Easy Tuner 1.0 - Guitar & Violin tuning application

R.A.C.L.Mendis

Department of Computer Science &Engineerin, University of Moratuwa

Sri Lanka

cresclux.10@cse.mrt.ac.lk

Abstract **– This paper presents the main steps followed during the development of Easy Tuner, a brief discussion on each step, justifications for the choices made and most importantly, special/different methodologies used in the application.**

Index Terms - Audio Record, Media Recorder, Zero Crossing Method, Fast Fourier Transform, Pulse Code Modulation

I. Introduction

Easy Tuner is a guitar and violin tuning application for the Android platform. Even though there are guitar tuning applications in the Google Play Store, most of them lack usability, which is one of the most important requirements. Therefore, the basic idea of this application is to provide a user-friendly interface that makes this task easy, even for a beginner. The following sections will provide an overview of the main activities carried out.

II. Selecting A String

A. Select an instrument

Initially, the user is asked to select either guitar or violin as the instrument. After the selection is made, the user will be shown a screen with an image of the strings of the selected instrument.

B. Select a string

By touching a string, the user can proceed to tune that string. Soon after the touch event, the application will display a dialog box, which requires confirmation whether the user wants to proceed with tuning or not. This is to avoid responses for unintended touch events that may occur while utilizing the application.

C. Specific points to note

The images of guitar and violin strings appear as background images. Therefore, it would be compatible in any Android device.

On the other hand, by using the *Android view API method getWidth()* (which was deprecated in API level 13) allows the application to get the width of the display and calculate the x coordinate of each string based on a pre-determined scale (on a specific Android device). Since the touch event cannot be restricted to a particular x and y coordinate, this application allows 20 units to either side of every string. Touching in between strings won't create a dialog box as mentioned above.

For instance, if the width of Nexus 4 display is 480 units, x-coordinate of high E/E4 string is 424 (pre-determined scale). Therefore, if the width returned from the *getWidth()* method is stored in *width,* the respective region for a particular string is calculated as follows;

[width * (424/480) - 20] && [width * (424/480) + 20]

III. Getting The Input From Microphone

In Android, there are two classes in the *Android Media API* which can be used to collect the input from a microphone. One is the *AudioRecord* class and the other is *MediaRecorder* class. But both of these classes have different applications.

A. AudioRecord

If the user needs to perform analysis while a recording is still in progress, you need to use *AudioRecord*. At the time of creating an instance of the *AudioRecord* class, we need to define the audio source, sample rate, channel configuration, Audio Format and the buffer size in bytes. After calling *startRecording(),* you need to poll the data yourself from the *AudioRecord* instance using the method *read().* Also, you must read and process the data fast enough so that the internal buffer will not be overrun. In simple terms, *AudioRecord* just gives you the raw sound stream and you have to compress it by yourself.

B. MediaRecorder

MediaRecorder is a black box which gives compressed audio file on the (as mp3, wav and etc.) output.

C. Choice for Easy Tuner

In most cases *MediaRecorder* is the best choice except those in which some complicated sound processing is made, and access to the raw audio stream is needed. Therefore, upon the need to process the sound, *AudioRecord*was selected.

D. Recording time

Starting from the first time the user touches the ‘Start Recording’ button, the application enters into a while loop and takes the input from the microphone for a duration of 6 seconds. At the end of each while loop, it analyzes the frequency of each set of buffer values, encoded in PCM.

E. Challenges

In most of the available guitar tuning applications, the input is collected without any ‘start recording’ button. However, in Easy Tuner, recording and analyzing processes start soon after touching the ‘start recording’ button. Therefore, it can be considered as a potential drawback of application, compared to other similar software.

On the other hand, Easy Tuner does not have a function/method to filter the background noise, of the input signal. Therefore, the application assumes that the user is in an environment where background noise is minimal.

IV. Analysing The Waveform

In order to find out the frequency of the input signal, various algorithms can be used. Few examples being Zero crossing method, FFT, phase-locked loops, delay-locked loops, auto correlation or an intelligent combination of these methods. Below is a brief introduction to the methods that were developed, and reasons for discarding.

A. Zero-crossing method

A "zero-crossing" is a point where the sign of a function changes (e.g. from positive to negative), represented by a crossing of the axis (zero value) in the graph of the function. Counting zero-crossings is also a method used in speech processing to estimate the fundamental frequency of speech.

Therefore, during the first phase, a Zero-crossing algorithm was developed to analyze the frequency. However, due to the lack of background noise filters and not using a complex analytic signal [1], a correct frequency was never achieved.

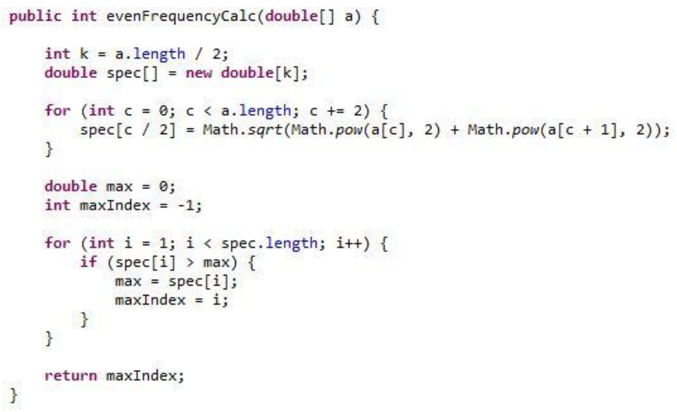
B. Fast Fourier Transform

Due to the failure of the Zero-crossing method, a more accurate method named FFT was found. However, due to the complexity of the algorithm, implementing it from scratch was hard.

Therefore, after careful research I found a library named *jtransforms* [2], which is capable of signal analysis in Java. *jtransforms* is the first, open source, multithreaded FFT library written in pure Java.

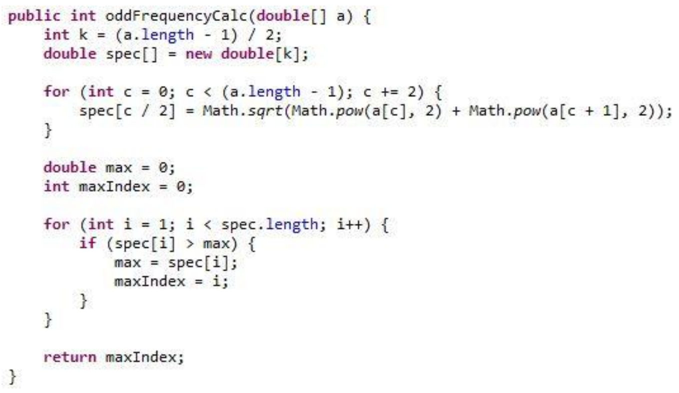
By instantiating a DoubleFFT_1D object and calling the *realForward()* function along with the array of short elements, it writes complex values to the same array. Then this array is used to calculate the fundamental frequency of that set of buffer values. This is done at the end of each while loop, until 6 seconds run out.

Below are the code segments showing the steps in calculating the maximum index of the complex array, when the buffer size is even/odd (parameter *double[] a* is the complex array).



```java
public int evenFrequencyCalc(double[] a) {

    int k = a.length / 2;
    double spec[] = new double[k];

    for (int c = 0; c < a.length; c += 2) {
        spec[c / 2] = Math.sqrt(Math.pow(a[c], 2) + Math.pow(a[c + 1], 2));
    }

    double max = 0;
    int maxIndex = -1;

    for (int i = 1; i < spec.length; i++) {
        if (spec[i] > max) {
            max = spec[i];
            maxIndex = i;
        }
    }

    return maxIndex;
}
```

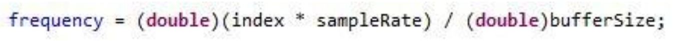
Fig. 1 Java code segment to calculate the maximum index of the complex array when buffer size is even



```java
public int oddFrequencyCalc(double[] a) {
    int k = (a.length - 1) / 2;
    double spec[] = new double[k];

    for (int c = 0; c < (a.length - 1); c += 2) {
        spec[c / 2] = Math.sqrt(Math.pow(a[c], 2) + Math.pow(a[c + 1], 2));
    }

    double max = 0;
    int maxIndex = 0;

    for (int i = 1; i < spec.length; i++) {
        if (spec[i] > max) {
            max = spec[i];
            maxIndex = i;
        }
    }

    return maxIndex;
}
```

Fig. 2 Java code segment to calculate the maximum index of the complex array when buffer size is odd

By using the maximum index returned by the above functions, the fundamental frequency is calculated as follows.



```java
frequency = (double)(index * sampleRate) / (double)bufferSize;
```

Fig. 3 Java code segment to calculate the fundamental frequency of a set of buffer values

V. Making The Final Decision

At the end of each loop, after calculating the fundamental frequency, it is compared with the standard frequency of that particular string. When comparing, a +1.52% and -1.52% error is allowed. Based on the comparison, a global string variable named *decision* is set to ‘matched’, ’high’ or ‘low’. Then a count is kept for each type of string.

In a test with 50 samples, it was observed the count of ‘matched’ string is at least equal to 1 when the string is tuned. and 0 otherwise.

As the while loop ends, the application will display a notification indicating whether the string is tuned or not. If it is not tuned, the notification will display whether the string should be tightened or loosened (based on the relationship of frequency and tension).

Thereafter, the user can carry out the necessary action and then touch the ‘start recording’ button and proceed as earlier until the string is tuned. Or else, the user could also go back to the selecting Guitar/Violin string or main menu, and change the instrument.

Acknowledgment

I sincerely thank my parents, lecturers, and friends, especially Dhanushka and Jayaruwan, who helped me in carrying out this research. Last, but not least, the Android Developer website and the people who have expressed their views for questions in Stack Overflow forum.

References

[1] Thomas, N., Leeman, S., "Mean frequency via zero crossings [medical ultrasound]," *Ultrasonics Symposium, 1991. Proceedings., IEEE 1991* , vol., no., pp.1297,1300 vol.2, 8-11 Dec 1991 doi: 10.1109/ULTSYM.1991.234055

[2] Wendykier, Piotr, JTransforms version 2.3. Retrieved September 1, 2013,
[Online]. Available:
https://sites.google.com/site/piotrwendykier/software/jtransform