TECHNISCHE
UNIVERSITÄT
WIEN

VIENNA
UNIVERSITY OF
TECHNOLOGY

MASTERARBEIT

Mirage

High-Performance Music Similarity Computation and Automatic Playlist Generation

ausgeführt am
Institut für Computational Perception
der Johannes Kepler Universität Linz

unter der Anleitung von Univ.-Prof. Dr. Gerhard Widmer

durch

Dominik Schnitzer
Guglgasse 12/372, A-1110 Wien

Datum	Unterschrift

Kurzfassung

In großen Musiksammlungen und bei der Arbeit mit großen Mengen von Musikstücken, wird berechenbare Musikähnlichkeit ein immer wichtiger werdendes Forschungsgebiet. Denn mit einem berechenbaren Ähnlichkeitsmaß läßt sich Musik zum Beispiel automatisch klassifizieren, sortieren oder zum automatischen Erstellen von Playlists nutzen.

Viele der aktuell verwendeten Algorithmen zur computerbasierten Berechnung von Musikähnlichkeit sind leider aber zu rechen- und speicherintensiv, um tatsächlich breiten Einsatz finden zu können. Diese Arbeit widmet sich jenen Problemen und schlägt konkret Lösungen vor: MP3 Dateien werden direkt zur Ähnlichkeitsberechnung verwendet, die Rechenoperationen der Ähnlichkeitsberechnung drastisch reduziert und geeignete Speichermethoden vorgeschlagen.

Jeder der diskutierten Vorschläge wird im Zuge der Arbeit auch in einer Programmbibliothek zur Musikähnlichkeitsberechnung (*Mirage*) implementiert. *Mirage* kann frei aus dem Internet geladen werden. Die Bibliothek wird zur Demonstration ihrer Einsatzfähigkeit sowohl in einem Musik Player am PC als Plugin zur automatischen Playlist-Generierung eingesetzt, als auch zur Veranschaulichung, wie Musikähnlichkeit tatsächlich auf einem iPod-MP3 Player eingesetzt werden könnte, verwendet.

Zur Evaluation wird die Bibliothek nach Qualitäts- und Performanzkriterien getestet. Es zeigt sich dabei, dass die vorgeschlagenen Opimierungen keinen nennenswerten negativen Einfluß auf die Qualität der Ergebnisse haben, gleichzeitig aber durch den Einsatz der vorgeschlagenen Änderungen sehr beschleunigte Musikähnlichkeitsberechnung möglich ist.

Abstract

When working with large quantities of music or huge music collections, computational music similarity is constantly getting more important. This demand exists, because computable music similarity enables automatic music classification, sorting or unsupervised creation of music playlists for listening.

Unfortunately the methods which are currently utilized for computing music similarity are all too processor- and memory intensive to be used on a large scale. This work focuses on these problems and proposes concrete solutions: MP3 files are directly processed for similarity computation, arithmetic operations of the matrix calculations are reduced drastically and adequate storage methods are proposed.

Each optimization which is proposed is implemented in a program library for music similarity computation (*Mirage*). *Mirage* is published and available freely on the Internet. To demonstrate the capabilities of the library, it is included in a music player application as a plugin to automatically generate playlists and it is used to show how music similarity could actually be used on an iPod-MP3 player.

To evaluate the library, it is tested for quality and speed of the similarity measure. It is shown that the proposed optimizations have no noteworthy negative impact on the quality of the results, but at the same time highly accelerated music similarity is possible by implementation of the proposed changes.

Contents

List of Figures

List of Tables

1. Introduction

Over the last years ways of collecting and listening to music underwent radical changes. Today a music collection is no longer a library of a few Compact Discs kept in a box. Nowadays a music collection is rather a collection of usually thousands of digitally compressed audio files stored on a computer's hard drive or a portable MP3 player.

Starting with the launch of Apple's iTunes Music Store[1] on April 28th 2003, conventional distribution channels for music started finally shifting towards the Internet. Today a lot more music stores are online, like Real Network's Rhapsody[2], eMusic[3] or Napster[4] from Bertelsmann.

With large music collections and easy ways to buy music online, new alternatives of managing the music libraries are necessary. There are many programs available to manually organize music into genres or playlists. But with increasing size manual sorting of music becomes an infeasible job. New automatic ways to manage music are demanded. This is where the research field of music information retrieval (MIR) comes into play. Current music information retrieval methods offer ways to automatically extract useful information from and about music. To do so, the web or the audio signal is used.

The focus of this thesis lies in extracting information from the audio signal, more precisely in music similarity algorithms and their optimization to work with very large music collections. Computational music similarity algorithms concentrate on the extraction of information from the song's audio signal, to characterize it and to find similar sounding music. To be used with large music collections high speed and memory-efficient music similarity methods are needed, which is the central point of this work.

The paper is divided into the following sections: *Section 2* introduces to audio formats and related work to processing compressed audio. *Section 3* gives an overview of available music similarity algorithms. Their functioning is examined and performance critical points in the different phases of the algorithms are identified. This leads leads to *Section 4*, where radical optimizations are proposed. In *Section 5* a music similarity

[1]`http://www.apple.com/iTunes/store/`, last visited March 13, 2007
[2]`http://www.rhapsody.com/`, last visited March 13, 2007
[3]`http://www.emusic.com/`, last visited March 13, 2007
[4]`http://www.napster.com/`, last visited March 13, 2007

program-library is implemented including all performance optimizations proposed earlier. The library is evaluated in *Section 6* for performance and quality of the results.

Main contributions of this thesis include:

- A program and a way to directly utilize MPEG 1 audio for music similarity, doubling the feature extraction speed for MP3 files (*Section 4.1*).
- Technical optimization of a music similarity measure for high performance operations. Feature comparisons could be made about 100 times faster compared to a standard implementation (*Section 4.2*).
- Development of a fast music similarity library in C# usable for large music collections. The library is published free of charge on the Internet (*Section 5.2*).
- Implementation of a plugin for a digital audio player program to show how the automatic playlist generation technique can be used for large music collections (*Section 5.3*).
- The proposal of an interactive playlist generation algorithm to improve automatic playlist generation results (*Section 5.3.2*).
- A prove-of-concept to show how the developed high performance music similarity library could be used to support browsing and listening to music on a portable iPod audio player (*Section 5.4*).

2. Music Audio Formats and Processing

On Computers audio data can be represented in two primary ways: In the time or frequency domain. Non-technically spoken the time domain representation shows how a signal changes over time, whereas the frequency domain representation shows how much of the signal lies within each given frequency band over a range of frequencies. The most common representation of audio-signals and waveforms is in the time domain, digitally represented as Pulse-Code Modulation (*PCM*) on a computer. PCM data is usually stored using the well known WAV, AU or AIFF file formats. However, most signal analysis techniques work only in the frequency domain. This is why usually standard PCM time domain samples which are to be analyzed have to be transformed first, which is typically done by using a Fast Fourier Transformation [Bri74] (*FFT*).

Besides PCM related file formats, there are also the popular standardized audio file formats like MP3, OGG, or AAC. The difference of these formats compared to PCM audio is that the audio signal is stored in a compressed form in its frequency domain. Because these formats compress by removing inaudible frequencies, they are also called "lossy". The basic techniques behind the MP3 and AAC audio format are described in [Bra99]. The royality free OGG Vorbis audio format is described in [Xip04]. Since most of the audio nowadays is stored in MP3, OGG, or AAC format, utilizing the fact that they are already stored in a frequency domain representation, can be used in digital audio analysis. Other related and popular audio formats like Windows Media Audio (WMA) and Real Audio (RA) are closed file formats and can not be used like this since no public documentation is available.

One of the first music information retrieval works which was done in the compressed MPEG-1 audio spectrum was [ZY02], where a speech recognition system was proposed. The system worked withoud decompressing the MPEG audio stream. [TC00] works with the uncompressed MPEG audio spectrum without transforming the audio into a PCM signal. Descriptors which were computed were the Spectral Centroid/Rollof/Flux and the Root-Mean-Square. A wide range of high- and low-level audio descriptors from MPEG-1 compressed audio were tested in [PV01]. Higher level features included features for audio segmentation and music/speech determination. Further processing done in combination with compressed audio files was done by [WV01] and [SXWK04] for beat tracking and music summarization.

This thesis adds to these publications by implementing a computational music similarity measure working directly with MP3 files.

3. Automatic Playlist Generation and Music Similarity

To enjoy listening to music in a certain situation, a mostly coherent music experience is desirable. Consecutive pieces should somehow fit together, sound similar or have something in common.

A successful radio station remains true to its style, to keep its listeners. A Dj selects those tracks which mix together best. The perfect automatic playlist generator tries to do all of this automatically and on the user's own music collection. It is a personal Dj or radio station, making listening to music easier and more enjoyable. To support users with such a system the two closely related research fields of automatic playlist generation and computational music similarity play the key role.

3.1. Playlist Generation Techniques

An automatic playlist generator needs to be initiated with some input from the user. The user needs to specify what he wants to listen to. This initial music style selection can be done in a very straightforward way, where the user just supplies music genres or terms which describe the music he wants to listen to. This approach has many issues, since most music just does not belong to a single specific genre. Besides that, the definition of a music genre is a very subjective one.

A different approach to an automatic playlist generator is a visual one. The user selects songs on a map visualization of his music. On the map similar songs are clustered together by acoustic similarity. Playlists can be created by drawing paths on this music maps. An example of such a technique is presented in a prototype called PlaySom [NDR05]. This approach presupposes a fully analyzed music collection, which in turn requires a good and efficient similarity measures.

A third technique which can be used for automatic playlist generation is query-by-example. With this technique, the playlist is usually initiated by sample tracks which should indicate the musical style the other songs should have in common. Based on these seed songs the playlist generator tries to find the most similar tracks in the users music collection to play them. [Log02] evaluates this technique in conjunction with a computational music similarity measure. [PPW05a] improve the query-by-example technique by incorporating song-skipping behavior into the playlist generation algorithm. Skipping a song is seen as negative feedback, listening to a whole song is regarded as good feedback. By analyzing the feedback data, further playlists are improved. [AP02b] combines metadata and computationally extracted music information to create playlists in huge music databases using an adaptive search technique.

Let's have a closer look at at the basics behind the playlist generation techniques, to show the diversity of approaches available and discuss their assets and drawbacks.

Standard Playlist Generation Standard automatic playlist generation usually relies on meta-information like the genre of the track, number a track has been played, an optional users rating and a good pinch of randomness to select the next song. This information is usually encoded in the audio file or collected from the playback behavior of the user. It is used by the audio player to sort or search for music. If the music archive is in a good shape, the tracks are all rated and classified by the user himself, then this technique works quite well. Typically this strategy fails, since standard music collections are poorly classified into genres or are simply too large for manual classification.

Web-Based Metadata A more intelligent way to overcome some of the shortcomings of the standard approach is using the Internet to automatically try grabbing additional metadata for a song. This is for example done in [BH03], [KPW04] or [LKM04]. The information gained through this techniques can then be used to generate playlists. This works quite well on the artist level, given the artist is well known or has a broad appearance on the Internet. It does not work well if similarity estimates are needed for comparing individual songs, since information about a specific song is usually very sparse on the Internet.

Playlists through Social Networks Social networks are a very recent way to acquire information about users and music taste. A social network requires the user to sign up and fill their network profile. After that the services of the network can be used. If the network is used regularly, it is able to classify the user's preferences, by comparing the profile to the profile of all other users. The underlying technique is a collaborative filter. A collaborative filter has the basic assumption that those who agreed in the past will also agree in the future [KSS97].

Examples of social networks are MySpace[5] or Orkut[6]. Social networks have recently also been used for automatic playlist generation. The last.fm[7] network collects information about your songs and music listening habits and compares these to others. This is done in two ways and is working quite well: A plugin for various music players reports the playing statistics of songs back to the network. And second, it is possible to tag tracks with keywords to feed the network.

Audio Analysis for Playlist Generation Another way to automatically generate a playlist of similar tracks is by computationally extracting patterns from the audio signal to describe the song. These patterns are called descriptors or features and are typically numerical values. If two music pieces have similar descriptor values they should sound the same in the aspect, the descriptor tries to extract from the audio signal. Thus it is possible to compare the tracks to others or automatically order the tracks. Extracting descriptor information is the most difficult part, since the descriptors should match an aspect of what humans hear and what they perceive in music. Over all other playlist generation techniques automatic audio analysis has the advantage of handling all pieces of music equally. The results do not depend on what most users think, like in social networks, or how somebody selected a genre for the song.

3.2. Music Similarity

In order to be able to automatically generate playlists, similarity of music has to be computable somehow. Music similarity is the very basic thing to enable generation of automatic playlists, based on audio analysis.

A music similarity measure makes computational comparisons between two music pieces possible, and thus enables an automatic similarity ranking of music pieces (a *playlist*). A music similarity measure can also be used to automatically cluster similar pieces together. Today the best

[5]`http://www.myspace.com/`, last visited March 13, 2007
[6]`http://www.orkut.com/`, last visited March 13, 2007
[7]`http://www.last.fm/`, last visited March 13, 2007

techniques to compute music similarity are all based somehow on statistical frequency spectrum analysis. These methods usually compute the so called timbre similarity.

Besides similarity measures trying to describe the overall frequency spectrum of a music piece, there exist also a huge variety of descriptors trying to extract certain perceptual parameters from the track. Some of these include for example: *Audio Power* [AHH$^+$01], *Spectrum Spread/-Flatness/Centroid* [MPE], *Harmonicity* [MPE], *Periodicity* [PDW04] or *Tempo* [Ell06] [GKD$^+$06].

The currently best working techniques to compute music similarity are the frequency spectrum analysis methods. The results of the yearly MIREX contest at the ISMIR[8] conference clearly show that the timbre similarity methods dominate and currently perform best. Summarizing the frequency spectrum of a music piece to compute something like timbre similarity became popular through the works of Acoutourier and Pachet [AP00], who use Gaussian Mixture Models to approximate spectra of music. The next sections give an introduction and overview to the most prevalent music similarity algorithms available today.

3.2.1. Mel Frequency Cepstral Coefficients

The Mel Frequency Cepstral Coefficients (*MFCCs*) play an important role in the field of music analysis. Although they were first used for speech analysis [YWB93], they are currently adopted as the dominant feature in music information retrieval to compactly describe the amplitude spectrum of music. MFCCs have been introduced to music information retrieval researchers by Beth Logan in [Log00].

MFCCs are short-term spectral-based features, where each step in the creation of MFCC features is motivated by perceptual or computational considerations. The perceptual intention of the MFCCs can be seen in the logarithmic spacing of the frequency bands to approximate the cochlea in the human ear more closely. Computationally MFCCs also try to reduce the data by eliminating unnecessary redundancies. This is done through a Discrete Cosine Transform (*DCT*) [ML93].

The calculation of the MFCCs includes the following steps:

[8]`http://www.ismir.net/`, last visited March 13, 2007

1. The audio signal is first divided into frames, usually spanning about $20 - 25ms$ of audio.
2. A windowing function (usually a Hamming or Hann window) is applied to the audio frame to avoid edge effects in further processing.
3. The discrete Fourier transform (DFT) [OS89] transforms each audio frame from the time domain into the frequency domain. So the short-time power spectrum $P(f)$ is obtained.
4. The spectrum $P(f)$ is transformed along its frequency axis f (Hz) into its Mel [P+93] $M(f)$ representation, to approximately reflect the human ear's perception.

$$M(f) = 2595log_{10}\left(1 + \frac{f}{700}\right). \tag{1}$$

5. Finally the highly correlated Mel values are reduced by applying the Discrete Cosine Transform [ANR74] (DCT) on the Mel values. This results in the final N cepstral features for each frame.

3.2.2. Logan & Salomon (LS)

Beth Logan and Ariel Salomon were one of the first to publish a music similarity function based on audio signal analysis [LS01].

Feature Extraction The feature extraction process of this method works by describing the spectrum of a music piece by trying to find clusters of similar audio data frames. To do so first the audio data is downsampled to a $16khz, mono$ signal and N $25.6ms$ wide frames overlapping the previous frame by $10ms$ are created. Each of these frames is then transformed into the frequency domain by an FFT. After that the MFCC coefficients $(13 - 30)$ are computed, so that the music piece is now described by N MFCC frames each one describing the spectrum of $25.6ms$ of audio.

The frames are then clustered to create the model. If k-means clustering is used, the number of clusters has to be fixed, it can be variable if if X-means [PM00] clustering is used. The signatures of each cluster represent a song's model. This is the mean, covariance and the weight of the cluster in respect to the other clusters.

Feature Comparison To compare songs or their respective models, the spectral signatures of the songs are used to compute the Earth Movers Distance (*EMD*) [RTG00]. The EMD computes the distance between two distributions. In this case the song signatures are the distributions and the distance is seen as a similarity measure.

The EMD is defined as the minimum amount of work needed to transform one signature into the other. In the case of Logan & Salomon's algorithm, "work" is defined as the symmetrized Kullback-Leibler (KL) divergence [Pen01] between two distributions. Let $d_{p_i q_j}$ be the KL divergence between two clusters p_i, q_j and $f_{p_i q_j}$ the cost of moving probability mass from one cluster to the other [RTG00], then the EMD is defined as,

$$EMD(P,Q) = \frac{\sum_{i=1}^{m}\sum_{j=1}^{n} d_{p_i q_j} f_{p_i q_j}}{\sum_{i=1}^{m}\sum_{j=1}^{n} f_{p_i q_j}}. \tag{2}$$

The EMD can also be computed if the numbers of clusters in the two signatures is different which happens when X-means clustering was used.

3.2.3. Aucouturier & Pachet (AP)

In 2002 Jean-Julien Aucouturier and Francois Pachet presented a technique for music similarity in [AP02a] which shaped the field of timbre similarity together with [LS01].

Feature Extraction This feature extraction process tries to describe the spectrum of music pieces using multiple Gaussian mixture models (*GMMs*) [Bis95].

A short-time Fourier transformation is done on $50ms$ of audio at a time. Then the MFCCs are calculated for the selected audio frames. Only the first eight MFCC coefficients are used in this implementation. After that an Expectation Maximization (*EM*) [Bis95] algorithm tries to find mixtures of three Gaussians to fit the MFCC vectors best. The EM algorithm is initialized by k-means clustering. In the end a song is modelled by three eight dimensional Gaussian distributions fitting the distribution of the MFCC vectors over a song.

Feature Comparison Similarity computation is done by computing the likelihood of samples from GMM_{songa} given GMM_{songb} and vice versa. Since the original MFCC vectors are not available at this point any more, Monte Carlo sampling [Bis95] is used to generate samples for the likelihood computation. The resulting similarity should be symmetrized (see *Equation 3*).

$$sim_{a,b} = \frac{p(a|b) + p(b|a)}{2} \tag{3}$$

3.2.4. Mandel & Ellis (ME)

Michael Mandel and Dan Ellis published a new method to compute music similarity in 2005 [ME05]. Their proposed music similarity algorithm was combined with Support Vector Machines (see [CST00] for an introduction to *SVMs*) for automatic genre classification of songs. The algorithm took part in the MIREX genre classification contest at the ISMIR conference 2005[9], where it performed very fast and was the third best entry.

Feature Extraction The feature extraction process of the algorithm is as simple as this: The first 20 MFCCs are calculated for a given song. The mean and covariance are computed for the resulting MFCC vectors. A model is represented by a 20×20 covariance matrix and a 20-dimensional mean vector. Thus the song is represented as a single Gaussian distribution. These values can be interpreted as the overall frequency spectrum distribution of the song. Like in all other timbre music similarity models, all temporal aspects of music are ignored here too.

Feature Comparison Comparison of two songs modeled by this method is done by computing the Kullback-Leibler (*KL*) divergence [Pen01]. Since the KL divergence is not symmetric, its symmetrized form is used for similarity computation.

[9]`http://ismir2005.ismir.net/`, last visited March 13, 2007

3.2.5. Pampalk, Rauber & Merkl (PRM)

Elias Pampalk and Andreas Rauber published a music similarity measure named *Rhythm Patterns* [PRM02] in 2002. Since 2004 the rhythmic descriptors are also referenced as "Fluctuation Patterns", when Pampalk published a Matlab toolbox for music similarity [Pam04].

Feature Extraction The feature extraction process for the fluctuation patterns is divided into two stages. In the first stage several psychoacoustic transformations are applied to the audio signal, including a transformation to the Bark scale [SIA99], spectral masking effects, computation of the sound pressure level (dB-SPL) and finally computation of its Sone values ("Sonogram"). In the second step the rhythm patterns, a time-invariant representation for each music piece is computed. Rhythm patterns try to describe how strong and fast beats are played within each analyzed frequency bands.

Feature Comparison Comparison of rhythm patterns is very simple. Similarity is computed as the Euclidean distance between the two rhythm patterns which are simply seen as vectors.

3.2.6. Flexer, Pampalk & Widmer (FPW)

In 2005 Arthur Flexer, Elias Pampalk and Gerhard Widmer proposed an algorithm to work with Hidden Markov Models [Rab89] (*HMMs*), to include the temporal context of music in a similarity measure [FPW05a]. HMMs allow analysis of time series by statistically modeling the locally stationary data and their transition probabilities.

However it is shown that using HMMs does not improve the performance of the music similarity measure, although HMMs seem to better capture details of the sound of music recordings.

Feature Extraction To create a model for a song the first eight MFCC coefficients are computed for the song. The frame size used is 23.2ms with a 50% overlap. To capture a bigger temporal context differently sized texture windows [TC02] can be used too. The HMMs are then trained with the MFCC values using a Gaussian Observation Hidden

Markov Model. The Expectation Maximization [Bis95] algorithm trains the HMM, which finally models the song.

Feature Comparison The similarity of two HMM models is computed by first using the forward algorithm to identify the most likely state sequences corresponding to a given time series and second by the use of the log-likelihood function to compute the similarity.

3.3. The Most Suitable Algorithm

So which algorithm is the most suitable to truly achieve a highly performant music similarity measure in regard to speed and quality?

Again the results from the last MIREX Audio Music Similarity and Retrieval contest (see *Table 1*) from the ISMIR 2006 conference show that the currently best working techniques (EP and TP in *Table 1*) are all based on ME [ME05]. EP combines the ME technique with fluctuation patterns and TP includes a technique called ”proximity verification“ to improve results.

EP [Pam06a]	TP [Poh06]	VS	LR [LR06]	KWT	KWL
0.430	0.423	0.404	0.393	0.372	0.339

Table 1: The final Audio Music Similarity and Retrieval Results from the MIREX contest 2006[1]. The scores are averaged ratings of playlists, which were generated by the respective algorithm.

[1]http://www.music-ir.org/mirex2006/index.php/Audio_Music_Similarity_and_Retrieval_Results

In addition to the MIREX contest results, the evaluations in [Pam06b] also show very good results for ME. In [Pam06b] the ME technique is referenced as a one-Gaussian (1G) method and scored best when combining it with fluctuation pattern information. Besides the rather good quality of the results, ME also has the advantage of being very simple and fast to compute - making it an ideal candidate for truly high performance music similarity.

Based on these results and considerations the ME technique for computing music similarity is further used as foundation for developing the high performance music similarity library in this thesis.

3.4. Identifying Performance Critical Points

Performance is a very critical point when it comes to adaptation of techniques for real applications. In our case a playlist generator which takes days to analyze a 5000 songs music collection and minutes to generate a single playlist, is rather useless, even if it all works well. Users just do not like long delays, they are annoying. A playlist generator working in the background and using up all your computers resources is not desireable either.

As it can be seen in *Table 2*, music similarity was very slow in 2004, but current techniques are already getting faster and more usable. To take performance to the next level, first the main performance bottlenecks in the current music similarity computation process chain have to be identified.

	Feature Extraction (per song)	Distance Computation (per song)
2004	60 seconds	500 milliseconds
2005	3 seconds	3 milliseconds

Table 2: A table from [Pam05] shows the progress made in music similarity computation in terms of performance. *Note:* in this case feature extraction assumes that the music piece is already in PCM/mono/22khz format, which is not usual for standard music collections.

3.4.1. Feature Extraction

In the feature extraction phase multiple components play an important role. Usually feature extraction has to be done once per track in the collection. Since nowadays almost all music is available and stored in compressed form, analysis usually includes the decoding of the files. Popular encoding techniques for digital music include AAC/MP4, WMA and MP3, where the most popular format is certainly MPEG 1 Audio Layer

3 (*MP3*). For an MP3 file the feature extraction process with the ME method would look like this (*Figure 1* illustrates this):

1. Decoding of the MP3.
2. Conversion to a mono signal.
3. Further downsampling to a 11025hz signal.
4. Applying a STFT with a typical window-size of 1024-2048 samples.
5. Transform the signal to the Mel scale.
6. Calculate the MFCCs for each frame.
7. Compute the mean and covariance of the MFCC vectors to retrieve the final song model.

So if the music is available in compressed form, like it can be seen in *Figure 1* for an MPEG 1 Audio Layer 3 file, the usual way of analyzing the file would include transforming the audio data from the frequency domain to the time domain and back which includes resynthesizing, an FFT step and a downsampling step. Avoiding this would result in a big performance win.

3.4.2. Feature Comparison

The second stage which is very time consuming and plays a key role in the whole process, since it is used every time a playlist is computed, is the feature comparison process.

For t_n track models a query for similar tracks includes two major functions which are repeated n times and account for most of the time spent in a query:

1. The comparison computation. To find the most similar tracks in a collection with n songs, given a single seed song t_s, all other tracks have to be compared with t_s. An optimized comparison algorithm would therefore noticeably speed up a query.
2. Loading the track models from disk. A query requires an iteration over all n track-models. Unless it is feasible to hold all n track models in memory, the iteration over all tracks is usually a disk i/o heavy operation. Reducing or optimizing disk i/o would result in a big speedup.

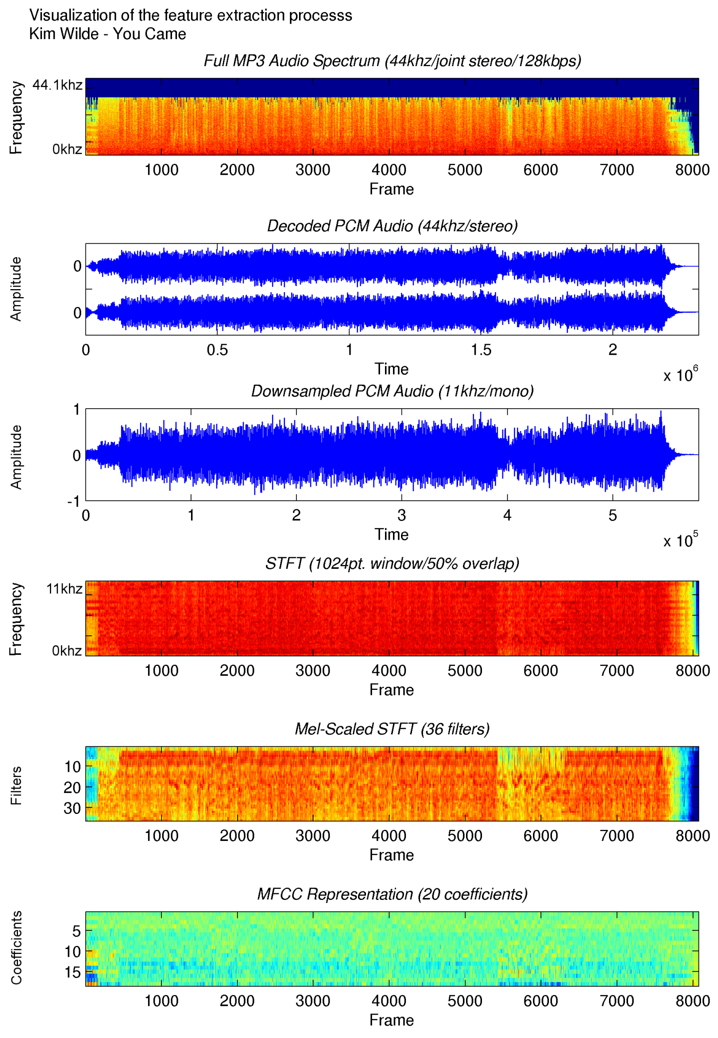


Figure 1: Normal similarity feature extraction.

The following paragraphs describe the aspects in detail which would need to be optimized to achieve a high-performance query system.

Feature Comparison Since the tracks t_n are modelled as multivariate Normal densities (single Gaussians), similarity between two tracks t_1, t_2 can be computed by calculating the Kullback-Leibler (KL) divergence as described in [ME05]. The KL divergence for normal densities $t_1(x) = N(x; \mu_{t_1}, \Sigma_{t_1})$ and $t_2(x) = N(x; \mu_{t_2}, \Sigma_{t_2})$ is

$$KL_N(t_1\|t_2) = \frac{1}{2}\left(\log\left(\frac{\det\left(\Sigma_{t_1}\right)}{\det\left(\Sigma_{t_2}\right)}\right) + Tr\left(\Sigma_{t_1}^{-1}\Sigma_{t_2}\right) + \left(\mu_{t_1}-\mu_{t_2}\right)'\Sigma_{t_1}^{-1}\left(\mu_{t_2}-\mu_{t_1}\right) - d\right), \tag{4}$$

where $Tr(A)$ denotes the trace of the matrix A, $Tr(A) = \Sigma_{i=1..n}a_{i,i}$. The KL divergence is not symmetric. To symmetrize it the two divergences are simply added up,

$$D_{KL}(t_1, t_2) = KL_N(t_1\|t_2) + KL_N(t_2\|t_1). \tag{5}$$

By symmetrizing the similarity the combined KL divergence can be further simplified. The sum $\log(\frac{\det(\Sigma_{t_1})}{\det(\Sigma_{t_2})}) + \log(\frac{\det(\Sigma_{t_2})}{\det(\Sigma_{t_1})})$ is 0 and can be left out. For similarity comparisons where all track models t_n have the same dimension d, d can be left out too. Finally the constant factor $\frac{1}{2}$ can be skipped too, simplifying a single feature comparison to a modified symmetrized KL divergence,

$$\begin{aligned} D_{KL}(t_1, t_2) = \\ Tr\left(\Sigma_{t_1}^{-1}\Sigma_{t_2}\right) + Tr\left(\Sigma_{t_2}^{-1}\Sigma_{t_1}\right) + \\ Tr\left(\left(\Sigma_{t_1}^{-1} + \Sigma_{t_2}^{-1}\right) \times \left(\mu_{t_1}-\mu_{t_2}\right) \times \left(\mu_{t_2}-\mu_{t_1}\right)'\right). \end{aligned} \tag{6}$$

By removing the normalizing factors d and $\frac{1}{2}$ the result is no true distance measure any more. But since the KL divergence is only used for similarity comparisons among identically computed values, leaving out those constant factors has no impact on the results.

Loading Track Models After the feature extraction process every music track in the collection has an associated track-model t_i which is computed from the audio track i in the feature extraction phase. A single query for similar tracks requires all n models to be compared with the seed song and therefore being loaded from disk once. If there are only few tracks in the collection, the models can be loaded from disk into memory once, if memory is limited or there is a huge collection, the tracks will have to be loaded from disk for each similarity query. Since disk i/o is a very time consuming process, it should be little.

To achieve low disk i/o the track models themselves should contain absolutely no unnecessary data, to keep the models small in size. Efficient and clever storage enables fast access times. A threaded iteration process for unblocked comparisons should be implemented to achieve this.

4. Towards High-Performance Playlist Generation

After having identified key points which should be addressed when creating a high-performance playlist generator, possible solutions to these problems are proposed.

4.1. Rethinking the Feature Extraction Process

Current feature extraction is a computationally very intensive task and takes about 4 seconds (*Table 2*) per piece on a reasonable new computer, if the music piece also needs to be decoded and downsampled to be analyzed as described in *Section 3.4.1*.

Due to the popularity of the MP3 file format, the next sections take a deeper look at MP3 files and evaluate possibilities to speed things up for MP3 audio. Other compressed audio formats are discussed in *Section 4.1.3*.

4.1.1. MP3

MPEG-1 Audio Layer 3, usually referred as MP3 [MPE92], is the most common and most widely known audio compression standard today. It is a lossy audio compression format, meaning that the original audio can not be restored 1:1. The encoding and compression of audio is achieved by removing perceptually inaudible frequencies, which makes MP3 files very small and ideal for streaming over the internet or sharing.

MP3 has its origins in the EU supported EUREKA project and was first published as part of the MPEG-1 standard in 1994. It quickly got momentum after the Fraunhofer Institute released the first MP3 encoder*l3enc*. Coupled with the rise of the Internet and peer to peer filesharing clients, MP3 became the standard for listening, collecting and sharing music on computers. Although with recent successes of online music stores, different compression formats with copy control mechanisms emerged, the majority of all digital music is still MP3. Even car stereos or DVD players today support MP3 playback.

Bit- and Samplingrates MP3 offers many bitrates to encode audio. Valid bitrates are listed in *Table 3*. $128kbit/s$ and $160kbit/s$ at $44.1khz$ are used most, since few audible differences between CD quality can be heard. Audio on a CD has $1411.2kbit/s$ ($16bits/sample \times 44100hz \times 2channels$). MP3 was specified to achieve best compression rates and quality with $128kbit/s$ audio sampled at $44khz$. In addition to static bitrates, MP3 also offers variable bitrates (VBR). Variable bitrate MP3 files are switching the bitrate of their frames dynamically, as required by the encoded signal complexity.

Valid bitrates (kbit/s)	$32, 40, 48, 56, 64, 80, 96, 112, 128, 160, 192, 224, 256, 320$
Sampling frequencies (khz)	$32, 44.1, 48$

Table 3: Available bitrates and sampling frequencies for MP3 encoding [MPE92].

Filestructure Audio data in MP3 files is stored in a very convenient way. Each compressed frame is preceded by a header identifying it as MPEG 1 Audio Layer 3 audio and specifying bitrate and samplingrate. The detailed MP3 header format can be seen in *Table 4*.

Since the header precedes every audio frame, MP3s can be streamed easily over the Internet.

Encoder MPEG-1 does not specify how to encode MP3 files, it only specifies precisely how the data is to be decoded, thus giving much space for implementation of optimal encoders.

An MP3 audio data frame consists of 576 Huffman encoded values, describing about $32ms$ of audio. Each of the 576 values in the audio frame describes a frequency value and is spaced from $0 - \frac{fs}{2}hz$. Besides these standard frames, MPEG-1 Audio Layer 3 also defines frames with 192 frequency values (*short block*), which have a higher time resolution and can be used by the encoder to encode transient signals.

Bits	Description
$0-10$	Frame sync
$11-12$	MPEG Audio Version (1)
$13-14$	Layer version (Layer 3)
15	CRC protection
$16-19$	Bitrate (see *Table 3*)
$20-21$	Samplingrate (see *Table 3*)
22	Padding
23	Private
$24-25$	Channels (mono, joint stereo, Dual channel stereo)
$26-27$	Joint Stereo mode extension
28	Copyrighted
29	Original
$30-31$	Emphasis

Table 4: The MP3 frame header [MPE92].

Decoder MP3 decoding is precisely defined in the MPEG standard [MPE92], where a reference MP3 decoder is described. If an MP3 decoder achieves the same output as the reference decoder, it is called "bitstream compliant". The MP3 decoding process is split into the following steps (*Figure 2*).

The first three steps, *Huffman Decoding*, *Requantize* and *Reordering* transform the MP3 bitstream into its 576 frequency values. These values are divided into 32 equally spaced subbands each containing 18 frequency values ($32 \times 18 = 578$). Spacing of the subbands and frequency values is done linearly in the range of $0 - fs/2$ according to the Nyquist-Shannon sampling theorem (i.e. the i^{th} value of a frame describes the frequency $f = \frac{i}{576} fs$).

To reconstruct the original PCM signal three consecutive MP3 frames are needed, which are processed by the *Antialiasing*, the *Inverse MDCT* and *Filterbank and Windowing* components. Three frames at a time have to be used, since the original frames overlap 50% with the previous and next one.

Huffman Decoding The raw MP3 bitstream is Huffman coded. Huffman coding [Huf52] is a method to code a sequence of data by using the minimum number of bits necessary. The Huffman encoded

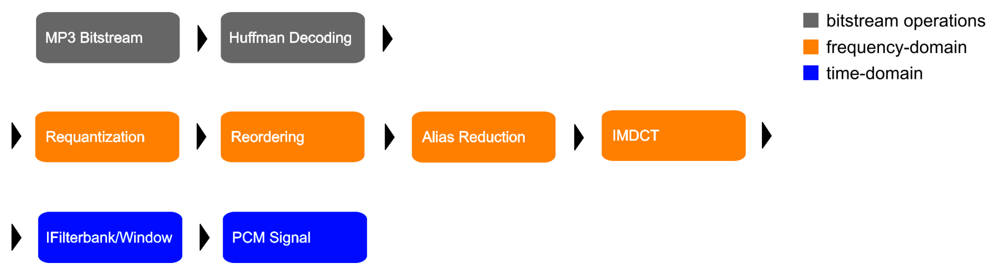


Figure 2: Steps to decode an MP3.

MP3 bitstream is decoded using the appropriate decompression tables. The decompression tables are specified in the ISO/IEC standard document [MPE92]. The decompressed bitstream contains the original 576 scaled frequency values and their scalefactors to reconstruct the original audio. After Huffman decoding the values are still scaled and make up three different components: Values between -15 and 15 with high precision for the low frequency subbands, the values -1, 0 and 1 for high frequency subbands and zero values for the highest inaudible subbands which should be removed. The encoder uses psychoacoustic analysis to identify and remove inaudible frequencies.

Requantizing and Reordering In the requantizion step, the scaled frequency values obtained from the Huffman decoding process, are requantized to get the original frequency values. To do so the scalefactors extracted in the previous step are used to compute the original values by rescaling. If a short block is being processed the frequency values need to be reordered, since short blocks are stored differently to improve Huffman compression efficiency.

Antialiasing Before reconstructing the original audio, alias reduction needs to be done. To do so the MPEG standard requires eight butterfly calculations to be applied between the frequency lines.

Inverse Modified Discrete Cosine Transform (IMDCT) The IMDCT is the inverse operation of the MDCT [BR01]. The MDCT transforms $2n$ time samples, $x_0, \ldots, x_{2n-1}$, into n frequency samples, $f_0, \ldots, f_{n-1}$. The IMDCT does the opposite and retransforms the n frequency domain samples into $2n$ time domain samples,

$$x_i = \sum_{k=0}^{\frac{n}{2}-1} f_k \cos\left(\frac{\pi}{2n}\left(2i+1+\frac{n}{2}\right)(2k+1)\right). \tag{7}$$

At the first glance it may seem strange that the IMDCT works, because of the different number of input and output samples. But indeed the IMDCT is perfectly reversible, because subsequent blocks overlap, samples are simply added and the error cancels out over time. This effect is called time domain aliasing cancellation (*TDAC*) and is described in [BR01].

Filterbank and Windowing In the final step of the MP3 decoding process the synthesis polyphase filterbank transforms the 32×18 time-domain samples into 576 PCM samples. The synthesis polyphase filterbank is the reversion of the polyphase filters which were applied during the encoding of the MP3 and work on the 32 subbands. The MPEG standard strictly [MPE92] defines the filters, which are very similar to the polyphase quadrature filters from [Rot83]. Windowing is used to filter out undesired aliasing.

4.1.2. Exploiting MP3

So how can the fact that audio is stored as an MP3 file help speeding up the feature extraction process? As one can see in the previous section, an MP3 decoder has to transform audio from the frequency domain into the time domain to retrieve PCM samples. On the other hand all music information retrieval algorithms usually work in the frequency domain. So the signal is retransformed from the time (the PCM signal) into the frequency domain.

It is obvious that there would be a big speed increase if one could get the frequency domain representation of an audio file directly by working with the frequency-domain samples of the MP3 file. To do so the MP3 needs to be decoded just before the uncompressed frequency-domain data is transformed into a PCM signal. This must be be done right after antialiasing, just before resynthization of the audio into PCM. If this is not done after antialasing (i.e. before reordering the bitstream) disordered or unscaled values would make the results unusable.

By omitting the retransformation of the signal into the time-domain, a computationally intensive step is skipped and further steps like computing the STFT not necessary any more. Downsampling the signal is simply done by removing the higher frequency lines from the decoded bitstream. So if it is necessary to downsample a $44.1khz$ MP3 to $11.025khz$, only the first 144 frequency lines (576/4) of the 576 have to be decoded resulting in another speed boost. Overall this means that the necessary feature extraction steps would be reduced to (illustration in *Figure 3*):

1. Intercept the decoding of the MP3 after antialasing and before resynthesizing to PCM.
2. Transform the signal onto the Mel scale.
3. Calculate the MFCCs for each frame.

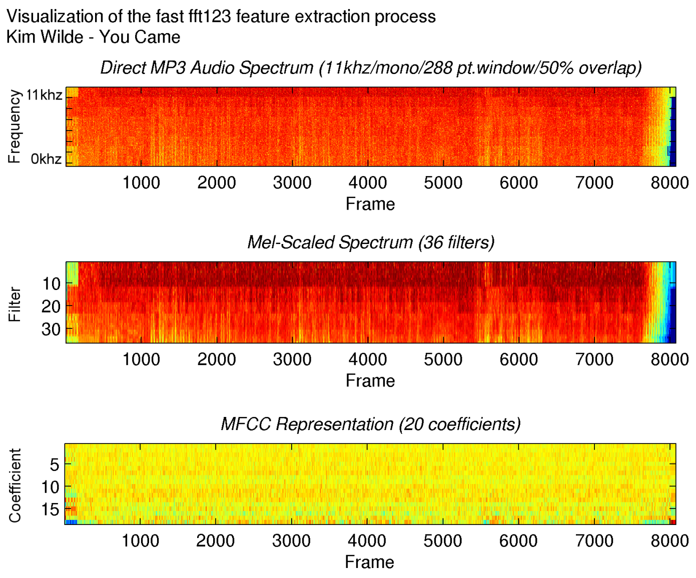


Figure 3: Modified feature extraction.

Section 6.2 shows the performance gained by using this approach. However some peculiarities have to be thought of, when using the MP3 in its frequency domain representation. The MPEG-1 Audio Layer 3 works with window sizes of 1152 usually at $44.1khz$ ($\approx 26ms/window$). It has a high time resolution. Current music similarity methods like [AP04], [ME05] or [Pam06b] work with larger window sizes at lower sampling rates. The usual window sizes vary from $1024 - 2048$ at a sampling rates of $11025 - 22050hz$. This is equivalent to approximately $93ms$ of audio and a three times higher frequency resolution.

But in the further feature analysis, the frequency resolution is reduced further anyway by the application of the MFCCs, so that the smaller window size does not carry weight in terms of quality.

Another parameter of the STFT applied in the feature extraction phase is the window function and window overlapping. Common window functions applied in music similarity algorithms is the Hann or Hamming window [OS89]. MPEG-1 Audio Layer 3 [MPE92] defines its own window function (see *Equation 8* and *Figure 4* for an illustration),

$$w_n = \sin\left(\frac{\pi}{2}\sin^2\left[\frac{\pi}{2N}\left(n+\frac{1}{2}\right)\right]\right). \tag{8}$$

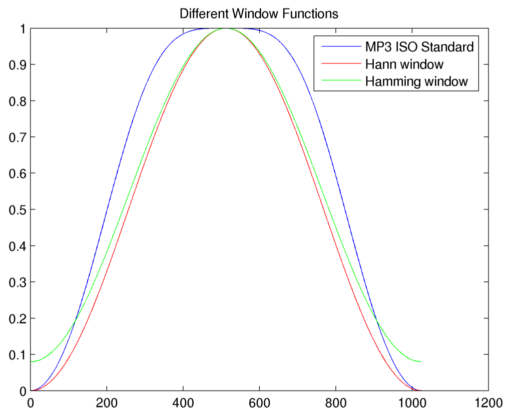


Figure 4: Window functions used in music similarity algorithms compared to the window function used in the MP3 ISO/IEC Standard [MPE92].

The consequences of working with a higher time resolution (smaller window sizes) and the different window functions are evaluated in *Section 6.3*, where the results indicate that the differences have almost no negative impact on the results of the music similarity algorithm. Similar findings are published in [AP04], where the results of music similarity algorithms with different window sizes were evaluated.

4.1.3. Other Audio Formats

All assumptions which were made for MP3 files are valid for most of the compressed audio files, if they are stored compressed in the frequency domain. This enables fast feature for AAC/MP4 or WMA if the adequate decoder is modified in a similar way as was done in the case of MP3.

One thing would have to tested when mixing direct models of different compressed file formats, for example models of AAC and MP3 files. Their respective frequency domain representation uses different window sizes and sampling rates. It has to be evaluated if their final models can be compared without a notable reduction of quality in the results.

4.2. Faster Feature Comparison

The feature comparison is, in contrast to the feature extraction, required to be run every time a query for similar songs is started[10]. As discussed earlier two key parts need to be taken care of to ensure fast generation of playlists.

4.2.1. Optimizing the Similarity Computation

To decrease the time needed for the similarity computations (see *Equation 6*), the necessary raw arithmetic operations are examined, to see if some of the matrix operations can be optimized.

A single similarity computation (the symmetrized KL divergence, see *Section 3.4.2*, *Equation 6*) can be split into four parts: the trace of a matrix and vector product, the difference of two matrices and vectors, and a product of a matrix and a vector. A detailed list of matrix operations and their required number of arithmetic operations can be seen in *Table 5*.

All together a single comparison takes $4n^3 + 2n^2 + 5n - 1$ operations to finish, so for a standard song model with a $d = 20$ it takes exactly 32899 arithmetic operations.

[10] An alternative could be to store the entire $n \times \frac{n}{2}$ matrix of pairwise similarities once and for all. That is not practicable for large music collections for obvious reasons.

Matrix Operation	Number of Operations
$Tr(N \times M)$	$2n^3 - n^2 + n - 1$
$N - M$	n^2
$v - w$	n
$Tr(v \times w')$	$n^2 + n - 1$
$N \times v$	$2n^2 - n$
Operations for D_{KL}, *Equation (6)*	$4n^3 + 2n^2 + 5n - 1$

Table 5: Number of arithmetic operations for computing the KL divergence in a standard way. N, M are $n \times n$ matrices, v, w are $n \times 1$ vectors.

The two matrices used in the modified KL divergence similarity computation (*Equation 6*) are the covariance matrix and its inverse. To optimize the computation one can take advantage of the symmetry of the matrices and calculation of the matrix trace ($Tr(A)$), which are the main keys for cutting back the number of arithmetic operations to compute the KL divergence drastically:

Matrix Symmetry The covariance matrix and its inverse are symmetric matrices. To calculate matrix calculations with these matrices the part below the matrix diagonal does not have to be computed. This reduces the number of arithmetic operations required for calculations with symmetric matrices by about 50 percent. See *Table 6* for detailed estimates in respect to the KL divergence.

Matrix Trace The trace of a matrix is the sum of its diagonal elements, $\Sigma a_{i,i}$. The KL divergence includes two matrix trace operations of a matrix product. Since the matrix trace is the sum of the matrix diagonal the full product of the two matrices is useless in this case. Only the diagonal elements of the product have to be calculated to compute the trace. This reduces the number of operations by an order of magnitude from $O(n^3)$ to $O(n^2)$.

These considerations lead to a new table of required operations for an optimized KL divergence (*Table 6*).

After these optimizations to compute music similarity, a single comparison of standard song models with $d = 20$ takes 1470 operations, which is about 95% less than it takes with the standard version. *Figure 5*

Matrix Operation	Number of Operations
$Tr(N \times M)$	$n^2 - 1$
$N - M$	$0.5(n^2 + n)$
$v - w$	n
$Tr(v \times w')$	n
$N \times v$	n^2
Operations for D_{KL}, *Equation (6)*	$3.5(n^2 + n)$

Table 6: Reduced arithmetic operations for a similarity comparison, taking the special properties of the matrices into account. N, M are symmetric $n \times n$ matrices, v, w are $n \times 1$ vectors.

pictures the reduction of arithmetic operations for computing the KL divergence.

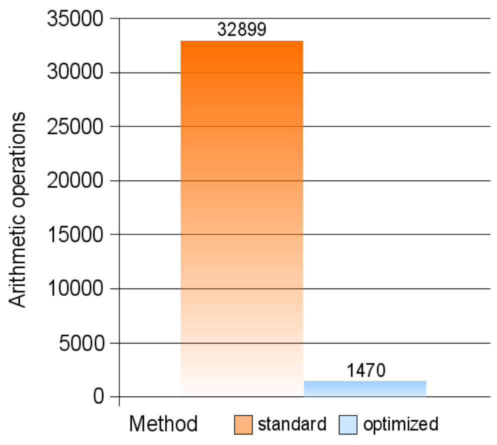


Figure 5: KL Divergence, for ME optimized KL Divergence.

4.2.2. Persistence

Fast access to and loading of models is of course also a key part in fast playlist generation. Since smaller models can be loaded faster, the symmetry of the covariance and its inverse is again taken into account. Utilizing the symmetry only about half of the original values need to be stored, reducing the number of values per model v from $2 * n^2 + n$ $(n = 20, v = 820)$ down to $n^2 + 2n$ $(n = 20, v = 440)$.

Especially for huge collections it is it is impossible to hold all models concurrently in memory, so there has to be a fast way to load the models from disk. A small but optimized plain database file fits this requirements best to ensure quick access to the stored models. Oracle Berkeley DB[11] or SQLite[12] are good candidates for this. The implementation described in the next section shows a practical solution to this using SQLite.

[11] `http://www.oracle.com/database/berkeley-db/db/index.html`, last visited March 13, 2007

[12] `http://www.sqlite.org/`, last visited March 13, 2007

5. Mirage

Mirage is the name of the music similarity library which was programmed during the work for this thesis. The name *Mirage* is a short form for *M*usic *I*nformation *R*etrieval *Age*nt and because of its focus on performance also an allusion to the fast French military jet called Mirage. Its goal was to make fast, state-of-the-art music similarity easily accessible for developers and to show its usage in two selected applications. The implementations which were done include:

Mirage/An Efficient Music Similarity Library The *Mirage* music similarity library implements the music similarity measure as described in [ME05]. The library is presented in *Section 5.2.* It is programmed in the C# Language [Spe02] and uses the Open Source Mono[13] .NET framework to implement its functionality. It includes all performance optimizations proposed in the chapters before.

1. Direct feature extraction from MP3 files. MP3 files are read through a modified MP3 decoder. This makes full decoding of the MP3, re- and downsampling unnecessary and eliminates the need for a subsequent STFT to analyze a file, speeding up the feature extraction process (see *Section 4.1*). Byproduct is a modified MP3 decoder which can be used for other audio analysis tasks involving MP3s.

2. Optimized similarity computation. By taking the properties of the selected music similarity algorithm into account, many redundant arithmetic operations could be discarded. All those findings were implemented in a very optimized way making *Mirage* high-performance.

An Automatic Playlist Generation Plugin *Mirage* was integrated as a plugin in a popular digital music player program. In addition to making *Mirage* easily usable in the player, a fast, incremental playlist generation algorithm is proposed and implemented. The digital audio player was chosen to be easily extensible to integrate well with an automatic playlist generation plugin. *Figure 11* shows the music player program with the automatic playlist generation plugins ready for use. The plugin, its usage and the proposed playlist generation algorithm are described in detail in *Section 5.3.*

[13]`http://www.mono-project.com/`, last visited March 13, 2007

The Traveler's Sound Player Implemented on an Apple iPod A proof-of-concept for the idea of making use of music similarity on a portable MP3 player was realized on an Apple iPod. It shows how *Mirage* could be used to support a popular portable audio player with music similarity information. This is described in *Section 5.4*. The idea is based on the so called "Traveler's Sound Player" was presented in [PPW05b].

All source code of *Mirage* and its applications described here is freely available under the GPL License Version 2. Its homepage is `http://hop.at/mirage` where the source code can be downloaded.

5.1. Selecting and Modifying an MP3 Decoder

As described in *Section 4.1.2*, the MP3 decoding process needs to be intercepted just before the decoded signal is transformed from the frequency-domain into the time-domain. In the MP3 decoding process (depicted in *Figure 2*) the interception has to be done right after the IMDCT and antialiasing step before the signal resynthesis. The computationally intense resynthesis step can be skipped.

For the *Mirage* library Mp3Sharp[14], a native MP3 decoder written in C#, was evaluated. Mp3Sharp is a port of the opensource Java MP3 decoder JavaLayer[15]. Similarly as in Java, Mp3Sharp operates in a fully managed C# environment and unfortunately performs very poorly. The idea to develop *Mirage* entirely in C# was therefore discared. At least the modified MP3 decoder had to be developed outside the managed .NET stack. For this reason well known opensource MP3 decoders were evaluated regarding decoding performance. *Table 7* summarizes the results of the evaluation.

The managed decoder Mp3Sharp is about six times slower than Mpg123, the fastest evaluated MP3 decoder. Mpg123 is also two times faster than Madplay and the Libmad based decoder, SoX. Ffmpeg is performing fast too, but still Mpg123 is faster and was therefore picked for *Mirage*.

[14]`http://www.heroicsalmonleap.net/mle/mp3sharp/`, last visited March 13, 2007

[15]`http://www.javazoom.net/javalayer/javalayer.html`, last visited March 13, 2007

MP3 decoder **Homepage** **Evaluation command**	**Decoding Time (s)**

madplay,
`http://www.underbit.com/products/mad/`
`madplay -o wave:test.wav test.mp3`
	$4.261s$

mpg123,
`http://www.mpg123.de/`
`mpg123 -w test.wav test.mp3`
	$2.104s$

ffmpeg,
`http://ffmpeg.mplayerhq.hu/`
`ffmpeg -i test.mp3 test.wav`
	$2.474s$

sox,
`http://sox.sourceforge.net/`
`sox test.mp3 test.wav`
	$4.639s$

mp3sharp,
`http://www.heroicsalmonleap.net/mle/mp3sharp/`
`Mp3Sharp.exe test.mp3`
	$12.203s$

Table 7: Speed comparison of MP3 decoders. Ten different MP3 files were used to test the speed of the MP3 decoders. Timings are averaged over three consecutive runs and the ten MP3s encoded with $192kbit/sec$ at $44.1khz$. Average song length was $286sec$.

Mpg123 Mpg123 is able to decode MPEG-1 Audio Layer 1, 2 and 3 files. It is available under the open source Lesser GNU Public License (LGPL), which allows modifications to the source if the modifications to the source code are published too. For this project the source code for the Audio Layer-3 decoder-part was modified. The modification is called Fft123 and is available on the Internet[16]. Like Mpg123, Fft123 is released under the LGPL.

Modifying Mpg123 was straightforward. The main loop for MP3 decoding is in the `layer3.c:do_layer3()` function. Right after the IMDCT (`layer3.c:III_hybrid()`) the output buffer is simply written to a file and the resynthesis is skipped by returning from `layer3.c:do_layer3()`, after the buffer is written:

As mentioned before in *Section 4.1.2*, the resynthesis can be left out, since the transformation into the time-domain is not necessary. Skipping resynthesis is done in Fft123 by returning to the main library right after writing to the file descriptor with `fwrite()`. This makes the processing

[16]`http://hop.at/mirage/fft123/`, last visited March 13, 2007

```
layer3.c:do\_layer3()

       III_get_scale_factors();
       III_dequantize_sample();
       III_antialias();
       III_hybrid();

+      for (i = 0; i < SSLIMIT; i++) {
+          for (j = 0; j < SBLIMIT_CLIP; j++) {
+              fbuf[j*SSLIMIT + i] = hybridOut[0][i][j];
+          }
+      }
+      fwrite(fbuf, sizeof(real), SSLIMIT*SBLIMIT, FILE);
+
+      return clip;
```

Figure 6: The changes needed to be made to the Mpg123 MP3 decoding function are marked with "+". The figure shows a simplified version of what has to be done to write the frequency representation of an MP3 to a file. Changes like opening the file, closing it and infrastructure around these calls are left out for convenience, but are of course necessary.

even faster, since full decoding is not needed. Skipping the resynthesis enables one to read an MP3 about 2.5 times faster compared to the full decoding process. *Table 8* illustrates this.

MP3 decoding (`mpg123`)	**Power spectrum decoding (`Fft123`)**
$0.954s/mp3$	$0.382s/mp3$

Table 8: Speed comparison of fully decoding an MP3-file to PCM and only decoding the power spectrum with Fft123. The average speed over decoding ten MP3 files encoded at $192kbit/sec$ with $44.1khz$ to a mono/$11025khz$ signal is given in the table. The average length of the MP3 files was $286sec$. The respective calls to compare the decoding speed were: `fft123 -m -w test.fft test.mp3` and `mpg123 -m -4 -w test.wav test.mp3`

Fft123 is a standalone application. The modified Mpg123 decoder writes the powerspectrum to a specified file. The file is basically a raw floating point matrix ($576 \times n$). The frames (576, $32bit$ float values) are stored sequentially. The frame's values are spread linearly over half of the sampling frequency of the original MP3 file ($fs/2$). A single value of an MP3-frame at $44.1khz$ describes a frequency range of $39hz$.

The syntax for calling Fft123 is documented in *Figure 7*. *Figure 8* visualizes the output of Fft123 using two different MP3 music files as input.

```
fft123 [-k N] [-n N] -m -w <ps-filename> <mp3-filename>

Writes the power-spectrum as encoded in <mp3-filename>
to <ps-filename>

       -m     power spectrum mode
       -w     <ps-filename> write power spectrum to <ps-filename>
       -n N   decode only N frames
       -k N   skip first N frames
```

Figure 7: Usage of the fft123 command.

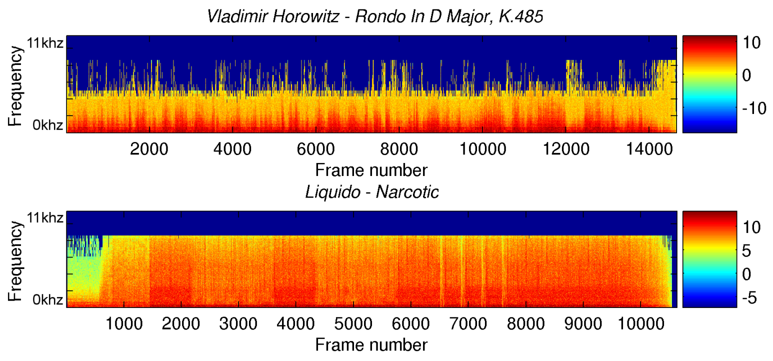


Figure 8: Visualization of the Fft123 powerspectrum output. Two $128kbps$ MP3 files were used. It can be seen that high frequencies are cut off, because the encoder perceives them as inaudible. However, in some frames higher frequencies were conserved by the encoder to ensure proper signal reconstruction. The amount of high frequencies which are conserved by the encoder depends on the bitrate used to encode the files. The two files are very different kind of music. The Horowitz piece is piano music while Liquido is heavy guitar punk music.

5.2. Library

The core of *Mirage* is its main library - *Mir*. The functionality of this library can be easily included into any C# program requiring access to a fast music similarity measure.

During implementation of the library, it was taken care that where possible high performance was a goal. Internally this sometimes led to unattractive coding results, since some object oriented design patterns had to be thrown overboard. From a user's point of view these optimizations are fortunately hidden.

Important points in achieving a high performance music similarity library were:

- Usage of the C# `unsafe` operator in large matrix operations, to disable array boundary checks.
- Directly exposing variables in objects, to avoid slow access through get/set functions. This is against all object oriented ideas, but was absolutely necessary to perform fast.
- Integration of the SQLite[17] database into *Mirage* to store the models.

Usage of *Mirage* is very simple. To compute the similarity between two MP3 files three lines of code are enough (*Figure 9*).

```
Scms m1 = MirNg.Analyze('song1.mp3');
Scms m2 = MirNg.Analyze('song2.mp3');

System.Console.WriteLine('Similarity between m1 and m2 is:' +
    m1.Distance(m2));
```

Figure 9: Using the *Mirage* library for similarity computation.

For a full documentation of the Library including usage examples see *Appendix A*.

[17] `http://www.sqlite.org/`, last visited March 13, 2007

5.3. Banshee Plugin

Banshee[18] is a program to manage large music collections on the GNOME desktop environment[19]. It indexes the user's music collection and makes rearranging, renaming, creating playlists, and listening to music easy. It has interfaces for the most popular portable digital audio players and is able to rip and import audio CDs. It is a typical "digital music hub". Programs like this are usually referred to as iTunes[20]-clones, since Apple made this way of managing music popular with its iTunes application.

Banshee was chosen as a base platform to develop an automatic playlist generation plugin based on the *Mirage* music similarity library. The plugin should handle large music collections, make easy playlist generation possible and integrate seamless into the player. Banshee was selected, because it is programmed in C# and has a very flexible plugin interface to integrate well with an automatic playlist generator. *Mirage* fits into Banshee very well, due to the fact both are written in the C# programming language.

Registering as a plugin with Banshee is easy. The plugin has to put itself into the `Banshee.Plugins` namespace and derive the methods from the abstract class `Banshee.Plugins.Plugin`. Multiple functions need to be overwritten to be recognized as a Banshee plugin (*Figure 10*).

```
public class MiragePlugin : Banshee.Plugins.Plugin
{
    protected override string ConfigurationName;
    public override string DisplayName;
    public override string Description;
    public override string[] Authors;
    protected override void PluginInitialize();
    protected override void PluginDispose();
}
```

Figure 10: Functions a Banshee plugin has to implement.

After the plugin is registered with Banshee through the `PluginInitialize()` call, the music library of the user is immediately scanned for new MP3s. If new MP3s are found, they are analyzed with the help of *Mirage*

[18]`http://banshee-project.org/`, last visited March 13, 2007
[19]`http://www.gnome.org/`, last visited March 13, 2007
[20]`http://www.apple.com/itunes/`, last visited March 13, 2007

and added to the database to be used for playlist generation. This is done in a background thread, so all playlist generation functions are available in parallel. If a playlist is generated while new MP3s are being analyzed, files which are not yet analyzed are simply not considered during playlist generation. `PluginInitialize()` adds the user interface elements of the plugin to Banshee.

5.3.1. Playlist Generation Algorithms

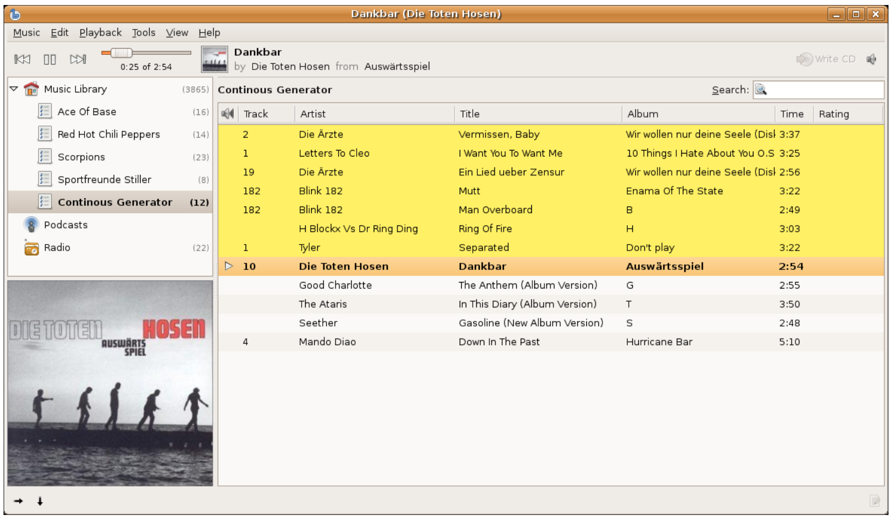
The plugin implements two possible ways to generate a playlist. Both methods can be used in Banshee by dropping the seed song(s) on the respective playlist generator item (as seen in *Figure 11*).

The Standard Playlist Generation Algorithm. This is a very straightforward way to automatically generate a music playlist. A playlist is computed using one or more seed songs. If a single song is used as a seed song first the according similarity model is searched in the database. After the model is available to the algorithm, the seed model is compared with all other song models in the database and their pairwise similarity is computed. After that the resulting list of similarities is ordered and the n closest songs are returned as the final playlist. If $n > 1$ songs are given as a seed, the similarity values are summed and weighted equally.

The Continuous Playlist Generation Algorithm. The standard playlist generation as described before has one issue. All songs in the playlist are just similar to the seed song. The playlist could contain songs which are found to be similar because of different acoustic aspects. So they are all similar in some aspect to the seed song, but are not necessarily similar among each other, which is disturbing when listening to the playlist piece-by-piece.

To work around this issue, a continuous playlist generation algorithm was implemented. The basic playlist generation concept to find similar songs stays the same but the playlist dynamically adopts itself as the user progresses with it. When the algorithm is initialized by a seed song only the five most similar songs are returned. If the user then then starts listening and plays more than 60% of a song, a new playlist, taking the currently played song as seed, is appended and replaces the unplayed songs from the old playlist. This simple technique ensures continuous music experience and works very well. By listening to music this way, the user can also easily develop the playlist in a certain musical direction of his choice, by simply skipping songs he does not like, to initiate new playlists on suggested songs he does like. It is the recommended technique to automatically generate playlists in Banshee.

5.3.2. Using the Plugin



Track	Artist	Title	Album	Time	Rating
2	Die Ärzte	Vermissen, Baby	Wir wollen nur deine Seele (Disl	3:37	
1	Letters To Cleo	I Want You To Want Me	10 Things I Hate About You O.S	3:25	
19	Die Ärzte	Ein Lied ueber Zensur	Wir wollen nur deine Seele (Disl	2:56	
182	Blink 182	Mutt	Enama Of The State	3:22	
182	Blink 182	Man Overboard	B	2:49	
	H Blockx Vs Dr Ring Ding	Ring Of Fire	H	3:03	
1	Tyler	Separated	Don't play	3:22	
▷ **10**	**Die Toten Hosen**	**Dankbar**	**Auswärtsspiel**	**2:54**	
	Good Charlotte	The Anthem (Album Version)	G	2:55	
	The Ataris	In This Diary (Album Version)	T	3:50	
	Seether	Gasoline (New Album Version)	S	2:48	
4	Mando Diao	Down In The Past	Hurricane Bar	5:10	

Figure 11: The Mirage plugin in the Banshee digital audio player.

In order to use the plugin in Banshee the following prerequisites need to be obtained and installed:

- Mono (Version >= 1.1.17) available at `http://www.mono-project.com`
- Fft123 available at `http://hop.at/mirage/fft123`
- The Banshee music player, of course.

Installation of the plugin is straightforward. `Mir.dll`, which can be obtained from the *Mirage* homepage has to be copied into the `/.gnome2/banshee/plugins` directory. After that Banshee needs to be started, where the plugin can now be activated. To activate it select “Edit” then “Plugins” on the Banshee menu and activate it. Immediately after the plugin was successfully loaded two new playlists appear on the right panel: “Playlist Generation” and “Continuous Playlist Generation”.

After the plugin is activated all songs which are availabe in the users collection need to be analyzed once first before automatic playlist generation is possible. This takes some time, depending on the size of the users collection and the computer being used. If Banshee was started from a console window, progress can be monitored there. When all songs are analyzed, they are ready to be used for automatic playlist generation.

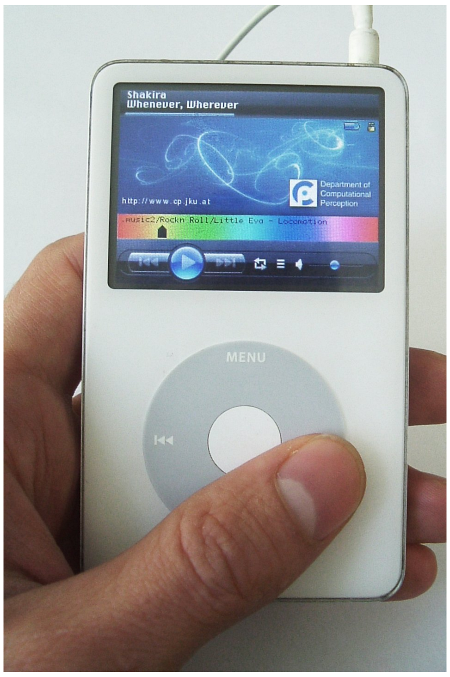
To use the playlist-generator in Banshee, a seed song to start with has to be dragged & dropped onto one of the playlist generator items on the left sidebar. You can see the sidebar and playlist generator items in *Figure 11*. The items “Playlist Generator” and “Continuous Generator” expose the two playlist generation techniques which were proposed in *Section 5.3.1*.

5.4. Portable Music Similarity on the iPod

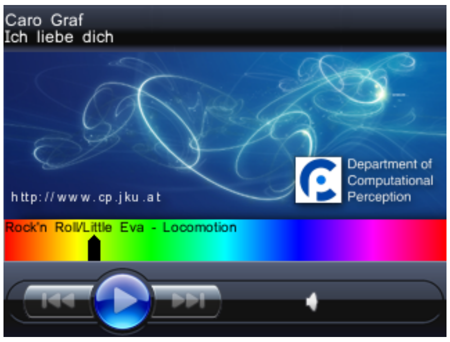
Another application of *Mirage* and music similarity in general was inspired by [PPW05b]. With the “Traveler’s Sound Player” it describes a way of making use of music similarity information on a portable audio player. The main idea of the player is to arrange all songs in a collection in a playlist, so that each subsequent song has on average maximum similarity with its predecessor. This is done by first computing the full audio similarity for each pair of tracks and second by using a travelling salesman algorithm for optimal arrangement of the playlist. With the automatic arrangement by similarity, a playlist with areas of similar songs emerges. These areas can be browsed by the Traveler’s Sound Player using a turning knob. When a selected song is finished, the player advances to the next similar track in the list.

The idea here was to use *Mirage* for music similarity computation and modify a real iPod MP3 player to work like the Traveler’s Sound Player. See *Figure 12(a)* for a picture of the modified iPod.

To make the TSP player idea work on the iPod, the collection first needs to be analyzed. This needs to be done offline on a computer since the processing power of an iPod can not cope with such tasks. After having



(a) A modified Apple iPod with Music Similarity.


Caro Graf
Ich liebe dich
Department of Computational Perception
http://www.cp.jku.at
Rock'n Roll/Little Eva - Locomotion

(b) User interface of the *Mirage* iPod application

Figure 12: Using *Mirage* on the iPod.

analyzed the collection, the full similarity matrix is computed. All of this is easily done using the *Mirage* library. The full similarity matrix is further needed to compute the optimum arrangement of the songs around the TSP player's knob. The optimum arrangement is calculated using Kruskal's minimum spanning tree algorithm [KJ56]. After that the playlist is ready to be transferred to the iPod.

To use the playlist on the iPod in a similar way like it is done on the TSP player, the iPod needs to be modified. Since Apple does not support modifications or third party plugins on the original iPod firmware, the iPod was modified to use the Open Source firmware Rockbox[21] which is freely available on the Internet. Rockbox comes with a full cross-compilation environment making development for the iPod ARM platform easy. Specific installation installation instructions of the firmware and the development environment are published on the Rockbox homepage[22]. The

[21] `http://www.rockbox.org/`, last visited March 13, 2007
[22] `http://www.rockbox.org/twiki/bin/view/Main/IpodPort`, last visited March

Open Source firmware was then adopted to match the functionality of the Traveler's Sound Player (*TSP*):

- The iPod wheel is used to emulate the turning knob in the TSP player. To do so the main loop in the Rockbox firmware was modified to check the iPod wheel sensor for its status and position.
- The standard user interface of the Rockbox firmware was modified to allow TSP player like browsing of the music collection. The main difference between the TSP player and this implementation is, that the songs are not arranged along the iPod wheel/a turning knob, but along a colorbar on the screen (see *Figure 12(b)*). By sliding the finger over the iPod wheel sensors, different regions of the colorbar can be selected. The current selection is indicated on the screen with a black arrow. On top of the colorbar the currently selected song is displayed and can be listened to by pressing the select button on the iPod. The rainbow colors of the colorbar have no meaning, they just support the user in learning where different styles of music can be found.

So actual usage on the modified MP3 player is relatively simple: The scrollwheel of the iPod allows to move the slider on the colorbar into different color regions and thus music similarity areas. Once a song or area is picked, it can be played by using the select button from the iPod. When the currently selected song is finished, the player progresses to the next most similar song in the list. It so allows fast browsing of a music archive and quick selection of music styles to listen to, making it perfect for quick playlist generation on-the-go and thus showing another nice possibility of how to use *Mirage* for music similarity applications.

13, 2007

6. Evaluation

This section evaluates performance and quality of the music information retrieval library *Mirage*. All evaluations were done on the same computer, in the same operating system and environment. The machine where everything was evaluated was an AMD AthlonXP 3000 machine with 2100MHz. It has 1GB of RAM and a fast 250 Gigabyte Serial ATA harddrive. All tests were run under the Ubuntu Linux (Version 7.04, "Feisty Fawn") operating system with Mono C# (Version 1.2.3) installed.

To compare *Mirage* and its feature extraction, *Mirage* was extended for testing purposes. It was adapted to include normal processing of MP3 files. In the adapted version these steps are done during feature extraction (see *Figure 1*, which depicts the usual way of processing MP3 files).

- The MP3 was fully decoded using Mpg123: `mpg123 -q -n 4600 -k 570 -m -w wavfile mp3file`
- The PCM data is then downsampled to 11025hz mono using SoX: `sox -q -t wav wavfile -r 11025 -c 1 -l -f -t raw rawfile rate`
- The raw downsampled PCM data is then windowed using a standard 2048 point Hann window, with no overlapping. An STFT is computed for each window using the fast Fftw library[23].
- Further processing is the same as in the standard *Mirage* library.

In the experiments this feature extraction process will be referenced as "standard" feature extraction, the fast feature extraction implemented in *Mirage* will be referenced as "Fft123" feature extraction. Like for the main library, it was taken care that standard feature extraction works as fast and well as possible.

6.1. Testsets

There were four different testsets used for evaluating *Mirage*. The testsets differ in size, genre distribution and, of course, music style.

[23]`http://www.fftw.org/`, last visited March 13, 2007

	Small	*ISMIR04*	*Personal*	*Huge*
Pieces	120	1311	3852	16781
Genres	16	8	*n/a*	21

Table 9: Statistics and names of the testsets used for evaluating *Mirage*.

Small The smallest collection is a very artificial one. It consists of only 120 different songs, but compared to the other collections it has a very broad genre distribution. This collection was hand selected. It was taken care that all genre classes are about the same size. The genres of this collection include:

Genre	*Tracks*	*Genre*	*Tracks*
Alternative	7	Classic Orchestra	6
Dance	9	Happy Sound	6
Hip Hop	12	Pop	6
Rock	4	Romantic Dinner	6
Blues	8	Classic Piano	9
Eurodance	9	Hard Pop	8
Mystera	8	Punk Rock	9
Rockn Roll	7	Talk	6

Table 10: Genres and number of tracks for the Small testset.

The full list of tracks and their genre assignment for this testset can be seen in *Appendix B*. This small collection is quite useful for quick tests.

ISMIR04 This collection consists of 1311 songs. The songs are all royalty free and come from Magnatune[24]. They can be downloaded for free and were being made available at the ISMIR conference in 2004. It is still possible to download the full collection from the ISMIR 2004 website[25] and compare genre classification results with this testset.

Personal This is a personal user's collection and consists of 3852 MP3 files. This music collection is just used for performance measuring of

[24] `http://www.magnatune.com/`, last visited March 13, 2007

[25] `http://ismir2004.ismir.net/genre_contest/index.htm`, last visited March 13, 2007

Genre	*Tracks*	*Genre*	*Tracks*
Classical	640	Electronic	229
Jazz & Blues	52	Metal	29
Pop	6	Punk	16
Rock	95	World	244

Table 11: Genres and number of tracks for the ISMIR04 testset.

the feature extraction and comparison. The tracks are not assigned to genres.

Huge The Huge collection consists of 16781 different files, which have all assigned genre labels from GraceNote[26].

Genre	*Tracks*	*Genre*	*Tracks*
Alternative & Punk	2132	Blues	275
Books & Spoken	187	Classical	627
Country	1894	Easy Listening	134
Electronica & Dance	2828	Folk	174
Gospel & Religious	10	Hip Hop & Rap	718
Holiday	68	Industrial	22
Jazz	3388	Latin	378
Metal	53	New Age	280
Pop	550	R&B	1120
Reggae	227	Rock	1395
World	321		

Table 12: Genres and number of tracks for the Huge testset.

6.2. Performance

Since high performance was a key goal in this project, it is evaluated thoroughly. First the two different feature extraction processes (Normal and Fft123) are compared to measure the performance gain of using the Fft123 method. In a second part the optimized KL divergence component of the feature comparison is compared to an unoptimized one.

[26]`http://www.gracenote.com/`, last visited March 13, 2007

6.2.1. Feature Extraction

To compare feature extraction performance, the *Personal* music collection was analyzed. A small program using the *Mirage* library was written to test performance. Two runs were conducted, in the first run normal feature extraction was used, in the second run feature extraction using Fft123 was activated and timed. The runtimes are illustrated in *Figure 13*.

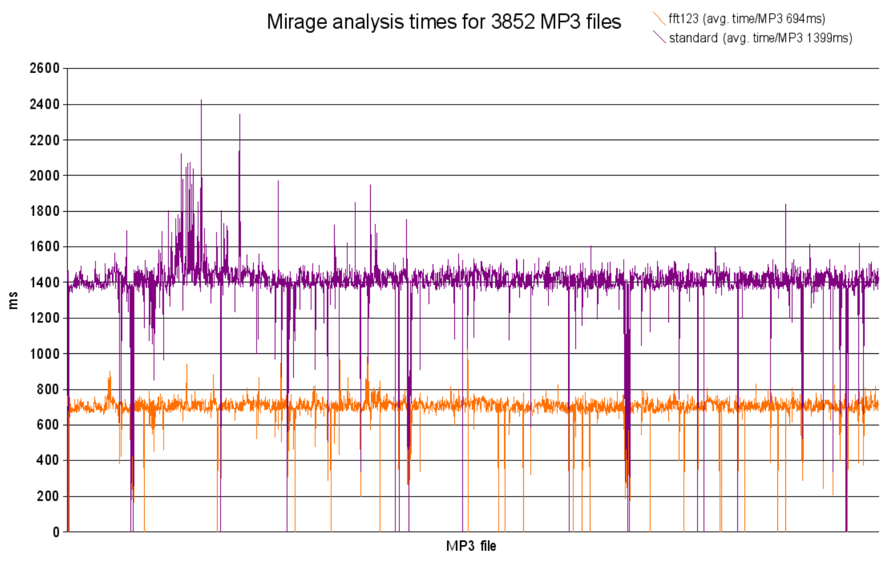


Figure 13: Speed of importing MP3 files using two the *standard* and *Fft123* feature extraction technique. Standard feature extraction is about twice as slow as Fft123 extraction. In total, analysis of the whole collection took 45 minutes using the Fft123 technique and 1 hour 29 minutes using the standard way. The fluctuations come from different MP3 encoding quality. If the graph touches the zero line, the MP3 could not be decoded or an error occurred during the process.

During the feature extraction of the 3852 MP3 files, some files failed to produce usable feature matrices. This usually happens if the MP3 decoder is not able to properly decode the file. Typical observed failures for Fft123 were mislabled MP3 files and non-standard Variable bitrate files (see *Table 13*).

Failed files (standard)	Failed files (fft123)
20	42

Table 13: Failed MP3 files in the feature extraction. Fft123 is not as robust yet. Some mislabled MP2 files or non standard conforming files are left out, since Fft123 is unable to process them properly yet.

6.2.2. Feature Comparison

To compare performance between an unoptimized version of the KL divergence and the optimized version in *Mirage*, the runtime for 100000 comparisons was measured and averaged. The standard KL divergence without the possible optimizations was implemented in Matlab. Matlab was chosen, because it is a mathematical programming language optimized for fast matrix operations. It does the computation very fast, but does not optimize any part of it, whereas the C# version allows all fine grained optimizations to be implemented. The Matlab code to compute the standard symmetrized KL divergence is depicted in *Figure 14(a)*. The variables `c1, c2` are the covariance matrices, `ic1, ic2` the precomputed inverse of the covariance matrices and `m1, m2` the mean vectors of the two models.

```
tic;
for i=1:100000,
    kld = trace(c1*ic2) + trace(c2*ic1)+...
        trace((ic1+ic2)*(m1-m2)*(m1-m2)');
end
toc/100000
```

(a) Matlab

```
Timer t = new Timer();
t.Start();
for i=1:100000 {
    float kld = scms1.Distance(scms2);
}
Dbg.WriteLine(t.Stop());
```

(b) *Mirage*

Figure 14: The KL divergence performance test implementations.

The code which was used to evaluate the speed of the *Mirage* library is shown in *Figure 14(b)*. The variables `scms1, scms2` are the two statistical cluster model object instances to be compared.

The results of comparing the two versions can be seen in *Table 14*. The optimized KL divergence implemented in *Mirage* is about 100 times faster than the Matlab version, but yields exactly the same results, since only unnecessary computations are left out. When working with reduced floating point precision, the optimized version also yields more accurate results, because less operations need to be carried out which results in a smaller overall error.

Feature Comparison (Matlab)	Feature Comparison (Mirage)
1.297075ms	0.01262ms

Table 14: Speed of a single feature comparison using Matlab and *Mirage*. *Mirage* is about 100 times faster than the standard Matlab code

6.3. Quality

As mentioned the faster feature comparison has absolutely no impact on the quality of the similarity computation results. But what impact does the faster feature extraction process have on the quality of the models which are extracted?

To be able to somehow compare the quality of the music similarity algorithm, automatic genre classification tests were carried out. Automatic genre classification tests can be done if all tracks in a collection are labeled with a genre. When doing this kind of test, it absolutely should be kept in mind that a genre label is no indicator of music similarity at all. If the genres of two songs match, it is just more probable that they sound similar, nothing more. Despite of these problems with music genres, automatic genre classification is a good indicator of how good a music similarity algorithm works. Genre classification tests are currently accepted as a method to automatically evaluate music similarity algorithms. They are also part of the annual genre classification contest carried out at the ISMIR[27] (International Conference on Music Information Retrieval).

[27]`http://www.ismir.net/`, last visited March 13, 2007

The automatic genre classification carried out here is a simple nearest-neighbor leave-one-out classification and produces a genre confusion matrix:

1. All pieces in a music collection are assigned an adequate genre label.
2. The files in the collection are analyzed and the full similarity matrix is computed for all pieces in the collection.
3. Then a genre classification confusion matrix is computed so that each song is assigned the genre of its most similar song.
4. In the confusion matrix the predicted genre of a song is plotted against its actual membership.
5. The confusion matrix diagonal shows the classification accuracies for each genre, which is the number of correctly classified songs divided by the total number of songs in the class.

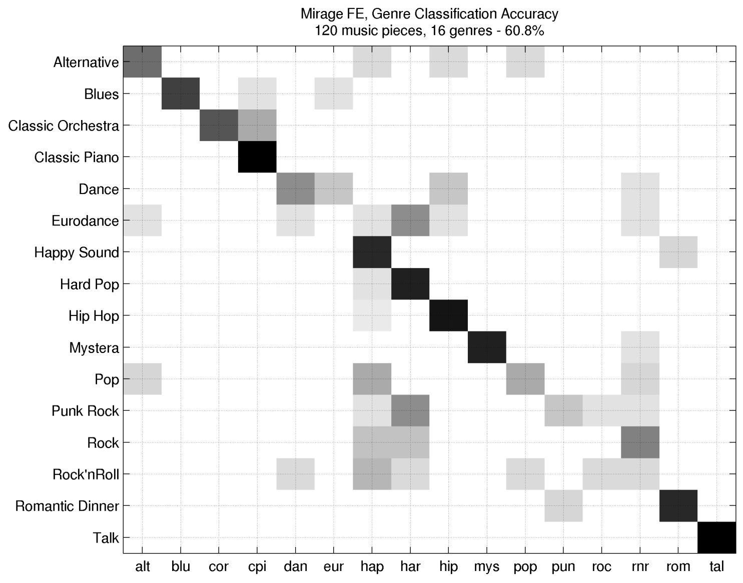
Here are the overall results from the genre classification tests for the *Small*, *ISMIR04* and *Huge* collections.

Classification Accuracy	Small	ISMIR04	Huge
Standard Method	60.8%	82.8%	52.0%
Fft123 Method	52.2%	82.2%	52.0%

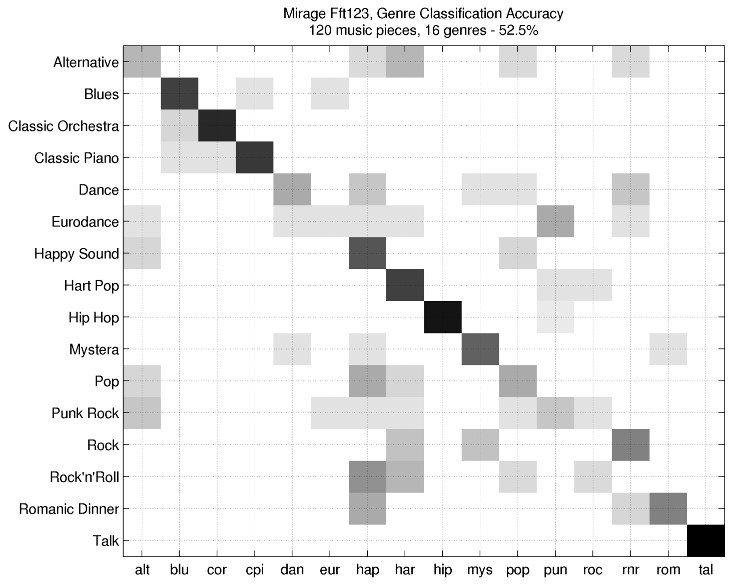
Table 15: The overall genre classification accuracy. It is defined as the number of all correctly classified songs divided by the number of songs.

In general it can be seen that the genre classification accuracy of the Fft123 method is a bit lower than the accuracy of the normal method. 8% difference in the Small testset may seem large at first, but absolute difference is only 10 falsely classified songs. The fluctuations in this testset are large. As the other tests on the *Huge* and *ISMIR04* sets show, genre classification accuracy is about the same. This makes working with the fast Fft123 feature extraction a compelling option, giving a two fold speed increase during feature extraction.

Detailed results of the genre classification tests are visualized on the following pages via confusion matrices.

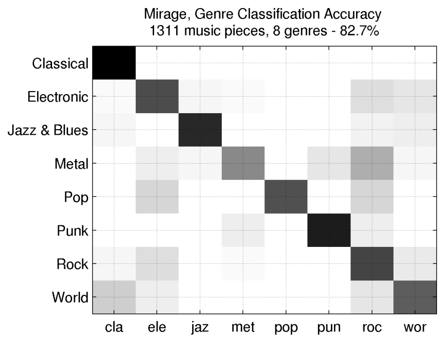


(a) Standard

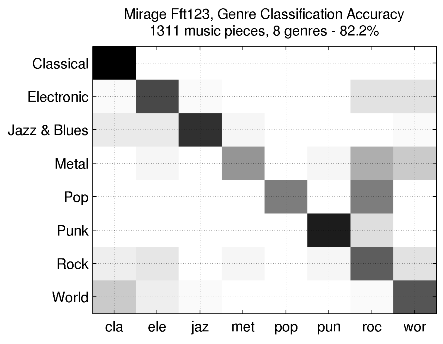


(b) Fft123

Figure 15: Genre classification confusion matrix for the *Small* collection.

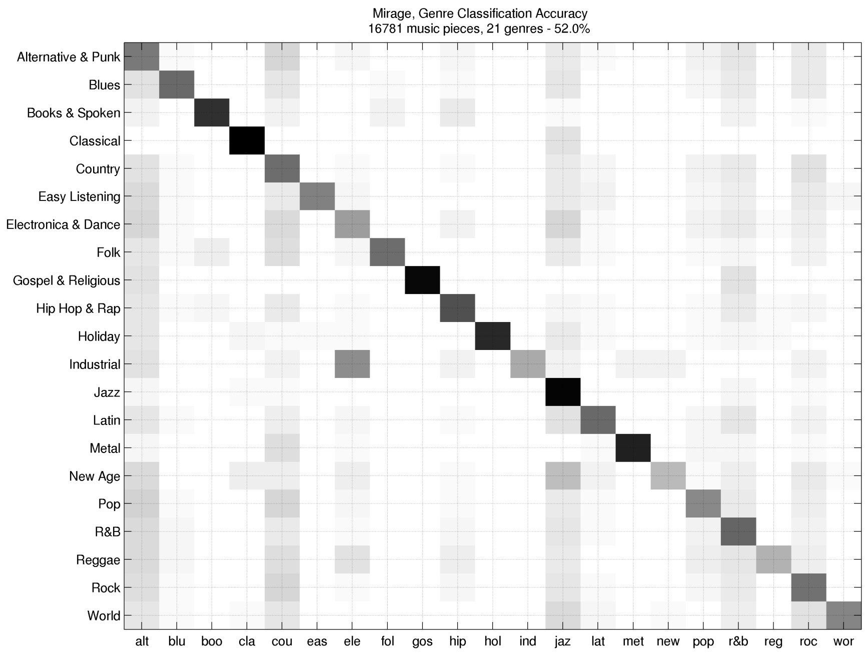


(a) Standard

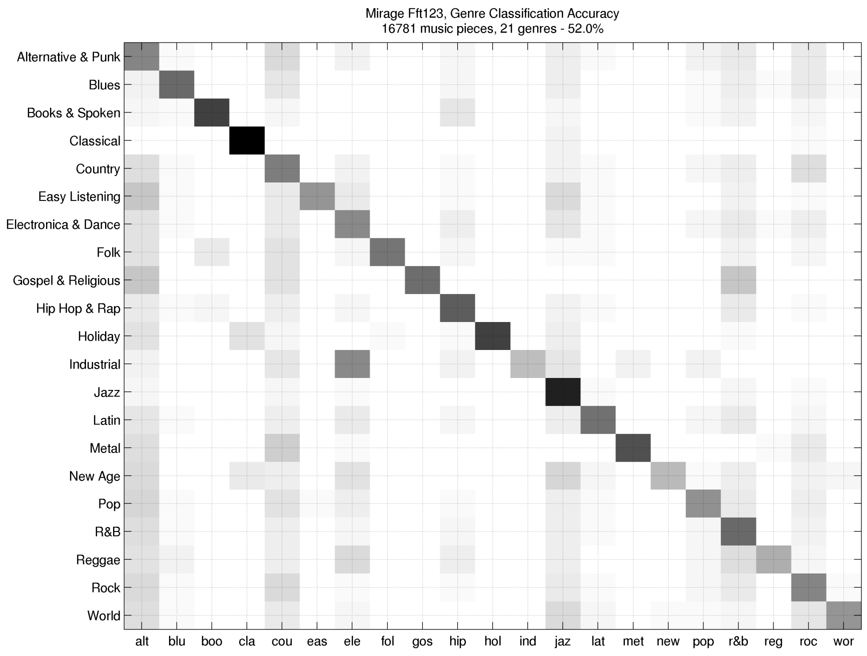


(b) Fft123

Figure 16: Genre classification confusion matrix for the *ISMIR04* collection



(a) Standard



(b) Fft123

Figure 17: Genre classification confusion matrix for the *Huge* collection

6.4. Subjective Evaluation

To give the reader a feeling how playlist generation works, some example playlists are presented in this section. All playlists were generated on the *Personal* collection. Good, bad and average results are shown.

	Ace of Base - Beautiful Morning (Groove Radio Edit)
1	Stefan Raab - Ein Bett im Kronfeld
2	Lara Fabian - I Am Who I Am (Album Version)
3	Vanessa Amorosi - Absolutely Everybody
4	Anastacia - You Trippin' (Album Version)
5	Black Eyed Peas - Where Is The Love (Radio Edit)
6	Die Deutschmacher - Geh West
7	Melanie Thorton - Wonderful Dream
8	Antonia - Wenn Der Hafer Sticht (Radio Version)
9	Basis - Ich Lieb' Dich Immer Noch
10	JoJo - Leave Out (Get Out)

Table 16: An average playlist. The seed song is a typical happy and groovy kind of music. The best result is the 4^{th} song which fits perfectly to the seed. The grey highlighted pieces do not fit in the playlist, they are both German songs and are very country style.

	Die Ärzte - Sommer, Palmen, Sonnenschein
1	Die Ärzte - Vokuhila
2	Die Ärzte - Ein Lied für dich
3	Audiosmog feat. Tobi Schlegel - Daylight in your Eyes
4	Die Ärzte - Helmut Kohl Schlägt Seine Frau
5	Die Prinzen - Deutschland
6	Die Ärzte - Schunder Song
7	JBO - Ich Sag' J.B.O.
8	Lorie - Week End (Album Version)
9	Franz Ferdinand - Darts of Pleasure
10	The Ordinary Boys - Boys Will Be Boys

Table 17: A very good playlist result. Initial seed song is a guitar heavy punk rock song. Three similar songs come from the same artist, and the other songs fit the theme of the seed very well.

One may argue that the very good playlist (see *Table 17*) is not that good at all, since the French song *Lorie - Week End (Album Version)* does not fit into a playlist generated with a German punk rock song, or it is no punk rock at all. But that kind of results is the best you can expect, since this method does not do language nor genre detection. It is best in finding songs with similar instrumentation, making it a good choice for casual music listeners, not for music purists.

	Robbie Williams & Nicole Kidman - Something Stupid
1	Marc Anthony - You Sang To Me
2	Jürgen von der Lippe - Guten Morgen Liebe Sorgen
3	Ren - Rainyday
4	Blondie Maria
5	Nena - Lass Mich Dein Pirat Sine (New Version)
6	Ayreon - My House On Mars
7	Udo Jürgens - Tausend Jahre Sind Ein Tag
8	Chicago - Let's Take A Lifetime
9	Sonny & Cher - I Got You Babe
10	Steiermark Quintett - Matterhorn

Table 18: A very bad playlist result. Initial seed song is a very romantic song. Only 5 out of 10 songs were romantic too, and matched the seed theme.

	Robbie Williams & Nicole Kidman - Something Stupid
1	Marc Anthony - You Sang To Me
2	Marc Anthony - Yo Te Quiero
	~~Marc Anthony - Amor Aventurero~~
3	Anastacia - How Come The Work Won't Stop
	~~Anastacia - One More Chance~~
	~~Anastacia - Secrets~~
	~~Scorpions - White Dove~~
4	Lara Fabien - No Big Deal (Album Version)
5	Celine Dion - A New Day Has Come
	~~Shakira - Whenever, Wherever~~
6	Alizee - Moi Lolita
7	Mariah Carey - Vision Of Love
8	Sarah Connor - Sweet Thang
9	Sarah Connor - Make U High
10	Sarah Connor - Can't Get None

Table 19: A continuous playlist generated using the continuous playlist algorithm proposed in *Section 5.3.1* initiated with the same seed song as in the playlist from *Table 18*. The last song listened to is always the seed for the next playlist. Songs which were skipped are crossed out. This results in a very continuous music experience, in contrary to *Table 18* no unfitting songs appear in during listening to the playlist. Even the skipped songs fit into the romantic theme of the seed song. To avoid repeating songs from the same artist, an artist filter could be added easily.

The bad playlist result shown in *Table 18* shows some weaknesses of the algorithm, namely songs where the statistical spectrum models do not suffice to produce good playlists. One can escape these very bad results when using the continuous playlist generator, a technique which was implemented in the *Mirage* Banshee plugin and was presented in *Section 5.3.1*. *Table 19* uses the same seed song, but the continuous playlist generation algorithm. The songs which are crossed out were skipped, the other songs in the playlist were used during the continuous playlist generation process.

The continuous playlist generator can be seen as a generator which gives intelligent proposals for the next song to be played.

7. Conclusion

The work on the *Mirage* library and its surrounding components which was described in this thesis should be seen as a starting point to make music similarity work for a broader audience. Primary goal was a scalable implementation of a well working music similarity algorithm usable for large digital music collections which users tend to have nowadays.

7.1. Summary

This thesis has demonstrated how to achieve high performance improvements on a selected music similarity algorithm. More precisely faster ways to extract features and compare them were proposed. Optimized feature extraction dealt with extracting features directly from compressed audio files like MP3s, and an improved feature comparison method showed that there is much space to optimize the Kullback Leibler divergence for similarity comparisons.

Mirage, a music similarity library, was developed (*Section 5*) including all proposed optimizations. To demonstrate the utilizability of the library, a plugin for a music player was written making automatic playlist generation possible for everyone. Besides this an iPod was modified to show how music similarity information could already be used on portable music players. Evaluations of the library (*Section 6*) evince that feature extraction performance could be doubled and similarity computation could be accelerated to be about 100 times faster than an unoptimized routine. All of this is possible without reducing the quality of the similarity algorithm results.

7.2. Future Work

To improve on the performance and quality several directions for further work are possible: *Mirage* could be extended to directly work with other popular compressed music filetypes besides MP3. Popular audio filetypes like AAC, WMA or OGG are important candidates. The speed of a similarity computation could be enhanced by using heuristics to faster find matching song models. For the chosen way to compute similarity models are nothing else than a gaussian distribution. A simple heuristic could just look at the mean values to prefilter songs which are definitely not similar to the seed song to exclude them from similarity-search.

More work could also be done on improving the quality of the similarity measure. The measure could be made better by combination with other methods like it was successfully done in [Pam06a]. Additional descriptors could be used to postfilter playlist results to, for example, include only songs with similar tempo[Ell06]. Tempo is a good candidate, because rhythmic aspects are wholly left out in the current similarity measure. Another way to improve the quality could be the inclusion of an outlier detection filter like described in [FPW05b] to filter a playlist for unwanted outliers.

To summarize, this work has shown some very compelling enhancements to current music similarity algorithms, which should be thought about when designing a high performance music similarity application, making it hopefully useful for further research in this topic.

Appendix A

A. Mir Library Documentation

A.1. Main Classes

MirNg (← System.Object)

Is the main class, which is used for playlist generation and searching for similar songs. Usually one just uses the `Analyze()` method to create a `Scms` similarity object out of an MP3. To get a playlist use the `SimilarTracks()` method, which queries the database for the most similar tracks. Adding track models to the database is done with the `Db` class.

```
static Scms Analyze(string file)
```

Analyzes the given MP3 file for automatic playlist generation and returns a `Scms` object describing the MP3 file. The returned `Scms` can be used to compare the model directly to another `Scms` object or it can be stored in the database by using the `Db::Add()` method. The method internally utilizes the `Mpg123FileReader` and `Mfcc` classes to generate the Model.

file	The filename of the MP3 which needs to be analyzed.
Return value	The `Scms` object which was created by analyzing the MP3 file specified as parameter.

```
static void CacheIt(Db db)
```

Caches all `Scms` objects from the database for faster access. To use the cached objects and take advantage of the cache, the `SimilarTracksCached()` method has to be used after calling this method.

db	The database connection which should be used to cache the `Scms` objects.

```
static int[] SimilarTracks(int[] id, int[] exclude, Db db)
```

Searches for the most similar tracks to the track IDs in `id` and returns a sorted playlist by similarity. By using `exclude` IDs can be excluded from being considered in similarity computation.

id	An array of track IDs which should be included in finding the most similar tracks (or its `Scms` objects)
exclude	An array of track IDs which should be excluded from the search.
db	The database connection which should be used for the queries.

```
static int[] SimilarTracksCached(int[] id, int[] exclude)
```

Searches for the most similar tracks to the track IDs in `id` and returns a sorted playlist by similarity. By using `exclude` IDs can be excluded from being considered in similarity computation. This method uses a cache for retrieving the `Scms` objects and can only be used if the cache was initialized once with the `CachIt()` method.

id	An array of track IDs which should be included in finding the most similar tracks (or its `Scms` objects)
exclude	An array of track IDs which should be excluded from the search.

Scms (← System.Object)

Scms is short for Statistical Cluster Model Similarity and is the class in the *Mirage* library which integrates most of the Music Information Retrieval techniques. An instance of an `Scms` model can be obtained by creating it manually or using the `MirNg.Analyze()` method. The most important method in the `Scms` class is the `Distance` method - it computes the similarity between two `Scms` objects, and thereby the similarity between its associated tracks. Persistence functions like `FromBytes()` or ToBytes() can be used to store the models on disk, if the database provided by *Mirage* can not be used.

```
Scms(Matrix mfcc)
```

Creates a new Scms object using an MFCC matrix computed by the `Mfcc` class.

mfcc	An Mfcc `Matrix` which should be used to create the model from.

```
float Distance(Scms scms2)
```

Computes the similarity between two `Scms` models. The lower the returned value, the more similar are the compared `Scms` objects (and its associated tracks). The higher the returned value is, the more distinct are the two models. Because of this relation the function is called `Distance()`.

scms2	The `Scms` to compare.
Return value	The perceived similarity of two `Scms` objects. The closer they are the more similar are the associated tracks of the objects. This function can be used to create a playlist of similar songs.

```
static Scms FromBytes(byte[] buf)
```

This function is used to deserialize an `Scms` object from its raw byte array.

scms2	The serialized bytes, which represent the `Scms` object.
Return value	Returns the `Scms` object which was reconstructed using the given byte array

```
byte[] ToBytes()
```

Serializes the `Scms` object to a byte array. This byte array can be written to a file or database, and can be reconstructed by using the static `FromBytes()` method.

Return value	The Scms object serialized to a byte array.

`CovarianceMatrix cov`

The covariance describing the `Scms`. Do not write to this field. It is exposed this way for performance reasons.

`CovarianceMatrix icov`

The inverted covariance describing the `Scms`. Do not write to this field. It is exposed this way for performance reasons.

`Vector mean`

The mean vector describing the `Scms`. Do not write to this field. It is exposed this way for performance reasons.

Db (← System.Object)

The database connection object. It is used to store `Scms` models, which are used to for playlist generation. To iterate over all `Scms` objects available in the database you could use this strategy:

```
Db db = new Db();
int[] trs = db.GetTrackIds();
IDataReader = getTracks(trs);
Scms[] s = new Scms[10];
int[] m = new int[10];
Scms q = db.GetTrack(1);

while (int n=db.GetNextTracks(ref dr, ref s, ref m, 10 ))>0)
{
  // compute the distance to all tracks
  for (int i = 0; i < n; i++)
     float d = q.Distance(Scms[i]);
}
```

`Db()`

Initializes a new *Mirage* SqLite database. Looks in the directory `/.mirage/` if there exists a database (usually `db.sqlite3`. If it does, a database connection to this file is established. If not, a new *Mirage* database is created and a connection to it established.

```
int AddTrack(int trackid, Scms scms)
```

Adds the `Scms` song model to the databases using the internal database identificator `id`. After adding it to the database the `Scms` can be retrieved by using the `GetTrack()` or `GetTracks()` method.

trackid	The id of the Scms to add the track to the db.
scms	The model to add to the db. The `Scms` object is serialized and written to the database as a binary blob.
Return value	Returns the `trackid` if successful, `-1` otherwise.

```
Scms GetTrack(int trackid)
```

Retrieves one Scms for the given `trackid` from the database. If it does not exist, `null` is returned.

trackid	The id of the Scms to retrieve from the database.
Return value	The requested `Scms` object if the `trackid` was found in the database. `null` otherwise.

```
int[] GetTrackIds()
```

Returns all IDs from the database. This method is useful for iterating over all available `Scms` models in the database. It is usually used in conjunction with the

```
GetTracks()
```

method.

Return value	Returns an array of integer values, including all IDs of `Scms` objects found in the database.

```
System.Data.IDataReader GetTracks(int[] trackid)
```

Returns an `IDataReader` for the requested tracks. This method initializes an iteration over the selected track ids. To actually retrieve the `Scms` models use the `GetNextModels()` method.

trackid	An array of int values, which specify the track IDs to be retrieved. Usually the array is built by the `GetTrackIds()` method, if all `Scms` models need to be retrieved.
Return value	Returns an `IDataReader` object, which should be then used in the `GetNextModels()` method to retrieve the objects from the database.

```
int GetNextTracks(ref IDataReader tracksIterator, ref Scms[]
tracks, ref int[] mapping, int len)
```

Method to retrieve stored `Scms` objects from the database. This iterating method is called with an `IDataReader` as parameter, which has to be retrieved by the `GetTracks()` method.

ref tracksIterator	The `IDataReader` object returned from the `GetTracks()` call. This is used for further iteration over all values.
ref tracks	An array where all retrieved `Scms` objects are stored.
ref mapping	A mapping of the `tracks` to track IDs, to identify the `Scms` objects in the `tracks` array.
len	The maximum number of `Scms` objects to be returned at once.
Return value	Returns the number of proper returned `Scms` models. If it is lower than `len` or zero, all `Scms` objects were read and there are no more results available.

A.2. Basic Types

Matrix (← System.Object)

The Matrix class models a two dimensional matrix. It offers direct access to the values by exposing the data array `d`. Functions to add (`Add()`), substract (`Substract()`) or multiply (`Multiply()`) matrices are offered. Statistic functions to compute the mean vector (`Mean()`) or covariance (`Covariance()`) are implemented.

```
Matrix(int rows, int columns)
```

The constructor creates a new `Matrix` with the dimensions specified.

rows	Number of rows
columns	Number of columns
Return value	The new `Matrix`

```
Matrix Add(Matrix m)
```

Adds `m` to the `Matrix` and returns the summed `Matrix`. `m` needs to have the same dimensions as the `Matrix` to succeed, otherwise a `MatrixException` is thrown.

m	The `Matrix` to be added.
Return value	The summed `Matrix`.

```
Matrix Covariance()
```

Computes the Covariance Matrix.

Return value	The full covariance of the `Matrix`.

```
Matrix Inverse()
```

Uses the Gauss-Jordan elimination to compute the inverse A^{-1} of the Matrix, so that $A \times A^{-1} = I$, where I is the identity matrix. The Gauss-Jordan elimination is very fast. Calculations are done in decimal precision, to omit wrong results due to lack of numerical precision.

Return value	The inverse of the `Matrix`.

```
Matrix Inverse2()
```

Uses the LU decomposition to compute the inverse A^{-1} of the Matrix.

Return value	The inverse of the `Matrix`.

```
Vector Mean()
```

Calculate the mean vector for a `Matrix`.

Return value	A `Vector` with the mean values.

```
Matrix Multiply(Matrix m)
```

Multiplies `m` with the Matrix. The elements of the returned matrix are calculated as: $c_{ik} = a_{ij}b_{jk}$. If the two matrices do not have the same number of rows the multiplication is impossible and a `MatrixException` is thrown.

m	The `Matrix` to be multiplied. The matrix dimensions must match to succeed
Return value	The multiplied `Matrix`.

```
void Print()
```

Prints the `Matrix` to the System output. Used for debugging purposes.

```
Matrix Substract(Matrix m)
```

Substracts `m` from the `Matrix` and returns the result. `m` needs to have the same dimensions as the `Matrix` to succeed, otherwise a `MatrixException` is thrown.

m	The `Matrix` to be subtracted.
Return value	The subtracted `Matrix`.

```
void Write(string file)
```

Writes the `Matrix` to the specified file in binary format.

file	The file to write the `Matrix` to.

```
int columns
```

The Number of columns in the `Matrix`. It is directly exposed as integer because of performance reasons. Do not directly write to this field.

```
float[,] d
```

Gives you direct access to the `Matrix` values. It is a two-dimensional array with the size $rows \times columns$

```
int rows
```

The Number of rows in the `Matrix`. It is directly exposed as integer because of performance reasons. Do not directly write to this field.

Vector (← Mirage.Mir.Matrix)

A `Vector` represents a one-dimensional `Matrix`. In *Mirage* it is used to handle mean values correctly.

```
Vector(int rows)
```

The constructor creates a new `Vector` of the given size.

rows	Number of rows.
Return value	The new `Vector` object.

CovarianceMatrix (← System.Object)

The `CovarianceMatrix` class is a special class, it was created for memory and performance reasons. Since the covariance matrices in *Mirage* are all square matrices and symmetric, about half the values stored in a the covariance matrix repeat. The `CovarianceMatrix` class uses the knowledge and stores only the required values. This reduces the memory needed for about 50%.

```
CovarianceMatrix(int n)
```

The constructor creates a new `CovarianceMatrix` with the dimensions $n \times n$ as specified.

n	Number of columns/rows
Return value	The new `CovarianceMatrix`

```
CovarianceMatrix(Matrix m)
```

The constructor creates a new `CovarianceMatrix` using `m` as source. This can be used to transform the resulting Matrix from a `Matrix::Covariance()` call to a more memory efficient one. This only works for symmetric square matrices.

m	The `Matrix` to be converted into a more memory efficient one. `m` has to be symmetric and square.
Return value	The new `CovarianceMatrix`

```
CovarianceMatrix Add(CovarianceMatrix m)
```

Adds `m` to the `CovarianceMatrix` and returns the summed `CovarianceMatrix`. `m` needs to have the same dimensions as the `CovarianceMatrix` to succeed, otherwise a `MatrixException` is thrown.

m	The `CovarianceMatrix` to be added.
Return value	The summed `CovarianceMatrix`.

```
Matrix Multiply(CovarianceMatrix m)
```

Multiplies `m` with the `CovarianceMatrix`. The elements of the returned matrix are calculated as: $c_{ik} = a_{ij}b_{jk}$, taking the special memory alignment of the class into account. Since multiplying two symmetric marices does not yield to a symmetric matrix as a result, a full `Matrix` has to be returned. If the two covariance matrices do not have the same number of rows the multiplication is impossible and a `MatrixException` is thrown.

m	The `Matrix` to be multiplied. The matrix dimensions must match to succeed
Return value	The multiplied `Matrix`.

```
int dim
```

The Number of columns and rows in the square and symmetric `CovarianceMatrix`. It is directly exposed as integer because of performance reasons. Do not directly write to this field.

```
float[] d
```

Gives you direct access to the `CovarianceMatrix` values. The array has a special layout. To access the value a_{ij} use the formula $idx_{a_{ij}} = jd + i - \frac{(j+1)^2-(j+1)}{2}$, where $d = dim$ and i, j are the indices for the two dimensional matrix. Since the special-case CovarianceMatrix object stores only one half of the Matrix, the index values i, j need to be swapped if $j > i$ to retrieve the correct value.

A.3. Internal

Mfcc (← System.Object)

An object used for computing the Mel Frequency Cepstral Coefficients.

```
Mfcc(int winsize, int srate, int filters, int cc)
```

Initializes the class with the basic parameters for computing the MFCCs. The MFCC filter weights are precomputed in the constructor to speed up MFCC calculation.

winsize	The windowsize for the MFCCs.
srate	The samplingrate of subsequent data.
filters	The number of MFCC filters.
cc	The number of Cepstral Coefficients.

```
Matrix Apply(Matrix m)
```

Computes the MFCCs for the given `Matrix`. The `Matrix` is usually the result of an short time Fourier transformation (STFT), or the result of a `Mpg123FileReader::Read()` call.

m	The STFT for which the MFCCs should be computed.
Return value	The MFCCs for the input.

Mpg123FileReader (← System.Object)

MP3 file reading class using the Fft123 tool.

```
static Matrix Read(string fileIn)
```

This static method is used to read the given MP3 file. It returns the STFT representation of the selected MP3. It uses the Fft123 tool, which has to be in the PATH otherwise all calls to this method fail.

fileIn	The filename of the MP3 which should be read.
Return value	A `Matrix` with the STFT representation of the MP3. This `Matrix` can be used to calculate the MFCC values of the MP3 to further compute an `Scms` model of it.

A.4. Debugging

Dbg (← System.Object)

Debugging functions are available in this class.

```
static void WriteLine(String l)
```

Writes the given string to the Terminal and appends a newline character.

l	Debug String, which should be written to the Terminal.

```
static void Write(String l)
```

Writes the given string to the Terminal.

l	Debug String, which should be written to the Terminal.

Timer (← System.Object)

Used to measure execution time of processes. It can be used very easily. The total execution time in milliseconds is returned by `Stop()`, after calling Start() to start measuring.

```
Timer()
```

Default constructor, initializing the Timer.

```
void Start()
```

Starts the time measurement.

```
long Stop()
```

Stops the time measurement and returns the number of milliseconds which lay between the `Start()` and `Stop()` call.

Return value	Number of milliseconds since calling `Start()`.

Appendix B

B. Tracklist Small Testset

Alternative/Everlast - Black Jesus
Alternative/Garbage - Only Happy When It Rains
Alternative/Garbage - Supervixen
Alternative/Maxim Feat. Skin - Carmen Queasy
Alternative/Placebo - Every You Every Me
Alternative/Placebo - Slave To The Wage
Alternative/The Strokes - Someday
Blues/01 - Dave Brubeck Quartet - Blue Rondo A La Turk
Blues/02 - Dave Brubeck Quartet - Strange Meadow Lark
Blues/03 - Dave Brubeck Quartet - Take Five
Blues/04 - Dave Brubeck Quartet - Three To Get Ready
Blues/05 - Dave Brubeck Quartet - Kathy's Waltz
Blues/06 - Dave Brubeck Quartet - Everybody's Jumpin'
Blues/07 - Dave Brubeck Quartet - Pick Up Sticks
Blues/St Germain - Tourist Rose Rouge
Classic Orchestra/03 - - Nikolai Rimsky Korsakow - Das Maerchen vom Zarenn Saltan - Hummelflug
Classic Orchestra/04 - - Maurice Ravel - Bolero
Classic Orchestra/05 - - Richard Strauss - Salome op54 - Tanz der Sieben Schleier (Karajan)
Classic Orchestra/06 - - Modest Mussorgsky - Chowantschina - Vorspiel, Morgendaemmerung an der Moskwa
Classic Orchestra/07 - - Jacques Offenbach - Hoffmanns Erzaehlungen - Barcarole
Classic Orchestra/08 - - Peter I Tschaikowsky - Romeo und Julia - Liebesthema (Karajan)
Classic Piano/03 - Vladimir Horowitz - Piano Sonata in B flat major, K.281 - Rondeau- Allegro
Classic Piano/04 - Chopin - Etude, Op 25, No 7
Classic Piano/04 - Vladimir Horowitz - Piano Sonata in C major, K.330 - Allegro Moderato
Classic Piano/05 - Vladimir Horowitz - Piano Sonata in C major, K.330 - Andante Cantibile
Classic Piano/06 - Vladimir Horowitz - Piano Sonata in C major, K.330 - Allegretto
Classic Piano/07 - Chopin - Waltz, Op 69, No 1
Classic Piano/08 - Chopin - Andante spianato, Op 22
Classic Piano/ - Johann Sebastian Bach - Brandenburgisches Konzert Nr. 6, B-Dur, BWV 1051 - Adagio, ma non tanto
Classic Piano/ - Samuael Barber - Adagio fuer Streicher op. 11
Dance/Djs At Work - Someday (Vocal Edit)
Dance/Djs At Work - Time To Wonder
Dance/Jan Wayne Meets Lena - Total Eclipse Of The Heart
Dance/Kai Tracid - Tiefenrausch
Dance/Snap - Rhythm Is A Dancer 2003
Dance/Snap - Rythm Is A Dancer
Dance/Daddy Dj - Daddy Dj
Dance/Groove Coverage - God Is A Girl
Dance/Fragma - You Are Alive
Eurodance/666 - Paradoxx
Eurodance/Doki Doki - Too Fast For Love
Eurodance/Ephony - Dancing In The Rain
Eurodance/Flex - Spider
Eurodance/Magic Affair - Omen 3
Eurodance/Scooter - Maria (I Like it Loud)
Eurodance/La Bouche - Be My Lover
Eurodance/La Bouche - Another Night Another Dream
Eurodance/Eurythmics - Sweet Dreams
Happy Sound/Ace of Base - Beautiful Morning (Groove Radio Edit)
Happy Sound/Celine Dion - Im Alive
Happy Sound/Celine Dion - Thats The Way It Is
Happy Sound/Celion Dion - A New Day Has Come
Happy Sound/Vanessa Amorosi - Absolutely Everybody
Happy Sound/Vanessa Amorosi - Everytime I Close My Eyes
Hard Pop/Bon Jovi - Bad Medicine
Hard Pop/Bon Jovi - Everyday
Hard Pop/Bon Jovi - Its My Life
Hard Pop/Bon Jovi - Living On A Prayer
Hard Pop/Evanescene - Bring Me To Life
Hard Pop/Guano Apes - Open Your Eyes
Hard Pop/Guano Apes - No Speech
Hard Pop/The Killers - Somebody Told Me
Hip Hop/Dmx - Ruff Riders Anthem
Hip Hop/Dmx - Up In Here
Hip Hop/Dmx - Why Do Good Girls Like Bad Guys
Hip Hop/Mya - Case Of The Ex
Hip Hop/Mya - Free
Hip Hop/Nelly - Country Grammar
Hip Hop/Nelly - EII
Hip Hop/Blu Cantrell Feat. Sean Paul - Breathe (Remix)
Hip Hop/Die Fantastischen Vier - Troy
Hip Hop/Seeed - Dickes B
Hip Hop/Xzibit \& Dr Dre - Symphony In X Major (Explicit Version)
Hip Hop/Xzibit - 'X' (Explicit)
Mystera/Enya - Anywhere Is
Mystera/Enya - Book Of Days
Mystera/Enya - Exile
Mystera/Enya - La So adora
Mystera/Enya - Only Time
Mystera/Enya - Orinoco Flow (Sail Away)
Mystera/Enya - Storms In Africa
Mystera/Vangelis - Chariots Of Fire
Pop/Britney Spears - Crazy
Pop/Britney Spears - Lucky
Pop/Christina Aguilera - Genie In A Bottle
Pop/Emma Bunton - What Took You So Long
Pop/Phil Collins - Can't Stop Loving You
Pop/Texas - Summer Son
Punk Rock/Blink 182 - All The Small Things
Punk Rock/Bloodhound Gang - Along Comes Mary
Punk Rock/Die Aerzte - Westerland
Punk Rock/Die Ärzte - Schrei Nach Liebe
Punk Rock/Die Toten Hosen - Paradies
Punk Rock/Offspring - Self Esteem
Punk Rock/Offspring- Defy You
Punk Rock/Papa Roach - Last Resort
Punk Rock/Bad Religion - 21st Century Digital Boy
Rock/Beach Boys - Surfin Usa
Rock/Bruce Springsteen - Born In The Usa
Rock/Steppenwolf - Born To Be Wild
Rock/U2 - Sunday Bloody Sunday
Rockn Roll/Bill Haley \& The Comets - Rock Around The Clock
Rockn Roll/Bill Haley \& The Comets - See You Later Alligator
Rockn Roll/Little Eva - Locomotion
Rockn Roll/Manfred Mann - Do Wah Diddy Diddy
Rockn Roll/Queen - Dont Stop Me Now
Rockn Roll/Rubettes - Juke Box Jive
Rockn Roll/Grease - You're The One That I Want
Romantic Dinner/Elvis Costello - She
Romantic Dinner/Etta James - At Last
Romantic Dinner/Evita - Don't Cry For Me Argentina
Romantic Dinner/Hothouse Flowers - I Can See Clearly Now
Romantic Dinner/Tammy Wynette - Stand By Your Man
Romantic Dinner/Tony Bennett - I Left My Heart In San Francisco
Talk/Dorfer u. Dueringer - Auf dem Wachzimmer
Talk/Maurer u. Scheuba - Voelkische Comedy
Talk/Maurer u. Scheuba - Zwischenspiel
Talk/Otto Waalkes - Der Bergdoktor
Talk/Otto Waalkes - Die Gruene Hoelle
Talk/Otto Waalkes - Wodka Zischer}

C. References

[AHH$^+$01] E. Allamanche, J. Herre, O. Hellmuth, B. Froeba, T. Kastner, and M. Cremer. Content-based Identification of Audio Material Using MPEG-7 Low Level Description. *Proceedings of the International Symposium of Music Information Retrieval*, 2001.

[ANR74] N. Ahmed, T. Natarajan, and KR Rao. Discrete cosine transform. *IEEE Trans. Comput*, 23(1):90–93, 1974.

[AP00] J.J. Aucouturier and F. Pachet. Finding songs that sound the same. *log*, 2:1, 2000.

[AP02a] Jean-Julien Aucouturier and François Pachet. Music similarity measures: What's the use? In *ISMIR*, 2002.

[AP02b] J.J. Aucouturier and F. Pachet. Scaling up Music Playlist Generation. 2002.

[AP04] J.-J. Aucouturier and F. Pachet. Improving timbre similarity: How high is the sky? *Journal of Negative Results in Speech and Audio Sciences*, 1(1), 2004.

[BH03] S. Baumann and O. Hummel. Using Cultural Metadata for Artist Recommendation. *Proc. of Wedel-Music*, 2003.

[Bis95] C.M. Bishop. *Neural Networks for Pattern Recognition.* Clarendon Pr, 1995.

[BR01] V. Britanak and KR Rao. An Efficient Implementation of the Forward and Inverse MDCT in MPEG Audio Coding. *IEEE SIGNAL PROCESSING LETTERS*, 8(2), 2001.

[Bra99] K. Brandenburg. MP3 and AAC explained. *AES 17th International Conference on High-Quality Audio Coding*, 1999.

[Bri74] E.O. Brigham. The fast Fourier Transform. *Englewood Cliffs, NJ: Prentice-Hall, 1974*, 1974.

[CST00] N. Cristianini and J. Shawe-Taylor. An introduction to Support Vector Machines. 2000.

[Ell06] T. Ellis. Beat Tracking with Dynamic Programming. 2006.

[FPW05a] A. Flexer, E. Pampalk, and G. Widmer. Hidden Markov models for spectral similarity of songs. *Proceedings of the 8th International Conference on Digital Audio Effects*, 2005.

[FPW05b] A. Flexer, E. Pampalk, and G. Widmer. Novelty detection based on spectral similarity of songs. *Proc. of the 6 thInt. Symposium on Music Information Retrieval, London, UK*, 2005.

[GKD$^+$06] F. Gouyon, A. Klapuri, S. Dixon, M. Alonso, G. Tzanetakis, C. Uhle, and P. Cano. An experimental comparison of audio tempo induction algorithms. *IEEE Transactions on Speech and Audio Processing*, 14(5), 2006.

[Huf52] D.A. Huffman. A method for the construction of minimum redundancy codes. *Proc. IRE*, 40(9):1098–1101, 1952.

[KJ56] J.B. Kruskal Jr. On the Shortest Spanning Subtree of a Graph and the Traveling Salesman Problem. *Proceedings of the American Mathematical Society*, 7(1):48–50, 1956.

[KPW04] Peter Knees, Elias Pampalk, and Gerhard Widmer. Artist classification with web-based data. In *Proceedings of 5th International Conference on Music Information Retrieval (ISMIR'04)*, pages 517–524, Barcelona, Spain, October 2004.

[KSS97] H. Kautz, B. Selman, and M. Shah. Referral Web: combining social networks and collaborative filtering. *Communications of the ACM*, 40(3):63–65, 1997.

[LKM04] B. Logan, A. Kositsky, and P. Moreno. Semantic analysis of song lyrics. *Multimedia and Expo, 2004. ICME'04. 2004 IEEE International Conference on*, 2, 2004.

[Log00] Beth Logan. Mel frequency cepstral coefficients for music modeling. In *Proceedings of the First International Symposium on Music Information Retrieval (ISMIR)*, Plymouth, Massachusetts, oct 2000.

[Log02] B. Logan. Content-Based Playlist Generation: Exploratory Experiments. 2002.

[LR06] T. Lidy and A. Rauber. MIREX 2006 Computing Statistical Spectrum Descriptors for Audio Music Similarity and Retrieval. 2006.

[LS01] B. Logan and A. Salomon. A music similarity function based on signal analysis. *Multimedia and Expo, 2001. ICME 2001. IEEE International Conference on*, pages 745–748, 2001.

[ME05] Michael Mandel and Dan Ellis. Song-level features and support vector machines for music classification. In *ISMIR*, pages 594–599, 2005.

[ML93] N. Merhav and C.H. Lee. On the asymptotic statistical behavior of empirical cepstralcoefficients. *Signal Processing, IEEE Transactions on [see also Acoustics, Speech, and Signal Processing, IEEE Transactions on]*, 41(5):1990–1993, 1993.

[MPE] MDS MPEG. Group, Text of ISO/IEC 15938-5 FDIS Information Technology - Multimedia Content Decsription Interface - Part 5 Multimedia Description Schemes.

[MPE92] MPEG. International Standard IS 11172-3, ISO/IEC JTC1/SC29 WG11, Coding of moving pictures and associated audio for digital storage media at up to 1.5 Mbit/s, part 3: Audio. *Motion Pictures Expert Group*, 1992.

[NDR05] Robert Neumayer, Michael Dittenbach, and Andreas Rauber. Playsom and pocketsomplayer: Alternative interfaces to large music collections. In *Proceedings of the Sixth International Conference on Music Information Retrieval (ISMIR 2005)*, pages 618–623, London, UK, September 11-15 2005.

[OS89] A.V. Oppenheim and R.W. Schafer. *Discrete-time signal processing*. Prentice-Hall, Inc. Upper Saddle River, NJ, USA, 1989.

[P$^+$93] J.W. Picone et al. Signal modeling techniques in speech recognition. *Proceedings of the IEEE*, 81(9):1215–1247, 1993.

[Pam04] E. Pampalk. A Matlab Toolbox to compute music similarity from audio. *Proceedings of the 5th International Conference on Music Information Retrieval (ISMIR'04), Universitat Pompeu Fabra, Barcelona, Spain*, pages 254–257, 2004.

[Pam05] Elias Pampalk. Speeding up music similarity. In *2nd Annual Music Information Retrieval Evaluation Exchange*, September 2005.

[Pam06a] E. Pampalk. Audio-Based Music Similarity and Retrieval: Combining a Spectral Similarity Model with Information Extracted from Fluctuation Patterns. 2006.

[Pam06b] E. Pampalk. Computational Models of Music Similarity and their Application in Music Information Retrieval. *Docteral dissertation, Vienna University of Technology, Austria, March*, 2006.

[PDW04] E. Pampalk, S. Dixon, and G. Widmer. Exploring Music Collections by Browsing Different Views. *Computer Music Journal*, 28(2):49–62, 2004.

[Pen01] WD Penny. KL-Divergences of Normal, Gamma, Dirichlet and Wishart Densities. Technical report, Technical Report, Wellcome Dpt of Cognitive Neurology, University College London, 2001.

[PM00] D. Pelleg and A. Moore. X-means: Extending K-means with efficient estimation of the number of clusters. *Proceedings of the 17th International Conf. on Machine Learning*, pages 727–734, 2000.

[Poh06] T. Pohle. Post Processing Music Similarity Computations. 2006.

[PPW05a] E. Pampalk, T. Pohle, and G. Widmer. Dynamic Playlist Generation Based on Skipping Behaviour. *Proceedings of ISMIR 2005 Sixth International Conference on Music Information Retrieval, September*, 2005.

[PPW05b] T. Pohle, E. Pampalk, and G. Widmer. Generating similarity-based playlists using traveling salesman algorithms. *Proceedings of the 8th International Conference on Digital Audio Effects (DAFx-05)*, pages 20–22, 2005.

[PRM02] E. Pampalk, A. Rauber, and D. Merkl. Content-based organization and visualization of music archives. In *Proceedings of ACM Multimedia 2002*, pages 570–579, Juan-les-Pins, France, December 1-6 2002. ACM. `http://www.ifs.tuwien.ac.at/ifs/research/publications.html`.

[PV01] Silvia Pfeiffer and Thomas Vincent. Formalisation of MPEG-1 compressed domain audio features. Technical Report

01/196, CSIRO Mathematical and Information Sciences, December 2001.

[Rab89] LR Rabiner. A tutorial on hidden Markov models and selected applications inspeech recognition. *Proceedings of the IEEE*, 77(2):257–286, 1989.

[Rot83] J. Rothweiler. Polyphase quadrature filters–A new subband coding technique. *Acoustics, Speech, and Signal Processing, IEEE International Conference on ICASSP'83.*, 8, 1983.

[RTG00] Y. Rubner, C. Tomasi, and L.J. Guibas. The Earth Mover's Distance as a Metric for Image Retrieval. *International Journal of Computer Vision*, 40(2):99–121, 2000.

[SIA99] JO Smith III and JS Abel. Bark and ERB bilinear transforms. *Speech and Audio Processing, IEEE Transactions on*, 7(6):697–708, 1999.

[Spe02] E.C.L. Specification. Technical Report Standard ECMA-334. *European Computer Manufacturers Association, December*, 2002.

[SXWK04] X. Shao, C. Xu, Y. Wang, and M.S. Kankanhalli. Automatic music summarization in compressed domain. *IEEE International Conf. on Acoustics, Speech, and Signal Processing (ICASSP04)*, 2004.

[TC00] George Tzanetakis and Perry Cook. Sound analysis using MPEG compressed audio. Istanbul, Turkey, 2000.

[TC02] G. Tzanetakis and P. Cook. Musical Genre Classification of Audio Signals. *IEEE TRANSACTIONS ON SPEECH AND AUDIO PROCESSING*, 10(5):293, 2002.

[WV01] Y. Wang and M. Vilermo. A compressed domain beat detector using MP3 audio bitstreams. *Proceedings of the ninth ACM international conference on Multimedia*, pages 194–202, 2001.

[Xip04] Xiph.org. Vorbis 1 Specification, 1.1. 2004.

[YWB93] SJ Young, PC Woodland, and WJ Byrne. HTK: Hidden Markov Toolkit V1. 5, 1993.

[ZY02] G.L. Zick and L. Yapp. Speech recognition of MPEG/audio encoded files. *Acoustical Society of America Journal*, 112(6):2520–2520, 2002.