Song/Music Recognition Software

Created by Senyuan Fan, Jiwei Zou, and John Lynch

Our chosen final project for Audio Signal Processing was to build a series of MatLab scripts to extract the fundamental frequencies of monophonic signals, and compare the results to an existing database of already processed songs/music. The goal with this program was to recreate the same sort of technology seen in applications and websites with a focus on monophonic melodies. These melodies are often located in the mid-range of the audible hearing spectrum, which allows us to condense the program down to this area of frequency, and create a condensed version.

The process for our code is as follows. Once the input signal has been stored into a matrix with its sampling frequency, we apply windowing over the signal, then bandpass filtering to isolate the key frequencies we intend to use. From here, we then use fast-Fourier transform, or *fft*, to rework the signal and graph it with respect to frequency. This allows us to determine the strongest frequencies from each sample, and determine the fundamental frequency of that frame of signal. Once we have stored the fundamental frequencies, we then run these against the written list of note names and their respective frequencies (Note: we used the standard 88-note piano range for reference), and then return what the closest notes are to each fundamental frequency. Finally, we run our results from note-taking through the database of already-processed music, and compare the values between each. Our hypothesis is that, when purposely imitating a particular song in the database, this program will return with said song to the user, with the input having the greatest correlation with this song among the list.

To start with the scripts written, *Identifier.m* is a class definer for incorporating object-oriented coding into the program. As commented in the script itself, the key properties for this class include the bank of audio files, their respective names, sampling rates, and notes, and storage for test results of input signals. While these values are kept private, users can write to them directly once an object has been constructed. *Identifier* objects also possess functions to modify the bank post-construction and to process the test signals. With these, the class script is set to hold the database of music files, update as necessary, and begin the core function of the

main program when commanded. When elaborating on the core process of the program, we are following the process within the *Identifier* object.

After the preliminary windowing of the input signal (performed via MatLab command *windowing*), we then use the *AudioToMIDI* script to convert the audio into a series of note frequencies. This begins with using a bandpass filter ranging from 100 - 4,000 Hz, to remove unnecessary data. Once that is done, the script sends the audio through the built *find_Freq.m* function, which takes the absolute fft of the signal, and returns the fundamental frequency, using the built-in MatLab function *pitch*. (Note: One assumption we make here is that the strongest frequency at any point will be the main melody of the input signal.) We originally created our own script to find pitches from the signal, but as shown in the slides, the built-in function from MatLab returned less spikes in the results.

The script finally calls on the *Round_Note.m* script to interpret the frequencies as notes on a standard scale. We approach this with a do-while loop, which will break once the distance between the input frequency and the currently checked note reaches its minimum. One it does this, it will return both the note name and frequency. For the scripts and functions used in *AudioToMIDI*, we specifically send in single frames of audio at a time, in order to keep the for loops outside of the functions themselves. This is a simpler process than sending in entire matrices as parameters.

Upon receiving our matrices of notes, we send these values to be compared to the songs in the database. We do this by applying dynamic time warping (dtw) to find the lowest local cost, which is our means of returning correlation; the lower the local cost, the greater the correlation. By use of dtw, we also bypass the complication of tempo difference. This analysis also runs multiple times to transpose the input signal, in the event that there is a difference in key from the given audio and the intended music in the database.

We have a database of 6 songs stored for the purpose of testing:

- Bach - Violin Sonata No. 1, Parts 1 and 2
- Jules Massenet - Meditation from Thais
- Maroon 5 - Memories
- Yasunori Nishiki - Ophelia the Cleric
- Coldplay - Viva la Vida

This range of pieces allows us to test with different instruments, including vocals, for leading melody. This is important, since different instrument types have different signature

frequencies when excited. When inputting audio, we used violin to imitate the main parts of each music, so that we could test strictly with other violin-focused songs, and ones that weren't.

In our results, we found that the program was able to successfully answer the majority of the time. This carried across vocal tracks as well, despite the possible issue of vowel/consonant usage. For the set of 8 different tracks we tested, 6 had accurate returns of the audio. In the instances that the results were inaccurate, we used pizzicato, or string plucking, which the software could correctly assess only 1 out of 3 tests done. The other tests, all of which used arco (return to bowing after pizzicato), were accurate in every assessment.

One of the notable limitations with this software is with music that includes pizzicato, which is likely due to the lower intensity of the signal generated. Sustained notes have more instances of the frequency that overlap with that of input signals, which make them easier to compare than pizzicato.

Another noteworthy issue we ran into with the music is with faster pieces. Bach's Sonata has a faster tempo than other songs, which relays the same limitation from pizzicato notes. With less sustain on individual notes, there are less frames of each note to compare with one another. Zero padding is one option for solving the problem of short note durations, as it would increase the number of samples, therefore providing greater overlap of compared notes/frequencies. The drawback with this approach is the amount of storage it would require, especially since we'd need to apply zero padding to the database audio clips as well, which are already about 90 seconds long.

When originally discussing approaches for song recognition software, the other option we came up with was pitch shift detection. What this process would do is simply detect any changes in the fundamental frequency (a different note is played), and subsequently record and store both the timing and magnitude of the shift (this would include negative shifts - notes going down in pitch). This has the potential of greater efficiency over the current method, as a matrix of times and magnitudes would be inherently smaller than one with the frequencies of each frame of audio processed. This could be expanded on by detecting the initial note, then allowing the program to determine the rest of the notes from the signal using the detected shifts.