Signals and Systems

EECE 301

Toy Version of "Shazam" Music Identification Algorithm Report

By: Jung Park and Adam Gluck

The goal of our project was to create a program that has very similar functionality when compared to the popular app, Shazam. Shazam is an app available today that listens to small snippets of a song, recognizing the song even in the presence of background noise. Our program is similar to Shazam, but with a much smaller database of songs and processing power. The program we developed in MATLAB compares a snippet a song input to a database of mp3 files that we created, matching the song to the one in the database. By looking at the "score" of each song, we can determine which song in the database is the song in the snippet. The "score" will be explained later in the paper, but it is a number that ranges from 0 to 1; the closer you are to 1, the closer you are to a match. We then tested how the song signal functions under different conditions by applying various filters to it at the end.

We had to develop a myriad of scripts in order to create, test, and implement the program. In order to read in songs to MATLAB, we had to complete a script called "Read_Songs.m". This script would allow us to read into MATLAB the samples from the song's mp3 file. The read_song script takes in an input of a five second song snippet and stores it into a matrix called Y. More specifically, the song takes in a snippet from the 60 seconds to 65 seconds in the song. This matrix Y contained 220,501 rows and 2 columns. We know this because the song had a sampling rate of 44100 $\frac{samples}{second}$ and was read for 5 seconds. There were only 2 columns in this matrix since there are only two stereo channels for the signal. The sampling rate is also stored into a variable, called FS, so we can check that the sampling rate of the song is 44100 $\frac{samples}{second}$. The first part of the read_song script averages together the two stereo channels for each row and stores them into a variable called y. We do this to create a mono signal to deal with instead of two stereo channels. We then remove the DC offset of the signal by subtracting the mean value of the mono signal from the mono signal for each sample of our mono signal and store it back into y. We do this to improve the effectiveness of the processing of our songs. At the end of the script, we resampled the song and stored it into a variable called y1. We also noted that when we resample, our sampling rate becomes one fifth of what it used to be because when y = (x,p,q), the length of the result y is $\frac{p}{q}$. After completing the "read_song.m" function, we ran the function on the command line with one of the mp3 files in our database as the parameter and made sure that the output of the function had one row. This confirmed that we averaged together the two stereo
channels together correctly.

To fully test our read_song function and the resampling that occurs at the end of the function, we developed a script called "test_resample.m". In this script, we created a signal x, the sum of two cosine signals, to test our resample function on. We then created two different resampled signals based upon our original signal. One sample was when the signal was resampled without aliasing, and one where the signal was resampled with aliasing. We computed the magnitude for each of the three signals and plotted them vs. frequency in Hertz to verify that

we were performing our test correctly. These graphs can be seen in Figures 1-3. We then compared those signals to the samples provided by a song from our database that was read in and stored into vector Y. We also removed the "DC offset" of vector Y before plotting it and comparing the magnitude of the vector to the magnitude of our original sum of sinusoids. The DFT of vector Y can be seen in Figure 4. Lastly, we also compared our resampled sum of sinusoids that has no aliasing to the resampled signal of Y that has no aliasing. This comparison is shown when you compare Figure 5 to Figure 2.

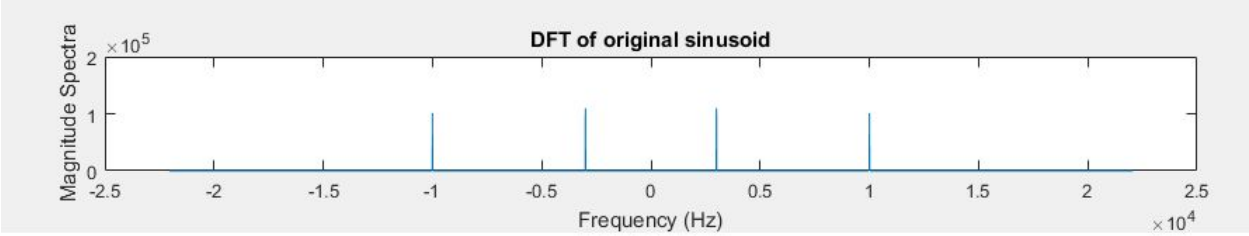

Figure 1: Signal without resampling

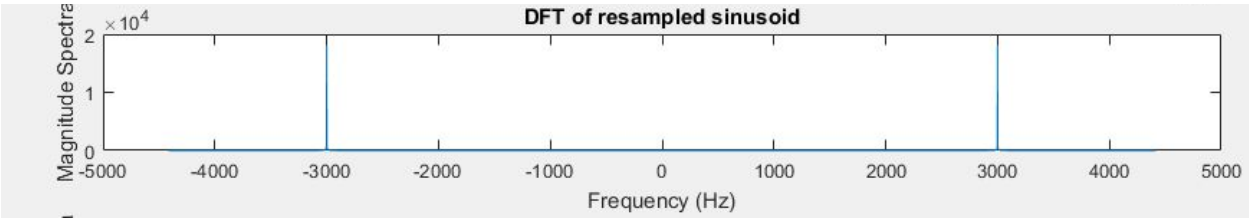

Figure 2 : Signal resampled without aliasing

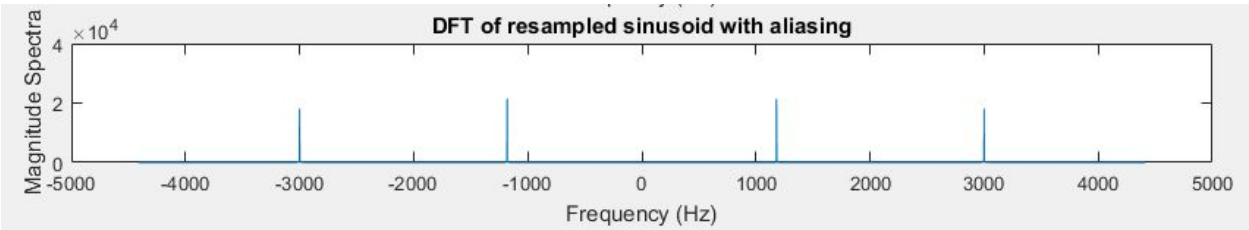

Figure 3: Signal resampled with aliasing

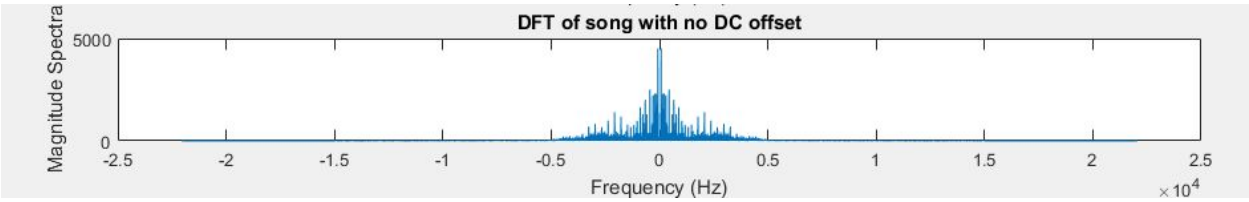

Figure 4: Song snippet vector

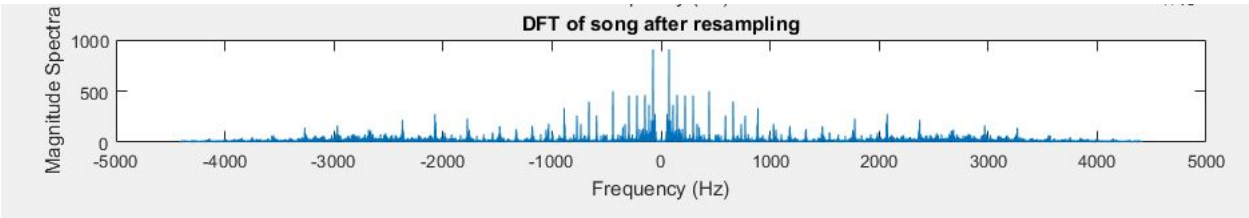

Figure 5: Song snippet vector resampled without aliasing

In order to confirm that our graph was outputting correctly, we did a few calculations. For our sum of sinusoids, the first frequency, f1, was 3000 Hz and the second frequency, f2, was 10,000 Hz to create the equation $x = cos(2\pi f_1 t) + cos(2\pi f_2 t)$. The DFT of this signal can be seen in figure 1. Since the DFT of a cosine signal is just a delta function at the cosine signal's frequency, we were able to confirm that the first graph was outputting the correct information. Our second plot had the original signal re-sampled without aliasing, the only peaks that we expected to show up were at positive and negative 3000 Hz, which worked perfectly. This is because the peak at 10,000 Hz was cut out since it was greater than Fs/2. The difference between figure 2 and figure 3 was that figure 3 contained our aliased signals as well. Because of this, we knew that we not only should see peaks at negative and positive 3000 Hz, but also see peaks at at positive and negative 1180 Hz. This is because our peak at positive and negative 10,000 Hz should have peaks a distance equivalent to the sampling rate, 8,820 Hz, away from it. $10000\ Hz - 8820\ Hz = 1180\ Hz$. We then plotted the DFT of a song from our database and the DFT of the resampled song. To verify that we were seeing what we were supposed to, we checked to make sure that all signals over 4,410 Hz were not in the resampled plot. This is because the resampled plot of our song was supposed to contain no aliasing. As one can clearly see by looking at figure 5, our graphs outputted exactly what we wanted it to.

The next step in order to create our "Shazam" program was to create a function, that can compute a signal's spectrogram. To start off, we created the STFT function with the parameters of an input signal, block size, step size, zero-pad length, and sampling rate. This function would output a matrix that holds the result of STFT, a vector holding the time values, and a vector called the frequency values. After defining the function, we had to do initial checks to confirm that the user did not input any impossible inputs. We checked that the user inputted a block size that is less than the zero-pad length, but greater than the step size**.** If the block size was less than the step size, then we wouldn't be able to read any samples and if the block size was greater than the zero-pad length then there probably would not be any point in zero-padding at all. After performing the initial checks to confirm that we have inputs that can work in the real world, we created an indexing matrix to store index through our signal matrix. We do this so that we can obtain "blocks" of samples from our song in our signal matrix. Doing this will allow us to reduce the amount of smearing we have. In order to create the indexing matrix, we created two vectors that we would add together. The signal matrix was created by using the indexing matrix to obtain "blocks" of samples . To test this command, we ran the STFT function with correct parameters in order code to generate a spectrogram of the 5-second snippet at the end. This spectrogram is shown in figure 6. The spectrogram appeared to be correct, but the only way to know for certain that STFT was working correctly was to perform further testing. Additionally, one can notice that we only have half the values from the original fourier transform function as it is not mirrored in the spectrogram of the song.

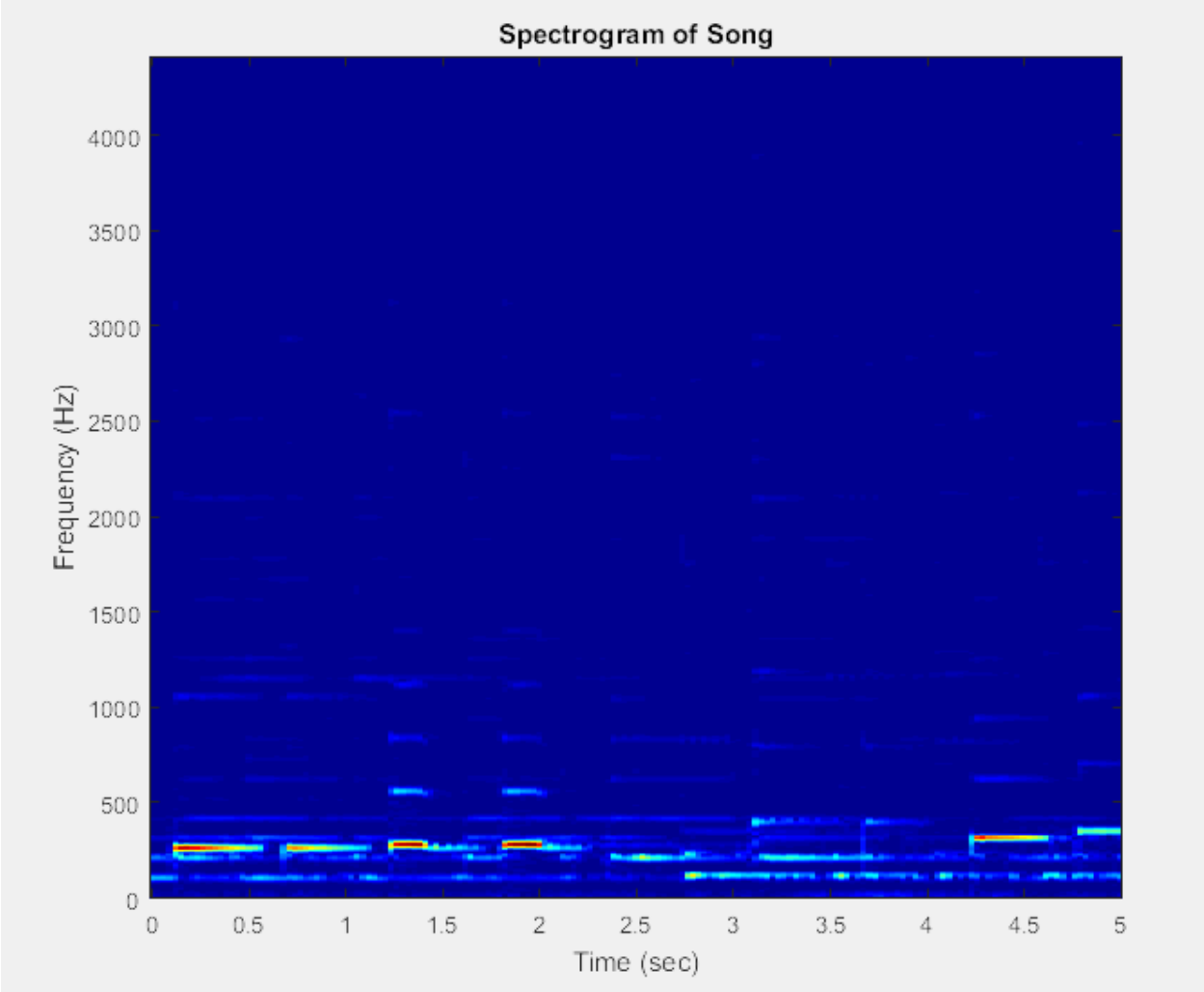

Figure 6: Spectrogram of a song

To perform further testing of this command, we created a test script called "test_STFT.m". The purpose of this script was to ensure that our STFT command was outputting the correct information. In order to test this, we created two different sinusoids to plot on a spectrogram. The first sinusoid had a constant frequency and the second sinusoid changed frequencies halfway through the signal's duration. At first, we were stumped as to how we would create a sinusoid that changes frequencies halfway through its duration. Our solution was to create a vector that would hold the value of 1 for the first half of it and the value 2 for the second half. We created this vector of 1s and 2s and made it the same size as our time vector so that we could bitwise multiply the two together in order to force a change in frequency halfway through our sinusoid. Plotting these two sinusoids confirmed that we were outputting the correct values. We knew this because our starting frequency for both signals was 1500 Hz, and since our frequency varying sinusoid just doubled its frequency halfway through, we knew the second

spectrogram should just go from a straight line at 1500 Hz to a straight line at 3000 Hz. You can see the two spectrograms in figures 7 and 8.

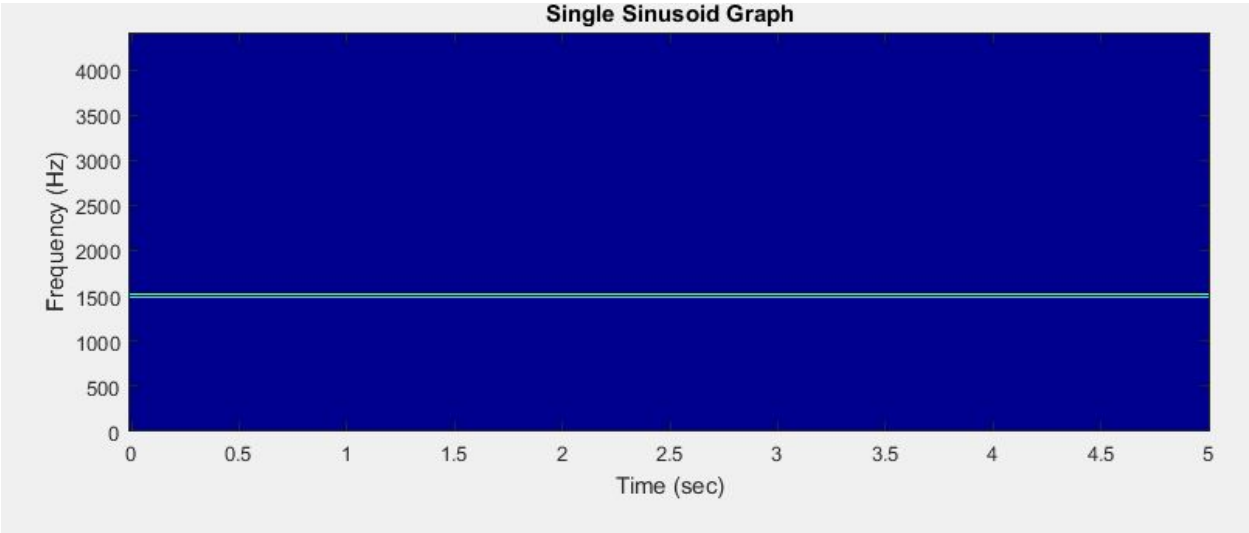

Figure 7: A sinusoid graph plotted at 1500 Hz.

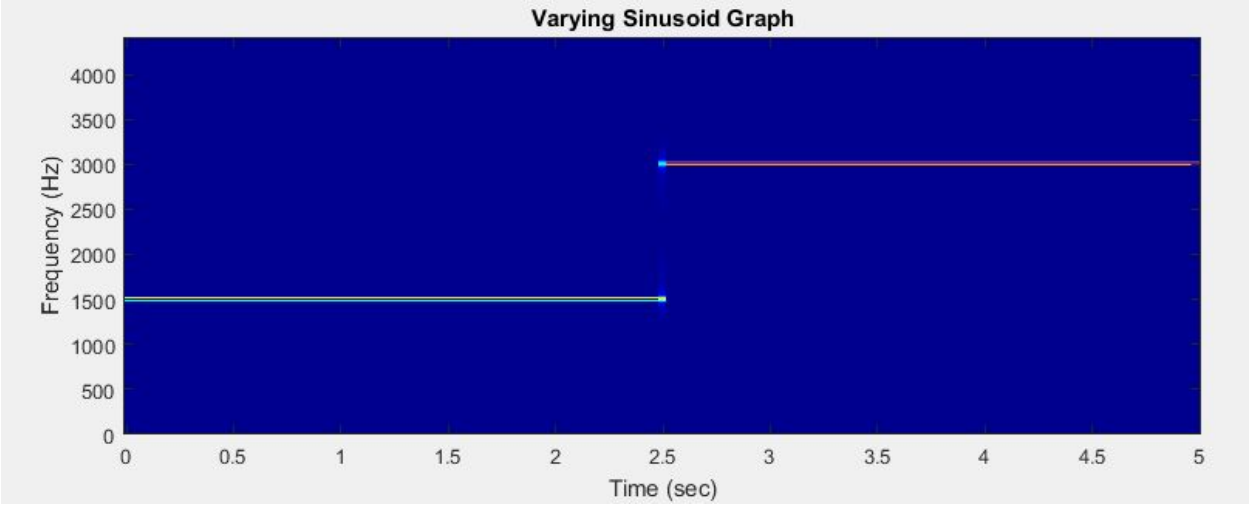

Figure 8: A sinusoid plotted at 1500 Hz and then increases to 3000 Hz halfway through the sinusoids duration

Since we knew that our STFT function worked properly, we implemented the STFT function into our partially completed “fingerprint.m” file to complete the fingerprinting process. The fingerprinting process found local peaks of the magnitude of the output of our STFT function. If the peak was less than the calculated threshold, the peak was discarded. After discarding the local peaks, our function would plot the remaining peaks onto a spectrogram. These peaks appear on the graph as small blue rectangles. These peaks can be seen throughout the spectrogram of a song from our database, shown in figure 9. As one can clearly see, there are many peaks in the spectrogram, even after eliminating the local peaks whose magnitude was less than the threshold. This is because there are 44100 samples every second that our song is listened

to and since our input to the STFT function is a 5-second song snippet, there are $44100 * 5 \frac{samples}{second}$. Since there are so many samples, there obviously will be a lot of peaks as well!

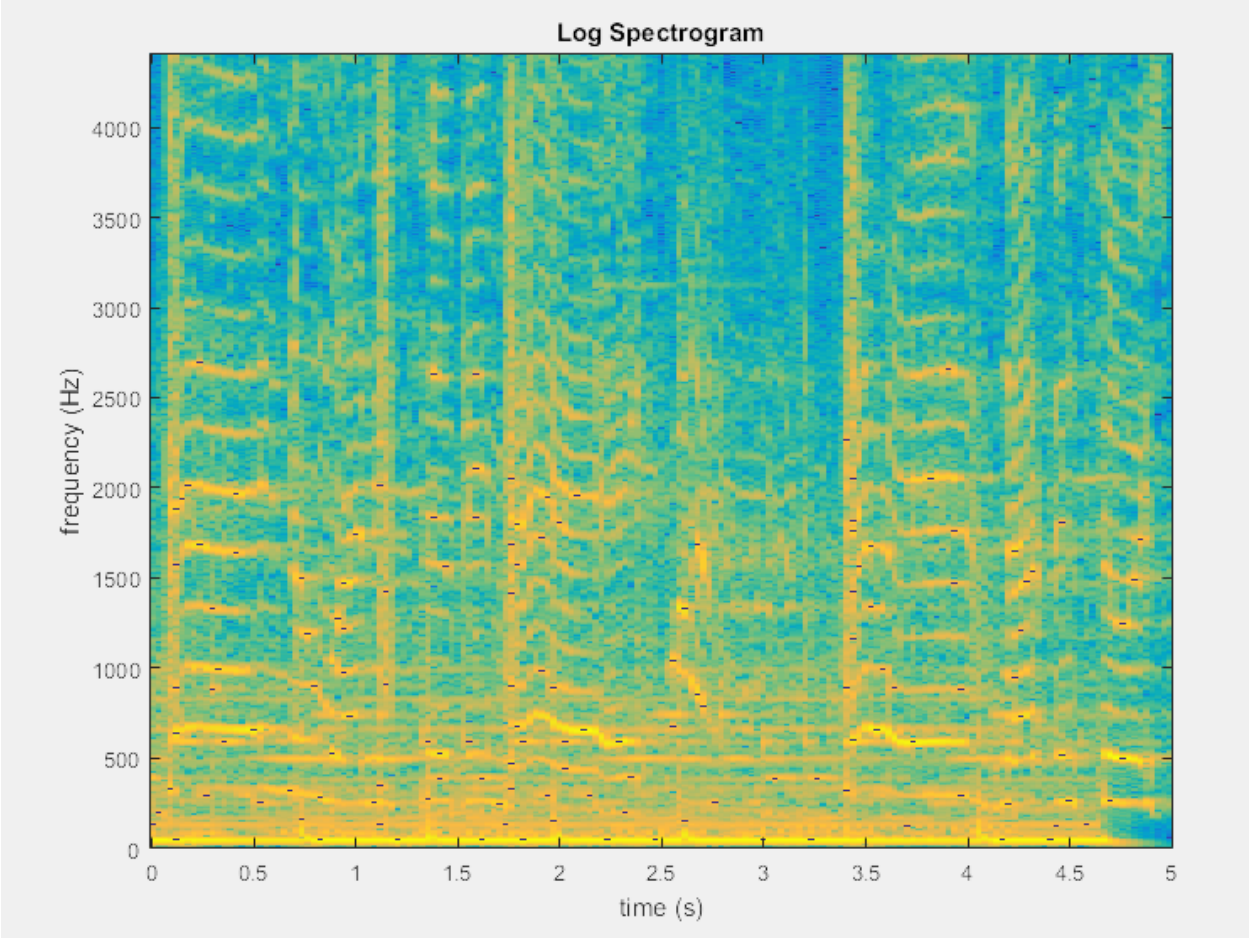

Figure 9: A spectrogram of a 5-second snippet of a song from the database with local peak locations indicated by blue rectangles

In order to determine which song our 5-second snippet input was from, we created the last function called "ID_Song.m". The function would compare our 5-second song snippet to every song in our database and would store the "score" in a row vector, showing the user how closely the song and song snippet matched. An output of 1 or very close to 1 indicated that the song snippet was from that song in our database. The further that the "score" of a song was when compared to our song snippet, the less likely that song snippet was from that specific song. In order to perform this task, we downloaded random 21 random mp3 files that we would use as our database. We stored the each song's samples from our database into a matrix called P. We then took our 5-second song snippet and compared it to each song's samples to determine a "score" for each song in our database. To check to see if our newly created function was outputting the proper information, we ran different 5-second snippets of mp3 files into the ID_Song function and made sure that the output was 1 only when the 5-second song snippet was

compared to the song it is from. The score of every other song from our database should be close to 0, if not 0. Figure 10 shows our group testing the ID_Song function in the command line to make sure that the "Score" vector is 1 for only one song and has very low values when the snippet is compared to the rest of the database. As one can clearly see, our ID_Song function was properly working.

```
Trial>> y = read_song('7.mp3');
Trial>> ID_Song(y,P)

ans =

  Columns 1 through 6

    0.0067         0    0.0067    0.0067    0.0067         0

  Columns 7 through 12

    0.0133    0.0067    0.0133    0.0133         0         0

  Columns 13 through 18

         0    0.0067    0.0133    0.0067    0.0133         0

  Columns 19 through 21

    1.0000         0    0.0067

Trial>> y = read_song('1.mp3');
Trial>> ID_Song(y,P)

ans =

  Columns 1 through 11

    1.0000    0.0067    0.0067         0    0.0133    0.0067    0.0133    0.0133    0.0067         0    0.0067

  Columns 12 through 21

         0    0.0133         0    0.0133         0         0         0    0.0067         0    0.0133
```

Figure 10: Two tests of ID_Song.m and the output of the function.

After completing the "ID_Song.m" function, the mimicked "Shazam" app was complete. However, since we knew that our program was not going to always be used in ideal conditions, we created a few functions to test our program under different conditions. The first filter we created to test the ID_Song function with was a lowpass filter. A lowpass filter does exactly what it sounds like; it only allows lower frequencies to pass through and cuts off all high frequencies. Our lowpass filter would output a vector called "Score" which would store the "Score" of each song, similarly to the way it did in ID_Song, after being put through a low-pass filter. In order to plot our low-pass filter, we needed to calculate the frequency vector and response vector of our 5-second song snippet. We also had to make sure that we plotted our frequency vector in hertz and our response vector in dB to show the lowpass filter of our song snippet. To calculate these two vectors, we had to use a command called "freqz" and input in a vector of coefficients, which

we named b, that was the values of the numerator and denominator coefficients of our frequency response. After performing the necessary calculations, we plotted our lowpass filter to verify that it not only looked like a lowpass filter, but also determine that the correct processes were occurring at our stopband and passband frequencies. We knew that at our passband frequency, the filter should slowly start to decrease in dB until it hit our stopband frequency. Everything past the stopband frequency we knew should be cut off. Knowing this information, we plotted the lowpass filter using a stop band attenuation of 60, and a stopband and passband edge of 500 Hz and 1500 Hz. As one can clearly see in figure 11, the low pass filter was correctly drawn using the provided inputs. The filter is 0 or close to 0 until our passband frequency and then decreases at an exponential rate until it hits our stopband frequency. Everything past that is clearly cutoff. To perform further testing, we performed another low pass filter test. This time, we used a stop band attenuation of 30, and stopband and passband edges of 1000 Hz and 2000 Hz. The plot of this test is shown in figure 12 and confirms that our lowpass filter is functioning properly. If we wanted the first dip in our lowpass filter to be exactly at 2000 Hz, we would have to use a higher stop band attenuation.

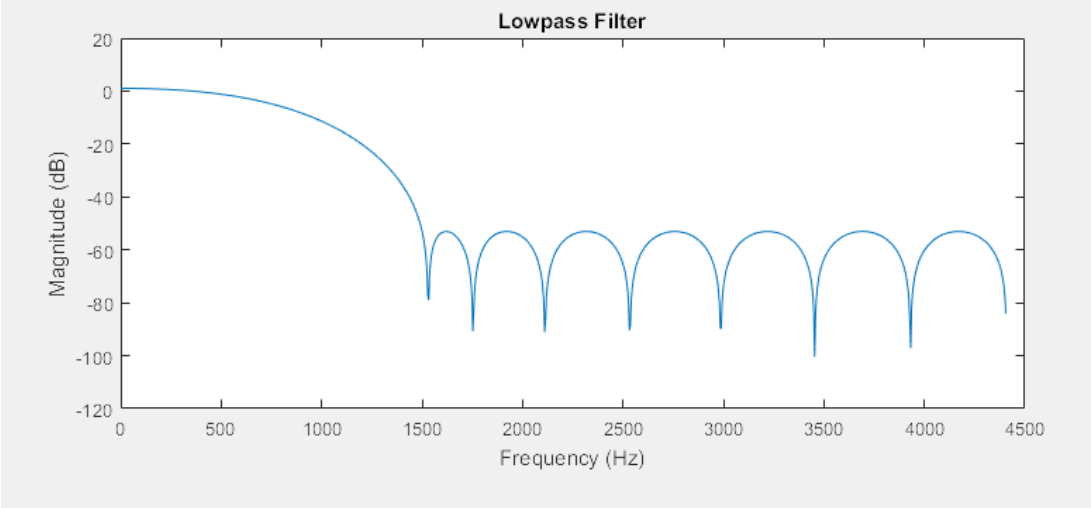

Figure 11: Low Pass filter with a stopband and passband edge of 500 Hz and 1500 Hz

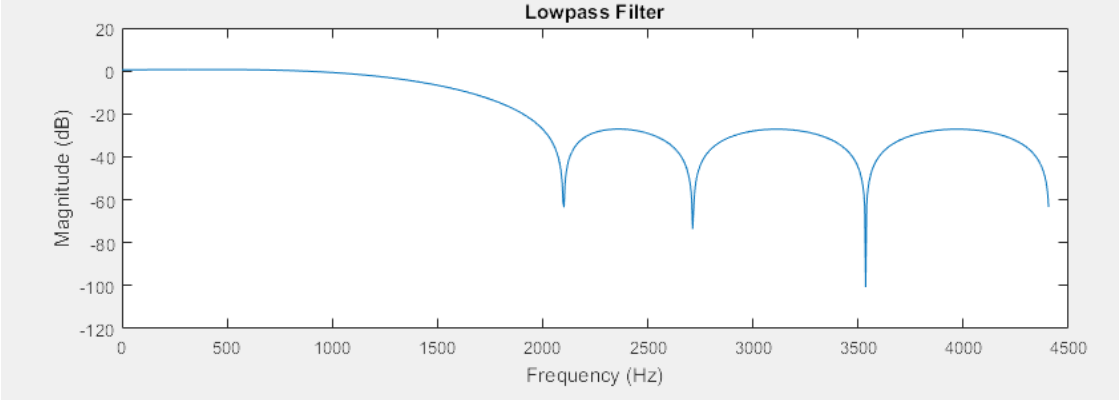

Figure 12: Low Pass filter with a stopband and passband edge of 1000 Hz and 2000 Hz

The second test we put our ID_Song function under was a highpass filter test. The design of the high-pass filter was the same as the lowpass filter design with only two modifications. The two modifications that we had to make was the definition the anti-aliasing filter used and the calculation of variable called “dev”. This variable was used in order to calculate our vector of coefficients, b. Our anti-aliasing filter determines where out frequency cutoff was and since the highpass filter only lets in higher frequencies, we had to flip our anti-aliasing filter. Instead of being high at first and then going low at the stopband frequency, this time the anti-aliasing filter went low at first and then high at higher frequencies. Similar to how we flipped the anti-aliasing vector, we also flipped the value of dev. We did this in order to calculate the correct vector b to use in our calculation of our frequency response. Changing these two variables allowed higher frequencies to pass through our filter and cutoff lower frequencies, as seen in figure 13. In figure 13, we used a stopband attenuation 30, and passband and stopband edges of 1000 Hz and 1500 Hz. To verify that this was not a fluke, we also performed a highpass filter test with a stopband attenuation of 40, and passband and stopband edges of 500Hz and 1000Hz. This graph can be seen in figure 14. We knew our function was working properly because when we increased our stopband attenuation, our graphs became more precise and because only lower frequencies were cut-off..

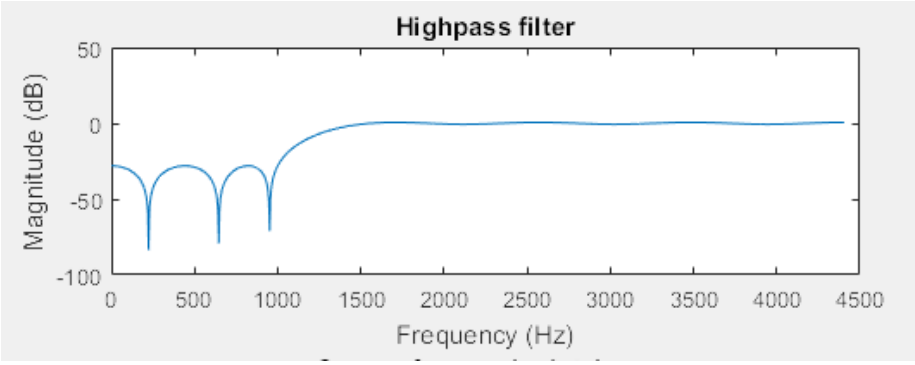

Figure 13: Highpass filter with a stopband and passband edge of 1000 Hz and 1500 Hz

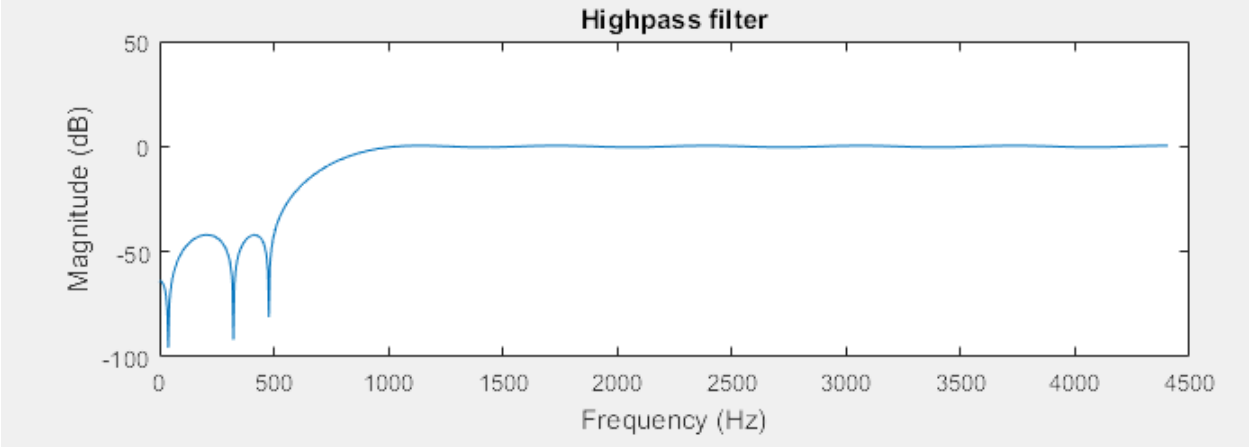

Figure 14: Highpass filter with a stopband and passband edge of 500 Hz and 1000 Hz

The third filter that was created to test our ID_Song function with was a multipath filter. The multipath filter's purpose is to see how our function will react when there is multipath interference. Multipath interference occurs when there are multiple acoustical propagation paths to the microphone that is reading the song. These paths can be created by any object that has a high enough reflection coefficient. This test must be performed to make sure that the multipath interference does not hinder our ID_Song function to badly since it would be nearly impossible to not have multiple acoustical propagation paths when listening to a song. In order to model the multipaths for this filter, we create the vector b instead of calculating it. The first element in vector b must be 1 in order to create the direct path of the signal. Depending on how long one wants the multipath delay to be, the vector b must contain a number of zeros and then finally at the end contain another 1. The second one will model a multipath of the direct path and the number of zeros between the two 1 elements will determine how long the multipath delay is. The output of the filter is another "Score" vector which holds the "Score" values of each song when put through the multipath filter. In order to create the multipath filter, we first had to calculate the response vector, H, and the frequency vector, ff, based upon the vector of coefficients that the user inputs. We then created the multipath signal by applying a filter to b to obtain the transfer function. Using these calculated values, we made a stemplot of b to ensure that b was outputting the right values and a plot of the multipath filter. We created b to have a value of 1 and indices 1 and 14, which can be seen in figure 16. By looking at the graph, it is clear that b is outputting exactly what we want it to. We also made sure that the multipath for this b value was created correctly. This plot can be seen in figure 17. To ensure that our multipath filter is working, we tested it with a different song input and a b vector that goes high at indices 1 and 12. This test can be seen in figure 18 and as one can clearly see, our multipath filter functions correctly.

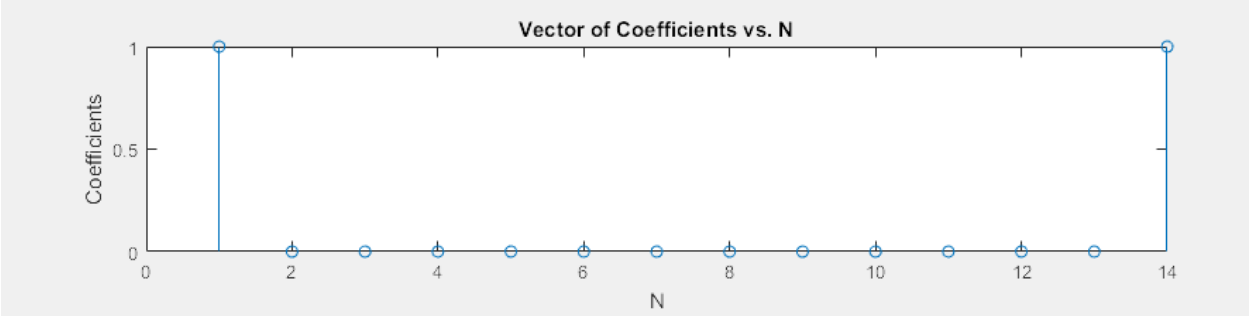

Figure 16: Vector of b coefficients where b=1 at indices 1 and 14, i.e. b= [1 0 0 0 0 0 0 0 0 0 0 0 0 1]

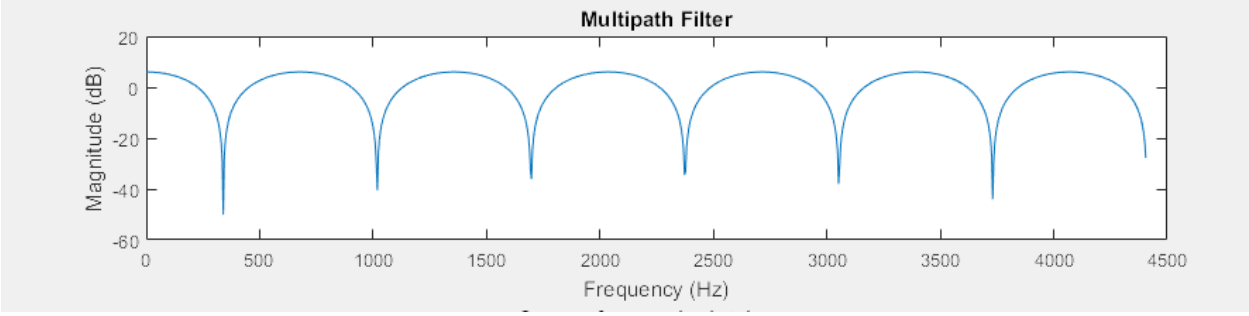

Figure 17: Multipath filter when b=1 at indices 1 and 14.

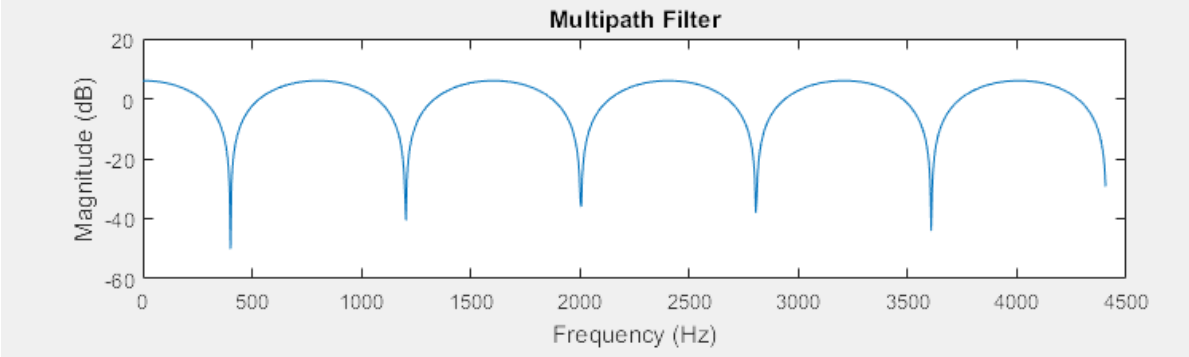

Figure 18: Multipath filter when b=1 at indices 1 and 12.

The fourth and final test that our ID_Song function has to go through is the noise test. In a real world setting, people usually have at least some background noise nearby them. This background noise can be anything from the chatter of people around them, to the sound of a TV in the distance. Either way, in order to understand how well our function would work in the presence of noise, we created a function called "Test_Noise.m" This function differs vastly from the previous three since it does not use a vector b to determine the paths. Instead, this function uses something called a Signal-Power-to-Noise-Ratio (SNR). The SNR value determines the level of noise in dB. The higher the snr value is, the lower the noise will be in our noisy signal. The output argument of this function is once again, the Score vector. The noisy signal used was created by following the differential equation provided labeled (1) where snr was not in dB to ensure that we were using correct units and the Gaussian noise vector is a randomly generated noise signal.

I.e. $y_n = y + \sqrt{\frac{power}{snr}} * (Gaussian\ noise\ vector)$

(1)

To test the noise filter, we plotted the noise free signal on the same graph as the noisy signal with a SNR value of 10 dB. As one can see in figure 19, the noisy vector, y_n, has slightly higher amplitudes than the original vector, y. This is because our SNR value is relatively low. To verify that that this works, we tested the same song with a SNR value of 50 dB in order to decrease the amplitudes of the noisy signal. This graph is shown in figure 20 and the noisy signal was shrunk

so much that the original signal actually has greater amplitudes than the noisy signal verifying that our function works.

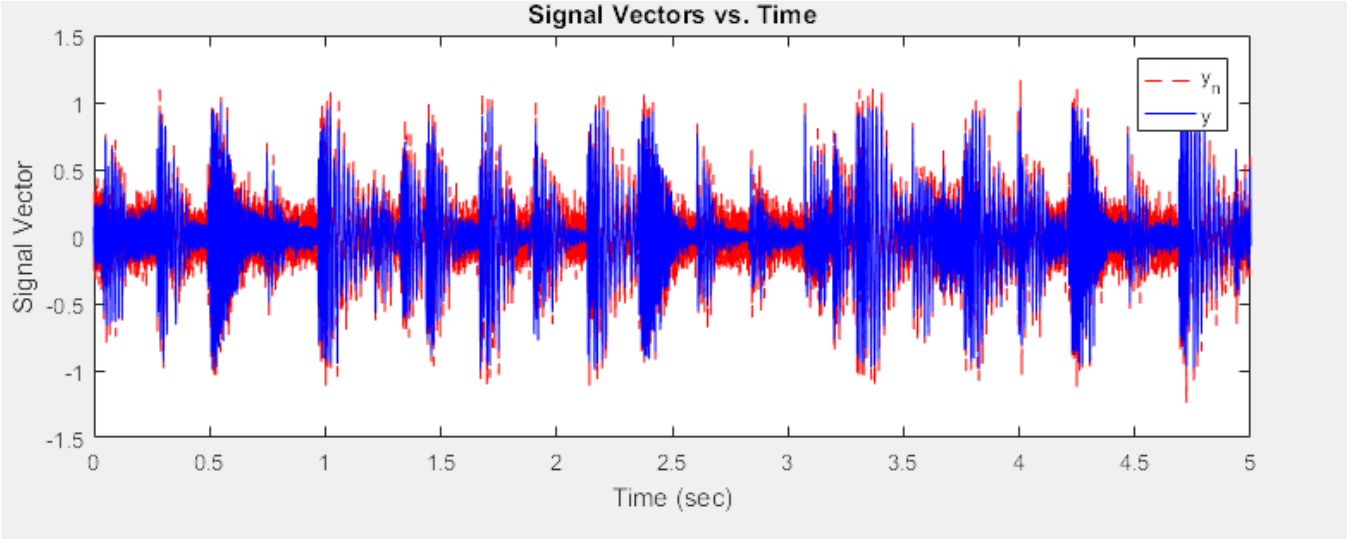

Figure 19: The noisy signal and original signal with a SNR value of 10 dB.

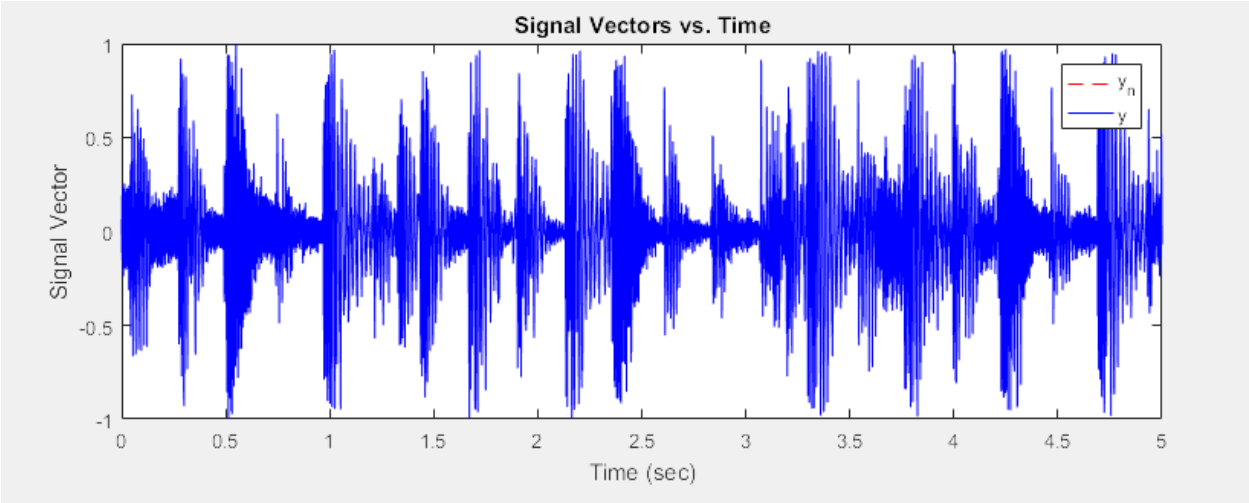

Figure 20: The noisy signal and original signal with a SNR value of 50 dB.

To make testing all how our ID_Song function works under these four conditions, we created a function called “run_tests.m” which just runs each filter once with values that might be found in real-life situations. After running each of the functions, the graphs of each filter appear and look similar to the original ones that we tested, confirming that the filters work under real-life conditions.

Each one of our filters ran correctly and did what we wanted it to do, but what we haven’t talked about yet is how they affected the score vector results of each audio signal. The Score vector in each script held the “score” of each song after going through a given filter. In order to see how the score vector was affected by each filter, we compared the score vector of ID_Song,

which contains the 5-second song snippet without being filtered, to the score vector of all four filters. The Score vector for a given 5-second song snippet, y is shown in figure 21 where we can clearly see that the song snippet perfectly matches up to the song sample in column 10 of our database of songs. The first score vector we decided to compare the output of ID_Song to is the score vector computed after applying a lowpass filter to the song snippet. The score vector of the song snippet after going through the lowpass filter is shown in figure 22. We used a passband and stopband cutoff frequency of 500 and 550 Hz. The stopband attenuation we used is 50 dB. Since a stopband attenuation of 50 dB is relatively high, most of the samples are cut off after applying the lowpass filter. This makes sense since the stopband attenuation of a filter defines where frequencies will be cutoff. This means that any sample that has an attenuation that is less than the stopband attenuation, and a frequency that is less than the between the stopband edge frequency, in the 5-second song snippet is cutoff when the lowpass filter is applied. The magnitudes of samples at frequencies that are not between the stopband and passband cutoff frequencies are either 0, if they are before the passband or cut-off if they are after the cutoff frequency. Any samples that are between the two frequencies are considered to be in the transition state and have magnitudes that are slowly decreasing for a lowpass filter.

```
Trial>> ID_Song(y,P)

ans =

  Columns 1 through 6

         0         0    0.0133         0    0.0133         0

  Columns 7 through 12

         0         0         0    1.0000    0.0067         0

  Columns 13 through 18

    0.0067    0.0067         0    0.0133    0.0133         0

  Columns 19 through 21

    0.0133    0.0133    0.0067
```

Figure 21: Score vector of ID_Song

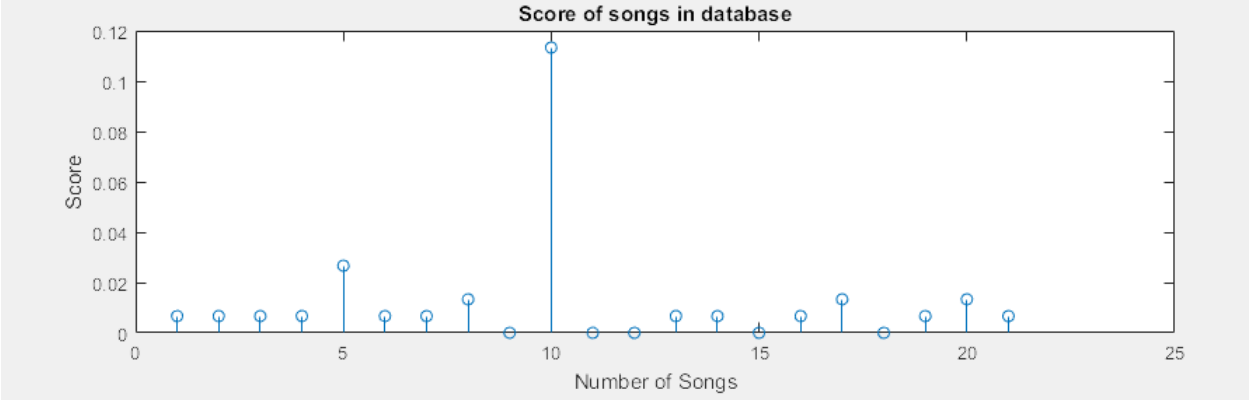

Figure 22: Score vector of lowpass filter

The next score vector we chose to compare the output of ID_Song.m to was the highpass filter. The score vector of the highpass filter is shown in figure 23. We ran the highpass filter command with a stopband attenuation of 30 dB and had the passband and stopband edges set to 300 Hz and 390 Hz. Since the stopband attenuation of this test was lower than the previous test, we see that the score for the correct song is higher than the score for the correct song when run through the lowpass filter. This is because more samples of the signal are not cut off since more samples of the signal have an attenuation higher than 30 dB rather than 50 dB.

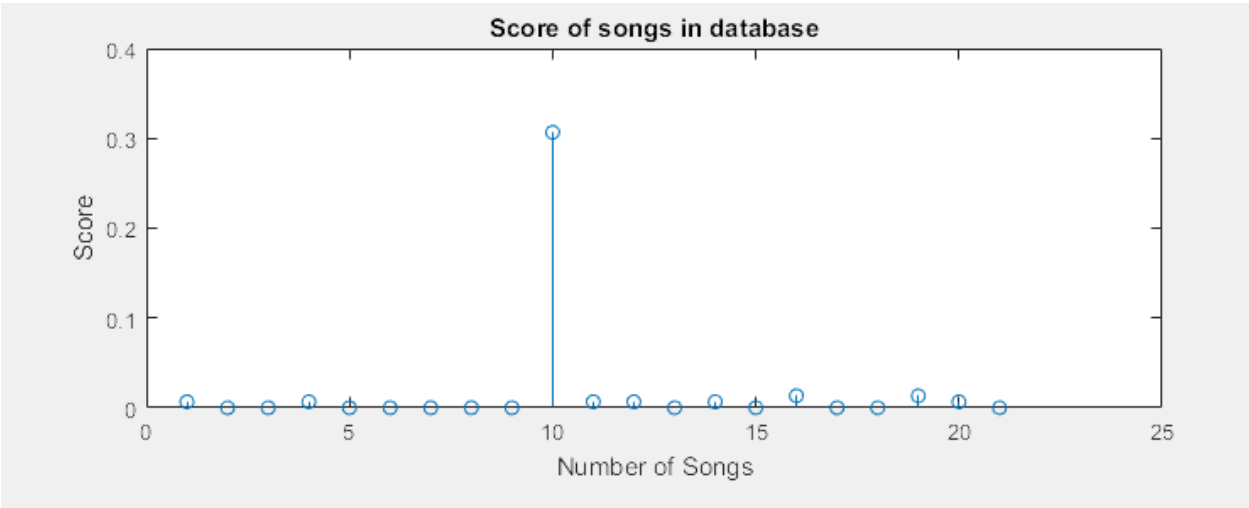

Figure 23: Score vector of highpass filter

We then decided to compare our ID_Song output to the score vector of the multipath filter. We ran our test_mp_filter command with a b vector of [1 zeros(1,850) 1]. This graph can be seen in figure 24. The more zeros that are put into vector b, the higher the score value will be for the multipath filter. This is because b is supposed to model the multipath and direct path for the filter. The direct path will always be the 1st index which is why the first element of b is always 1. The amount of zeros determines how long the multipath delay is and the second 1, after all the zeros, indicates the point where the second multipath delay is. Since there are 852 vector b coefficients, and the sampling rate of our signal is 8820 $\frac{samples}{second}$ **,** the multipath filter will

have a delay of $\frac{852\ samples}{8820\ samples/second}$. The longer the delay of the multipath filter, the higher the score will be because the audio signals then have longer to get to the correct location and more paths can be read.

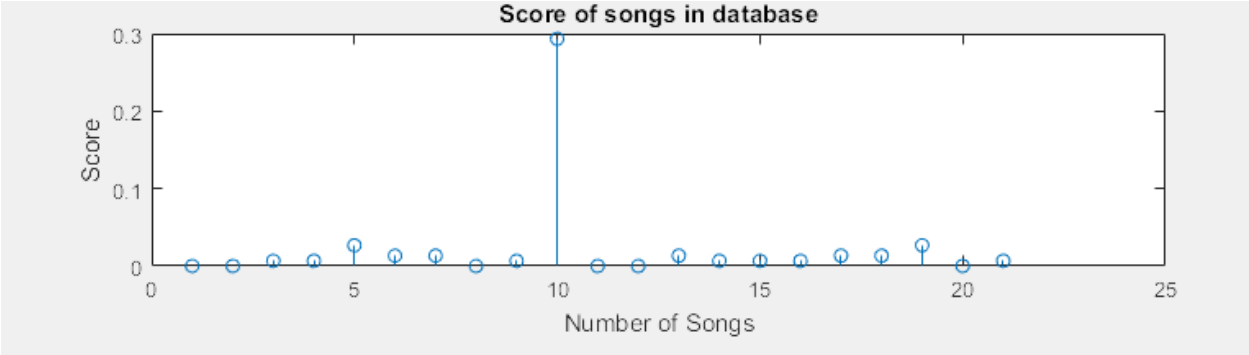

Figure 24: Score vector of multipath filter

Finally, we compared the score of the test_noise function to the output of the ID_Song. We ran our test with a snr value of 10. This graph can be seen in figure 25. Since our value of snr is small there will be a lot of noise in our audio signal. If we wanted to decrease the noise, we would have to increase our snr value until the snr is high enough where the audio signal has greater amplitudes than the noise. Our score vector is relatively high but since there is still some noise present in our corrupted signal, the score vector is not equal to one. Once the audio signal's amplitudes surpass the noisy signal's amplitudes, our score vector will equal 1.

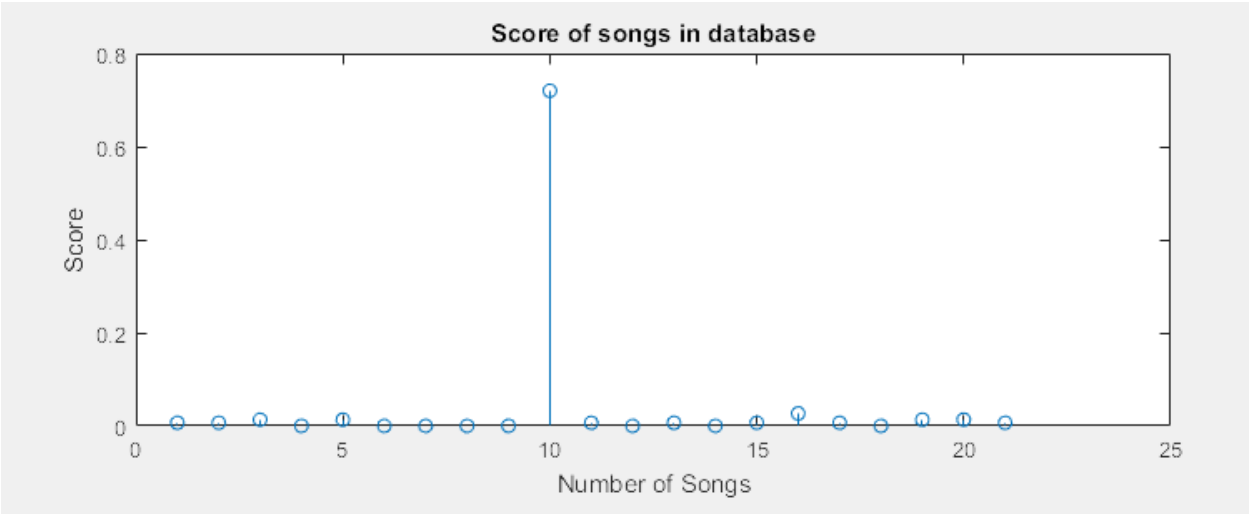

Figure 25: Score vector of noisy filter

Completing this project was an immense feat and a valuable learning experience. The project was successfully completed with all the specifications met. While our program only has a database of 21 songs, we were able to accomplish a lot with the information we learned from class. We saw the effects of the filters, based upon the generated score plots, and its effect on the signals. Consequently, we saw in figure 26 how the time domain relates to the frequency domain through this project. For each filter we created, we performed tests to ensure that after putting our original signal, x(t), through a filter that we would still obtain a valid score plot for our y(t). When we passed our input signal through a lowpass filter and plotted its score, as seen in figure 22, we were able to determine that the filter was correctly working. Even though our score was relatively low when the song snippet was compared to the song in the database, it was high enough for us to determine that our ID_Song function was still recognizing the correct song. After putting the same signal through a highpass filter, we saw it had a similar effect as a lowpass filter on the score plot. The reason why our score plot was higher in figure 23 than in figure 22 was because of the stopband attenuation level. When we run the highpass filter with the same parameters as the lowpass filter seen in figure 22, we obtain a score vector that can be seen in figure 28. Here we can clearly see that when the same parameters are put in for the highpass and lowpass filter, a very low score is obtained when the song snippet matches the song. This is because most of the samples at a high attenuation are cut-off. Although our multipath filter score, as seen in figure 24 was higher than our lowpass filter score, it was still relatively low. Even though we added in a good delay for the multipath filter, the score was still relatively low because there are *many possible paths* that the audio signal can take before reaching the microphone. Increasing the delay will increase the score of our multipath filter but if one wants to have the score be very high, the multipath delay will have to be at least a couple of seconds, not milliseconds. When doing our tests, the best score we actually obtained was from the noisy signal. This is because we had a positive snr value. If our snr value was 0 or lower, our score vector would be very low. However, we also do not need a very high snr value in order to make the noise completely disappear and make the score vector equal 1 when it compares the 5-second song input to the correct song. Once we were able to confirm that all our filters worked properly, we decided to test input signal for stability. Since all of our filters and our original input were non-recursive, we were able to determine that our inputs had no feedback. If there is no feedback in a difference equation, then the frequency response will have no roots in the denominator, which means that they are stable. This is because stability is determined based upon the location of the poles, and since all the poles were determined to be located at the origin, all of our inputs were stable.

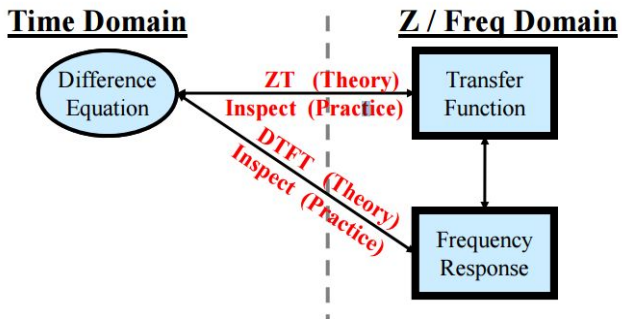

Figure 26: The BIG IDEA

Non-Recursive Filters… Have **No Feedback** in the Difference Equation

$$y[n] = b_0 x[n] + b_1 x[n-1] + \ldots + b_M x[n-M]$$

Figure 27: Non-Recursive FIR filter

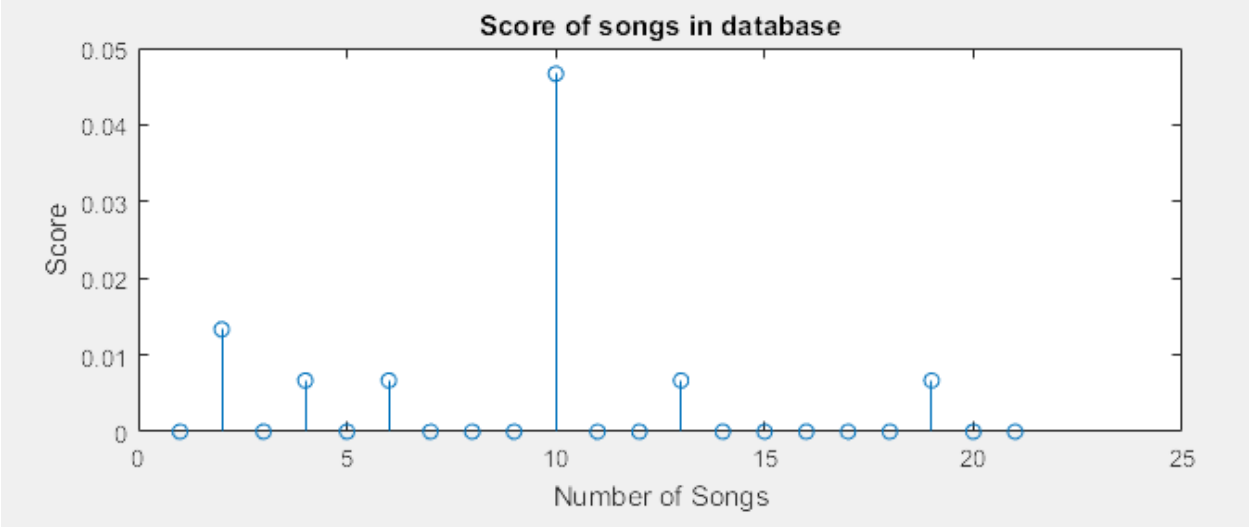

Figure 28: Highass filter score