4096); % about 33 seconds in, aligned
start_pos = 1;
end_pos = start_pos + (10*11025) - 1;

% Go through each song
for i = 1:num_songs

    % To test aginst a given distortion
    %song_name = [int2str(i) '_genhighpass'];

    % To test the original wav file
    %song_name = [int2str(i)];

    % To test the live clips
    song_name = sprintf('liveclip%i.wav',i);

    % Read the song in
    [test_song, Fs, nbits] = wavread(song_name);

    % Grab a test clip from the song
    test_song = test_song(start_pos:end_pos);

    % Try to find a match - full algorithm
    [matched_song, cnts]  = findClosestClipFull(db_path, test_song,
        frame_size,i);

    % Try to find a match - using single frame
    %test_song = test_song(1:frame_size);
    %[matched_song, cnts]  = findClosestClipSingle(db_path, test_song,
        frame_size);

    % Try to find a match - using 3 frames
    %test_song = test_song(1:frame_size*3);
    %[matched_song, cnts]  = findClosestClipThree(db_path, test_song,
        frame_size);

    % If we get either an "N/A" (no clear guess), or a count of 2 (weakly
    % confident guess), begin by shifting the input audio by increments of
    % quarter frame sizes and re-evaluate to attempt to find a better,
    % strong (3 count) answer. At the end, if we still haven't come up with
    % a confident guess, take the best one that we have.
    if (cnts ≠ 3)

        % Initially save this as the best guess
        best_match = matched_song;
        best_cnts = cnts;

        disp(sprintf('Clip %i returned weak guess (%i counts, songID = %s),
            shifting and retesting...', i,cnts, matched_song));

        for j = (frame_size/16):(frame_size/16):(frame_size/4)

            disp(sprintf('\tShifting by %i and retesting...',j));
            [matched_song, cnts]  = findClosestClipFull(db_path, test_song(j:
                end), frame_size,i);

            if (cnts == 3)
```

```
                disp(sprintf('\tFound a strong answer (3 counts, songID = %s)!',matched_song));

                break;
            elseif (cnts == 2)

                % If this is the second time we've guessed this song with 2
                % counts, consider this the "guess" and stop retrying
                if ((strcmp(matched_song,best_match) == 1) && (best_cnts == 2))
                    disp(sprintf('\tFound a second instance of the same weak answer. Using this as our guess.'))
                    break;
                end

                disp(sprintf('\tFound a weak answer (2 counts, songID = %s). Save and keep trying...',matched_song));
                best_match = matched_song;
                best_cnts = cnts;
            end

            if (j == (frame_size/4))
                % If we get here, and we haven't gotten either a strong or
                % two matching weak answers. Look at the best guess from
                % the shifting excercise, and if it has 2 counts, use that
                % as the guess. If not ("N/A"), then consider this clip unknown.
                matched_song = best_match;
                cnts = best_cnts;
            end
        end
    end

    disp(sprintf('Song with max hits for clip %i is: %s (%i counts)', i, matched_song, cnts))

    if (strcmp(matched_song,'N/A') == 1)
        total_unknown = total_unknown + 1;
    elseif ((str2num(strrep(matched_song,'song','')) == i))
        total_right = total_right + 1;
    else
        total_wrong = total_wrong + 1;
    end
end

disp(sprintf('\nTotal results: %i correct guesses, %i incorrect guesses, %i
    unknown, %f%% accuracy (correct vs. incorrect)', total_right, total_wrong
    , total_unknown, ((total_right)/(total_right+total_wrong))*100));
```

findClosestClipFull.m

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% findClosestClipFull(printDB, test_clip, frame_size, songIdx) -
%
%   This function finds the closesst clip in the input fingerprint database
%   to the given test clip passed in. The sondIdx parameter is used to
%   specify the training song to test against, and is used for debugging
%   onle.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function [matched_song, cnts] = findClosestClipFull(printDB, test_clip,
    frame_size, songIdx)

% Get number of songs in database
load(printDB, 'num_songs');
for i = 1:num_songs

    song_name = sprintf('song%i',i);
    %song_name = sprintf('song%i',songIdx);

    result = testPrintAgainstSong(printDB, test_clip, song_name);

    if result
        matched_song = song_name;
        cnts = 3;
        return;
    end
end

% If we get here, we didn't find a match
matched_song = 'N/A';
cnts = 1;
return;

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% testPrintAgainstSong(printDB, print, song_name) -
%
%   This function loads the given training song from the database and tests
%   it against the given input print.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function result = testPrintAgainstSong(printDB, print, song_name)
```

```

% Load song from database
disp(sprintf('Loading song: %s from database',song_name))
load(printDB, song_name);
load(printDB, 'frame_size');

% Split the print up into frames
test_offset = 1;
aligned_cnts = 0;
frame_start = find(print, 1, 'first');
num_samps = size(print,1);
while (frame_start + frame_size - 1) ≤ num_samps

    % Calc frame end
    frame_end = frame_start + frame_size - 1;

    % Get the closest frame
    train_offset = getOffsetOfClosestTrainFrame(print(frame_start:frame_end),
         eval(song_name));

    if test_offset ≠ 1

        % If we go backwards, reset count
        if (train_offset < last_train_offset)
            aligned_cnts = 0;
        % Check if this is the next linear clip
        elseif (train_offset - last_train_offset == 1)
            aligned_cnts = aligned_cnts + 1;
            if (aligned_cnts == 3)
                result = 1;
                return;
            end
        end
    end

    % Update offset information
    test_offset = test_offset + 1;
    last_train_offset = train_offset;
    frame_start = frame_start + frame_size/2;
end

result = 0;
return;

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% getOffsetOfClosestTrainFrame(frame, loaded_song) -
```

```
%
%   This function returns the offset of the closest training frame for the
%   given test frame and training song.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function train_offset = getOffsetOfClosestTrainFrame(frame, loaded_song)

% Calculate MCLT on this test clip
test_clip = fmclt(frame);
test_clip = log(abs(test_clip));

% Apply perceptual preprocessing and de-equalization
test_clip = dct(test_clip);
A = test_clip' .* [linspace(1,0,6) zeros(1,(2048-6))];
test_clip = idct(test_clip - A');
test_clip = test_clip/norm(test_clip);

% Normalize
mu = mean(test_clip(:));
stdev = std(test_clip(:));
test_clip = (test_clip - mu)/stdev;

% Get the number of fingerprints in the DB for this song
num_prints = size(loaded_song,2);

% Get the number of song samples (distortions) to test against
num_dists = size(loaded_song(1).weights,2);

% Go through each print
min_dist = 9999999;
for printnum = 1:num_prints

    % Extract the training data from this clip
    train_mean_feats = loaded_song(printnum).mean_feats;
    train_eigvecs = loaded_song(printnum).eigvecs;
    train_weights = loaded_song(printnum).weights;

    % Get mean subtracted test features for this print
    test_sub_mean = test_clip(:) - train_mean_feats;

    % Get weight of test clip by projecting onto eigenspace
    test_weight = train_eigvecs'*test_sub_mean;

    % Calculate the min Euclidean distance between in one operation.
    % This is pretty neat actually - no repmat and no loops! This
    % resulted in a 100% speedup.
    test_min = min(sqrt(sum((test_weight(:, ones(num_dists,1)) - ...
                        train_weights).^2)));
    if ( min_dist > test_min)
        min_dist = test_min;
        train_offset = printnum;
    end
end
```

findClosestClipSingle.m

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% findClosestClipSingle(printDB, test_clip, frame_size, songIdx) -
%
%   This function finds the song in the training database with
%   the closest frame to the test frame.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function [matched_song, cnts] = findClosestClipSingle(printDB, test_clip,
    frame_size)

if (size(test_clip,1) ≠ frame_size)
    disp(sprintf('Error! Test clip size not equal to frame size!'));
    disp(sprintf('findClosestClipSingle() needs to be used on single frames!'
        ));
    matched_song = 'N/A';
    cnts = 0;
    return;
end

% Init
total_min_dist = 999999;
matched_song = 'N/A';
cnts = 3;

% Get number of songs in database
load(printDB, 'num_songs');
for i = 1:num_songs

    song_name = sprintf('song%i',i);

    % Load song from database
    disp(sprintf('Loading song: %s from database',song_name))
    load(printDB, song_name);
    load(printDB, 'frame_size');

    % Get closest distance to test frame
    [tmp, tmp_dist] = getOffsetOfClosestTrainFrame(test_clip, eval(song_name)
        );

    % Store the total min distance from all songs
    if (tmp_dist < total_min_dist)
        total_min_dist = tmp_dist;
        matched_song = song_name;
    end
end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% getOffsetOfClosestTrainFrame(frame, loaded_song) -
%
%   This function returns the offset of the closest training
%   frame for the given test frame and training song
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function [train_offset, min_dist] = getOffsetOfClosestTrainFrame(frame,
    loaded_song)

% Calculate MCLT on this test clip
test_clip = fmclt(frame);
test_clip = log(abs(test_clip));

% Apply perceptual preprocessing and de-equalization
test_clip = dct(test_clip);
A = test_clip' .* [linspace(1,0,6) zeros(1,(2048-6))];
test_clip = idct(test_clip - A');
test_clip = test_clip/norm(test_clip);

% Normalize
mu = mean(test_clip(:));
stdev = std(test_clip(:));
test_clip = (test_clip - mu)/stdev;

% Get the number of fingerprints in the DB for this song
num_prints = size(loaded_song,2);

% Get the number of song samples (distortions) to test against
num_dists = size(loaded_song(1).weights,2);

% Go through each print
min_dist = 9999999;
for printnum = 1:num_prints

    % Extract the training data from this clip
    train_mean_feats = loaded_song(printnum).mean_feats;
    train_eigvecs = loaded_song(printnum).eigvecs;
    train_weights = loaded_song(printnum).weights;

    % Get mean subtracted test features for this print
    test_sub_mean = test_clip(:) - train_mean_feats;

    % Get weight of test clip by projecting onto eigenspace
    test_weight = train_eigvecs'*test_sub_mean;

    % Calculate the min Euclidean distance between in one operation.
    % This is pretty neat actually - no repmat and no loops! This
    % resulted in a 100% speedup.
    test_min = min(sqrt(sum((test_weight(:, ones(num_dists,1)) - ...
                    train_weights).^2)));
    if ( min_dist > test_min)
        min_dist = test_min;
        train_offset = printnum;
    end
end
```

findClosestClipThree.m

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% findClosestClipThree(printDB, test_clip, frame_size, songIdx) -
%
%   This function finds the song in the training database with
%   the most closest frames to the three test frames
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function [matched_song, cnts] = findClosestClipThree(printDB, test_clip,
    frame_size)

if (size(test_clip,1) ≠ frame_size*3)
    disp(sprintf('Error! Test clip size not equal to 3 frames!'));
    disp(sprintf('findClosestClipSingle() needs to be used on three frames!')
        );
    matched_song = 'N/A';
    cnts = 0;
    return;
end

% Get number of songs in database
load(printDB, 'num_songs');

% Init matches struct
```

```matlab
for i = 1:num_songs
    song_name = sprintf('song%i',i);
    matches.(song_name) = 0;
end

% Go through each song for each frame
for j = 0:2

    % Init
    total_min_dist = 999999;
    matched_song = 'N/A';

    for i = 1:num_songs

        song_name = sprintf('song%i',i);

        % Load song from database
        disp(sprintf('Loading song: %s from database',song_name))
        load(printDB, song_name);
        load(printDB, 'frame_size');

        % Get closest distance to test frame
        frame_start = 1+((j*frame_size)/2);
        frame_end = frame_start + frame_size - 1;
        [tmp, tmp_dist] = getOffsetOfClosestTrainFrame(test_clip(frame_start:
            frame_end), eval(song_name));

        % Store the total min distance from all songs
        if (tmp_dist < total_min_dist)
            total_min_dist = tmp_dist;
            matched_song = song_name;
        end
    end

    % Store closest match count
    matches.(matched_song) = matches.(matched_song) + 1;
end

% Return frame with the most "hits"
cnts = 0;
for i = 1:num_songs
    if (matches.(sprintf('song%i',i)) > cnts)
        cnts = matches.(sprintf('song%i',i));
        matched_song = sprintf('song%i',i);
    end
end
```

```

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% getOffsetOfClosestTrainFrame(frame, loaded_song) -
%
%   This function returns the offset of the closest training
%   frame for the given test frame and training song
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function [train_offset, min_dist] = getOffsetOfClosestTrainFrame(frame,
    loaded_song)

% Calculate MCLT on this test clip
test_clip = fmclt(frame);
test_clip = log(abs(test_clip));

% Apply perceptual preprocessing and de-equalization
test_clip = dct(test_clip);
A = test_clip' .* [linspace(1,0,6) zeros(1,(2048-6))];
test_clip = idct(test_clip - A');
test_clip = test_clip/norm(test_clip);

% Normalize
mu = mean(test_clip(:));
stdev = std(test_clip(:));
test_clip = (test_clip - mu)/stdev;

% Get the number of fingerprints in the DB for this song
num_prints = size(loaded_song,2);

% Get the number of song samples (distortions) to test against
num_dists = size(loaded_song(1).weights,2);

% Go through each print
min_dist = 9999999;
for printnum = 1:num_prints

    % Extract the training data from this clip
    train_mean_feats = loaded_song(printnum).mean_feats;
    train_eigvecs = loaded_song(printnum).eigvecs;
    train_weights = loaded_song(printnum).weights;

    % Get mean subtracted test features for this print
    test_sub_mean = test_clip(:) - train_mean_feats;

    % Get weight of test clip by projecting onto eigenspace
    test_weight = train_eigvecs'*test_sub_mean;

    % Calculate the min Euclidean distance between in one operation.
    % This is pretty neat actually - no repmat and no loops! This
    % resulted in a 100% speedup.
    test_min = min(sqrt(sum((test_weight(:, ones(num_dists,1)) - ...
                    train_weights).^2)));
    if ( min_dist > test_min)
        min_dist = test_min;
        train_offset = printnum;
    end
end
```

A.2 Song List Used in Experiments

Artist	Album	Track
Beirut	Lon Gisland EP	Elephant Gun
Bon Iver	For Emma, Forever Ago	Flume
The Very Best	The Very Best Mixtape	Kamphopo
Neil Diamond	The Best of Neil Diamond	Sweet Caroline
Why?	Alopecia	The Vowels Pt. 2
Black Sabbath	Paranoid	War Pigs
Dr. Dre	The Chronic	F*** Wit Dre Day
Paul Simon	Graceland	Graceland
Brian Wilson	Smile	Heroes And Villains
Every Time I Die	Last Night in Town	Jimmy Tango's Method
Manowar	Kings of Metal	Kings of Metal
Girls	Album	Laura
Aretha Franklin	Sparkle	Something He Can Feel
The Octopus Project	Hello, Avalanche	Truck
Grizzly Bear	Veckatimest	Two Weeks
Har Mar Superstar	The Handler	DUI
Neutral Milk Hotel	In The Aeroplane Over The Sea	In The Aeroplane Over The Sea
Baroness	Blue Record	Jake Leg
David Bowie	Let's Dance	Let's Dance
John Coltrane	The Very Best of John Coltrane [Rhino]	Naima

Continued on next page

Artist	Album	Track
Tragedy	Can We Call This Life	The Ending Fight
Nirvana	Incesticide	Been A Son
Daft Punk	Homework	Da Funk
Iron Maiden	Powerslave	Flash of the Blade
The Band	Greatest Hits	I Shall Be Released
Michel Petrucciani	So What (Best Of)	J'aurais Tellement Voulu
Hot Chip	The Warning	Over And Over
Stevie Wonder	Greatest Hits	Sir Duke
The National	Boxer	Squalor Victoria
Animal Collective	Merriweather Post Pavilion	Summertime Clothes
Band Of Horses	Everything All The Time	The Funeral
Kanye West	Graduation	Good Life
The Flaming Lips	Yoshimi Battles the Pink Robots	In the Morning of the Magician
Jay-Z	American Gangster	Roc Boys (And The Winner Is)..
Fujiya & Miyagi	Lightbulbs	Sore Thumb
YACHT	See Mystery Lights	Summer Song
Thelonious Monk	The Best of the Blue Note Years	'Round Midnight
Talking Heads	Stop Making Sense	Life During Wartime
The Black Keys	Magic Potion	Modern Times
Buddy Guy	Damn Right, I've Got the Blues	Too Broke to Spend the Night
M. Ward	Post-War	Rollercoster
The Clash	London Calling	Death Or Glory
Marvin Gaye	Command Performances-15 Greatest Hits	Let's Get It On
Robert Johnson	King Of The Delta Blues Singers	Hell Hound On My Trail
Operation Ivy	Energy	Here We Go Again
Air	Premiers Symptomes	Casanova 70
Beck	Midnite Vultures	Get Real Paid
Crossexamination	Demo	The Foodening
Ween	Quebec	It's Gonna Be A Long Night