Application of DSP for Note-by-Note Analysis of .WAV Files

Jack Baskin School of Engineering

Report by Jacob Jonathan Giefer Lee
Professor Farid Dowla
EE153 - Digital Signal Processing

I. Background and Motivation

For years I have become increasing interested in the evolution of music and the advancing bond between digital technologies and physical instruments, but as an undergraduate engineering student I found myself with very little time to learn about the software and hardware involved in their improving marriage. Luckily, this DSP final project has served as a great platform for me to dive deep into topics of digital signal processing (understandably). Initially I was interested in building a tunable synthesizer pedal for my bass guitar; however, since I had no experience at all in either signal processing or coding in Python before this course, I ultimately decided to start small and write a script that could accurately decode the notes of a track for a single instrument. The process of writing this script taught me quite about the practical applications of the material I learned in this course, including DFTs and windowing. In the future I would like to use this new knowledge to build the pedal I initially planned to build.

II. Introduction

A file with the '.wav' extension is called a WAV file and contains minimally compressed audio data. The data is represented in either 1 (mono), 2 (basic stereo), or more channels for more complex .wav encodings. Each is made up of hexadecimal packets with lengths corresponding to twice the number of channels through which they are played. As is standard in audio files, these are generated by sampling an input signal at 44,100 samples per second. Since a WAV file contains a signal that is basically uncompressed, several Python packages have been developed to easily interface with and manipulate these files.

The process of analyzing the powers at different frequencies for an audio signal is most easily done using a discrete Fourier transform (DFT) of smaller constituent signal sections. The DFT of a signal with sample length N is given by equation 1:

$$X[k] = \sum_{n=0}^{N} x[n] e^{-j(2\pi/N)kn} \quad [1];$$
$$k = 0, 1, \ldots, N-1$$

This equation transforms N samples from the original time domain signal representation to N coefficients in the frequency domain. The most common iteration of this algorithm in computing is the fast Fourier transform, or FFT. This algorithm is simply a method for calculating the DFT, and it has a characteristic equation that is exactly equivalent to that of the DFT. The only difference between the two functions is the FFT is more efficient

and requires fewer total operations than the traditional method of computing the DFT, but that is a topic for another paper.

The frequency content of a signal cannot be determined by only one sample value. As the amount of samples used in a DFT is increased, the function more accurately plots the spectrum of the time domain signal, but its precision comes at the cost of losing any sort of accurate reference to time. Therefore, in order for the FFT to be useful to us in mapping the spectrum of the signal with respect to time, we must choose an appropriate window over which to sample. A window is a function with a finite width that represents a single discrete “view” of part of a larger signal. We can implement a window by multiplying its function by a section of the larger signal, shifting the window’s sample index, and then repeating the operation until we have reached the end of the large, discrete signal. In this way, we can use a window to ‘sample’ an input signal, but our “samples” are now arrays of samples that can be spectrally analyzed. Since adjacent windows can overlap, using the FFT over windows can actually provide accurate time-referenced spectral analysis.

There are several common windows that can be applied to a signal. A rectangular window is by far the most obvious window, but with a quick Fourier analysis of a discrete rectangular function produces a Dirichlet form with sidelobes that can cause unwanted signal distortion. A “good” window for the our purposes should have a frequency response on the initial signal as close to the so-called delta function as possible so that the output spectrum will be very close to that of the input. Thus, our search for a better window produces the von Hann window, which was briefly discussed in lecture and in the text, “DSP First”, starting on page 318. It is mathematically given for time index n and with length L by function 2:

$$w_h[n] = \begin{cases} 0.5 - 0.5\cos(2\pi(n+1)/(L+1)), & n=[0, L-1] \\ 0, & else \end{cases} \quad [2]$$

[McClellan, Schafer, and Yoder; 318]. A von Hann window increases in magnitude in a roughly Gaussian shape until (n+1) reaches (L+1)/2, and then decreases symmetrically about its axis past this. The shape of the von Hann window in the frequency domain really shows its usefulness. Figure 1 depicts a simulated Hann window function with its associated spectrum. Notice that the spectrum has very low powered sidelobes relative to one central peak, as opposed to the large sidelobes seen in a the Dirichlet form of a rectangular window. So if we use this window function to sample a signal that is built of a sum of sinusoids (like an audio signal), each component signal’s spectrum will stay essentially undistorted by our window. However, the von Hann window does have a wider central peak than a rectangular window, so some distortion of It is also worth

noting that logically the precision of the von Hann window increases proportionally to its window frame size, or the number of samples that it uses. Again, as this spectral precision goes up, the precision of its time index decreases, so we must find a "Goldilocks" balance: just precise enough in both spectral and time domain representations.

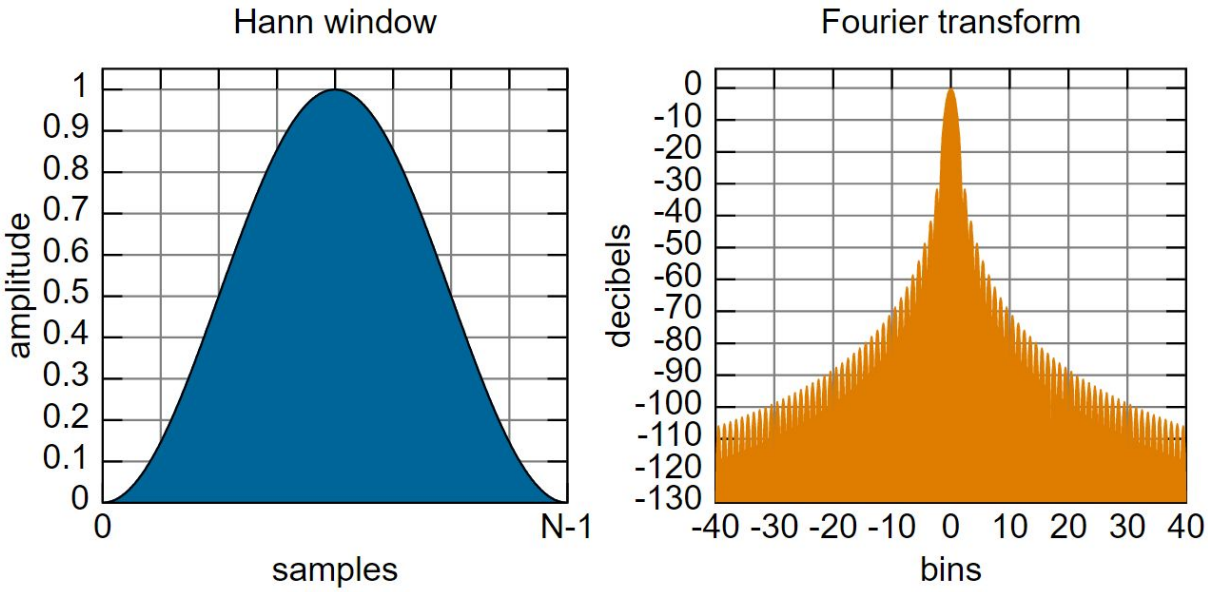

Figure 1: simulated Hann window in time domain (left) and its frequency response (right)
[via https://en.wikipedia.org/wiki/Hann_function]

III. Procedure and Methods

The task of converting an audio sample into a set of notes with respect to time can be broken down into manageable sections, namely:

1. Import and read audio sample
2. Parse the samples into window frames for individual analysis
3. Run a DFT on each window
4. Decode the each frame into its individual notes
5. Represent each note played with respect to time

1. *Import and read audio sample*

Python makes reading .WAV files easy and provides two essential packages, 'Wave' and 'Soundfile'[1], that together allow for placement of sample data (with respect to sample number) into an array. Soundfile adds functionality to more exotic WAV file formats, and it can very effectively deal with 3 or more channels of audio input.

[1] View 'Works Cited' on page 14 for downloaded package details and other credits.

2. *Parse the samples into window frames for individual analysis*

The "Goldilocks" window length mentioned previously was picked with some basic reasoning as follows:

- The lowest note I chose to represent was 'C1' at 30Hz, which has a period of 33.3 milliseconds
- The sampling rate of an audio file is 44,100Hz; therefore the sample time is about 2.27E-5 seconds
- The number of samples required to fully sample this signal is then [ceiling(33.3/2.27E-2)] = 1469 samples at least
- Nyquist's theorem states that accurate reconstruction requires at least twice this number of samples = 2938 samples or more per window

Our textbook uses 2^{10} (1024) samples to create a smooth plot in MATLAB on page 319, so I chose to stick with powers of two but two-up the text by using 2^{12} samples per window, which also satisfies Nyquist as mentioned above.

Then I chose a window overlap ratio of 50% since it sounded like a logical starting point, and it stuck. This was also the ratio I noted was used in a useful sample code written by Frank Zaklow that produces a very cool spectrogram of audio files. Since I was new to Python, I structured a large part of my windowing scheme in Python off of Zaklow's code as credited in my 'Works Cited' on page 14. Specifically, I learned most about the 'stride_tricks' functions from the more tricky corners of the Numpy library. Essentially, this set of functions greatly facilitated the creation of individual arrays for each window sampled by treating each copied array as a single element of a larger array and using 'as_strided' to generate this larger array with a fixed size. From here it is very simple to multiply each smaller array by a matching von Hann window function. For better code adaptability, I also normalized each window in magnitude by half of (1 + window length), which corresponds to the scale factor of a von Hann window.

3. *Run a DFT on each window*

Now that we are left with an array of normalized windows, we can simply run the Numpy FFT algorithm, 'numpy.fft.rfft'[2] on our larger array, and a normalized frequency domain array of equal size is output, with each window accurately time-referenced.

4. *Decode the each frame into its individual notes*

Now that we have accurate spectra for each time-indexed window, we need to create a lookup table (LUT) that decodes frequency into a note. This essentially

[2] 'numpy.fft.rfft' gives the single-ended fast Fourier transform and is used for real signals like audio

translates to creating a bulky ‘switch’ statement that I built using an amalgamation of ‘if’ and ‘elif’ statements.

No frequency component represented in each window had truly zero power, I used the frequency component with the largest magnitude in each window to represent each note played. If the magnitude of the largest component in a window had less than an experimentally determined 0.1...units, then the window was considered a ‘REST’ where no notes were played.

5. *Represent each note played with respect to time*

Now that all the data is collected, we have an opportunity to take some artistic liberties and represent the data any way we desire. I personally chose to use formatted print statements to represent each value in a simple but elegant manner, as will be shown shortly.

IV. Results

Below is a copy of my script, titled *notereader.py*:

```
#!/usr/bin/env python
"""A SCRIPT THAT MAKES YOUR DREAMS COME TRUE."""
import numpy as np
import wave
import soundfile as sf
from numpy.lib import stride_tricks


# Function declarations
def noteTrans(freqIn):
    """RETURNS ALL THE NOTES."""
    # Note LUT
    noteOut = ''
    if freqIn > 0 and freqIn <= 33:
            noteOut = 'C1'
    elif freqIn > 33 and freqIn <= 35:
            noteOut = 'C#1'
    elif freqIn > 35 and freqIn <= 37:
            noteOut = 'D1'
    elif freqIn > 37 and freqIn <= 39.5:
```

```
        noteOut = 'D#1'
elif freqIn > 39.5 and freqIn <= 42:
        noteOut = 'E1'
elif freqIn > 42 and freqIn <= 45:
        noteOut = 'F1'
elif freqIn > 45 and freqIn <= 47:
        noteOut = 'F#1'
elif freqIn > 47 and freqIn <= 50:
        noteOut = 'G1'
elif freqIn > 50 and freqIn <= 53:
        noteOut = 'G#1'
elif freqIn > 53 and freqIn <= 56.5:
        noteOut = 'A1'
elif freqIn > 56.5 and freqIn <= 59.5:
        noteOut = 'A#1'
elif freqIn > 59.5 and freqIn <= 63:
        noteOut = 'B1'
elif freqIn > 63 and freqIn <= 67:
        noteOut = 'C2'
elif freqIn > 67 and freqIn <= 71:
        noteOut = 'C#2'
elif freqIn > 71 and freqIn <= 75:
        noteOut = 'D2'
elif freqIn > 75 and freqIn <= 79:
        noteOut = 'D#2'
elif freqIn > 79 and freqIn <= 85:
        noteOut = 'E2'
elif freqIn > 85 and freqIn <= 89:
        noteOut = 'F2'
elif freqIn > 89 and freqIn <= 95:
        noteOut = 'F#2'
elif freqIn > 95 and freqIn <= 100:
        noteOut = 'G2'
elif freqIn > 100 and freqIn <= 105:
        noteOut = 'G#2'
elif freqIn > 105 and freqIn <= 112:
        noteOut = 'A2'
elif freqIn > 112 and freqIn <= 120:
        noteOut = 'A#2'
```

```
    elif freqIn > 120 and freqIn <= 126:
        noteOut = 'B2'
    elif freqIn > 126 and freqIn <= 133:
        noteOut = 'C3'
    elif freqIn > 133 and freqIn <= 140:
        noteOut = 'C#3'
    elif freqIn > 140 and freqIn <= 150:
        noteOut = 'D3'
    elif freqIn > 150 and freqIn <= 160:
        noteOut = 'D#3'
    elif freqIn > 160 and freqIn <= 170:
        noteOut = 'E3'
    elif freqIn > 170 and freqIn <= 180:
        noteOut = 'F3'
    elif freqIn > 180 and freqIn <= 190:
        noteOut = 'F#3'
    elif freqIn > 190 and freqIn <= 201:
        noteOut = 'G3'
    elif freqIn > 201 and freqIn <= 212:
        noteOut = 'G#3'
    elif freqIn > 212 and freqIn <= 225:
        noteOut = 'A3'
    elif freqIn > 225 and freqIn <= 240:
        noteOut = 'A#3'
    elif freqIn > 240 and freqIn <= 255:
        noteOut = 'B3'
    else:
        noteOut = 'wat'
    return noteOut


def stft(signal, frameSize, window=np.hanning, olf=0.5):
    """GIVES SHORT TIME DFT OF SIGNAL."""
    windo = window(frameSize)
    # windows overlap by 'olf' of window size
    winShift = frameSize - np.floor(olf * frameSize)
    sigSamp = np.append(np.zeros(int(frameSize/2)), signal)
    # total number of window frames
    frameNum = int(len(sigSamp) / winShift)
```

```
    sigSamp = np.append(sigSamp, np.zeros((1 - olf) * frameSize))
    scaleFac = (frameSize + 1) / 2
    sigOut = []
    # now iterate through sample with window framing
    sigOut = stride_tricks.as_strided(sigSamp, shape=(frameNum, frameSize),
                                      strides=(sigSamp.strides[0]*winShift,
                                      sigSamp.strides[0])).copy()
    sigOut *= windo
    sigOut /= scaleFac

    return np.fft.rfft(sigOut)


# Open up le file...
track = 'Bass/bass080A/bassA001.wav'
wavread = wave.open(track, "r")
sigIn, fsamp = sf.read(track)
wavread.close

# GOAL: use DFT of sample over intervals to estimate note
winLength = 2**12
thisFT = stft(sigIn, winLength)
timeBins, freqBins = np.shape(thisFT)
freqs = np.abs(np.fft.fftfreq(freqBins*2, 1/fsamp)[:freqBins])
sampCount = 0
# for each Hann, find greatest freq component, see if it's a rest/note, decode
tOut = []
while sampCount < timeBins:
    tOut = np.append(tOut,
                np.round(((sampCount + 1) * (winLength / (2 * fsamp))),
                        decimals=3))
    sampCount += 1

goingOut = []
for hSamp in thisFT:
    freqOutIndex = np.argmax(hSamp)
    leMax = abs(hSamp[freqOutIndex])
    if abs(hSamp[freqOutIndex]) > 0.1:
        goingOut = np.append(goingOut, noteTrans(freqs[freqOutIndex]))
```

```
    else:
        goingOut = np.append(goingOut, "REST")

print('    {0:5}  | {1:9}\n\t   |'.format('TIME:', 'NOTE:'))
sampCount = 0
while sampCount < len(tOut):
    print(' {0:8}s | {1:8}'.format(tOut[sampCount], goingOut[sampCount]))
    sampCount += 1
```

This script produces the following output for the given input WAV file:

```
    TIME:   | NOTE:
            |
    0.046s | REST
    0.093s | REST
    0.139s | REST
    0.186s | REST

   0.232s | REST
    0.279s | REST
    0.325s | REST
    0.372s | REST
    0.418s | REST
    0.464s | REST
    0.511s | REST
    0.557s | REST
    0.604s | REST
     0.65s | REST
    0.697s | E3
    0.743s | A1
    0.789s | A#2
    0.836s | REST
    0.882s | REST
    0.929s | A2
    0.975s | A2
    1.022s | A2
```

```
   1.068s | A1
   1.115s | REST
   1.161s | A1
   1.207s | REST
   1.254s | A1
     1.3s | REST
   1.347s | REST
   1.393s | REST
    1.44s | REST
   1.486s | REST
   1.533s | REST
   1.579s | REST
   1.625s | REST
   1.672s | REST
   1.718s | REST
   1.765s | REST
   1.811s | REST
   1.858s | REST
   1.904s | REST
    1.95s | REST
   1.997s | REST
   2.043s | REST
    2.09s | REST
   2.136s | REST
   2.183s | REST
   2.229s | REST
   2.276s | REST
   2.322s | REST
   2.368s | REST
   2.415s | REST
   2.461s | REST
   2.508s | A2
   2.554s | A2
   2.601s | A2
   2.647s | REST
   2.694s | REST
    2.74s | REST
```

```
    2.786s | REST
    2.833s | REST
    2.879s | A2
    2.926s | A2
    2.972s | REST
    3.019s | REST
    3.065s | REST
    3.111s | REST
    3.158s | REST
    3.204s | REST
    3.251s | REST
    3.297s | REST
    3.344s | REST
     3.39s | REST
    3.437s | REST
    3.483s | REST
    3.529s | REST
    3.576s | REST
    3.622s | REST
    3.669s | E3
    3.715s | REST
    3.762s | REST
    3.808s | A2
    3.855s | A2
    3.901s | A2
    3.947s | REST
    3.994s | REST
     4.04s | REST
    4.087s | REST
    4.133s | REST
     4.18s | REST
    4.226s | REST
    4.272s | REST
    4.319s | REST
    4.365s | REST
    4.412s | REST
    4.458s | REST
```

```
   4.505s | REST
   4.551s | REST
   4.598s | REST
   4.644s | REST
    4.69s | REST
   4.737s | REST
   4.783s | REST
    4.83s | REST
   4.876s | REST
   4.923s | REST
   4.969s | REST
   5.016s | REST
   5.062s | REST
   5.108s | REST
   5.155s | REST
   5.201s | REST
   5.248s | REST
   5.294s | REST
   5.341s | REST
   5.387s | REST
   5.433s | REST
    5.48s | REST
   5.526s | E3
   5.573s | A1
   5.619s | A#2
   5.666s | E3
   5.712s | A#2
   5.759s | REST
   5.805s | A2
   5.851s | A2
   5.898s | A1
   5.944s | REST
   5.991s | A1
   6.037s | REST

[Finished in 0.631s]
```

The input WAV file was actually a purely bass guitar track that repeated only the note A. The presence of other notes is due to note intonation, cutoff, approach, and harmonics. The sample heavily uses so-called ‘ghost notes’ as well, which I believe contribute to the odd sharp and off-pitch notes that were mapped. Overall, this script works quite well, but it will work best at decoding audio notes for input notes with very clean intonation.

V. Conclusion

This project was successful in teaching me the fundamentals of Python and some of the practical applications of digital signal processing. It was also a success in the fact that my script worked roughly as predicted, but it would be interesting to find better ways to map the spectra to specific instruments, including drums and vocal tracks, or even analyze more complex audio tracks like full songs with a variety of instruments. I hope that soon I will be able to apply the skills I learned in this project towards creating a bass synthesizer pedal, and I will continue building on this knowledge base. I really enjoyed this course and completing this final project, and I hope that you as a grader enjoyed reading this report.

VI. Works Cited

1. “DSP First”, 2nd ed. McClellan J.H., Schafer R.W., Yoder M.A..
2. “Create audio spectrograms with Python”, Zaklow, Frank. http://www.frank-zalkow.de/en/code-snippets/create-audio-spectrograms-with-python.html?ckattempt=1
3. “2.2: Advanced Numpy”, Virtanen, Pauli. http://www.scipy-lectures.org/advanced/advanced_numpy/#indexing-scheme-strides

Packages downloaded used ‘pip install’:

- Numpy from http://www.numpy.org/
- Soundfile from https://github.com/bastibe/PySoundFile