# Drop the Beat

A guide to nearly turning music into notes

Steven Libby

June 6, 2019

## 1 Introduction

This project will automatically transcribe a recording of a snare drum solo. Since this is a project for a 10 week class, I'm making a lot of simplifying assumptions here. First, I'm only transcribing the notes themselves. I'm not worrying about dynamics or accents. I'm also assuming that there is only one instrument. Finally I'm assuming all pieces are in common time. This is likely not true, but since I don't keep dynamic or accent information, it'd be difficult, if not impossible, to guess at other time signatures.

With those caveats there is still a lot of work to do. First I must read in the wave file. I've be using the haskell WAVE library for this. Then I need to determine what point in time each note was hit. This is the beat detection algorithm. Next I need to make a reasonable guess at the tempo of the song. Finally given the time of each note, and the tempo, I need to determine where each note lands in the piece. This is piece reconstruction. Finally I write the notes out to a latex file in the lilypond format.

```
import System.Environment
import Data.WAVE
import Data.List


main :: IO ()
main = do args ← getArgs
          file ← getWAVEFile (head args)
          let (pcm, sampleRate) = toPCM file
          let beats = beatDetect pcm
          let tempo = getTempo sampleRate beats
          let notes = toNotes sampleRate tempo beats
          putStrLn $ showMusic tempo notes


toPCM :: WAVE → (PCM, SampleRate)
toPCM = fork (map (sampleToDouble ∘ head) ∘ waveSamples, waveFrameRate ∘ waveHeader)
```

```
fork :: (a → b, a → c) → a → (b, c)
fork (f, g) x = (f x, g x)
```

## 2 Data Types

There are 10 possible notes we are going to allow, divided into two categories. Straight notes are whole, half, quarter, eighth, sixteenth, and $32^{nd}$ notes. Triplets are divided into half, quarter, eighth, and sixteenth note triplets. In this project I've opted to ignore triplets because it's not feasible to distinguish between triplets and straight notes.

```
type Subdivision = Int
data Note = Straight Subdivision
          | Rest Subdivision
  deriving (Show)
```

```
showMusic tempo notes = "\\score {\n" ++
                        "\\relative c''\n" ++
                        "{\n\\tempo 4 = " ++ show tempo ++ "\n" ++
                        concatMap showNote notes ++
                        "\n}" ++
                        "\n}"
```

```
showNote (Straight n) = "c" ++ (show n) ++ "  "
showNote (Rest n) = "r" ++ (show n) ++ "  "
```

A Tempo is an integer within the range $\{60 \ldots 300\}$ While we could have music at other tempos, it's not likely to come up. The sample rate is the number of samples per minute. A beat is the number of samples since the start of the song. A time is a number of samples. This might be the difference between two beats, or the number of samples per beat. PCM is just a list of doubles representing the amplitude of the audio signal.

```
type Tempo = Int
type SampleRate = Int
type Beat = Int
type Time = Int
type PCM = [Double]
```

# 3 Beat Detection

Our first task is to determine when a drum beat has occurred in our wave file. There are several ways to do beat detection, but my approach is to use is to find the energy of the wave over a small window. If the window has a high amount of energy, then that means that a beat probably happened within that window. So, to determine when a beat happens, I find the energy over a sliding window, and return the points where the energy jumps rapidly.

This has a few advantages over looking of any spikes in the wave. The first is that snare drum hits are themselves waves, so they'll possibly have many spikes in a short amount of time. We could solve this with a lowpass filter, but then I risk losing smaller snare drum hits. This problem is worse than it first seams. Snare drums have a period of about 200 to 500 samples. This means that if I'm measuring peaks, then I how no way to distinguish between a drum hit, and the next peek of a previous drum hit.

Another option is to look at the second derivative of the wave, and look for concave points. However, this suffers from the same problem. The second derivative of a wave is still a wave, and will still produce many spikes. This approach at least has the advantage of amplifying smaller waves. If our wave is $w(t) = A\sin(\omega t)$ then $\frac{dw}{dt} = -A\omega^2 \sin(\omega t)$. So, as long as our frequency $\omega$ is larger than 1 rad/s, the wave will be amplified.

So, how can I compute the energy of the wave $w$. It's really easy. $E_{[a,b]} = \frac{\int_a^b w^2(t)}{b-a}$. Since our window is always the same size I don't need to worry about the proportionality constant. This is a pretty standard integral, so I square the wave, and sum each window along the wave. We can do this in linear time with a sliding window. This is the *getEnergy* function. Because of the problem of the snare drum's long wavelength, I've set the window size to be 500 samples.

Next, I filter out all beats that doesn't have a high enough energy. Right now this is set at 30% of the maximum energy, but that can be changed.

Finally I record what sample each beat was at. We only take the first beat that survived our filter. This avoids the problem of deciding if a large energy is a beat, or simply an earlier beat decaying.

```
beatDetect :: PCM → [Beat]
beatDetect = offset ∘ beats 0 False ∘ candidates 0.3 ∘ energy 500 ∘ map (↑2)
  where getEnergy _      []     _ = []
        getEnergy (x : xs) (y : ys) e = e : getEnergy xs ys (e − x + y)
        energy n xs = getEnergy xs (drop n xs) (sum $ take n xs)
        candidates p e = let f = p ∗ maximum e in map (λx → if x ⩾ f then 1 else 0) e
        beats _ _     []     = []
        beats c True  (0 : cs) =      beats (c + 1) False cs
        beats c True  (_ : cs) =      beats (c + 1) True  cs
        beats c False (0 : cs) =      beats (c + 1) False cs
        beats c False (_ : cs) = c :  beats (c + 250) True (drop 249 cs)
```

$$offset\ (x : xs) = 0 : map\ (\lambda y \to y - x)\ xs$$

# 4 Tempo

Tempo reconstruction is easily the hardest part of this project. There are a few different approaches to this problem.

The first approach is to use a histogram, and just pick the tempo that had the most hits. This is problematic for several reasons, but it's a reasonable place to start.

The next approach is to use a phase lock loop. The idea is pretty straight forward. We want to build a machine outputs a wave at the same frequency as the tempo of the music. If the frequencies of two waves are the same, then the phase of the two waves will remain constant. So, then I'd want to build a machine that locks the phase in. Unfortunately this approach has one major drawback. We don't have a wave. We have discrete pulses. So, while I could generate a click track at a constant frequency, and measure the difference, I can't use that result to feed back into our click track.

The final common approach is to use autocorrelation. Again the idea is pretty simple. I compare our piece to a piece that is offset by a short time. If that short time is the length of a beat, then I'd expect the offset track to sync up with the original track. This seems promising, but I had my own idea that I wanted to try.

As far as I'm aware this idea is entirely novel. That probably means that it's bad, but hey, I wanted to see. It's inspired off the of video "Music and Measure Theory" `https://www.youtube.com/watch?v=cyW5z-M2yzw` That video was aimed at determining if two pitches were in tune, but the problem of finding out if two tempos match up is really the same. It's just a much slower version of that problem.

To do this I compute an "error" for each note that was played. Then the tempo with the smallest error is probably the right one.

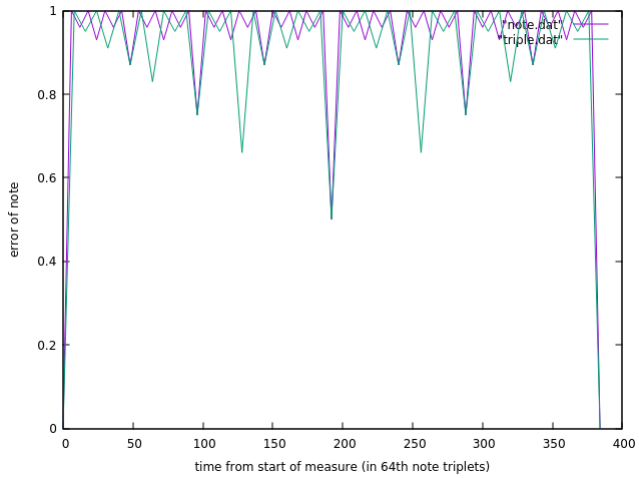
$$\begin{aligned}
&getTempo :: SampleRate \to [Beat] \to Tempo \\
&getTempo\ sampleRate\ beats = fst\ \$\ minimumBy\ cmpErr\ \$\ zip\ tempos\ \$\ map\ totalError\ tempos \\
&\quad \textbf{where}\ tempos = [60\,..\,300] \\
&\qquad\quad cmpErr\ (t1, e1)\ (t2, e2) = compare\ e1\ e2 \\
&\qquad\quad totalError\ t = sum\ \$\ map\ (measureError\ sampleRate\ t)\ beats
\end{aligned}$$

In order to measure the error for a note, I compare the nearest 32nd note. I get the number of samples per beat, and divide that by 8 to get the number of samples per $32^{nd}$ note.

```
measureError :: SampleRate → Tempo → Beat → Double
measureError sampleRate bpm beat = noteError beat 32 note32
  where spb = (sampleRate * 60) `div` bpm
        note32 = spb `div` 8
```

In order to actually measure the error, I develop a new distance from the note to the start of a measure. However, the metric I need is a little weird. I actually need to use a 2-adic metric space. This idea also came from a youtube video `https://www.youtube.com/watch?v=XFDM1ip5HdU`. The idea is that the note at the start of the measure is the most important, then the half note, then the quarter notes, and so on. So I measure how close the beat is to the nearest 32nd note, and multiply by a scaling factor by the importance of the note. Formally, let $\sigma$ the value of the nearest $32^{nd}$ note $p$. $d(x, y)$ be the standard distance on $\mathbb{R}^1$. Then the our new error function is $e(n) = 2 \cdot \sigma d\ (p, n)$. For example, if $p$ is the third quarter note, and the beat is $1/128$ past the nearest note, then our error is $e(n) = 2 \cdot 4 \cdot -1\ /\ 128 = 1/16$

If I graph the error function for both 32nd notes and 16th note triplets, then I get the following graph. You can see that more prominent notes, like the half note, have a much lower error.




In order to actually compute the error first I move our note a 64th note forwards, so that I can take the previous 32nd note instead of finding the nearest one. Then I can apply the error formula above.

```
noteError :: Beat → Int → Int → Double
noteError note subdivision duration = fromIntegral (2 * sigma * dist position center) / fromIntegral subd
  where halfDuration = duration `div` 2
        measure      = duration * subdivision
        center       = (note + halfDuration) `mod` measure
```

```
position      = center `div` duration
sigma         = fromIntegral $ gcd subdivision position
dist p n      = abs (n - (2 * p + 1) * halfDuration)
```

# 5 Piece Reconstruction

The final part of this project is actually come up with the notes that we need. There are a lot of sophisticated models for doing this well. I'm not using any of them.

So it turns out notes don't actually tell you where they are in the measure. They tell you how far it is until the next note. This is our strategy. We take two notes, take the difference between them, and find out which note that lines up with.

```
toNotes :: SampleRate -> Tempo -> [Beat] -> [Note]
toNotes s t [] = []
toNotes s t [a] = notes spb (measure - a)
  where spb     = 60 * s `div` t
        spm     = spb * 4
        measure = ((a `div` spm) + 1) * spm
toNotes s t (a : b : ns) = notes spb (b - a) ++ toNotes s t (b : ns)
  where spb = 60 * s `div` t
```

To find out which note something is, we compute the number of $32^{nd}$ notes between them. Then we just count notes until we've used up all of the 32 notes. We might need to insert rests between two notes if they don't exactly match one of the subdivisions.

```
notes :: Time -> Time -> [Note]
notes spb dn = (if pos > 32 then [Rest 1] else []) ++ noteVal 32 pos
  where note32 = spb `div` 8
    pos        = (dn + note32 `div` 2) `div` note32
```

```
noteVal 1 _ = [Straight 1]
noteVal n x
  | x > n     = Straight (32 `div` n) : restVal (n `div` 2) (x - n)
  | otherwise = noteVal (n `div` 2) x
```

```
restVal 0 _ = []
restVal n x
```

$$
\begin{aligned}
&\mid x > n \quad\;\; = \mathit{Rest}\ (32\ `\mathit{div}`\ n) : \mathit{restVal}\ (n\ `\mathit{div}`\ 2)\ (x - n) \\
&\mid \mathit{otherwise} = \mathit{restVal}\ (n\ `\mathit{div}`\ 2)\ x
\end{aligned}
$$