Lab 7: Build your own Shazam

1 Introduction

This lab is about using the DFT to do real audio signal processing. In particular, you will build a music recognition tool (like Shazam) in MATLAB. You will start out by experimenting with spectrograms and their properties and then build your own music recognition system. We have provided a skeleton of the code; all the key functions need to be filled in to make the code actually recognize audio clips.

You should start by copying the `Shazam` directory from bSpace to your local directory, and navigate to it as your working directory. We recommend working in groups. This lab can be somewhat challenging.

1.1 Lab Goals

- Review and demonstrate how spectrograms work
- Learn why we use spectrograms for audio analysis
- Learn how to use spectrograms for audio analysis
- Build a music recognition tool
- Design your own audio features for music recognition (optional)

1.2 Check-Off Areas

- Basics of windowing
- Why we need spectrograms
- Feature extraction (NB: you should get here by the end of week 1)
- Generating a query
- Conducting the search
- Evaluating the algorithm

2 Playing with spectrograms

Spectrograms are plots that tell us about both the frequency and the time information in a signal. They are typically represented as colored plots with time on the horizontal axis and frequency on the vertical axis. Each column of the spectrogram describes a discrete Fourier transform of a fixed number of signal samples. This fixed number of samples is known as a *window*. Figure 1 shows how to find a spectrogram for signal $x(n)$. We're going to step through this process in MATLAB.

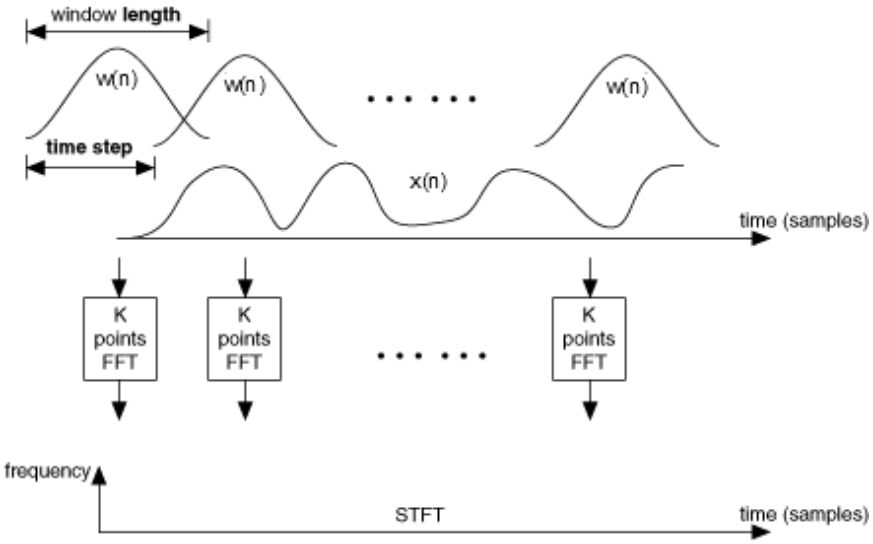

Figure 1: Diagram of short-time Fourier transform (STFT), modified from http://zone.NI.com.

2.1 Basics of windowing

1. Open the MATLAB script `windowing.m`; all your work from this subsection will go into this script. In part 1 of the script, start by defining a variable `dt = 0.1` and creating a time vector `t` that lasts 2π seconds (approximately) and increases by increments of `dt`. From this, generate a sine wave `x=sin(t);`, which has a frequency of $\omega = 1$ rad/s. In part 2 of the script, plot the signal as a function of time using the command `plot(t,x)` to make sure you have generated it correctly. How many periods do you expect to see?

2. Now take the discrete Fourier transform (DFT) of `x`, in part 3 of the script. Before computing the DFT, what do you think the DFT should look like? To actually check this, use the function `X = fft(x)`. This function executes a Fast Fourier Transform (FFT) on your input signal. The FFT is an algorithm for efficiently calculating the DFT of a signal.

3. In part 4, plot the magnitude of the DFT of your signal and confirm that it looks as you would expect. To get the magnitude of `X`, you can use `abs(X)`. Note that `X` is a frequency domain representation of the signal, so your horizontal axis should be in units of frequency

(what is the range of frequencies that show up?) You can generate a vector of frequencies `w` of the same length as `X` for plotting the DFT.

4. Back in part 1, create a similar signal `xhat` that looks almost like `x`, except it only lasts 5.5 seconds instead of 2π seconds (you may want to define a new vector `t2`). Plot `xhat` in the same figure as `x` using the command `plot(t2,xhat,'r')`, where the input `'r'` makes the plot line red (`'g'` would make it green). The point here is to generate `xhat` so that the signal is an incomplete period of a sine wave.

5. Again, take the DFT of `xhat`. Since this array is shorter than `x`, you will need to generate a new frequency vector `w2` in order to plot the DFT of `xhat` (does the range of frequencies change with respect to `w`? Does the spacing of the frequency components change?). What is different about this frequency-domain representation compared to that of `x`? Why do you think this is this happening? (Hint: Think about what the DFT is doing in terms of signal periodicity. Also, the point is not that the peaks are in different places. Look in between the peaks instead!)

6. The effects you saw before are windowing effects. Recall that when we take the DFT of a finite-length, discrete-time signal, the signal is treated as if it were periodic. So if our signal is an incomplete period of a sinusoid, for instance, then that signal's periodic extension (shown below) will introduce some unexpected frequency components to the DFT.

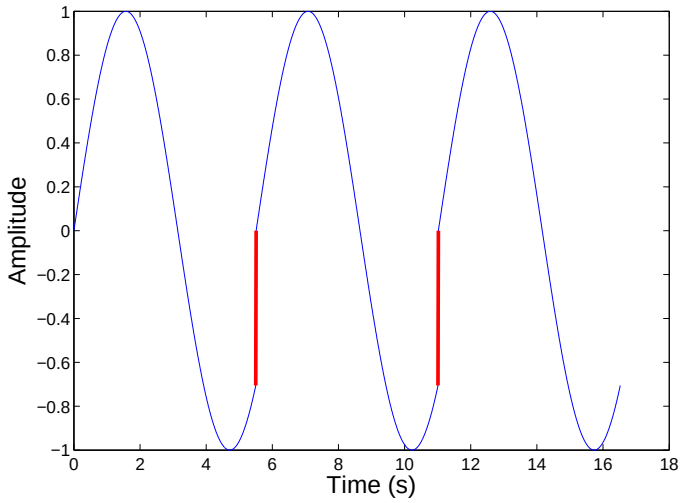

Figure 2: Periodic extension of an incomplete cycle of a sinusoid.

Figure 1 shows that in a spectrogram each window-length segment of $x(n)$ is multiplied pointwise by the window function $w(n)$. In the figure, the window is bell-shaped. When you took the DFT of an incomplete period of a sinusoid, it was as if you were applying a rectangular window of length 2π seconds to a sine wave signal. Now let's try applying a different window. The Hamming window is a commonly-used window function (Fig. 3).

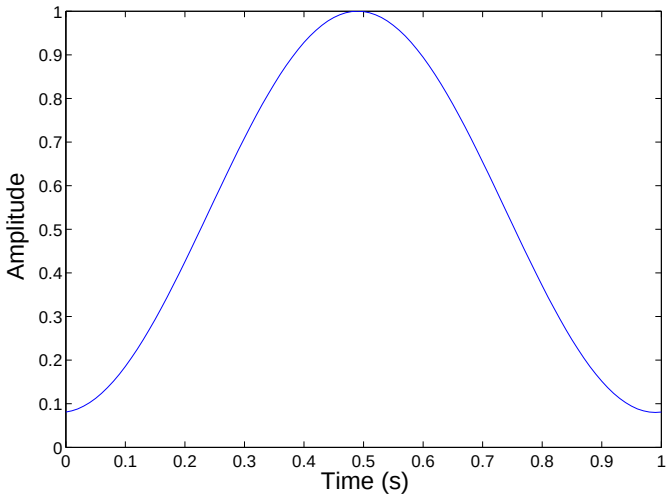

Figure 3: Profile of a Hamming window.

Take `xhat` from the previous step and multiply it pointwise by a Hamming window of the same size. Hamming windows can be generated using the function `w = hamming(L)` where `L` specifies the number of samples in the Hamming window. Now plot the DFT of the Hamming-windowed version of `xhat` (i.e. `xhat.*w`). If `xhat` is a column vector and `w` is a row vector, you will need to transpose one of the arrays using the transpose command, which is an apostrophe (i.e. `xhat'.*w`). How does this look different from the DFT of `xhat`? Why is this happening? Qualitatively, what aspect of the Hamming window is causing this?

2.2 Why we need spectrograms

In the last section, our signal contained only one frequency, and that frequency was present through the entire signal. What happens if there are different frequencies present at different times? Open the script `spectrogramTest.m`; use this script for all the parts in this subsection.

1. In your MATLAB command window, type

   ```
   load testAudio.mat
   ```

   This gives you access to all the variables saved in `testAudio.mat`. Listen to `testAudio` (which was generated with a sampling frequency of `fs`) using the command

   ```
   sound(testAudio,fs);
   ```

   What do you expect the DFT of this audio segment to look like? Plot the DFT of `testAudio` and make sure that you understand why it looks as it does. Now listen to the audio segment called `scrambledAudio` and look at its DFT. Is it different? Why or why not?

2. The last step shows that we can use a DFT to understand what frequencies are present in a signal, but not to understand *where* in the signal those frequencies are located. Now

we're going to start looking at the spectrogram of this signal. To do this we will use the function `spectrogram`. The syntax for this is as follows:

`[S F T P] = spectrogram(x,window,noverlap,NFFT,fs).`

This function computes overlapping STFTs and outputs a spectrogram. In this function, `x` is the input signal, `window` is the length of the Hamming window (in samples, so this should be a scalar!), `NFFT` is the size of the DFT to be used, and `fs` is the sampling frequency. `noverlap` is the number of overlapping samples between two adjacent sliding windows. `fs` only affects the way the spectrogram is plotted, not the spectrogram itself. The relation between `NFFT` and `window` should be confusing to you—doesn't the window size define the length of the DFT? This is typically the case, so you can take `NFFT` equal to the window size. `S` is the 2D spectrogram matrix, `F` is a vector of the output frequencies for which the spectrogram was computed, and similarly, `T` is a vector of times. `P` is a matrix of the same size as `S` that describes the power content at each frequency and time (no complex values). So to plot a spectrogram, you can use the following commands:

```
imagesc(T,F,10*log10(P)); axis tight;
axis('xy'); colormap(jet);
xlabel('Time (sec)'); ylabel('Freq (kHz)');
```

In part 2 of the script, try this out with the `testAudio` signal. Use the provided parameters in `spectrogramTest.m` for `window`, `NFFT`, and `noverlap`. From this picture, can you tell what the structure of the audio clip is? Why are there vertical bands when the note changes? Now type `figure` in your MATLAB command window to generate a new figure, and plot the spectrogram of the `scrambledAudio` signal. Can you tell what the melody is in the second file by looking at the spectrogram? Save both figures.

3. Suppose you are given a spectrogram of a mystery signal that looks like this:

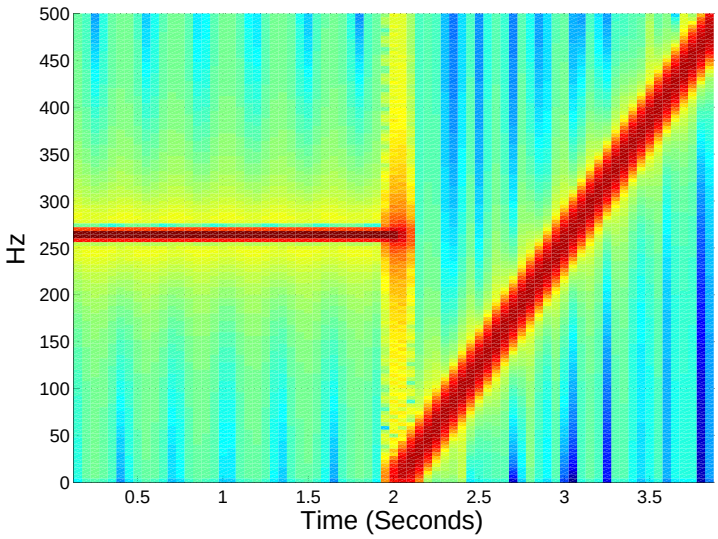

Figure 4: Spectrogram of mystery audio signal.

Qualitatively, how do you think this audio signal should sound? In Part 3 of `spectrogramTest.m`, generate the mystery signal and play it. This will just be an array of numbers (named `mysterySignal` for instance), which you can play with the `sound(mysterySignal,fs)` command. What information is the spectrogram telling you?

Hints: The maximum frequency of the spectrogram tells you what your sampling rate should be. The musical note C_4 has a frequency of 261.63 Hz. You can use the same overlap and window parameters as before when you compute the spectrogram of your mystery signal. Don't worry too much about making the numbers and slopes match—the overall shape is more important.

So spectrograms teach us about both the time and the frequency content of signals. We saw in this section that using only frequency-domain information (i.e. the DFT of the whole audio clip) isn't enough to understand the structure of an audio clip. On the other extreme, why can't we use only time-domain information to understand the structure of the signal? One very important reason is that time-domain representations of audio make it hard to understand what pitches are being played, especially in the presence of noise. Below you can see the time-domain representation of a clean sine wave at 440 Hz and the same wave with noise added.

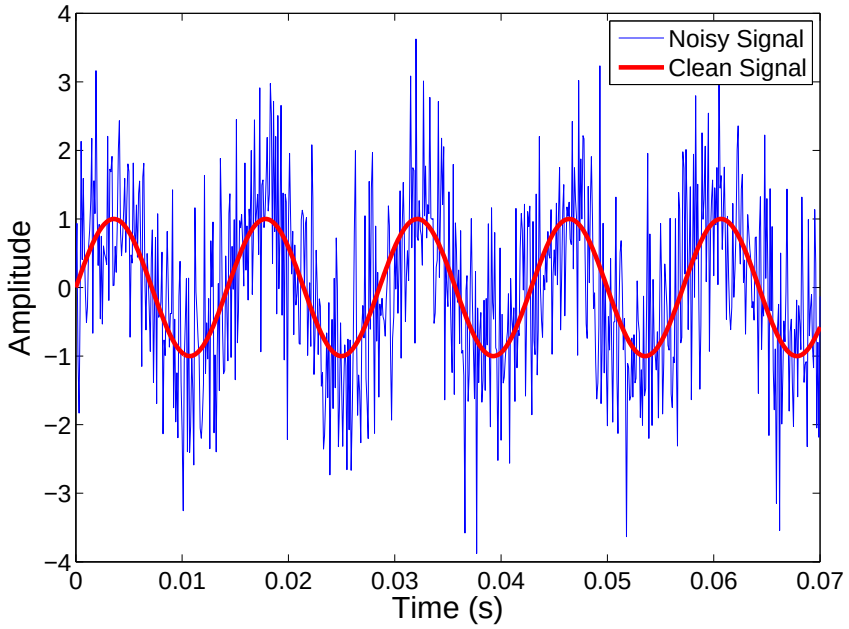

Figure 5: Noisy vs. clean audio signal.

These two signals look very different! However, if we look at the spectrogram, we can see fairly clearly that there is a frequency component at 440 Hz.

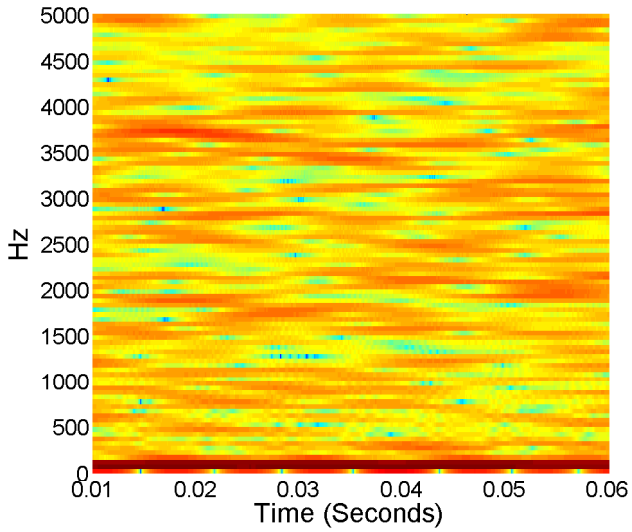

Figure 6: Spectrogram of the noisy signal in Figure 5. Notice how you can still clearly see the dark red line at the main frequency present.

Note that in real songs, there would be many frequencies playing at the same time, so the time version would look even messier, but we would still be able to see the distinct frequencies in the spectrogram.

3 Audio Recognition

Now we're going to move on to build an audio recognition tool. You will be given a "database" of a few songs, but feel free to add more songs if you have any available; note that classical or jazz music is generally harder for computers to recognize than more heavily instrumented genres like pop or rock. To generate a query, you will randomly select a clip from one of the songs in your database and add Gaussian noise. Your engine should then figure out which song the noisy clip comes from.

There are a many algorithms for efficient audio recognition, like those used by Shazam or SoundHound. Most of these algorithms rely on concisely representing spectrogram information. Conceptually, the flow of most algorithms is as follows:

1. Beforehand, find the time-frequency features of each song in the database (for instance, a spectrogram is a set of time-frequency features).
2. During the query, collect a few seconds of sample audio from the noisy query song.
3. Find the time-frequency features of the sample audio clip.
4. Match the features of the sample clip to a segment of one of the database songs.

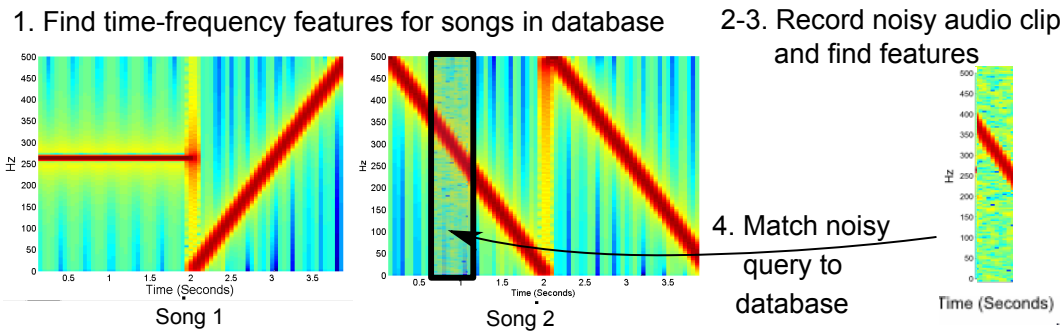

Figure 7: High level diagram of how to do an audio search.

This description almost suggests that we could find the correct audio file by doing an image similarity search—i.e. by interpreting each spectrogram as an image, and treating the query spectrogram as a subsection of an image. This is a valid interpretation, but there are two important observations to make. One is that such a search must be resistant to noise. The other is that the search must be very efficient—tools like Shazam are meant to search through very large databases.

You will implement a simple but effective audio search algorithm by Jaap Haitsma and Ton Kalker.[1] We will step through this algorithm in segments.

3.1 Feature Extraction

We'll start with step 1—extracting features for the database songs. We mentioned that spectrograms can be thought of as a set of time-frequency features. However, the search algorithm needs to be efficient and robust to noise. To make the algorithm efficient, we need to reduce the amount of information stored for each song. To make the algorithm resistant to noise, we need to introduce quantization. Fortunately, these requirements are consistent with one another. You will now simultaneously introduce quantization and reduce the dimensionality of features by converting each column of the spectrogram into a feature vector that is 16 bits long. This number is fairly arbitrary—it was simply observed to work well in practice.

1. Open the file `extractFeatures.m`. You will be modifying this function to extract features from an input audio file. The syntax of this function is as follows:

   ```
   features = extractFeatures(filename,window_time,fs_target,print_length)
   ```

   where `filename` is the audio *.wav file under consideration, `window_time` is the length of the STFT window in seconds (a typical window size is 0.37 sec), `fs_target` is the desired sampling rate (you will be downsampling the signal to a sampling rate of 4000 Hz), and `print_length` is the number of bits we want in our audio features (we will use 16-bit features).

[1] Haitsma, Jaap, and Ton Kalker. "A highly robust audio fingerprinting system." Proc. ISMIR. Vol. 2. 2002.

2. Make `extractFeatures` load the input file `filename` using the command

```
% read in the file and the sampling frequency
[audio fs] = wavread(filename);
% wav files consist of two columns of data; we'll extract only the first column
audio = audio(:,1);
```

   The function `wavread(...)` returns two arguments—the audio content `audio` and the sampling frequency `fs`. You can verify that you did this correctly by using the function

```
playAudio(audio,fs,play_time)
```

   where `audio` is the input audio (not the filename—here you should put in a vector of numbers, like `audio`), `fs` is the sampling frequency of `audio`, and `play_time` is the number of seconds to play the audio file. Make sure that `audio` plays properly.

3. Currently, the audio signal has too much information for us to process it efficiently. Start by downsampling `audio` to a sampling rate of 4000 Hz. You can do this with the function `audio_ds = resample(audio,fs_target,fs)`.

4. Find the spectrogram of `audio_ds` and plot it. You can use an overlap factor of half the window length.

5. Currently the number of frequency components in the spectrogram is equal to `NFFT`. However, as mentioned earlier, we want to compress each column of the spectrogram to a 16-bit feature. This will be done in the following way: Start by dividing the spectrogram into 17 evenly-spaced frequency bands. You can find the `[numRows numColumns]` size of the spectrogram `S` with the command `size(S)`. Is the number of frequency bands equal to the number of rows or columns? To divide the frequency components into 17 bands, how many rows or columns belong in each band? (This may not be an integer.)

6. For each column in the spectrogram, sum up the power in all the indices of each frequency band. If your answer from the previous part was not an integer, just use the floor of that number. This means that your last frequency band may have more elements than the other 16 bands, which is fine. Note that the function `sum(x,dim)` returns the sum of a matrix `x` in dimension `dim`. If `dim`=1, it sums the rows of a column, while if `dim`=2, it sums the columns.

7. Then for each column, compare the total power of each frequency band to that of the frequency band below it. If the ith frequency band has more power than the $(i-1)$st one, assign a 1 to the ith bit, and a 0 otherwise. This is illustrated in Figure 8. Note that for identically-sized matrices `A` and `B`, the MATLAB command `A>B` returns a binary matrix in which the `(i,j)` index is 1 if `A(i,j)>B(i,j)` and 0 otherwise. What information do these features tell us? Why do you think such a feature is resistant to noise?

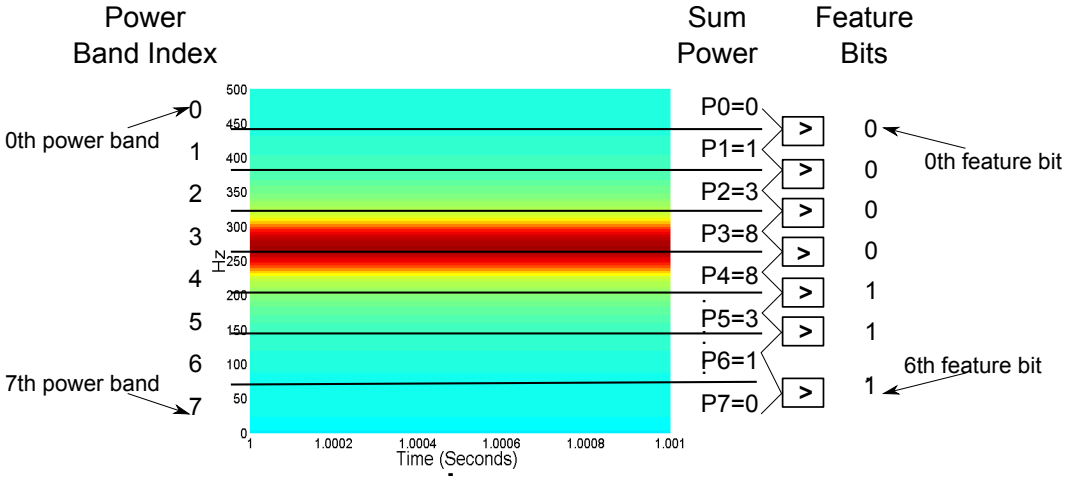

Figure 8: How to extract a 7-bit feature from a single column of a spectrogram.

After this step you should have a set of 16-bit feature vectors. For each song, there should be as many feature vectors as there are columns in the spectrogram. Verify that this is the case.

8. Currently your function outputs a matrix of bits. Before exiting the function, convert the 16-bit feature vectors to decimal numbers using the function `bi2de(x)`. This function can accept a binary matrix and it converts each binary row to a decimal number, but make sure that your matrix is rotated properly (otherwise, you will get only 16 feature vectors!). So now, `extractFeatures` should return a 1D vector of 16-bit numbers.

9. To check that your feature extraction code works properly, start by loading and listening to `tone.wav` in the MATLAB command window as follows:

```
[w fs]=wavread('tone.wav');
wavplay(w,fs)
```

Now look at the spectrogram with the command `spectrogram(w)`. The resulting spectrogram does not give units of Hertz, but it is good for showing you the general shape of the spectrogram. Now extract your features from `tone.wav` by calling your function from the command window:

```
test_features = extractFeatures(filename,window_time,fs_target,print_len);
```

Now ensure that the returned features make sense. You can visualize the returned features with the function `imshow(de2bi(test_features)')`. This will display an image of the binary `test_features` matrix with white '1' bits and black '0' bits. What do you think the features should look like for `tone.wav`? Does this look like your spectrogram? You can use Figure 8 to reason about how the features should look.

Checkoff point: Week 1—Get checked off by your lab TA for week 1 of this lab once you reach this point.

---

10. Open the function `makeDatabase.m`. In this function, we want to extract the features from each song in the database and store them in an appropriate structure. Start out by looping through the files in `filelist`. Note that `filelist` is a cell array (rather than a regular MATLAB array), so to access the ith filename, use the command `filelist{i}`. Cell arrays are good for storing a list of differently-sized data. For each file, extract the features from that file.

    Store the results in the struct array `songs`. Each element in this struct array should have two attributes: `songs(i).name` and `songs(i).features`. If you are unfamiliar with structs in MATLAB, you don't have to define the struct anywhere. You can just type

    ```
    songs(i).name = filelist{i};
    songs(i).features = extractFeatures(...);
    ```

    in each iteration of your loop. The function `makeDatabase(...)` will return this structure of songs as the database description.

11. Why do you think these features rely on spectrogram information? Do you think you could use just time-domain information or just frequency-domain information? Why or why not?

4 Query Generation

Now you have a complete description of the database in terms of feature vectors, it's time to create a noisy query clip for the system to recognize.

1. Open the file `main.m`. In the section titled 'Query Generation', start by selecting a random filename from `filelist`. Call this randomly selected filename `queryFilename`. You can look up the function `randi(...)` to do this.

2. Now open `generateQuery.m`. This function is designed to extract a clip from the randomly selected file. You will fill in the missing parts of this function (you should be able to reuse a lot of code from before). Start by opening the input `queryFilename` as before and keep only the first column. Call this variable `clean`, since it represents your clean data.

3. Now you're going to add some Gaussian noise to the query audio file, and store the noisy version in a variable called `noisy`. Look up the function `awgn(...)` to do this. You can use the SNR that is provided in the function input parameters. Listen to the noisy audio. Can you hear the noise?

4. Downsample the noisy file to 4000 Hz. Extract the features for the audio file as before.

5. From the set of noisy audio features, randomly choose a set of 100 consecutive features. These are the noisy audio features that you will use to identify the song, and these 100 features should be returned from `generateQuery` as the variable `features`.

5 Search Algorithm

Now you have extracted features for the database, and you've also extracted features for a noisy audio clip. To complete the search process, you will successively search for each of the 100 query features.

1. To begin, you will make a function that returns the bit error rate of two (identically-sized) vectors of decimal numbers. Open `ber.m`. Start by making sure that both vectors are the same size. It they are, convert both `a` and `b` to their binary representations. You can use the function `de2bi(...)` to do this.

2. Now you should have two matrices filled with bits–one for `a` and one for `b`. To find the bit error rate between these two structures, count the number of indices where the two matrices differ and divide by the number of elements in one of the matrices. You may find the function `xor(...)` useful, but there are many ways to do this task. Return the error rate you found. Test your `ber(...)` function to make sure it works properly. For instance, what would be the BER of the numbers 3 and 2?

3. Open `findMatch.m`. This is the function that will actually search for the query. Here's how the search will work: You start out with 100 16-bit features. You will loop through all these features, and search for the feature in the songs in your database. For instance, if your first noisy query feature is 12, you would search through each database song's features for occurrences of the number 12. Each time you find the feature '12' in a song, you extract that song's corresponding chunk of 100 features. If the bit error rate between the noisy query features and the database song's features is below threshold, then declare a match and return the name of the song as the match. If you don't find any matches below threshold for the first feature (12), then move on to the next feature.

   To start out, loop through all the 100 query features (i.e. the noisy ones). For each of the query features (assume we're on the ith one), loop through all the songs in the database. For each song, find the indices where the ith feature occurs. You may find the function `find(...)` useful. For each occurrence of the ith feature, first check if there is an appropriately sized chunk of 100 features surrounding it. E.g. if the 1st subfingerprint occurs as the last feature of song 1, then it is not possible to find such a chunk.

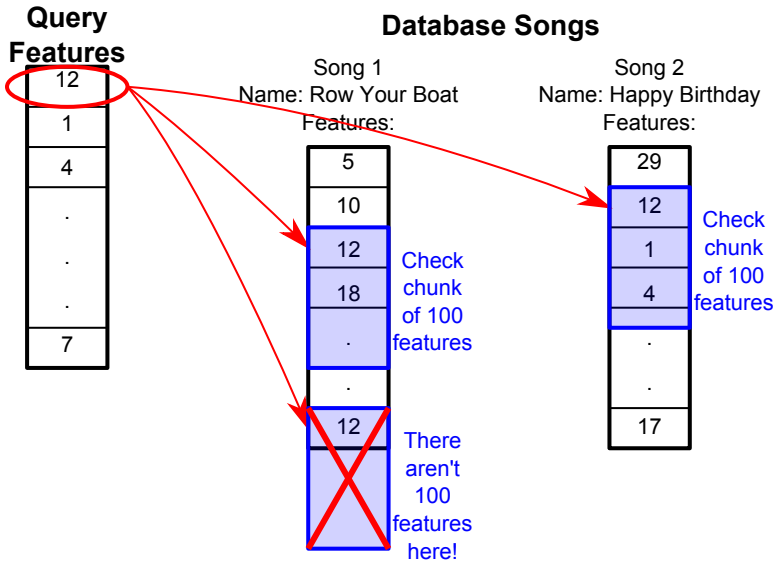

Figure 9: How to search for query features in the database. Here we're searching for the 1st query feature, which has value 12. In song 1, the feature 12 occurs at the 3rd index and the last index.For the 3rd index, the we can check the bit error rate between the query and the 100 features starting at index 3. However, for the last index, there aren't 99 more features following 12, so that is clearly not the source of the noisy clip.

4. Once you've found a chunk of 100 features in the database that contain the ith query feature at the ith index, compare the bit error rate of the two vectors using the function you already built. So you should be comparing two vectors of size 100. If that error rate is below threshold, then you can return the name of the database song. Otherwise, keep searching. This should complete your function `findMatch(...)`.

5. Now go back to `main.m`. After receiving the name of the match, check whether it gave the correct answer, i.e. the same filename as `query_filename`. If this is *not* the case, there is probably a bug in your code somewhere. You can try a few things while debugging, like not adding any noise to the query audio (just comment out the part where you add noise), and/or taking the audio sample from the beginning of the song instead of from the middle.

6. Once you have the search working for one song, modify `main.m` so that you will test multiple songs (make sure you don't rebuild the database each time!). What percentage of songs are being classified correctly? You can also save the database information in `songs` as a `.mat` file so that you don't have to rebuild the database every time. Try looking up the `save(...)` command to do this.

7. Try lowering the SNR so that you can hear a lot more noise in the signal. Now what kind of recognition rates are you getting?