NTNU
Innovation and Creativity

Musical descriptors
An assessment of psychoacoustical models in the presence of lossy compression

Steinar Heimdal Gunderson

Master of Science in Communication Technology
Submission date: May 2007
Supervisor: Jan Tro, IET

Norwegian University of Science and Technology
Department of Electronics and Telecommunications

Problem Description

Due to the huge amount of net based searchable musical data the availability of reliable fingerprint information – musical descriptors – in digitized music signals is essential for recognition and identification. In order to decrease the need for data storage and transmission capacity several dedicated data reduction methods based on psychoacoustical models have been developed and are frequently in use.

The aim of this study is to explore and evaluate the influence of variable data reduction and compression methods regarding reliability and robustness of musical descriptors.

Assignment given: 28. April 2007
Supervisor: Jan Tro, IET

Abstract

A simple system for recognizing music is presented, based on various *musical descriptors*, numbers that describe some aspect of the music. Various descriptors are discussed; in particular, a novel descriptor, the *floor-1 cepstral coefficient* (F1CC) measure, a refinement of MFCCs based on the Vorbis psychoacoustical model is presented and evaluated. Also, various forms of statistical dimensionality reduction, among them PCA and LDA, are considered in the present context. Finally, a few directions for future work are discussed.

Acknowledgments

First of all, I would like to thank my advisor Jan Tro, who patiently provided feedback and guidance over the course of the entire semester. However, several other people have played important roles: Greg Maxwell originally proposed the idea that eventually led to the development of F1CCs, and Chris Montgomery provided helpful guidance on the internals of the Vorbis encoder. Mette Langaas helped with various insights on statistics, in particular dimensionality reduction. Håvard Midtkil provided his entire music collection in FLAC format as data material, saving countless hours of ripping labor. Finally, Rune Holm and Magne Mæhre proofread the manuscript at various stages, providing invaluable feedback, corrections and suggestions.

Contents

Chapter 1

Introduction

1.1 Music Information Retrieval

As ever more data is produced globally, the need to find and extract useful information from it has become increasingly apparent. The primary interest has traditionally been centered around that of textual search, in more recent times that of the information found on the World Wide Web (WWW).

However, while textual search has become a mature field, it is by definition limited to only a certain part of the world's available information. Thus, there is increased interest in extending information retrieval into non-textual areas. *Music Information Retrieval* (or *Multimedia* Information Retrieval), often abbreviated *MIR*, is a field of study dedicated to search and classification in music data – frequently, its audio representation, but also in MIDI information[1] or metadata.

In MIR research, many different applications have been discussed, including:

- Genre classification, where music is automatically classified into different genres.
- Neighbor search, similar to genre classification, where similar music is grouped together. (This can help answer questions such as "if I like song X, Y and Z, what other music might I like?".)
- Music identification, in which a piece of music is identified from its (possibly distorted) audio representation. This is not only useful to automatically correct or add metadata, but also to automatically track the use of music in radio, in shorter movie clips or as played as part of a longer mix.
- Automated transcription, deducing note and instrument information from a monophonic or polyphonic audio signal, possibly for further analysis.
- Various forms of musical queries from derived forms, for instance "query by humming" or "query by beatboxing", which attempt to locate a specific piece of music from a human query only partially related to the original signal.

[1] "MIDI information" in this context means music represented as notes with given tone heights and durations, instead of as a waveform. MIDI itself is a digital music communications protocol, standardized in 1983[36] and today almost universally supported by electronic music equipment.

In this project, the primary use case is that of *personal music identification.* A user wants to match songs from his/her music library (in the form of digitally compressed music files) against a central library in order to update the files with correct metadata. To this end, a set of *descriptors* (also called *features*) – numbers that describe some aspect of the music – is extracted from each song, and compared against the descriptors stored in the library in order to search for a match. High-quality descriptors are essential to the overall performance of a music identification system – in particular, a descriptor should be robust against the distortion arising from use of lossy compression such as MP3.

1.2 Aim of study

The aim of this paper is to explore the feature space and assess the quality of common descriptors in a musical setting, as well as consider various forms of data refinement. In addition, a novel descriptor, *F1CC*, is introduced, based on the Vorbis psychoacoustic model. F1CC better models the *auditory masking* present in the human ear than existing descriptors, increasing the overall robustness in the presence of lossy encoding.

1.3 Structure

The remaining text is intended to be read in order, and is structured as follows:

- In chapter 2, the use of descriptors is further motivated, and a formal notation is laid down. Also, a few simple descriptors are described.
- In chapter 3, *mel frequency cepstral coefficients* (MFCCs), a common spectral measure tuned towards imitating the human auditory system, is introduced. In chapter 4, *floor-1 cepstral coefficients* (F1CCs), a novel refinement of MFCCs based on the Vorbis psychoacoustical model, is introduced.
- In chapter 5, a simple test scenario is described, and descriptor assessment is discussed. In chapter 6, results from various tests comparing different sets of descriptors are presented.
- Finally, in chapter 7 the results from chapter 6 are discussed, followed by more general considerations regarding descriptor use and processing. Finally, in chapter 8, results are summarized, and some ideas for further work are outlined.

1.4 Previous work

Several MIR systems capable of music recognition already exist – however, the performance varies somewhat, and most are commercial systems with little or no published documentation. However, there are open and semi-open systems available, as well as some systems that are partially documented through papers or presentations.

One of the oldest and most widely used music recognition systems is Relatable's *TRM* (short for *TRM Recognizes Music*)[33], in particular due to its use in the *MusicBrainz* open content music encyclopedia. TRM has been criticized for being too simplistic[34], with problems of *collisions* (two or more distinct tracks being regarded as equal by the algorithm), eventually leading to MusicBrainz migrating towards MusicIP's *PUID/OFA*[31] (Open Fingerprint Architecture) system instead. Yet, TRM delivers relatively good results for a rather simple system.

Several other systems exist, both open and closed, for various different use cases. A particularly interesting system, *Shazam*[42], is designed to be highly robust, in that it is capable of recognizing a track from 20 seconds of audio over the telephone, even with severe distortion (such as high levels of ambient noise), by analyzing spectral peaks over time. A similar system, dubbed *MusicID*, has recently been licensed by Sony Ericsson, and is as of 2007 being introduced in current-generation consumer music mobile phones.

Finally, *Tuneprint* should be mentioned – even though the company behind the system no longer exists, it was relatively recently documented in a research paper.[35] Many of the ideas employed in the design of Tuneprint were new for its time, and it did show that music recognition by audio fingerprinting was indeed possible on a larger scale.

Chapter 2

Audio descriptors

2.1 Motivation

Just as with other kinds of information (such as text), there is a growing need for categorization, recognition and searching in audio, in particular music (a field often denoted *Music Information Retrieval*, or *MIR*). A typical three-minute song, however, contains a large amount of information (about eight million samples, or in excess of 30MB in the industry-standard 44.1kHz 16-bit PCM format used in audio CDs), making it impractical to even transfer or store it uncompressed, much less perform searches in a large music catalog.

Even discounting storage and processing problems, there is a large amount of *redundancy* in the unprocessed data, which is undesirable in many classes of learning and searching algorithms. Thus, even accounting for ever-increasing storage and processing power, there is a need for a more compact representation than what is used for regular audio playback.

An *audio descriptor*, or sometimes, an "audio fingerprint", is a reduced-dimensionality version of an audio signal – in essence, a set of numbers describing some aspect of the audio in question. Most (but not all) audio descriptors are designed to extract useful information from the audio as heard by a human listener, not the specific bits and bytes in a given signal.

Even the best descriptor is not very useful on its own – it is merely a convenient representation for storage, searching and comparing against other descriptors. In every MIR system, some sort of processing of the finished descriptors is required after the extraction stage, and as with descriptors, systems with various levels of sophistication have been proposed. A simple back-end system, primarily designed to compare the qualities of various descriptor configurations, is described in chapter 5.

2.2 Formal description

Formally, an audio descriptor is an N-dimensional vector generated from an input vector (consisting of the original audio waveform) of length M, where usually $N \ll M$. (M need not be constant, but usually is, and will in general be treated as such in the rest of the text.) Mathematically:

$$\vec{y} = f(\vec{x})$$

where $f : \mathcal{R}^M \rightarrow \mathcal{R}^N$ is the descriptor function. Often, the notion of an input sequence $\{x_i\}_0^{M-1}$ will be used in place of a vector; the two notations are equivalent and will be used interchangeably.

2.3 Desired properties

The parameters of design for a good audio descriptor[1] depend on the application in question. In this text, the discussed use-case is that of *music recognition*: A complete song (or at the very least, the beginning of it) is available in digital format, and the user wants to search for the song in some sort of musical library. Notably, this use case implies that the tracks in the reference library and the track to be searched for are approximately aligned in time, and as such, one does not need to consider extreme time shift.

The actual methods used to extract the descriptors from the audio data vary widely. Descriptors can be time-based (ie., work directly on the waveform representation), spectrally based (ie., work on a frequency representation) or a combination. (Also, there are approaches, such as wavelet-based analysis, that do not fit well into either category.) From section 2.6 onwards, descriptors from each category will be presented, and as will be seen, they differ significantly in both basic workings, complexity, robustness and performance.

However, almost all audio descriptors share the property that they should somehow be *well-correlated to human hearing* – in other words, pieces that are perceived as musically "the same" should have audio descriptors that are close[2], while distinct pieces should be spaced relatively far apart.[3] In some applications, such as *neighbor search*, it is also desired that distinct pieces of the same artist or in the same genre are closer to each other than to other pieces.

2.3.1 Dimensionality

N, the dimensionality of the descriptor, can be as low as 1 (making it a *scalar descriptor*), but usually, it will be higher. As the dimensionality increases, the volume of the descriptor (hyper-)space increases, ideally increasing the distance between different music pieces' descriptor points. However, adding more dimensions will not help if there is significant correlation between the vector elements – the end effect will be more data without much increase in distance.

However, there is a much worse problem than data bloat – adding more dimensions also involves larger potential for noise, and many classification algorithms will perform progressively worse as the amount of training data gets smaller compared to the number of dimensions.[4] This problem, which is commonly known as the *curse of dimensionality* (a term originally coined in [3]), motivates the desire to *reduce* the number of dimensions. Thus, when designing a descriptor, one should aim for a "sweet spot" in the number of elements.

It should be mentioned that both increasing and reducing the number of dimensions in a descriptor is possible, with certain restrictions. For instance, concatenating two or more smaller descriptors will yield a higher-dimensional combined descriptor – however, if the descriptors are not sufficiently independent, the gain will not be very large, as described above. The opposite procedure, dimensionality *reduction*, will be discussed in chapter 5.

[1]For brevity, one often speaks about the descriptor (which is just an output vector) as if it were to be designed, when more precisely, it the fingerprinting *method* is what is being discussed.

[2]The usage of the word "close" implies that some sort of distance measure is available to compare features. Chapter 5 contains a brief discussion on the topic of descriptor distance measures.

[3]It could be argued that it is not always obvious what is the "same" piece of music – for instance, is the same sonata played twice by the same piano player the same? What about by a different piano player? Edge-cases like this will mostly be ignored here.

[4]Just *how* much is difficult to quantify as long as the back-end algorithms have not been specified, nor the precise amount of training data available – however, a practical music matching system might have only a single reference point available.

2.3.2 Intellectual property issues

Ideally, one would want to design a system entirely on its own merits. However, in most legislations one would have to at least partially consider so-called intellectual property issues – that is, *copyright* and *patent* issues.

With regard to copyright issues, it is beneficial if the descriptor is *destructive.* A descriptor is destructive if it is not reasonably possible to reconstruct the original audio data from the descriptor. If destructive, the extracted data does not become subject to the copyright that governed the original audio, and can be transferred freely without worrying about copyright issues, in particular to a central server. Most practical descriptors, and all presented here, are destructive.

No particular patent search has been performed for this project, partially because it was deemed to be outside the project's scope, and partially because the current situation regarding software patents is somewhat unclear. However, it is believed that the work as described does not infringe on any patents, with the notable exception that MP3 encoding and decoding is claimed to be patented by several different entities, among them the Fraunhofer institute and Thomson Consumer Electronics.[38]

2.4 Distortion and noise

The human auditory system is capable of recognizing music even in the presence of severe distortion – even in the presence of highly perceptible distortion (such as the distortion imposed by a standard telephone line), the recognition rate is usually quite high. Thus, one will frequently encounter distorted audio (whether the distortion is perceptible or non-perceptible for the listener) in real-life data, and *robustness* becomes an important parameter in the design of a musical descriptor. In the following sections, a few common forms of distortion will be discussed.

2.4.1 Time shift

Time shifting is usually imperceptible, within certain limits. Although it could be argued that it should not be considered to be distortion at all, it could still create problems for some classes of descriptors. Time shifting occurs in practice even for content digitally ripped from CD – not only because of various slightly different variations between the same track on different CD publications, but also because of codec delay. In particular, MP3 encoding and decoding commonly results in over a thousand samples (25ms at 44.1kHz) of delay[24], depending on various factors. Thus, one must either somehow re-align the musical piece in question, or use descriptors that are robust against a reasonable amount of time shifting.

2.4.2 Amplitude changes

In general, the playback volume can be turned up and down (within limits) without noticeable distortion – a musical piece is not perceived as much different or distorted simply because the amplitude has changed. In the presence of *compression*[5], as frequently used on radio, the change

[5]The term "compression" for dynamic range limiting is somewhat unfortunate, as it is easily confused with digital compression algorithms such as MP3 or Ogg Vorbis. In general, "compression" will be used only to refer to lossy, digital compression from the next section onwards.

of amplitude could even change within very short time spans.

Even disregarding compression and assuming all-digital transfer (e.g. in a personal music library), one must be prepared for some change of the overall amplitude – in particular, MP3 requires an overall downscaling of the volume at some bit rates to avoid clipping in the decoded output. Fortunately, volume normalization is a relatively simple process, at least assuming uniform volume over a given time frame.

2.4.3 Lossy compression

Musical content is frequently encountered in *digitally compressed* form, processed to save storage space and bandwidth. A lossy codec will usually attempt to maintain perceptual transparency (or at least, avoid annoying *artifacts*), often at the cost of more distortion as measured under mathematical similarity measures.[8] (There are, however, a few examples of codecs taking special measures to ensure some aspects of mathematical transparency – in particular, voice codecs being designed to allow low-bitrate modem traffic to pass through.[15])

A typical lossy encoding/decoding cycle will introduce various forms of signal distortion – not only time shifting or amplitude changes as described above, but also frequently filtering, loss of stereo fidelity and various forms of quantization noise. Many of these effects are highly nonlinear in nature, and somewhat difficult to predict mathematically. Thus, experimental results are needed when assessing a descriptor's quality in the face of lossy compression.

Is it impossible to test each and every lossy codec, especially given the amount of bit rates and other tuning options frequently available. Thus, only two codecs were used, chosen as being relatively representative of typical use:

- First, the ubiquitous *MPEG-1 audio layer III* codec, more commonly known as *MP3*. MP3, standardized as early as 1991, is without doubt the most widely used audio codec for personal use as of 2007, and for many, synonymous with music on the desktop. Although several MP3 decoders are available, *LAME*, a freely available, high-quality encoder, was chosen, in the typical bit rate configurations of 128, 192 and 256 kbit/sec.

- Second, the *Vorbis* codec (sometimes *Ogg Vorbis*, as technically, Ogg is the bitstream format and Vorbis is the audio codec), designed by a volunteer developer team led by Chris Montgomery in 1998 after concerns over the licensing of MP3. Vorbis is specifically designed to be freely licensable without patent concerns, and although the claim that Vorbis is not covered by any patents has been contested[1], no company has yet claimed any ownership over Vorbis.

  In this project, Vorbis was chosen as representative of the ever-growing field of "next-generation" audio codecs, having shown itself competitive in audio quality in several independent listening tests.[28, 27, 11] The bit rates were set to 64, 128 and 192 kbit/sec[6] – although it is still disputed whether *any* of the "next-generation" audio codecs can produce audio at 64 kbit/sec rivaling a good MP3 encoder at 128 kbit/sec, data at this bit rate would serve as a "low anchor", as it is unlikely that most people would store music in a lower bit rate.

[6]As Vorbis is a true VBR codec, these are nominal bit rates – the encoder parameters are set to produce bit streams that on average are close to the nominal bit rates, but there is no attempt to *force* the stream under a given target. Thus, the actual bit rate will vary with the audio content.

2.5 Choice of source fragment

Before discussing actual descriptors, one will need to decide what part of the input track to use. In the given use case, there are three basic strategies:

- One could use the entire track. This will ensure no data is missed, but requires decoding of the entire track (which might not always even be available) for proper classification. As decoding is often among the most computationally intensive parts of the process, increasing the amount of data needed might be undesirable. Still, it is a very viable strategy, and it opens up the possibility for variable-length descriptors (having, for instance, one descriptor set per 250ms of audio) for easier partial matching – see section 3.4 for a brief discussion on this topic.

- One could use only the first N seconds of each track, where N usually varies from 5 to 120. (If a track is shorter than N seconds, it is usually padded with zeros.) When decoding and processing of only a fraction of each track is needed, overall processing overhead is reduced. Furthermore, all fingerprints will be based on the same amount of data, simplifying some calculations and considerations.

- In [5] and others, *thumbnailing* algorithms are described, designed to extract the "most representative" N seconds of a song (or more generally, N seconds that represent the song well – the thumbnail does not need to consist of a single N-second cut from the original track); these could be used as a high-quality, fixed-length input to the descriptor algorithm. However, computing a thumbnail requires decoding of the entire track, and it is in general not known how robust thumbnail selection is under lossy compression or other forms of distortion.

Both the first and second options have been used in existing fingerprinting systems; the author knows of no fingerprinting systems using musical thumbnails. In this project, a fixed 30-second clip from the beginning of each track was used as a performance/complexity trade-off; in general, 30 seconds should be more than enough to both identify a track for a human, and collect enough data for statistical measures to stabilize.[7]

2.6 Basic musical descriptors

In this section, a few basic, scalar musical descriptors will be introduced – although by no means a complete list, the descriptors provided will serve as simple examples of what could constitute a descriptor. Being simple mathematical formulas, they work mostly in the time domain, without much psychoacoustical or musical justification. Nevertheless, they should not be immediately discounted as "primitive" – as will be evident in chapter 6, many of them perform quite well in the given test scenario, and can play a useful part in a larger system.

In the present implementation, a certain amount of preprocessing was done prior to the descriptor calculations. Although many of the given descriptors could work equally well without preprocessing, some would not, and it was regarded more sensible to let all descriptors work on the same sound signal. The assumptions made were:

[7] This choice also allowed the zero crossing rates to be replaced with simple zero crossing counts, the difference between the two measures only being a constant factor.

- The signal was assumed to be *monoaural* (consisting of only one sound channel). While there certainly is much fidelity in a stereo signal not present in a mono signal, the loss of stereo information usually will not affect *recognition* for a human listener at all. Also, under lossy compression, large amounts of stereo information is typically distorted or even completely sacrificed[9], making descriptors based on stereo information less useful in the presence of digital compression.

  All stereo signals were down-converted to mono signals by averaging the value of the two channels sample by sample; although a linear formula might be regarded as less than ideal with regard to perceived signal power[19], it is by far the most used approach for stereo-to-mono down-conversion.

- The signal was assumed to have *zero DC offset*. Change in the DC offset, the average of the signal, does not influence hearing at all, and is computationally simple to remove. Testing did not show much DC offset in practical recordings, but it was nevertheless preprocessed out.

- The signal was assumed to be *energy normalized* – in other words, all tracks were assumed to contain the same amount of total energy. (This was realized simply by measuring the original total energy, and scaling the entire signal by a factor to correct the energy level if wrong.)

- The signal was assumed to have a *constant sampling* frequency of 44.1kHz, matching that of an audio CD.

2.6.1 Mean/square ratio

The *mean/square* ratio is a simple time-domain scalar descriptor. It measures the normalized average distance from zero, and is calculated as:[8]

$$r = \frac{\text{avg}(|x|)}{\text{RMS}} = \frac{\frac{\sum |x_i|}{N}}{\sqrt{\frac{\sum x_i{}^2}{N}}} = \frac{\sum |x_i|}{\sqrt{N \sum x_i{}^2}}$$

Note that due to the normalization discussed in the previous step, the RMS is constant or near-constant. However, the normalization factor will still influence the nominator less than the denominator (due to the squaring in the denominator), so the mean/square ratio is not scale-invariant.

2.6.2 Rate of zero crossings

The rate of zero crossings is simply how many times a signal goes from being positive to negative, or vice versa, divided by the total number of samples. It is simultaneously a measure of frequency *and* noise.

[8]Note that the limits of the summations have been dropped for brevity; when not written, the summation is generally taken to be over the entire sequence – that is, i ranges from 0 to $N-1$, inclusive.

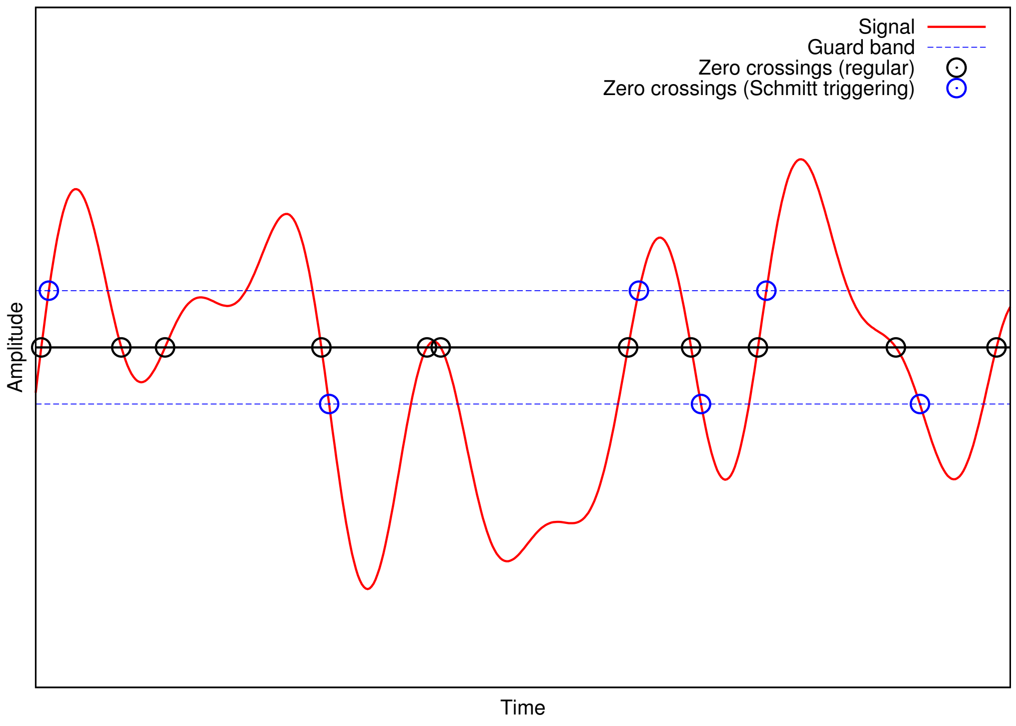

Figure 2.1: A signal with zero crossings marked, both using the normal zero crossing rate as described in section 2.6.2, and using Schmitt triggering as described in section 2.6.3.

It is possible to express the zero-crossing rate in closed form (assuming that no samples are exactly zero), although the resulting expression is not very instructive:

$$r = -\frac{\sum_{i=1}^{N-1} \left(\frac{x_i x_{i-1}}{|x_i x_{i-1}|} - 1 \right)}{2(N-1)}$$

The zero-crossing rate is scale-invariant, but easily skewed by only a moderate amount of noise (in particular high-frequency noise). It can also be skewed by DC offset.

2.6.3 Schmitt triggering

Schmitt triggering (after Otto H. Schmitt) is an incremental improvement on counting the zero crossings. A guard band $-G < x < G$ is defined, wherein any samples that fall inside the guard band are ignored (also known as *hysteresis*). This improves noise resilience, especially in the quieter areas of a recording, as a zero crossing is not counted until the signal has crossed the entire guard band. Depending on the choice of G, Schmitt triggering can also be used for primitive tempo estimation.

Figure 2.1 illustrates the difference between the simple zero crossing count and Schmitt triggering – note how the sensitivity to small changes near origin is reduced when Schmitt triggering is employed.

2.6.4 Steepness

Steepness is, intuitively, a measure of the rate of change in the signal. It is the mean value of the numerical derivative of the signal:

$$s = \frac{1}{N-1} \sum_{i=1}^{N-1} |x_i - x_{i-1}|$$

The steepness is, like, the zero crossing rate, a measure of frequency, but it is less influenced by noise, in that small amounts of noise will only increase the steepness marginally.

2.6.5 Track length

The track length is simply the length of the source track in seconds (or samples; the end effect will be the same, but seconds is usually regarded as a more natural measure). The usefulness of the track length as a descriptor is highly dependent on the use case in question: In discerning different tracks from CD, it can be an excellent measure, in genre classification, it is nearly useless[9], and finally, in some cases (such as when identifying what music is being played on the radio at a given instant), it might not be available at all.

It should be mentioned that in the case of lossy compression, finding track length may require decoding the entire track, or at least a partial decoding – not all audio file formats store the track length explicitly in the file. Care must be taken in the decoding implementation if one wishes to avoid a computationally intensive full decode, negating some of the benefits of using only the first N seconds of the track. (In particular, there is no officially standardized way of storing the track length in MP3, and not all MP3 decoders support scanning through the file to determine the track length without a full decode and PCM synthesis.) However, it is extremely robust to lossy compression; even though not all codecs preserve the length to the sample (some round up to the nearest block, and some introduce extra delay, as discussed earlier), the change will usually be small even at very low bit rates.

2.6.6 Centroid

The *centroid* is the first spectral measure introduced. First, the entire sequence is transformed from the time domain into the frequency domain by means of the *discrete Fourier transform* (DFT)[32]:

$$X_k = \sum_{n=0}^{M-1} x_n e^{-\frac{2\pi i}{M} kn}; k = 0, \ldots, N-1 \tag{2.1}$$

The centroid (center of mass) of the spectrum is then found by calculating

$$f_c = \sum_{k=0}^{M-1} \frac{f_k |X_k|}{|X_k|}$$

where f_k is the center frequency of bin k.

[9]One could imagine that *some* genres could be estimated by length – for instance, if a track is 30 minutes long, it is much more likely to be a classical symphony than a pop track.

Taking the DFT of the *entire* sequence might seem excessive, but in practice it is not a problem – by computing the DFT by means of an efficient *Fast Fourier Transform* (FFT), the FFTW[12] library computes a 1,323,000-point DFT (30 seconds of 44.1kHz audio) in about 150-200ms on a modern 1.2GHz computer.

The centroid is relatively insensitive to white noise – however, it is easily skewed by filtering, and it is not a very good measure of transients in general.

2.7 Human descriptors

Not much is known about the exact methods used by the human auditory system to *recognize* music, but several intuitive measures exist for *describing* music, and it is reasonable to assume that these measures also play a role in recognition. Among these are measures like tempo (and more detailed, rhythm), the genre, the instruments used, the lyrics (if any) and characteristic melody lines (so-called *melodic contour*).

Many of the more elusive descriptors of this class are those dealing with *timbre* – that is, the "feel" of the sound not described by pitch or amplitude (or, as it has been described, "the psychoacoustician's multidimensional wastebasket category"[29]). In particular, timbre is what makes it possible to distinguish different instruments from each other, but a skilled musician can produce various different timbres from the same instrument. Timbre encompasses measures such as coloration, roughness, attack time and many others, mostly subjective measures that can be hard to quantify mathematically.

The measures used by humans are usually considerably more high-level than the current state of the art in signal processing – even the seemingly simple task of finding the precise position of beats and thus the tempo is surprisingly complex[23]. Nevertheless, systems inspired by such measures, such as *query-by-humming* systems[13, 22], complete transcription systems[21] and even systems capable of discerning the style of different concert pianists[44] have been proposed.

Yet, there are many applications where such descriptors would be useful – in particular, when a descriptor is assigned a textual description that a human can relate to, it is possible for the user to manually *modify* the searching criteria in, for instance, a neighbor search. A typical query might be phrased "something like song X, but with more percussion, and with a female singer", depending on the user's wishes and/or needs. In the *Music Genome Project*[43], experienced human analysts have rated a relatively large corpus of music manually, using 150-400 descriptors such as "Acid Jazz Roots", "Prominent Mandolin Part" or "Tight Kick Sound" on a one-to-five-scale. While this allows for much flexibility, it is also very intensive in manual labor.

Due to the complexity inherent in implementing this class of descriptors well, they have been left out in this project. Nevertheless, much of the post-processing and searching in a MIR system is believed to be applicable to these descriptors in the same way as the lower- and medium-level descriptors discussed presently.

Chapter 3

Mel frequency cepstral coefficients (MFCC)

In this chapter, *mel frequency cepstral coefficients* (abbreviated MFCCs) will be introduced and discussed. MFCCs have been used and researched extensively in various acoustical applications – they are probably best known from speech recognition systems (where they have long enjoyed widespread use), but are now also finding increased use in musical descriptor systems.

3.1 Psychoacoustical motivation

MFCC is a *spectral* measure, attempting to mimic certain key aspects of the human auditory system. The basis of MFCC is the *mel scale* (from the word "melody"), a perceptual frequently scale designed to approximate the human experience of pitch heights at constant loudness, as compared to the usual unit of 1 Hz, which is a strictly physical measure.

The conversion from the regular frequency scale into mel scale is expressed as

$$B(x) = 1127 \ln(1 + \frac{x}{700})$$

where x is measured in Hertz and the result is in mel. The inverse transform (from mel to Hertz) thus becomes

$$B^{-1}(x) = 700(e^{\frac{x}{1127}} - 1).$$

In MFCC, closely related frequencies are grouped into triangular spectral bands, placed linearly across the mel scale (see figure 3.1). The spectral bands are inspired by the *critical bands* in the human inner ear, which play a central role in the ability or inability to distinguish two simultaneous tones from each other.

However, pitch is not the only measure that is perceived roughly logarithmically; sound *energy* is also perceived logarithmically, and thus frequently measured in decibels (dB).[1] Thus, what is used in further calculations is not the total energy per band, but rather the *logarithm* of the energy per band. (To avoid problems with floating-point underflow, a small "fudge factor" was added before the logarithm, to ensure that a filter output of zero or nearly zero would not adversely affect the entire descriptor.)

[1]The notion of logarithmic scale in this context must be understood as an approximate measure. In reality, the human notions of both pitch and loudness are complex and interrelated – however, the logarithmic scale is usually a good approximation.

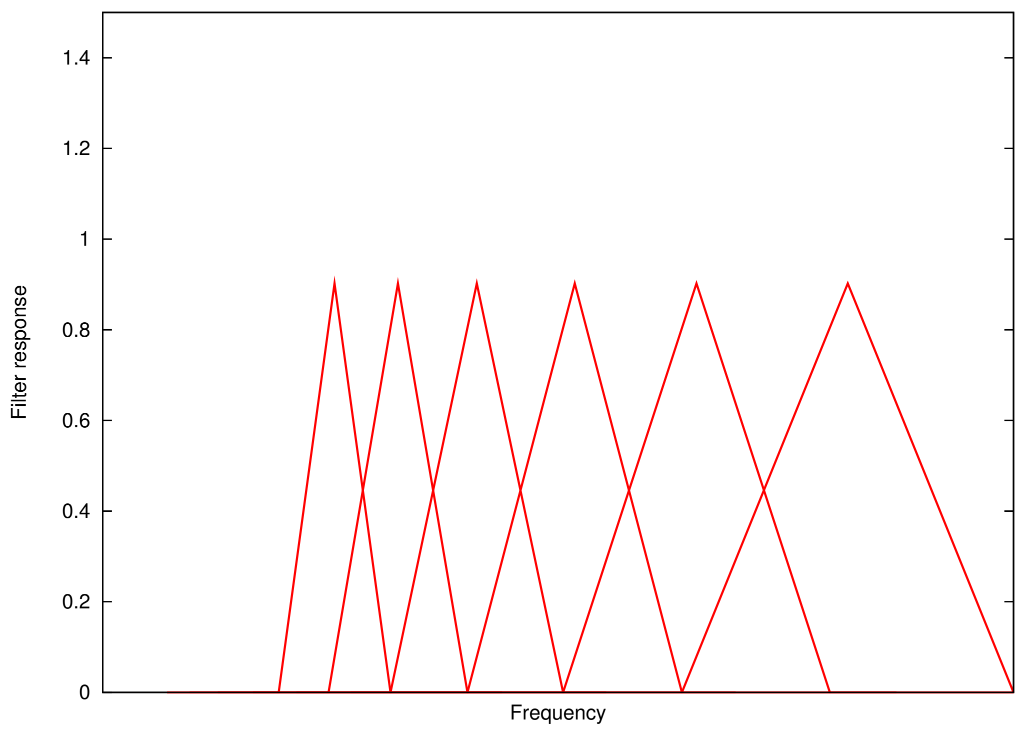

Figure 3.1: Frequency response for a few MFCC filters. For easier viewing, only the last few filters are shown – the full filter bank covers the entire spectrum, unlike in this figure.

3.2 Homomorphic transforms

A *homomorphic transform* $\mathcal{H} : \mathcal{C}^n \to \mathcal{C}^n$ is a transform that satisfies

$$\mathcal{H}\{f \star g\} = \mathcal{H}\{f\} + \mathcal{H}\{g\}$$

where $f \star g$ is the *convolution* of the sequences f and g.

Homomorphic transforms are interesting for several reasons – in the context presently discussed, the most interesting interpretation is viewing f as the original sound data and g as the *impulse response* of a filter. Here, "filter" covers not only the usual class of low-pass, high-pass, band-pass etc. filters, but also other effects as reverb, which are usually possible to describe in terms of a convolution. Thus, any filtering will only each affect each frame by adding a constant vector (depending on the filter in question), and thus will be canceled out under for instance differentiation. The use of this effect will be discussed further in section 3.4.

3.3 Formal description

As there are several, slightly different variations of MFCC calculation, a brief description on the precise method used in this project is given here. The method is largely based on [20].

First, the input data is broken into chunks of 1024 samples each, or about 25ms. The chunks overlap by 50%, which means that the beginning of successive chunks are 512 samples apart. First, the input chunk is *windowed* using a Hamming window of length $N = 1024$, that is,

$$x'_n = x_n w_n;\ n = 0, \dots, N-1$$

where w_n is the Hamming window, defined as

$$w_n = 0.54 - 0.46\cos(\frac{2\pi n}{N-1}).$$

It should be mentioned that with this choice of window and overlap, so-called *perfect overlap* is realized, in that the overlapping windows sum to unity, and thus every sample is given the same weight.[37]

After windowing, the DFT (equation 2.1) of the signal is calculated, again using the FFTW library as in section 2.6.6. A filter bank with $M = 32$ overlapping, triangular filters is then applied, where each filter H^m (with index $m = 1, \dots, M$) is defined by:

$$H^m{}_k = \begin{cases} 0 & ; \quad k < f_{m-1} \\ \frac{k-f_{m-1}}{f_m - f_{m-1}} & ; \quad f_{m-1} \le k \le f_m \\ \frac{f_{m+1}-k}{f_{m+1}-f_m} & ; \quad f_m \le k \le f_{m+1} \\ 0 & ; \quad k > f_{m+1} \end{cases} \tag{3.1}$$

The center frequencies[2] of the filters, f_m, are placed linearly on the mel scale:

$$f_m = B^{-1}(m\frac{B(F_s/2)}{M+1}),$$

which makes the filter bank span the entire available spectrum, from 0 to $F_s/2 = 22050$ Hz.

From each filter, the logarithm of the output energy of the filter is then computed:

$$S_m = \ln\left(\sum_{k=0}^{N-1} |X'_k|^2 H^{m+1}{}_k\right); m = 0, \dots, M-1 \tag{3.2}$$

Finally, the resulting vector is transformed using the *discrete cosine transform* (DCT), more precisely, the DCT-II[2]:

$$c_n = \sum_{m=0}^{M-1} S_m \cos\left(\frac{\pi n(m-\frac{1}{2})}{M}\right); 0 \le n < M$$

Analogously with the DFT, the DCT is computed efficiently in form of a *fast cosine transform* (FCT)[26], again using the FFTW library.

The sequence c_n contains the final set of MFCC coefficients. It should be noted that MFCC as described is not a homomorphic transform (it would be if the order of the logarithm and summation in equation 3.2 were reversed) – however, it is approximately homomorphic for filters with a smooth transfer function, and the given form is somewhat more robust to noise and estimation errors.[20]

[2]In this description, f is taken to be a frequency; however, for a practical filter implementation, it must be converted to a bin number before use. In particular, equation 3.1 assumes that this conversion has already taken place, *or* that k is the center frequency of the bin instead of a bin number.

3.4 Statistical data extraction

Even though the MFCC procedure yields a significant data reduction, the amount of data is still considerable – in the present case, 32 values for each frame, which equates to about 80000 values (about 2500x32) for the chosen 30-second window. How to further process this set of values is an open question, and several measures have been proposed.

One of the most tempting solutions is to simply keep all the values, either in full form or in some quantized version. It could be argued that this is simply postponing the problem to the searching stage – however, few other approaches preserve the multiple layers of temporal information present in most music as well. On the other hand, the fingerprint gets large, and somewhat complex searching algorithms are needed to search this data. Most systems employing this solution use either larger windows than the 1024-sample windows used here[14], vector quantization[34], dimensionality reduction[35] or other measures of reducing the total amount of data somewhat while keeping most of the temporal information intact.

The alternative to keeping all the data usually involves some form of statistical data extraction. Various statistical tools have been employed to this end, from the simple computation of averages to the slightly exotic suggestion of using the partial output of a singular value decomposition (SVD)[31]. For this project, a relatively simple method was chosen. As primary measure, the *mean* (or *first zero moment*) was estimated from the data for each coefficient m:

$$\hat{\mu}_m = \frac{1}{N} \sum_{i=0}^{N-1} c_{m,i}$$

Also, the first *central moments* were extracted – in particular, the second central moment (variance), but also the third and fourth moments (which in turn can be used to define *skewness* and *kurtosis*). The expression for the dth central moment of the mth MFCC coefficient is:

$${C^d}_m = \frac{1}{N-1} \sum_{i=0}^{N-1} (c_{m,i} - \hat{\mu}_m)^d$$

In order to prevent the final descriptors from becoming unreasonable large, the dth root was extracted from this expression, yielding finally:

$${C'^d}_m = \sqrt[d]{\frac{1}{N-1} \sum_{i=0}^{N-1} (c_{m,i} - \hat{\mu}_m)^d}$$

Notably, due to this extraction, a scaling of the input signal would transfer directly to the output descriptors – in other words, the scale sensitiveness of the measure is reduced (or rather, linearized) by the root extraction.

When extracting statistical moments, one discards all temporal information – information that is, as mentioned above, often important to recognition. There are multiple ways of alleviating this. One method is using *Hidden Markov Models* (HMMs), modeling the signal as a finite state machine where each state has an associated probability distribution and transition probabilities to other states. HMMs have been found to be better than regular Gaussian models (such as the one presented here) for spectral modeling, but unfortunately, this does not necessarily translate into better accuracy[6] – also, estimation (and in part, comparison) of these models is a rather complex task. Another approach would be using multi-scale approaches such as *wavelets*, which might be well-suited to the multiple layers of temporal structure present in music – however, it is not readily apparent exactly how wavelets would be used to extract useful information from the MFCC data.

A third and much simpler option is simply using the derivatives of the MFCC coefficients:

$$\Delta c_{m,i} = |c_{m,i+1} - c_{m,i}|$$

The derivatives are treated in the same way as the regular coefficients – the mean and first central moments are extracted for each m, and the results are stored and used as part of the final descriptor. (The absolute value is taken to avoid getting a mean value that is almost always very close to zero.) While the use of derivatives still involves discarding long-term temporal information, it does help retain short-term transient information, and is commonly used in among others speech recognition systems, sometimes in conjunction with double derivatives.[20] In this project, this was the favored approach, primarily for its simplicity and low processing overhead.

Chapter 4

Floor-1 cepstral coefficients (F1CC)

While MFCC works reasonably well for modeling the inner ear, its use as a musical descriptor in the presence of lossy compression could be improved on. In particular, modern sound compression systems rely on *auditory masking* to single out parts of the spectrum that can be compressed more aggressively (or removed altogether) without affecting the perceived result – energy in one part of the spectrum can make the ear relatively insensitive to energy in other, nearby parts of the spectrum (following certain rules and curves), and this phenomenon makes more efficient audio coding possible, as less detail needs to be preserved in masked areas of the spectrum. However, these spectral differences between the compressed and uncompressed audio, while imperceptible (or only weakly perceptible) to the human ear, still introduce undesirable noise in the MFCC coefficients. Thus, it it would be desirable to refine the MFCC model to make it more robust towards these kinds of artifacts.

In this chapter, a novel MFCC refinement based on the Vorbis psychoacoustical model is presented. By reusing an internal representation of the Vorbis encoder, *tone-tone* masking that occurs when a louder tone can "hide" another weaker nearby tone perceptually is taken into account.[1] As the input to the MFCC calculations is based on masking strength at different frequencies, a reasonable name would be *mask-frequency MFCC*; however, the resulting natural acronym, MFMFCC, was deemed too cumbersome. Instead, the name *floor-1 cepstral coefficients* (F1CC) was chosen, after the Vorbis representation from which it is derived.

4.1 An overview of the Vorbis encoder

In the following, only a rough overview of the Vorbis encoder and format will be given. A formal description of the Vorbis audio format can be found in [10], and [39] contains a description of the psychoacoustical model used in the encoder.

Vorbis encoding is a three-stage process. First, the audio is split into overlapping frames which are then processed separately. For each frame, the encoder computes a *floor*. The floor is a measure of approximate discrimination threshold as estimated by the encoder – that is, a measure of how much noise would be imperceptible in a given frequency area, given the rest of the audio spectrum. The floor is quantized, stored and then subtracted from the input

[1]The resulting model is not linked to the Vorbis codec; the reuse of the Vorbis code is primarily to ease the prototyping of the system. By using a known-working model and code, much tedious and error-prone reimplementation (in particular with regard to tone curve measurements) is avoided.

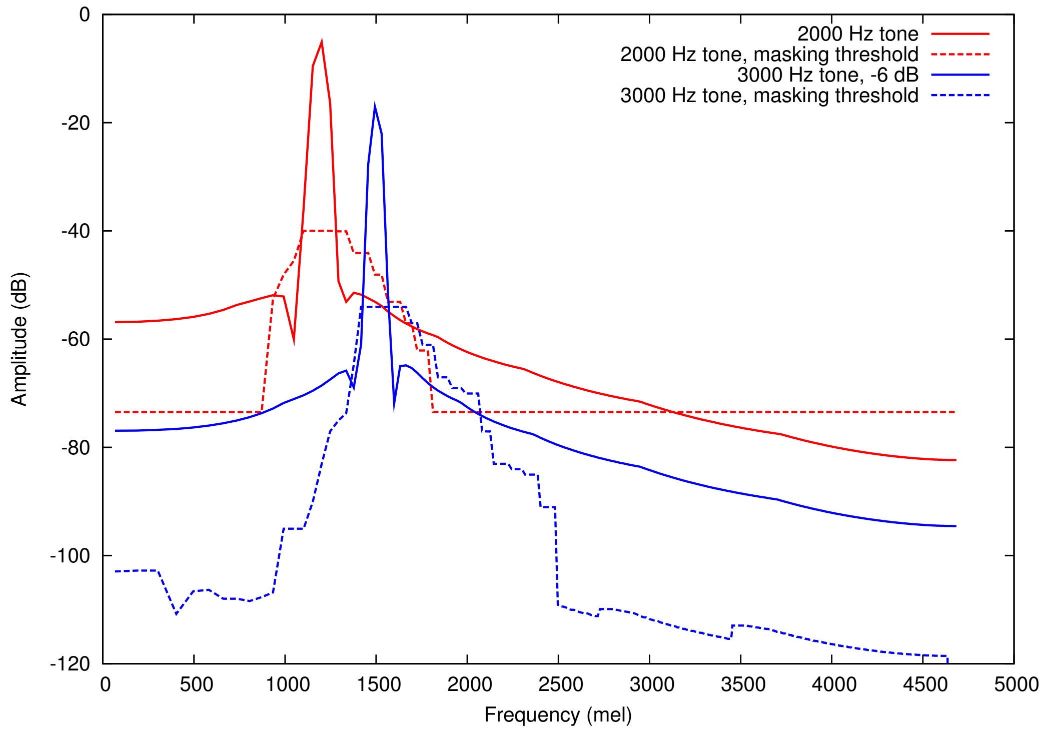

Figure 4.1: A tone being masked off by a 6 dB louder tone in the F1CC model. Note that the maskee is almost completely hidden, as the union of the masking curves is what is used as input to the F1CC calculations.

signal (resulting in a whitening of the spectrum), and the resulting residue is encoded using a codebook-based vector quantization algorithm.

The primary point of interest in this context is the floor computation – the residual encoding will not be described further here, nor will other practical aspects of the codec be discussed. Again, the reader is referred to [10] for a full description.

4.2 Floor computation

The floor[2] is computed by combining two separate curves. The first, the *noise masking curve*, is somewhat misnamed, as it models many effects besides just noise masking. In particular, it is constructed from estimates of tonality, overall noise envelope and a hard-wired bias curve.[39] Many of these values are not expected to be very well preserved by lossy encoding[30], and thus, the noise masking curve will not be discussed further here.

The *tone masking curve*, however, is of more use. For each tone (ie. frequency band), a tonal masking curve is looked up from a table. (The tonal masking curves were independently measured for Vorbis, citing quality concerns with the "standard", classic set of curves originally measured by Ehmer[4].) The superposition, or union, of these curves is then used as a combined

[2]Technically, Vorbis can accommodate multiple different types of floors. However, only two floors are defined in the Vorbis I specification, of which one, *floor 0*, is not in general use, and floor 1 is what is described here.

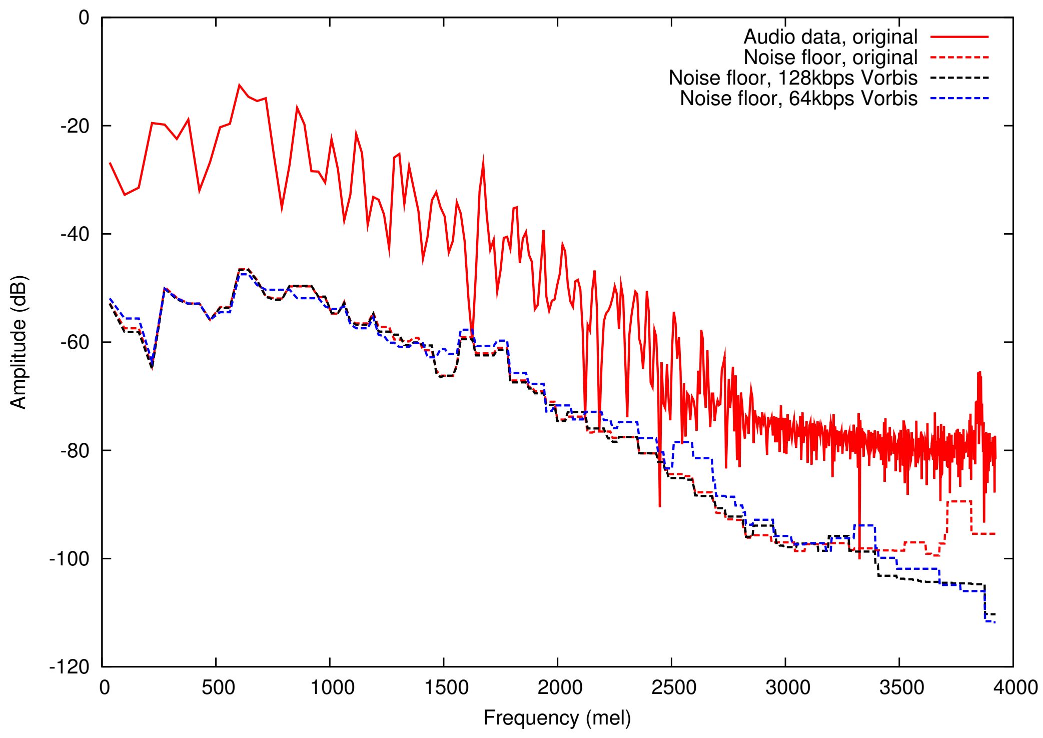

Figure 4.2: A 1024-sample frame of music, with noise floors for both the original audio and two Vorbis-encoded versions – note in particular the divergence between the encoded versions over about 3200 mel. Audio data from "The tale of Room 420" by Ehren Starks.

tonal masking curve, which is finally combined with an ATH (Absolute Threshold of Hearing) curve to produce the final tone masking curve.

Two points are worth noting:

- First, the tone masking curve does not represent audio energy; in particular, it has no phase information. However, it is definitely *correlated* with the audio energy, and in particular, a tone that is masked by another tone will not affect the tone masking curve at all. (See figure 4.1 for an example.) Thus, it makes sense to treat the curve as if it were colored noise; it will share many spectral characteristics with the original audio spectrum, and the MFCC process in particular does not use the phase information.

- Second, it is not possible to know the exact ATH curve at encoding time, as the actual sound volume is not known before at playback time. (This could even change mid-piece, as the listener turns the volume up and down.) Thus, the Vorbis encoder makes a pessimistic estimate based on the maximum peak in the spectrum for each frame – as the loudest peak can not be higher than about 100 dB SPL, the most pessimistic ATH curve can be applied at 100 dB below that. However, the ATH curve will then vary between frames, based on the highest peak, which is not desirable for an audio descriptor. Thus, the ATH logic was disabled when used in F1CC computation.

After the tone masking curve has been computed, it is treated as an audio spectrum as described above, and input to a modified version of the original MFCC process described in chapter 3. The modifications are minor, but should still be mentioned:

- First, as the tone masking algorithm never emits a value that is exactly zero (even for an all-zero input spectrum), there is no need for the "fudge factor" described earlier, and it was consequently dropped.
- Second, the tone masking curves were found to not be very precise over about 3200 mel (about 11 kHz), being significantly distorted by lossy encoding. Thus, the triangular windows were re-aligned so that the highest frequency used was 3200 mel, reducing the overall amount of noise entering the system.

It should be mentioned that the tone masking curve generation is relatively compute-intensive – in general, extracting F1CC coefficients was about twice as slow as extracting MFCC coefficients. However, the increase in computational complexity would seem to be worth it in most cases, as F1CC indeed appears to be more robust to lossy encoding than MFCC (see section 6.3 for detailed results).

Chapter 5

Methods

5.1 Testing model and distance metrics

When assessing the quality of descriptors, a test model is required in which to test the various descriptors (and combinations thereof) against each other. For this project, a relatively simple *classification* or *matching* test was used:

- First, all tracks were distorted, by lossy compression in the formats and bit rates discussed in section 2.4.3: 128, 192 and 256 kbit/sec for MP3, and 64, 128 and 192 kbit/sec for Vorbis.
- Then, for each track, its feature vector was compared to the feature vectors of all reference tracks (the original, unencoded tracks).
- If the closest match was the correct reference, a "success" would be noted; in the opposite case, a "failure".

Intuitively, the classification can be visualized as the N-dimensional hyperspace being partitioned into areas, one for each reference – a so-called *Voronoi diagram* (see figure 5.1, next page). Any test point that lands inside the cell of the parent is counted as a success – if, however, the cell border is crossed, the track will be inevitably misclassified and a failure is counted.

While the model used is primitive (it does not, for instance, consider false positives, in that a track not in the reference database should be matched as "none found"), it was found to be sufficient for comparing the various descriptor sets against each other. For a practical application, the choice of testing model would probably be different, depending on the terms imposed by the use case and the data available. (In particular, an unmodified copy of the original tracks might not be available.)

When comparing descriptor vectors (in order to consider what track is the "closest match" in the model above), a distance metric is needed. Two common distance metrics were used and evaluated: The standard *Euclidean* distance and the *Mahalanobis* distance.

5.1.1 Euclidean distance

The *Euclidean distance* is one of the most well-known distance metrics, and is easily extended from the familiar two- and three-dimensional cases to N-dimensional hyperspace. The Euclidean

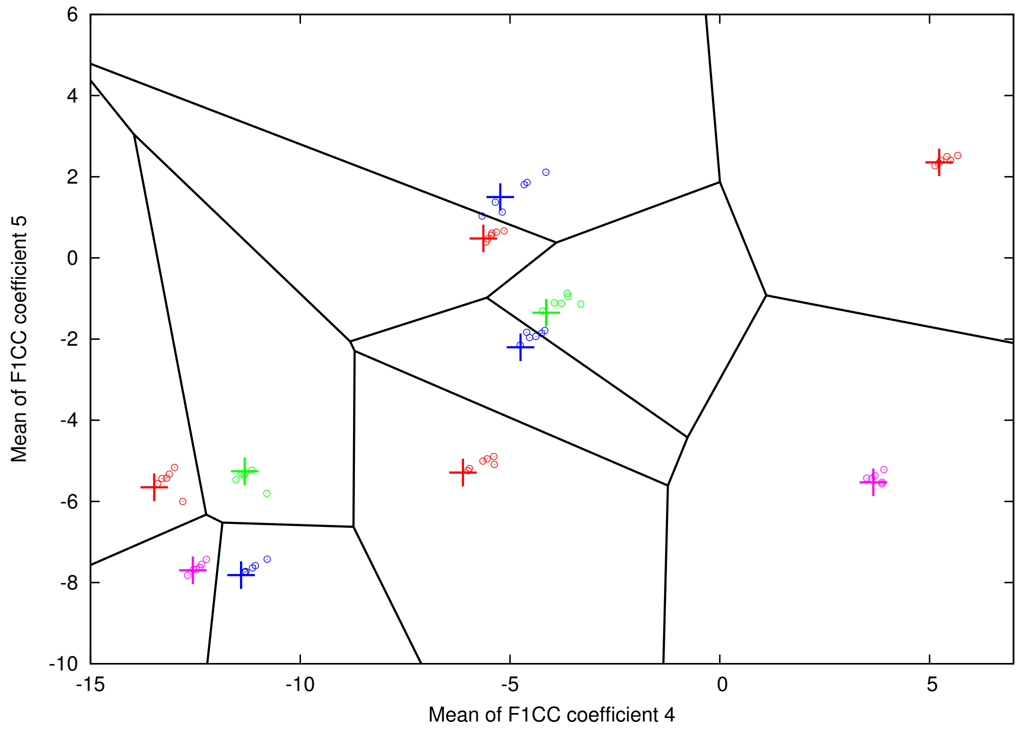

Figure 5.1: Fourth and fifth F1CC coefficients from all eleven tracks from the album "Nightwatch" by Silje Nergaard. The large crosses represent the FLAC originals; the smaller circles in the same color represent the MP3 and Vorbis versions. Note the black decision boundaries generated at equidistant points from the references, forming a Voronoi diagram; three tracks end up in the wrong cell and will thus be misclassified.

distance between two N-dimensional vectors $\vec{a}$ and $\vec{b}$ (represented as column vectors, as in $\vec{a} = (a_1, a_2, \ldots, a_N)^T$) is defined as:

$$D_E(\vec{a}, \vec{b}) = \sqrt{(\vec{a} - \vec{b})^T (\vec{a} - \vec{b})} = \sqrt{\sum_{i=1}^{N} (a_i - b_i)^2}$$

As $\sqrt{x}$ is a monotonous function (for $x \geq 0$), and the distance metric was only used for comparison, ${D_E}^2$ was used instead of D_E. This modified measure is equivalent to the standard Euclidean distance when used to find the closest match, but computationally cheaper, as one square root is saved per comparison.

5.1.2 Mahalanobis distance

The *Mahalanobis distance* is a different, statistical distance measure, first presented in [25]. It is not unlike the Euclidean distance, but better suited to spaces where the vector elements might be correlated. Also, unlike the Euclidean distance, it is *scale-invariant* in that an overall (uniform or non-uniform) scaling of the data set will not change the computed distance.

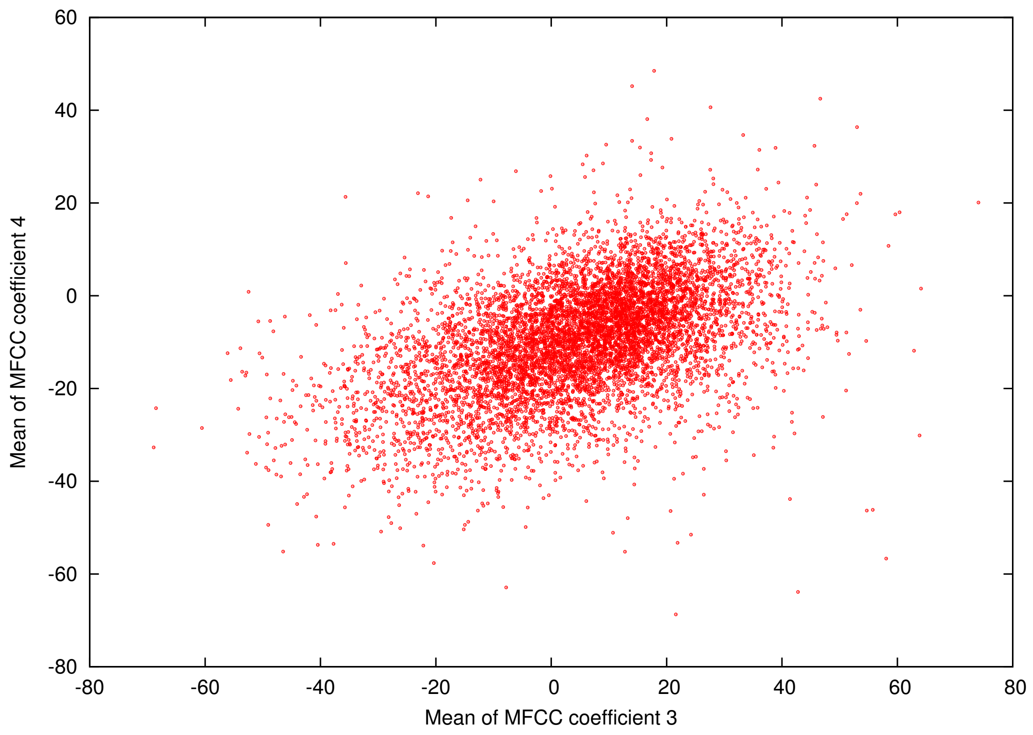

Figure 5.2: MFCC coefficients 3 and 4 for the full song list (uncompressed versions only), with each song represented as a point. Note the diagonal shape of the point cloud, indicating correlation between the coefficients.

The Mahalanobis distance is computed as:

$$D_M(\vec{a},\vec{b}) = \sqrt{(\vec{a}-\vec{b})^T\Sigma^{-1}(\vec{a}-\vec{b})} \tag{5.1}$$

where Σ is the *covariance matrix* for the given data set.[1] (In the presently described classifier, the covariance matrix was estimated from the set of all feature vectors. This works well as long as there is enough data compared to the number of the elements in the covariance matrix to be estimated.) Note that when $\Sigma = I$ (all vector elements are uncorrelated and normalized), the Mahalanobis distance is equivalent to the Euclidean distance.

For performance reasons, again ${D_M}^2$ was used instead of D_M. Also, equation 5.1 was expanded into

$$D_M(\vec{a},\vec{b})^2 = (\vec{a}-\vec{b})^T(\Sigma^{-1}\vec{a}-\Sigma^{-1}\vec{b}). \tag{5.2}$$

As $\Sigma^{-1}\vec{x}$ could be precomputed for each feature vector $\vec{x}$, $O(N^2)$ matrix multiplications were saved per N-element classification run.

The increased sophistication of the Mahalanobis distance does not come without a cost. Even with the expansion optimization from equation 5.2, the Mahalanobis distance is computationally more expensive than Euclidean distance – about twice the amount of computation is needed

[1]It can be confusing that Σ is used both as summation sign and to denote the covariance matrix. Usually, however, the meaning should be clear from the context.

(ignoring the pre-computation). Furthermore, sophisticated methods for accelerating range queries in Euclidean space exist, among them *R-trees*[16]; not all of these can be expected to be readily adaptable to the Mahalanobis metric.

Mahalanobis distance was not used together with dimensionality reduction (which will be presented in section 5.4), as it would constitute an overlap of functionality, in that the dimensionality reduction methods used also include decorrelation measures similar to the Mahalanobis distance.

5.2 Data material

For this project, the data set consisted of a personal music collection consisting of 7372 CD tracks in digital, lossless form. Consisting mostly of contemporary music, it was believed to be a reasonable approximation to a medium-sized CD collection. After a few duplicates and other anomalies were removed[2], the final set numbered 7242 tracks, totaling about 514 hours of music, about 175 GB of FLAC data. The complete list of albums and the number of tracks from each can be found in appendix C.

Although 7372 tracks is a reasonable number for a personal CD collection, it is probably a bit smaller than a typical digital music collection, and it is not remotely close to the total amount of tracks commercially available today, which has been estimated informally to be several tens of millions of tracks.[34] (As a point of reference, the Shazam music recognition system cites a database of 1.8 million commercial tracks.[42]) Unfortunately, as will be seen, the data set is simply too small to draw conclusions about the feasibility of a larger-scale system based on the same principles – in a sense, the problem of matching against a database this small is "too easy", which makes it difficult to separate good algorithms and descriptors from bad ones.

The demand that all tracks be available in uncompressed (or at least, non-lossy compressed) form makes acquisition of a larger test library difficult, as most personal music collections contain mainly MP3-compressed tracks. However, using already-compressed music would risk severely skewing the study of how lossy compression would affect the chosen descriptors, as almost all tracks would be compressed twice, accumulating (possibly distinct) distortion in both compression steps. Thus, it was decided to stay with using the given track set, even though a music library of ten or possibly even hundred times the size would be desirable for a more realistic, full-scale test.

5.3 Encoding and feature extraction

The data set was stored in the FLAC format, reducing the storage requirement by about 2:1 while still maintaining bit-accuracy. All FLAC files were then encoded into MP3 and Vorbis, using the bit rates selected in chapter 2: 128 kbit/sec, 192 kbit/sec and 256 kbit/sec for MP3, and 64 kbit/sec, 128 kbit/sec and 192 kbit/sec for Vorbis.

Although the encoding process is CPU intensive, it is also easily parallelizable, as each track can be encoded separately. Thus, a simple job distribution system was written, storing the job queue in an relational database and having the worker machines execute them on a first-come, first-serve basis. At most, thirteen separate CPU cores were involved in the encoding, which

[2]Duplicate removal by hand is a rather tedious task – however, during testing, the classifier got good enough that looking at the list of misclassified tracks showed most of the duplicates right away. As these tests ended up largely replacing manual searching, there is no guarantee that the set was completely free of duplicates.

was finished in a little under 24 hours. (A sequential, single-machine implementation would most likely need over a week to do to the same encoding task.)

As it was anticipated that there would be a need for multiple runs of the descriptor and classification algorithms, a time/space trade-off was made, in that the encoded files were stored permanently on disk instead of being encoded anew every time. All in all, the encoded files (including the FLAC originals) consumed about 375 GB of storage space; however, the pre-encoding facilitated relatively simple reruns whenever an aspect of the feature extraction implementation was changed.[3]

It should be mentioned that as only the first thirty seconds of each sample were used (as described in section 2.5), storing partial tracks would have reduced the storage requirements significantly; however, this decision was made at a later stage, and keeping full tracks allowed for both increased flexibility and uncovered several performance bugs related to longer tracks.

After all tracks were encoded, the actual work of descriptor generation and test classification commenced. As feature extraction (even with 27 scalar and vector descriptors in all, totaling 648 floating-point values per track) was computationally less intensive than the encoding, no need was seen to use a work distribution system, and all extraction was done on a single dual-core laptop. (The extraction process was not timed, but the final run over all available data took less than 24 hours. As all tracks were pre-encoded, changing the feature extraction code would require only a re-run of this part of the task, as compared to the much more computationally intensive lossy encoding.) The results of the feature extraction were stored in a format designed for rapid loading into the classification engine – in general, each classification run over the full data set completed in under a minute on a single core.

5.4 Correlation and dimensionality reduction

As mentioned in chapter 2, adding more descriptors will not always lead to a better result. Not only will gathering more data require more processing (and to a lesser extent, demand more storage), but too many dimensions will invoke the *curse of dimensionality*, wherein there is too little training data available to properly train the descriptors.

Thus, some sort of *dimensionality reduction* is often desired. Dimensionality reduction takes on two primary forms[17]: *Feature selection*, where certain descriptors (or elements thereof) are simply discarded, and *feature extraction*, where data from multiple descriptors are combined into a (often smaller) set of descriptors. (Gathering the "raw" descriptors from the audio waveform is, of course, a form of feature extraction in itself.)

In this project, a combined feature selection/extraction approach was decided upon. First, the standard and modified MFCC coefficients were split into distinct sets, together with the "simple" descriptors described in chapter 2. (That is, MFCC and F1CC descriptors were never used in the same set. In chapter 6 the exact descriptor subsets used are described in detail.) Also, all possible subsets of the "simple" descriptors (already making up a very small part of the entire descriptor space) were tested.

Second, further dimensionality reduction by use of feature extraction was evaluated. Feature extraction is usually semi-automated, in that there are methods to find new bases for the data set that are optimal or locally optimal in some (method-specific) sense. Two linear, relatively

[3]375 GB is not particularly much as of 2007; however, as mentioned earlier, a sample at least ten times the size of the present data set would be desirable. Although 3-4 TB is far from unattainable in current systems, the cost quickly increases beyond that of simple desktop storage as more data gets added.

simple methods that are in common use are *principal components analysis* (PCA) and *linear discriminant analysis* (LDA)[17]; both will be discussed briefly in the following sections.

When using PCA and LDA, assumptions about the underlying data sets and processing methods are made that may not be valid in the present situation, especially with regard to the rather simplistic test methods. Nevertheless, it will be clear that both methods can be employed for relatively effective dimensionality reduction, although often some human interpretation or other post-correction measures are needed.

5.4.1 Principal components analysis

Principal components analysis, or *PCA*, is a statistical transform or *rotation* of a given data set. PCA essentially yields an input-dependent, square, orthogonal matrix that will transform a vector from the given data set into an *ordered* output vector, where the lower-order components will contribute more (often much more) to the overall variance than the higher-order components. Thus, by discarding or zeroing the higher-order components and retaining the lower-order components, the overall variance is kept approximately unchanged while achieving an overall dimensionality reduction. Furthermore, the vectors of the PCA matrix will often contain indications of what are the "principal components" (hence the name of the method), or true underlying variables, of the input data set.

The PCA algorithm itself will not be explained nor derived here; the reader is instead referred to [18]. PCA was not implemented from scratch; instead, the GNU R statistical analysis package[40] was employed for both PCA and LDA analysis.

The interpretation of the PCA matrix and the ordering of the columns must be done somewhat carefully. In particular, one must not blindly equate variance with *information* from a given component; a descriptor could be very useful for discrimination, yet simply be too small to contribute to the overall vector length. PCA exists in both *unscaled* and *scaled* variants. In scaled PCA, the columns are first pre-normalized to have equal variance, often increasing accuracy in cases where the elements of the input vectors differ in scale. In unscaled PCA, no pre-scaling is performed.

Results of PCA testing, both for unscaled and scaled PCA, can be found in section 6.2.1.

5.4.2 Linear discriminant analysis

As noted, one of the problems of PCA is that its primary metric of variance is not necessarily the best metric for classification. *Linear discriminant analysis* (LDA) is a different, slightly more complex method, designed to maximize *class separation* instead of variance. Unlike PCA, LDA is a *supervised* learning algorithm – the algorithm makes explicit use of the grouping information present in the training set. However, the result of the algorithm is comparable to that of PCA; in particular, a matrix is returned that transforms each input vector onto a different basis, with the more significant elements first.

Like with PCA, LDA will not be explained in detail here – the reader is referred to [18] for a description of the algorithm. However, it should be mentioned that LDA, like PCA, is not ideal for dimensionality reduction in the present situation. In particular, it is designed to discriminate between a *pre-defined* set of groups or classes; usually, in a system designed to discriminate between tracks, it will not be possible to know all possible groups (in this case tracks) beforehand, meaning that there will be more groups in the full data set than in the training set. (The situation would probably be different for a genre classification system, though.) Yet, one could hope that the transformation computed by LDA is useful in a more general sense than only for the groups in the training set.

Results of LDA testing can be found in section 6.2.2.

5.5 K-fold cross-validation

As has been seen, some algorithms need a *training set* for estimation of some set of model parameters, in addition to the testing set itself. (For instance, calculation of the Mahalanobis distance requires estimation of the covariance matrix for the given data.) One could let these two sets be the same; however, one would then risk over-fitting the model to the given data, yielding artificially low error rates for a model that would perform poorly on a different test set. Thus, the data set is usually split into two distinct parts, where one is used for training and one is used for testing. However, a training/testing split necessarily involves a trade-off between testing and training set sizes, as a larger testing set would usually be desirable to increase the precision of the measured results and a larger training set would usually be desirable to get a better estimation of the needed parameters.

K-fold cross-validation is a common statistical technique for alleviating this problem. The data is split into K parts of approximately the same size. Each part is then in turn used as the test set, using a model trained using the other $\frac{K-1}{K}$ of the data; the results from each run are then averaged. Various values for K can be chosen (trading off data set size, computation time and variance of the final results against each other), but in line with recommendations made in [18], $K = 5$ was chosen, and the data set was thus partitioned randomly into five pieces of roughly the same size.[4]

It is important to remember that in the test scenario discussed in section 5.1, the final error rate will be directly influenced by the number of elements in the test set, as all non-references in the test set are checked against all references in the same set. As discussed in section 5.2, there is already too little data for a realistic test, and using only $\frac{1}{5}$ of it at a time would make the problem even worse, artificially lowering the error rates. Thus, an inverted split was decided upon, where only $\frac{1}{5}$ of the data was used for training, and the remaining $\frac{4}{5}$ was used for the testing. This meant that less data was available for model estimation (reducing its quality, probably also reducing the prediction accuracy somewhat), but was deemed an acceptable trade-off.

Note that as reducing the data set by 20% increases the accuracy somewhat, *all* models were assessed using five-fold cross-validation, including the models that had no use for a training set at all (models using Euclidean distance with no dimensionality reduction). This ensured comparability between trained and non-trained models.

5.6 Hypothesis testing

Frequently, it is desirable to know whether using a given configuration (that is, the choice of descriptors, distance metric and dimensionality reduction, if any) C_1 yields better results than a second configuration C_2, in the sense that using C_1 yields higher accuracy than C_2 for the same test set. To this end, a standard one-sided hypothesis test[41] was applied, modeling the success rate of the two tests as coming from two distinct probability distributions X_1 and X_2. If the mean of the first distribution, μ_1, is higher than the mean of the second distribution, μ_2,

[4]The data set was never split across tracks – a FLAC original and the six distorted copies were always kept in the same set.

configuration C_1 is said to be better than C_2. For a set of given measurements, the hypothesis test yields a *p-value* between 0 and 1, indicating the level of significance of the test, where a lower p-value indicates a higher level of statistical significance. (One can also build up a confidence interval of the estimated difference between the means, but when working when success rates close to unity, the confidence interval is typically less interesting.)

Formally, the null hypothesis and alternative hypothesis are:

- $H_0 : \mu_2 \leq \mu_1$ (null hypothesis)
- $H_1 : \mu_2 > \mu_1$ (alternative hypothesis)

By use of the hypothesis test, one can reject the null hypothesis H_0 at some level of significance (depending on the p-value), lending support to the alternative hypothesis H_1, confirming that C_2 is indeed better than C_1. (H_0 should *not*, however, be regarded as confirmed if the hypothesis test yields a high p-value.)

For a single data run, the two distributions can be modeled as binomial (which is approximately normal, due to the number of elements). However, the use of modified five-fold cross-validation described in section 5.5 complicates the discussion, as *five* estimations of the mean were returned instead of one, computed from overlapping data sets. These five values then form a new distribution, whose mean is a different, hopefully better estimator of the true mean of the original distribution. As the variances of these new, five-element sets are unknown, they must be estimated from the data. This leads to the use of the *Student's t-test* instead of the z-test – more precisely, a paired t-test with unknown and unequal variances (the *Welch's t-test*) was used. (The use of a paired test was found appropriate as both configurations were applied to the same five partitions in order.)

Again, the actual statistical calculations were performed in GNU R.

Chapter 6

Results

In this chapter, results from a few different experiments, performed as described in chapter 5, are presented. If not otherwise mentioned, all tests are on the full data set of slightly over 7000 tracks in seven different versions (the original, three MP3 encoded versions and three Vorbis encoded versions), as described in chapter 5.

6.1 Simple descriptors only

As a first test, the "simple" descriptors first introduced in chapter 2 were tested on their own, without combining them with MFCC or F1CC. The primary motivation for these tests was assessing the results of each descriptor (both on its own, and combined with other descriptors), but the results also proved to illustrate more general trends.

As there were only six "simple" descriptors (length, mean/square-ratio, steepness, spectral centroid, number of zero crossings, and number of zero crossings with Schmitt triggering), testing all possible $2^6 - 1 = 63$ subsets of descriptors was possible. Each subset was tested separately, both under Euclidean and Mahalanobis distance, except for the subsets consisting only of a single descriptor (as Euclidean and Mahalanobis distance reduce to the same measure for these purposes). For all tests, the five-fold cross-validation method was used, as discussed in section 5.5.

As the results from 63 tests using Euclidean distance and 57 using Mahalanobis distance would prove too much data to be useful, only a subset of the test data is included here. The chosen subsets are:

- All subsets consisting of a single descriptor are included, the results shown in table 6.1.
- The five best tests overall are also included, and summarized in table 6.2. Note that all of these were run using Mahalanobis distance – for reference, the best single result obtained with Euclidean distance are included in the same table.
- Finally, test data for the subset containing of all six descriptors are included, both under Euclidean and Mahalanobis distance. The results are summarized in table 6.3.

The results show some interesting trends; of particular note are the following points:

- Scalar descriptors are, on their own, of little use for classification – however, when combined relatively good results can be achieved, at least in a data set as small as the one used in these tests.

- Tracks encoded in lower bit rate are in general harder to classify correctly than those in higher bit rate – unsurprisingly, they are less distorted also in an objective sense.
- Even when the covariance matrix was estimated from a training set distinct from the test set, the use of Mahalanobis distance yielded significantly better overall results than Euclidean distance.
- Different subsets of descriptors are better suited to some formats and bit rates than others. In particular, different distortion arises from the use of MP3 than from Vorbis – an extreme case is that the track length alone is a significantly more reliable measure in Vorbis than in MP3, due to MP3 delay and block padding, as discussed in section 2.6.5. It should be mentioned, though, that the inter-format difference can appear at first sight to be larger than it actually is, as lower bit rates were used for Vorbis than for MP3. Nevertheless, the results would seem to support the decision to include multiple formats and bit rates in the data set.
- Adding more descriptors to an existing subset can indeed, as suggested in section 2.3.1, lower the overall recognition rate. The exact results are, however, again dependent on the format and bit rate used.

Interestingly, length and the mean/square ratio are present in all the five best combinations, which suggests that these two descriptors are the best suited among the six tested. Similarly, the centroid is absent from the "top five" list – however, the best combination involving the centroid (actually, the combination of all six descriptors) is not far behind, with an error rate of 14.7%.

6.2 Effect of PCA/LDA on simple descriptors

6.2.1 Principal components analysis

As described in sections 5.4.1 and 5.4.2, PCA and LDA can be used for dimensionality reduction, as an alternative and/or supplement to simple selection. To investigate the feasibility of dimensionality reduction, a five-fold cross-validation test was first run for unscaled and scaled PCA, with the six "simple" descriptors again used as input data. For each partition, the $\frac{1}{5}$ training set was used to estimate the PCA matrix, and the remaining $\frac{4}{5}$ of the data was used as test set. The results, together with the results for the full (ie. non-reduced) descriptor set from the previous section, are summarized in table 6.6 and in figure 6.1. Note that as described in section 5.1, both PCA and LDA were consistently measured under Euclidean distance only.

Examining the rotation matrices, or *loadings*, output by PCA can yield interesting insights in the sample data. One would *a priori* assume that the feature vector consisting of all six descriptors would have in the order of five dimensions: Track length, mean/square ratio, steepness and centroid would be assumed to be independent of each other and the zero crossing descriptors, while the two zero crossing descriptors would be assumed to be somewhat interdependent. Table 6.4 does suggest that it is indeed largely so, but with a slightly different set of interdependencies. (One can also observe the general PCA trend of maximizing variance – indeed, the component with the least weight is the mean/square ratio, although it was found in section 6.1 to be of significant interest. This is, of course, since it takes on values at least three orders of magnitude smaller than any of the other descriptor values.)

	MP3			**Vorbis**			**Total**
	128	**192**	**256**	**64**	**128**	**192**	
Length (L)	83.6%	83.6%	83.6%	17.7%	17.7%	17.7%	50.6%
Mean/square ratio (M)	97.6%	98.0%	98.0%	97.2%	94.7%	92.3%	96.3%
Steepness (S)	97.2%	93.0%	93.2%	99.1%	98.1%	99.3%	96.6%
Centroid (C)	99.8%	99.8%	99.8%	99.6%	99.5%	99.7%	99.7%
Zero crossings (Z)	98.1%	96.4%	96.8%	99.6%	98.6%	99.1%	98.1%
Zero crossings, Schmitt (ZS)	96.4%	93.4%	93.9%	98.9%	97.7%	98.4%	96.4%

Table 6.1: Error rates for simple descriptors used alone (lower is better).

	MP3			**Vorbis**			**Total**
	128	**192**	**256**	**64**	**128**	**192**	
L+M+S+ZS	2.2%	0.6%	0.3%	32.2%	8.7%	0.6%	7.4%
L+M+S+Z+ZS	8.9%	3.9%	2.2%	27.7%	9.8%	3.7%	9.4%
L+M+ZS	2.6%	0.4%	0.3%	40.2%	5.8%	8.4%	9.6%
L+M+S	4.5%	1.1%	0.6%	49.7%	10.8%	8.1%	12.5%
L+M+Z+ZS	11.6%	6.2%	4.0%	36.4%	14.3%	6.2%	13.1%
L+M (Euclidean)	65.1%	65.2%	65.2%	0.4%	0.2%	0.1%	32.7%

Table 6.2: Error rates for best five descriptor subsets under Mahalanobis distance, and best single subset under Euclidean distance. The abbreviations are expanded in table 6.1.

	MP3			**Vorbis**			**Total**
	128	**192**	**256**	**64**	**128**	**192**	
Euclidean	54.3%	29.6%	24.1%	90.7%	62.9%	72.9%	55.7%
Mahalanobis	27.7%	6.4%	2.8%	41.6%	7.9%	2.1%	14.7%

Table 6.3: Error rates for full descriptor set, measured under both Euclidean and Mahalanobis distance.

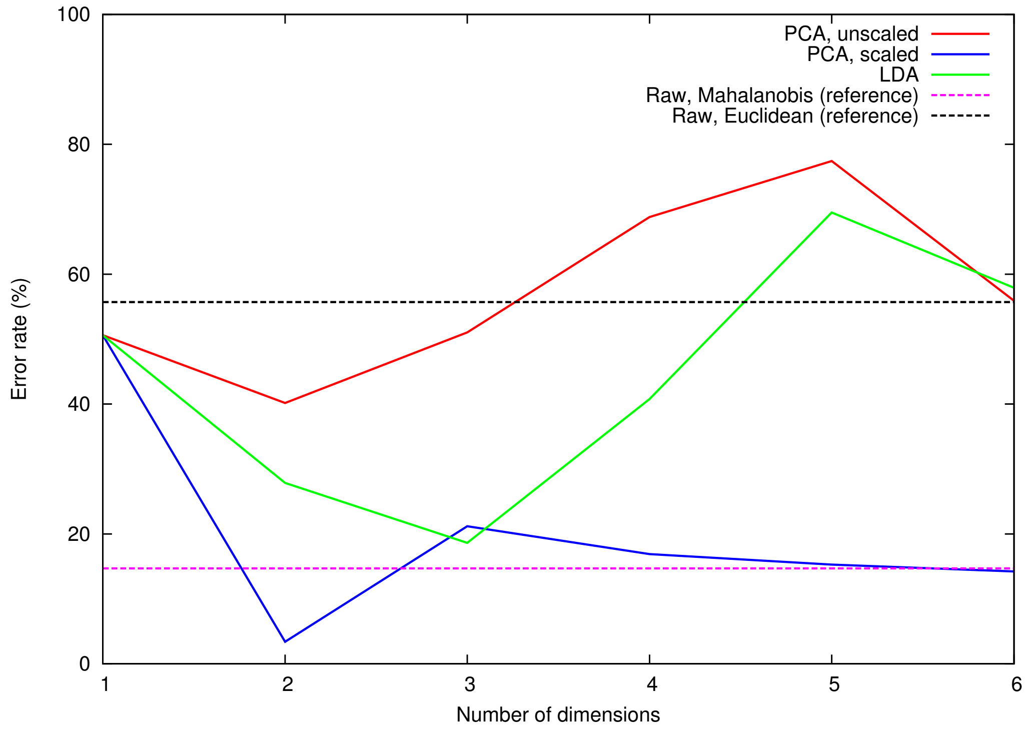

Figure 6.1: Error rate for PCA- and LDA-transformed "simple" descriptors plotted against the number of dimensions, and compared against the full-dimensionality results. Note that all three methods have a certain "sweet spot" in the number of dimensions, above which the error rate increases.

However, when the data set is corrected for scaling, the matrix becomes significantly more convoluted, as is seen in table 6.5; even the track length appears to be correlated with the other measures. Although one should be careful in placing too much weight on the PCA matrix itself (in particular as it is possible to get multiple equivalent rotations through the same basis vectors), the high amount of correlation suggests that adding many of these descriptors will not add very much real dimensionality to the data set, as they are all relatively correlated.

Overall, PCA is not problem-free: In particular, evidence can be seen of the PCA matrix being over-fitted to the test set. As one moves past using two or three of the resulting vectors, the overall accuracy goes down – sometimes very much so. (MP3 and Vorbis are influenced differently by this effect; for instance, from 2D to 3D in the test of unscaled PCA, accuracy on MP3 files goes up while accuracy on Vorbis files goes down.) However, these problems are somewhat less pronounced in the case of scaled PCA, which in general yields better results then unscaled PCA. Scaled PCA also gives somewhat better results than the selection methods used in section 6.1 – however, the results are not conclusive, and one should weigh the extra processing time and estimation overhead involved in using PCA against the relatively small increase in accuracy.

	PC1	PC2	PC3	PC4	PC5	PC6
Length				−0.95	−0.32	
Mean-square ratio						−1.00
Steepness			−0.11	0.32	−0.94	
Centroid			−0.99		0.10	
Zero crossings	0.90	−0.43				
Zero crossings (Schmitt)	0.43	0.90				
Standard deviation	38477.67	8834.12	809.98	117.85	107.36	0.05
Proportion of Variance	0.95	0.05	0.00	0.00	0.00	0.00
Cumulative Proportion	0.95	1.00	1.00	1.00	1.00	1.00

Table 6.4: Principal components of sample partition, unscaled PCA. Matrix coefficients below 0.1 are omitted. Column sign is arbitrary.

	PC1	PC2	PC3	PC4	PC5	PC6
Length		−0.58	0.81			
Mean-square ratio		0.77	0.57	−0.12	0.24	
Steepness	0.54			0.12	−0.30	−0.77
Centroid	0.43	−0.11		−0.86	−0.14	0.21
Zero crossings	0.51	−0.18		0.16	0.82	
Zero crossings (Schmitt)	0.51			0.45	−0.41	0.60
Standard deviation	1.82	1.07	0.96	0.70	0.32	0.18
Proportion of Variance	0.55	0.19	0.15	0.08	0.02	0.01
Cumulative Proportion	0.55	0.74	0.90	0.98	0.99	1.00

Table 6.5: Principal components of sample partition, scaled PCA. Matrix coefficients below 0.1 are omitted. Column sign is arbitrary.

	MP3			**Vorbis**			**Total**
	128	**192**	**256**	**64**	**128**	**192**	
Raw, Euclidean	54.3%	29.6%	24.1%	90.7%	62.9%	72.9%	55.7%
Raw, Mahalanobis	27.7%	6.4%	2.8%	41.6%	7.9%	2.1%	14.7%
Unscaled PCA, 1D	83.6%	83.6%	83.6%	17.7%	17.7%	17.7%	50.6%
Unscaled PCA, 2D	80.1%	80.1%	80.1%	0.4%	0.2%	0.1%	40.2%
Unscaled PCA, 3D	48.7%	19.9%	17.2%	86.4%	58.7%	75.4%	51.0%
Unscaled PCA, 4D	81.5%	69.4%	56.1%	83.1%	59.1%	63.6%	68.8%
Unscaled PCA, 5D	81.2%	58.9%	57.3%	96.0%	80.7%	90.5%	77.4%
Unscaled PCA, 6D	54.1%	29.6%	24.4%	90.6%	63.3%	73.5%	55.9%
Scaled PCA, 1D	83.5%	83.5%	83.5%	17.7%	17.7%	17.7%	50.6%
Scaled PCA, 2D	4.7%	3.7%	3.7%	6.3%	1.3%	0.7%	3.4%
Scaled PCA, 3D	11.3%	3.0%	1.6%	64.4%	20.3%	26.5%	21.2%
Scaled PCA, 4D	11.9%	2.7%	1.1%	58.3%	14.1%	13.3%	16.9%
Scaled PCA, 5D	10.1%	2.0%	0.8%	56.5%	12.5%	9.7%	15.3%
Scaled PCA, 6D	8.7%	1.7%	0.6%	55.1%	11.5%	7.8%	14.2%

Table 6.6: Error rates for five-fold cross-validation test under scaled and unscaled PCA, with varying number of dimensions used, compared to raw results.

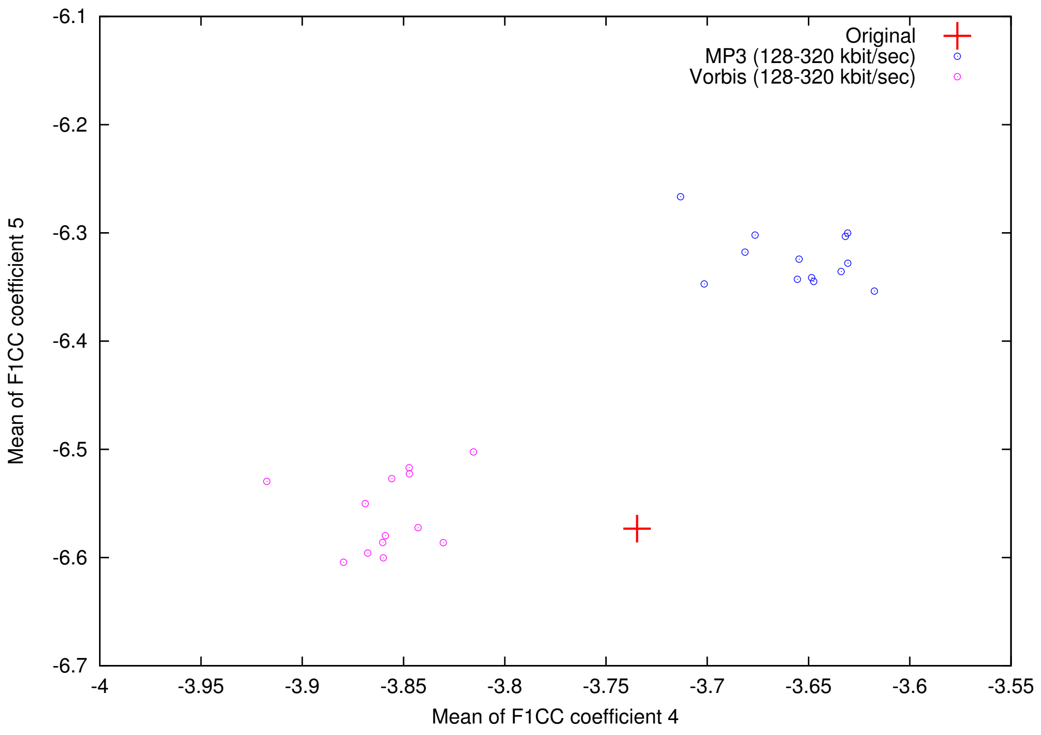

Figure 6.2: Fourth and fifth F1CC coefficients for the same song (The National Bank: "I hear the sparrow sing") encoded in multiple different bit rates. Note the poor fit of the distribution as a whole to a normal distribution.

6.2.2 Linear discriminant analysis

Like with PCA, a five-fold cross-validation test was run using LDA. Except for the use of LDA (which has no unscaled version), the test methodology was identical to that described in section 6.2.1, and will not be repeated here. In figure 6.1 the overall results for PCA and LDA are summarized and compared against the raw results (ie. the full set of "simple" descriptors, without dimensionality reduction.) – the LDA-specific results are summarized in table 6.7.

Looking at the results, one can see similar problems to those of PCA, in that after a certain point, performance gets worse as more dimensions are added. This is another example of the curse of dimensionality – as only seven elements (the original, the three MP3 versions and the three Vorbis versions) are available in each group, the covariance matrix for each group (which is used as input for the LDA algorithm) is insufficiently estimated, with only seven vectors estimating a 36-element matrix.

Results are overall somewhat worse than PCA, but it is difficult to determine if this is because LDA is less suited the given scenario than PCA, or simply due to the small size of the data set (in particular, the training set in this case). It should, however, be mentioned that the variance between the five runs was much higher in the LDA test than in any of the other tests – for instance, the results when using the three first dimensions varied from 70.0% to 92.3% depending on the partition used.

It should also be mentioned that when using LDA, it is assumed that the distribution within each group is approximately normal. Figure 6.2 shows a more detailed view of a single song, where one reference has been encoded into many different bit rates – clearly, the resulting distribution for the two F1CC coefficients selected is not normal, at least not for the given two formats. This distribution discrepancy may be the source of some of the accuracy issues that

	MP3			**Vorbis**			**Total**
	128	**192**	**256**	**64**	**128**	**192**	
Raw, Euclidean	54.3%	29.6%	24.1%	90.7%	62.9%	72.9%	55.7%
Raw, Mahalanobis	27.7%	6.4%	2.8%	41.6%	7.9%	2.1%	14.7%
LDA, 1D	83.6%	83.6%	83.6%	17.7%	17.7%	17.7%	50.6%
LDA, 2D	55.6%	55.5%	55.5%	0.4%	0.2%	0.1%	27.9%
LDA, 3D	10.9%	3.8%	2.7%	55.7%	17.4%	21.2%	18.6%
LDA, 4D	60.6%	40.6%	27.8%	59.5%	32.2%	23.8%	40.8%
LDA, 5D	69.5%	48.4%	46.5%	93.2%	73.6%	85.7%	69.5%
LDA, 6D	55.2%	34.7%	30.6%	89.6%	62.4%	74.8%	57.9%

Table 6.7: Error rates for five-fold cross-validation test using LDA, with varying number of dimensions used.

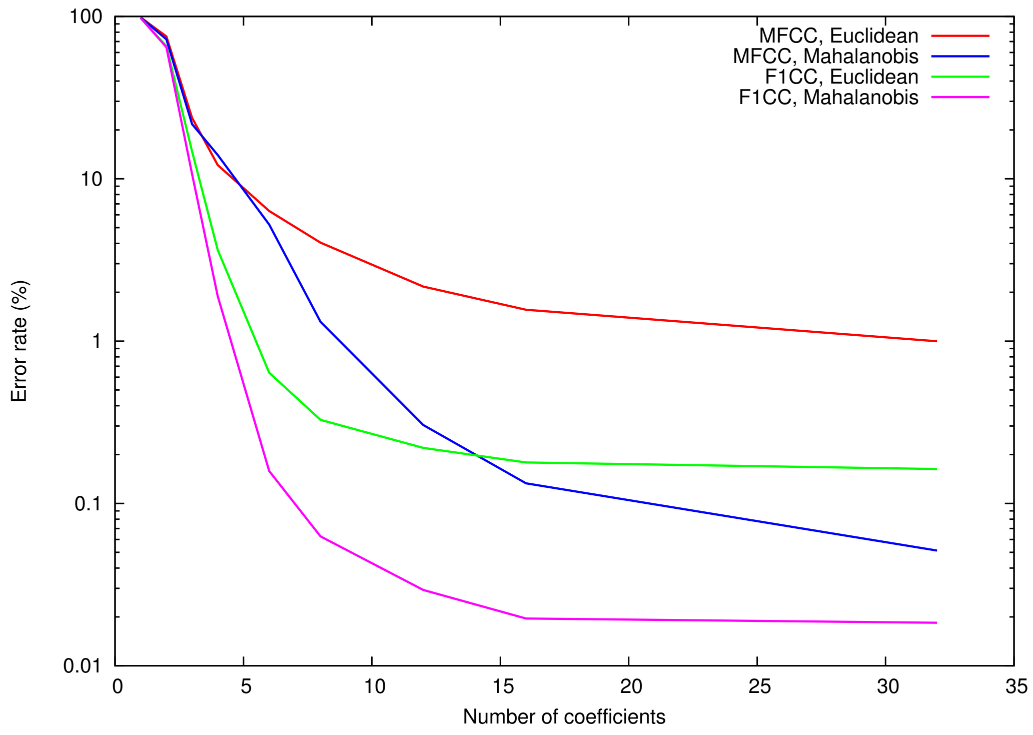

Figure 6.3: Error rate comparison between MFCC and F1CC for N descriptors, logarithmic Y scale.

were observed.

As can be seen, LDA, like PCA, is not an ideal algorithm in the given scenario. However, it should not be completely discounted – even though neither PCA nor LDA were pursued further in this project, they might be more useful in a different scenario, depending on the use case and the available data.

6.3 Comparison of MFCC and F1CC

To compare the merits of MFCC and F1CC (described in chapters 3 and 4, respectively), a series of five-fold cross-validation tests were run with the same test data and differing numbers of descriptors, under both Euclidean and Mahalanobis metrics. The results are summarized in tables 6.8 and 6.9, and figure 6.3.

Many of the same trends as were found for the simple descriptors can be recognized in the given set of results. To confirm these trends statistically, a series of one-sided, paired t-tests were carried out as described in section 5.6, with the following results:

- In all tests, use of F1CC yielded higher recognition rate than the same number of MFCC coefficients under the same distance metric ($p < 10^{-6}$ for all tests). Also, use of four F1CC coefficients yielded better results than six MFCC coefficients, and use of six F1CC coefficients yielded better results than eight MFCC coefficients ($p < 10^{-6}$, Mahalanobis distance).

	MP3			Vorbis			Total
	128	192	256	64	128	192	
MFCC 1D	97.4%	99.1%	99.2%	99.5%	97.1%	99.0%	98.5%
MFCC 2D (E)	61.0%	75.8%	82.1%	97.3%	58.6%	77.4%	75.4%
MFCC 2D (M)	71.2%	68.8%	68.3%	97.7%	60.6%	66.8%	72.2%
MFCC 3D (E)	5.9%	10.9%	15.1%	85.8%	14.1%	10.6%	23.7%
MFCC 3D (M)	8.0%	3.5%	2.9%	86.4%	26.6%	3.0%	21.7%
MFCC 4D (E)	0.5%	1.0%	1.6%	64.4%	4.3%	1.0%	12.1%
MFCC 4D (M)	0.5%	0.1%	0.0%	65.0%	18.5%	0.1%	14.0%
MFCC 6D (E)	0.1%	0.0%	0.0%	36.4%	1.4%	0.0%	6.3%
MFCC 6D (M)	0.1%	0.0%	0.0%	24.0%	7.2%	0.0%	5.2%
MFCC 8D (E)	0.1%	0.0%	0.0%	23.1%	1.0%	0.0%	4.0%
MFCC 8D (M)	0.1%	0.0%	0.0%	4.6%	3.1%	0.0%	1.3%
MFCC 12D (E)	0.0%	0.0%	0.0%	12.2%	0.7%	0.0%	2.2%
MFCC 12D (M)	0.1%	0.0%	0.0%	0.7%	1.1%	0.0%	0.3%
MFCC 16D (E)	0.0%	0.0%	0.0%	8.7%	0.6%	0.0%	1.6%
MFCC 16D (M)	0.0%	0.0%	0.0%	0.3%	0.4%	0.0%	0.1%
MFCC 32D (E)	0.0%	0.0%	0.0%	5.4%	0.5%	0.0%	1.0%
MFCC 32D (M)	0.0%	0.0%	0.0%	0.1%	0.1%	0.0%	0.1%

Table 6.8: Error rates for five-fold cross-validation test using various numbers of MFCC coefficients, under Euclidean (E) and Mahalanobis (M) distance.

	MP3			Vorbis			Total	MFCC
	128	192	256	64	128	192		
F1CC 1D	97.9%	98.6%	98.6%	98.8%	96.6%	95.8%	97.7%	98.5%
F1CC 2D (E)	67.0%	70.1%	72.3%	86.3%	51.8%	46.3%	65.6%	75.4%
F1CC 2D (M)	64.7%	62.2%	62.9%	85.5%	57.8%	55.1%	64.7%	72.2%
F1CC 3D (E)	12.6%	13.6%	14.5%	41.1%	4.3%	2.5%	14.8%	23.7%
F1CC 3D (M)	7.7%	4.5%	4.2%	41.2%	3.9%	2.7%	10.7%	21.7%
F1CC 4D (E)	1.9%	1.9%	2.1%	15.2%	0.5%	0.2%	3.6%	12.1%
F1CC 4D (M)	0.3%	0.1%	0.1%	10.5%	0.3%	0.2%	1.9%	14.0%
F1CC 6D (E)	0.1%	0.1%	0.2%	3.3%	0.1%	0.0%	0.6%	6.3%
F1CC 6D (M)	0.0%	0.0%	0.0%	0.8%	0.1%	0.0%	0.2%	5.2%
F1CC 8D (E)	0.0%	0.0%	0.0%	1.8%	0.1%	0.0%	0.3%	4.0%
F1CC 8D (M)	0.0%	0.0%	0.0%	0.3%	0.0%	0.0%	0.1%	1.3%
F1CC 12D (E)	0.0%	0.0%	0.0%	1.2%	0.1%	0.0%	0.2%	2.2%
F1CC 12D (M)	0.0%	0.0%	0.0%	0.1%	0.0%	0.0%	0.0%	0.3%
F1CC 16D (E)	0.0%	0.0%	0.0%	0.9%	0.1%	0.0%	0.2%	1.6%
F1CC 16D (M)	0.0%	0.0%	0.0%	0.1%	0.0%	0.0%	0.0%	0.1%
F1CC 32D (E)	0.0%	0.0%	0.0%	0.8%	0.1%	0.0%	0.2%	1.0%
F1CC 32D (M)	0.0%	0.0%	0.0%	0.1%	0.0%	0.0%	0.0%	0.1%

Table 6.9: Error rates for five-fold cross-validation test using various numbers of F1CC coefficients, under Euclidean (E) and Mahalanobis (M) distance. The rightmost column is copied from table 6.8 for comparison.

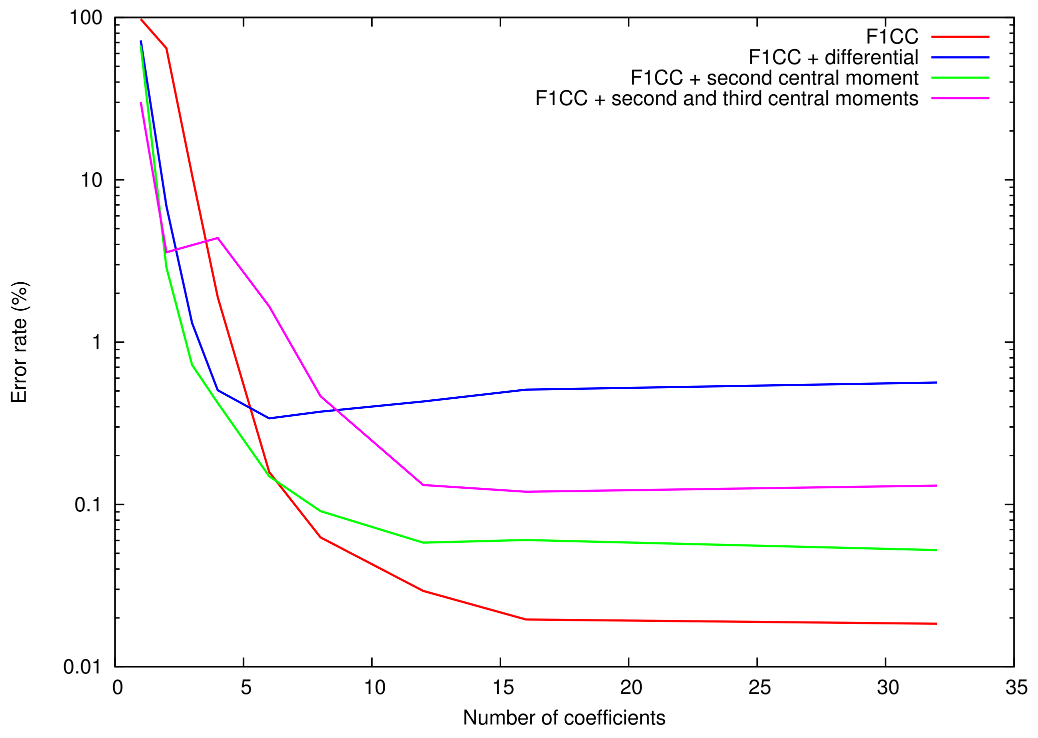

Figure 6.4: Error rate for F1CC with various extra statistical moments, plotted against the number of base coefficients; logarithmic Y scale.

- In almost all tests, use of the Mahalanobis distance yielded higher recognition rate than the Euclidean distance metric ($p < 10^{-4}$). The sole exceptions were 1D cases (where the two metrics will yield identical results), and the case of 4D MFCC, where curiously enough, Euclidean metric was better (also with $p < 10^{-4}$).
- Adding more coefficients does in general improve accuracy; however, less is gained for each extra coefficient. There is even a marginal *negative* difference in accuracy from 16 to 32 coefficients, regardless of MFCC/F1CC choice and distance metric – however, the statistical significance for this result (difference between 16 to 32 F1CC coefficients, under Mahalanobis distance) was weaker than in most of the other tests ($p = 0.027$).

In general, using 8-16 F1CC coefficients measured under Mahalanobis distance would seem to give the best accuracy/data amount trade-off for this testing set.

6.4 Effect of F1CC derivatives and central moments

As explained in section 3.2, MFCC is *homomorphic* – that is, convolution (and equivalently, filtering) will only influence the mean of the coefficient vector. In other words, derivatives and central moments can be expected to remain approximately unchanged. Thus, tests were performed to see how the accuracy rate would be influenced by incorporating derivative and central moment data in the feature vector, which would presumably be less influenced by the filtering and other effects introduced by lossy compression.

As shown in the previous section, the use of F1CC yielded higher accuracy than MFCC, and thus, it was decided to run these tests using F1CC only. Even though it has not been proved that F1CC is a homomorphic transform (it most likely is not, even if based on a homomorphic

	MP3			**Vorbis**			**Total**	**F1CC**
	128	**192**	**256**	**64**	**128**	**192**		
F1CC+Δ 1D	80.9%	82.1%	83.5%	83.1%	53.6%	43.1%	71.1%	98.5%
F1CC+Δ 2D	12.2%	8.4%	8.2%	11.7%	0.6%	0.3%	6.9%	72.2%
F1CC+Δ 3D	2.5%	1.5%	1.5%	2.3%	0.0%	0.0%	1.3%	21.7%
F1CC+Δ 4D	1.1%	0.6%	0.6%	0.7%	0.0%	0.0%	0.5%	14.0%
F1CC+Δ 6D	0.6%	0.6%	0.6%	0.1%	0.0%	0.0%	0.3%	5.2%
F1CC+Δ 8D	0.8%	0.6%	0.6%	0.1%	0.0%	0.0%	0.4%	1.3%
F1CC+Δ 12D	0.9%	0.8%	0.7%	0.1%	0.0%	0.0%	0.4%	0.3%
F1CC+Δ 16D	1.1%	0.9%	0.9%	0.1%	0.0%	0.0%	0.5%	0.1%
F1CC+Δ 32D	1.1%	1.1%	1.1%	0.1%	0.0%	0.0%	0.6%	0.1%
F1CC+C^2 1D	74.9%	79.4%	80.9%	73.5%	67.1%	57.4%	72.2%	98.5%
F1CC+C^2 2D	1.7%	1.0%	1.0%	12.4%	0.8%	0.5%	2.9%	72.2%
F1CC+C^2 3D	0.4%	0.1%	0.1%	3.1%	0.5%	0.2%	0.7%	21.7%
F1CC+C^2 4D	0.4%	0.0%	0.0%	1.6%	0.4%	0.1%	0.4%	14.0%
F1CC+C^2 6D	0.1%	0.0%	0.0%	0.5%	0.1%	0.1%	0.1%	5.2%
F1CC+C^2 8D	0.1%	0.0%	0.0%	0.3%	0.1%	0.0%	0.1%	1.3%
F1CC+C^2 12D	0.1%	0.0%	0.0%	0.2%	0.0%	0.0%	0.1%	0.3%
F1CC+C^2 16D	0.1%	0.0%	0.1%	0.2%	0.0%	0.0%	0.1%	0.1%
F1CC+C^2 32D	0.1%	0.1%	0.1%	0.1%	0.0%	0.0%	0.1%	0.1%
F1CC+C^2+C^3 1D	47.2%	46.3%	47.0%	32.1%	32.1%	21.9%	37.8%	98.5%
F1CC+C^2+C^3 2D	4.6%	4.5%	4.5%	5.1%	1.8%	1.0%	3.6%	72.2%
F1CC+C^2+C^3 3D	5.0%	4.1%	4.0%	7.0%	2.4%	1.2%	4.0%	21.7%
F1CC+C^2+C^3 4D	5.8%	3.6%	3.5%	8.5%	3.2%	1.7%	4.4%	14.0%
F1CC+C^2+C^3 6D	1.9%	0.7%	0.5%	4.7%	1.5%	0.7%	1.7%	5.2%
F1CC+C^2+C^3 8D	0.4%	0.1%	0.1%	1.4%	0.4%	0.2%	0.5%	1.3%
F1CC+C^2+C^3 12D	0.2%	0.1%	0.1%	0.2%	0.1%	0.0%	0.1%	0.3%
F1CC+C^2+C^3 16D	0.2%	0.1%	0.1%	0.2%	0.1%	0.0%	0.1%	0.1%
F1CC+C^2+C^3 32D	0.3%	0.2%	0.2%	0.2%	0.0%	0.0%	0.1%	0.1%

Table 6.10: Error rates for five-fold cross-validation test using various numbers of F1CC coefficients and their differentials (Δ), second order central moments (C^2) and third order central moments (C^3), all under Mahalanobis distance. The rightmost column is copied from table 6.9 for reference.

MFCC variant, as the tone curves used are nonlinear in nature), it is similar enough to MFCC that it would not seem unreasonable that it is *approximately* homomorphic.

Three configurations were tested for various numbers of dimensions, all with added descriptors over the F1CC tests presented in the previous section. The three configurations consisted of, for each coefficient:

- The mean, plus the delta mean (the mean of the absolute value of the change from one MFCC frame to the next).
- The mean, plus the second central moment, the variance.
- The mean, plus the second *and* third central moments.

More information about each of these descriptors can be found in section 3.4.

The results can be found in table 6.10, which contains results for all three tests. Note that only results for the Mahalanobis distance are presented, as the general trends proved not to be too different for Euclidean distance, and Mahalanobis distance was earlier found to be an overall better measure. The same results are also visualized in figure 6.4.

Comparing the data in table 6.10 against the reference results from table 6.9, it is clear that adding delta or variance information for F1CC coefficients is beneficial when using few coefficients. However, simply adding more F1CC mean coefficients gives a similar result, and as more coefficients are added, the overall accuracy is marginally worse than when only using the regular mean. (This is most likely related to the fact that the covariance matrix for the largest test is as big as 96x96 = 9216 elements, which is poorly estimated at the given training set size.) In general, however, the results are somewhat inconclusive – as has been seen, adding more coefficients yields only marginal improvements after a while, and for a bigger set, the extra information provided by non-mean data might be beneficial.

6.5 Combining simple descriptors with F1CC

Finally, tests were run combining "simple" descriptors with F1CC. In line with results in previous sections, the selected configuration consisted of length, mean/square ratio and a variable number of F1CC coefficients, measured under Mahalanobis distance, and without any dimensionality reduction. Although more configurations could be tested, in particular with regard to the selection of the simple descriptors, this was believed to be relatively representative of what could be achieved by combining elements from both classes.

The results can be found in table 6.11, and are summarized in figure 6.5. Note that the plots should be interpreted with caution, as the general error rates at high dimensionality are too low for accurate measurements. In particular, at 32 dimensions, there was not sufficient statistical evidence to show that the results for F1CC with and without "simple" descriptors were different (two-sided paired Welch's t-test, $p = 0.079$).

All in all, however, it would appear that supplementing the F1CC data with the relatively simple measures of track length and mean/square ratio yields a reasonable increase in overall accuracy.

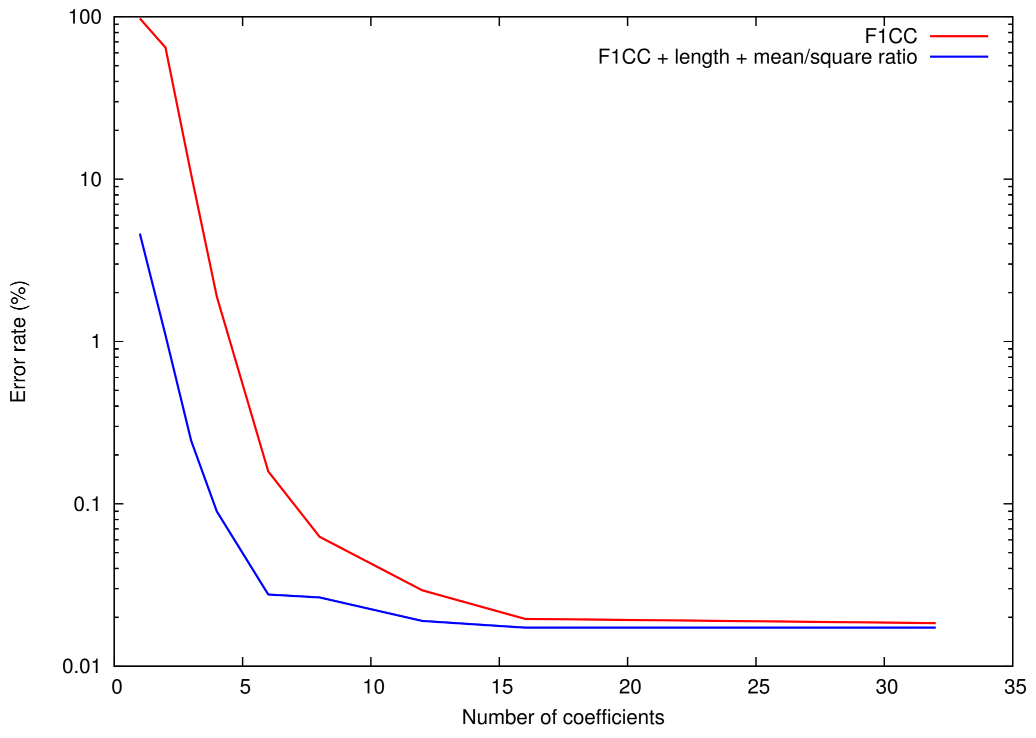

Figure 6.5: Error rate for F1CC, both alone and supplemented with length and mean/square ratio, plotted against the number of F1CC coefficients; logarithmic Y scale.

	MP3			**Vorbis**			**Total**	**F1CC**
	128	**192**	**256**	**64**	**128**	**192**		
L+M+F1CC 1D	36.5%	43.6%	46.2%	51.9%	14.0%	9.6%	33.6%	98.5%
L+M+F1CC 2D	0.2%	0.1%	0.1%	5.6%	0.3%	0.1%	1.1%	72.2%
L+M+F1CC 3D	0.0%	0.0%	0.0%	1.4%	0.0%	0.0%	0.2%	21.7%
L+M+F1CC 4D	0.0%	0.0%	0.0%	0.5%	0.0%	0.0%	0.1%	14.0%
L+M+F1CC 6D	0.0%	0.0%	0.0%	0.1%	0.0%	0.0%	0.0%	5.2%
L+M+F1CC 8D	0.0%	0.0%	0.0%	0.1%	0.0%	0.0%	0.0%	1.3%
L+M+F1CC 12D	0.0%	0.0%	0.0%	0.1%	0.0%	0.0%	0.0%	0.3%
L+M+F1CC 16D	0.0%	0.0%	0.0%	0.1%	0.0%	0.0%	0.0%	0.1%
L+M+F1CC 32D	0.0%	0.0%	0.0%	0.1%	0.0%	0.0%	0.0%	0.1%

Table 6.11: Error rates for five-fold cross-validation test using length, mean/square ratio and various numbers of F1CC coefficients, measured under Mahalanobis distance. The rightmost column is copied from table 6.9 for reference.

Chapter 7

Discussion

7.1 Overall system performance

As mentioned earlier, the test set of 7242 tracks is too small for a realistic test of real-life application performance, where the number of reference tracks can number several millions. Yet, a comparison to existing systems may give an indication of the overall scaling ability.

In general, the system presented here is relatively simplistic – often, when there was a choice between simplicity and sophistication, the simpler solutions were chosen. (In particular, the decision to extract statistical moments from the MFCC/F1CC coefficients, as described in section 3.4, is probably less than ideal for accuracy.) However, even a simple solution may scale surprisingly well – the TRM system has been criticized for being overly simplistic (being mostly based on "simple" descriptors such as the ones described in chapter 2, in addition to a few spectral measures)[34], yet the MusicBrainz database contains over 8.8 million TRMs[7] and still works reasonably well.[1]

The scaling discussion is somewhat complicated by lack of reliable accuracy measures for practical systems. Judging from MusicBrainz statistics found in [7], it appears Relatable's TRM and MusicIP's PUID system would have about the same collision rate (about 14% of all TRMs or PUIDs in MusicBrainz' database point to more than one distinct track) – however, at the time of writing, the database contains about 8.8 million distinct TRMs and only 1.7 million distinct PUIDs, which makes the comparison somewhat unfair. Tuneprint claims 100% matching accuracy in a test against a database of 100,000 reference tracks[35], with descriptors reminding somewhat of F1CC without the triangular bands and DCT – however, with only 73 samples in the test set, the confidence interval would seem somewhat broad. For other systems, large databases are cited to be used in production (such as Shazam, with 1.8 million tracks[42]), but no numbers for overall matching accuracy are given. Also, error rates will vary naturally with the amount of distortion in the test set – again, a complicating factor.

When discussing scaling, it is also important to consider the amount of processing power needed for each query, in particular on the server side. (The processing time needed for the actual feature extraction is less important here, as it will usually be done on the client side.) For instance, the strategy of comparing against each reference in turn, as described in section 5.1,

[1]This is not to say that TRM scaled *comfortably* to these levels – TRM has ongoing scaling problems, both computationally and performance-wise, which has lead MusicBrainz to start a long-transition towards MusicIP's PUID system[31]. Yet, the TRM system is far from being unusable even for a data set of this size.

is not very efficient on a larger scale, and should ideally be replaced by a more sophisticated method for a practical system. However, given the size of the data set used, computational scaling was not a primary concern.

7.2 Applicability of dimensionality reduction

At first sight, dimensionality reduction appears very attractive: Not only does it reduce the amount of data that needs to be stored, transferred and searched, it also comes with a promise of decorrelation and increased resilience to noise. However, as has been seen, there are multiple problems with both PCA and LDA in practice.

First, the optimal decorrelation matrix for a data set can be difficult to estimate from only a part of it, leading to possible overtraining unless great care is taken to make the selected part representative (and large enough). In other words, the curse of dimensionality discussed in 2.3.1 can be reintroduced in the very methods that were supposed to alleviate these problems.

Second, every method of dimensionality reduction will necessarily depend on a given set of assumptions. Violations of these assumptions can lead to poor or unpredictable results, depending on the method and the concrete nature of the violations. In the present case, many of these assumptions are poorly met by the input data:

- For PCA, it was seen in section 5.4.1 that the notion of variance as the primary information-bearing measure was ill-suited to the classification scheme. While the issue is somewhat alleviated by using scaled PCA instead of unscaled PCA, the fundamental problem remains.

- As described in section 5.4.2, LDA is designed to separate between N pre-defined classes (and usually, N is less than the number of dimensions in the data), all known at the time of matrix estimation. As has been seen, reusing the resulting matrix to also separate *other* classes (actually, *many* other classes) from each other can lead to less than ideal results.

- Finally, the entire assumption of approximate normality is most likely false, as discussed in section 6.2.2 – in particular, the various compressed versions of each track will not be identically distributed, as they are encoded in different bit rates and formats.

Especially in the light of the overall good results achieved by using Mahalanobis distance (which compensates for much of the correlation seen) instead of Euclidean distance, one should consider carefully whether the negative and somewhat unpredictable effects of PCA and LDA are outweighed by the gains in the particular situation at hand.

It should be mentioned that in a situation with more descriptors than in the given test setup (for instance, if much more temporal information was kept, as described in section 3.4), dimensionality reduction might again be more attractive, as the combined fingerprint would simply become impractically large without it. In such cases, one would also usually have more data overall to work with, depending on the internal structure of the descriptors – for instance, if one represented each 200ms slice of audio by an N-dimensional vector, a thirty-second audio clip would yield 150 vectors, as opposed to only a handful when using statistical moments. Again, the use of PCA/LDA should be carefully considered for each case, in particular with regard to the descriptors extracted from each track.

7.3 Descriptor layering

The various forms of description of audio, and in particular music, can be organized in a layered fashion. Although there is no standardized segmentation, these layers can be described approximately as follows:

- On the least sophisticated levels of description is the purely physical, time-based perception. All signals can be described in a physical fashion, most obviously in the case of "real" instruments with vibrating strings or air moving through a physical system, but also for electronically generated audio. Analysis of time-based representations usually involves considering only very short time segments at a time. Most of the "simple" descriptors introduced in chapter 2 utilize only time-based information.
- One layer up lies spectral measures of various sophistication, such as MFCC and F1CC introduced in chapters 3 and 4. Information at this level often corresponds better to the workings of the human auditory system (including the outer and inner ear) than information from the first level. When analyzing audio on a short-time spectral scale, the signal is usually analyzed in blocks of about 10-50ms.
- At the next layer, the difference between arbitrary audio and music becomes increasingly important, as it contains descriptors such as single tones, time intervals (which in turn lead to rhythm), timbre, chords, and short melody lines. Information on this level is usually processed in blocks of up to a few seconds, corresponding to the human short-term memory.
- Finally, the uppermost layer corresponds to overall artistic ability. This layer incorporates long-term musical structure and development, overall interpretation of the piece and even measures as subjective as musical quality.

This "layer stack" also describes the relationship between the different fields of musical study – from pure physics and signal processing at the bottom of the stack, ultimately progressing towards psychology at the top. As human perception frequently is centered around the upper layers, it would seem beneficial for a MIR system to utilize the information on these levels – indeed, as has been shown in the experiments in previous chapters, descriptors utilizing spectral information are significantly better choices than time-based descriptors from the bottom layer.

Unfortunately, general knowledge about the upper layers is still incomplete, and as extracting information on each layer commonly depends on having accurate information from the lower layers, the complexity is increased and data extraction becomes increasingly difficult. Limited knowledge about several stages of the human auditory system further increases these difficulties.

It should also be mentioned that as one moves towards the top of the stack, descriptors and information become less objective, and more subject to the individual listener's preferences and experiences. (Considering a measure such as "musical quality" for a given piece, one can expect to find a great deal of disagreement in the general population.) This can pose a challenging task to a MIR system, but also opportunities for delivering results better tuned towards the individual. What the optimal layer for a given MIR application is, remains an open question.

Chapter 8

Conclusion

8.1 Conclusion

As has been seen, it is possible to build a personal music recognition system with reasonable accuracy even with relatively simple descriptors and methods. However, more sophisticated descriptors and measures do in general deliver better results – in particular, use of F1CC gives markedly better results in this case than MFCC does, and Mahalanobis distance likewise over Euclidean distance. However, the use of dimensionality reduction methods such as PCA or LDA should be carefully considered in each case – while effective in some cases, their use can also *reduce* matching accuracy significantly in others. Also, overall complexity is increased.

All in all, F1CC, combined with a few "simple" descriptors such as mean/square ratio or track length, appears to be a promising choice for a system of this form. However, extracting statistical moments from the F1CC data might not be the best post-processing, and depending on the use case, measures better preserving the temporal structure present in music should be considered.

8.2 Further work

As the use of F1CC has shown promising results, it would be natural to consider its use in other situations than the one presently described – in particular to assess whether accounting for auditory masking makes for a better descriptor model, or if its increased usefulness in the present use case stems primarily from the ability to better "ignore" artifacts created by lossy encoding. One might also want to experiment with different methods of post-processing after the floor computation.

Furthermore, larger-scale tests would be interesting – if a given MIR system is capable of working well without uncompressed originals, it should not be impossible to run tests with 100,000 tracks or more, possibly yielding interesting insights into larger-scale behavior. However, with larger data sets, more sophisticated searching and processing algorithms would also probably be needed, eventually moving into the realm of distributed computing.

Finally, assuming an appropriate feature extraction system, the use of more complex, human-oriented descriptors (such as tempo/rhythm, instrument selection, chord structure or melodic contour) could make a very worthwhile addition to a music recognition system. However, as always, the use of a given descriptor must be considered in the light of implementation complexity, accuracy, robustness and performance.

Bibliography

[1] Vorbis development, status & patent issues. Hydrogenaudio discussion forum, September 2003. `http://www.hydrogenaudio.org/forums/index.php?showtopic=13531`, accessed 2007-05-12.

[2] N. Ahmed, T. Natarajan, and K. R. Rao. Discrete cosine transform. *IEEE Transactions on Computers*, pages 90–93, January 1974.

[3] Richard Ernest Bellman. *Dynamic Programming.* Princeton University Press, 1957.

[4] Richard H. Ehmer. Masking patterns of tones. *Journal of the Acoustical Society of America*, 31:1115–1120, August 1959.

[5] Gianpaolo Evangelista and Sergio Cavaliere. Event synchronous wavelet transform approach to the extraction of musical thumbnails. In *Proceedings of the 8th International Conference on Digital Audio Effects (DAFx'05)*, pages 232–236, September 2005.

[6] Arthur Flexer, Elias Pampalk, and Gerhard Widmer. Hidden Markov models for spectral similarity of songs. In *Proceedings of the 8th International Conference on Digital Audio Effects (DAFx'05)*, pages 131–136, September 2005.

[7] MetaBrainz Foundation. MusicBrainz database statistics. Available online at `http://musicbrainz.org/stats.html`, accessed 2007-05-28.

[8] Xiph.org Foundation. Ogg Vorbis: Fidelity measurement and terminology discussion. Available online at `http://xiph.org/vorbis/doc/vorbis-fidelity.html`, accessed 2007-05-12.

[9] Xiph.org Foundation. Ogg Vorbis stereo-specific channel coupling discussion. Available online at `http://xiph.org/vorbis/doc/stereo.html`, accessed 2007-05-12.

[10] Xiph.org Foundation. Vorbis I specification. Available online at `http://www.xiph.org/vorbis/doc/Vorbis_I_spec.pdf`, accessed 2007-05-12.

[11] Niechcial Francis. MPC vs VORBIS vs MP3 vs AAC at 180 kbps. Hydrogenaudio discussion forum, August 2005. Available online at `http://www.hydrogenaudio.org/forums/index.php?showtopic=36465`, accessed 2007-05-12.

[12] Matteo Frigo and Steven G. Johnson. The design and implementation of FFTW3. *Proceedings of the IEEE*, 93(2):216–231, February 2005.

[13] Asif Ghias, Jonathan Logan, David Chamberlin, and Brian C. Smith. Query by humming – musical information retrieval in an audio database. *Proceedings of ACM Multimedia 95*, November 1995.

[14] Pascutto Gian-Carlo. foosic - the living music database. `http://foosic.org/libfooid.php`, accessed 2007-05-16.

[15] Jerry D. Gibson, Toby Berger, Tom Lookabaugh, Dave Lindbergh, and Richard L. Baker. *Digital Compression for Multimedia: Principles & Standards*. Morgan Kaufmann Publishers, Inc., 1998.

[16] Antonin Guttman. R-trees: A dynamic index structure for spatial searching. In *Proceedings of the 1984 ACM SIGMOD International Conference on Management of Data*, pages 44–57.

[17] Isabelle Guyon and André Elisseeff. An introduction to variable and feature selection. *Journal of Machine Learning Research*, 3:1157–1182, March 2003.

[18] Trevor Hastie, Robert Tibshirani, and Jerome Friedman. *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*. Springer, 2001.

[19] Tammo Hinrichs. The workings of fr-08's sound system, part 4: Let's talk about synthesizers. *PAiN Magazine*, February 2002.

[20] Xuedong Huang, Alex Acero, and Hsiad-Wuen Hon. *Spoken Language Processing*. Prentice-Hall, 2001.

[21] Anssi Klapuri and Manuel Davy, editors. *Signal Processing Methods for Music Transcription*. Springer Science+Business Media LLC, 2006.

[22] Naoko Kosugi, Yuichi Nishihara, Tetsuo Sakata, Masashi Yamamuro, and Kazuhiko Kushima. A practical query-by-humming system for a large music database. *Proceedings of the eighth ACM international conference on Multimedia*, pages 333–342, 2000.

[23] Jean Laroche. Estimating tempo, swing and beat locations in audio recordings. *2001 IEEE Workshop on the Applications of Signal Processing to Audio and Acoustics*, pages 135–138, October 2001.

[24] Gaoyong Luo. Ultra low delay wavelet audio coding with low complexity for real time wireless transmission. *Proceedings of 2005 International Symposium on Intelligent Signal Processing and Communication Systems*, pages 741–744, December 2005.

[25] Prasanta Chandra Mahalanobis. On the generalised distance in statistics. In *Proceedings of the National Institute of Science of India*, volume 12, pages 49–55, 1936.

[26] John Makhoul. A fast cosine transform in one and two dimensions. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, 28(1):27–34, February 1980.

[27] Sebastian Mares. Results of public, multiformat listening tests at 128kbps, December 2005. Available online at `http://www.listening-tests.info/mf-128-1/results.htm`, accessed 2007-05-12.

[28] Sebastian Mares. Results of public, multiformat listening tests at 48kbps, November 2006. Available online at `http://www.listening-tests.info/mf-48-1/results.htm`, accessed 2007-05-12.

[29] Stephen McAdams and Albert Bregman. Hearing musical streams. *Computer Music Journal*, 3:26–63, 1979.

[30] Chris Montgomery. Re: Getting masked FFT data out of libvorbisenc. Communication on `vorbis-dev` mailing list, April 2007. Available online at `http://lists.xiph.org/pipermail/vorbis-dev/2007-April/018811.html`, accessed 2007-05-12.

[31] MusicIP. Open Fingerprint Architecture whitepaper, version 1.0, March 2006. Available online at `http://www.musicip.com/dns/files/Open_Fingerprint_Architecture_Whitepaper_v1.pdf`, accessed 2007-05-16.

[32] John G. Proakis and Dimitris G. Manolakis. *Digital Signal Processing*. Prentice-Hall, third edition, 1996.

[33] Relatable. TRM$^{\text{TM}}$: The universal barcode for music and media from Relatable®. Available online at `http://www.relatable.com/tech/trm.html`, accessed 2007-05-28.

[34] Geoff Schmidt. Acoustic fingerprinting. Communication on the `mb-devel` mailing list, October 2005. Available online at `http://lists.musicbrainz.org/pipermail/musicbrainz-devel/2005-October/001432.html`, accessed 2007-05-25.

[35] Geoff R. Schmidt and Matthew K. Belmonte. Scalable, content-based audio identification by multiple independent psychoacoustic matching. *Journal of the Audio Engineering Society*, 52(4):366–377, April 2004.

[36] Inc. Sequential Circuits. The MIDI specification, August 1983.

[37] Julius Smith III and Xavier Serra. PARSHL: An analysis/synthesis program for non-harmonic sounds based on a sinusoidal representation. Technical report, Center for Computer Research in Music and Acoustics (CCRMA), Department of Music, Stanford University, 1987.

[38] Thomson. MP3 patent portfolio. Available online at `http://www.mp3licensing.com/patents/index.html`, accessed 2007-05-12.

[39] Jean-Marc Valin and Christopher Montgomery. Improved noise weighting in CELP coding of speech – applying the Vorbis psychoacoustic model to Speex. *Proceedings of the AES 120th Convention*, May 2006.

[40] W. N. Venables, D. M. Smith, and the R Development Core Team. An introduction to R, version 2.5.0, April 2007. `http://cran.r-project.org/doc/manuals/R-intro.pdf`, accessed 2007-05-12.

[41] Ronald E. Walpole, Raymond H. Myers, Sharon L. Myers, and Keying Ye. *Probability & Statistics for engineers and scientists*. Prentice-Hall, seventh edition, 2002.

[42] Avery Wang. An industrial-strength audio algorithm. Presentation from ISMIR 2003, October 2003.

[43] Tim Westergren. The Music Genome Project. Available online at `http://www.pandora.com/mgp.shtml`, accessed 2007-05-30.

[44] Gerhard Widmer, Simon Dixon, Werner Goebl, Elias Pampalk, and Asmir Tobudic. In search of the Horowitz factor. *AI Magazine*, pages 111–130, fall 2003.

Index of terms

ATH Absolute Threshold of Hearing, the (frequently-dependent) minimum sound level of a pure tone an average ear can hear, assuming an otherwise silent environment.

CBR Constant Bit Rate, where a multimedia clip is encoded is the same bit rate at all times (except over very short, strictly limited time periods). Contrast VBR.

codec COder/DECoder (originally an acronym, but now usually written in lowercase), a device or program capable of digital encoding or decoding of a signal. Most codecs are *lossy*, in that an encoding/decoding cycle leaves an imperfect representation of the original signal – while lossless codecs exist, lossy codecs are typically much more bandwidth efficient, making the (often imperceptible) degradation a reasonable trade-off.

DFT Discrete Fourier Transform, a common transformation of discrete signals from a time- to a frequency-based representation.

FFT Fast Fourier Transform, a particularly efficient implementation of the DFT. (Actually, several different FFT algorithms exist; the term "FFT" can refer to any one of them.) Computation of the FFT of an n-element array requires only $O(n \log n)$ operations, while a naïve implementation of the DFT would need $O(n^2)$.

FFTW The "Fastest Fourier Transform in the West", a free FFT library developed by Matteo Frigo and Steven G. Johnson.[12]

FLAC Free Lossless Audio Codec, a codec for lossless audio compression.

LAME LAME Ain't an MP3 Encoder (an example of a recursive acronym), a free, high-quality MP3 encoder. The name remains from a time when LAME was distributed only as a patch set against the ISO MP3 encoder, and thus was not a complete encoder in itself; however, in these days no ISO source remains in LAME, and thus, the acronym is now a misnomer.

LDA Linear Discriminant Analysis, a statistical technique for finding linear combinations of a given data set that best separate two or more classes from each other.

MIDI Musical Instrument Digital Interface, a digital music communications protocol, standardized in 1983 and today almost universally supported by electronic music equipment.

MIR Music Information Retrieval (or sometimes, Multimedia Information Retrieval), the scientific field concerned with retrieving information from music.

MP3 MPEG-1 audio layer III, a highly popular audio codec.

Ogg An open container format for digital multimedia. Typically, Ogg streams contain Vorbis, FLAC or Speex (an open speech codec) audio. (Ogg is not an acronym, and as such is not fully capitalized.)

PCA Principal Components Analysis, a statistical transform for decorrelation and dimensionality reduction.

PCM Pulse Code Modulation, the most common representation of uncompressed audio, representing a signal in uniformly sampled, linearly quantized form.

PUID Portable Unique IDentifier, an ID used in the Open Fingerprint Architecture from MusicIP.

SPL Sound Pressure Level, a physical measure of sound strength. Usually measured in decibels (dB), where 0 dB is equivalent to 20 micropascals.

SVD Singular Value Decomposition, a factorization of real or complex matrices especially important in signal processing and statistics.

TRM TRM Recognizes Music, a semi-open commercial fingerprinting system from Relatable. TRM can also refer to the ID given to each distinct track (as determined by the TRM server) in the TRM system.

VBR Variable Bit Rate, a technique where a multimedia encoder can use different bit rates in different parts of the material to increase overall quality while keeping the total bit rate roughly constant. Contrast CBR.

Vorbis A free, lossy audio codec, intended to serve as a patent-free, next-generation alternative to MP3.

Appendix A

Programming environment

In the making of this project, a wide range of software, mostly free software, was used. The final list of software eventually used in or by the finished code is:

- Debian GNU/Linux (`http://www.debian.org/`), versions 3.1 ("sarge") and 4.0 ("etch"), was used as the base operating system for nearly all tasks.
- The GNU C Compiler Collection, GCC (`http://gcc.gnu.org/`), version 4.1.2, was used for compiling C and C++ code.
- The LAME MP3 encoder (`http://lame.sourceforge.net/`), version 3.97, was used for MP3 encoding.
- libvorbis (`http://www.vorbis.com/`), version 1.1.2, was used for both Vorbis encoding, Vorbis decoding and as the base for the F1CC tone-tone masking curve calculations.
- Underbit MAD, MPEG Audio Decoder (`http://www.underbit.com/products/mad/`), version 0.15.1b, was used for MP3 decoding.
- FLAC, Free Lossless Audio Codec (`http://flac.sourceforge.net/`), version 1.1.4, was used for lossless encoding and subsequent decoding of audio files.
- The FFTW library (`http://www.fftw.org/`), version 3.1.2, was used for fast DFT and DCT calculations.
- The Perl programming language (`http://www.perl.org/`), version 5.8.8, was used for various prototyping tasks and smaller processing tasks, including the job control system for distributed encoding.
- The PostgreSQL relational database (`http://www.postgresql.org/`), version 8.2.3, was used as data storage for the processing queue in the distributed encoding effort.
- The Bazaar-NG version control system (`http://www.bazaar-vcs.org/`), versions 0.12 through 0.16, was used for revision control for both of the code and the report.

Also, the following software was used for the preparation of this report:

- Leslie Lamport's LaTeX document markup language (`http://www.latex-project.org/`), based on Donald Knuth's TeX typesetting system (`http://www.tug.org/`), was used for the typesetting and layout of the report.
- Gnuplot (`http://www.gnuplot.info/`), version 4.0.0, was used for figures and charts.
- The R programming language and statistics environment (`http://www.r-project.org/`), version 2.5.0, was used for PCA and LDA calculation, hypothesis testing, and covariance matrix estimation.
- Qhull (`http://www.qhull.org/`), version 2003.1, was used for the calculation of the Voronoi diagram in figure 5.1.
- Vim (`http://www.vim.org/`), version 7.0, was used for all editing.

Appendix B

Source code

This chapter contains an excerpt of the descriptor extraction code and Voronoi/closest-match searcher described in the project. It does not include the build scripts, the various scripts used to generate tables and graphs automatically, and certain other auxiliary code – however, the complete, buildable code used, including all necessary scripts to reproduce the material (although excluding the song material used), is available in electronic form, along with a copy of this thesis, from `http://descriptors.sesse.net`.

All code except the source adapted from libvorbis is Copyright 2007 Steinar H. Gunderson, and licensed under the GNU General Public License, version 2 (see section B.1). The source adapted from libvorbis (residing in `f1cc.cpp`) is Copyright 1994-2002 by the Xiphophorus Company (`http://www.xiph.org/`), and licensed under the BSD-style license in section B.2.

B.1 The GNU General Public License

The following is the text of the GNU General Public Licence, under the terms of which this software is distributed.

GNU GENERAL PUBLIC LICENSE
Version 2, June 1991

Copyright (C) 1989, 1991 Free Software Foundation, Inc.
675 Mass Ave, Cambridge, MA 02139, USA
Everyone is permitted to copy and distribute verbatim copies
of this license document, but changing it is not allowed.

B.1.1 Preamble

The licenses for most software are designed to take away your freedom to share and change it. By contrast, the GNU General Public License is intended to guarantee your freedom to share and change free software—to make sure the software is free for all its users. This General Public License applies to most of the Free Software Foundation's software and to any other program whose authors commit to using it. (Some other Free Software Foundation software is covered by the GNU Library General Public License instead.) You can apply it to your programs, too.

When we speak of free software, we are referring to freedom, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for this service if you wish), that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs; and that you know you can do these things.

To protect your rights, we need to make restrictions that forbid anyone to deny you these rights or to ask you to surrender the rights. These restrictions translate to certain responsibilities for you if you distribute copies of the software, or if you modify it.

For example, if you distribute copies of such a program, whether gratis or for a fee, you must give the recipients all the rights that you have. You must make sure that they, too, receive or can get the source code. And you must show them these terms so they know their rights.

We protect your rights with two steps: (1) copyright the software, and (2) offer you this license which gives you legal permission to copy, distribute and/or modify the software.

Also, for each author's protection and ours, we want to make certain that everyone understands that there is no warranty for this free software. If the software is modified by someone else and passed on, we want its recipients to know that what they have is not the original, so that any problems introduced by others will not reflect on the original authors' reputations.

Finally, any free program is threatened constantly by software patents. We wish to avoid the danger that redistributors of a free program will individually obtain patent licenses, in effect making the program proprietary. To prevent this, we have made it clear that any patent must be licensed for everyone's free use or not licensed at all.

The precise terms and conditions for copying, distribution and modification follow.

B.1.2 Terms and conditions for copying, distribution and modification

0. This License applies to any program or other work which contains a notice placed by the copyright holder saying it may be distributed under the terms of this General Public License. The "Program", below, refers to any such program or work, and a "work based on the Program" means either the Program or any derivative work under copyright law: that is to say, a work containing the Program or a portion of it, either verbatim or with modifications and/or translated into another language. (Hereinafter, translation is included without limitation in the term "modification".) Each licensee is addressed as "you".

   Activities other than copying, distribution and modification are not covered by this License; they are outside its scope. The act of running the Program is not restricted, and the output from the Program is covered only if its contents constitute a work based on the Program (independent of having been made by running the Program). Whether that is true depends on what the Program does.

1. You may copy and distribute verbatim copies of the Program's source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice and disclaimer of warranty; keep intact all the notices that refer to this License and to the absence of any warranty; and give any other recipients of the Program a copy of this License along with the Program.

   You may charge a fee for the physical act of transferring a copy, and you may at your option offer warranty protection in exchange for a fee.

2. You may modify your copy or copies of the Program or any portion of it, thus forming a work based on the Program, and copy and distribute such modifications or work under the terms of Section 1 above, provided that you also meet all of these conditions:

   (a) You must cause the modified files to carry prominent notices stating that you changed the files and the date of any change.

   (b) You must cause any work that you distribute or publish, that in whole or in part contains or is derived from the Program or any part thereof, to be licensed as a whole at no charge to all third parties under the terms of this License.

   (c) If the modified program normally reads commands interactively when run, you must cause it, when started running for such interactive use in the most ordinary way, to print or display an announcement including an appropriate copyright notice and a notice that there is no warranty (or else, saying that you provide a warranty) and that users may redistribute the program under these conditions, and telling the user how to view a copy of this License. (Exception: if the Program itself is interactive but does not normally print such an announcement, your work based on the Program is not required to print an announcement.)

   These requirements apply to the modified work as a whole. If identifiable sections of that work are not derived from the Program, and can be reasonably considered independent and separate works in themselves, then this License, and its terms, do not apply to those sections when you distribute them as separate works. But when you distribute the same sections as part of a whole which is a work based on the Program, the distribution of the whole must be on the terms of this License, whose permissions for other licensees extend to the entire whole, and thus to each and every part regardless of who wrote it.

   Thus, it is not the intent of this section to claim rights or contest your rights to work written entirely by you; rather, the intent is to exercise the right to control the distribution of derivative or collective works based on the Program.

   In addition, mere aggregation of another work not based on the Program with the Program (or with a work based on the Program) on a volume of a storage or distribution medium does not bring the other work under the scope of this License.

3. You may copy and distribute the Program (or a work based on it, under Section 2) in object code or executable form under the terms of Sections 1 and 2 above provided that you also do one of the following:

   (a) Accompany it with the complete corresponding machine-readable source code, which must be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,

   (b) Accompany it with a written offer, valid for at least three years, to give any third party, for a charge no more than your cost of physically performing source distribution, a complete machine-readable copy of the corresponding source code, to be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,

B.2 libvorbis license

B.3 Code listings

B.3.1 Descriptor extraction code

descriptor.cpp

```
// descriptor.cpp -- main program, responsible for reading in the file
   (by
// calling the appropriate file reader), preprocessing the file
   appropriately,
// call all extraction functions in turn and finally output the
   extracted
// descriptors to standard output.

#include <stdio.h>
#include <math.h>

#include "common.h"
#include "centroid.h"
#include "wavelet.h"
#include "mfcc.h"
#include "f1cc.h"

#include "vorbis.h"
#include "mp3.h"
#include "flac.h"

void remove_dc_offset(Sample &s)
{
        double sum_x = 0.0;
        size_t N = std::min(s.samples.size(), (size_t)FEATURE_SAMPLES);

        for (unsigned i = 0; i < N; ++i) {
                sum_x += s.samples[i];
        }

        double dc = sum_x / N;
        for (unsigned i = 0; i < N; ++i) {
                s.samples[i] -= dc;
        }

        fprintf(stderr, "-dc(%.2f)...␣", dc);
}

void normalize_gain(Sample &s)
{
        double sum_x2 = 0.0;
        size_t N = std::min(s.samples.size(), (size_t)FEATURE_SAMPLES);

        for (unsigned i = 0; i < N; ++i) {
                sum_x2 += s.samples[i] * s.samples[i];
        }
        double gain = 8192.0 / sqrt(sum_x2 / N);

        for (unsigned i = 0; i < N; ++i) {
                s.samples[i] *= gain;
        }

        fprintf(stderr, "gain(%.2fdB)...␣", 10.0 * log10(gain));
}

void find_len(Sample &s)
{
        std::vector<double> d;
        d.push_back(s.length);
        s.descriptors["length"] = d;

        fprintf(stderr, "len(%.3fs)...␣", s.length);
}

// avg / RMS
void find_msratio(Sample &s)
{
        std::vector<double> d;
        double sum_x2 = 0.0, sum_x = 0.0;
        size_t N = std::min(s.samples.size(), (size_t)FEATURE_SAMPLES);

        for (unsigned i = 0; i < N; ++i) {
                double x = s.samples[i];
                sum_x2 += x * x;
                sum_x  += fabs(x);
        }

        double rms = sqrt(sum_x2 / N);
        double avg = sum_x / N;

        d.push_back(avg / rms);
        s.descriptors["msratio"] = d;
}

// number of zero-crossings
void find_zerocrossings(Sample &s)
{
        std::vector<double> d;
        unsigned zc = 0;
        double last_val = 0.0;
        size_t N = std::min(s.samples.size(), (size_t)FEATURE_SAMPLES);

        for (unsigned i = 0; i < N; ++i) {
                double x = s.samples[i];
                if (x > 0.0 && last_val < 0.0)
                        ++zc;
                else if (x < 0.0 && last_val > 0.0)
```

```
                                ++zc;
                        last_val = x;
                }

                d.push_back(zc);
                s.descriptors["zerocrossings"] = d;
}

// number of zero-crossings, with guard band (schmitt triggering)
void find_zerocrossings_guard(Sample &s)
{
                std::vector<double> d;
                unsigned zc = 0;
                double last_val = 0.0;
                size_t N = std::min(s.samples.size(), (size_t)FEATURE_SAMPLES);

                for (unsigned i = 0; i < N; ++i) {
                        double x = s.samples[i];
                        if (fabs(x) < 2048.0)
                                continue;
                        if (x > 0.0 && last_val < 0.0)
                                ++zc;
                        else if (x < 0.0 && last_val > 0.0)
                                ++zc;
                        last_val = x;
                }

                d.push_back(zc);
                s.descriptors["zerocrossings_guard"] = d;
}

// rate of signal change
void find_steepness(Sample &s)
{
                std::vector<double> d;
                double sum_delta_x = 0.0;
                size_t N = std::min(s.samples.size(), (size_t)FEATURE_SAMPLES);

                double last_x = s.samples[0];
                for (unsigned i = 1; i < N; ++i) {
                        double x = s.samples[i];
                        sum_delta_x += fabs(x - last_x);
                        last_x = x;
                }

                d.push_back(sum_delta_x / (N-1));
                s.descriptors["steepness"] = d;
}

int main(int argc, char **argv)
{
                FILE *wisdom = fopen("fft-wisdom", "rb");
```

```
        if (wisdom == NULL) {
                fprintf(stderr, "Optimizing␣FFTs␣for␣your␣system;␣this␣
                   will␣take␣a␣few␣minutes\n");
                fprintf(stderr, "even␣on␣a␣reasonably␣fast␣machine,␣so␣
                   please␣wait.␣This␣will␣only\n");
                fprintf(stderr, "need␣to␣happen␣once␣unless␣you␣remove␣
                   the␣\"fft-wisdom\"␣file.\n\n");
                init_mfcc();
                init_vp();
                init_centroid();

                wisdom = fopen("fft-wisdom", "wb");
                if (wisdom == NULL) {
                        perror("fft-wisdom");
                        exit(1);
                }
                fftw_export_wisdom_to_file(wisdom);
                fclose(wisdom);
        } else {
                fftw_import_wisdom_from_file(wisdom);
                fclose(wisdom);
        }

        for (int i = 1; i < argc; ++i) {
                Sample s;

                fprintf(stderr, "%s...␣", argv[i]);
                if (strstr(argv[i], ".ogg"))
                        read_vorbis_file(s, argv[i]);
                else if (strstr(argv[i], ".flac"))
                        read_flac_file(s, argv[i]);
                else
                        read_mp3_file(s, argv[i]);

                remove_dc_offset(s);
                normalize_gain(s);
                find_len(s);

                fprintf(stderr, "msratio...␣");
                find_msratio(s);

                fprintf(stderr, "zerocrossings...␣");
                find_zerocrossings(s);

                fprintf(stderr, "zerocrossings_guard...␣");
                find_zerocrossings_guard(s);

                fprintf(stderr, "steepness...␣");
                find_steepness(s);

                fprintf(stderr, "centroid...␣");
                find_centroid(s);
                fprintf(stderr, "mfcc...␣");
                find_mfcc(s);

                fprintf(stderr, "mfcc_floor1...␣");
                find_mfcc_floor1(s);

                fprintf(stderr, "done.\n");

                printf("filename=%s\n", argv[i]);
                for (std::map<std::string, std::vector<double> >::
                   const_iterator j = s.descriptors.begin(); j != s.
                   descriptors.end(); ++j) {
                        printf("%s=", j->first.c_str());
                        for (unsigned k = 0; k < j->second.size(); ++k)
                            {
                                printf("%f", j->second[k]);
                                if (k != j->second.size() - 1) {
                                        printf(",");
                                }
                        }
                        printf("\n");
                }
                printf("\n");
        }

        printf("END\n");
        exit(0);
}
```

centroid.cpp

```
// centroid.cpp -- finds the spectral centroid of the given sample.

#include "centroid.h"
#include <fftw3.h>

bool centroid_initialized = false;
fftw_plan centroid_fft_plan;
double *centroid_fft_in;
std::complex<double> *centroid_fft_out;

void init_centroid()
{
        if (centroid_initialized)
                return;

        // FFTW_PATIENT is not worth it -- it takes about two hours to
           plan and gives very little
        // extra performance in practice.

        centroid_fft_in = reinterpret_cast<double *> (fftw_malloc(
            sizeof(double) * FEATURE_SAMPLES));
        centroid_fft_out = reinterpret_cast<std::complex<double> *> (
            fftw_malloc(sizeof(std::complex<double>) * (FEATURE_SAMPLES
            / 2 + 1)));
        centroid_fft_plan = fftw_plan_dft_r2c_1d(FEATURE_SAMPLES,
            centroid_fft_in, reinterpret_cast<fftw_complex *> (
            centroid_fft_out), FFTW_MEASURE | FFTW_DESTROY_INPUT);

        centroid_initialized = true;
}

void find_centroid(Sample &s)
{
        init_centroid();
        size_t N = std::min(s.samples.size(), (size_t)FEATURE_SAMPLES);

        // get the data, and pad
        for (unsigned i = 0; i < N; ++i) {
                centroid_fft_in[i] = s.samples[i];
        }
        for (unsigned i = N; i < FEATURE_SAMPLES; ++i) {
                centroid_fft_in[i] = 0.0;
        }

        // takes some time...
        fftw_execute(centroid_fft_plan);

        // find the centroid from the spectrum
        double weighted_sum = 0.0, total_sum = 0.0;
        for (unsigned i = 0; i < (FEATURE_SAMPLES / 2 + 1); ++i) {
                double f = (i + 0.5) * 22050.0 / (FEATURE_SAMPLES / 2 +
                    1);
                weighted_sum += f * abs(centroid_fft_out[i]);
                total_sum += abs(centroid_fft_out[i]);
        }

        std::vector<double> d;
        d.push_back(weighted_sum / total_sum);
        s.descriptors["centroid"] = d;
}
```

mfcc.cpp

```
// mfcc.cpp -- extracts MFCC coefficients from the given sample, and
   computes
// statistical moments from that.

#include "mfcc.h"

bool mfcc_initialized = false;
fftw_plan fft_plan, dct_plan;
double *fft_in;
```

```
std::complex<double> *fft_out;
double *mfcc_in, *mfcc_out;
double *window;

void init_mfcc()
{
        if (mfcc_initialized)
                return;

        // FFT
        fft_in = reinterpret_cast<double *> (fftw_malloc(sizeof(double)
             * MFCC_WIN_LEN));
        fft_out = reinterpret_cast<std::complex<double> *> (fftw_malloc
           (sizeof(std::complex<double>) * (MFCC_WIN_LEN / 2 + 1)));
        fft_plan = fftw_plan_dft_r2c_1d(MFCC_WIN_LEN, fft_in,
           reinterpret_cast<fftw_complex *> (fft_out), FFTW_PATIENT |
           FFTW_DESTROY_INPUT);

        // DCT
        mfcc_in = reinterpret_cast<double *> (fftw_malloc(sizeof(double
           ) * MFCC_COEFFICIENTS));
        mfcc_out = reinterpret_cast<double *> (fftw_malloc(sizeof(
           double) * MFCC_COEFFICIENTS));
        dct_plan = fftw_plan_r2r_1d(MFCC_COEFFICIENTS, mfcc_in,
           mfcc_out, FFTW_REDFT10, FFTW_PATIENT | FFTW_DESTROY_INPUT);

        // Initialize the Hamming window
        window = new double[MFCC_WIN_LEN];
        for (unsigned i = 0; i < MFCC_WIN_LEN; ++i) {
                window[i] = 0.54 - 0.46 * cos(2.0 * M_PI * double(i) /
                   double(MFCC_WIN_LEN - 1));
        }

        mfcc_initialized = true;
}

void find_mfcc(Sample &s)
{
        unsigned pos = 0;
        unsigned chunk_len = MFCC_WIN_LEN / MFCC_OVERLAP;
        double buf[MFCC_WIN_LEN];
        double mel_spacing = hz_to_mel(44100.0 / 2.0) / double(
           MFCC_COEFFICIENTS + 1);  // ??
        std::vector<double> coeffs[MFCC_COEFFICIENTS];
        double mu[MFCC_COEFFICIENTS];

        init_mfcc();

        // fill the first buffer
        for (unsigned i = 0; i < MFCC_WIN_LEN - chunk_len; ++i, ++pos)
           {
                buf[i + chunk_len] = s.samples[pos];
        }

        for ( ;; ) {
                if (pos + MFCC_WIN_LEN / MFCC_OVERLAP >=
                    FEATURE_SAMPLES)
                        break;

                // move everything back a chunk
                memmove(buf, buf + chunk_len, (MFCC_WIN_LEN - chunk_len
                    ) * sizeof(double));

                // read a new chunk
                for (unsigned i = 0; i < chunk_len; ++i, ++pos) {
                        buf[i + MFCC_WIN_LEN - chunk_len] = s.samples[
                            pos];
                }

                // window it
                for (unsigned i = 0; i < MFCC_WIN_LEN; ++i) {
                        fft_in[i] = buf[i] * window[i];
                }

                fftw_execute(fft_plan);

                // apply the triangular windows
                for (unsigned i = 0; i < MFCC_COEFFICIENTS; ++i) {
                        double start = mel_to_hz(mel_spacing * (i    ))
                            ;
                        double mid   = mel_to_hz(mel_spacing * (i + 1))
                            ;
                        double end   = mel_to_hz(mel_spacing * (i + 2))
                            ;

                        double s = 0.0;

                        for (int j = int(ceil(hz_to_bin(start))); j <=
                            floor(hz_to_bin(end)); ++j) {
                                double f = bin_to_hz(j);
                                double energy = abs(fft_out[j]) * abs(
                                    fft_out[j]);

                                if (f < mid) {
                                        s += energy * (f - start) / (
                                            mid - start);
                                } else {
                                        s += energy * (end - f) / (end
                                            - mid);
                                }
                        }

                        mfcc_in[i] = log(s + 0.001);
                }
                fftw_execute(dct_plan);

                for (unsigned i = 0; i < MFCC_COEFFICIENTS; ++i) {
                        coeffs[i].push_back(mfcc_out[i]);
                }
        }

#if 0
        // output a little grayscale image
        {
                FILE *pgm = fopen("out.pgm", "wb");
                fprintf(pgm, "P5\n%u␣%u\n255\n", coeffs[0].size() - 1,
                    MFCC_COEFFICIENTS);

                double max = 0.0;
                for (unsigned i = 0; i < MFCC_COEFFICIENTS; ++i) {
                        for (unsigned j = 0; j < coeffs[0].size(); ++j)
                            {
                                max = std::max(max, log(fabs(coeffs[i][
                                    j])));
                        }
                }
                for (unsigned i = 0; i < MFCC_COEFFICIENTS; ++i) {
                        for (unsigned j = 1; j < coeffs[0].size(); ++j)
                            {
                                double z = 255.0 * log(fabs(coeffs[i][j
                                    ])) / max;
                                if (z < 0.0)
                                        fprintf(pgm, "%c", 0);
                                else
                                        fprintf(pgm, "%c", (unsigned
                                            char)z);
                        }
                }

                fclose(pgm);
        }

        // output a graph of the 7th MFCC coeffient
        {
                draw_graph("graph.pgm", coeffs[6]);

                std::vector<double> c_level1;
                std::vector<double> d_level1;

                {
                        std::vector<double> tmp;
                        wavelet_filter_haar(coeffs[6], tmp);
                        decimate(tmp, c_level1, 2);
                }
                {
```

```
                std::vector<double> tmp;
                wavelet_filter_haar_detail(coeffs[6], tmp);
                decimate(tmp, d_level1, 2);
        }

        std::vector<double> c_level2;
        std::vector<double> d_level2;

        {
                std::vector<double> tmp;
                wavelet_filter_haar(c_level1, tmp);
                decimate(tmp, c_level2, 2);
        }
        {
                std::vector<double> tmp;
                wavelet_filter_haar_detail(c_level1, tmp);
                decimate(tmp, d_level2, 2);
        }

        draw_graph("c_level2.pgm", c_level2);
        draw_graph("d_level2.pgm", d_level2);

        std::vector<double> c_level3;
        std::vector<double> d_level3;

        {
                std::vector<double> tmp;
                wavelet_filter_haar(c_level2, tmp);
                decimate(tmp, c_level3, 2);
        }
        {
                std::vector<double> tmp;
                wavelet_filter_haar_detail(c_level2, tmp);
                decimate(tmp, d_level3, 2);
        }

        draw_graph("c_level3.pgm", c_level3);
        draw_graph("d_level3.pgm", d_level3);

        std::vector<double> c_level4;
        std::vector<double> d_level4;

        {
                std::vector<double> tmp;
                wavelet_filter_haar(c_level3, tmp);
                decimate(tmp, c_level4, 2);
        }
        {
                std::vector<double> tmp;
                wavelet_filter_haar_detail(c_level3, tmp);
                decimate(tmp, d_level4, 2);
        }
```

```
                draw_graph("c_level4.pgm", c_level4);
                draw_graph("d_level4.pgm", d_level4);

                std::vector<double> c_level5;
                std::vector<double> d_level5;

                {
                        std::vector<double> tmp;
                        wavelet_filter_haar(c_level4, tmp);
                        decimate(tmp, c_level5, 2);
                }
                {
                        std::vector<double> tmp;
                        wavelet_filter_haar_detail(c_level4, tmp);
                        decimate(tmp, d_level5, 2);
                }

                draw_graph("c_level5.pgm", c_level5);
                draw_graph("d_level5.pgm", d_level5);

                std::vector<double> c_level6;
                std::vector<double> d_level6;

                {
                        std::vector<double> tmp;
                        wavelet_filter_haar(c_level5, tmp);
                        decimate(tmp, c_level6, 2);
                }
                {
                        std::vector<double> tmp;
                        wavelet_filter_haar_detail(c_level5, tmp);
                        decimate(tmp, d_level6, 2);
                }

                draw_graph("c_level6.pgm", c_level6);
                draw_graph("d_level6.pgm", d_level6);
        }
#endif

        // first find the first zero moment (average)
        {
                std::vector<double> d;
                for (unsigned i = 0; i < MFCC_COEFFICIENTS; ++i) {
                        mu[i] = 0.0;
                        for (unsigned j = 0; j < coeffs[i].size(); ++j)
                            {
                                mu[i] += coeffs[i][j];
                        }
                        mu[i] /= coeffs[i].size();

                        d.push_back(mu[i]);
                }
                s.descriptors["mfcc_avg"] = d;
        }

        // then the other moments (which are central moments)
        for (unsigned m = 2; m <= MFCC_MOMENTS; ++m) {
                std::vector<double> d;

                for (unsigned i = 0; i < MFCC_COEFFICIENTS; ++i) {
                        double v = 0.0;
                        for (unsigned j = 0; j < coeffs[i].size(); ++j)
                            {
                                v += pow(coeffs[i][j] - mu[i], m);
                        }
                        d.push_back(v / coeffs[i].size());
                }

                char name[32];
                sprintf(name, "mfcc_moment%u", m);
                s.descriptors[name] = d;
        }

        // deltify
        {
                std::vector<double> d;
                for (unsigned i = 0; i < MFCC_COEFFICIENTS; ++i) {
                        mu[i] = 0.0;
                        for (unsigned j = 1; j < coeffs[i].size(); ++j)
                            {
                                mu[i] += fabs(coeffs[i][j] - coeffs[i][
                                   j - 1]);
                        }
                        mu[i] /= (coeffs[i].size() - 1);

                        d.push_back(mu[i]);
                }
                s.descriptors["mfcc_delta_avg"] = d;
        }
        for (unsigned m = 2; m <= MFCC_MOMENTS; ++m) {
                std::vector<double> d;

                for (unsigned i = 0; i < MFCC_COEFFICIENTS; ++i) {
                        double v = 0.0;
                        for (unsigned j = 1; j < coeffs[i].size(); ++j)
                            {
                                v += pow(fabs((coeffs[i][j] - coeffs[i
                                   ][j - 1])) - mu[i], m);
                        }
                        d.push_back(v / (coeffs[i].size() - 1));
                }

                char name[32];
                sprintf(name, "mfcc_delta_moment%u", m);
                s.descriptors[name] = d;
        }
}
```

f1cc.cpp

```
// f1cc.cpp -- extracts F1CC coefficients from the given sample, and
   computes
// statistical moments from that.

#include <stdio.h>
#include <math.h>

#include "mfcc.h"
#include "f1cc.h"

// Headers imported from vorbisenc, with some adaption to allow
// the C code to be used from C++.
#define class _class
extern "C" {
        #include "vorbis/codec.h"
        #include "codec_internal.h"
        #include "registry.h"
        #include "psy.h"
        #include "misc.h"
        #include "masking.h"
        #include "scales.h"
}
#undef class
#undef max
#undef min

vorbis_look_psy vp_p;
void _vp_mod_tonemask(vorbis_look_psy *p, float *logfft, float *logmask
   , float global_specmax, float local_specmax);

void find_mfcc_floor1(Sample &s)
{
        unsigned pos = 0;
        unsigned chunk_len = MFCC_WIN_LEN / MFCC_OVERLAP;
        double buf[MFCC_WIN_LEN];

        // the tone curves seem to go haywire from about 3200 mel (
           about 11khz), so stop there
        double mel_spacing = 3200.0 / double(MFCC_COEFFICIENTS + 1);

        std::vector<double> coeffs[MFCC_COEFFICIENTS];
        double mu[MFCC_COEFFICIENTS];
        float tone[chunk_len];
        float logfft[chunk_len];
```

```
init_mfcc();
init_vp();

// fill the first buffer
for (unsigned i = 0; i < MFCC_WIN_LEN - chunk_len; ++i, ++pos)
   {
        buf[i + chunk_len] = s.samples[pos];
}

for ( ;; ) {
        if (pos + MFCC_WIN_LEN / MFCC_OVERLAP >=
           FEATURE_SAMPLES)
                break;

        // move everything back a chunk
        memmove(buf, buf + chunk_len, (MFCC_WIN_LEN - chunk_len
           ) * sizeof(double));

        // read a new chunk
        for (unsigned i = 0; i < chunk_len; ++i, ++pos) {
                buf[i + MFCC_WIN_LEN - chunk_len] = s.samples[
                   pos];
        }

        // window it
        for (unsigned i = 0; i < MFCC_WIN_LEN; ++i) {
                fft_in[i] = buf[i] * window[i] * (1.0/32768.0);
        }

        fftw_execute(fft_plan);

        float scale = 4.0f / chunk_len;
        float scale_dB = todB(&scale) + .345;

        // find amplitude maximum
        float temp = abs(fft_out[0]);
        float ampmax = logfft[0] = scale_dB + todB(&temp) +
           .345;

        //printf("logfft: %f", logfft[0]);
        for (unsigned i = 1; i < MFCC_WIN_LEN / 2; ++i) {
                temp = abs(fft_out[i]);
                logfft[i] = scale_dB + todB(&temp) + .345;
                //printf(" %f", logfft[i]);

                ampmax = std::max(ampmax, temp);
        }
        //printf("\n");

        // tone masking curve
        _vp_mod_tonemask(&vp_p, logfft, tone, ampmax, ampmax);
```

```
#if 0
                printf("tone␣dump:␣");
                for (unsigned i = 0; i < chunk_len / 2; ++i) {
                        printf("%f␣", tone[i]);
                }
                printf("\n");
#endif

#if 0
                // debugging output for gnuplot et al
                if (pos == (MFCC_WIN_LEN / MFCC_OVERLAP) * 860) {
                        FILE *f1 = fopen("fft.plot", "w");
                        for (unsigned i = 0; i < MFCC_WIN_LEN / 2; ++i)
                            {
                                fprintf(f1, "%f␣%f\n", hz_to_mel(
                                   bin_to_hz(i)), logfft[i]);
                        }
                        fclose(f1);

                        FILE *f2 = fopen("tone.plot", "w");
                        for (unsigned i = 0; i < MFCC_WIN_LEN / 2; ++i)
                            {
                                fprintf(f2, "%f␣%f\n", hz_to_mel(
                                   bin_to_hz(i)), tone[i]);
                        }
                        fclose(f2);

                        usleep(50000);
                }
#endif

                // apply the triangular windows
                for (unsigned i = 0; i < MFCC_COEFFICIENTS; ++i) {
                        double start = mel_to_hz(mel_spacing * (i    ))
                            ;
                        double mid   = mel_to_hz(mel_spacing * (i + 1))
                            ;
                        double end   = mel_to_hz(mel_spacing * (i + 2))
                            ;

                        double s = 0.0;

                        for (int j = int(ceil(hz_to_bin(start))); j <=
                            floor(hz_to_bin(end)); ++j) {
                                double f = bin_to_hz(j);
                                double energy = fromdB(tone[j]) *
                                   fromdB(tone[j]);

                                if (f < mid) {
                                        s += energy * (f - start) / (
                                           mid - start);
                                } else {
                                        s += energy * (end - f) / (end
                                            - mid);
                                }
                        }

                        mfcc_in[i] = log(s);
                }

                fftw_execute(dct_plan);

                for (unsigned i = 0; i < MFCC_COEFFICIENTS; ++i) {
                        coeffs[i].push_back(mfcc_out[i]);
                }
        }

        // first find the first zero moment (average)
        {
                std::vector<double> d;
                for (unsigned i = 0; i < MFCC_COEFFICIENTS; ++i) {
                        mu[i] = 0.0;
                        for (unsigned j = 0; j < coeffs[i].size(); ++j)
                            {
                                mu[i] += coeffs[i][j];
                        }
                        mu[i] /= coeffs[i].size();

                        d.push_back(mu[i]);
                }
                s.descriptors["mfcc_f1_avg"] = d;
        }

        // then the other moments (which are central moments)
        for (unsigned m = 2; m <= MFCC_MOMENTS; ++m) {
                std::vector<double> d;

                for (unsigned i = 0; i < MFCC_COEFFICIENTS; ++i) {
                        double v = 0.0;
                        for (unsigned j = 0; j < coeffs[i].size(); ++j)
                            {
                                v += pow(coeffs[i][j] - mu[i], m);
                        }
                        d.push_back(v / coeffs[i].size());
                }

                char name[32];
                sprintf(name, "mfcc_f1_moment%u", m);
                s.descriptors[name] = d;
        }

        // deltify
        {
                std::vector<double> d;
                for (unsigned i = 0; i < MFCC_COEFFICIENTS; ++i) {
                        mu[i] = 0.0;
                        for (unsigned j = 1; j < coeffs[i].size(); ++j)
                            {
                                mu[i] += fabs(coeffs[i][j] - coeffs[i][
                                    j - 1]);
                        }
                        mu[i] /= (coeffs[i].size() - 1);

                        d.push_back(mu[i]);
                }
                s.descriptors["mfcc_f1_delta_avg"] = d;
        }
        for (unsigned m = 2; m <= MFCC_MOMENTS; ++m) {
                std::vector<double> d;

                for (unsigned i = 0; i < MFCC_COEFFICIENTS; ++i) {
                        double v = 0.0;
                        for (unsigned j = 1; j < coeffs[i].size(); ++j)
                            {
                                v += pow(fabs(coeffs[i][j] - coeffs[i][
                                    j - 1]) - mu[i], m);
                        }
                        d.push_back(v / (coeffs[i].size() - 1));
                }

                char name[32];
                sprintf(name, "mfcc_f1_delta_moment%u", m);
                s.descriptors[name] = d;
        }
}

// Everything below this line is copied/adapted from libvorbisenc,
   mostly
// from lib/floor1.c, and is Copyright 1994-2002 by the Xiphophorus
   Company
// (http://www.xiph.org/). See the COPYING file included with libvorbis
    for
// the full license.

typedef struct {
        int att[P_NOISECURVES];
        float boost;
        float decay;
} att3;

typedef struct {
        int   pre[PACKETBLOBS];
        int   post[PACKETBLOBS];
        float kHz[PACKETBLOBS];
        float lowpasskHz[PACKETBLOBS];
} adj_stereo;
```

```
typedef struct {
        int lo;
        int hi;
        int fixed;
} noiseguard;

typedef struct vp_adjblock{
        int block[P_BANDS];
} vp_adjblock;
typedef struct {
        int data[NOISE_COMPAND_LEVELS];
} compandblock;
typedef struct {
        int data[P_NOISECURVES][17];
} noise3;
#include "modes/psych_44.h"

bool vp_initialized = false;
vorbis_info_psy vp_vi = _psy_info_template;
vorbis_info_psy_global vp_gi = _psy_global_44[0];

void init_vp()
{
        if (vp_initialized)
                return;

        _vp_psy_init(&vp_p, &vp_vi, &vp_gi, MFCC_FLOOR0_DATA_LEN / 2,
           44100);

        vp_initialized = true;
}

#define NEGINF -9999.f

/* octave/(8*eighth_octave_lines) x scale and dB y scale */
static void seed_curve(float *seed,
                       const float **curves,
                       float amp,
                       int oc, int n,
                       int linesper,float dBoffset){
  int i,post1;
  int seedptr;
  const float *posts,*curve;

  int choice=(int)((amp+dBoffset-P_LEVEL_0)*.1f);
  choice=std::max(choice,0);
  choice=std::min(choice,P_LEVELS-1);
  posts=curves[choice];
  curve=posts+2;
  post1=(int)posts[1];
  seedptr=oc+(posts[0]-EHMER_OFFSET)*linesper-(linesper>>1);
  for(i=posts[0];i<post1;i++){
    if(seedptr>0){
      float lin=amp+curve[i];
      if(seed[seedptr]<lin)seed[seedptr]=lin;
    }
    seedptr+=linesper;
    if(seedptr>=n)break;
  }
}

static void seed_loop(vorbis_look_psy *p,
                      const float ***curves,
                      const float *f,
                      const float *flr,
                      float *seed,
                      float specmax){
  vorbis_info_psy *vi=p->vi;
  long n=p->n,i;
  float dBoffset=vi->max_curve_dB-specmax;

  /* prime the working vector with peak values */

  for(i=0;i<n;i++){
    float max=f[i];
    long oc=p->octave[i];
    while(i+1<n && p->octave[i+1]==oc){
      i++;
      if(f[i]>max)max=f[i];
    }

    if(max+6.f>flr[i]){
      oc=oc>>p->shiftoc;
      if(oc>=P_BANDS)oc=P_BANDS-1;
      if(oc<0)oc=0;

      seed_curve(seed,
                 curves[oc],
                 max,
                 p->octave[i]-p->firstoc,
                 p->total_octave_lines,
                 p->eighth_octave_lines,
                 dBoffset);
    }
  }
}

static void seed_chase(float *seeds, int linesper, long n){
  long  posstack[n];
  float ampstack[n];
  long   stack=0;
  long   pos=0;
  long   i;

  for(i=0;i<n;i++){
    if(stack<2){
      posstack[stack]=i;
      ampstack[stack++]=seeds[i];
    }else{
      while(1){
        if(seeds[i]<ampstack[stack-1]){
          posstack[stack]=i;
          ampstack[stack++]=seeds[i];
          break;
        }else{
          if(i<posstack[stack-1]+linesper){
            if(stack>1 && ampstack[stack-1]<=ampstack[stack-2] &&
               i<posstack[stack-2]+linesper){
              /* we completely overlap, making stack-1 irrelevant.  pop
                  it */
              stack--;
              continue;
            }
          }
          posstack[stack]=i;
          ampstack[stack++]=seeds[i];
          break;

        }
      }
    }
  }

  /* the stack now contains only the positions that are relevant. Scan
     'em straight through */

  for(i=0;i<stack;i++){
    long endpos;
    if(i<stack-1 && ampstack[i+1]>ampstack[i]){
      endpos=posstack[i+1];
    }else{
      endpos=posstack[i]+linesper+1; /* +1 is important, else bin 0 is
                                        discarded in short frames */
    }
    if(endpos>n)endpos=n;
    for(;pos<endpos;pos++)
      seeds[pos]=ampstack[i];
  }

  /* there.  Linear time.  I now remember this was on a problem set I
     had in Grad Skool... I didn't solve it at the time ;-) */

}
```

```
/* bleaugh, this is more complicated than it needs to be */
#include<stdio.h>
static void max_seeds(vorbis_look_psy *p,
                      float *seed,
                      float *flr){
  long   n=p->total_octave_lines;
  int    linesper=p->eighth_octave_lines;
  long   linpos=0;
  long   pos;

  seed_chase(seed,linesper,n); /* for masking */

  pos=p->octave[0]-p->firstoc-(linesper>>1);

  while(linpos+1<p->n){
    float minV=seed[pos];
    long end=((p->octave[linpos]+p->octave[linpos+1])>>1)-p->firstoc;
    if(minV>p->vi->tone_abs_limit)minV=p->vi->tone_abs_limit;
    while(pos+1<=end){
      pos++;
      if((seed[pos]>NEGINF && seed[pos]<minV) || minV==NEGINF)
        minV=seed[pos];
    }

    end=pos+p->firstoc;
    for(;linpos<p->n && p->octave[linpos]<=end;linpos++)
      if(flr[linpos]<minV)flr[linpos]=minV;
  }

  {
    float minV=seed[p->total_octave_lines-1];
    for(;linpos<p->n;linpos++)
      if(flr[linpos]<minV)flr[linpos]=minV;
  }

}

void _vp_mod_tonemask(vorbis_look_psy *p, float *logfft, float *logmask
    , float global_specmax, float local_specmax)
{
        int i, n = p->n;
        float seed[p->total_octave_lines];

        for (i = 0; i < p->total_octave_lines; i++)
                seed[i] = NEGINF;
#if 1
        for (i = 0; i < n; i++)
                logmask[i] = local_specmax + p->vi->ath_adjatt;
#else
        float att = local_specmax + p->vi->ath_adjatt;
        if (att < p->vi->ath_maxatt)
                att = p->vi->ath_maxatt;
        for (i = 0; i < n; i++)
                logmask[i] = p->ath[i] + att;
#endif

        /* tone masking */
        seed_loop(p, (const float ***)p->tonecurves, logfft, logmask,
           seed, global_specmax);
        max_seeds(p, seed, logmask);
}

// End of libvorbisenc code.
```

vorbis.cpp

```
// vorbis.cpp -- reads in Vorbis files, using libvorbisfile.

#include <stdio.h>
#include <vorbis/vorbisfile.h>
#include <cassert>

#include "vorbis.h"

void read_vorbis_file(Sample &s, char *filename)
{
        int junk __attribute__ ((unused));
        OggVorbis_File vf;
        unsigned num_read = 0;

        s.samples.reserve(5000000);

        FILE *fle = fopen(filename, "rb");
        if (fle == NULL) {
                perror(filename);
                exit(1);
        }

        if (ov_open(fle, &vf, NULL, 0) < 0) {
                fprintf(stderr, "Couldn't␣open␣bitstream\n");
                exit(1);
        }

        for ( ;; ) {
                char buf[4096];
                short *ptr = (short *)buf;

                int ret = ov_read(&vf, buf, 4096, 0, 2, 1, &junk);
                if (ret < 0) {
                        fprintf(stderr, "Error␣in␣Vorbis␣bitstream\n");
                        exit(1);
                }
                if (ret == 0)
                        break;
                assert(ret % 4 == 0);
                num_read += ret / 4;

                for (int i = 0; i < ret / 2; i += 2) {
                        s.samples.push_back(0.5 * (double(ptr[i]) +
                           double(ptr[i+1])));
                }

                if (num_read >= FEATURE_SAMPLES)
                        break;
        }

        s.length = ov_time_total(&vf, -1);
        ov_clear(&vf);
}
```

mp3.cpp

```
/*
 * Simple MP3 decoder using MAD. We use the low-level API (documented
    in
 * madlld) so we can stop synthesizing PCM and only count time when we
    have
 * the data we need; a bit more involved, but much faster. Heavily
    based
 * on madlld.c.
 */

#include <stdio.h>
#include <assert.h>
#include <mad.h>

#include "mp3.h"

static inline
signed int scale(mad_fixed_t sample)
{
        /* round */
        sample += (1L << (MAD_F_FRACBITS - 16));

        /* clip */
        if (sample >= MAD_F_ONE)
                sample = MAD_F_ONE - 1;
        else if (sample < -MAD_F_ONE)
                sample = -MAD_F_ONE;

        /* quantize */
        return sample >> (MAD_F_FRACBITS + 1 - 16);
}

// Read in the entire file, which is the simplest way to provide
// the EOF semantics libmad assumes.
char *gobble_file(char *filename, long *file_len)
{
        FILE *fle = fopen(filename, "rb");
        if (fle == NULL) {
                perror(filename);
                exit(1);
        }

        fseek(fle, 0, SEEK_END);
        *file_len = ftell(fle);

        char *mp3_buf = new char[*file_len + 8]; // eight guard bytes
        rewind(fle);
        fread(mp3_buf, *file_len, 1, fle);
        fclose(fle);

        memset(mp3_buf + *file_len, 0, 8);

        return mp3_buf;
}

void read_mp3_file(Sample &s, char *filename)
{
        s.samples.reserve(FEATURE_SAMPLES + 44100); // some extra room

        unsigned mp3_pos = 0;
        long mp3_len;
        char *mp3_buf = gobble_file(filename, &mp3_len);

        struct mad_stream stream;
        struct mad_frame frame;
        struct mad_synth synth;
        mad_timer_t timer;

        mad_stream_init(&stream);
        mad_frame_init(&frame);
        mad_synth_init(&synth);
        mad_timer_reset(&timer);

        for ( ;; ) {
                // need to fill buffer?
                if (stream.buffer == NULL || stream.error ==
                    MAD_ERROR_BUFLEN) {
                        if (stream.next_frame != NULL) {
                                size_t remaining = stream.bufend -
                                    stream.next_frame;
                                mp3_pos -= remaining;
                        }

                        mad_stream_buffer(&stream, (unsigned char *)
                            mp3_buf + mp3_pos, mp3_len + 8 - mp3_pos);
```

```
                                stream.error = MAD_ERROR_NONE;
                        }

                        if (mad_frame_decode(&frame, &stream)) {
                                if (MAD_RECOVERABLE(stream.error) && stream.
                                    error == MAD_ERROR_LOSTSYNC && stream.
                                    this_frame == (unsigned char *)mp3_buf +
                                    mp3_len) {
                                        // end of stream
                                        break;
                                } else {
                                        fprintf(stderr, "mad␣error.␣exiting.\n"
                                            );
                                        exit(1);
                                }
                        }

                        mad_timer_add(&timer, frame.header.duration);

                        if (timer.seconds < 30) {
                                mad_synth_frame(&synth, &frame);
                                assert(MAD_NCHANNELS(&frame.header)==2);

                                for (unsigned i = 0; i < synth.pcm.length; ++i)
                                    {
                                        signed short l, r;

                                        l = scale(synth.pcm.samples[0][i]);
                                        r = scale(synth.pcm.samples[1][i]);

#if 0
                                        /*
                                         * There's a lot of codec delay in MP3.
                                             This skips
                                         * the first 1105 samples (LAME's
                                            estimate of the
                                         * delay for our test MP3s) to make
                                            lining up plots
                                         * easier.
                                         */
                                        static unsigned delayed = 0;
                                        if (++delayed < 1105)
                                                continue;
#endif

                                        s.samples.push_back(0.5 * (double(l) +
                                            double(r)));
                                }
                        }
                }

                delete[] mp3_buf;
```

```
        s.length = double(timer.seconds) + double(timer.fraction) /
           double(MAD_TIMER_RESOLUTION);
}
```

flac.cpp

```
// flac.cpp -- reads in FLAC files using libflac (1.1.4 or newer).

#include <stdio.h>
#include <math.h>
#include <assert.h>
#include <FLAC/all.h>

#include "flac.h"

extern "C" FLAC__StreamDecoderWriteStatus flac_write_callback(const
   FLAC__StreamDecoder *decoder, const FLAC__Frame *frame, const
   FLAC__int32 *const buffer[], void *client_data)
{
        Sample *s = (Sample *)client_data;
        assert(frame->header.channels == 2);
        assert(frame->header.bits_per_sample == 16);
        assert(frame->header.sample_rate == 44100);

        const FLAC__int32 *lptr = buffer[0], *rptr = buffer[1];

        for (unsigned i = 0; i < frame->header.blocksize; ++i) {
                s->samples.push_back(0.5 * (double(*lptr++) + double(*
                   rptr++)));
        }

        return FLAC__STREAM_DECODER_WRITE_STATUS_CONTINUE;
}

extern "C" void flac_metadata_callback(const FLAC__StreamDecoder *
   decoder, const FLAC__StreamMetadata *metadata, void *client_data)
{
        if (metadata->type != FLAC__METADATA_TYPE_STREAMINFO)
                return;

        Sample *s = (Sample *)client_data;
        s->length = metadata->data.stream_info.total_samples / 44100.0;
}

extern "C" void flac_error_callback(const FLAC__StreamDecoder *decoder,
    FLAC__StreamDecoderErrorStatus status, void *client_data)
{
        fprintf(stderr, "FLAC␣status␣%d,␣exiting\n", status);
        exit(1);
}
```

```
void read_flac_file(Sample &s, char *filename)
{
        s.samples.reserve(5000000);
        s.length = -1.0;

        FILE *fle = fopen(filename, "rb");
        if (fle == NULL) {
                perror(filename);
                exit(1);
        }

        FLAC__StreamDecoder *dec = FLAC__stream_decoder_new();
        if (FLAC__stream_decoder_init_FILE(dec, fle,
           flac_write_callback, flac_metadata_callback,
           flac_error_callback, &s) !=
           FLAC__STREAM_DECODER_INIT_STATUS_OK) {
                fprintf(stderr, "FLAC__stream_decoder_init_FILE␣failed
                   .\n");
                exit(1);
        }

        /*
         * Decode until we have enough samples _and_ we know the length
            . Hopefully
         * the length will come from metadata; if not, we will decode
            the entire
         * stream to memory. This could be avoided, but it's not a use-
            case we need
         * to worry particularly about in this context.
         */
        while (s.samples.size() < FEATURE_SAMPLES || s.length < 0.0) {
                if (FLAC__stream_decoder_get_state(dec) ==
                   FLAC__STREAM_DECODER_END_OF_STREAM)
                        break;

                if (FLAC__stream_decoder_process_single(dec) == false)
                   {
                        fprintf(stderr, "
                           FLAC__stream_decoder_process_until_end_of_stream
                           ␣failed.\n");
                        exit(1);
                }
        }
        FLAC__stream_decoder_finish(dec);
        FLAC__stream_decoder_delete(dec);
}
```

B.3.2 Voronoi/closest-match searcher

voronoi.cpp

```
// Simple descriptor tester -- tests a given descriptor configuration
// using the nearest-neighbor test described in the report.
```

```
#include <stdio.h>
#include <math.h>
#include <string.h>
#include <stdlib.h>
#include <getopt.h>
#include <map>
#include <string>
#include <vector>
#include <algorithm>

// Basic data structures.
typedef std::map<std::string, std::vector<double> > descriptor;
struct song {
        descriptor desc;
        unsigned file_num, subfile_num;
        std::string filename;
        unsigned class_id;
};

// The collection of all song data, unprocessed.
std::vector<song> songs;

// All vectors for references (r) and others (s). The "trans" versions
// are transformed by the covariance matrix (inv_cov_mat, below), if
   any.
// rnum and snum are counts.
double *rvec, *svec, *rtransvec, *stransvec;
int *rnum, *snum;

// How many references are there in all? (Non-references is songs.size
   ()
// minus this.)
unsigned num_ref = 0;

// Various filenames and options.
static std::string output_data_filename;
static std::string output_class_filename;
static std::string output_tex_filename;
static std::string output_raw_filename;
static std::string input_matrix_filename;
static std::string input_center_filename;
static std::string input_scale_filename;
static std::string input_inv_cov_filename;
static bool show_misclassified = false;
static std::string dump_filename;
static bool do_read_dump = false, do_write_dump = false;
static int num_dimensions = -1;

// Various matrices, usually read from tet files.
double *rot_mat = NULL, *center = NULL, *scale = NULL, *inv_cov_mat =
   NULL;
```

```
// Options to be given to getopt_long().
const static struct option longopts[] = {
        { "features", required_argument, NULL, 'f' },
        { "output-data", required_argument, NULL, 'o' },
        { "output-class", required_argument, NULL, 'O' },
        { "output-tex", required_argument, NULL, 'x' },
        { "output-raw", required_argument, NULL, 'X' },
        { "show-misclassified", no_argument, NULL, 'm' },
        { "write-dump", required_argument, NULL, 'w' },
        { "read-dump", required_argument, NULL, 'r' },
        { "num-dimensions", required_argument, NULL, 'd' },
        { "transform-matrix", required_argument, NULL, 't' },
        { "transform-center", required_argument, NULL, 'c' },
        { "transform-scale", required_argument, NULL, 's' },
        { "inv-covariance-matrix", required_argument, NULL, 'i' },
};

// Read a descriptor file as generated by the feature extractor.
void read_file(const char * const filename)
{
        static unsigned file_num = 0;
        unsigned subfile_num = 0;
        std::string song_filename;

        descriptor d;
        FILE *f = fopen(filename, "r");
        if (f == NULL) {
                perror(filename);
                exit(1);
        }

        for ( ;; ) {
                char buf[4096];
                char *ptr = fgets(buf, 4096, f);
                if (ptr == NULL) {
                        fprintf(stderr, "%s:␣Unexpected␣EOF\n",
                           filename);
                        exit(1);
                }

                // strip trailing EOL
                ptr = strchr(buf, '\n');
                if (ptr != NULL)
                        *ptr = '\0';

                // EOF marker
                if (strcmp(buf, "END") == 0)
                        break;

                // end-of-record marker
                if (strlen(buf) == 0) {
```

```
                song s;
                s.file_num = file_num;
                s.subfile_num = subfile_num++;
                s.desc = d;
                s.filename = song_filename;

                // Hard-coded class list
                if (strstr(song_filename.c_str(), ".flac") !=
                    NULL) {
                        s.class_id = 0;
                } else if (strstr(song_filename.c_str(), ".128
                    kbps.mp3") != NULL) {
                        s.class_id = 1;
                } else if (strstr(song_filename.c_str(), ".192
                    kbps.mp3") != NULL) {
                        s.class_id = 2;
                } else if (strstr(song_filename.c_str(), ".256
                    kbps.mp3") != NULL) {
                        s.class_id = 3;
                } else if (strstr(song_filename.c_str(), ".64
                    kbps.ogg") != NULL) {
                        s.class_id = 4;
                } else if (strstr(song_filename.c_str(), ".128
                    kbps.ogg") != NULL) {
                        s.class_id = 5;
                } else if (strstr(song_filename.c_str(), ".192
                    kbps.ogg") != NULL) {
                        s.class_id = 6;
                } else {
                        fprintf(stderr, "Unable␣to␣classify␣
                           file␣'%s'\n",
                                song_filename.c_str());
                        exit(1);
                }

                songs.push_back(s);
                d = descriptor();
                continue;
        }

        ptr = strchr(buf, '=');
        if (ptr == NULL) {
                fprintf(stderr, "%s:␣Malformed␣line␣'%s'\n",
                    filename, buf);
                exit(1);
        }

        *ptr = '\0';
        char *key = buf;
        char *value = ptr + 1;

        if (strcmp(key, "filename") == 0) {
```

```
                        song_filename = value;
                        continue;
                }

                std::vector<double> values;
                values.reserve(16);

                ptr = strtok(value, ",");
                while (ptr != NULL) {
                        double v = atof(ptr);

                        // some fiddling that should really be done in
                           descriptor.cpp
                        if (strcmp(key, "mfcc_moment2") == 0 || strcmp(
                           key, "mfcc_f1_moment2") == 0)
                                v = sqrt(v);
                        else if (strcmp(key, "mfcc_moment3") == 0 ||
                           strcmp(key, "mfcc_f1_moment3") == 0)
                                v = cbrt(v);
                        else if (strcmp(key, "mfcc_moment4") == 0 ||
                           strcmp(key, "mfcc_f1_moment4") == 0)
                                v = pow(v, 1.0/4.0);
                        else if (strcmp(key, "mfcc_moment5") == 0 ||
                           strcmp(key, "mfcc_f1_moment5") == 0)
                                v = (v < 0.0) ? -pow(-v, 1.0/5.0) : pow
                                   (v, 1.0/5.0);

                        values.push_back(v);
                        ptr = strtok(NULL, ",");
                }

                d.insert(std::make_pair(key, values));
        }

        fclose(f);
        ++file_num;

        if (file_num % 64 == 0)
                fprintf(stderr, "Reading:␣%u\r", songs.size());
}

// Read several files -- the file names are first read from a file list
   ,
// and each file name is then passed to read_file().
void read_filelist(char *filename)
{
        FILE *f = fopen(filename, "r");
        if (f == NULL) {
                perror(filename);
                exit(1);
        }
        for ( ;; ) {
                char buf[4096];
                char *ptr = fgets(buf, 4096, f);
                if (ptr == NULL)
                        break;

                // strip trailing EOL
                ptr = strchr(buf, '\n');
                if (ptr != NULL)
                        *ptr = '\0';

                read_file(buf);
        }

        fclose(f);
}

// Extract a single feature from a "song" struct. A feature can either
// be of the form "name[index]", or just "name" (in which case index
// is taken to be zero). No error checking is performed, in particular
// not on out-of-bounds indices.
double extract_feature(song &s, std::string feature)
{
        char buf[256];
        unsigned i = feature.find_first_of("[");
        if (i == std::string::npos) {
                return s.desc[feature][0];
        } else {
                strncpy(buf, feature.c_str(), i);
                buf[i] = '\0';

                return s.desc[buf][atoi(feature.c_str() + i + 1)];
        }
}

// Euclidean distance between two vectors of dimensionality LEN.
double distance(double *p1, double *p2, unsigned len)
{
        double sum = 0.0;
        for (unsigned i = 0; i < len; ++i, ++p1, ++p2) {
                sum += (*p2 - *p1) * (*p2 - *p1);
        }

        return sum;
}

// Mahalanobis distance between two vectors of dimensionality LEN. p1t
   and p2t
// are taken to be transformed (by the inverse covariance matrix)
   versions of p1
// and p2, respectively.
double distancem(double *p1, double *p2, double *p1t, double *p2t,
   unsigned len)
{
        double sum = 0.0;
        for (unsigned i = 0; i < len; ++i, ++p1, ++p2, ++p1t, ++p2t) {
                sum += (*p2 - *p1) * (*p2t - *p1t);
        }

        return sum;
}

// Split a feature string (feature[1],other_feature[0],etc.) into a
   list of single
// feature names.
std::vector<std::string> parse_features(const char * const
   feature_string)
{
        std::vector<std::string> features;
        char *fs = strdup(feature_string);

        char *ptr = strtok(fs, ",");
        while (ptr != NULL) {
                features.push_back(ptr);
                ptr = strtok(NULL, ",");
        }

        free(fs);
        return features;
}

// From the list of songs and the list of desired features, create a
   simple
// two-dimensional arrays of just the features we need for processing.
   This
// structure is much smaller, simpler and faster to work on than the
   general-
// purpose "song" structure.
void extract_features(const std::vector<std::string> &features)
{
        double tmp[features.size()];

        // count the number of references
        for (unsigned i = 0; i < songs.size(); ++i) {
                if (songs[i].class_id == 0)
                        ++num_ref;
        }

        // extract the feature vectors
        rvec = new double[num_ref * features.size()];
        svec = new double[(songs.size() - num_ref) * features.size()];
        rnum = new int[num_ref];
        snum = new int[songs.size() - num_ref];
```

```
FILE *fp = NULL, *cl = NULL;

if (output_data_filename.size() > 0) {
        fp = fopen(output_data_filename.c_str(), "w");
        if (fp == NULL) {
                perror(output_data_filename.c_str());
                exit(1);
        }
        for (unsigned j = 0; j < features.size(); ++j) {
                fprintf(fp, "%s", features[j].c_str());
                if (j == features.size() - 1)
                        fprintf(fp, "\n");
                else
                        fprintf(fp, ",");
        }
}
if (output_class_filename.size() > 0) {
        cl = fopen(output_class_filename.c_str(), "w");
        if (cl == NULL) {
                perror(output_class_filename.c_str());
                exit(1);
        }
        fprintf(cl, "class\n");
}

for (unsigned i = 0, m = 0, n = 0; i < songs.size(); ++i) {
        double *ptr;
        if (songs[i].class_id == 0) {
                ptr = rvec + m * features.size();
                rnum[m] = i;
                ++m;
        } else {
                ptr = svec + n * features.size();
                snum[n] = i;
                ++n;
        }

        for (unsigned j = 0; j < features.size(); ++j) {
                ptr[j] = extract_feature(songs[i], features[j])
                    ;
        }

        // Apply centering, scaling and rotation if applicable.
        if (center) {
                for (unsigned j = 0; j < features.size(); ++j)
                    {
                        ptr[j] -= center[j];
                }
        }
        if (scale) {
                for (unsigned j = 0; j < features.size(); ++j)
                    {
```

```
                                        ptr[j] *= scale[j];
                                }
                        }
                        if (rot_mat) {
                                for (unsigned j = 0; j < features.size(); ++j)
                                    {
                                        tmp[j] = 0.0;
                                        for (unsigned k = 0; k < features.size
                                            (); ++k) {
                                                tmp[j] += rot_mat[j * features.
                                                    size() + k] * ptr[j];
                                        }
                                }
                                memcpy(ptr, tmp, features.size() * sizeof(
                                    double));
                        }

                        if (fp) {
                                for (unsigned j = 0; j < features.size(); ++j)
                                    {
                                        fprintf(fp, "%f", ptr[j]);
                                        if (j == features.size() - 1)
                                                fprintf(fp, "\n");
                                        else
                                                fprintf(fp, ",");
                                }
                        }
                        if (cl) {
                                fprintf(cl, "%u\n", songs[i].file_num);
                        }
                }

                if (fp)
                        fclose(fp);
                if (cl)
                        fclose(cl);
}

// Precalculate transformed versions of all vectors, given an inverse
    covariance
// matrix.
double *precalc_transformed_features(double *vecs, double *matrix,
    unsigned num_vec, unsigned num_features)
{
        double *ret = new double[num_vec * num_features];

        for (unsigned i = 0; i < num_vec; ++i) {
                for (unsigned j = 0; j < num_features; ++j) {
                        double sum = 0.0;
                        for (unsigned k = 0; k < num_features; ++k) {
                                sum += vecs[i * num_features + k] *
                                    matrix[j * num_features + k];
                        }
                        ret[i * num_features + j] = sum;
                }
        }

        return ret;
}

// Perform the actual classification, matching all non-references
   against all
// references.
void do_search(unsigned num_features, unsigned stride)
{
        // nearest-neighbor search
        unsigned correct = 0, total = 0;
        unsigned correct_class[7], total_class[7];

        for (unsigned i = 0; i < 7; ++i) {
                correct_class[i] = total_class[i] = 0;
        }

        for (unsigned i = 0; i < (songs.size() - num_ref); ++i) {
                int best_match = -1;
                double best_match_val = HUGE_VAL;

                for (unsigned j = 0; j < num_ref; ++j) {
                        double this_match_val;

                        if (rtransvec == NULL) {
                                // Euclidian distance
                                this_match_val = distance(svec + i *
                                   stride, rvec + j * stride,
                                   num_features);
                        } else {
                                // Mahalanobis distance
                                this_match_val = distancem(svec + i *
                                   stride, rvec + j * stride, stransvec
                                    + i * stride, rtransvec + j *
                                   stride, num_features);
                        }

                        if (best_match == -1 || this_match_val <
                            best_match_val) {
                                best_match = j;
                                best_match_val = this_match_val;
                        }
                }

                if (songs[snum[i]].file_num == songs[rnum[best_match]].
                   file_num) {
                        ++correct;
                        ++correct_class[songs[snum[i]].class_id];
                } else {
                        if (show_misclassified)
                                fprintf(stderr, "Misclassified:␣'%s'␣=>
                                   ␣'%s'\n", songs[snum[i]].filename.
                                   c_str(),
                                        songs[rnum[best_match]].
                                           filename.c_str());
                }
                ++total;
                ++total_class[songs[snum[i]].class_id];

                if (total % 64 == 0)
                        fprintf(stderr, "%5u/%5u␣(%5.1f%%)...␣\r",
                           correct, total, (100.0 * correct / total));
        }

        if (output_tex_filename.size() > 0) {
                FILE *f = fopen(output_tex_filename.c_str(), "wb");
                if (f == NULL) {
                        perror(output_tex_filename.c_str());
                        exit(1);
                }

                for (unsigned i = 1; i <= 6; ++i) {
                        fprintf(f, "%6.1f\\%%␣&␣", 100.0 * (1.0 -
                           correct_class[i] / total_class[i]));
                }

                fprintf(f, "%6.1f\\%%␣\n", 100.0 * (1.0 - correct /
                   total));

                fclose(f);
        }
        if (output_raw_filename.size() > 0) {
                FILE *f = fopen(output_raw_filename.c_str(), "wb");
                if (f == NULL) {
                        perror(output_raw_filename.c_str());
                        exit(1);
                }

                for (unsigned i = 1; i <= 6; ++i) {
                        fprintf(f, "%u␣%u\n", correct_class[i],
                           total_class[i]);
                }

                fprintf(f, "%u␣%u\n", correct, total);

                fclose(f);
        }
        printf("Classifier␣was␣right␣%.1f%%␣of␣the␣time␣(%u/%u).\n",
           (100.0 * correct / total), correct, total);
}
```

```
// Serializes the "songs" array down to a single file for quicker
   reading back.
void write_dump(const std::string &filename)
{
        FILE *f = fopen(filename.c_str(), "wb");
        if (f == NULL) {
                perror(filename.c_str());
                exit(1);
        }

        unsigned num_songs = songs.size();
        fwrite(&num_songs, sizeof(num_songs), 1, f);

        for (std::vector<song>::const_iterator i = songs.begin(); i !=
           songs.end(); ++i) {
                fwrite(&i->file_num, sizeof(i->file_num), 1, f);
                fwrite(&i->subfile_num, sizeof(i->subfile_num), 1, f);
                fwrite(&i->class_id, sizeof(i->class_id), 1, f);

                unsigned filename_len = i->filename.size();
                fwrite(&filename_len, sizeof(filename_len), 1, f);
                fwrite(i->filename.c_str(), filename_len, 1, f);

                unsigned num_descriptors = i->desc.size();
                fwrite(&num_descriptors, sizeof(num_descriptors), 1, f)
                   ;

                for (std::map<std::string, std::vector<double> >::
                   const_iterator j = i->desc.begin(); j != i->desc.end
                   (); ++j) {
                        unsigned key_len = j->first.size();
                        fwrite(&key_len, sizeof(key_len), 1, f);
                        fwrite(j->first.c_str(), key_len, 1, f);

                        unsigned num_elem = j->second.size();
                        fwrite(&num_elem, sizeof(num_elem), 1, f);

                        for (std::vector<double>::const_iterator k = j
                           ->second.begin(); k != j->second.end(); ++k)
                            {
                                fwrite(&*k, sizeof(*k), 1, f);
                        }
                }
        }
        fclose(f);
}

// Reads the "songs" array back from a file created by write_dump().
void read_dump(const std::string &filename)
{
        char buf[4096]; // vulnerable to overflows, but we're only
           reading in trusted data anyhow
        FILE *f = fopen(filename.c_str(), "rb");
        if (f == NULL) {
                perror(filename.c_str());
                exit(1);
        }

        unsigned num_songs;
        fread(&num_songs, sizeof(num_songs), 1, f);

        songs.reserve(num_songs);

        for (unsigned i = 0; i < num_songs; ++i) {
                song s;

                fread(&s.file_num, sizeof(s.file_num), 1, f);
                fread(&s.subfile_num, sizeof(s.subfile_num), 1, f);
                fread(&s.class_id, sizeof(s.class_id), 1, f);

                unsigned filename_len;
                fread(&filename_len, sizeof(filename_len), 1, f);
                fread(buf, filename_len, 1, f);
                buf[filename_len] = '\0';
                s.filename = buf;

                unsigned num_descriptors;
                fread(&num_descriptors, sizeof(num_descriptors), 1, f);

                for (unsigned j = 0; j < num_descriptors; ++j) {
                        unsigned key_len;
                        fread(&key_len, sizeof(key_len), 1, f);
                        fread(buf, key_len, 1, f);
                        buf[key_len] = '\0';

                        std::string key = buf;

                        unsigned num_elem;
                        fread(&num_elem, sizeof(num_elem), 1, f);

                        std::vector<double> elem(num_elem);

                        for (unsigned k = 0; k < num_elem; ++k) {
                                double tmp;
                                fread(&tmp, sizeof(tmp), 1, f);
                                elem[k] = tmp;
                        }

                        s.desc.insert(std::make_pair(key, elem));
                }

                songs.push_back(s);
        }
```

```
        fclose(f);

        fprintf(stderr, "Read␣%u␣songs␣from␣dump␣file.\n", songs.size()
           );
}

// Allocate and read a NUM_ELEM-matrix from a text file.
void read_matrix(double **mat, const std::string &filename, unsigned
   num_elem)
{
        FILE *f = fopen(filename.c_str(), "r");
        if (f == NULL) {
                perror(filename.c_str());
                exit(1);
        }

        *mat = new double[num_elem];

        for (unsigned i = 0; i < num_elem; ++i) {
                double tmp;
                if (fscanf(f, "%lf", &tmp) == 0) {
                        fprintf(stderr, "%s␣contains␣only␣%u␣elements,␣
                           expected␣%u\n",
                                filename.c_str(), i, num_elem);
                        exit(1);
                }
                (*mat)[i] = tmp;
        }

        double tmp;
        if (fscanf(f, "%lf", &tmp) == 0) {
                fprintf(stderr, "%s␣contains␣more␣than␣the␣expected␣%u␣
                   elements\n",
                        filename.c_str(), num_elem);
                exit(1);
        }

        fclose(f);
}

int main(int argc, char **argv)
{
        std::vector<std::string> features;

        songs.reserve(70000);

        int option_index = 0;
        for ( ;; ) {
                int c = getopt_long(argc, argv, "f:o:O:x:X:w:r:d:t:c:s:
                   i:m", longopts, &option_index);
                if (c == -1)
                        break;
```

```
switch (c) {
case 'f':
        if (features.size() > 0) {
                fprintf(stderr, "-f␣given␣twice;␣
                   exiting.\n");
                exit(1);
        }
        features = parse_features(optarg);
        break;
case 'o':
        output_data_filename = optarg;
        break;
case 'O':
        output_class_filename = optarg;
        break;
case 'x':
        output_tex_filename = optarg;
        break;
case 'X':
        output_raw_filename = optarg;
        break;
case 'w':
        dump_filename = optarg;
        do_write_dump = true;
        break;
case 'r':
        dump_filename = optarg;
        do_read_dump = true;
        break;
case 'm':
        show_misclassified = true;
        break;
case 'd':
        num_dimensions = atoi(optarg);
        break;
case 't':
        input_matrix_filename = optarg;
        break;
case 'c':
        input_center_filename = optarg;
        break;
case 's':
        input_scale_filename = optarg;
        break;
case 'i':
        input_inv_cov_filename = optarg;
        break;
default:
        fprintf(stderr, "Unknown␣option␣character␣'%c'\
           n", c);
        exit(1);
```

```
                }
        }

        if (features.size() == 0 || (optind >= argc && !do_read_dump))
           {
                fprintf(stderr, "Usage:␣voronoi␣-f␣FEATURELIST␣FILES␣
                    ...");
                exit(1);
        }
        if (do_read_dump && do_write_dump) {
                fprintf(stderr, "Error:␣--read-dump␣and␣--write-dump␣
                    can␣not␣both␣be␣specified\n");
                exit(1);
        }

        if (do_read_dump) {
                read_dump(dump_filename);
        }

        while (optind < argc) {
                if (argv[optind][0] == '@') {
                        read_filelist(argv[optind] + 1);
                } else {
                        read_file(argv[optind]);
                }
                ++optind;
        }

        if (do_write_dump) {
                write_dump(dump_filename);
        }

        if (input_matrix_filename.size() > 0) {
                read_matrix(&rot_mat, input_matrix_filename, features.
                    size() * features.size());
        }
        if (input_center_filename.size() > 0) {
                read_matrix(&center, input_center_filename, features.
                    size());
        }
        if (input_scale_filename.size() > 0) {
                read_matrix(&scale, input_scale_filename, features.size
                    ());
                for (unsigned i = 0; i < features.size(); ++i) {
                        scale[i] = 1.0 / scale[i];
                }
        }

        extract_features(features);

        if (input_inv_cov_filename.size() > 0) {
                read_matrix(&inv_cov_mat, input_inv_cov_filename,
                   features.size() * features.size());
                rtransvec = precalc_transformed_features(rvec,
                   inv_cov_mat, num_ref, features.size());
                stransvec = precalc_transformed_features(svec,
                   inv_cov_mat, songs.size() - num_ref, features.size()
                   );
        }

        if (num_dimensions == -1) {
                do_search(features.size(), features.size());
        } else if (num_dimensions > 0) {
                do_search(std::min((unsigned)num_dimensions, features.
                   size()), features.size());
        } else {
                fprintf(stderr, "No␣classification␣done.\n");
        }
}
```

Appendix C

Album list

This chapter contains the final album list that was used in the project (the complete track list was deemed too verbose to list), with the number of tracks used from each. Note that as the metadata was automatically added at ripping time, minor errors and inconsistencies may be present.

- 10,000 Maniacs – Blind Man's Zoo (11 tracks), 10,000 Maniacs – In My Tribe (10 tracks), 10,000 Maniacs – MTV Unplugged (14 tracks), 10,000 Maniacs – Our Time In Eden (13 tracks), 10,000 Maniacs – What's the Matter Here (2 tracks), 10000 Maniacs – The Wishing Chair (15 tracks)

- A-ha – Memorial Beach (10 tracks), Abbaye de Sylvanès – Salve Regina des bergers du Rouergue (2 tracks), Abercrombie Holland DeJohnette – Gateway (6 tracks), Alanis Morissette – Jagged Little Pill (13 tracks), Ali Farka Toure – The Source (10 tracks), Alistair Cochrane – Vital Interests (7 tracks), Alnæs Værnes Reiersrud Klakegg – 4G (13 tracks), Amsterdam Symphony Orchestra – Ludwig von Beethoven, Symphony No. 3 and 9 (5 tracks), Anderson, Bruford, Wakeman, Howe – Anderson, Bruford, Wakeman, Howe (9 tracks), Andrew Charles Newcombe, David Scott Hamnes – Mantra (14 tracks), Anew Voice – Anew Voice (10 tracks), Anita Baker – Rhythm Of Love (12 tracks), Anne Grete Preus – Alfabet (13 tracks), Anne Grete Preus – Fullmåne (14 tracks), Anne Grete Preus – Lav sol! Høy himmel. (9 tracks), Anne Grete Preus – Millimeter (9 tracks), Anne Grete Preus – Månens elev (promo CD-single) (one track), Anne Grete Preus – Når dagen roper (11 tracks), Anne Grete Preus – Og høsten kommer tidsnok (9 tracks), Anne Sofie Von Otter Meets Elvis Costello – For The Stars (18 tracks), Anneli Drecker – Tundra (11 tracks), Antonin Dvorak – Cello Concerto (Truls Mørk) (12 tracks), Antonsen, Marianne – Pickin' up the spirit (13 tracks), Aqua – Aquarius (12 tracks), Arild Andersen – Arv (16 tracks), Arild Andersen – If you Look Far Enough (12 tracks), Arild Andersen – Kristin Lavransdatter (18 tracks), Arild Andersen – Sagn (15 tracks), Atomic Swing – A car crash in the blue (10 tracks)

- Baby Blue – Baby Blue (11 tracks), Beatles – Live At The BBC - CD2 (35 tracks), Beatles – Revolver (14 tracks), Beatles, The – Abbey Road (17 tracks), Bel Canto – Magic Box (11 tracks), Bel Canto – Shimmering, Warm & Bright (10 tracks), Bendik Hofseth – IX (11 tracks), Benny Green & Russell Malone – Jazz at the Bistro (15 tracks), Bette Midler – Beaches Soundtrack (10 tracks), Big Country – Peace In Our Time (11 tracks), Big Country – The Seer (10 tracks), Bill Evans – At the Montreux Jazz Festival (11 tracks), Billy Joel – Storm Front (10 tracks), Bjørn Alterhaug – A Ballad (11 tracks), Bjørn Eidsvåg – Allemannsland (11 tracks), Bjørn Eidsvåg – Alt du vil ha (14 tracks), Bjørn Eidsvåg – Landet lenger bak (13 tracks), Bjørn Eidsvåg – På svai (13 tracks), Bjørn Eidsvåg – Tapt uskyld (12 tracks), Bob Dylan – Bob Dylan at Budokan (CD 1) (11 tracks), Bob Dylan – Bob Dylan at Budokan (CD 2) (11 tracks), Bob Gaudio – Little Shop Of Horrors (13 tracks), Bob Geldof – Deep In The Heart Of Nowhere (14 tracks), Bob Geldof

– Room 19 (Sha La La La Lee) [Maxi] (4 tracks), Bob Geldof – The Happy Club (12 tracks), BobbyZ – BobbyZ (9 tracks), Boomtown Rats – Greatest Hits (10 tracks), Bremnes, Kari – Gåte ved gåte (10 tracks), Bremnes, Kari – Mitt Ville Hjerte (12 tracks), Bremnes, Kari – Månestein (11 tracks), Bremnes, Kari – Svarta Bjørn (10 tracks), Bruce Springsteen – The Ghost Of Tom Joad (12 tracks), Brunborg - Wesseltoft - Jormin - Christensen – Tid (9 tracks), Bud Powell – Jazz At Massey Hall, Vol. 2 (16 tracks), Bård Wessel – Atlantic Traveller (12 tracks)

- Camel – Music Inspired By The Snow Goose 2002 Remastered Expand (18 tracks), Carlos Santana - Mahavishnu John McLaughlin – Love Devotion Surrender (7 tracks), Carola – Blott En Dag (11 tracks), Charles Mingus – Mingus Ah Um (11 tracks), Charles Mingus – Mingus At Antibes (6 tracks), Charles Mingus – Mingus Mingus Mingus Mingus Mingus (8 tracks), Charles Mingus – New Tijuana Moods (9 tracks), Charlie Parker – Charlie Parker At Storyville (9 tracks), Chick Corea Electric Band – Light Years (12 tracks), Christopher Hogwood – The Academy of Ancient Music (12 tracks), Clawfinger – Warfair (4 tracks), Codona - Walcott, Cherry & Vasconcelos – Codona II (6 tracks), Cornelis Vreeswijk – Guldkorn Från Mäster Cees Memoarer (19 tracks), Cornelis Vreeswijk – Guldkorn Från Mäster Cees Memoarer Vol. 2 (20 tracks), Cornelis Vreeswijk – Mäster Cees Memoarer - Vol. 3 (23 tracks), Cornelis Vreeswijk – Mäster Cees Memoarer - Vol. 4 (25 tracks), Cornelis Vreeswijk – Mäster Cees Memoarer - Vol. 5 (28 tracks), Count Basie, Oscar Peterson – The Timekeepers: Count Basie Meets Oscar Peterson (7 tracks), Cowboy Junkies – Pale Sun, Crescent Moon (12 tracks), Cowboy Junkies – The Trinity Session (12 tracks)

- Dalbello – whomanfoursays (9 tracks), Dance With A Stranger – Look What You've Done (11 tracks), Dave Holland Quartet – Conference Of The Birds (6 tracks), Dave Stewart and the Spiritual Cowboys – Spiritual Cowboys 1990 - about the album (23 tracks), David Holland – Life Cycle (11 tracks), David Motion and Sally Potter – Orlando (16 tracks), David Sanborn – David Sanborn (8 tracks), Del Amitri – Change Everything (12 tracks), deLillos – Enda mere (4 tracks), deLillos – Før var det morsomt med sne (14 tracks), deLillos – Hjernen Er Alene (20 tracks), deLillos – Suser avgårde (14 tracks), Delillos – Svett Smil (14 tracks), deLillos – Sveve over byen (Re Mix) (4 tracks), deLillos – Varme Mennesker (12 tracks), Depeche Mode – Exciter (13 tracks), Depeche Mode – Only When I Lose Myself (Maxi) (3 tracks), Depeche Mode – Ultra (12 tracks), Di derre – Jenter & Sånn - 11 sanger om jenter og én om sånn. (12 tracks), Dire Straits – Love Over Gold (5 tracks), Dire Straits – Money For Nothing (12 tracks), Disney Characters – Disney Presents A Family Christmas (18 tracks), Dolly Parton – Legends (Disc 1) (15 tracks), Dolly Parton – Legends (Disc 2) (16 tracks), Dolly Parton – Legends (Disc 3) (16 tracks), Dolly Parton – Ultimate Dolly Parton (20 tracks), Doors – Waiting For The Sun (11 tracks), Douglas Wood – Deep Woods (11 tracks), Dronning Mauds Land – Dronning Mauds Land (10 tracks), Duke Ellington & John Coltrane – Duke Ellington & John Coltrane (7 tracks)

- Eddie Cochran – Lil' Bit Of Gold (4 tracks), Edie Brickell & New Bohemians – Shooting Rubberbands at the Stars (12 tracks), Edvard Hoem, Henning Sommerro, Hildegun Riise, Voskresenije – Den fattige Gud - Salmar av Edvard Hoem (12 tracks), Elgar - LSO - Barbirolli - Baker – Elgar Cello Concerto in E minor, Op.85 - Sea Pictures Op.37 (9 tracks), Ella Fitzgerald – 16 Greatest Hits (17 tracks), Ella Fitzgerald – The Harold Arlen Songbook (Volume One) (13 tracks), Ella Fitzgerald & Louis Armstrong – Ella & Louis Again (12 tracks), Elton John – Reg Strikes Back (10 tracks), Elvis Costello – Deep Dark Truthful Mirror (4 tracks), Elvis Costello – Mighty Like A Rose (14 tracks), Elvis Costello – Spike (14 tracks), Elvis Costello and the Attractions – Get Happy!! (20 tracks), Elvis Costello and The Brodsky Quartet – The Juliet Letters (20 tracks), Enya – Amarantine (12 tracks), Enya – How Can I Keep From Singing (3 tracks), Enya – Shepherd Moons (12 tracks), Enya – Watermark (11 tracks), Erasure – Chorus (10 tracks), Eric Leeds – Times Squared (11 tracks), Everything But The Girl – Idlewild (11 tracks), Everything But The Girl – Missing (The remix EP) (4 tracks), Extended Noise – Slow but suden Langsam aber plötzlich (9 tracks)

- Fairground Attraction – A smile in a whisper (4 tracks), Fairground Attraction – Original Hit Singles (4 tracks), Ferenc Snetberger – Nomad (10 tracks), Fine Young Cannibals – The Raw & The Cooked (10 tracks), Focus – Live at the BBC (9 tracks), Forskjellige artister – Perleporten (Frelsesarmeen) (14 tracks), Four Men and a Dog – Barking Mad (13 tracks), Fra Lippo Lippi – The Colour Album (9 tracks), Francis Lai, Philippe Servain – La Belle Histoire (16 tracks), Frankie Goes To Hollywood – Relax [Maxi CD] (6 tracks), Frelsesarmeen – Stolpesko (11 tracks), Frode Alnes, Arild Andersen, Stian Carstensen – Julegløggen (16 tracks), Frøydis Armand, Stein Mehren & Ketil Bjørnstad – For den som elsker (15 tracks)

- Gabrielle – Find Your Way (10 tracks), Garbarek Jan – Visible World (15 tracks), Gateway – Homecoming (9 tracks), Geirr Lystrup – Samme gamle greia - nye sanger om vår og kjærlighet (12 tracks), Gentle Giant – Acquiring The Taste (8 tracks), Gentle Giant – Gentle Giant (7 tracks), Gentle Giant – Octopus (8 tracks), Gentle Giant – Three Friends (6 tracks), George Frideric Händel – Messiah (CD1) (15 tracks), George Frideric Händel – Messiah (CD2) (15 tracks), George Harrison – When We Was Fab (4 tracks), George Michael – Listen Without Prejudice Vol. 1 (10 tracks), Gloria Estefan – Cuts Both Ways (12 tracks), Grieg - Dreier – Peer Gynt, incidental music (17 tracks), Grieg - Dreier – Peer Gynt, incidental music (CD2) (13 tracks), Grieg – Peer Gynt (11 tracks)

- Halvdan Sivertsen – Frelsesarmeens Juleplate (12 tracks), Hothouse Flowers – Home (14 tracks), Händel, Georg Friedrich – Wassermusik & Feuerwerksmusik (CD1) (18 tracks)

- Indigo Girls – Rites Of Passage (13 tracks), Ivar Antonsen – Double Circle (11 tracks), Iver Kleive – Alle Menschen müssen sterben (5 tracks), Iver Kleive – Hyrdenes Tilbedelse - Meditasjoner over kjente julesanger (15 tracks), Iver Klieve & Knut Reiersrud – Nåde (one track)

- J. S. Bach – Toccata (Peter Hurford) (4 tracks), J.A.C. Redford – Oliver & Company (11 tracks), Jacob Young – Evening Falls (9 tracks), James Ingram – Always You (10 tracks), Jamiroquai – The Return Of The Space Cowboy (10 tracks), Jan Akkerman & Thijs Van Leer – Focus (7 tracks), Jan Eggum – Eggum (10 tracks), Jan Eggum – Underveis (11 tracks), Jan Garbarek - Ustad Fateh Ali Khan – Ragas And Sagas (5 tracks), Jan Garbarek – All Those Born With Wings (6 tracks), Jan Garbarek – Dis (6 tracks), Jardar Eggesbø Abrahamsen – Noko (19 tracks), Jarrett, Keith – Paris Concert (3 tracks), Javed Bashir, Sondre Bratland – Dialogue (13 tracks), Jean-Michel Jarre – Musik aus Zeit und Raum (13 tracks), Jeff Lynne – Armchair Theatre (11 tracks), Jethro Tull – 20 Years Of Jethro Tull (21 tracks), Jethro Tull – In Concert (10 tracks), Jethro Tull – This Was (10 tracks), Jimi Hendrix – Are You Experienced (11 tracks), Jimi Hendrix – The Cry of Love (10 tracks), Jimi Hendrix – War Heroes (9 tracks), Jmre Szabo, Hans-Christoph Becker-Foss – Romantic Organ Works (6 tracks), John Abercrombie - Dave Holland - Jack DeJohnette – Gateway 2 (5 tracks), John Coltrane – A Love Supreme (4 tracks), John Coltrane – A Love Supreme (CD 2) (9 tracks), John Coltrane – Coltrane Plays The Blues (7 tracks), John Coltrane – Giant Steps (12 tracks), John Coltrane & Don Cherry – The Avant-Garde (5 tracks), John McLaughlin – Extrapolation (10 tracks), John Scofield & Pat Metheny – I Can See Your House From Here (11 tracks), John Surman – Nordic Quartet (9 tracks), Jon Eberson – STASH (14 tracks), Jonathan Richman – The Best Of Jonathan Richman And The Modern Lovers (18 tracks), Joni Mitchell – Blue (8 tracks), Joni Mitchell – Song to a Seagull (10 tracks), Joni Mitchell – Turbulent Indigo (10 tracks), Jordi Savall – Tous les matins du monde (16 tracks), Joybells – Joy & Praise (12 tracks), Jukkis Uotila – Introspection (7 tracks), Jøkleba – Live (10 tracks)

- KABAT – Devky ty to znaj (18 tracks), Kari Bremnes – 11 ubesvarte anrop (11 tracks), Kari Bremnes – Blå Krukke (10 tracks), Kari Bremnes – Folk i husan (17 tracks), Kari Bremnes – Løsrivelse (15 tracks), Kari Bremnes – Over en by (14 tracks), Kari Bremnes – Spor (11 tracks), Kari Bremnes Rikard Wolff – Desemberbarn (16 tracks), Kari Bremnes & Lars Klevstrand – Tid å hausta inn (19 tracks), Kari og Ola Bremnes – Mit Navn er Petter Dass (14 tracks), Kari, Ola

& Lars Bremnes – Soløye (14 tracks), Kari, Ola, og Lars Bremnes – Ord fra en fjord (12 tracks), Karin Krog – Something borrowed ... Something new (13 tracks), Karin Krog – Where you at (11 tracks), Kate Bush – Aerial (CD1) (7 tracks), Kate Bush – Aerial (CD2) (9 tracks), Kate Bush – Hounds Of Love (12 tracks), Kate Bush – Lionheart (10 tracks), Kate Bush – Never for Ever (11 tracks), Kate Bush – The Kick Inside (13 tracks), Kate Bush – The Red Shoes (12 tracks), Kate Bush – The Sensual World (10 tracks), Kate Bush – The Sensual World (single) (2 tracks), Keith Jarrett - Gary Peacock - Jack DeJohnette – Standards, Vol.1 (5 tracks), Keith Jarrett – Personal Mountains (5 tracks), Keith Jarrett – The Köln Concert (4 tracks), Keith Jarrett – The Melody At Night, With You (10 tracks), Keith Jarrett, Gary Peacock, Jack DeJohnette – The Out-of-Towners (6 tracks), Keith Jarrett, Gary Peacock, Jack DeJohnette – Whisper Not - Live In Paris 1999 (Disc 1) (7 tracks), Keith Jarrett, Gary Peacock, Jack DeJohnette – Whisper Not - Live In Paris 1999 (Disc 2) (7 tracks), Ketil Bjørnstad – Odyssey (11 tracks), Ketil Bjørnstad m.fl. – Himmelrand - Tusenårsoriatoriet (21 tracks), Ketil Bjørnstad & David Darling – The River (10 tracks), Kine Hellebust – 15 salmar og 1 song - Kine Hellebust syng Elias Blix (16 tracks), KJØTT – 1979 - 1981 (12 tracks), Knauskoret – Stars in stripes -evt. Et liv med striper (13 tracks), Knut Reiersrud – Sub (16 tracks), Knut Reiersrud – Sweet Showers Of Rain (10 tracks), Knut Reiersrud og Iver Kleive – Nåde over Nåde (12 tracks), Kor på menyen – Kor på menyen (10 tracks), Kristusbilder fra salmeboken – Smak av himmel - spor av jord - cd1 (14 tracks), Kristusbilder fra salmeboken – Smak av himmel - spor av jord - cd2 (15 tracks)

- Lars Bremnes – Søndag (10 tracks), Legoland Band – Legoland Garden (14 tracks), Lenny Kravitz – 5 (13 tracks), Lenny Kravitz – Circus (11 tracks), Leonard Cohen – The Essential Leonard Cohen (Disc 2) (13 tracks), Leonard Cohen – The Essential Leonard Cohen [Disc 1] (18 tracks), Les Négresses Vertes – Mlah (14 tracks), Lillebjørn Nilsen – 40 spor (CD1) (20 tracks), Lillebjørn Nilsen – 40 spor (CD2) (21 tracks), Lou Reed – New York (14 tracks), Lou Reed – Transformer [Expanded Edition] (13 tracks), Lou Reed & John Cale – Songs For Drella (14 tracks), Lynni Treekrem – Ut i vind (10 tracks)

- Madonna – Bedtime Stories (11 tracks), Madonna – Confessions On A Dance Floor (10 tracks), Madonna – I'm Breathless (12 tracks), Madonna – Ray of Light (13 tracks), Magni Wentzel – All Or Nothing At All (15 tracks), Magni Wentzel – My Wonderful One (18 tracks), Magni Wentzel, Roger Kellaway & Red Michell – New York Nights (11 tracks), Mano Negra – Puta's Fever (18 tracks), Mari Boine – Gávcci jahkejuogu (Eight seasons) (12 tracks), Marianne Antonsen – Blomster i Soweto (11 tracks), Marie Fredriksson – Efter Stormen (12 tracks), Marie Fredriksson – Het vind (11 tracks), Marius Muller – Seks (12 tracks), Marius Müller's Funhouse – BIG (9 tracks), Mark Knopfler – Local Hero (14 tracks), Mary Black – Babes In The Wood (12 tracks), Mary Black – Circus (12 tracks), Matchstick Sun – Flowerground (11 tracks), Matchstick Sun – Itchy Bitchy (12 tracks), Matt Molloy, Paul Brady, Tommy Peoples – Molloy - Brady - Peoples (15 tracks), Mavis Staples – The Voice (12 tracks), McCartney & Davis – Paul McCartney's Liverpool Oratorio - RLPO, Davis (EMI 1991) (Disc 1) (24 tracks), McCartney & Davis – Paul McCartney's Liverpool Oratorio - RLPO, Davis (EMI 1991) (Disc 2) (26 tracks), Metallica – Enter Sandman (Single CD) (3 tracks), Michael Jackson – Dangerous (14 tracks), Michael Monroe – Simple Life (10 tracks), Michael Nyman – Gattaca (24 tracks), Michael Nyman – The Cook The Thief His Wife Her Lover (5 tracks), Michael Nyman – The Piano (OST) (19 tracks), Michelle Shocked – Captain Swing (11 tracks), Michelle Shocked – The Texas Campfire Tapes (12 tracks), Mick Jagger – Wandering Spirit (14 tracks), Midnight Oil – Blue Sky Mining (10 tracks), Mikael Wiehe – Sevilla (10 tracks), Mike Lawrence – Nightwind (7 tracks), Mike Oldfield – Amarok (one track), Mike Oldfield – Earth Moving (9 tracks), Mike Oldfield – Islands (USA) (7 tracks), Mike Oldfield – Orchestral Tubular Bells (2 tracks), Mike Oldfield – The Killing Fields (17 tracks), Mike Stern – Odds Or Evens (8 tracks), Miles Davis – Agharta (CD1) (one track), Miles Davis – Agharta (CD2) (2 tracks), Miles Davis – Bopping The Blues (9 tracks), Miles Davis – Cookin' With The Miles Davis Quintet (4 tracks), Miles Davis – In a Silent Way (2 tracks), Miles Davis – Miles Ahead (14 tracks), Miles Davis – Miles in Antibes (5 tracks), Miles Davis – Nefertiti (6 tracks), Miles Davis

– On the Corner (4 tracks), Miles Davis – Quiet Nights (8 tracks), Miles Davis – Seven steps to heaven (5 tracks), Miles Davis – Sketches Of Spain (8 tracks), Miles Davis – Steamin' With The Miles Davis Quintet (5 tracks), Miles Davis – This is Jazz - Miles Davis - Acoustic (7 tracks), Miles Davis – Tutu (8 tracks), Miles Davis & John Coltrane – Miles & Coltrane (7 tracks), Miles Davis & Quincy Jones – Live At Montreux (14 tracks), Millions Like Us – ...millions like us (11 tracks), Moloney, O'Connell, Keene – Kilkelly (5 tracks), Monica Zetterlund – Varsamt (11 tracks), Monty Alexander Trio – Impressions in Blue (11 tracks), Monty Alexander_Ray Brown_Herb Ellis – Triple Treat II (8 tracks), Monty Python – Monty Python Sings (25 tracks), Morcheeba – Big Calm (11 tracks), Mork, Truls, Thibaudet, Jean Yves – Grieg - Sibelius - Cello nad Piano Works (11 tracks), Mormon Tabernacle Choir – Silent Night (11 tracks), Morten Harket – Wild Seed (12 tracks), Motorpsycho – 8 soothing songs for Rut (8 tracks), Motorpsycho – Angels And Daemons At Play (11 tracks), Motorpsycho – Black Hole - Blank Canvas (CD1) (8 tracks), Motorpsycho – Black Hole Blank Canvas (CD2) (9 tracks), Motorpsycho – Blissard (10 tracks), Motorpsycho – Demon Box (14 tracks), Motorpsycho – Hey Jane (EP) (5 tracks), Motorpsycho – Let Them Eat Cake (9 tracks), Motorpsycho – Mountain EP (5 tracks), Motorpsycho – Phanerothyme (9 tracks), Motorpsycho – The Nerve Tatto E.P. (5 tracks)

- Nick Cave And The Bad Seeds – Murder Ballads (10 tracks), Nidarosdomeds Guttekor – Tidenes Juleplate (14 tracks), Niels Henning Ørsted Pedersen & Sam Jones – Double Bass (10 tracks), Niels-Henning Orsted Pedersen (NHOP) – This Is All I Ask (10 tracks), Niels-Henning Orsted Pedersen & Palle Mikkelborg – Hommage - Once Upon A Time (9 tracks), Nirvana – Nevermind (12 tracks), Norah Jones – Come Away With Me (14 tracks), Norah Jones – Feels Like Home (13 tracks), NYMARK COLLECTIVE – Contemporary Tradition (13 tracks)

- Oasis – (What's The Story) Morning Glory (11 tracks), Oasis – Definitely Maybe (11 tracks), October Project – bury my lovely (CD single) (3 tracks), Odd Børretzen - Lars Martin Myhre – Noen ganger er det all right (12 tracks), Odd Børretzen og Alf Cranner – Hva Er Det De Vil? (12 tracks), Odd Børretzen og Julius Hougen – På den ene siden - på den andre siden (7 tracks), Ola Bremnes - Bodø Domkor – Vær Hilset (15 tracks), Ole Paus - Oslo Kammerkor – Det begynner å bli et liv - det begynner å ligne en bønn (6 tracks), Ole Paus – Stjerner i rennesteinen (12 tracks), Ole Paus, Mari Boine Persen, Kari Bremnes – Salmer på veien hjem (12 tracks), Oscar Peterson Trio – The Trio (12 tracks), Oscar Peterson, Benny Green – Oscar and Benny (10 tracks), Oslo Gospel Choir – Det skjedde i de dager (11 tracks), Oslo Gospel Choir – Get Up ! (11 tracks)

- Paul Brady – Songs and Crazy Dreams (12 tracks), Paul Simon – Concert In The Park - Disc 1 (11 tracks), Paul Simon – Concert In The Park - Disc 2 (12 tracks), Pavement – Crooked Rain, Crooked Rain - L.A.'s Desert Origins (Disc 1) (24 tracks), Pavement – Crooked Rain, Crooked Rain - L.A.'s Desert Origins (Disc 2) (25 tracks), Peter Sellers – A Hard Day's Night (Single) (4 tracks), Petr Eben – Organ Works (Hallgeir Schiager, orgel) (13 tracks), Phil Collins – Hello, I Must Be Going! (10 tracks), Pilgrimage – 9 Songs Of Ecstasy (9 tracks), Pink Floyd – A Collection of Great Dance Songs (6 tracks), Pink Floyd – The Wall (CD1) (13 tracks), Pink Floyd – The Wall (Disc 2) (13 tracks), Poi Dog Pondering – Poi Dog Pondering (10 tracks), Poi Dog Pondering – Volo Volo (14 tracks), Portishead – Dummy (11 tracks), Prince – 1999 (10 tracks), Prince – 3121 (12 tracks), Prince – A time 2 dream (12 tracks), Prince – Batman - Motion Picture Soundtrack (9 tracks), Prince – Chaos And Disorder (9 tracks), Prince – Come (10 tracks), Prince – Controversy (7 tracks), Prince – Dinner With Delores (3 tracks), Prince – Dirty Mind (8 tracks), Prince – For You (8 tracks), Prince – Interactive (one track), Prince – Letitgo (6 tracks), Prince – Lovesexy (one track), Prince – Music From Graffiti Bridge (17 tracks), Prince – Musicology (12 tracks), Prince – Prince (8 tracks), Prince – Sign 'O' The Times (Disc 2) (6 tracks), Prince – Sign 'O' The Times (Disk 1) (9 tracks), Prince – Space (4 tracks), Prince – The Beautiful Experience (6 tracks), Prince – The Gold Experience (17 tracks), Prince – The Hits 1 (14 tracks), Prince – The Hits 2 (16 tracks), Prince – The Hits The B Sides (Disc 3) (20 tracks), Prince – The Legendary Black Album (8 tracks), Prince – The Most Beautiful Girl in The World (CD Single) (2 tracks), Prince – The Vault... Old Friends 4 Sale (10 tracks), Prince And The New Power Generation – 7 (4 tracks), Prince and the New Power Generation – Love Symbol (18 tracks), Prince And The Revolution – Parade (Under the Cherry Moon) (12 tracks), Prince and The Revolution – Purple Rain (9 tracks), Prince & The N.P.G. – 4AM JAM (9 tracks), Prince & The N.P.G. – Diamonds & Pearls - Single (3 tracks), Prince & The New Power Generation – Diamonds and Pearls (13 tracks), Prince & The Revolution – Around The World In A Day (8 tracks), Proclaimers – Sunshine on Leith (12 tracks)

- R.E.M. – Dead Letter Office [The I.R.S. Years Vintage 1987] (22 tracks), R.E.M. – Near Wild Heaven (CD Single) (4 tracks), Radka Toneff – Fairytales (10 tracks), Radka Toneff – Live in Hamburg (11 tracks), Raga Rockers – BLAFF (12 tracks), Ray Brown Trio – Bassface - Live at Kuumbwa (9 tracks), Ray Brown Trio – Live At Scullers (7 tracks), Red Hot Chili Peppers – Blood Sugar Sex Magik (17 tracks), Richie Havens – Cuts to the Chase (13 tracks), Ritchie Valens – Lil' Bit Of Gold (4 tracks), Robbie Williams – Intensive Care (12 tracks), Roger Waters – Amused To Death (14 tracks), Roy Orbison – Mystery Girl (10 tracks)

- Sade – Love Deluxe (9 tracks), Santana – Abraxas (9 tracks), Sarah McLachlan – Afterglow (10 tracks), Sarah McLachlan – Fumbling Towards Ecstasy (13 tracks), Sarah McLachlan – Solace (11 tracks), Sarah McLachlan – Surfacing (10 tracks), Sarah McLachlan – Touch (10 tracks), Schola Sanctae Sunnivae – Fingergullofficiet (21 tracks), Shankar – Vision (5 tracks), Sidsel Endresen – Exile (11 tracks), Sidsel Endresen – So I Write (8 tracks), Sidsel Endresen – Undertow (8 tracks), Sidsel Endresen & Bugge Wesseltoft – Duplex Ride (11 tracks), Sidsel Endresen & Bugge Wesseltoft – Nightsong (10 tracks), Sidsel Endresen & Christian Wallumrød – Merriwinkle (13 tracks), Sigurd Ulveseth Quartet – Infant eyes (9 tracks), Sigvart Dagsland – Det er makt i de foldede hender (12 tracks), Simple Minds – Glittering Prize (16 tracks), Simple Minds – Good News From The Next World (9 tracks), Simple Minds – Love Song (Maxi) (4 tracks), Simple Minds – Simple Minds [CDS] (3 tracks), Simple Minds – Street Fighting Years (11 tracks), Siri Gjære – Survival kit (11 tracks), Siri"s Svale Band – Necessarily so... (11 tracks), Siri's Svale Band – Blackbird (9 tracks), Sissel Kyrkjebø – Sissel (12 tracks), Skunk Anansie – Paranoid & Sunburnt (11 tracks), Skunk Anansie – Post Orgasmic Chill (12 tracks), Skunk Anansie – Stoosh (12 tracks), Slim Dunlap – Times Like This (11 tracks), Soundtrack – Barb Wire (11 tracks), Stabat Mater – Stabat Mater (15 tracks), Stan Getz & Bill Evans – Stan Getz & Bill Evans (11 tracks), Stan Ridgway – Mosquitos (10 tracks), Steeles – Heaven Help Us All (10 tracks), Steve Taylor – Squint (10 tracks), Steve Wynn – Kerosene Man (11 tracks), Steve Wynn – Kerosene Man (Single CD) (5 tracks), Steve Wynn – Take Your Flunky And Dangle (11 tracks), Stevie Wonder – Fulfillingness' First Finale (10 tracks), Stevie Wonder – Innervisions (Gold Disc) (9 tracks), Stevie Wonder – Songs In The Key of Life (Disc 1) (10 tracks), Stevie Wonder – Songs In The Key of Life (Disc 2) (11 tracks), Sting - – If I Ever Lose My Faith In You (4 tracks), Sting – Love Is Stronger Than Justice - The Munificent Seven (Maxi) (4 tracks), Sting – Seven Days - (CD-Single) (4 tracks), Sting – Songs From The Labyrinth (23 tracks), Sting – Ten Summoner's Tales (12 tracks), Susanne Lundeng – Drag (13 tracks), Susanne Lundeng – Vals til den røde fela (12 tracks), Susanne Lundeng – Ættesyn (15 tracks), Suzanne Vega – 99.9 F (13 tracks), Suzanne Vega – Days Of Open Hand (11 tracks), Suzanne Vega – In Liverpool (4 tracks), Suzanne Vega – Solitude Standing (11 tracks), Suzanne Vega – Suzanne Vega (10 tracks), Suzanne Vega – Suzanne Vega Single Mini-Album 3 (3 tracks)

- Take 6 – Take 6 (9 tracks), Talking Heads – Stop Making Sense (9 tracks), Tanita Tikaram – The Sweet Keeper (10 tracks), Terence Trent D'Arby – Symphony Or Damn (16 tracks), Terence Trent D'Arby – Vibrator (13 tracks), Terje Isungset – Igloo (10 tracks), Terje Rypdal – Bleak house (6 tracks), Terje Rypdal – Descendre (6 tracks), Terje Rypdal – If Mountains Could Sing (11 tracks), Terje Rypdal – To Be Continued (7 tracks), Terje Rypdal og Ronni Le Tekrø – Rypdal og Tekrø (9 tracks), Terje Rypdal & The Chasers – Blue (8 tracks), The Beatles – Live At The BBC (CD1) (34 tracks), The Beatles – Please Please Me (14 tracks), The Beatles – Sgt. Pepper's Lonely Hearts

Club Band (13 tracks), The Beautiful South – Welcome To The Beautiful South (11 tracks), The Chieftains – The Bells of Dublin (23 tracks), The Chieftains – The Long Black Veil (13 tracks), The Commitments – The Commitments (14 tracks), The Draghounds – Angel Boots (14 tracks), The Dream Syndicate – The Day Before Wine & Roses-(Live at KPFK, Sept. 5, 1982) (9 tracks), The Jimi Hendrix Experience – Axis - Bold As Love (13 tracks), The Jimi Hendrix Experience – Electric Ladyland (16 tracks), The Pogues – Hell's ditch (12 tracks), The Pogues – If I Should Fall From Grace With God (15 tracks), The Pogues – Peace And Love (14 tracks), The Pogues – Red Roses for Me (16 tracks), The Pogues – Rum Sodomy & the Lash (13 tracks), The Pogues – White city (3 tracks), The Police – Ghost In The Machine (11 tracks), The Police – Outlandos d'Amour (10 tracks), The Police – Reggatta de Blanc (11 tracks), The Police – Synchronicity (11 tracks), The Police – Zenyatta Mondatta (11 tracks), The Prodigy – Music For The Jilted Generation (13 tracks), The Quintet – Jazz At Massey Hall (6 tracks), The Rembrandts – I'll Be There for You (4 tracks), The September When – Hugger Mugger (11 tracks), The Style Council – The Singular Adventures of the Style Council (16 tracks), The Waterboys – Dream Harder (12 tracks), The Waterboys – Room To Roam (17 tracks), The Waterboys – This Is The Sea (9 tracks), Thelonious Monk – Criss-Cross (12 tracks), Thelonious Monk – With John Coltrane (6 tracks), They Might Be Giants – Flood (19 tracks), Thorbjørn Egner, Christian Hartmann, Egil Monn-Iversen – Klatremus og de andre dyrene i Hakkebakkeskogen (one track), Tin Machine – Tin Machine (14 tracks), Tom Lehrer – In Concert (23 tracks), Tom Lehrer – Tom Lehrer Revisited (15 tracks), Tom Waits – Alice (15 tracks), Tom Waits – Blood Money (13 tracks), Tom Waits – Foreign Affairs (9 tracks), Tom Waits – Mule Variations (16 tracks), Tom Waits – Orphans - Bastards (20 tracks), Tom Waits – Orphans - Bawlers (20 tracks), Tom Waits – Orphans - Brawlers (16 tracks), Tom Waits – The Black Rider (20 tracks), Tomasz Stanko – Selected Recordings (ECM) (12 tracks), Tomasz Stanko Quartet – Lontano (9 tracks), Trond-Viggo Torgersen og Eivind Solås – Samleplate (22 tracks)

- U2 – Achtung Baby (12 tracks), U2 – How to Dismantle an Atomic Bomb (11 tracks)
- Vadested – Sanger fra Iona (10 tracks), Vamp – Horisonter (13 tracks), Van Morrison – Hymns To The Silence - CD 1 (10 tracks), Van Morrison – Hymns To The Silence - CD 2 (11 tracks), Van Morrison & The Chieftains – Irish Heartbeat (10 tracks), Vangelis – Blade Runner (12 tracks), Various – 1-800-NEW-FUNK (11 tracks), Various – 16 Big Hits From The Early 60's (16 tracks), Various – A Love Affair - The Music Of Ivan Lins (11 tracks), Various – Bringing It All Back Home BBC (Disk 2) (20 tracks), Various – CD 7 - Rock Furore (8 tracks), Various – Classical Preview Single, No. 1 (6 tracks), Various – Falling From Grace (Soundtrack) (13 tracks), Various – Folkways - A Vision Shared (14 tracks), Various – High Fidelity Reference CD No. 9 (15 tracks), Various – High Fidelity Reference [Disc 11] (13 tracks), Various – High Fidelity Reference [Disc 1] (19 tracks), Various – High Fidelity Reference [Disc 2] (17 tracks), Various – High Fidelity Reference [Disc 4] (16 tracks), Various – High Fidelity Reference [Disc 5] (16 tracks), Various – High Fidelity Reference [Disc 6] (16 tracks), Various – High Fidelity Reference-CD (Disc 7) (21 tracks), Various – La Bamba (12 tracks), Various – Natural Born Killers (27 tracks), Various – Norske Utslipp - Støtteplaten for Bellona (13 tracks), Various – Norwegian Wood Festival 1996 (10 tracks), Various – On The Road Again (21 tracks), Various – Pretty In Pink (10 tracks), Various – Red Hot + Blue (20 tracks), Various – Rock Furore - CD10 (8 tracks), Various – Rock Furore - CD11 (7 tracks), Various – Rock Furore - CD6 (8 tracks), Various – Rock Furore - CD8 (8 tracks), Various – Stay Awake - Various Interpretations of Music from Vintage Disney Films (11 tracks), Various – Surround Sounds 2 (13 tracks), Various – The Big Lebowski (14 tracks), Various – The Last Temptation of Elvis [CD1] (13 tracks), Various – The Last Temptation of Elvis [CD2] (13 tracks), Various – The Tree and the Bird and the Fish and the Bell (12 tracks), Various Artists – Bringing It All Back Home (Disk 1) (17 tracks), Various Artists – Hadde månen en søster - Cohen på norsk (12 tracks), Various Artists – High Fidelity Reference [Disc 3] (21 tracks), Various Artists – Irish Folk Favourites (Disc 1) (19 tracks), Various Artists – Irish Folk Favourites (Disc 2) (20 tracks), Various Artists – Irish Folk Favourites (Disc 3) (18 tracks), Various Artists – Irish Folk Favourites (Disc 4) (23 tracks), Various Artists – Kirkelig Kulturverksted 25

år - Varige spor (CD 1) (18 tracks), Various Artists – Kirkelig Kulturverksted 25 år - Varige spor (CD 2) (16 tracks), Various Artists – Rock Furore - CD9 (8 tracks), Various Artists – Until The End Of The World (19 tracks), Vassilis Tsabropoulos, Arild Andersen & John Marshall – Achirana (9 tracks), Velvet Belly – The Landing (10 tracks), Vernon Reid – Mistaken Identity (16 tracks), Vivaldi – The Four Seasons (Nigel Kennedy) (12 tracks), Vömmöl Spellmannslag – Vömlingen (12 tracks), Vømmøl Spellemannslag – Vømmølåret (12 tracks), Vømmøl Spellmannslag – Vømmøl'n (11 tracks), Vømmøl Spellmannslag – Vømmølmusikken (12 tracks)

- Warren Zevon – Transverse City (10 tracks), Whale – I'll Do Ya (4 tracks), Whale – Pay For Me (5 tracks), Whale – We Care (13 tracks), Whitney Houston – Whitney (11 tracks), Willie Nelson – Across The Borderline (14 tracks), Wolfgang Amadeus Mozart – The Complete Edition (19 tracks), World Party – Bang! (12 tracks), World Party – Goodbye Jumbo (12 tracks)

- YES – Close To The Edge (7 tracks), Yes – The Yes Album [Expanded & Remastered] (9 tracks), Yuri Honing Trio – A matter of conviction (12 tracks)

- Zakir Hussain – Making Music (7 tracks), Zbigniew Preisner – La double vie de Véronique (18 tracks), Zbigniew Preisner – Trois Couleurs Bleu (24 tracks)

- Åge Aleksandersen & Sambandet – Ramp (11 tracks), Åsne Valland – Den ljose dagen (12 tracks)