QUAD-BASED AUDIO FINGERPRINTING ROBUST TO TIME AND FREQUENCY SCALING

Reinhard Sonnleitner
Department of Computational Perception,
Johannes Kepler University
Linz, Austria
reinhard.sonnleitner@jku.at

Gerhard Widmer
Department of Computational Perception,
Johannes Kepler University
Linz, Austria
gerhard.widmer@jku.at

ABSTRACT

We propose a new audio fingerprinting method that adapts findings from the field of blind astrometry to define simple, efficiently representable characteristic feature combinations called *quads*. Based on these, an audio identification algorithm is described that is robust to large amounts of noise and speed, tempo and pitch-shifting distortions. In addition to reliably identifying audio queries that are modified in this way, it also accurately estimates the scaling factors of the applied time/frequency distortions. We experimentally evaluate the performance of the method for a diverse set of noise, speed and tempo modifications, and identify a number of advantages of the new method over a recently published distortion-invariant audio copy detection algorithm.

1. INTRODUCTION

The typical use of an audio fingerprinting system is to precisely identify a piece of audio from a large collection, given a short query. This is typically done by extracting highly discriminative content features, a "fingerprint", from the collection of audio files as well as the query piece, and subsequently comparing these features. The focus in fingerprinting systems is on being able to discriminate between non-identical pieces of audio, even if they sound very similar to a listener, as in the case of different versions of the same song (e.g., "Album Version" vs. "Radio Version").

There are numerous application scenarios for audio fingerprinting. A well established one is to enable curious users to learn the name of the song they are currently listening to [1]. Other applications, mainly used by industry, include the large-scale tasks of media monitoring and audio copy/plagiarism detection [2].

Depending on the application, audio fingerprinting systems should be robust to different kinds of distortions of query audio material. The minimum requirement would seem to be robustness to various types of lossy audio compression and a certain amount of noise. Application domains such as DJ set track identification, media monitoring, audio copy detection and plagiarism detection, pose additional requirements. There, it is also necessary to recognize audio material that was modified in tempo and/or in pitch, and perhaps also tolerate nonlinear noise encountered when a DJ blends two songs in order to achieve a perceptually smooth transition. In the latter case, the time and frequency scale changes may not even be constant over the duration of a song. All these are significant challenges to automatic audio identification systems.

In this work we propose an efficient audio fingerprinting method that meets the above robustness requirements. It is not only robust to noise and audio quality degradation, but also to large amounts of speed, tempo or frequency scaling.[1] In addition, it can accurately estimate the scaling factors of applied time/frequency distortions. The key technique that makes this possible was found by researchers working on blind astrometry, who use a simple and fast geometric hashing approach to solve a generalization of the "lost in space" problem [3]. The specific task is to determine the pointing, scale and orientation of an astronomical image (a picture of a part of the sky), without any additional information beyond the pixel values of the image. We adapt this method to the needs of audio fingerprinting and based on this, develop an extremely efficient and robust audio identification algorithm. The central components in this are compact hash representations of audio that are invariant to translation and scaling, and thereby overcome the inherent robustness limitations of systems that depend on equal relative distances of reference and query features to find matches, such as the well-known Shazam algorithm [1]. More precisely, our algorithm uses a compact four-dimensional, continuous hash representation of quadruples of points, henceforth referred to as "quads". The quad descriptor [3] has also recently been adopted in the field of computer vision, for the task of accurate alignment of video streams [4, 5].

The system we propose can be used for DJ set monitoring and original track identification, audio copy detection, audio alignment, as well as other tasks that demand robustness to certain levels of noise and scale changes.

The paper is organized as follows. Section 2 discusses relevant related work and, in particular, focuses on a very recently published state-of-the art method [2] that will act as our main reference here. Section 3 gives a brief overview of the main points of our new method, in order to set the context for the precise method description, which comes in two parts: Section 4 describes the process of constructing audio fingerprints – the extraction of special features from audio, how to obtain hash representations from these features, and how to store these in special data structures for efficient retrieval. The actual identification method, i.e. the process of matching query audio with reference data by using the extracted features and their hash representations, is then explained in Section 5. Section 6 systematically evaluates the performance of our

[1] To clarify our terminology: if both scales are changed proportionally, we call this a change in *"speed"*: the song is played faster and at the same time at a higher pitch. This can be achieved by simply changing the rotational speed of the turntable, or by modifying the sampling rate of a digital media player – while keeping the sampling rate of the audio encoding unchanged. Changing the time scale only will be referred to as a *"tempo"* change: here, the audio is sped up or slowed down without observable changes in pitch. Vice versa, if only the frequency scale is modified, this will be called *pitch shifting*.

method in two experiments. The first experiment considers song identification on a database consisting of one thousand full length songs of different musical genres. For the second experiment we extend the database to 20,000 full length songs.

2. RELATED WORK

Numerous fingerprinting algorithms have been described in the literature (e.g., [1, 6, 7, 8, 9]), not all of which are applicable to the tasks described in the introduction. Some algorithms extract global fingerprints for entire songs, and are not suitable for our given tasks because identifying snippets or excerpts of songs in the reference database requires local fingerprints. In [9], a fingerprinting approach is proposed that computes the scale invariant feature transform (SIFT) [10] on the audio spectrograms with logarithmically scaled frequencies. While good results are shown, SIFT is not invariant to scale changes if only one of the two dimensions is scaled. Therefore, the algorithm proposed in [9] is not robust to what we call tempo changes.

A tempo- and speed-invariant audio fingerprinting algorithm has recently been proposed in [11]. By using pitch class histograms as fingerprint features, it tends to compute similar fingerprints for similar content, which is an interesting property on its own. The algorithm has a certain robustness to tempo and speed changes, but its performance degrades considerably for noisy queries. Also, the performance degrades when the query audio is shorter than the specific references, which is why the method is not well suited for the tasks we described above.

A very recent publication proposing a fingerprinting algorithm for audio copy detection, that also meets the demands of robustness against speedup, tempo changes and pitch-shifting of query audio is [2]. The work reports near perfect percentages of correct song association. The described method performs feature extraction on a two-dimensional time-chroma representation of the audio. First a set of candidate feature points is selected, which are then purified by extracting and comparing up to 30 two-dimensional image patches of different width, centered around the candidate feature point. A candidate point is selected as feature point if most of the (up to 30) extracted image patches fulfill a similarity criterion. This is determined via k-means clustering, which assigns the extracted patches to a number of c classes. Similarity is calculated by computing low frequency discrete cosine transformation (DCT) coefficients which represent the actual similarity metric. The proposed method performs feature point selection for an average of 20 candidate points per second of audio, of which approximately 40% pass the similarity constraints and are used for fingerprint computation. The actual fingerprints are generated from a number of low frequency DCT coefficients of the extracted image patches, and are scaled and translated to result in vectors of zero mean and unit variance. According to the explanation given in the work, such a fingerprint should result in a vector of 143 floating point values. Fingerprint matching is done by nearest neighbor lookups, with distance defined as the angle of two fingerprint vectors.

We will take this as our reference method in the present paper, because it is the latest publication on this topic, and it reports extremely high robustness and performance results for a large range of tempo and speed modifications (though based on experiments with a rather small reference database – see Section 6 below).

3. METHOD OVERVIEW

The basic idea of our proposed method is to extract spectral peaks from the two-dimensional time-frequency representation of reference audio material, then group quadruples of peaks into quads, and create a compact translation- and scale-invariant hash for each quad (a single hash is a point in a four-dimensional vector space). Quads and their corresponding hashes are stored in different data structures, i.e. quads are appended to a global index, and an inverse index is created to assign the corresponding audio file-id to its quad indices. The continuous hashes are stored, together with the index of their quad, in a spatial data structure, such that the index that is associated with the hash corresponds to the index of the quad that forms the hash.

For querying we extract quads and their hashes from the query audio excerpt. For each query hash we perform a range search in the spatial data structure and collect the indices of search results, which in turn give the indices of matching reference quads in the global index. The time and frequency scaling factor can be found by comparing a query quad to its matching reference quad. To predict the match file ID for a query snippet, we adapt the histogram method from [1].

4. FEATURE EXTRACTION

In this section we describe the extraction of audio features to be used for audio identification, how to obtain hashable representations from these features, and how to finally store these for efficient retrieval. The same feature extraction process is applied to the *reference recordings* that are used to build the fingerprint database, and the *query audio* that is to be identified in the recognition phase.

To begin with, all audio files are downmixed to one-channel monaural representations and processed with a sampling rate of 16 kHz. We compute the STFT magnitude spectrogram using a Hann-window of size 1024 samples (64 ms) and a hopsize of 128 samples (8 ms), discarding the phases.

4.1. Constructing Quads

The fingerprinting algorithm works on translation- and scale-invariant hashes of combinations of spectral peaks. Spectral peaks are local maxima in an STFT magnitude spectrogram, and identified by their coordinates in the spectrogram. Since the notion of a peak P as a point in 2D spectrogram space will be used extensively in the following, let us formally introduce the notation:

$$P = (P_x, P_y)$$

where P_x is the peak's time position (STFT frame index), and P_y is the peak's frequency (index of STFT frequency bin).

The extraction of spectral peaks is implemented via a pair of two-dimensional filters, a max filter and a min filter, where the neighbourhood size is given by the filter window size. We use a max filter to find the coordinates of spectral peak candidates in the spectrogram, and use a min filter with a smaller window size to reject peaks that were extracted from uniform regions in the spectrogram, e.g., silence. In the following we explain how quads are created from spectral peaks, and how compact hash values are computed from quads.

To create translation- and scale-invariant hashes from spectral peaks, we first have to group peaks into *quads* [3]. A quad consists of four spectral peaks A, B, C, D, where we define A to be the

root point of the quad, which is the peak with the smallest frame index (i.e. the first of the four peaks in time) and B is the most distant point in time from A (thus C, D lie somewhere between the STFT frames of A, B). The quad is *valid* if $B > A$ and C, D lie within the axis-parallel rectangle defined by A, B:

$$A_x < B_x \tag{1}$$
$$A_y < B_y \tag{2}$$
$$A_x < C_x, D_x \leq B_x \tag{3}$$
$$A_y < C_y, D_y \leq B_y \tag{4}$$

At the top level, the quad grouping process proceeds through an audio file from left to right, trying each spectral peak as a potential root point A of a set of quads, with the goal of creating up to a number of q quads for each peak.

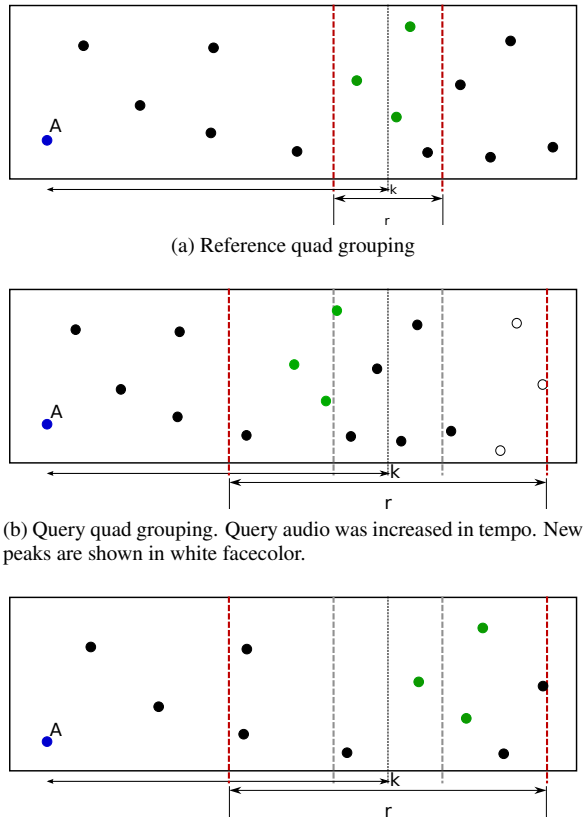
For a given root point A, the process of constructing up to q quads by selecting appropriate sets of B, C, D points is as follows.[2] We construct a region of width r, spanning r STFT frames, such that the region is centered k STFT-frames from A and A is outside of the region (earlier in time, i.e., the region is located to the right of A), as shown in Figure 1. We then sort the peaks that are contained in the region, by time. We let $t = 0$ and pick the first n peaks $p_t, p_{t+1}, \ldots p_{t+n-1}$ in the region and try all $\binom{n}{3}$ combinations of 3 peak indices in order to construct a valid quad with root point A and the points from the current combination. If a valid quad can be constructed we append it to a list of quads and proceed until q quads are created. If no valid quad could be constructed, we increase t by one and try again until there are no more peaks in the region.

The total number of resulting valid quads for a given root point A depends not only on the parameter values, but is fundamentally dependent on the specific layout of spectral peaks, and thus on the signal itself. As already mentioned, for creating the reference database we want to create a small number of quads. We therefore choose a small n and a region of small width r. For queries we create an extensive set of quads by choosing a larger n, $r_{\text{query}} \gg r_{\text{ref}}$, and $q_{\text{query}} \gg q_{\text{ref}}$. k is the same in both cases.

The reason for different parameterization for query quad construction is as follows: When the time scale of a query audio is modified, this affects not only the density of peaks in the given audio snippet, but also their relative positions. An example is given in Figure 1, which shows the grouping for a quad for a given root point A. In 1a a reference quad is created for a region of width r that is centered k frames from A. The analogous example for grouping a query quad for the same audio, but increased in tempo, or decreased in tempo, is given in 1b and 1c, respectively. We see that the green points, which are points B, C, D for the reference quad, may happen to move outside of the grouping region of width r if the time scale of the audio is modified. By choosing a larger region width r and a larger number q of quads that may be created for a root point A, we can ensure to obtain a quad that corresponds to the reference quad.

Note that when we consider audio queries of a fixed, limited duration d (e.g., $15sec$), there is an important difference between

[2] We will parametrize this process differently, depending on whether we compute quads for the reference database, or for a piece of query audio. For reference database creation, we choose parameters in such a way that we only create a small number of reference quads to keep the resulting reference database as small as possible. For a query snippet, we will choose parameters to create a large amount of quads. The explanation for this will be given later in this section.



(a) Reference quad grouping

(b) Query quad grouping. Query audio was increased in tempo. New peaks are shown in white facecolor.

(c) Query quad grouping. Query audio was decreased in tempo.

Figure 1: *Reference quad grouping (1a) and query quad grouping with increased tempo (1b), and decreased tempo (1c).*

increased speed/tempo and decreased speed/tempo. Increasing the tempo of the query audio excerpt relative to the reference leads to a higher density of relevant audio content; all the content that was used during the phase of reference quad creation is also present when creating the quads for the query. However, decreasing the tempo of the query, i.e., stretching the time scale, may cause some of the relevant spectral peaks to fall out of the $15sec$ (i.e., not be part of the query any more), so some important quads do not emerge in the query. This problem arises when tempo or speed are decreased by large amounts. This difference in increasing vs. decreasing the time scale is actually reflected in the evaluation results (see Section 6). To summarize, if the same parameters are used for both reference and query quad grouping, and the time scale changes, it is very likely that no matching quads will be found in subsequent queries.

4.2. From Quads to Translation- and Scale-invariant Hashes

We now have created quads from spectral peaks in audio, but these quads are not the actual summarizing representation that we later use to find match candidates between a query audio and the reference database. That representation should be translation- and scale-invariant, and quickly retrievable. To achieve this, we compute *translation- and scale-invariant hashes* from the quads. For a given quad (A, B, C, D), the constellation of spectral peaks is translated to the origin and normalized to the unit square, resulting

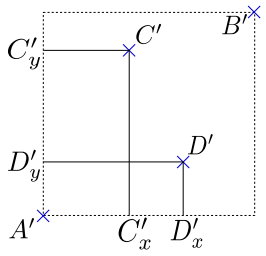


Figure 2: *Example of a quad hash computed from an arbitrary quad* A, B, C, D.

in the four points A', B', C', D' such that $A' = (0, 0)$ and $B' = (1, 1)$, as shown in Figure 2. The actual continuous hash of the quad is now given by C', D', and is stored as a four-dimensional point (C'_x, C'_y, D'_x, D'_y) in a spatial data structure. Essentially, C', D' are the relative distances of C, D to A, B in time and frequency, respectively. Thus, the hash C', D' is not only translation invariant (A' is always $(0, 0)$), but also scale invariant. A feature extraction example showing spectral peaks and resulting quads is shown in Figure 3

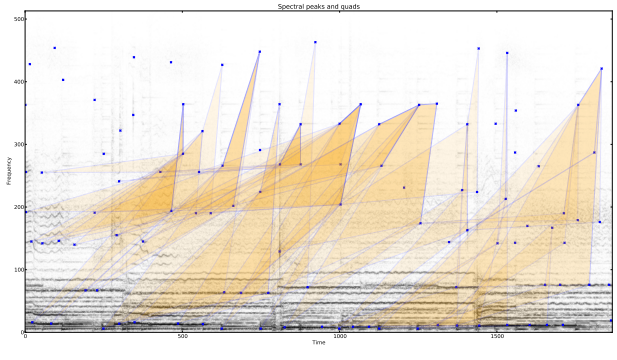


Figure 3: *Extracted spectral peaks and grouped quads on a* 15 *seconds excerpt of "Radiohead - Exit Music (For a Film)".*

4.3. Fingerprints: Storing Hashes for Efficient Recognition

Once peaks, quads and their hashes are computed, we store these in data structures that allow for efficient selection of match candidates and subsequent verification of match hypotheses from query audio. The reference data consist of four data structures:

- quadDB: A file that contains all reference quads (the original, *unnormalized* ones).
- fidindex: An index file that stores file-id, quad index range (into quadDB) and filename for each reference audio file.
- reftree: A spatial data structure containing all reference quad hashes.
- refpeaktrees: Two dimensional search trees for the spectral peaks that are extracted from reference audio files.

The quadDB is a binary file that stores the sequence of quads for all reference files, and the fidindex is an index file which maps each reference file to a unique file-id and also stores the index range (i.e. startindex, number of quads in quadDB) for the sequence of quads that was extracted from the reference files. For the spatial data structure (reftree) we use an R-Tree [12] with an R* split-heuristic that stores all quad hashes, together with their positional index in the quadDB. The R-Tree enables us to add songs the reference database without the need of rebuilding the tree in order to maintain high query performance. In addition, the R-Tree is well suited for large out-of-memory databases.

The refpeaktrees are used for the verification of match candidates, which will be explained later.

4.4. Chosen Parameter Values

The specific set of parameter values that we chose for our implementation and that are used in the evaluation in Section 6, is as follows: The extraction of spectral peaks is performed with a max-filter width of 91 STFT-frames, and a filter height of 65 frequency bins. The min-filter, used to reject maxima that resulted from uniform magnitude regions, has a width of 3 STFT-frames and a height of 3 frequency bins. For reference quad grouping we choose the center of the grouping window k to be four seconds from each root point A. The width r of this region window for reference quad extraction is two seconds. We group $q = 2$ quads for each root point A along with a group size of $n = 5$. This results in an average number of roughly 8.7 quads per second of audio. For query quad extraction we choose the same k of four seconds, and a large grouping window width r that spans 7.9 seconds. A number of up to $q = 500$ quads are extracted from a group size of $n = 8$.

5. RECOGNITION ALGORITHM

The method for identifying the correct reference recording, given a query audio excerpt, consists of several stages: the selection of match candidates, a filtering stage in which we try to discard false positive candidates, and a verification step adapted from the findings in [3]. After the verification stage we efficiently estimate a match sequence with the histogram binning approach that is used in algorithm [1]. In the following the selection of match candidates is explained.

5.1. Match Candidate Selection and Filtering

For each quad hash that was extracted from a query audio, an epsilon search in the reftree is performed. This lookup returns a set of raw match candidate indices: the indices of those quads in the quadDB whose quad-hashes are similar (identical up to epsilon: $B^q_x - \epsilon \leq B^r_x \leq B^q_x + \epsilon$ etc.) to the query quad-hashes. We call this the set of raw candidates, as it will most likely be a mixture of true positives and a (large) number of false positive matches. The raw candidates are used to obtain estimates of the time/frequency scale modification of the query audio, by looking at the different orientation of the original (non-normalized) quads corresponding to the query (q) and reference (r) hash, giving us the scaling factors for time and frequency:

$$s_{time} = (B^q_x - A^q_x)/(B^r_x - A^r_x) \quad (5)$$

$$s_{freq} = (B^q_y - A^q_y)/(B^r_y - A^r_y) \quad (6)$$

It makes sense to parametrize the system with scale tolerance bounds as, depending on the application, one might not be interested in trying to identify audio that is played at, e.g., half the speed or

double the tempo, or has undergone extreme pitch-shifting modifications. Such constrained tolerances considerably speed up the subsequent hypothesis testing by rejecting raw match candidates that lie outside the specified bounds.

Instead of directly starting with hypothesis tests on the raw candidate set we first apply filters in order to clean up the match candidates. This filtering process aims at discarding false positive matches while keeping a large number of true positives. In addition to the previously mentioned scale tolerances we perform a spectral coherence check, similar to the spatio-temporal coherence check described in [5]. Here we reject match candidate quads whose root point A is far away in the frequency domain compared to root point A of the query quad.

We now consult the fidindex to sort the remaining match candidates by file-id, and enter the verification step (Section 5.2) – those candidates that pass the following step are considered true matches and are passed to the match-sequence estimation.

5.2. Match Verification and Sequence Estimation

Match verification is performed once all match candidates for all query quads are collected and filtered as described above. Most likely, the remaining match candidates correspond to a large number of file-ids that are referenced in the database. Since our goal is to identify the correct file-id, we perform this stage of match candidate verification on a per-file-id basis. To do this we consult the fidindex file (cf. Section 4.3) and look up the file-ids for all match candidates, and sort the match candidates by file-id.

Verification is based on the insight that spectral peaks that are nearby a reference quad in the reference audio, should also be present in the query audio [3]. Naturally, depending on the audio compression, the amount of noise or other distortions, there might be a larger or smaller number of nearby peaks in the query. We define the nearby peaks as the set of N peaks closest to the match candidate's root point A (for some fixed N), and retrieve those by performing a k-nearest-neighbor search in the refpeak-trees (cf. Section 4.3) for the given file-id. We define a threshold t_{min}, the minimal number of nearby peaks that have to be present in the query in order to consider the candidate an actual match. Note that in order to find relevant nearby peaks in the query, we have to align the query- and reference-peaks by transforming the query peak locations according to the previously estimated time/frequency scale (cf. Section 5.1). The candidates that pass the verification step are considered true matches, and they are annotated with the number $v \leq N$ of correctly aligned spectral peaks, and the scale transformation estimates. This number v will be used for an optimization described below.

After match candidates for a given file-id are verified, we try to find a sequence of matches for this file-id by processing the matches with a histogram method similar to the one used in the Shazam algorithm [1], with the difference that the query time (the time value of root point A of each query quad in the sequence) is scaled according to the estimated time scale factor. Finally, the file-id for the largest histogram bin (the longest match sequence) is returned, together with the match position that is given by the minimal time value of the points in the histogram bin. We now know the reference file that identifies the query audio, the position of the query audio in the reference track, and the associated time/frequency scale modification estimates. Note that the reported scale transformation estimates are expected to be quite accurate, because with these estimates, for each "surviving" match candidate at least t_{min} nearby spectral query peaks could be correctly aligned to corresponding reference peaks during the verification phase. A lookup in the fidindex now gives us the filename of the reference audio as well as any kind of optional meta data.

To speed up the verification process, we define a threshold for the number of correctly aligned nearby peaks $t_{\text{v}} > t_{\text{min}}$. When the v value of a match reaches or exceeds this threshold, we allow a so-called "early exit" for this file-id. Once all match candidates of an early exit file-id are verified, we directly enter the match sequence estimation for this file-id, without subsequent verification of any other file-id.

5.3. Runtime and Data Size Considerations

Our system operates on a number of data structures (cf. Section 4.3) that together constitute what we call the *reference database*; the largest components are the reftree and the refpeak trees.

The quadDB linearly stores binary records of quads. A quad consists of four two-dimensional discrete points (coordinates in the STFT spectrogram) and can be represented and stored as $8 * 32$bit integers, which amounts to 32 byte per quad. It is not necessary to keep this file in-memory, as the proposed method is designed to operate on big out-of-memory reference data.

There exists exactly one quad hash per quad. A quad hash is a continuous four-dimensional point that is stored as an array of four float32 values by the reftree. The actual number of quads in the quadDB depends on the filter size parameters of the spectral peak extraction and the quad grouping parameters. For an example reference database consisting of 20,000 full length songs we choose the parameters such that we create an average of approximately 8.74 quads per second of audio, with a median of ≈ 8.68 and a standard deviation of $\sigma \approx 1.19$. The histogram of the number of quads per second is shown in Figure 4. This specific database consists of $\approx 4.29 * 10^7$ quads, or roughly 1.3GB. The reftree, a four-dimensional R-Tree, consumes approximately 4.8GB. To speed up the verification process we also store two dimensional trees of spectral peaks for each file-id, which consume roughly 3GB for $20,000$ songs. We currently store the fidindex file as text, along with some meta data. In this example the size of the fidindex amounts to 2.3MB.

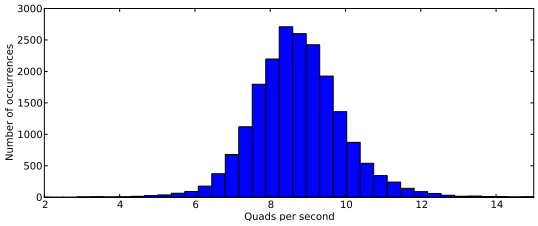


Figure 4: *Histogram of the average number of quads per second for all files in a reference database of 20,000 songs.*

Naturally, depending on application scenarios and hardware constraints, it is possible to trade runtime for storage space and vice versa. If minimal space consumption is of priority, one can pack the binary quad records of the quadDB to 16 bytes by exploiting the limited number of STFT frequency bins (i.e. 512), and storing the time values of points B, C, D as offsets from point A. This saves 50% per quad, reducing the size of the quadDB file to ≈ 650MB.

Regarding the runtime, using our unoptimized pure Python implementation of the method, feature extraction and quad creation for the database of $20,000$ songs took approximately $24h$ utilizing seven out of eight logical cores of an Intel Core i7 860 (2.8GHz) Processor.

The runtime for a query is made up of audio decoding, feature extraction, querying the database and filtering the results, match candidate verification and match sequence binning. For a query on a 15 seconds audio excerpt against a reference database of 1000 songs, this takes approximately 4 to 12 seconds. Here, half of the time is taken by the preprocessing (decoding, quad extraction).

Querying a larger database of $20,000$ songs takes considerably (though, of course, not proportionally) longer. The main reason is the higher number of match candidates that have to be processed. The same query excerpt as used above is processed in approximately 14 to 60 seconds. Here, at least half of the time is consumed by the reftree range queries. Again, this is based on an unoptimized, experimental Python implementation; there is ample room for improvement. Section 6.3 gives more detail.

6. EVALUATION

We systematically evaluate the performance of the system for different speed, tempo and noise modifications of 15 seconds query audio snippets. The reference database for the first experiment is constructed from $N = 1000$ full length songs in .mp3 format. To create test queries, we randomly choose 100 reference songs and subject these to different speed, tempo, and noise level modifications. We then randomly select a starting position for each selected song, and cut out 15 seconds from the audio, such that we end up with 100 $15sec$ audio queries. The evaluation considers speed and tempo ranges from 70% to 130% in steps of 5%. To evaluate the noise robustness of our system we mix each query snippet with white noise to create noisy audio in SNR level ranges from 0 dB to $+50$ dB in steps of 5 dB. Furthermore, we create all query audio snippets from .mp3 encoded data, and encode the modified snippets in the Ogg Vorbis format [13], using the default compression rate (cr $= 3$). We do this to show that the system is also robust to effects that results from a different lossy audio encoding. All modifications are realized with the free SoX audio toolkit [14]. The following terms are used in defining our performance measures: tp (*true positives*) is the number of cases in which the correct reference is identified from the query. fp (*false positives*) is the number of cases in which the system predicts the wrong reference. fn (*false negatives*) is the number of cases in which the system fails to return a reference id at all.[3]

We define two performance measures: *Recognition Accuracy* is the proportion of queries whose reference is correctly identified:

$$\text{Accuracy} = \frac{tp}{tp + fp + fn} = \frac{tp}{N} \tag{7}$$

Precision is the proportion of cases, out of all cases where the system claimed to have identified the reference, where its prediction is correct:

$$\text{Precision} = \frac{tp}{tp + fp} \tag{8}$$

Thus, high precision means low number of false positives.

[3]There are no *true negatives* (tn) in our scenario (i.e., cases where the system correctly abstains from identifying a reference because there *is* no correct reference) because for all queries, the matching reference is guaranteed to be in the DB.

	db 1000 songs				db 20,000 songs			
Speed	tot.	tree	feat.	match	tot.	tree	feat.	match
130%	12	1	6	5	60	31	7	22
110%	11	2	6	3	52	28	6	18
100%	9	1	6	2	35	20	6	9
90%	9	1	5	2	37	22	5	10
70%	4	0	3	1	14	7	3	1

Table 1: Query runtimes in seconds. "tot" is the total time, "tree" is the time taken by tree intersection, "feat." is the feature extraction time for spectral peaks and quad grouping and "match" is the matching and verification time.

6.1. Detailed Results on Small Reference-DB ($1,000$ Songs)

Each data point in the visualisation shows one of the aforementioned quality measures for 100 queries. The overall system performance for speed, tempo, and SNR changes is shown in Figure 5. For this experiment a total of 5900 queries of length 15 seconds were run against the database consisting of thousand songs.

Figure 6 shows the performance for the tested SNR levels for speed and tempo modifications of 95% and 105%.

Concerning the noise robustness of the proposed method, the results show that a stable performance of $> 95\%$ for the tested quality measures is achieved for SNR levels down to $+15$dB. According to these results the proposed quad-based hashes seem to be sufficiently robust for queries of various noise levels that may be encountered in real application scenarios.

6.2. Extending the Reference-DB to $20,000$ Songs

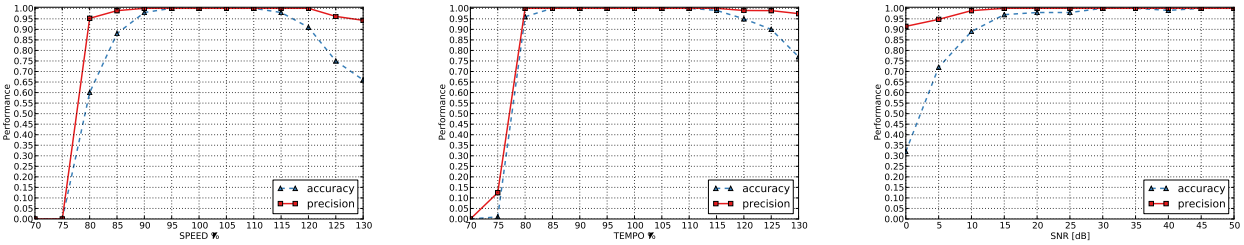
In the previous experiment on a database of thousand songs we reach a very high precision. To further investigate the precision of our proposed algorithm we extend the reference database to 20,000 songs, and query the same audio excerpts that we created for the previous experiment, with the same modifications, against this large database. Figure 7 shows that the performance of our approach does not degrade even if there are many songs in the reference. Note that we parametrized our system to discard any match candidates if their transformation estimates are outside the scale tolerance bounds of $\pm 32\%$ for either frequency and time scale. The performance is comparable to that of the first experiment, resulting in more false positives only for the larger speed modifications. For tempo modifications the system gives basically the same performance as in the first experiment.

6.3. Runtimes

In Table 1 we give information about the runtimes observed in the two above experiments. We randomly pick one of the generated audio query excerpts, and compare the query runtime for the small and the large databases for different scale modifications. The increased runtime for faster speed and tempo is a result of the higher number of spectral peaks in the audio excerpt, for which a larger number of tree-intersections and raw match candidates have to be processed.

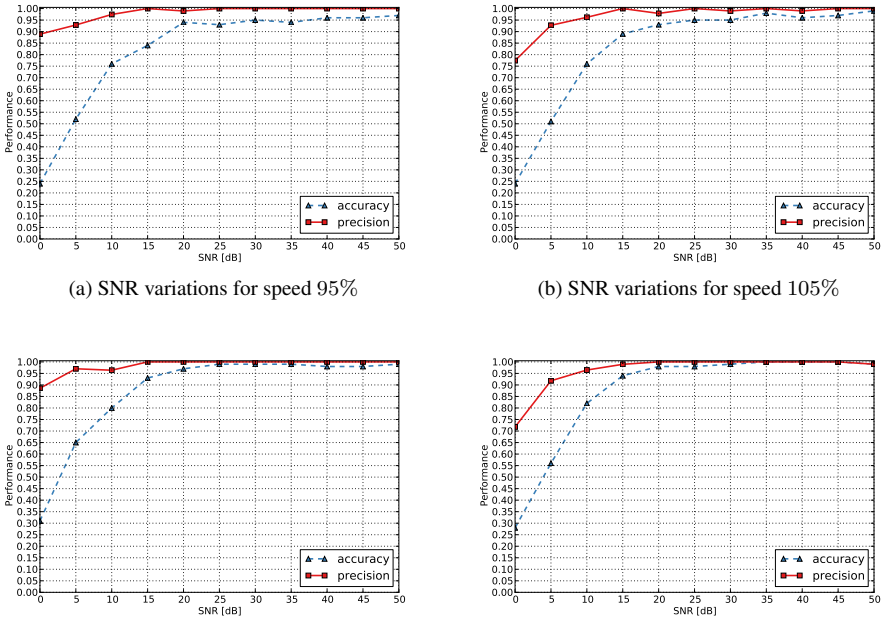
6.4. Comparison with Reference Method [2]

While it is not possible to directly compare our results to those of [2] (because we do not have access to their test data), from the published figures it seems fair to say that in terms of recognition



(a) Speed variations (b) Tempo variations (c) SNR variations

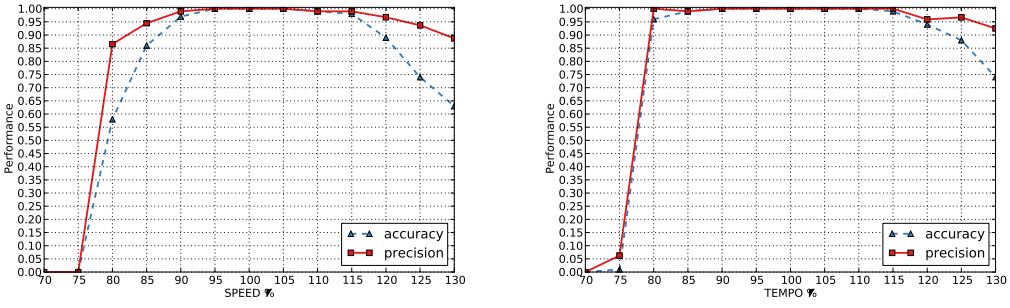
Figure 5: *Precision and accuracy for speed (5a), tempo (5b) and SNR (5c) modifications, on a database of 1000 songs.*



(a) SNR variations for speed 95% (b) SNR variations for speed 105%

(c) SNR variations for tempo 95% (d) SNR variations for tempo 105%

Figure 6: *SNR variations for* 95% *and* 105% *speed and tempo on a database of 1000 songs.*



(a) Speed variations (b) Tempo variations

Figure 7: *Precision and accuracy for speed (7a) and tempo modifications (7b) on a database of 20,000 songs.*

accuracy and robustness, both approaches seem comparable, and both are at the upper end of what can be expected of an audio identification algorithm – for tempo and pitch distortion ranges that are larger than what we expect to encounter in real applications. It should be noted that the results reported in [2] are based on a small reference collection of approximately 250 songs only and that, unfortunately, no information is given about either the size of the resulting database, or about runtimes.

The advantage of our proposed method is its efficiency in terms of data size and fingerprint computation. In contrast to [2], where log-spaced filter banks have to be applied to the spectrogram in order to compute the time-chroma representation, the selection of spectral peaks for quad grouping is done directly on the STFT magnitude spectrogram of the audio, and the quads can be grouped in linear time. Our proposed method chooses spectral peaks that are local maxima of the magnitudes in the spectrogram, in contrast to the method of [2], where 600 DCT computations per second of audio have to be performed (similarity computations for 30 rectangular image patches for each of approximately 20 feature candidates per second) in order to find stable feature points. The hash representation we propose is very compact and can be stored as four float32 values, while the algorithm of [2] uses fingerprints that are represented by 143-dimensional vectors.

Our match candidates are retrieved via an efficient range search in a spatial data structure, for which we use an R-Tree. The distance of hashes is the euclidean distance between four-dimensional points, while the distance measure used in [2] is the measure of the angles of the 143-dimensional fingerprint vectors.

7. CONCLUSIONS

We have presented a new audio identification method that is highly robust to noise, tempo and pitch distortions, and verified its ability to achieve overall high performance on a medium-large database consisting of 20,000 songs. While there is a lot of potential for false positive matches in a database of this size (roughly 43 million quads) in combination with the rather large tolerated scaling ranges, the method's filtering stage and the subsequent verification process enable the system to maintain high precision and accuracy. The results show a stable high performance for a large range of scale changes, with as few as ≈ 9 compact fingerprints per second of audio.

In preliminary experiments on DJ sets and media broadcast data we have not yet found any examples that exceeded 7% of speed or tempo scale modifications. We also assume that scale modification attacks against audio copy detection algorithms are usually done with very subtle scale changes, almost imperceptible to human ears. Our proposed algorithm achieves near perfect overall performance within scale modification ranges of 90% to 115% for speed, and 80% to 120% for tempo scale modifications.

8. ACKNOWLEDGMENTS

We would like to thank Rainer Kelz, Jan Schlüter and Harald Frostel for many intense and fruitful discussions. This research is supported by the Austrian Science Fund FWF under projects TRP307 and Z159.

9. REFERENCES

[1] A. Wang, "An industrial-strength audio search algorithm," in *Proc. Intl. Conf. on Music Information Retrieval*, 2003.

[2] M. Malekesmaeili and R. Ward, "A local fingerprinting approach for audio copy detection," *Signal Processing*, vol. 98, pp. 308–321, 2014.

[3] D. Lang, D. Hogg, K. Mierle, M. Blanton, and S. Roweis, "Astrometry.net: Blind astrometric calibration of arbitrary astronomical images," *The Astronomical Journal*, vol. 137, pp. 1782–2800, 2010.

[4] G. Evangelidis and C. Bauckhage, "Efficient and robust alignment of unsynchronized video sequences.," in *DAGM-Symposium*. 2011, vol. 6835 of *Lecture Notes in Computer Science*, pp. 286–295, Springer.

[5] G. Evangelidis and C. Bauckhage, "Efficient subframe video alignment using short descriptors.," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 35, no. 10, pp. 2371–2386, 2013.

[6] J. Haitsma and T. Kalker, "A highly robust audio fingerprinting system," in *Proc. Intl. Conf. on Music Information Retrieval*, 2002.

[7] F. Kurth, T. Gehrmann, and M. Müller, "The cyclic beat spectrum: Tempo-related audio features for time-scale," in *Proc. Intl. Conf. on Music Information Retrieval*, Victoria (BC), Canada, October 8-12 2006.

[8] S. Baluja and M. Covell, "Waveprint: Efficient wavelet-based audio fingerprinting," *Pattern Recogn.*, vol. 41, no. 11, pp. 3467–3480, Nov. 2008.

[9] B. Zhu, W. Li, Z. Wang, and X. Xue, "A novel audio fingerprinting method robust to time scale modification and pitch shifting," in *Proc. Intl. Conf. on Multimedia*, New York, NY, USA, 2010, pp. 987–990, ACM.

[10] D. Lowe, "Distinctive image features from scale-invariant keypoints," *Int. J. Comput. Vision*, vol. 60, no. 2, pp. 91–110, Nov. 2004.

[11] J. Six and O. Cornelis, "A robust audio fingerprinter based on pitch class histograms: applications for ethnic music archives," in *Proc. Intl. Workshop of Folk Music Analysis*. 2012, p. 7, Ghent University, Department of Art, music and theatre sciences.

[12] A. Guttman, "R-trees: A dynamic index structure for spatial searching," *SIGMOD Rec.*, vol. 14, no. 2, pp. 47–57, June 1984.

[13] "Ogg Vorbis," Available at http://www.vorbis.com.

[14] "SoX - Sound eXchange," Available at http://sox.sourceforge.net/.