1 Introduction

In this project, we created a **Shazam-like** app, called **GiMY – Give Me the Song**. The program is designed to recognize any song that is present in its database when any 10-second interval of the song is played in the environment. Our test database consists of 285 different songs. The project is inspired by the working principle of the app Shazam [1]. Ideally, such applications search in their databases using multiple servers with great computational power which we are not able to satisfy by employing our personal computers. However, one should note that the number of songs can be varied without any loss of generality since the algorithm itself remains the same independent of the size of the database. Our program is designed via MATLAB and is also supported by an interactive GUI application.

2 Algorithm

The algorithm can be divided into four essential operations.

2.1 Creating Constellations

Each song is identified and compared with each other for a possible match based on their **fingerprints**. The fingerprint of each song consists of a set of **hashes**. The concept of the hash will be explained in the later stages of the report. However, at this point, it is important to note that we need to have the frequency components of the song with peak amplitudes to create its hashes. For this purpose, we get the spectrogram of each audio. The spectrogram of any audio shows the PSD amplitudes of the frequency components present in the audio at each time instant by taking the DFT of time-overlapping windows. The window we have chosen contains $W = F_s \times w_t$ samples where w_t is the time duration of the window. The overlap is chosen as $0.6W$, this means we have points on the spectrogram at each time instant $t = 0.4 \times k$ until the end of the audio file where $k \in \mathbb{N}$. One of the example spectrograms can be seen in Fig.1.

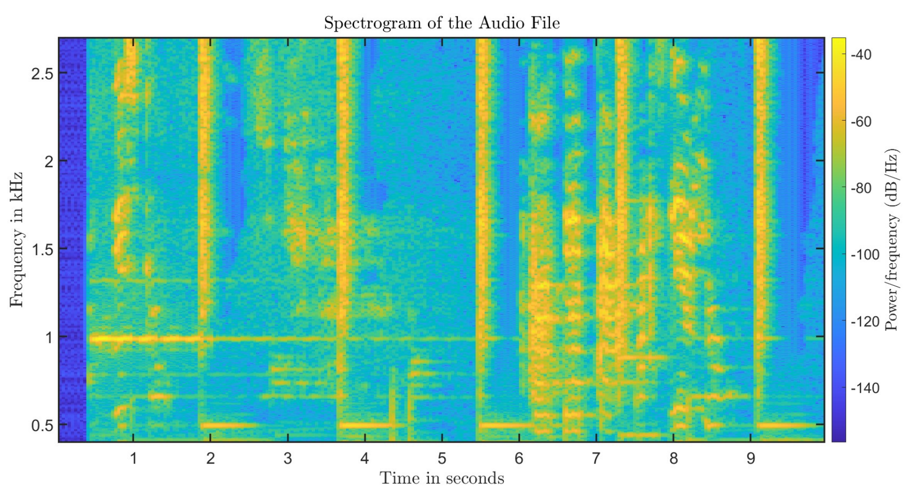

Figure 1: Spectrogram of the 10-second audio

For our purposes, we only need the magnitudes of the spectrogram. As a result, we have a spectrogram matrix of the audio that represents the time axis along its columns and frequencies along its rows and therefore any cell of the matrix holds the magnitude of the corresponding frequency component at that specific time. Another important point is that for improved efficiency, we should only consider the relevant frequencies which are the most likely to come up in songs. Based on our observation during tests, it is unusual for a song to have frequency components larger than 2700 Hz. Furthermore, the low-frequency components are heavily affected by the noise and therefore such components in audio are often unreliable for recognition and detection purposes. Therefore, we have only considered the frequencies from 400 Hz to 2700 Hz while creating the fingerprints.

Now, it is time to select the peaks of the spectrogram matrix as the next step of creating fingerprints of audio. We conclude that a point in the matrix is a peak if and only if it is larger than its neighbors. For this, we use local windows of size 15×15 (225 elements) such that each element of the spectrogram matrix has its own local window centered at the element itself. *Note that these local windows are different than the window used in spectrogram which meant the number of samples during a DFT operation.* Then, an element is a peak if it is larger than all the other elements encapsulated by its local window. For comparing each of the elements to their neighbors, an efficient way is to make use of the circular shifts. For each element in the matrix, we start comparing starting from the bottom right up to the top left of the local window. At each iteration, we update the elements of a matrix called "mask_peaks" which has the size of our spectrogram matrix and is initialized with ones at each element. At the end of circular shift operations, any element of the "mask_peaks" is one if and only if it is larger than all of its neighbors otherwise, it is set to zero. When we multiply this matrix with the spectrogram magnitude matrix, we effectively select the peak element at each window. We call this collection of peak points as a "constellation" and therefore this above-described operation of selecting the peak values is done in the file "getConstellation.m". The constellation of the same audio file given in Fig.1 is shown in Fig.2.

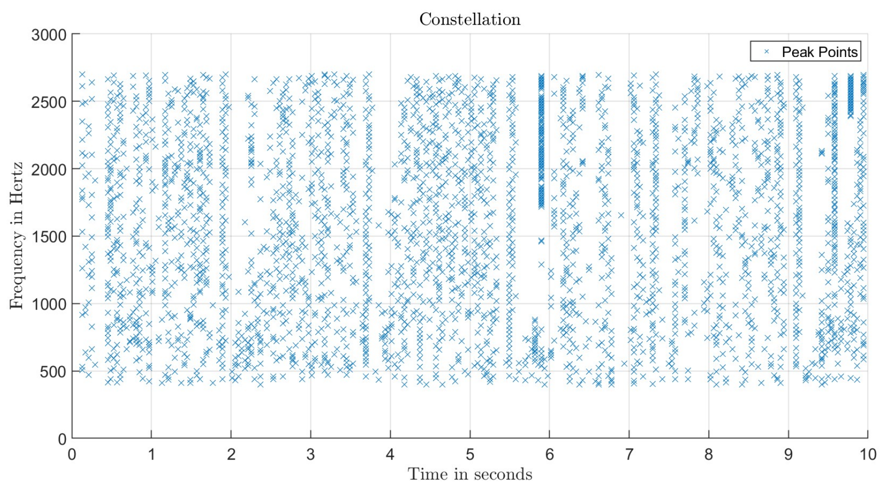

Figure 2: The constellation of the 10-second audio

2.2 Reducing the Number of Peaks in a Constellation

However, the total number of peaks in the constellation is still too high and it is unpractical to work with such a high number of peaks. Furthermore, the distribution of peaks is non-uniform in its present form, and therefore, without a uniform distribution of peaks, each time interval of the audio would not be given the same importance when creating its fingerprint. Therefore, we take the constellation that we have just created and divide it into many time slices. At each slice, we find the highest 30 peaks and leave them as they are except for the last slice. Since the last slice can be shorter than the other slices, the number of surviving peaks are scaled with $30 \times \frac{\text{length of the last slice}}{\text{slice length}}$. All the other peaks at each slice are set to zero. This way, we ensure the uniformity of the peaks and also avoid having more hashes than needed per audio by creating an audio fingerprint. This described operation is done in the file "getReducedConstellation.m" and the matrix we obtain at the end that includes fewer peaks than the original constellation matrix is the reduced constellation matrix as shown in Fig.3.

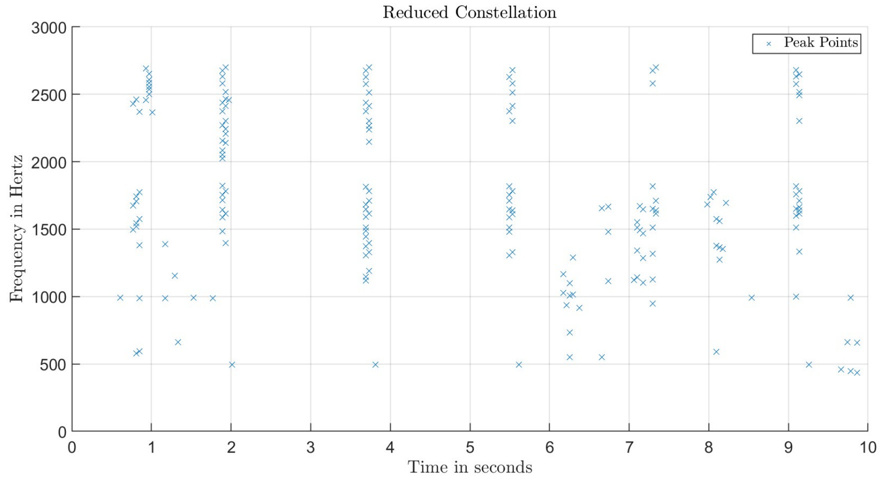

Figure 3: The reduced constellation of the 10-second audio

2.3 Creating Hashes

As mentioned, the fingerprint of a song consists of a series of hashes. A hash is a vector, which describes the relative position of two peaks in the *reduced* constellation matrix. For each hash, one of these points is the *anchor point*, relative to which many other hashes will/may be calculated. We have selected all the peaks in the reduced constellation as anchor points, one by one. Each anchor point has its own *target zone* and from this anchor point, multiple hashes will be created. A particular hash corresponding to this anchor point will be formed for each peak present in the target zone, if there are any.

For instance, let us say that one of our peaks is located at the location (i, j) in the reduced constellation matrix and we want to treat it as an anchor point to form the hashes of this anchor point. Furthermore, assume that the point (i, j) corresponds to the frequency f_1 and time t_1 respectively in our reduced constellation matrix. We now have to consider the target zone of this anchor point, in which we will search for the other peaks. We defined the width of the target zone as 40 and its height as 50. In addition, the target zone begins at 5 elements to the right of the anchor point. Therefore, the target zone of an anchor point at the (i, j) of the reduced constellation is a rectangle, the four corners of which are located at $(i + 25, j + 5)$, $(i + 25, j + 45)$, $(i - 25, j + 5)$ and $(i - 25, j + 45)$. We look for non-zero elements (in other words, peaks) in this zone. Let us assume that an arbitrary peak in the target zone is located at a point that corresponds to time t_2 and frequency f_2 of the reduced constellation. Then, the **hash** indicating the relative position of the anchor point and this peak in its target zone in frequency and time is defined and saved as $[f_1;\ f_2;\ t_2 - t_1;\ t_1]$. In other words, each hash stores the frequency of its anchor point, the frequency of the specific peak in its target zone, the time difference between the two, and the absolute time of the anchor point. The purpose of storing the absolute time of the anchor point for each hash will be explained in a later section of the report. The procedure of hash formation is illustrated in Fig.4.

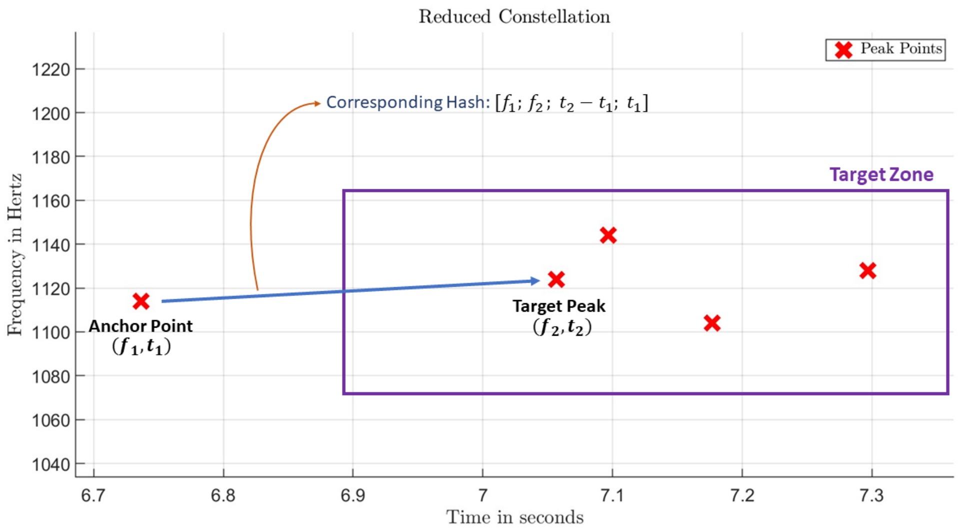

Figure 4: Creating a single hash

2.4 Searching

At this point, we must have generated a list of hashes for each song in the database. The data is kept as a $(5 \times N_h)$ matrix in a text file where N_h is the total number of hashes. The rows indicate the values for [Track ID; f_1; f_2; Δt; t_1] respectively. A sample song of duration 10 seconds is given as input to the GUI. A constellation is created for the sample, then it is reduced and the its hashes are created. An efficient searching algorithm is taking each hash of the sample song and subtracting it from the data matrix. The second, third, and fourth rows of the difference matrix corresponding to [$f_1 - f_1'$; $f_2 - f_2'$; $\Delta t - \Delta t'$] are compared to three threshold values separately. If all the difference values are below their respective threshold values, the hashes are matching.

We stored the matching hashes and from now on, we only care about the absolute time differences and track IDs of the matching hashes. If the sample audio belongs to a particular song in the database, we must observe a prominent value for the absolute time differences among their matching hashes. Even though the audio and the corresponding song in the database have the same hashes at possibly different absolute time positions, overall matching hashes will still have the same positions relative to each other. For instance, let us assume we have two matching hashes from the sample audio H_1' and H_2' with absolute time values t_1' and t_2'. Also assume that we have the corresponding hashes from the correct song H_1 and H_2 with absolute time values t_1 and t_2. Then, the absolute time differences must satisfy $t_1 - t_1' = t_2 - t_2' = C$ where C is a constant. This result can be expanded to any number of absolute time differences.

Using a histogram for the absolute time differences is feasible to detect the prominent constant C in this case. Even though the sample audio will have matching hashes with all other songs in the database, the absolute time differences will be mostly arbitrary. Consequently, we will only observe a prominent time difference among others if the recorded audio indeed belongs to a particular song in the database. The histograms of time absolute time differences of matching hashes are shown in Fig.5.

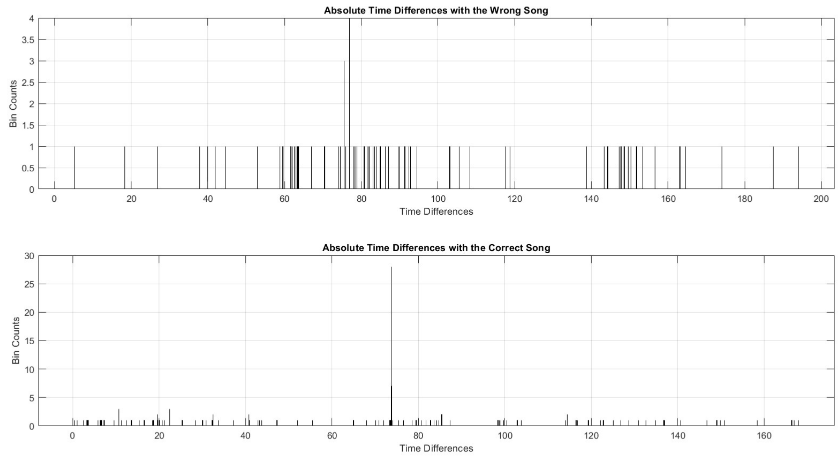

Figure 5: Comparison of absolute time differences of matching hashes of the actual song and the second-best predicted song in the database

3 Application

The app has a “RECORD” button and when the button is pressed, the program listens to the environment through a microphone and records the audio for 10 seconds. The “RECORD” button is accompanied by a lamp that is white (non-active) in standby mode. When the button is pressed and the program is recording the audio, it flashes green. When the recording phase is over and the program is searching in the database to find the matching songs, it flashes red, signaling that the recording is over, but the button is not yet available for another push. After the search is over and the results are listed, it turns white again as the program is now in standby mode and recording/searching for another song is possible. The app also has a “Progress” text bar that informs the user about the state and/or result of the program. When in standby mode, it displays: “Waiting for a song”. During the recording and searching, it informs the user by displaying “Recording...” and “Processing...” respectively. When the search is over and there is no song in the database similar to the recording, it displays: “The song does not have a match in the database.”. Lastly, when there is at least one song similar to the recording, it displays: “The song is successfully identified.”. In such a case, the app also displays the names of the five most similar songs and their corresponding probabilities of being the correct match of the recorded song.

4 Difficulties and Solutions

4.1 Finding a Peak in a Local Neighborhood

To check whether an element is the largest in its local window, we have come up with the idea of utilizing circular shifts both in horizontal and vertical axes to compare with each element of the window. This method worked better than nesting for loops and reduced the computational overhead.

4.2 Slow Search

Instead of looking at every song one by one to find a match with the hashes in the audio, we hold hashes of all songs in a huge matrix and perform subtraction operations on it, and then compare the differences with thresholds to find matches. This difference operation also helped us during the detection of the prominent absolute time difference value of the matching hashes.

5 Numerical Results

5.1 Accuracy

We have achieved 92.63% accuracy in a database consisting of 285 songs from various genres. We used our microphones to record a 10-second segment of the songs. These audio samples were recorded in EE Lounge which is always crowded and noisy as it is a place for group studies.

5.2 Rejecting the Negative Samples

We used 40 sample audios that were not in the database. Expectedly, GiMY did not find any match for the songs and inform the user about the absence of the song in its database. We set a threshold of 30% match percentage and if the matching percentage of the best matching song that GiMY found is below this threshold, it is concluded that this song is not from the data. The match percentage is calculated using the top 5 songs returned by GiMY. The **match percentage** is defined as *the ratio of the number of time-matching hashes of a song to the total number of the time-matching hashes of the top 5 matching songs.*

5.3 Performance in a Noisy Environment

While recording the sample audio we have also talked next to the audio source. If the chat sounds overwhelmingly suppress the sample audio, even though matches are found with the true song, the matching percentage mostly drops to values under the threshold. However, if the song is distinctive enough, we have observed that GiMY still finds the song, but of course with a decrease in the match percentage.

5.4 Speed

5.4.1 Finding a Match

The algorithm returns the most matching songs in 5 seconds on average. We observed that the inference time was around 4.5 seconds when the data consisted of 100 songs. Increasing the number of songs did not result in excessive overhead such that the operation time increases slower than a linear fashion with respect to the song number in the database.

5.4.2 Creating the Data

Since the songs in the database must be stored with their corresponding hashes, all the songs must go through the same process as the sample audio. This process takes a lot of time as the number of songs increases. However, this is a one-time process that we do not repeat again and again unless we change the parameters of the hashing process and want to create a new database accordingly.

6 Conclusion

We have created GiMY as our term project for EE 473 course. GiMY decides which song is played by using the fingerprints of the database and the sample audio. GiMY also comes with a graphical user interface. The user can record a sample audio using the GUI and GiMY returns the matching song after listening to any 10-second interval of the song. It takes only 5 seconds for GiMY to give you the top matches if the actual song is in the database.

Appendix A Code

A.1 Main

```
%% Creating the database if Data.txt is not in the folder
%createDatabase();
%% Reading the .txt files
global data
global songNames
[data,songNames] = loadData();
%% Running the GUI
run('GiMY_app.m');
```

A.2 createData

```
function [] = createData()
% Generates the hashes of the songs in the .mp3 format. The hashes are then
% written into 'Data.txt' and names of the songs are written into
% 'SongNames.txt'

    % Gets names of all .mp3 files in current directory
    tic
    audioInDir = dir('*.mp3');
    audioNames = {audioInDir.name};
    hashes = []; % To hold all the hashes of stored songs
    filenames = cell(1,length(audioNames));
    if(isempty(audioNames))
        disp 'No Audio Files in Current Directory'
    else
        %Audio files are present in the directory
        for i=1:length(audioNames)

            filename = char(audioNames(i));

            hash = hashing(filename);
            % Append song ID at the beginning.
            hashes = [hashes, [repmat(i,[1,size(hash,2)]);hash]];
            filenames{i} = filename;
        end
         % Write the songs whose hashes extracted to a single matrix.
         writematrix(hashes,'Data.txt','Delimiter',' ');
         % Keep the name of the songs in a txt file.
         writecell(filenames,'SongNames.txt','Delimiter',',');
    end
    toc
end
```

A.3 loadData

```
function [data,songNames] = loadData()
% Loads Data.txt in which the hashes for the whole songs in the database
% are located and SongNames.txt in which the names of the songs are located
    .
    tic
    global data;
    global songNames;
    datafile = 'Data.txt';
    namefile = 'SongNames.txt';
    data = readmatrix(datafile,'Delimiter',' ');
    songNames = readcell(namefile,'Delimiter',',');
    toc
end
```

A.4 getConstellation

```
function [peak_magnitudes, spec_f, spec_t, new_fs, window] =
    getConstellation(song_file)
% Takes a .mp3 or .wav file and returns the peak spectrum points.
% Inputs:
```

```
% song_file                  An audio file containing the song.
% Outputs:
% peak_magnitudes            Spectrogram matrix containing peak magnitude in
%                            each local neighbourhood.
% spec_f                     Frequencies
% spec_t                     Time instants
% new_fs                     New sampling rate
% window                     Number of samples per window of spectrogram



new_fs = 8192; % 8192 is new sampling rate of signal. Corresponds to the highest note on a piano.


[song, Fs] = audioread(song_file);
% Since the songs read are stereo, we average the two columns and get a one
% dimensional column vector, we later convert it to a row vector.
if length(song(1,:))>1
    song_mono = ((song(:,1)+song(:,2))./2)';
else
    song_mono=song';
end
song_mono = song_mono - mean(song_mono); % Remove DC component

% Resampling the mono song so that the sequence length is decreased and
% high frequencies are excluded. This results in not taking high
% frequencies into account and might reduce the probability of detecting
% a song correctly. However, we also reduce time and memory complexity.
song_rs = resample(song_mono, new_fs,Fs);

% We also do not want to account for low frequencies since it reduces
% reliablity.
frequencies = 400:2:2700; % We can represent up to newfs/2 frequency.

timePerWindow = 0.1; % Window time duration for spectrogram

window = round(timePerWindow*new_fs); % Number of samples per window
nOverlap = round(0.6*window); % Number of samples overlapping while sliding the window

% Spectrogram takes window-point dft at every slice of the song.
% spec_f is the corresponding frequencies to row indices of the spec_song
% spec_t is the corresponding time instants to column indices of the
% spec_song
[spect_song,spec_f,spec_t] = spectrogram(song_rs,window,nOverlap,frequencies,new_fs,'yaxis');

spec_magnitude = abs(spect_song); % Magnitude of the spectrogram

%% Finding Peak Magnitudes

% One side length of the local window. The local window will be a
% (len_localWindow + 1) x (len_localWindow + 1) window. Each element in
% spec_magnitude will be compared with (len_localWindow + 1)^2 - 1 elements
% around it. If a specific element is greater than all of others, the
% mask_peaks matrix will still contain 1 in the corresponding index,
% otherwise we update it to 0.
len_localWindow = 7;

mask_peaks = ones(size(spec_magnitude)); % The boolean mask containing ones, will be updated.

% For comparing each of the element to neighbors, an efficent way is to
% make use of the circular shifts. For each element in the spec_magnitude
% we start comparing starting from the bottom right up to top left of the
% local window and update mask_peaks.
for shift_hor = -len_localWindow:len_localWindow
    for shift_ver = -len_localWindow:len_localWindow
        if(shift_ver ~= 0 || shift_hor ~= 0) % Avoid comparing to self
            mask_peaks = mask_peaks.*( spec_magnitude > circshift(
```

```
    spec_magnitude,[shift_hor,shift_ver]) );
        end
    end
end
% The actual values of peak magnitudes will be used for choosing local
% maximum peaks. The peaks will be further reduced.
peak_magnitudes = mask_peaks.*spec_magnitude;
```

A.5 getReducedConstellation

```
function [reduced_constellation, spec_t, fs] = getReducedConstellation(
    peak_magnitudes, spec_f, spec_t, fs, window)
% Among the peak_magnitudes, this function selects top N peaks from each
% interval. The peak_magnitudes matrix is further updated to have 0 if
% peak magnitudes are not in top N.
% Inputs:
% peak_magnitudes              The peak magnitude matrix obtained from
%                              getConstellation function.
%
% spec_t                       The corresponding time instants to
%                              peak_magnitudes columns.
%
% spec_f                       The corresponding frequency components to
%                              peak_magnitudes rows.
%
% fs                           The sampling rate of the spectrogram
%
% window                       Number of samples in a window of
%                              spectrogram.
% Outputs:
% reduced_constellation        Constellation with reduced the number of
%                              peaks in peak_magnitudes that is more
    uniform.
%
% spec_t                       The corresponding time instants to
%                              peak_magnitudes columns.
%
% fs                           The sampling rate of the spectrogram


% Number of top peaks to survive.
NtopPeaks = 30; % 30;
% Updated and more uniform spectrogram, initialization with zero
reduced_constellation = zeros(length(spec_f), length(spec_t));

% Interval length
index_interval = floor((fs/(window/4))*1);

for i = 0:ceil(length(spec_t)/index_interval)-1

    % At the end of the time array we must get whatever is left regardless
    % of the interval length
    if i == ceil(length(spec_t)/index_interval)-1
        % Choosing the last sub interval of the peak spectrogram
        relevant_spect = peak_magnitudes(:,i*index_interval+1:length(spec_t
    ));
        % Update the NtopPeaks according to the size of the last interval
        % updated_NtopPeaks is scaled according to last_interval/interval
        updated_NtopPeaks = ceil(size(relevant_spect,2)/index_interval *
    NtopPeaks);
        % Choosing the maximum updated_NtopPeaks number of values to later
        % use the last one as a value for threshold
        temp = maxk(relevant_spect(:),updated_NtopPeaks);
    else
        % Choosing the given interval of the peak spectrogram
        relevant_spect = peak_magnitudes(:,i*index_interval+1:(i+1)*
    index_interval);
        % Choosing the maximum NtopPeaks to later use as a threshold
        temp = maxk(relevant_spect(:), NtopPeaks);
    end
    relevant_spect(relevant_spect < temp(end)) = 0;
```

```

    % Assign values to the reduced_spect
    if i == ceil(length(spec_t)/index_interval)-1
        reduced_constellation(:,i*index_interval+1:length(spec_t)) =
    relevant_spect;
    else
        reduced_constellation(:,i*index_interval+1:(i+1)*index_interval) =
    relevant_spect;
    end
end
end
```

A.6 getHashes

```
function [hashes] = getHashes(reduced_constellation, spec_t, spec_f)
% For each anchor point in the reduced constellation getHashes function
% returns the hashes. Hashes are created by pairing each anchor point with
% points in its target zone. The hashes are translation invariant. A hash
% consist of frequency of the anchor point, f1; frequency of target point,
% f2; time difference of these two points, delta t. We use each nonzero
% constellation point (peak) as anchor points.

[size_rowsCons, size_colsCons] = size(reduced_constellation); % Dimensions
    of constellation
[i_rowPeak, i_colPeak] = find(reduced_constellation); % Indices of nonzero
    constellation points (peaks)
nPeaks = nnz(reduced_constellation); % Number of peaks


width_TargetZone = 40; % Width of the target zone in terms of time instant
    indices, right of the anchor point
height_TargetZone = 50; % Height of the target zone in terms of frequency
    indices. Must be an even integer.
initial_HorDist = 5; % Initial horizontal distance from anchor point to
    target zone.

nMaxPairs = 5; % Number of points in target zone to pair for each anchor
    point.


f1=[];
f2=[];
delta_t=[];
t1 = [];


% Each peak is considered an anchor point so we will loop over them.
for i = 1:nPeaks
    % The frequency and time indices of the current anchor point
    i_row_anchor = i_rowPeak(i);
    i_col_anchor = i_colPeak(i);

    % If the target zone right end is outside of the indices, update the
    % target zone right limit
    if i_col_anchor + width_TargetZone + initial_HorDist > size_colsCons
        target_right_end = size_colsCons;
    % Otherwise assign the target zone right end
    else
        target_right_end = i_col_anchor + width_TargetZone +
    initial_HorDist;
    end

    % If the target zone left end is outside of the indices update the left
    % target zone limit
    if i_col_anchor + initial_HorDist > size_colsCons
        target_left_end = size_colsCons;
    % Otherwise assign the target zone left end
    else
        target_left_end = i_col_anchor + initial_HorDist;
    end

```

```
    % If the target zone top end is outside of the indices, update the
    % target zone top limit
    if i_row_anchor - height_TargetZone/2 < 1
        target_top_end = 1; % Row 1
    % Otherwise assign the target zone top end
    else
        target_top_end = i_row_anchor - height_TargetZone/2;
    end

    % If the target zone bottom end is outside of the indices, update the
    % target zone bottom limit
    if i_row_anchor + height_TargetZone/2 > size_rowsCons
        target_bottom_end = size_rowsCons;
    % Otherwise assign the target zone bottom end
    else
        target_bottom_end = i_row_anchor + height_TargetZone/2;
    end

    nPairings = 0; % Number of current pairs for the anchor points

    % Find the peaks inside of the target zone
    target_zone = reduced_constellation(target_top_end:target_bottom_end
    ,...
        target_left_end:target_right_end);
    [j_row_targets, j_col_targets] = find(target_zone);
   for j = 1:length(j_row_targets)
            % Complying with the nMaxPairs constraint and not pairing with
            % anchor point itself
            if (nPairings < nMaxPairs && spec_t(j_col_targets(j) +
    target_left_end - 1) - spec_t(i_col_anchor) > 0)
                f1 = [f1, spec_f(i_row_anchor)'];
                f2 = [f2, spec_f(j_row_targets(j) + target_top_end - 1)'];
                delta_t = [delta_t, spec_t(j_col_targets(j) +
    target_left_end - 1) - spec_t(i_col_anchor)];
                t1 = [t1, spec_t(i_col_anchor)];
                nPairings = nPairings + 1;
            end
    end


end
hashes = [f1; f2; delta_t; t1];
end
```

A.7 hashing

```
function [hashes] = hashing(filename)
% A function just combining getConstellation, getReducedConstellation and
% getHashes for readibility.
[peak_magnitudes, spec_f, spec_t, fs, window] = getConstellation(filename);
[reduced_constellation, spec_t, ~] = getReducedConstellation(
    peak_magnitudes, spec_f, spec_t, fs, window);
[hashes] = getHashes(reduced_constellation, spec_t, spec_f);
end
```

A.8 GiMY_app

```
classdef GiMY_app < matlab.apps.AppBase

    % Properties that correspond to app components
    properties (Access = public)
        UIFigure                matlab.ui.Figure
        Image                   matlab.ui.control.Image
        Lamp                    matlab.ui.control.Lamp
        EditStatus              matlab.ui.control.EditField
        ProgressLabel           matlab.ui.control.Label
        Percentage5             matlab.ui.control.NumericEditField
        Percentage4             matlab.ui.control.NumericEditField
        Percentage3             matlab.ui.control.NumericEditField
        Percentage2             matlab.ui.control.NumericEditField
        Percentage1             matlab.ui.control.NumericEditField
```

```
        SongName5                matlab.ui.control.EditField
        Label_5                  matlab.ui.control.Label
        SongName4                matlab.ui.control.EditField
        Label_4                  matlab.ui.control.Label
        SongName3                matlab.ui.control.EditField
        Label_3                  matlab.ui.control.Label
        SongName2                matlab.ui.control.EditField
        Label_2                  matlab.ui.control.Label
        SongName1                matlab.ui.control.EditField
        Label                    matlab.ui.control.Label
        SongNameLabel            matlab.ui.control.Label
        PercentageofMatchLabel   matlab.ui.control.Label
        RECORDButton             matlab.ui.control.Button
        GiMYGivemetheSongLabel   matlab.ui.control.Label
    end

    % Callbacks that handle component events
    methods (Access = private)

        % Button pushed function: RECORDButton
        function RECORDButtonPushed(app, event)

            % Clear the song name and percentage value boxes each time the
            % "RECORD" button is pushed to avoid confusion.
            app.Image.Visible = "off";
            app.Lamp.Color = 'g';
            app.RECORDButton.Enable = "off"; %Make the button unavailable while in process.
            app.SongName1.Value = ' ';
            app.SongName2.Value = ' ';
            app.SongName3.Value = ' ';
            app.SongName4.Value = ' ';
            app.SongName5.Value = ' ';
            app.Percentage1.Value = 0;
            app.Percentage2.Value = 0;
            app.Percentage3.Value = 0;
            app.Percentage4.Value = 0;
            app.Percentage5.Value = 0;


            %Start recording the song when the "RECORD" button is pushed.
            app.EditStatus.Value = sprintf('Recording...');
            fs = 44100; % Recording sampling frequency
            recordingTime = 10; % Record for 10 seconds


%             Record the audio by using the "audiorecorder" function. Here,
%             we use sampling frequency of fs = 44100Hz, 16bit to represent
%             each sample and we record a mono audio since the songs in the
%             database are also represented as mono audios.

            RECORDING = audiorecorder(fs,16,1); %Recording the audio
            recordblocking(RECORDING,recordingTime); %Record via microphone for 15 seconds.
            RECORDED = getaudiodata(RECORDING); %Save the audio data in "RECORDED" variable.


%             Convert the data in a '.wav' file. Note that "audiowrite" function does
%             not support ".mp3" format since r2015b version.

            audiowrite('mic.wav',RECORDED,fs);

            % Create constellation for the recording
            hashMic = hashing('mic.wav');
            % Appending the track ID as 0
            hashMic = [zeros(1,size(hashMic,2)); hashMic];


%             We must access the song database and the names of the songs
```

```
%              to compare the recording with the database and display the name
%              of the matched songs in the app. Therefore, we declare these
%              variables here as global variables to use them for the rest of
%              the algorithm.


            global data;
            global songNames;

            app.Lamp.Color = 'r'; %"RECORD" button is not available while processing.
            app.EditStatus.Value = 'Processing...';
            drawnow()

            % Defining the match thresholds
            th_f1 = 3;
            th_f2 = 3;
            th_delta_t = 0.050;
            tic
            % For each recorded hash make matrix subtractions to find
            % matching hashes
            matches_diff = []; % Matching hashes
            % Search for the matching hashes
            for i = 1:size(hashMic,2)
                % Take the difference of each hash in the recording with
                % the who data.
                difference = abs(data - hashMic(:,i));
                % Logical index = 1 if the difference satisfies threshold
                % conditions. The indices represent the indices of the
                % matching hashes inside the difference matrix.
                indices = difference(1,:) > 0 & difference(2,:) < th_f1 & ...
                    difference(3,:) < th_f2 & difference(4,:) < th_delta_t;

                % Hold the values of time diffs and track ids
                matches_diff = [matches_diff, difference([1,5],indices)];
            end

            % Among the matching hashes we will look into the absolute time
            % differences, we expect to observe a constant absolute time difference
            % between the mathing hashes if the song is detected correctly.

            % We will hold the maximum bin. We expect a bin to be highly prominent.
            max_bins = zeros(1,length(songNames));
            for i = 1:length(songNames)
                % Select the matches from matches_diff from each track ID.
                time_diff_track_i = matches_diff(2,matches_diff(1,:) == i);
                % Group the time diffs into bins. Each time diff is shifted by a little
                % amount so that we wont have values at the edges of the bins.
                [bin_counts, edges] = histcounts(time_diff_track_i,'BinWidth',0.040);
                max_bins(i) = sum(maxk(bin_counts,2));
            end

            % Among the max_bins choose the most prominent bins.
            [max_max_bins, track_ids] = maxk(max_bins,5);
            toc
            %Find location (index) of the five songs with most matches and
            %their corresponding matches.

            % Calculate the probabilites
            percentages = max_max_bins/sum(max_max_bins)*100;


%              If none of the songs in the database has more than 1 percent
```

```matlab
%               similarity to the recording, than we conclude that the
%               recording does not correspond to any song of the database.

            if max(percentages) < 30
                app.EditStatus.Value = sprintf('The song does not have a match in the database.');
                app.Image.Visible = "off";


%               If at least one song in the database has more than 1 percent
%               similarity to the recording, than we conclude that the
%               recording is indeed similar enough to some songs in the
%               database. In that case, we display the names of the most 5
%               similar songs and their corresponding similarity percentages.

            else
                app.SongName1.Value = (songNames{track_ids(1)}(1:end-4));
                app.SongName2.Value = (songNames{track_ids(2)}(1:end-4));
                app.SongName3.Value = (songNames{track_ids(3)}(1:end-4));
                app.SongName4.Value = (songNames{track_ids(4)}(1:end-4));
                app.SongName5.Value = (songNames{track_ids(5)}(1:end-4));
                app.Percentage1.Value = percentages(1);
                app.Percentage2.Value = percentages(2);
                app.Percentage3.Value = percentages(3);
                app.Percentage4.Value = percentages(4);
                app.Percentage5.Value = percentages(5);
                app.EditStatus.Value = sprintf('The song is successfully identified.');
                app.Image.Visible = "on";
            end

            app.Lamp.Color = 'w'; %"RECORD" button returns to standby mode.
            app.RECORDButton.Enable = "on"; %Make the button available again.
        end
    end

    % Component initialization
    methods (Access = private)

        % Create UIFigure and components
        function createComponents(app)

            % Get the file path for locating images
            pathToMLAPP = fileparts(mfilename('fullpath'));

            % Create UIFigure and hide until all components are created
            app.UIFigure = uifigure('Visible', 'off');
            app.UIFigure.Color = [1 1 1];
            app.UIFigure.Position = [100 100 818 564];
            app.UIFigure.Name = 'MATLAB App';

            % Create GiMYGivemetheSongLabel
            app.GiMYGivemetheSongLabel = uilabel(app.UIFigure);
            app.GiMYGivemetheSongLabel.BackgroundColor = [1 1 1];
            app.GiMYGivemetheSongLabel.HorizontalAlignment = 'center';
            app.GiMYGivemetheSongLabel.FontName = 'Calibri';
            app.GiMYGivemetheSongLabel.FontSize = 48;
            app.GiMYGivemetheSongLabel.FontColor = [0.0157 0.3373 0.549];
            app.GiMYGivemetheSongLabel.Position = [161 481 499 67];
            app.GiMYGivemetheSongLabel.Text = 'GiMY - Give me the Song';

            % Create RECORDButton
            app.RECORDButton = uibutton(app.UIFigure, 'push');
            app.RECORDButton.ButtonPushedFcn = createCallbackFcn(app, @RECORDButtonPushed, true);
            app.RECORDButton.BackgroundColor = [1 1 1];
            app.RECORDButton.FontName = 'Calibri';
            app.RECORDButton.FontSize = 24;
            app.RECORDButton.FontWeight = 'bold';
```

```
            app.RECORDButton.FontColor = [1 0 0];
            app.RECORDButton.Position = [331 411 158 38];
            app.RECORDButton.Text = 'RECORD';

            % Create PercentageofMatchLabel
            app.PercentageofMatchLabel = uilabel(app.UIFigure);
            app.PercentageofMatchLabel.HorizontalAlignment = 'center';
            app.PercentageofMatchLabel.FontName = 'Calibri';
            app.PercentageofMatchLabel.FontSize = 20;
            app.PercentageofMatchLabel.FontWeight = 'bold';
            app.PercentageofMatchLabel.FontAngle = 'italic';
            app.PercentageofMatchLabel.FontColor = [0.0157 0.3373 0.549];
            app.PercentageofMatchLabel.Position = [608 304 178 40];
            app.PercentageofMatchLabel.Text = 'Percentage of Match';

            % Create SongNameLabel
            app.SongNameLabel = uilabel(app.UIFigure);
            app.SongNameLabel.HorizontalAlignment = 'center';
            app.SongNameLabel.FontName = 'Calibri';
            app.SongNameLabel.FontSize = 20;
            app.SongNameLabel.FontWeight = 'bold';
            app.SongNameLabel.FontAngle = 'italic';
            app.SongNameLabel.FontColor = [0.0157 0.3373 0.549];
            app.SongNameLabel.Position = [285 303 101 40];
            app.SongNameLabel.Text = 'Song Name';

            % Create Label
            app.Label = uilabel(app.UIFigure);
            app.Label.BackgroundColor = [1 1 1];
            app.Label.HorizontalAlignment = 'right';
            app.Label.FontName = 'Calibri';
            app.Label.FontSize = 18;
            app.Label.FontWeight = 'bold';
            app.Label.Position = [59 277 25 24];
            app.Label.Text = '1.';

            % Create SongName1
            app.SongName1 = uieditfield(app.UIFigure, 'text');
            app.SongName1.FontSize = 16;
            app.SongName1.Position = [99 279 473 25];

            % Create Label_2
            app.Label_2 = uilabel(app.UIFigure);
            app.Label_2.HorizontalAlignment = 'right';
            app.Label_2.FontName = 'Calibri';
            app.Label_2.FontSize = 18;
            app.Label_2.FontWeight = 'bold';
            app.Label_2.Position = [59 235 25 24];
            app.Label_2.Text = '2.';

            % Create SongName2
            app.SongName2 = uieditfield(app.UIFigure, 'text');
            app.SongName2.FontSize = 16;
            app.SongName2.Position = [99 237 473 26];

            % Create Label_3
            app.Label_3 = uilabel(app.UIFigure);
            app.Label_3.HorizontalAlignment = 'right';
            app.Label_3.FontName = 'Calibri';
            app.Label_3.FontSize = 18;
            app.Label_3.FontWeight = 'bold';
            app.Label_3.Position = [59 192 25 24];
            app.Label_3.Text = '3.';

            % Create SongName3
            app.SongName3 = uieditfield(app.UIFigure, 'text');
            app.SongName3.FontSize = 16;
            app.SongName3.Position = [99 194 473 27];

            % Create Label_4
            app.Label_4 = uilabel(app.UIFigure);
```

```
            app.Label_4.HorizontalAlignment = 'right';
            app.Label_4.FontName = 'Calibri';
            app.Label_4.FontSize = 18;
            app.Label_4.FontWeight = 'bold';
            app.Label_4.Position = [59 153 25 24];
            app.Label_4.Text = '4.';

            % Create SongName4
            app.SongName4 = uieditfield(app.UIFigure, 'text');
            app.SongName4.FontSize = 16;
            app.SongName4.Position = [99 152 473 25];

            % Create Label_5
            app.Label_5 = uilabel(app.UIFigure);
            app.Label_5.HorizontalAlignment = 'right';
            app.Label_5.FontName = 'Calibri';
            app.Label_5.FontSize = 18;
            app.Label_5.FontWeight = 'bold';
            app.Label_5.Position = [59 111 25 24];
            app.Label_5.Text = '5.';

            % Create SongName5
            app.SongName5 = uieditfield(app.UIFigure, 'text');
            app.SongName5.FontSize = 16;
            app.SongName5.Position = [99 110 473 25];

            % Create Percentage1
            app.Percentage1 = uieditfield(app.UIFigure, 'numeric');
            app.Percentage1.FontSize = 16;
            app.Percentage1.Position = [647 283 100 22];

            % Create Percentage2
            app.Percentage2 = uieditfield(app.UIFigure, 'numeric');
            app.Percentage2.FontSize = 16;
            app.Percentage2.Position = [647 241 100 22];

            % Create Percentage3
            app.Percentage3 = uieditfield(app.UIFigure, 'numeric');
            app.Percentage3.FontSize = 16;
            app.Percentage3.Position = [647 199 100 22];

            % Create Percentage4
            app.Percentage4 = uieditfield(app.UIFigure, 'numeric');
            app.Percentage4.FontSize = 16;
            app.Percentage4.Position = [647 154 100 22];

            % Create Percentage5
            app.Percentage5 = uieditfield(app.UIFigure, 'numeric');
            app.Percentage5.FontSize = 16;
            app.Percentage5.Position = [647 112 100 22];

            % Create ProgressLabel
            app.ProgressLabel = uilabel(app.UIFigure);
            app.ProgressLabel.HorizontalAlignment = 'right';
            app.ProgressLabel.FontName = 'Calibri';
            app.ProgressLabel.FontSize = 18;
            app.ProgressLabel.FontWeight = 'bold';
            app.ProgressLabel.FontAngle = 'italic';
            app.ProgressLabel.FontColor = [0.0157 0.3373 0.549];
            app.ProgressLabel.Position = [143 50 75 24];
            app.ProgressLabel.Text = 'Progress:';

            % Create EditStatus
            app.EditStatus = uieditfield(app.UIFigure, 'text');
            app.EditStatus.HorizontalAlignment = 'center';
            app.EditStatus.FontName = 'Calibri';
            app.EditStatus.FontSize = 18;
            app.EditStatus.Position = [246 47 430 27];
            app.EditStatus.Value = 'Waiting for a song';

            % Create Lamp
```

```
            app.Lamp = uilamp(app.UIFigure);
            app.Lamp.Position = [513 415 31 31];
            app.Lamp.Color = [1 1 1];

            % Create Image
            app.Image = uiimage(app.UIFigure);
            app.Image.Visible = 'off';
            app.Image.Position = [19 273 42 37];
            app.Image.ImageSource = fullfile(pathToMLAPP, 'Hand-drawn-gold-
    crown-Clipart-PNG.png');

            % Show the figure after all components are created
            app.UIFigure.Visible = 'on';
        end
    end

    % App creation and deletion
    methods (Access = public)

        % Construct app
        function app = GiMY_app

            % Create UIFigure and components
            createComponents(app)

            % Register the app with App Designer
            registerApp(app, app.UIFigure)

            if nargout == 0
                clear app
            end
        end

        % Code that executes before app deletion
        function delete(app)

            % Delete UIFigure when app is deleted
            delete(app.UIFigure)
        end
    end
end
```

References

[1] Avery Wang. An industrial strength audio search algorithm. 01 2003.