CSC 120: Building a Chromatic Tuner

Comprehensive Lab

1 Introduction

In this lab you will be creating a chromatic tuner, which is a device that tells a user what musical note is closest to the one being played into the tuner's microphone. In order to accomplish this on Android, you must utilize its onboard microphone and directly analyze the incoming audio data. Aiding you in this task is a function called the Fast Fourier Transform (FFT).

The FFT is a special case of the more general Fourier transform in that the FFT only applies to discrete data. The FFT has various applications, but the application for which it will be used here is to decompose an audio waveform into its constituent frequencies. A full discussion of the Fourier transform is beyond the scope of this lab and this course, but some examples of it in action will be given. Please note that these examples are meant to illustrate concepts rather than be perfectly accurate in their results. Figure 1 demonstrates the result of a continuous Fourier transform applied upon a sine wave. Figure 2 on the other hand illustrates the expected results of a discrete Fourier

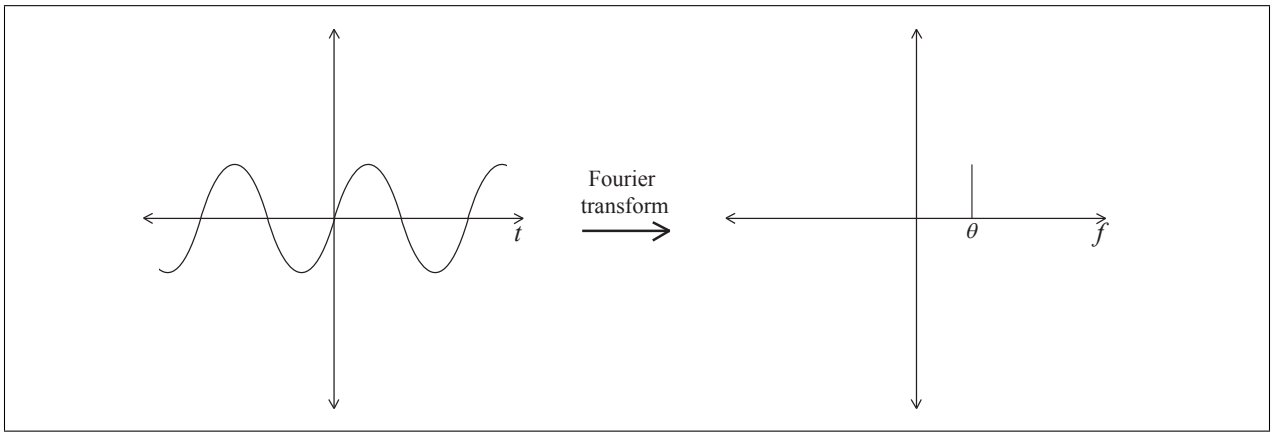

Figure 1: The Fourier transform of a sine wave with frequency θ in the time domain results in a single peak in the frequency domain at θ.

transform applied upon a discretized sine wave. As you can see, the frequency decomposition is not as well-defined in the discrete case. This lack of definition is due to the fact that the information given is incomplete and noisy (error prone). If you were to increase the number of samples taken, the frequency decomposition would expectedly become cleaner. However, taking more samples has a drawback in that it takes longer to collect and process them. The continuous case is basically the result of extending this thought and taking an infinite amount of samples. Since we can't

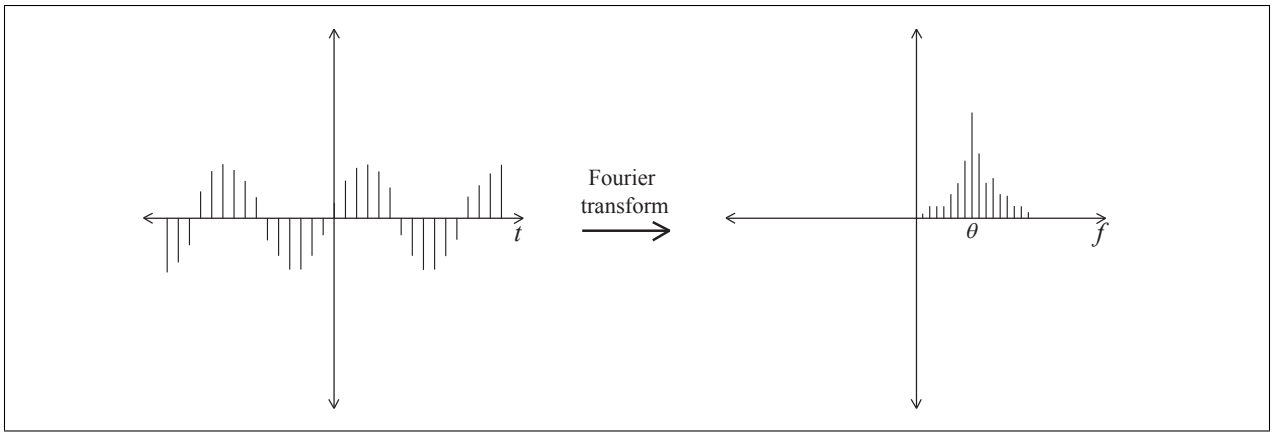

Figure 2: The Fourier transform of discrete samples approximating a sine wave with frequency θ in the time domain results in a rough distribution in the frequency domain peaked at (or near) θ.

take an infinite amount of samples, we will just have to settle with the rough peak in the discrete transform.

In Java, the FFT is implemented as a `static` function of the `FastFourierTransform` class. It takes an array of `Complex` objects as input, and places its output into the given array. This means that once you give it something, the input is lost (unless you want to implement the Inverse FFT).

The `Complex` objects the FFT takes as arguments are simply representations of complex numbers $a + bi$, where a is the real part and b is the imaginary part. Complex numbers are intimately related to waves and the FFT, but the only extent to which you need to understand them is very basic. As long as you can perform the operations of complex addition, subtraction, multiplication, and division, you should be fine. The `FastFourierTransform` and `Complex` classes are provided for you in the package `csc120.lab6.FFT`.

2 Objective

The purpose of this lab is to provide a comprehensive test of the abilities and knowledge that you have gained thus far. As a result, the solution to the problem necessitates the use of classes, methods, loops, arrays, APIs, and debugging tools. In addition, the problem itself will require you to critically think of ways to effectively solve it.

3 Activity

There are several tasks that you must complete in order to have a functional tuner. First, though the `Complex` class has been provided, it is not completely implemented. Before the FFT will work, you must provide the implementation for the following functions: `plus()`, `minus()`, `times()`, and `dividedBy()`. Each function takes a `Complex` parameter and returns a `Complex` argument. As a hint for the division of two complex numbers, consider multiplying the numerator and denominator of the fraction $(a + bi)/(c + di)$ by the conjugate of the denominator, $c - di$.

The second major component is the `SoundAnalyzer` class. The bulk of your work will likely be concentrated here. The `SoundAnalyzer` class fulfills the following objectives: it reads audio from the microphone, it provides input to the FFT, and it analyzes the FFT's output. Audio should be read in a similar manner to the method used in the Alarm lab at a rate of 44.1 kHz (44,100 times per second) into an array of shorts. These shorts are then transferred into the real components of a complex array (whose imaginary components are zero), and provided directly to the FFT. The size of this array must be a power of 2. Fortunately, Android should automatically provide you one of that size. 4096 is a suitable number of array elements, though you may wish to increase this for improved frequency resolution or decrease it for improved responsiveness. `FastFourierTransform.initialize()` should be called once for each time that you wish to change the size of the FFT input array. The `AudioRecord` class is strongly recommended for audio acquisition with 16-bit PCM encoding. Everything in `SoundAnalyzer` should be contained or accessible from within the function `onHandleIntent`.

The output of `FastFourierTransform.fft()` is of course the same size and type as its input since it modifies the input array, though strictly speaking it returns `void`. Even though the FFT produces a frequency decomposition, the output is not quite completely processed. For example, the complex numbers now have real and imaginary components. This is easily remedied. Just take the magnitude, $a^2 + b^2$, of the complex numbers; these magnitudes will correspond to the peaks of the Fourier transform as seen on the right side of Figures 1 and 2.

You then need to figure out which frequency matches with each magnitude, which is fairly straightforward, especially since the magnitudes are already ordered by frequency. Consider the following: you took samples at a rate of 44.1 kHz, and you have a certain number of frequencies (for which you have calculated magnitudes) that must cover that spectrum from 0 to 44.1 kHz. In other words, your frequency distribution is evenly spread out across your sampling rate such that your lowest frequency corresponds to 0 Hz and your highest to 44.1 kHz. The consequence of this is that, as predicted by Figure 2, you do not have well-defined frequencies. Instead, you have bins of $width = (samplerate)/(arraysize)$. It should be noted that since humans cannot hear above about 20,000 Hz, the second half of these bins may be safely ignored.

The next step is to determine the representative frequency. The method you choose determines how effective your tuner is at its intended task. The simplest method is to choose the frequency bin with the largest magnitude, but this solution is inflexible and unresponsive. In addition, it may be inaccurate since musical instruments usually have overtones that may hide the fundamental frequency. Don't be discouraged, however; it is acceptable if your tuner can identify a pure tone with its nearest note. For an example of a pure tone and an excellent test resource, simply search "a440" on YouTube and listen to the first clips. Alternative methods include curve fitting, interpolation, and harmonic analysis. Keep in mind that the method you use is not the point of this lab, though it can be a nice personal touch that makes your tuner stand out from the rest. The only requirement concerning your solution is that it must make use of the FFT and its output.

Once you have identified the fundamental frequency, you must label it with a note. We will assume that we are using the equal temperament chromatic scale. In this scale, there are 12 notes in an octave, with an octave merely an ordered collection of notes before they start over. That is, each octave contains the same 12 notes. The notes are as follows: A, A#, B, C, C#, D, D#, E, F, F#, G, G#, with # pronounced "sharp." The standard American scale is based upon the note A 440,

which is the A note, 4th octave, at 440 Hz. Each octave is a factor of 2 larger or smaller than the next or previous. That is to say, A3 is 220 Hz and A5 is 880 Hz. In addition, each note is evenly spread out in an octave in equal temperament, such that every note is a constant factor away from its neighbors. Since we are on a 12 note scale, this constant factor is the twelfth root of two, $\sqrt[12]{2}$. Note that since each successive octave is twice the size of the previous and the gap between notes gets larger and larger, simply finding the closest note to your estimated fundamental may not be sufficient in all cases. You may consider using a log scale to address this issue. It is required that you implement the musical scale in its own class.

Finally, at this stage you should have a note in mind that you think the user is trying to play. A simple function is provided in the class `TunerActivity`, `updateGUI()`, that takes four parameters to update the display for the user. The parameters in order are as follows: the note with octave (e.g. C#4), the estimated fundamental frequency, how far below the note the frequency is, and how far above the note the frequency is. Clearly, if the frequency is either above or below the note, then the third or fourth parameter should be zero, respectively. The call to `updateGUI()` and any final calculations should be contained within the `tune()` function. There is also an optional `tare()` method already bound to the "Tare" button in the `TunerActivity` layout that is meant to house any interactive noise filtering or thresholding that you may wish to perform.

As always, it is recommended that you do not alter functions that have not been explicitly referenced here as that may make the problem immensely more difficult.

4 Conclusion

This activity should test all that you have learned so far in this class. You have learned much about programming in Java: loops, arrays, control statements, API calls and other concepts. After completing this lab, you should understand sound a bit better in terms of pitch and have an even greater understanding of how different programming paradigms interact.

5 Deliverables

The deliverables include the Eclipse project export and the necessary modifications to the `SoundAnalyzer`, `TunerActivity`, and `Complex` classes. Along with these changes, you must also create your own class that implements a musical scale and provides labels and frequencies for all possible notes. Your program, at the minimum, should be capable of identifying the correct note when confronted with a pure tone.