Princeton ELE 201, Spring 2014
Laboratory No. 2
Shazam

1 Background

In this lab we will begin to code a Shazam-like program to identify a short clip of music using a database of songs. The basic procedure is:

1. Construct a database of features for each full-length song;
2. When a clip (hopefully part of one of the songs in the database) is to be identified, calculate the corresponding features of the clip;
3. Search the database for a match with the features of the clip.

Like Shazam, 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. 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. A clip is matched to a song by considering all possible shifts in time and comparing the features.

Matching a clip to a song this way can lead to computational challenges. To mitigate this, the features are simplified and preprocessed. Pairs of peaks that are close in both time and frequency are identified, as in Figure 1, resulting in the following table of information, one row for each peak pair:

t_1	t_2	f_1	f_2	songid
324	328	26	34	“song1”
			$\vdots$	

For each song in the Shazam database, these feature pairs are stored in a hash table for easy access. The hash value is calculated from the vector $(f_1, f_2, t_2 - t_1)$, so that peak pairs with the same frequencies and separation in time are considered a match. The timing t_1 and the `songid` are stored in the hash table.

When a clip is to be identifies, the list of pairs of peaks is produced, just as it would have been for a song in the database. Then the hash table is searched for each pair in the clip. This will produce a list of matches, each with different stored values of t_1 and `songid`. Some of these matches will be accidental, either because the same peak pair occurred at another time or in another song, or because the hash table had a collision. However, we expect the correct song match to have a consistent timing offset from the clip. That is, the difference between t_1 for the song and t_1 for the clip should be the same for all correct matches. The song with the most matches for a single timing offset is declared the winner.

2 Lab Procedure - Part I

This week you will build the complete training system for Shazam, which extracts the fingerprints from all MP3 files in a designated folder and creates a database. You can use any MP3 files you want.

The main steps of this procedure are the following:

1. Read in the song using `mp3read`.
2. Average the two channels, subtract the mean, and downsample.
3. Compute the spectrogram of the song using `spectrogram`.
4. Find the local peaks of the spectrogram by using `circshift` in a loop.
5. Threshold the result of step 5 to end up with a specified rate of peaks retained per second of sound.
6. Find pairs of proximal peaks and add load them into a hash table.

These steps are now explained in detail.

2.1 Reading an MP3 file: `mp3read`

In order to read an `.mp3` file into Matlab we will use the function `mp3read` designed by Professor Dan Ellis, Columbia University. The necessary files are available on the course website, and they must be in the working directory or path in order to use them. (On Windows, only the three files `mp3read.m`, `mpg123.exe`, and `mp3info.exe` are needed.) The command works like `wavread`, returning the sound signal and sample rate.

```
[y,fs] = mp3read('file_name.mp3');
```

This opens the file `file_name.mp3`, decodes the file, and returns the decoded signal in the vector `y`. The sampling rate is stored in `fs`.

2.2 Fingerprint

You have been provided with a template for the function `fingerprint.m`. This function takes as arguments a sound signal and its sampling frequency. The function is missing important components of the code, which you will produce.

2.2.1 Preprocessing

Many sound signals from MP3 or WAV contain more than one channel (left and right for stereo systems). In Matlab, these channels are stored as separate columns of a matrix. For our purposes it suffices to consider only the mean of the corresponding samples. Create a new signal that is a vector rather than a matrix by averaging the channels using `mean`. Check that the result is a vector. You may wish to play this through the speakers to see that it sounds like the original. We will work with this new signal.

Also, we wish to remove the DC bias of the sound signal. The DC bias is the average value of the signal, which is not audible but can affect the spectrogram. Remove this by subtracting the mean from the signal. Why might this re-centering of our signal be a good idea? You may find it illuminating to view the spectrogram with and without the mean removed. Q1

The sample rate `fs` may be very high, such as 44,100 Hz for CD audio, which would be costly to process. This sample rate allows the music to contain sounds as high as about 20 kHz, but we can successfully identify songs without the high frequency information. Resample the signal at 8,000 Hz using the command `resample`, as follows, where `y` is the sound vector:

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

This command performs an interpolation of the signal at the new sampling points and returns the result.

2.2.2 Spectrogram

Now will construct the spectrogram of the signal using the Matlab command `spectrogram`. Call it as follows:

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

where `window` is an integer that indicates the length of the segments for the DFTs, `noverlap` is the number of samples that adjacent segments will be overlapped, and `fs` is the sampling rate of the signal, in our case 8,000 Hz. Other than the sound signal, each of these arguments must be iteger valued. They are measured in terms of the number of samples.

Matlab will detect that the sound signal is real valued and only return the spectrogram for positive frequencies in the matrix `S`, which is exactly what we want. The frequency vector for the vertical axis is returned in `F` and the time vector for the horizontal axis is returned in `T`.

Compute the spectrogram with window length 64 ms and an overlap of 32 ms. Note that the number of samples in a window is simply the window length multiplied by the sampling rate. Do not hard-code numbers into your code. Instead, calculate them as functions of parameters that you list at the beginning of the m-file. To force a calculation to be an integer, use `round`.

Plot the magnitude of the spectrogram of the song with axes appropriately labeled. Also plot the log of the magnitude of the spectrogram with axes appropriately labeled. M1

It is common to visually study the log of the magnitude of the spectrogram. Why might this be a good idea? Q2

After completing these plots, comment of remove them from the code. A visualization for the next task is provided in the code already (at the end), and can be turned on when needed.

2.2.3 Local peaks

Next, we 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. One way to find the local peaks is to iterate through each point in the spectrogram and compare the magnitude to that of each nearby point. You would probably have four nested for-loops. The first two would be used to index each location in the matrix as a candidate peak. The next two loops would be used to index the neighborhood around that point to make comparisons. Only if a location succeeded in being greater than each neighboring point would it be labeled as a peak. We use the parameter `gs` to specify how far to look in each time and frequency direction.

Matlab provides a command that will save us coding time and run time, but we have to be a little bit clever to use it. The command is `circshift`. This command shifts a matrix by a specified amount vertically and horizontally. It's a circular shift because the entries that fall off the edge after the shift are wrapped around. This wrapping is actually not ideal, but it doesn't affect the outcome much either, so we will just ignore it.

Consider the following example to see how to use `circshift`:

```
CS = circshift(S, [0,1]);
P = (S > CS);
```

These two lines return a Boolean matrix `P` with entries 1 for the positions in `S` that are greater than their neighbor immediately to the right. This has the effect of comparing each position in the matrix to one of its neighbors, without having to explicitly loop through the entire matrix.

The provided code loops through all horizontal and vertical shifts within distance `gs`. Use `circshift` to make the comparison to its neighbor, as a single matrix operation. Only locations in the matrix that survive each round of the comparison should remain as 1 in the peak matrix.

Plot the peaks using `imagesc`. Change the color-map using the following command:

```
colormap (1-gray);
```

which will display the entries where there are peaks as black pixels and the rest of the matrix as white pixels.

Try several values for `gs` and plot the constellation map. Note the effect of changing `gs`. Compute the $\boxed{\text{M2}}$
constellation map for `gs=4`, i.e. 4 points in each direction. Calculate how many peaks there are and record
your answer. How many peaks are there per second on average? $\boxed{\text{D1}}$

If time permits, you can try to locate time-frequency troughs instead of peaks. Do you think fingerprinting
the song using peaks provides any inherent advantage over using troughs? $\boxed{\text{D2}}$

2.2.4 Thresholding

We want to use only the larger peaks. Why? (Hint: Think about the quality of the clip we would like to
identify.) $\boxed{\text{Q3}}$

To get rid of small peaks, we will set a threshold and get rid of peaks that don't surpass the threshold in magnitude. The code is already written for this. You simply need to assign a value to `threshold`. Try different values to see what happens to your peak constellation.

One reasonable way to determine the appropriate threshold to use is to target a certain number of peaks per second. That way, the threshold is adaptive. If the recording is louder, the threshold is also louder to achieve the targeted number of peaks. See how close you can get to 30 peaks per second by adjusting the
threshold. What threshold did you use? $\boxed{\text{Q4}}$

Using the threshold from above, display the constellation of peaks as before. Comment on the distribution
of peaks. Is it uniform? Are they closely packed? If so, is this a good thing? $\boxed{\text{M3}}$

Code has been provided to find a threshold to achieve 30 peaks per second. Uncomment this code now. $\boxed{\text{D3}}$

There are perhaps better ways to adaptively threshold. For example, a fixed threshold through the duration of the song might not be appropriate. Also, high frequencies should maybe have a lower threshold. Feel free to play with this once you have a complete Shazam system working.

2.2.5 Check results

There is code for an optional plot at the end of the template. Remove or comment all previous plots in function. To enable this plot, change the variable `optional_plot` to 1. This shows the spectrogram with blinking dots where the peaks are. Make sure this looks correct. It will blink a fixed number of times, after which you can zoom in if necessary.

2.3 Find peak pairs

We have provided the command `convert_to_pairs` which takes a matrix of peaks and returns a table of pairs that are close in both time and frequency, as shown in the introduction and illustrated on the next page. Experiment with the parameters of this function. Enable the plot in this function and show the results
to the TA. $\boxed{\text{M4}}$

The code finds pairs by considering each peak and looking for other peaks within a designated window located relative to it. During the search, we limit the number of pairs that we accept by the parameter `fanout`. The code is written to scan through the window column by column, accepting the first pairings that it finds. You might be able to improve performance by changing the window or changing the way it is scanned. For example, some people like to set a minimum time separation for the scan window.

2.4 Train database

Two other m-files are provided. The first one, `add_to_table.m`, edits a global variable called `hashtable`. We'll discuss that more in part 2 of the procedure. The other one, `make_database.m`, is a script that searches for all MP3 files in a designated folder and processes them if they are not already in the database.

Make the appropriate changes to `make_database` so that it properly processes the music files. It should call each of the three other functions discussed so far.

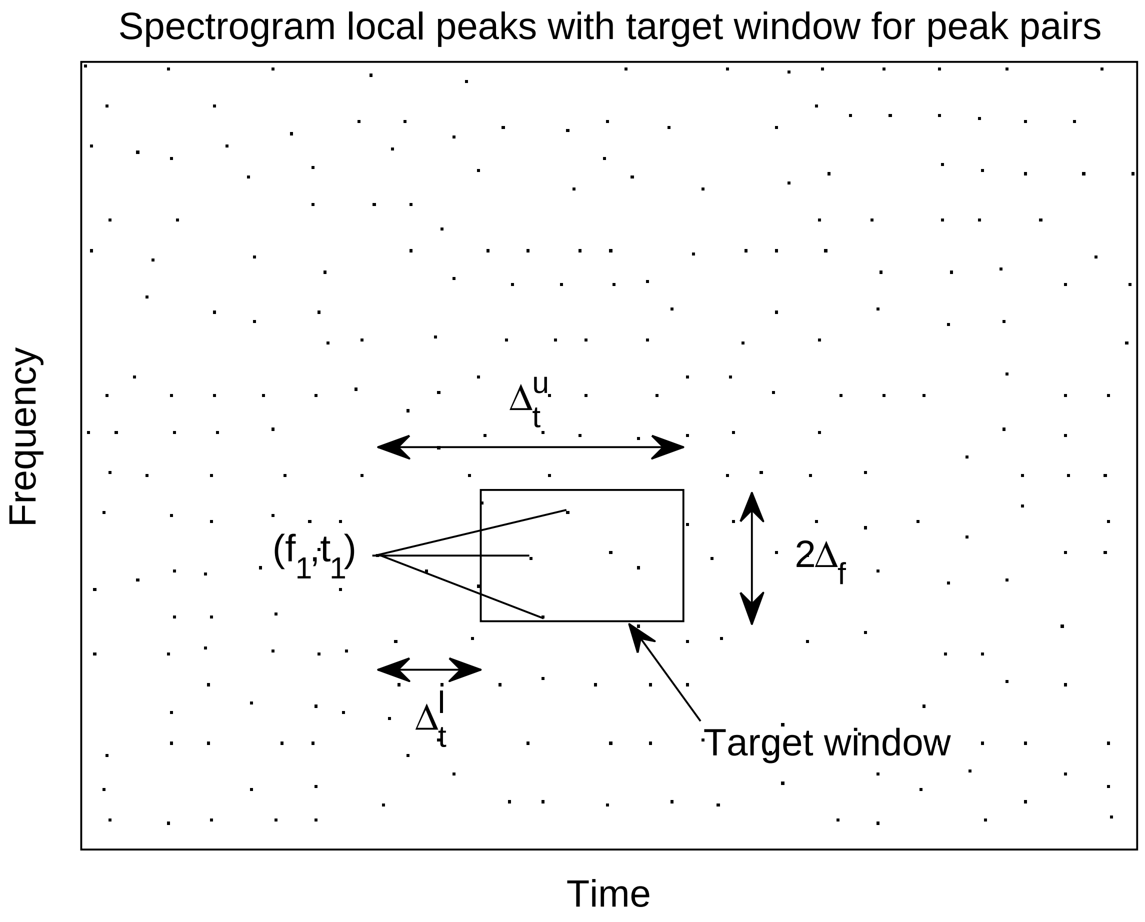

Figure 1: Peak pair identification

3 Lab Procedure - Part 2

This week we will build the part of Shazam that identifies a segment of music, using the database that we trained in the previous part. The main steps of the algorithm are the following:

1. Load `HASHTABLE` and `SONGID` that were created by `make_database.m` in part 1.
2. Prepare a clip of music for identification.
3. Extract the list of frequency pairs from the clip.
4. Look up matches in the hash table, calculate time offsets, and sort them by song.
5. Identify the song with the most matches for a single consistent timing offset.

We now discuss in more detail.

3.1 Match clip to song

The song matching will be accomplished in a function called `match_segment.m`. A template has been provided. This function accepts a sound segment and a sampling frequency as arguments, and outputs the song that best matches as well as a confidence level. The variables `hashtable` and `songid` must exist as global variables for this function to work properly.

You will need a 5-10 second segment of one of the songs in the training set in order to test your code for the following subsections.

3.1.1 Extract fingerprint

Begin by using the `fingerprint` function created in the previous part and the `convert_to_pairs` command to form a list of the peak pairs from the sound clip.

3.1.2 Recover matches from hash table

For each peak pair from the clip, we will find a list of potential matches in the hash table. A potential match is any peak pair from the training process where the two frequencies are the same and the time difference between the frequencies is the same. Notice that `simple_hash` was provided and was used previously in `add_to_table.m`. Use the frequencies and the time difference of the peak pair $(f_1, f_2, t_2 - t_1)$ as inputs to the hash function, exactly as was done in `add_to_table.m`. Then extract the two lists from the hash table saved at the location provided by the hash function. These lists are stored as vectors. Some of this code is provided for you.

The two lists that have been extracted contain potential matches for the peak pair. The first list contains the song ID numbers for each potential match, and the second list expresses the times `t1` where the matches occurred in the training data. Recall that the same song may contain the same peak pair at different times `t1`, so the same song ID number may appear multiple times in the extracted list.

Now convert the timing list to a list of timing offsets by subtracting the time `t1` that the peak pair occurred in the clip. This list of offsets is what we will save. Why do we care about the offsets rather than the timing vector itself? D4

We need to collect the lists of potential matches from each peak pair in the clip. These potential matches also need to be separated into different lists for each song in the database. The array called `matches` is defined for this purpose. You can separate the lists using the `find` command, which returns a list of indices for the non-zero entries of a vector. Usually this is used in conjunction with a Boolean expression For example, `find(x==3)` returns the list of indices where $x = 3$. Notice that `y(find(x==3))` returns a list of the values of y at the locations where $x = 3$.

Enable the optional plot in the code to see a graphical display of the extracted data, and show this to the TA. This shows one plot for each song in the database. This graphic is a histogram of the offset vector. M5

A histogram counts the number of occurrences of each value in a list, ignoring the order of the list. The heights of this bar graph indicate the number of occurrences of a particular value. Do you see what you expected? How can this be used to identify the correct song in the database? $\boxed{\text{D5}}$

3.1.3 Identify song

Find the song that has the most occurrences of any single timing offset. This is most easily done by looping through each song and using the `mode` command. The `mode` command returns two values: The first is the most common number in the vector; and the second is the count of the occurrences of that number.

Declare the winner to be the song with the most matches at a single time offset, and have `match_segment` return the index of the song as the variable `bestMatchID`.

3.1.4 Confidence

The function `match_segment` returns a second variable that indicates the confidence level of the song matching decision. Discuss at least one idea for measuring the confidence. Implement this in code if you'd like. Otherwise, just set the variable `confidence` to 1. $\boxed{\text{D6}}$

3.2 Test Shazam

We've provided the file `myshazam.m` for testing and using your Shazam system. It first loads the `hashtable` and `songid` variables that were saved during the training process, if they are not already in the workspace. Then this function provides two options. It will either select a random segment of a song from the training set, or it will record sound from the microphone. Insert the appropriate code toward the end to use your `match_segment` function to match the song. Demonstrate to the TA that everything is working. $\boxed{\text{M6}}$

What is the accuracy of your program using the following tests? Use at least ten random clips of length 10 seconds. Repeat for length 5 seconds. Notice that the code allows you to add artificial noise to the clip. Set the signal-to-noise-ratio (SNR) to 0dB and repeat the above experiments. The meaning of SNR is the following: $\boxed{\text{Q5}}$

$$\text{SNR}_{dB} = 10\log_{10}\frac{P_{\text{signal}}}{P_{\text{noise}}},$$

where P_{signal} is the power of the signal and P_{noise} is the power of the noise.

3.3 Possible improvements

This Shazam algorithm can be optimized in a variety of ways. To begin with, all of the parameters in the code can be adjusted (including even the sample rate). Keep in mind that some of these adjustments may affect the run-time of the program.

Additionally, more significant changes can be made. For example, the matching process could be based on triples of peaks or single peaks rather than pairs. Another idea is worth thinking about, though it may not be crucial: In `match_segment`, choosing the song match by only comparing the mode for each song may not be the optimal way. Some songs may have a longer list of potential matches, perhaps from being a longer song. The mode could be compared somehow to the length of the list. A third idea would be to replace our homemade hash function with an industry hash function. Matlab code for these can be found online. In our setting, the goal of the hash function is to evenly distributed the potential matches across our allocated table size.