Converting Electronic Music to Sheet Music
TJHSST Senior Research Project Paper
Computer Systems Lab 2009-2010

Hugh Smith

June 15, 2010

Abstract

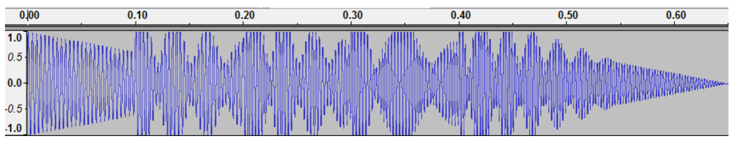
Electronic creation of music has become a wide-spread hobby or profession. Electronically generated music is when a user inputs data about a sound into a computer, and it produces said sound. Even though this is very widespread, the opposite is not true. That is, the conversion of raw sound data into sheet music, or something that defines the sound, is much less prevalent. Why is this? It is, of course, very challenging to do. This project aims to do just that - convert a piece of digital music (a .WAV file) into actual sheet music, using the Fourier transform algorithm. **Keywords:** music, analysis, fourier, transform, wave, digital, algorithm, wav, sheet, music

1 Introduction

This project will involve reading the audio data from an electronic audio file in .WAV format and converting it into another file format called ABC. The ABC format will be discussed in more detail later in the paper. This process will be difficult because converting music files to basic sound data can be very complicated. For this, one must use the Fourier transform algorithm, which converts a function with a time domain into one with a frequency domain. This algorithm has many applications, and it is very useful for this type of musical analysis.

2 .WAV files

The .WAV file is one way of storing music digitally. The following figure is an example of such a file, where time is moving from left to right and the height of the curve is the amplitude of the music signal.



WAVE files have a specific description for how they are constructed, like all file formats. First come the three header "chunks" (so called because they are collections of data) and then the actual sound data itself. First comes the RIFF Descriptor Chunk:

Offset	Size	Description	Value
0	4	Chunk ID	RIFF
4	4	Chunk data size	Length of File
8	4	RIFF type	WAVE

This is the chunk that describes to the file reader what exactly the file is. The first field always has a value of "RIFF." RIFF is a proprietary file type developed by Microsoft. It is used for many formats, including WAVE and MPEG files. The "RIFF" tells the reader that this is a music or video file, and that it will have the same format as any RIFF-type file. The next item in the table, the "chunk data size," is simply the total length of the file, in bits. The last item in the RIFF chunk, the RIFF type, tells the reader what specific type of file it is - if it says "WAVE," then it is a sound file, if it says "MPEG," then it is a video file. The next chunk is the format, or "FMT" chunk.

Offset	Size	Description	Value
12	4	Chunk ID	"fmt"
16	4	Chunk Data Size	16 + *
20	2	Compression code	Int
22	2	Number of channels	Int
24	4	Sample rate	Hex
28	2	Block align	Hex
32	2	Significant bits per sample	Int
34	2	Extra format bytes	Int

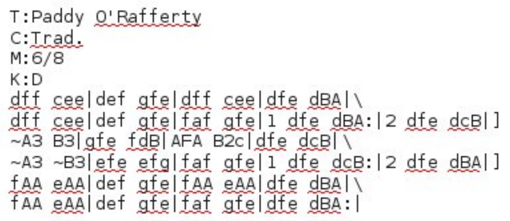
This chunk describes various constants for the music file. Besides the sound data itself, it's the most complex part of the WAVE file, just because of all the sound jargon it contains. The most important part of this chunk describes sample characteristics. Samples are how the WAVE file stores the sound data. Each sample in the file is a point on the sinusoidal sound wave, describing the amplitude of the wave at that time. The "chunk ID" will always show "FMT" to indicate that the file reader has reached the format chunk. The sample rate is how many samples are taken per second in the file. A higher number means a higher quality sound file, but also a larger file. Significant bits per sample identifies the storage size of each sample - for example, eight bits or sixteen bits. Again, a higher number means a higher-precision sample, which once again means higher quality. The final chunk in a WAVE file is the Data chunk.

Offset	Length	Description	Value
36	4	Chunk ID	"data"
40	4	Chunk size	Depends on file
44	--	--	--

The data chunk is the most important part because it contains the data itself. The "chunk ID" always says "data," just like the format chunk. The chunk size is the length in bits of the actual data, which can vary based on the length of the song and the quality, as described above. After that, the file goes right into the data. The data needs to be read in a binary mode in sets of the significant bits per sample specified in the format chunk.

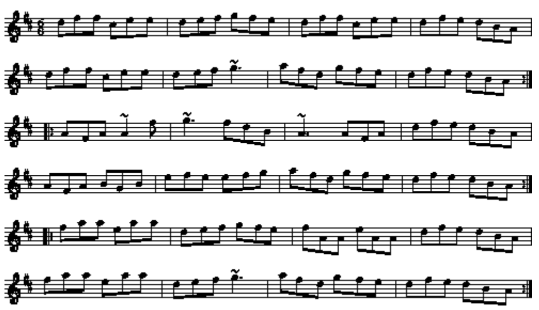
3 ABC Format

ABC notation is a way to represent sheet music in a text file. The format is simple, with the letters of the keys corresponding to their sheet-music symbols. You can add information such as the title, the arranger, the performer, and other things to the music. In addition, there are lots of programs available for converting this notation to a PDF document, so it can be actual sheet music. ABC notation is best for rendering single-melody songs, so it will be very useful for my project. The general format of an ABC file is as the following example:



```
T:Paddy O'Rafferty
C:Trad.
M:6/8
K:D
dff cee|def gfe|dff cee|dfe dBA|\
dff cee|def gfe|faf gfe|1 dfe dBA:|2 dfe dcB|]
~A3 B3|gfe fdB|AFA B2c|dfe dcB|\
~A3 ~B3|efe efg|faf gfe|1 dfe dcB:|2 dfe dBA|]
fAA eAA|def gfe|fAA eAA|dfe dBA|\
fAA eAA|def gfe|faf gfe|dfe dBA:|
```

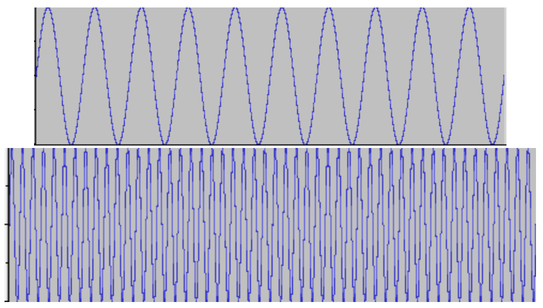
This creates a sheet music file like so:



ABC format is a well-used format, with many websites offering versions of their MIDI or MP3 files in ABC format as well. An example is The Session (http://thesession.org/). This website is a collection of Irish traditional folk music. This is useful, because, most folk music has one melody. For my project, these types of songs will be best to test my program on, as there is a low probability of getting the song wrong if the program is actually working.

4 Sound Waves

Sound waves are defined by three things: frequency, amplitude, and phase. Luckily, this project only requires a close look at the first one, frequency. The frequency of a wave is how many "cycles" it has per second (Hertz). In a sound wave, each different-toned note has a different frequency - it is what distinguishes high-pitch sounds from low-pitch sounds. This is easy to tell graphically.



The first wave has a lower frequency than the second. This means, in sound terms, the second wave has a higher pitch than the first. Also, both waves have the same amplitude. This means that, no matter the pitch, they will always sound as loud as one another. Looking at the graphs, the amplitude is found by seeing how far the waves go above or below the midpoint. The graphical example shows that the computer is able to show this information in an easy-to-understand notation. However, it can also get quite complex. Looking back at the example used in the .WAV file type section, it is hard to tell one frequency from another. The file itself actually describes the progression from middle C, to D, E, F, and G. It's part of a C major scale. Such a simple thing, when being played, sounds like it would be simple as well as a sound wave, but as one can see, it is not.

5 Fourier Transform

To go from these sound waves into actual notes, a piece of rather complicated math is required - the Fourier transform. The Fourier transform, in sound, takes a function in the time domain, and changes it into a function in the frequency domain.

$$\hat{f}(\xi) = \int_{-\infty}^{\infty} f(x)\, e^{-2\pi i x \xi}\, dx,$$

f(x) is the function describing the amplitude of the music in the wave file at time=x. Then, the Fourier transform defines a function of frequency, where frequency is denoted by the Greek letter xi. Since the .WAV file is made up of discrete samples, we use the Discrete Fourier Transform:

$$\hat{f}(\xi) = \sum f(x)\cos(2\pi\xi x) + i\sum f(x)sin(2\pi\xi x)$$

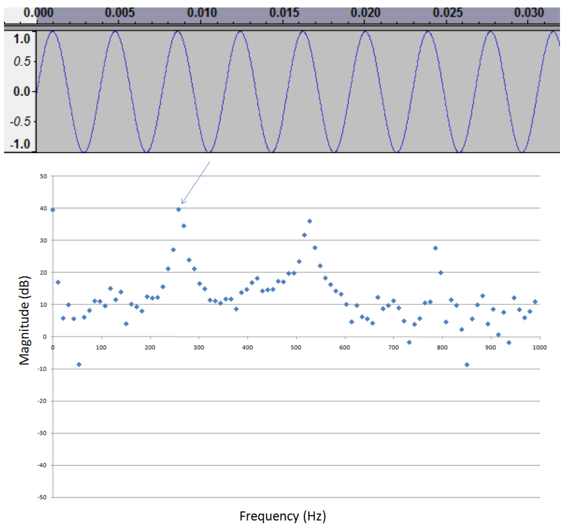
This can be simplified by substituting the two summations as R(freq) and I(freq), respectively.

$$\hat{f}(\xi) = R(\xi) + iI(\xi)$$

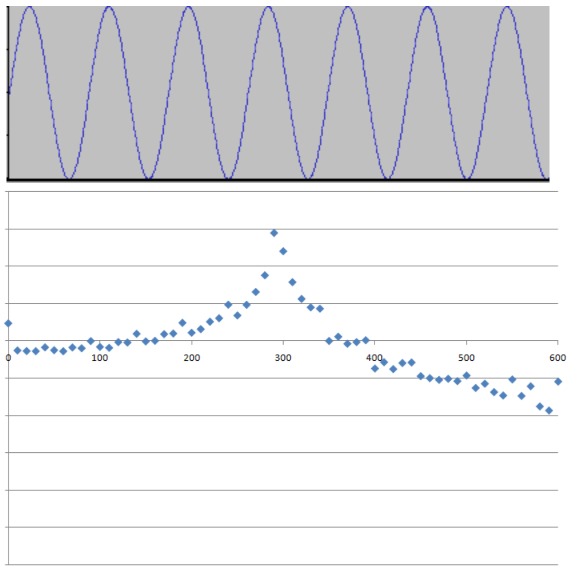
Then, the "magnitude" of the function would be described as applying the distance formula to the R and I:

$$M(\xi) = \sqrt{R(\xi)^2 + I(\xi)^2}$$

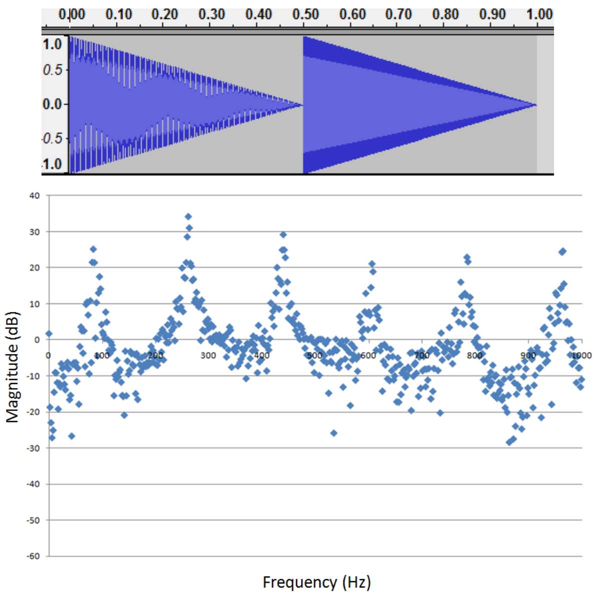
So, each frequency has a certain magnitude. These two are shown together in the next graphs:



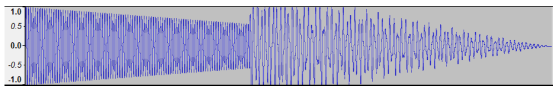
Looking at the second graph, there is a sequence of peaks at various frequencies. These represent the base frequency and harmonics of the note being played. In this case, it is middle C, so there are peaks at all the Cs shown in the graph. However, the important part is the highest peak. That indicates the "base frequency," or the original note that was played. The frequency of the highest peak is about 261 Hertz - which is the frequency of middle C. Each one of the dots on the graph represents a sample - although, this is a smaller graph window, and so the actual number of samples is much larger than shown. Another example is middle D -



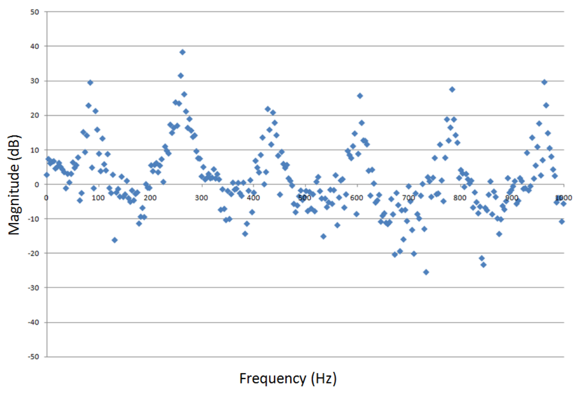
The frequency for middle D is about 290 Hertz, so this one checks out as well. However, what if the input file is a portion of middle A playing, and then a portion of middle C playing?



Here the first problem presents itself. We have two sets of peaks here, but how is the program supposed to know that there are two notes instead of just one? One solution is to check the raw sound data for a sudden increase in the loudness of the sound, and that would work in this case. However, not many songs have notes that are completely staccato. For something like the case of the C scale presented earlier, there is no clear indicator of where one note stops and another begins. Dealing with this case, though, it is relatively easy to check when this happens. The first part of the solution would involve finding the peaks of the data. First, take the absolute value of the raw data, and while the slope is positive, check whether the slope suddenly becomes negative. If it does, that is a peak, and save that time value. Keep on doing this until the end of the file. For the second part, you would go over the peaks, and if the value of the peak suddenly is higher than the previous one, that is a new note. Divide the array at that place. This also solves the problem of the chronological order of the notes. To show this, here is an example of a two-note file where the two notes overlap somewhat:



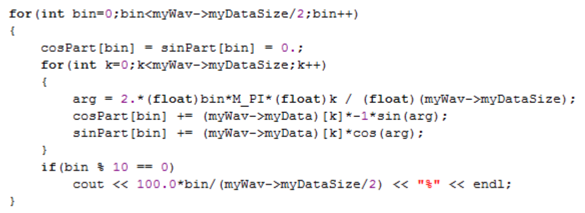
These two notes seem a bit harder to differentiate from one another. Obviously, taking the Fourier transform of the whole file together would give unreadable results. However, what if we split the file into two? The first part alone would obviously give us the correct answer, as it is just that note. However, the second part is a little bit different. It has part of the first note's fading in that part.



In fact, the highest point is the correct answer. This is because the second note is stronger than the first in the last part taken alone. Despite the difficulties in recognizing where the break actually occurs, this seems like a good method for actually rendering sheet music.

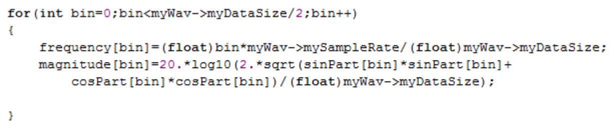
6 Implementation

The coding implementation of this was somewhat different from the theory. As a .WAV file is presented in samples, there was no defined function I could integrate to find the frequency domain. As such, I had to work with a long list of points. For about a 0.1-second file, there were about 30,000 samples. This creates a problem with time, that will be addressed later on in the paper. There were two parts to my implementation of the Fourier transform.



```
for(int bin=0;bin<myWav->myDataSize/2;bin++)
{
    cosPart[bin] = sinPart[bin] = 0.;
    for(int k=0;k<myWav->myDataSize;k++)
    {
        arg = 2.*(float)bin*M_PI*(float)k / (float)(myWav->myDataSize);
        cosPart[bin] += (myWav->myData)[k]*-1*sin(arg);
        sinPart[bin] += (myWav->myData)[k]*cos(arg);
    }
    if(bin % 10 == 0)
        cout << 100.0*bin/(myWav->myDataSize/2) << "%" << endl;
}
```

These first calculations convert the original list of points in the time dimension into two parts - real and imaginary - in the frequency dimension. This is based on the definition of the Fourier Transform as described earlier. Once that calculation is complete, the magnitude of the frequency function at each frequency is calculated.



```
for(int bin=0;bin<myWav->myDataSize/2;bin++)
{
    frequency[bin]=(float)bin*myWav->mySampleRate/(float)myWav->myDataSize;
    magnitude[bin]=20.*log10(2.*sqrt(sinPart[bin]*sinPart[bin]+
        cosPart[bin]*cosPart[bin])/(float)myWav->myDataSize);

}
```

These calculations test each frequency value in the magnitude equation, also specified in the Fourier Transform section. Using the frequency as the X axis, and the magnitude as the Y axis, they make a graph as seen before in the case of middle C. The base frequency is then calculated by finding the frequency with the maximum magnitude:

```
for(int x=1;x<myWav->myDataSize/2;x++)
{
    if(magnitude[x] > maxVal)
    {
        maxVal=magnitude[x];
        maxInd=x;
    }
}
```

This code, by using a simple max search algorithm, finds the highest amplitude value and saves its index in the array. Next, another simple algorithm is used to determine which note the frequency is closest to. This can give a wrong answer sometimes, because due to the sampling error based on the bits per sample and sample rate, the samples can express a lower-quality curve.

```
int note;
for(int x=0;x<100;x++)
{
    if(frequency[maxInd] <= notes[x])
    {
        if(fabs(notes[x]-frequency[maxInd]) < fabs(notes[x+1]-frequency[maxInd]))
        {
            note = x;
            break;
        }
        else
        {
            note = x+1;
            break;
        }
    }
}
```

The notes array used in this algorithm is a list of 100 frequencies of 100 notes. This is actually more notes than are on a piano. The program subtracts each of these from the frequency of the max amplitude, and finds the one it is closest to. Then, taking the frequency of the note, the program substitutes the appropriate letter designation:

```
        break;
    case 38:
        finalnote = "D3";
        break;
    case 39:
        finalnote = "D3#";
        break;
    case 40:
        finalnote = "E3";
        break;
    case 41:
        finalnote = "F3";
        break;
    case 42:
        finalnote = "F3#";
        break;
    case 43:
        finalnote = "G3";
```

Each note was hard-coded in, as was the array of the different frequencies. The frequency differences between the notes become greater as the pitch of the note becomes higher, and there seems to be no set factor to multiply by. As such, it was necesary to hard-code, and it is easier to identify higher pitch notes than lower pitch notes.

7 Time Issues

In the program being used to generate test frequencies, Audacity, the default setting for saving .WAV files is to use a 16-bit sample size. This caused problems in this project, because there is no good way to read in 16-bits into an array of any useful type. 8 bits is used for a character data type value, so that is what was used. Even if there were a good container of that size, however, that is twice as much data being read. If this were not enough, the Fourier transform involves a twice-nested for loop, which takes the running time to at least O(N^2) (an amount of time proportional to the number of elements being processed squared). After that, there is another for loop, which elevates the run time even higher. With so much data, this takes a long time even with a one-second file. The time required to fully run my program for a normal-length file does not bear thinking about, to say nothing of the problems resulting with hitting the max integer value limit. However, these problems were solved by creating samples of extremely fast-playing melodies and notes, and setting Audacity to render the .WAV with eight bits per sample instead of sixteen.

8 Conclusion

This project successfully implemented the Fourier Transform to identify single notes. This procedure also works well for pure staccato music, in which notes do not overlap in time. For multiple notes, the Fourier Transform process will identify the various notes, but does not indicate which order the notes come in. To determine that, the .WAV file data must be used to find out when in time a new note begins. The Fourier Transform process would then be applied sequentially, each time using that portion of the .WAV file between where the current note starts and the next note starts. These start times are easy to determine for staccato music. But when notes overlap, it becomes more difficult. A possible procedure to do this was identified. A lot more could be done to extend this project. In the current version, it uses the Discrete Fourier Transform, which certainly gets the job done. However, it would be faster (and probably easier in the long run) to instead implement the Fast Fourier Transform, another algorithm that uses less samples and finds an answer more quickly. Another thing that needs to be implemented was the actual conversion to ABC format. The current program simply prints out the note it has decided on, and also three text files detailing the frequency, magnitude, and phase of each sample. Future implementations of this project could also figure out the relative lengths of each note, and use those to find the time signature and which length of note each is - quarter note, half note, sixteenth note, etc.

9 Bibliography

http://www.sonicspot.com/guide/wavefiles.html
https://ccrma.stanford.edu/courses/422/projects/WaveFor
http://thesession.org/ Elements of Computer Music (F. Richard Moore) http://www.dspdimension.com/admin/dft-a-pied/