Automatic Transcription of Digital Audio Music

Justin Francos

April 5, 2003

The goal of this project is to have a computer unassistedly convert a digital audio sound file (e.g., ripped from a compact disk) to a similar-sounding MIDI file.

Digital audio files store sound as a sequence of numbers each of which indicate the position of the speaker cone for some fraction of a second. These numbers a called samples; audio CDs store sound data at a rate of 44100 samples per second. This means that 44100 times per second, the speaker cone goes to a new place as indicated by the data on the CD. As each sample is stored as 16 bits of data, there are 2^{16}, or 65536 different positions to which the speaker cone can go. This is not the whole story, since obviously the cone travels through an infinite number of points even when traveling between two adjacent positions, but neither properties of continuous functions nor D/A conversion are within the scope of this project.

MIDI files don't store anything nearly as low level as what is described above. MIDI files, in their simplest form (which for purposes of this project is just fine) store data about *notes*. Specifically, for every note pressed, there are just four bytes of data:

Byte 1 The number of ticks since the last note was played. The length of a tick is specified in the header of the MIDI file.

Byte 2 The "note pressed" signal (0x90).

Byte 3 The note number. This value can range from 0x00 to 0xFF; middle C is 0x3C.

Byte 4 The velocity of the note pressed. Between 0x00 and 0xFF.

Thus, the software that does this conversion must analyze the samples of a digital audio file and determine which notes are being played and when. For simplicity's sake, the goal is reduced to converting digital audio files which contain only solo piano. There are two main advantages to this. A piano has 88 distinct notes, in contrast to, for examples, fretless instruments (e.g. violin) and instruments for which adjusting one's embouchure may continuously raise or lower the pitch (e.g. clarinet). The length and tension of the strings that

the piano hammers hit are more or less fixed, and so instead of having to deal with an infinite number of pitches, only 88 need to be dealt with. The other advantage is that the amplitude spikes when a note is pressed. In other words, when a note is struck, its amplitude increases at a very high rate, and then even if it is sustained, it decreases at almost as high a rate. This makes notes on sounded by a piano much more noticeable (visually, on a graph that shows amplitude) than those sounded by a violin, for which a note can start out very quiet and remain very quiet.

I started out by doing Fourier analysis on the raw sound files, using various window sizes, only to find that the data contained a lot of things that were not helpful, and lacked specific information that I needed. Specifically, what the Fourier transform tells me is what the amplitude is for the frequencies of $\frac{sampleRate(Hz)}{windowSize}$ and multiples thereof. Thus, if I want to know the amplitude at some given point in time of A=440Hz, if I used a window size of 1024 samples, I would have to settle for knowing what the amplitude was for frequencies $10 * \frac{44100Hz}{1024} = 430.6640625Hz$ and $11 * \frac{44100Hz}{1024} = 473.7304688Hz$.

One could possibly customize the window size to accommodate for whatever frequency one is trying to find the amplitude of. There are two problems with this. The first problem is that adjusting the window size would affect the relative amplitudes between notes, and would no doubt be a painful process to correct for. The other problem is that the window size must be an integer, and so it is more likely than not that there would be no feasible $windowSize$ that when divided into the sample rate and then multiplied by some coefficient would give us our frequency. For example, if we wanted to find the amplitude for the C above A=440Hz, which is $440Hz \times 2^{\frac{3}{12}} = 523.2511306$, then we would need to find integers to plug into x and y for the following: $x \times \frac{44100Hz}{y} = 523.2511306$, and thus $\frac{x}{y} = \frac{523.2511306}{44100} = 0.011865105 = \frac{11865105}{1000000000}$, and in its reduced form $\frac{2373021}{200000000}$. Remember, y represents the window size, and 200000000 as a window size is just too big (that many samples represents a time length of more than an hour!).

Rather than using Fourier analysis, the software begins by taking the dot-product of each of 88 frequencies and the entire track. Taking the dot-product of two signals shows how similar they are during a given window. Let x be the window size. For each frequency, starting at sample $S_{j=0...numSamples-x}$, the software takes $\sum_{n=j}^{j+x} S_n \cdot cos(\frac{2 \cdot PI \cdot noteFrequency \cdot n}{sampleRate})$. It then does the same thing with a synthesized sine wave (this is the imaginary component), squares the two sums, adds them together, and then takes the square root of that as the actual instantaneous amplitude (energy) of that frequency. Much in the same way that the Fourier transform returns both a real and an imaginary number for each frequency, so does taking the dot product in this manner. That way, the phase of a given sinusoid is irrelevant.

The software is actually broken into three executable files. The first one that needs to be run is `wav2raw`. This is a short script that uses `sox` to convert the `wav` file into a raw sound file that my software can understand (it removes the header information and reduces it from stereo to a single channel). It is run like

this:

```
./wav2raw track01.wav track01.raw
```

thus producing the file `track01.raw`. Then, the raw sound file is processed using `raw2nad`:

```
./raw2nad track01.raw track01.nad
```

NAD stands for Note Amplitude Data. The file that `raw2nad` produces is a binary file that contains the amplitude of each of 88 notes, i.e., a hundred amplitudes are stored for each note per second. This file is processed by `nad2midi`:

```
./nad2midi track01.nad track01.midi
```

`nad2midi` takes the raw NAD information and does some further processing. Specifically, it

1. Runs a crude low pass filter on the data. Each amplitude datum becomes the average of ten samples in front of it and ten samples before it.

2. Computes the continuous instantaneous slope (the derivative?).

3. For each note, looks for places where the derivative crosses the x-axis, from positive to negative. Specifically, for any given point, if five points ahead of it is below the x-axis and five points before it is above the x-axis, it is considered a "candidate", and its amplitude is recorded.

4. The amplitude from the previous step is compared with the average of the amplitudes for each note at that instant. If it is at least eight times the average, it is considered to be a struck note.

5. When a note has been stricken, all candidates that appear for the next quarter second for that same note are thrown out.

After all the NAD information has been gone through, a corresponding MIDI file is created.

In addition, several `.dat` files are created that can be represented as graphs in gnuplot, as an aid to the ongoing effort to improve the heuristics of the program. `raw.dat` is the raw NAD data. For Bach's Prelude in C major, the first thirty seconds of the raw data look like this:

`lowpass.dat` shows the data after it has gone through the lowpass filter. Here are the first ten seconds of the same piece, showing both the raw data and the filtered version of the D above middle C:

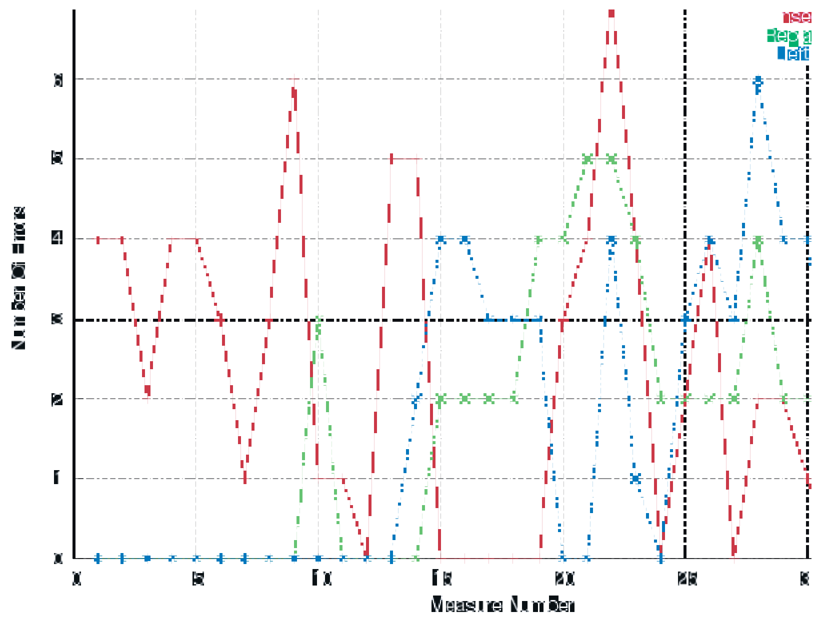

Figure 1: Constructing a torus with a $\{6, 3\}_{(2,1)}$ tessellation.

Here are the same ten seconds, showing just the first derivative:

The main reason the process is separated into two separate executables is that the `raw2nad` takes up to five or ten minutes to run, depending on the speed of the computer. `nad2midi` takes only a few seconds, and separating the two allowed me to tweak `nad2midi` and run it on the same original data without having to wait such an enormous amount of time. As it is, the process that takes place in `raw2nad` in its first incarnation would have taken (by way of extrapolation) about two weeks to run on the same size file that now takes only five or ten minutes.

Statistical error analysis for Bach's prelude in C major (BWV 846)

Errors in determining what notes there are seem to be of three types:

1. The software inserts a note when there isn't supposed to be one

   This is the most prominent error type that occurred, with 77 total. Of these 77, 72 occurred within 5ms of a harmonically related note (viz., the inserted note was the first, second or third harmonic of the note actually played) that was both in the original and in the midi version. 4 out of the remaining 5 were within 1s of a harmonically related note that was both in the original and in the midi version. Only one seemed to be a "random" insertion.

2. The software replaces a played note with a different note

   There were 57 instances of this type of error. 56 of these replaced notes were harmonically related to the note that was supposed to be played. The 1 remaining was harmonically related to a note that occurred less than 1s earlier.

3. The software leaves a note out

   There were 59 instances of this type of error. This error type most closely correlates to the RMS. In particular, when the RMS is relatively low (50s-80s) and when it is relatively high (100s-115s) this type of error is highest.

The following graph shows the number of each type of error measure by measure:

In general, the total number of errors corresponded loosely with the energy of the particular segment of the sound file. In other words, more errors occurred when things got louder. The following two graphs show the total number of errors that occurred and the RMS of the entire piece (with a window of 16384 samples):

Each measure is about 4 seconds long, and thus to relate the timeline of the error graph with the timeline of the RMS graph, simply multiply the measure number by four.

More specifically, notes that were left out were typically lower frequency or lower energy, and notes that were added were typically the result of lower frequency notes being played.

As can be seen by the decrease in the number of errors after the climax

of the piece, the time into the piece does not seem to effect the quality of the conversion.

In summary, out of 134 inserted notes (77 inserted plus 57 replaced, if we consider that a replaced note is both an inserted note and a left-out note), 99% were harmonically related to a note that was actually present. This type of error might be prevented by using a heuristic that, once it is established that some note has been played, decreases its likelyhood of deciding that one of that note's overtones is actually a played note.

The 59 left-out notes closely correlated to the energy at that point in time. These errors might be avoided by using a filter that increases low amplitudes and decreases high amplitudes, thus maintaining a relatively steady energy.

This system of software is by no means in a final stage. The final MIDI file that is created is far from an exact duplicate of the original; once refinements are made and better heuristics implemented to determine which notes are played and when, the MIDI file can also be made to reflect the velocity with which notes are struck, and their envelopes.

Further, additional optimizations can be made; AMD has a fine article discussing optimizations at both the assembly level and at the C level. And, if a DSP chip could be utilized, the MAC instruction would fit right in to the hot spots of `raw2nad`.

Beyond that, the potential exists to transcribe whole orchestras, very likely utilizing neural networks. In its present form, the software can turn piano music into MIDI files that are not perfect, but definitely past the stage of just-recognizable as the piece that it is supposed to be a copy of.