**CS 5660 Signal Processing**

Homework 2

November 5, 2015



Angel Wong ([axw3@cornell.edu)](mailto:axw3@cornell.edu))

Halil Akin ([ha338@cornell.edu)](mailto:ha338@cornell.edu))

Table of Contents

[I. Aligning Audio Signals – Attempt #2 2](#_Toc434399440)

[A. Using Line Fitting 2](#_Toc434399441)

[1) Anchor Points & Target Zone Representation 3](#_Toc434399442)

[2) Hash Table Representation 4](#_Toc434399443)

[3) Matching Hash Tables 5](#_Toc434399444)

[4) Matching Audio Segments: Line-fitting scatterplots 7](#_Toc434399445)

# Aligning Audio Signals – Attempt #2

In the first homework assignment, we attempted to align audio signals by examining the Short Time Fourier Transform of our audio signals. By determining the peak time-frequency pairs at each time t and correlating these peaks across multiple audio samples, we are able to make assumptions about audio signals with high levels of cross-correlation. For this assignment, we will once again examine the peaks of the transformed signal, but instead implement a different algorithm for audio aligning that emulates how Shazam matches audio signals.

## Using Line Fitting

For Shazam to storing full songs for every song ever created in their database, that would be very space-intensive. Instead, Shazam only keeps track of important sounds in the song, the time at which these intense points happen in the song, and the frequency at which they occur. These particular “peaks” are found by examining the audio signal’s original spectrogram. The peaks are generated by finding the highest value in every 20-by-20 grid around each (t,f) point in the spectrogram.

By leveraging the same functions from homework 1, we are able to recreate the peak maps for audio signals 1, 2, and 3. As you may remember, a peak map looks as follows in Figure 1.



**Figure 1:** Peak map for audio file 1.wav using functions from HW1

### Anchor Points & Target Zone Representation

Despite the simplified representation of the spectrogram as a peak map, there needs to be an efficient method by which to index through these peak maps. We do this by pairing each peak with a set of peaks in a region to the right of it on the peak map.

This “target zone” for each of these “anchor points” at ) is a rectangle with a width time gap of *T* and a height frequency gap of F, as shown in Figure 2. This means that the target zone’s x-axis coordinates are , and it’s y-axis coordinates are . We only care about the top N peaks in this target zone.



**Figure 2**: Anchor point with its associated target zone[[1]](#footnote-1)

The **target\_zone\_peaks** function creates a top\_peaks array that captures each anchor\_point and assigns it a new target\_zone array, which will store all peaks in the anchor\_point’s target zone. For each anchor\_point we iterate through, we evaluate if it is in the target zone of the anchor points preceding it, and if so, add it to that anchor point’s target\_zone array. We make use of the fact that peak\_list is ordered by a peak’s increasing time and frequency value to do so. The target zone peaks must appear later in the peak\_list than the originating anchor point.

Our selection of these variables for target\_zone\_peaks: N, F, and T are fairly arbitrary, but we have noticed that smaller target zones (smaller F and T) result in less noisy results later on. In order to define the best N, we decided to take the N closest peaks by Euclidean distance.

### Hash Table Representation

Now that we know the top N peaks in an anchor\_point’s target zone, we further simplify the representation by making use of a hash function that represents each pair of anchor point and target zone peak as a tuple:

We can more easily search for a match by leveraging this hash table, but first requires the selection of a hash function, which can vary from implementation to implementation. Ultimately, any hash function selection should rely on the frequencies of each peak (anchor point, target zone peak) and the time gap between the peaks.

We have used a linear hash function, as provided in Piazza, where the

***hash\_value =***

***frequency of the anchor point + difference in frequencies + difference in times***

We calculate this hash\_value for every pair and store it along with the time of the initial anchor point in the **hash\_table** function .

### Matching Hash Tables

Next, we conduct this hash\_table function on each of the audio files we seek to compare. We can compact the two hash\_tables by matching entries with the same hash values in each hash\_table, and keeping track of time values associated with each hash\_value in each of the audio file in **match\_hash\_tables()**. In other words, each entry of the table would look as follow:

{<hash\_value>, [{time values from audio file 1}, {time values from audio file 2}]}

We can further manipulate our hash\_table into an array of 2D pairs in match\_times\_2d\_pairs, where:  
 , and

.

for all values t\_1 in the first set of the second array in match\_hash\_tables() and t\_2 in the second set of the second array of match\_hash\_tables().

When we plot this list of 2D pairs where audio file 1 time values are on the x-axis, and audio file 2 time values are on the y-axis, which we do through **plot\_times()**. The scatterplots for each pair of audio files in 1, 2, and 3 are visible in Figure 3.

If there is a match between two audio files, we should see the formation of a line on this scatterplot. All of the subfigures in Figure 3 do not seem to show any obvious lines. In comparison, plotting an audio file against itself will show the formation of a line, as seen in Figure 4.

The results from Figure 3 align with the patterns we expected to see given our weak correlations from HW1. Even in the prior assignment, we did not see any strong cross-correlation between any two audio files we tested.

|  |
| --- |
| **Figure 3.1**: Scatterplot of two audio files: 1 and 2 |
| **Figure 3.2**: Scatterplot of two audio files: 1 and 3 |
| **Figure 3.3**: Scatterplot of two audio files: 2 and 3 |



**Figure 4**: Presence of a line in scatterplot of audio file 1 with itself

### Matching Audio Segments: Line-fitting scatterplots

As alluded to in the prior section, seeing a line in the scatter plot indicates a high correlation. This is because we seek to match two audio files by finding a sequence of hash values that appear in both audio files, which may start at different times in each audio file.

Not all lines are so easily apparent as they are in Figure 4. To better fit a line to the scatterplot, there are many line-fitting methods that can be leveraged. Specifically, for this assignment, we will go into depth on two of them: Hough transforms and RANSAC.

#### Hough Transforms

The basic premise for a Hough transform is to record all possible lines that can exist for a point . We can then transform the linear representation of each of these lines into a point in the Hough space – where the axes are .

As you may remember from geometry, a line equation of form also has a polar representation . We prefer this representation over the classic slope-intercept model because the latter takes infinite values and is undefined for vertical lines.

In this polar representation, the family of lines that go through will result in a sinusoid in the Hough space. If we do this same operation of converting a point’s family of lines into sinusoids for all points in the image, we expect to see these curves to intersect in the plane at a set of ’s in the Hough space. Each of these points in Hough space constitute a line in the (x, y) plane that best fits the points from our original scatter plot.

To do so, we must first clean our scatterplot and remove the axes of our original plots in Figure 3 and 4. We do so in **plot\_times\_img()** and end up with a figure as seen in Figure 5. (In this example, we have also replaced the blue ‘o’s’ with black ‘.’s’.



**Figure 5**: Cleaned scatterplot for audio file 1 against itself

Our Hough implementation in **hough()** takes in these cleaned images and returns an image of the Hough space for our cleaned scatterplot.

We find that in our sample where audio file 1 is compared against itself, a very clear intersection can be found in the transform map of Figure 6.



**Figure 6**: Hough transform of scatterplot from audio file 1 against itself

The intersections for these Hough transforms are less visible for all of the other scatterplots that compare pairs of audio files 1, 2, and 3, as seen in Figure 7. This aligns with what we’ve come to expect – that there are no obvious lines in our original scatterplot and that our audio files do not seem to have much in terms of audio sample correlation.

|  |
| --- |
| **Figure 7.1**: Hough transform of scatterplot from audio file 1 to audio file 2 |
| **Figure 7.2**: Hough transform of scatterplot from audio file 1 to audio file 3 |
| **Figure 7.3**: Hough transform of scatterplot from audio file 2 to audio file 3 |

That being said, this is a much better method to match audio segments than our prior method in HW1. For instances in which audio segment matches are clearer as in Figure 6, we can translate the intersection point of the Hough transform map back into an optimal line and demonstrates that there is a true correlation between two audio samples.

#### RANSAC line fitting

Another method of line fitting is RANSAC. What RANSAC does is take the standard linear model only on a set of “inliers”. Much of the problems that come with linear regression fitting is because outliers will skew the fitting of the line. RANSAC randomly selects a group of points, fits a model to the selected group, and then finds the inliers of the computed model. It calculates the number of inliers by finding out the number of points that lie within a threshold of the line.

If this number of inliers is large enough, the model is recomputed using only those inliers and the number of inliers is computed for this updated model. The best RANSAC model is the one with the largest number of inliers. There are many ways to determine when to stop RANSAC, including setting a minimum number of inliers needed or by reaching a maximum error threshold.

Our implementation of RANSAC in **audio\_ransac()** leverages the ransac() function from the scikit-image-processing library. This ransac() function requires you to provide the minimum number of data points needed to fit a line to, and the preferred threshold for a data point to be considered an inlier. This function of ransac() allows you to set a maximum number of trials as a stopping criterion for the algorithm, which we have done here, as opposed to providing a special stop criteria.

We expect to see a line fit to the best possible set of points that ignores the rest of the noise in the plot. Our control of audio file 1 against itself shows a line that is perfectly fit against the diagonal in Figure 8. This is to be expected for audio files that have not been changed and the intense sounds occur at the same times.



**Figure 8**: RANSAC fitting of audio file 1 against itself

Meanwhile, when we do RANSAC fitting against two separate audio files as seen in Figure 9, the RANSAC algorithm is unable to generate a decent line-fit, but makes the best possible guess, even if the fitted lines are horizontal. (A horizontally fit line would not make sense unless one audio file is repeating the same sound that occurred at one point of the other audio file). Once again, this matches our expectations from Homework 1 where we could not find strong correlation across audio files.

|  |
| --- |
| **Figure 9.1**: RANSAC fitting of audio file 1 and 2 |
| **Figure 9.2**: RANSAC fitting of audio file 1 and 3 (no line was fit here) |
| **Figure 9.3**: RANSAC fitting of audio file 2 against audio file 3 |

# Dynamic Time Warping

1. http://www.soyoucode.com/2011/how-does-shazam-recognize-song [↑](#footnote-ref-1)