Shazam

1. Theory: Fingerprints

Each audio file is “fingerprinted”,a process in which reproducible hash tokens are extracted. We can say that fingerprint is a kind of feature the audio signal has.
The features for each song (and clip) will be characterized by the location of local peaks in the magnitude of the spectrogram. The frequencies and timing of the peaks will be stored as features.

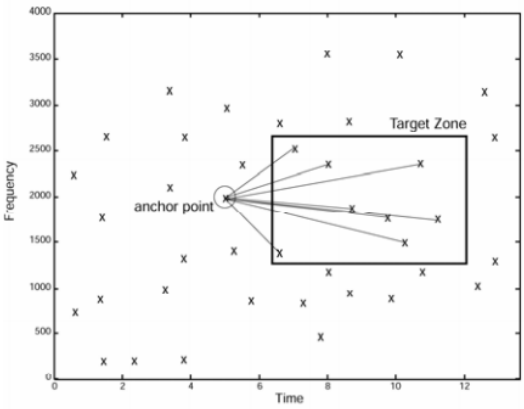

Fig. 1C - Combinatorial Hash Generation

Figure 1

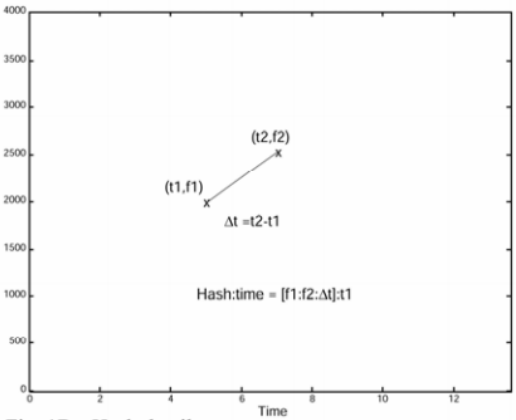

Fig. 1D - Hash details

Figure 2

From Figure 1 we can see that each point represents a local peak extracted from spectrum. For each anchor point, it has a target zone, points in this zone are time correlated to the anchor point. We establish Hash table based on this. In Figure.2, we get a peak pair. We record frequencies of this two points, and calculated the offset time between points. We need offset time because the clip may not start at the beginning of the original song. It may start in the middle. Each hash value in hash table is (f1,f2,t2-t1). We store a succession of hash values from a signal and add them into hash table.

From songs provided, we establish a hash table contain all the hash value of each songs. This hash table is what we called database, or fingerprints database.

To identify a clip, we need to extract fingerprints or hash values from the clip, and compare them with values in database. If values match in a constant way, we can say that the clip could match this song. However, we need to compare number of matching values for different songs, and find the maximum number of matching, the song with maximum number of matching (the clip) is the song we identify.

2. Fingerprints

The function fingerprints.m is provided. This function is made to compute the local peaks for a single signal.

2.1 Resampling

The sample rate fs of musicDB is 16kHz, which is relatively high. It would be costly to process. Therefore, I used *resample* command to resample the signal at 8000kHz.

y = resample(y, new_rate, old_rate); % rates must each be integers.

2.2 Spectrogram

[S,F,T] = spectrogram(y, window, noverlap, [ ], fs);

By using the frame mode in Project 2. I set window=64ms, noverlap or stepsize=32ms. I did short-time Fourier transform to resampled signal y and get its spectrogram S.
F is the frequency vector and T is the time vector. Then I calculated the magnitude of spectrogram.

2.3 Local peaks

I find the local peaks of the spectrogram and produce a binary matrix (the same size as the spectrogram) with a 1 at each location of a peak.

A local peak has magnitude greater than that of its neighbors. I need to compare each point with points nearby. Assuming that I only compare in horizontal and vertical direction, or the upper, lower, left, right points. I use *circshift* to compare in four directions.

```
peaks = ones(size(magS));
CS=circshift(S,[vertShift,horShift]);
Q=(S>CS);
peaks=peaks.*Q;
```

In each comparison, I get a matrix Q containing locations of max intensities.
I generate a all-one matrix called peaks, and multiply peaks with Q, so that only locations survive all the comparison are the peaks.

2.4 Thresholding

I use only larger peaks to reach a higher speed in matching. I set a threshold to get rid

of small peaks. The threshold should be proportional to loudness.

2.5 Check results

Set *optional_plot=1.* The matlab plots spectrogram with peaks as black points on it.

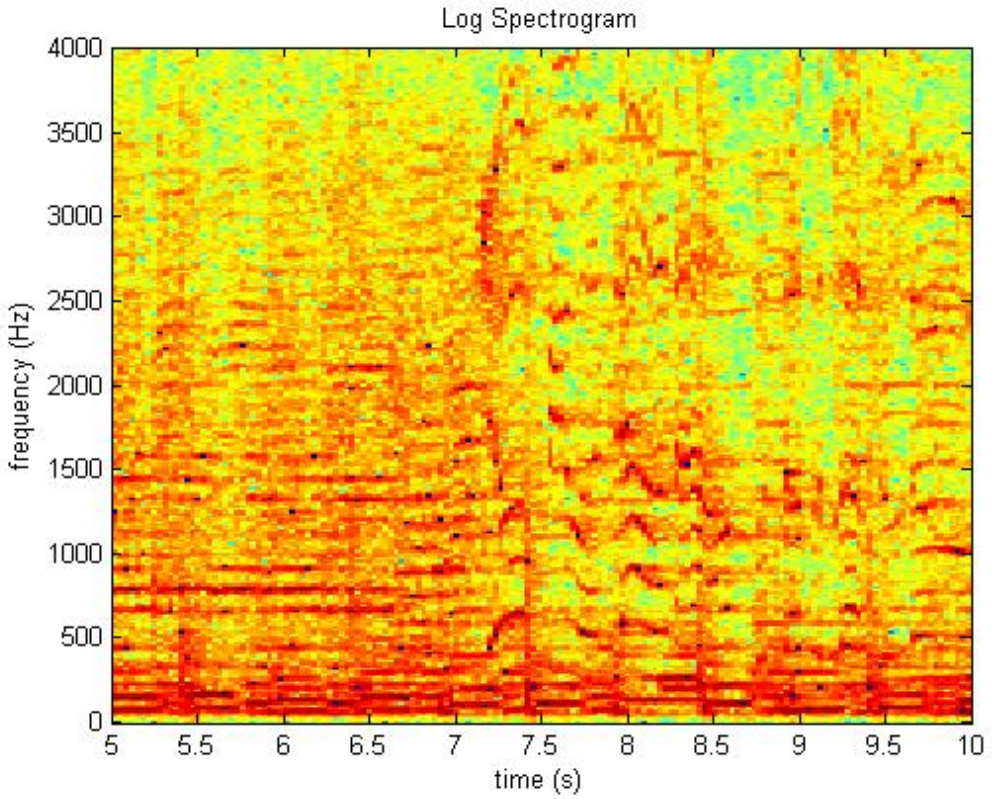

Figure 3

3 Compute fingerprints

Compute_fingerprints.m is provided to establish a database from musicDB.mat

3.1 Find peak pairs

Provided function file *convert_to_pairs.m,* which takes a matrix of peaks and returns a table of pairs that are close in both time and frequency.

The theory is shown in Figure 2. During the search, we limit the number of pairs that we accept by the parameter *fanout.*

3.2 Create database

Provided *simple_hash.m* and *add_to_table.m.*
Edit a global variable *hashtable.* Use functions mention above to create *compute_fingerprints.m.* In *compute_fingerprints.m,* I save two databases created, HASHTABLE.mat and SONGID.mat.

4 Identifying

Provided *identify_song.m*

4.1 Create a clip

This will be discussed in detail in Noise Robustness part.

4.2 Extract fingerprint

Use *fingerprints.m* function to extract fingerprints from the clip. Calculate hash values of the clip.

4.3 Recover matches from hash table

For each value, find the matching values in database. Then, two cell arrays I got: *matchID* and *matchTime. MatchID* are song ids for matching values. *MatchTime* means the time t1 where the matches occurred in database. The same song may contain the same peak pair at different times t1, so the same song ID number may appear multiple times in *matchID*.
Now convert *matchTime* into *offset* by subtracting t1 that the peak pair occurred in the clip.

4.4 Identify song

Find the song that has the most occurrences of any single timing offset. I just find the most occurred ID in *matchID.* However this method is not always correct. Most of the time, it will be a right match. Sometimes, it provides a wrong match. Because f1, f2,t1,t2 are not precise. They are not particular values. They vary in a range. These should be fairly robust to many possible forms of distortion, such as magnitude and phase error in the frequency domain due to the recording process or additive noise. The accuracy of matching will be improved in Noise Robustness part when I add noise in it.

4.5 Test the accuracy

By using the clip I create, test my system. I find that the longer duration of the clip, the more accurate of my matching. It is intuitive to understand. When I use Shazam to identify songs, the app takes fifteen or more seconds to “listen”. Three seconds is too

short.
I plot the histogram of offset for clip from song_22.

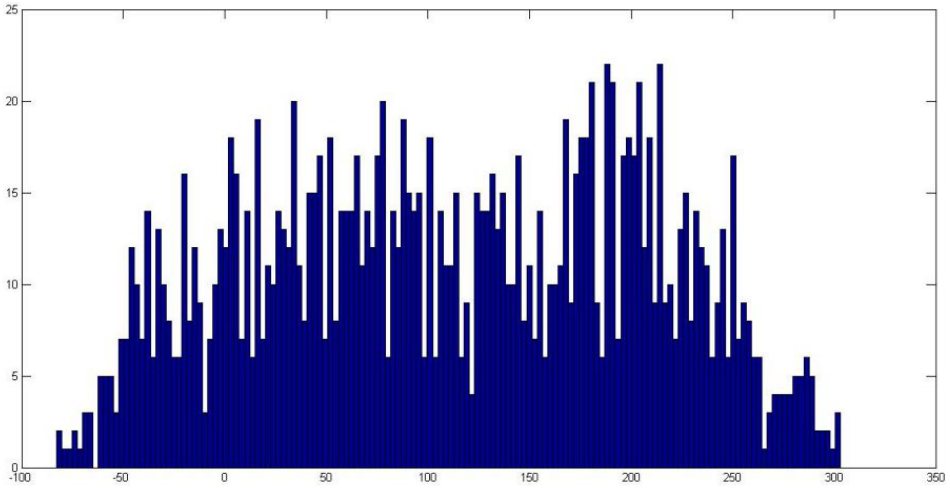

Figure 4

As we can see, it is difficulty to find the highest one. Therefore it will decrease the accuracy of matching.

Figure 5 shows the histogram from a noisy clip. Because of robustness, there are two obvious peaks. In this case, the accuracy will be better.

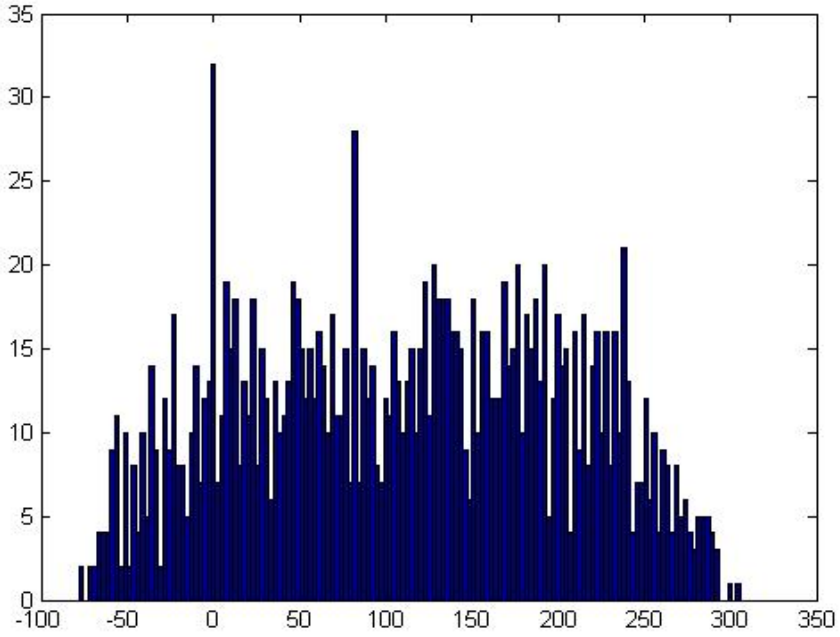

Figure 5

5 Noise robustness

Provided *noise_robust.m* , this file can also be used as a test driver to test *identify_song* and *compute_fingerprints.*

First use rand to select a random song. Next, select a random part from this song by shifting. Generate a white noise with SNR equaling the value I set.
Use identify_song function to identify clean and noisy songs. Compare the results. The accuracy is better than 0.8 most of time from SNR=0~100dB.

6. Difficulties

In identify_song.m, for each hash value, there were multiple matches. I didn't know how many, so I created cell arrays. However, I took a lot of time to deal with cell arrays, and got confused. At last, I converted cell arrays into matrices. I found the indexes to localize the values I wanted.

Another thing that I remembered was the use of compute_fingerprints. I added code in identify step. Therefore, everytime I ran the file, Matlab kept busy.

The accuracy of identification needs to improve by drawing the histogram of offset.

Reference

[1] http://www.ee.columbia.edu/~dpwe/papers/Wang03-shazam.pdf
[2] https://www.princeton.edu/~cuff/ele201/files/lab2.pdf