# Audio Fingerprinting

Andrew Lam, Han Joo Chae, Ryan Walsh and Steve Thompson
Electrical and Computer Engineering
Carnegie Mellon University
*{aclam, hchae, rwalsh, sathomps}@andrew.cmu.edu*

## Abstract

*This paper describes an implementation of a robust audio fingerprinting algorithm as mentioned in Shazam. [1] There are many challenges that need to be addressed when performing audio fingerprinting and the algorithm presented is able to overcome those challenges by utilizing characteristics of the audio that are persistent even if the audio is in a lower quality state than the original. Our results show that our implementation is extremely robust giving us no false positives and a near 100% identification rate.*

## 1. Introduction

Audio fingerprinting is best known for its ability to robustly identify a short audio track regardless of format. In this section, we will discuss the usefulness, general process and challenges of audio fingerprinting.

### 1.1. Usefulness

The biggest advantage of audio fingerprinting is the ability to discover new music with only a short, poorly recorded sample of it. It can even be recorded with a cell phone in the presence of ambient and burst noise. For example, if we want to get some information about a song that is playing when you are sitting in a café or music that is on the radio, we can simply record several seconds of the song through a microphone that is attached on a cell phone. In addition, audio fingerprinting is very useful when you are trying to automatically identify the author, album and name of unnamed audio files. By submitting those songs to an audio fingerprinting tool, we can fill out those missing information.

For the above reasons, audio fingerprinting has been integrated into many well-known and popular applications, however its usage is not limited to figuring out unknown information about the song. It can be used for an automated system that detects copyright infringers by fingerprinting any audio files that are being transferred. For example, copyright holders can simply run audio fingerprinting on servers to detect whether or not illegal audio files exist in such servers. If they find any violation, the system will report so that the copyright holders can take immediate actions.

Moreover, more intelligent search engines can be introduced by forming content aware networks. Rather than simply searching title names of audio files which is wrong or misleading a number of times, it can actually analyze those audio files to provide correct and better search results to users.

### 1.2 Process overview

The process of audio fingerprinting can be divided into four steps: generating a fingerprint, storing it in the database, processing the clip of a song for a match and returning the identity of the song if available. The first two are the most important steps because the performance will vary mostly based on how we choose and store a fingerprint. As there are millions of songs to compare against, if we fail to choose good features for a fingerprint and efficiently store fingerprints in the database, the matching process will take a significantly long time with unsatisfying results. Once the database is created, a clip of a song can be processed and the identity of the song will be returned if there is a match.

### 1.3. Challenges

Due to the high dimensionality and the significant variance of the audio data for perceptually similar content, audio fingerprinting that automatically identifies audio content is very difficult. The direct comparison of the digitalized waveform cannot be used mainly because it needs to deal with various factors such as noise, differing sample rates, transcoding, distortion from recording and playback devices, etc. Moreover, comparing an entire waveform one by one will take significant amount of time if a database contains millions of songs. Although hashing can be used to represent a song by taking it as binary file, the method is fragile because a single bit flip is sufficient for the hash to completely change. That is, a minimal distortion of any kind will completely change results leading the system to fail on content-based identification. In addition, audio fingerprinting is required to find a full song from a very short clip which is usually about ten seconds. Therefore, a new method needs to be introduced

to intelligently create a fingerprint and keep the database in reasonable size and speed.

## 2. Choosing a fingerprint

There are several properties to keep in mind when determining what an audio fingerprint should be.

**Uniqueness**. Like human fingerprints, no two songs should have the same fingerprint. A fingerprint should be able to uniquely identify a song. The ability to distinguish between different versions of the same song is a plus.

**Perceptually Relevant**. Bitwise comparisons of different encodings of the same song are very different. It is important that songs that are the same to human ears, but not necessarily the same waveform, have similar fingerprints.

**Robustness**. For this problem space, an audio clip may face a multitude of noise and filters. For example, the user may submit a cell phone recording of a radio broadcast. This clip is subject to radio band-limiting, attenuation as the sound travels from the speakers to the microphone on the cell-phone, not to mention noise from the microphone when recording and external noise from the environment.

**Fast Lookup/Matching**. In a database of millions of songs, it is not acceptable for the search to take more than a few seconds to return a hit. The features calculated for a fingerprint must be quick to compute, since a fingerprint must also be generated from the search clip.

**Compact Representation**. With millions of songs in the database, each fingerprint should have a small footprint in memory. However lookup speed is prioritized over size since the end user experience is more important than size and speed is the more difficult problem to solve.

Coming up with a fingerprint that fulfills all the criteria listed above is a difficult challenge. It is easy to choose a feature that excels in one area but is terrible in another. For example, a feature like the BPM of a song is a perceptually relevant feature, but is not unique and is difficult to calculate for certain songs that do not have a fixed BPM. It is possible to combine several of these features together to create a better fingerprint however computation time will increase.

Researchers at Shazam have come up with a fingerprint and an index/search method that fulfills all of the above desired properties. Their solution, linked power peaks, is detailed in later sections.

## 3. Related work

There are many proposed ways of calculating audio fingerprints. Many companies have been developing algorithms to solve this problem. Philips has developed their own algorithm for generating and storing audio fingerprints [3]. Last.fm has developed their own fingerprinter application [4]. Shazam has developed an alogrithm that has proven itself in a commercial environment [1] and this paper will describe Shazam's algorithm in more detail.

## 4. Basic algorithm overview

Section 2. describes some of the difficulties faced when selecting a good fingerprinting algorithm. This section will provide a basic overview of the theory behind Shazam's audio fingerprinting algorithm.

### 4.1. Generating a fingerprint

Shazam uses a constellation map of power peaks as an audio fingerprint. The intuition behind using power peaks is that these peaks are the most likely to survive in the presence of noise, filtering, and encoding artifacts. These peaks are simply local maxima of the log power spectrum of the STFT of a song. These local maxima are found within a certain time-frequency window of the power plot. The size of this window can control the number of peaks found. Peaks below a certain amplitude can also be removed, as these peaks are more susceptible to noise. The number of peaks is a trade-off between identification accuracy and speed.

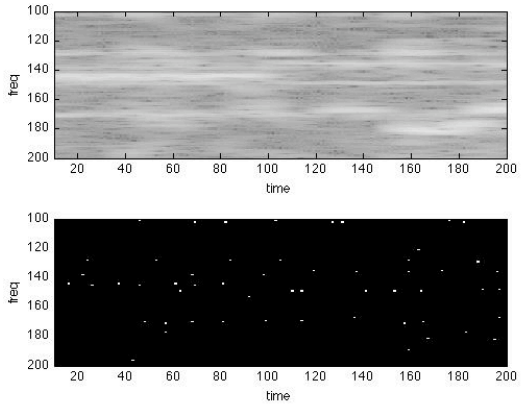


**Figure 4.1. Spectogram and corresponding Constellation Map**

### 4.2. Matching fingerprints

A fingerprint from a short clip of a song should be a subset of the full constellation map that was calculated from the full song, plus or minus a few peaks added/removed due to noise/filtering/encoding effects. The basic idea is to find the location in time where the constellation map of the short clip overlaps the constellation map of a song.

## 5. Implementation details

Several questions still remain: how do we represent this constellation map compactly and how do we match these constellation plots? For matching, a brute force sliding window is impractically slow. The obvious choice for a fast lookup is to use a hash table that is able to store multiple values with the same key. A lookup should return all the values associated with a key.

A naïve approach to represent a constellation map would be to store individual peaks in a hash table, with the frequency as the key and the song id and time offset as the value:

*<freq of peak, [song id, time offset]>*

This approach will return too many hits and loses a lot of information about its relationship to the song itself, as many songs have multiple peaks in the same frequency bin.

Shazam addresses this problem by increasing the entropy of keys by forming pairs between close peaks. Peaks in this method are now stored in the hash table as follows:

*<[f peak1, f peak2, offset], [song id, time offset]>*

Multiple pairs are formed with each peak in the case a peak is lost due to noise/filtering/encoding effects. This is a trade-off between space and speed. More hashes are required to store each peak, however look up speed is improved significantly. There is also a slight loss in accuracy since peaks must be found in pairs. A peak that is lost will affect other peaks that it is paired with. However by forming multiple pairs, the chance of a non-matching a peak due to the disappearance of another peak is minimized.

### 5.1. Generating fingerprints

Key values for the hashtable are generated by pairing peaks. Every peak is paired with peaks in a given target area. The target area is simply a time-frequency window centered around the frequency of the first value, and ahead in time of the first value. The size/shape of this window affects how many and which pairs are created and will directly affect accuracy. In our implementation, our target area is simply a rectangle in the time-frequency space. A fanout factor can also be specified to limit the number of pairs created per anchor point, creating a trade-off between accuracy and space.

### 5.2. Matching fingerprints

Once fingerprints have been generated for the input sample clip, we retrieve all values from our database which have the same key. Each value is then sorted by song. Any song with a large enough number of results is further analyzed.

Each value stored with the key contains the time offset into the song as well. The time offset into the sample clip of the fingerprint that matches the one given by the database is subtracted from the stored fingerprint. The resulting time values are then passed through a histogram. The idea here is that the returned values of the database should be in the song in the same order as the fingerprints that generated them. The figure below shows what one such histogram looks like. We then analyze the resulting histograms to determine whether the song was a match or not. What we are essentially looking for is for clear outliers in the histogram. With millions of songs, many songs will have some peaks, but when songs truly match, the peaks should be fairly obvious, as below.

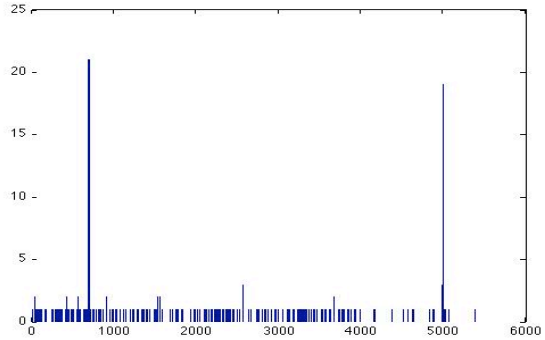


**Figure 5.2. Histogram of returned values minus time offsets. Interesting because the chorus is repeated in this song, and thus appears twice.**

## 6. Performance

This section details our results.

### 6.1. Database and test sets

Generated fingerprints have been stored in a hash table, which serves as our database. Our database consists of fingerprints generated from five different songs. In order to test our implementation accuracy, we recorded a set of short audio clips. We have created two different test sets: 45 audio clips taken from songs which exist in our database and 100 audio clips of samples not in our database. A number of different songs have been chosen ranging from different genres to very similar songs and we even chose songs that are from same artists. Matching samples have been generated by recording randomly chosen sections of song that are in the database ranging from 5 to 15 seconds. During the recording, a regular laptop microphone has been used in the presence of noise and other large noises are randomly introduced. Similarly, non-matching samples have been generated on songs not in the database.

### 6.2. Speed and accuracy

During our test, the best and worst times for a lookup was 1.65 seconds 13.79 seconds respectively*. Although

the time gap between the best and the worst time is quite large, most of the lookup time stayed on about 7 seconds.

***The running time is based on our MATLAB implementation.**

| | Matching | Non-matching |
|---|---|---|
| Correct | 36 | 0 |
| Incorrect | 0 | 0 |
| Match not found | 3* | 100 |

**Table 6.1. Test results of 39 matching samples and 100 non-matching samples. *These misses should actually be correct. The hash table implementation we are using does not support multiple values per key. Matching keys from different songs will overwrite each other in the hash table.**

Considering that we have tested with a various audio clips, our system fairly high performance with nearly 100 percent accuracy. Although the system missed 3 songs because of our hash table implementation issue that does not support multiple values per key, our system had 100 percent accuracy when tested individually, which prevents overwriting hash table values.

## 7. Web interface

Our web interface utilizes Java Server Pages with an Apache Tomcat backend. The MATLAB compiler is able to take MATLAB code and compile it into Java class files. This is what we did in order to integrate our fingerprinting algorithm with the web interface. The user is prompted with a screen that allows them to select a file to upload (currently we only support *.wav files and *.wav files under 10MB in order to spare the webserver) and once the upload has finished the JSP backend makes calls to Java in order to process the *.wav file. The name of the song (or lack thereof) will then be presented back to user and they are able to upload another *.wav if desired. The uploading interface was developed on top of JavaZoom's uploadbean interface.

## 8. Conclusion

Overall, the algorithm described by Shazam is very successful in terms of accuracy, robustness and processing speed. After implementing this algorithm, it is clear that the most difficult part is correctly determining which peaks to pick from the Constellation plot. While we were working on the implementation, we tried several techniques to try and reduce the number of peaks, and thus improving speed, without reducing the accuracy. The most successful ones were ones that took advantage of the way the algorithm is generally used. That is, most audio recording devices like laptop and cell phone microphones are not very good at picking up certain ranges of frequencies. Attempting to change the peaks selected based on content and other criteria of the song generally resulted in much lower accuracy and more false positives.

Despite its simplicity, this algorithm is very robust and we were amazed at its efficacy. We do not believe it is an overstatement to say that the Shazam algorithm has addressed and solved all the major challenges of audio fingerprinting.

## 9. References

[1] A. Wang “An industrial strength audio search algorithm,” Proc. ISMIR Baltimore, MD, 2003, p. 7.

[2] "Upload Bean Support". JavaZOOM. 11/25/09 <http://www.javazoom.net/jzservlets/uploadbean/uploadbean.html>.

[3] Jaap Haitsma, Antonius Kalker, “A Highly Robust Audio Fingerprinting System”, International Symposium on Music Information Retrieval (ISMIR) 2002, pp. 107-115.

[4] http://blog.last.fm/2007/08/29/audio-fingerprinting-for-clean-metadata