# ELE 301, Fall 2011
# Laboratory No. 6 - MyShazam Part II

## 1 Background

From last week's lab you should have a MATLAB function `make_table` that takes as input a clip and returns a table of peak pairs of the form:

$$\left|\begin{array}{c|c|c|c} f_1 & f_2 & t_1^c & t_2^c - t_1^c \\ \vdots & \vdots & \vdots & \vdots \\ f_j & f_k & t_j^c & t_k^c - t_j^c \\ \vdots & \vdots & \vdots & \vdots \\ f_m & f_n & t_m^c & t_n^c - t_m^c \end{array}\right|$$

Now we want to run `make_table` on each song that we want to include the database and produce a table for each. Each of these song tables will have the same form:

$$\left|\begin{array}{c|c|c|c} f_1 & f_2 & t_1^s & t_2^s - t_1^s \\ & & & \vdots \\ f_j & f_k & t_j^s & t_k^s - t_j^s \\ & & & \vdots \\ f_p & f_q & t_p^s & t_q^s - t_p^s \end{array}\right|$$

If the clip comes from a particular song, we expect each entry in the clip table to have a corresponding entry in the song table. In particular, if $(f_1, f_2, t_1^c, t_2^c - t_1^c)$ is an entry in the clip table and $(g_1, g_2, t_1^s, t_2^s - t_1^s)$ is its corresponding entry in the song table, then we should have $f_1 = g_1$, $f_2 = g_2$, and $t_2^c - t_1^c = t_2^s - t_1^s$ because these features are time shift invariant. So using the triple $(f_1, f_2, t_2^c - t_1^c)$ from a clip table entry is a good way to search for a match from the set of song tables.

To make this process efficient, the song tables need to be combined to form our database of song "fingerprints" and we need to form the database so that searches are fast.

### 1.1 Finding a Clip Match

To identify a clip, we run the function `make_table` on the clip to generate a clip table of peak pairs. Then we search the database for matches to each entry in the clip table. We want to use the triples $(f_1, f_2, t_2^c - t_1^c)$ to search the database. What we need is a fast way to determine if $(f_1, f_2, t_2^c - t_1^c)$ matches something in the database and if so, extract what we know about that match.

To ensure fast database lookup we construct a hash table that is indexed by a simple hash function of the triples $(f_1, f_2, t_2 - t_1)$. Denote the hash function by $h(f_1, f_2, t_2 - t_1)$. Think of $h(f_1, f_2, t_1 - t_2)$ as an integer (the hash) that indexes entries in a large table (called a hash table). For each entry in the song tables, we place both the code name for the song (songid) and the $t_1^s$ value at the location $h(f_1, f_2, t_2^s - t_1^s)$ in the hash table.

Then given a peak pair entry in the clip table, say $(f_1, f_2, t_1^c, t_2^c - t_1^c)$, we look up the database entry at $h(f_1, f_2, t_2^c - t_1^c)$. This entry is either empty (no match for the peak pair), or there is one entry $(t_1^s, \text{songid}_k)$ giving us a songid and the time $t_1^s$ where this peak pair occurs in the song, or there is a set of entries $\{(\text{songid}_k, t_{1k}^s)\}_{k=1}^p$ one for each song that has a song table entry of the form $(f_1, f_2, t_{1k}^s, t_{2k}^s - t_{1k}^s, \text{songid}_k)$.

All the matches from the clip table to the correct song should occur with the same difference $t_o = t_1^s - t_1^c$, where $t_1^s$ is the time of the pair in the song, and $t_1^c$ is the time of the pair in the clip. The time $t_0 = t_1^s - t_1^c$ is the offset in time we would need for the clip constellation to

line up with the song constellation. This suggests that we find the songid that has the most matches occurring with the same offset $t_0$ and assert that our clip comes from that song, or rank the matches by this criterion.

### 1.2 Noise and SNR

Once you have a working system, you will need to characterize its performance. One important metric is the performance with varying levels of noise added to the clip. We can construct multiple clips from each song in the database and add noise to each. Then we classify each clip with our algorithm and record the rate of correct classification for different signal to noise ratios (SNRs). Ideally, at high SNR the classification is close to 100% correct and it should degrade as the SNR decreases.

We define the SNR as:

$$SNR_{dB} = 10 \log_{10} \frac{P_{\text{signal}}}{P_{\text{noise}}}$$

To measure signal power in MATLAB, you can take the mean of the square of the signal. For 0-mean Gaussian noise with variance $\sigma^2$, the power $\sigma^2$.

## 2 Lab Procedure - MyShazam Part II

You will need to write a few different scripts. The following are suggestions for what the structure of the scripts should be. Top-level script `make_database`:

1. Name the directory where the .mp3s are located
2. Use `getMp3List` to obtain a list of songs
3. Call `add_to_hash` with the list of songs

Script `add_to_hash`: load current database (hash table) and songlist, add new songs, save updated database and songlist.

1. Initialize `HASHTABLE` and `SONGID` matrices if they do not exist. If they do exist, load `HASHTABLE.mat` and `SONGID.mat` using the command `load`.
2. For each song in the input list, first update `SONGID` then use the function `make_table` that you created last week to produce a table of 4-tuples $(f_1^s, f_2^s, t_1^s, t_2^s - t_1^s)$. For each 4-tuple $(f_1^s, f_2^s, t_1^s, t_2^s - t_1^s)$, find the corresponding hash index using the hash function: `index=`$h(f_1^s, f_2^s, t_2^s - t_1^s)$. Store $(t_1^s, songid)$ at `HASHTABLE(index)`. Be sure to handle collisions appropriately, as they will surely occur.
3. Save the updated `HASHTABLE` and `SONGID` using the `save` command.

Script `myshazam`: Take a clip and try to match it to one of the songs in the database.

1. Load `HASHTABLE` and `SONGID` that were constructed in the previous two scripts.
2. Run `make_table` on the input clip to produce a table of 4-tuples.
3. For each 4-tuple $(f_1^c, f_2^c, t_1^c, t_2^c - t_1^c)$, compute $h(f_1^c, f_2^c, t_2^c - t_1^c)$, the hash index. For each pair $(t_1^s, songid)$ residing at that index (there could be none, or several if collisions occurred there), store $(t_1^s - t_1^c, songid)$.
4. After going through the entire clip table, there will be a collection of $(t_1^s - t_1^c)$ values for each song in the database. The song that the clip matches will have a large mode, or a spike in the histogram of $(t_1^s - t_1^c)$ values. Determine a match by using `hist` or `mode`.

Some steps are now discussed in detail.

### 2.1 Constructing and Populating the Hash Table: `add_to_hash`

Since each frequency can be represented by a number from 0 to 255 (assuming for convenience that we drop the last (highest) frequency bin) we can represent each frequency with 8 bits. Depending on how large a time window we allow for our target box, we can also represent the time $t_2 - t_1$ by a $n$ bit number, for some small $n$. Therefore, we can construct a simple hash function of the form:

$$h(f_1, f_2, t_2 - t_1) = (t_2 - t_1) \cdot 2^{16} + f_1 \cdot 2^8 + f_2$$

(make sure that $f_1$ and $f_2$ range from 0 to 255 and not 1 to 256). Then, at each location you will store the songid number and the value of $t_1$ for the peak pair. There will be conflicts (collisions) when populating the table. You need to come up with some appropriate scheme to resolve them (e.g. separate chaining, linear probing, etc.). How big does your hash table need to be? Also, how does the distribution of peaks together with your method of choosing the peak pairs from within the target box affect the number of collisions?

Write a function `add_to_hash` which takes as its argument an array of the names of the songs. It should then run `make_table` on each of song and use the table produced to populate the hash table. At the end of the function, save the hash table to `HASHTABLE.mat` using the `save` command. You should also save an array of song titles to `SONGID.mat`. This will allow you to use indices to refer to the songs instead of strings. As a general rule of thumb, it is always a good idea to reserve space, and initialize values, for variables.

### 2.2 Finding a Match: `myshazam`

Next, write a function `myshazam` which takes in a clip and returns the song ID of the match, if there is a match. To find a match, the function will have to construct the peak pair table for the clip and then look up each entry in the hash table. The function should then find the song ID which has the most pairs with matching $t_0 = t_1^s - t_1^c$, where $t_1^s$ is $t_1$ for the pair in the song and $t_1^c$ is $t_1$ for the pair in the clip. If you draw a scatter plot of $t_1^c$ vs. $t_1^s$ for each song in the database, the matching song should have a line of points.

When you are done, test your program out by excerpting a 10 second clip from one of the files and running your match function on it. You can use `mp3write` to convert this clip as an mp3 if you want. Have it draw the scatter plot of the $t_1^c$ vs. $t_1^s$ for the matching song.

How could you measure confidence in a classification?

Run the matching function on multiple clips from each song in the database (automate this process, as it will be necessary for the next section) and record your correct correct classification rate below:

### 2.3 Performance with Noise

To test performance in the presence of noise, loop over `SNRdB=-15:3:15`, as in Wang's paper. For each SNR add noise to each of your test clips with the appropriate power, where you can generate a vector, length $n$, of 0-mean Gaussian noise with variance `sigma^2` with the following command:

```
noise=sqrt(sigma^2)*randn(n,1);
```

Obviously the noise power needed will depend on the power of the clip. For each fixed SNR record the percentage of correct classifications. Plot the percentage of correct classifications vs. SNR.

Do the above with clips of length 5, 10 and 15 seconds. How does the length of the clip affect the correct classification rate?

Test your program on a bunch of clips corrupted with speach or some other form of noise, how does it perform?