COMP9336 Assignment Report

Student ID: z5195349 Name: WENXUN PENG

TarsosDSP

TarsosDSP is the API I use in this Assignment. I will briefly introduce this API first. TarsosDSP is a Java library for audio processing. Its aim is to provide an easy-to-use interface to practical music processing algorithms implemented, as simply as possible, in pure Java and without any other external dependencies. The library tries to hit the sweet spot between being capable enough to get real tasks done but compact and simple enough to serve as a demonstration on how DSP algorithms works. TarsosDSP features an implementation of a percussion onset detector and a number of pitch detection algorithms: YIN, the Mcleod Pitch method and a “Dynamic Wavelet Algorithm Pitch Tracking” algorithm. Also included is a Goertzel DTMF decoding algorithm, a time stretch algorithm (WSOLA), resampling, filters, simple synthesis, some audio effects, and a pitch shifting algorithm.

Next, I'll introduce some of the important functions I've used.

① AudioRecord.getMinBufferSize(sampleRateInHz, channelConfig, audioFormat);

AudioRecord: The AudioRecord class manages the audio resources for Java applications to record audio from the audio input hardware of the platform.This is achieved by "pulling" (reading) the data from the AudioRecord object.

getMinBufferSize: Returns the minimum buffer size required for the successful creation of an AudioRecord object, in byte units.

Parameters:

sampleRateInHz (int): the sample rate expressed in Hertz.

channelConfig (int): describes the configuration of the audio channels. (1 is #CHANNEL_IN_DEFAULT)

audioFormat (int): the format in which the audio data is represented.(2 is AudioFormat#ENCODING_PCM_16BIT.)

② AudioDispatcherFactory.fromDefaultMicrophone(sampleRate, audioBufferSize, bufferOverlap);

This function creates a new AudioDispatcher connected to the default microphone. The default is defined by the Java runtime by calling AudioSystem.getTargetDataLine(format). The microphone must support the format of the requested sample rate, 16bits mono, signed big endian.

AudioDispatcherFactory: The Factory creates **AudioDispatcher** objects from various sources: the configured default microphone, PCM wav files or PCM samples piped from a sub-process. It depends on the javax.sound.* packages.

AudioDispatcher: This class plays a file and sends float arrays to registered **AudioProcessor** implementors. This class can be used to feed FFT's, pitch detectors,

audio players, ... Using a (blocking) audio player it is even possible to synchronize execution of AudioProcessors and sound. This behavior can be used for visualization.

AudioProcessor: AudioProcessors are responsible for actual digital signal processing. The interface is simple: a process method that works on an AudioEvent object. The AudioEvent contains a buffer with some floats and the same information in raw bytes. AudioProcessors are meant to be chained e.g. execute an effect and then play the sound. The chain of audio processor can be interrupted by returning false in the process methods.

Parameters:

sampleRate (int) - The requested sample rate must be supported by the capture device.

audioBufferSize (int) - The size of the buffer defines how much samples are processed in one step. Common values are 1024,2048.

bufferOverlap (int) - How much consecutive buffers overlap (in samples). Half of the AudioBufferSize is common.

③ Goertzel (audioSampleRate, bufferSize, frequencies, Goertzel.FrequenciesDetectedHandler handler)

Goertzel: Contains an implementation of the Goertzel algorithm. It can be used to detect if one or more predefined frequencies are present in a signal. E.g. to do DTMF decoding.

Parameters:

AudioSampleRate (float): The requested sample rate must be supported by the capture device.

BufferSize (int): The size of the buffer defines how much samples are processed in one step.

frequencies (double[]): The frequencies will be detected.

Goertzel.FrequenciesDetectedHandler handler: An interface used to react on detected frequencies. Its detail is showed below:

void handleDetectedFrequencies(double[] frequencies, double[] powers, double[] allFrequencies, double[] allPowers)

frequencies - A list of detected frequencies.

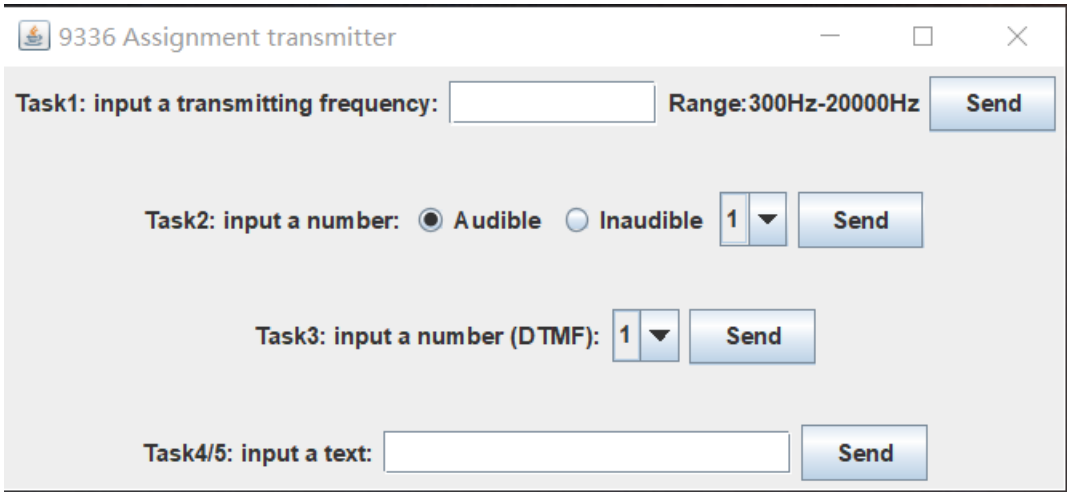
powers - A list of powers of the detected frequencies.

allFrequencies - A list of all frequencies that were checked.

allPowers - A list of powers of all frequencies that were checked.

Task 1 Single Tone Detection

In task1, I first wrote a simple Java-based interface for running on a laptop as a sender. The sender sends a single sinusoidal tone of a given frequency (frequency can be input through the interface).



For task1, as shown in the figure above, we can input a frequency range from 300 Hz to 20000 Hz (which is also the frequency range that my mobile device can detect), and then click "send" to play the sinusoidal tone of this frequency.

As for the receiver, I use the Goertzel algorithm for frequency identification. The principle of the Goertzel algorithm is as follows:

First, we can choose some variables:

① Decide on the **sampling rate, R**.

② Choose the **block size, N**.

③ Precompute one cosine and one sine term.

④ Precompute one coefficient.

These can all be precomputed once and then hardcoded in your program, saving RAM and ROM space; or you can compute them on-the-fly.

Sampling rate

As for how to determine the sampling rate, we need to follow the usual Nyquist rules apply: the sampling rate will have to be at least twice your highest frequency of interest. Because the frequency range we want to measure is between 300 Hz and 20000 Hz, our sampling rate is at least 40000 Hz. Here I chose the sampling rate of 44100Hz, because this sampling rate can even meet some DVD sound quality requirements. This sampling rate can also satisfy the frequency of 300 Hz-20000 Hz we want to search for.

Block size

Goertzel block size N is like the number of points in an equivalent FFT. It controls the frequency resolution (also called bin width). For example, if sampling rate is 8kHz and N is 100 samples, then the bin width is 80Hz.

This means that if the N is higher, the higher frequency resolution we can get. But the higher the N, the longer it takes to detect each tone, because the waiting time for detecting the tone is longer so that all samples can be entered. For example, at 8kHz sampling, it will take 100ms for 800 samples to be accumulated. If trying to detect tones of short duration, we will have to use compatible values of N. The other factor influencing choice of N is the relationship between the sampling rate and the target frequencies. Ideally the frequencies should be centered in their respective bins. In other words, the target

frequencies should be integer multiples of (sampling rate/N).
However, unlike the FFT, N doesn't have to be a power of two.

Precomputed constants

Once the sampling rate and block size are selected, it's a simple five-step process to compute the constants during processing:

$$K = \frac{Nf}{R}$$

K denotes the number of complete cycles of the detection section containing the target frequency. In addition, the K value should be an integer and be of the right size. If it is too large, it is not conducive to the time of the detection (It may cost much more time), but if it is too small, the detection may not be accurate. If we can't get an integer K, we may find the one with the least rounding error.
With the above parameters, we can calculate the radian ω of each sample in a period of the target frequency. Since N samples express K periods (2π), and then the ω should be calculated as follows:

$$\omega = \frac{2\pi K}{N}$$

It should be noted that since the K value may have been rounded, the above two formulas must be calculated one after another, and must not be combined to simplify K.
And then, we can get the coefficients C:

$$C = 2cos(\omega) = 2cos(\frac{2\pi K}{N})$$

Applying Goertzel algorithm

The above parameters can be calculated in advance without re-calculating in each sampling analysis. After that, we began to analyze and calculate N samples.
First initialize:

$$Q_1 = 0$$
$$Q_2 = 0$$

And then, according to the sequence, the following calculations are made for each value S of N samples:

$$Q_0 = CQ_1 - Q_2 + S$$
$$Q_1 = Q_0$$
$$Q_2 = Q_1$$

After the above calculation is completed, we can get the energy value P of the frequency f reflected in the N samples:

$$P = Q_1^2 + Q_2^2 - CQ_1Q_2$$

In fact, P is the square of energy, we can get the real energy value by extracting a root from P. In addition, we can set a threshold for P. The frequencies whose energy value is larger than this threshold can be used. Furthermore, we can make the largest value of power corresponding to frequency as our detected frequencies.

Task 2 Extension of Single Tone Detection

In task2, I tried to implement Goertzel algorithm by other method to distinguish frequency. I found that this method can distinguish frequency more quickly than TarsosDSP. Therefore, I used this method in task2 and task3, but in task4, when there are two consecutive frequencies, there are always omissions. Thus, in task4, I still make use of TarsosDSP API. Task 2 needs to find two hard-coded frequency range versions (audible and inaudible), so that the frequency of a number is sent to the sender and detected by the receiver, and then learned from the predefined rules.

To make the Goertzel algorithm work better, I make these frequencies 100 hertz apart (if the frequencies are too close to each other, the Goertzel algorithm may not work properly). Because we can detect frequencies from 300 to 20000 hertz from my phone, I chose the audio version from 450 to 1250 hertz, the inaudible version from 13150 hertz to 13950 hertz, which is hard to distinguish from human ears (the range of human ears is 20 hertz to 20000 hertz, which varies with age).

Specifically, I let 450Hz stand for 1, 550Hz stand for 2, 650Hz stand for 3, and so on. Similarly, in the inaudible frequency version, 13150 Hertz stands for 1, 13250 Hertz stands for 2... In fact, my initial version set the high frequency above 16150 Hz, but I found that my laptop speaker seems to have some problems, it does not always work properly, so I set it to 13150 Hz. However, it's still a frequency that's hard to hear in someone's ear.

Task 3 Dual Tone Detection

Task 3 and Task 2 both use my own Goertzel algorithm instead of Tarsos DSP, because I found that my own Goertzel algorithm can distinguish frequencies faster to distinguish frequencies. The difference is that in Task 3, a specific frequency combination is needed to achieve number 1-9 transmission. In order to judge which number it is faster, I used the sum of two frequencies to quickly determine the number (for example, 697Hz and 1209Hz represent 1, and their sum is 1906. If the sum of the two frequencies received is 1906Hz, and then it represents 1. Each number corresponds to only one frequency combination, and the different frequencies combination corresponding to the numbers are different, so this method is feasible. In addition, when we used Goertzel algorithm, we should search the two largest powers corresponding to frequencies as our detected frequencies since the tone has two frequencies.

	1209	1336	1477
697	1	2	3
770	4	5	6
852	7	8	9

Task 4 Packetized Data Communication with Audio Tones

In Task 4, I still use TarsosDSP API, because I found that TarsosDSP API can achieve Task4 relatively well. The difference of Task4 from the previous tasks is that it needs to create a new thread to process the received audio data. Otherwise, it will miss some frequencies when it receives other frequencies because it consumes too much resources to process the data. This ultimately affects the efficiency of transmission. Therefore, we should create a new thread to process the audio data.

I use ASCII code for data processing. For example, if we want to transfer a character, such as ‘A’, we will first convert it into ASCII code corresponding binary form (character ‘A’ corresponding ASCII code is 65, and the corresponding binary is 0100 0001), and then add 4-bit check bits to it (Hamming code, further explained in Task5). These will form 12 bits data, transmitted to the receiver.

Since I found it difficult to quickly distinguish the tones of continuous pronunciation (the same tone may be repeatedly received or directly omitted), I had to use 64 frequencies for transmission. As mentioned earlier, I convert each character into 12 bits of data for transmission (8 bits of data plus 4 bits of check bits) and then divide the 12 bits into 6 bits of high and 6 bits of low. Each 6 bits of binary number corresponds to a decimal number (ranging from 0 to 64). And then I use 400 Hz as the cardinal number and take a frequency every X Hz. That means the frequency used is always satisfied: 400 + X * i, where i corresponds to decimal numbers (range 0-64). In this way, at the receiver, we can calculate i by using the received frequency Y (i = (Y-400)/X), so that i can be restored, and then decoded, and the transmitted data can be obtained.

I wanted to use only 16 frequencies (i.e. Dividing 12 bits of data into three parts instead of two parts, each part has only 4 bits, which is the same principle as above. That can only use 16 frequencies for transmission), but I found that such transmission efficiency is low, so I used the 64 frequencies to transmit.

Task 5 Error correction

In the part of error detection and correction, I decided to use Hamming code. If we want to understand the principle of Hamming code, we need to know "parity check" first.

Parity check: We specify that the number of 1 in a sequence of codes is even. If the binary codes of 0 and 1 carry even numbers of 1, the information is right, otherwise it is wrong.

For example, we can add a parity check code to the code at the beginning to realize parity check. If we want to transmit the serial code 10010, we can transmit 010010, where the zero at the beginning is the check bit. Another example is that we want to transmit the code 10000, we can transmit 110000 where the first one is the check bit. In the above two examples, the number of 1 is even.

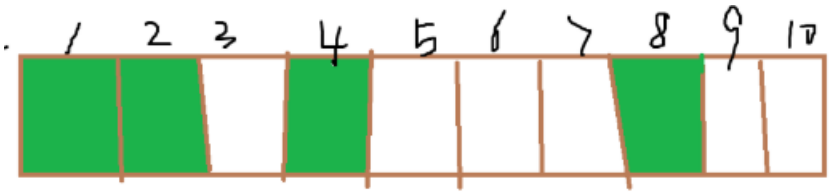
Next we can talk about Hamming code.

Hamming codes are codes with parity check. It uses a clever way of grouping the numbers and checking them by grouping to determine which one has made a mistake. Hamming code assumes that there is only one wrong bit in a series of data, so using Hamming code,

we can only detect and correct one error at most.

When using Hamming code, we put the check bits on the position of the i-th power of 2. Check bits are 1 or 0, so that the number of 1 in the group where the check bits are located is even.

As shown in the figure:



The green position is where the check bits are placed, 1,2,4,8,16... where the i-th power of 2 is.

So how to group? Hamming Code Provisions:

Where the location conforms to this form, XXX1 is assigned to P1.

Where the location conforms to this form, XX1X belongs to P2.

Where the location conforms to this form, X1XX is assigned to P3.

Where the location conforms to this form, 1XXX is assigned to P4.

Obviously, the check bits are also divided into groups, and each group has only one check code. For example:

The number of first position becomes the 0001 position.

The number of second position becomes the 0010 position.

The number of third position becomes the 0011 position.

The number of fourth position becomes the 0100 position.

The number of fifth position becomes the 0101 position.

The number of sixth position becomes the 0111 position.

…

In this Assignment, since I used ASCII code to encode each corresponding character, every character has 8 bits. Thus, if we want to determine the number of Hamming codes, we should know: if we assume that the check bits have a total of K bits, and the original data bits we want to transmit have a total of n bits, and the k-bit check bits can detect 2 ^ k bits, so the number of check codes should satisfy this formula:

2 ^ k > k + n, or 2 ^ k - 1 > = k + n.

So according to the ASCII code of 8 bits, we can use the Hamming code of 8 data bits plus 4 bits of check bits. Therefore, the code totally has 12 bits (8 data bits and 4 check bits). This also explains why I code this way in task4.

For example:

If we want to transmit character ‘A’, whose ASCII code is 65 (equal to 0100 0001 in binary), we can add Hamming check bits in corresponding position. Binary codes corresponding to 'A' should be split up XX0X100X001(X is check bits). Therefore, we can know the result is 100010 010001, which is equal to 34 17. Thus, the task4 can transmit these two numbers.

Finally, let's discuss how to decode:

Step1: Grouping, dividing P1, P2, P3... (like sender doing).

Step2: Check each group separately. Give it 0 without error and 1 with error.

Step3: Arrange P from high position to low position and get a string. For example:

Group: P5 P4 P3 P2 P1
Sign: 1 0 1 0 0

Let's continue with the example of sending 'A', which should have sent 34 and 17, assuming that 35 and 17 (actually the sixth bit out of 12 bits occurs error) are sent since a 1-bit-error occurs. That means the receiver will get 100011 010001. After grouping and calculating, we can get:

Group: P4 P3 P2 P1
Sign: 0 1 1 0

Now, we can calculate the decimal number corresponding to 0110, which is 6. This 6, which corresponds to the position of the bit where the error occurred, only needs to subtract 1 to get the corresponding index. In addition, if this sum is not 6, nor other numbers but 0, it means that there are no bit errors. Since Hamming codes can only detect and correct one bit error, we will not consider two or more bit errors. Here's a part of the code I wrote by implementing this algorithm, which also is the last task I completed.

```
1. int check_1 = even_one(decoding_bits.get(2) + decoding_bits.get(4) + decodin
   g_bits.get(6) + decoding_bits.get(8) + decoding_bits.get(10));
2. int check_2 = even_one(decoding_bits.get(2) + decoding_bits.get(5) + decodin
   g_bits.get(6) + decoding_bits.get(9) + decoding_bits.get(10));
3. int check_3 = even_one(decoding_bits.get(4) + decoding_bits.get(5) + decodin
   g_bits.get(6) + decoding_bits.get(11));
4. int check_4 = even_one(decoding_bits.get(8) + decoding_bits.get(9) + decodin
   g_bits.get(10) + decoding_bits.get(11));
5. ArrayList<Integer> errorfinding = new ArrayList<Integer>();
6. if (check_4 != decoding_bits.get(7)) {
7.     errorfinding.add(1); }
8. else if(check_4 == decoding_bits.get(7)){
9.     errorfinding.add(0); }
10. if (check_3 != decoding_bits.get(3)) {
11.     errorfinding.add(1);}
12. else if (check_3 == decoding_bits.get(3)) {
13.     errorfinding.add(0); }
14. if (check_2 != decoding_bits.get(1)) {
15.     errorfinding.add(1); }
16. else if (check_2 == decoding_bits.get(1)) {
17.     errorfinding.add(0); }
18. if (check_1 != decoding_bits.get(0)) {
19.     errorfinding.add(1); }
20. else if (check_1 == decoding_bits.get(0)) {
21.     errorfinding.add(0); }
22. int corret_flag = 8 * errorfinding.get(0) + 4 * errorfinding.get(1) + 2 * er
   rorfinding.get(2) + errorfinding.get(3);
23. if(corret_flag == 0){     // no bits error
24.     return BinaryAsciiToString(decoding_bits); }
25. else {
```

```
26.     if(decoding_bits.get(corret_flag-1) == 0){
27.         decoding_bits.set(corret_flag-1,1); }
28.     else if(decoding_bits.get(corret_flag-1) == 1) {
29.         decoding_bits.set(corret_flag-1, 0); }
```

Reference

[1] codeday. 2018. Java Beep. [ONLINE] Available at: https://codeday.me/bug/20181112/376662.html. [Accessed 25 July 2019].
[2] embedded. 2002. The Goertzel Algorithm. [ONLINE] Available at: https://www.embedded.com/design/configurable-systems/4024443/The-Goertzel-Algorithm. [Accessed 30 July 2019].
[3] CSDN. 2018. Hamming Code. [ONLINE] Available at: https://blog.csdn.net/Yonggie/article/details/83186280 [Accessed 30 July 2019].
[4] GeeksforGeeks. 2018. Computer Network | Hamming Code. [ONLINE] Available at: https://www.geeksforgeeks.org/computer-network-hamming-code/[Accessed 30 July 2019].
[5] 0110.be. 2016. TarsosDSP API 2.4. [ONLINE] Available at: https://0110.be/releases/TarsosDSP/TarsosDSP-latest/TarsosDSP-latest-Documentation/ [Accessed 30 July 2019].
[6] Github. 2014. TarsosDSP. [ONLINE] Available at: https://github.com/JorenSix/TarsosDSP [Accessed 30 July 2019].
[7] CSDN. 2019. AudioRecord. [ONLINE] Available at: https://blog.csdn.net/u010126792/article/details/86309592[Accessed 30 July 2019].
[8] Android.developer AudioRecord [ONLINE] Available at: https://developer.android.com/reference/android/media/AudioRecord[Accessed 25 July 2019].