Marco Ziccardi

Marco Ziccardi - Half computer scientist, half software engineer, half sport and mountain addict.

Beat Detection Algorithms (Part 1)

28 May 2015

You can think to the beat of a song as the rythm you tap your foot at while listening to it. The tempo of a song is usually measured in beats per minute. The tempo of a song, its beats, are usually felt by a listener and drive him, for instance, to dance according to the song's rythm. As it is often the case, things that a human can feel are not easy to detect through a computer program. This is the first of 2 posts on beat detection algorithms in which I introduce two simple algorithms I implemented in Scala scala-audio-file library.

Please notice that so far the library is only able to process WAV files.

Sound Energy Algorithm

The first algorithm I will talk about was originally presented here. I implemented it in the class `SoundEnergyBPMDetector`. The algorithm divides the data into blocks of samples and compares the energy of a block with the energy of a preceding window of blocks. The energy of a block is used to detect a beat. If the energy is above a certain threshold then the block is considered to contain a beat. The threshold is defined starting from the average energy of the window of blocks preceding the one we are analyzing.

If a block *j* is made of 1024 samples and the song is stereo, its energy can be computed as:

\[E _ j = \sum _ {i=0} ^ {1023} left[i]^2 + right[i]^2 \]

The block's energy is then placed in a circular buffer that stores all the energy values for the current window. Let us assume that the current window is made of 43 blocks (\(43 \cdot 1024 = 44032\) ~ \(1s\) with a sample rate of \(44100\)), the average window energy can be computed as:

\[avg(E) = \frac{1}{43} \sum _ {j=0} ^ {42} E _ j \]

We detect a beat if the instant energy \(E _ j\) is bigger than \(C \cdot avg(E)\). In the original article, a way to compute C is proposed as a linear regression of the energy variance in the corresponding window. In the `SoundEnergyBPMDetector` class I use a slightly modified equation that lowers the impact of variance:

$$ C = -0.0000015 \cdot var(E) + 1.5142857 $$

In general, however, the bigger the variance the more likely we consider a block to be a beat. The variance inside a window of blocks is defined as:

$$ var(E) = \frac{1}{43} \sum _ {j=0} ^ {42} (avg(E) - E _ j)^2 $$

This first algorithm is very easy to implement and fast to execute. However, the results computed are rather imprecise. To evaluate it add the scala-audio-file library to your project and instantiate the `SoundEnergyBPMDetector` class as:

```
val audioFile = WavFile("filename.wav")
val tempo = SoundEnergyBPMDetector(audioFile).bpm
```

Low Pass Filter Algorithm

This second algorithm is based on a post on the Beatport's engineering blog by Joe Sullivan. The original article uses the Web Audio Api to implement browser-side beat detection. A similar but more extensible implementation of the algorithm is provided by the class `FilterBPMDetector`. First, the algorithm applies to sampled data a biquad low-pass filter (implemented by the class `BiquadFilter`). Filter's parameters are computed according to the "Cookbook formulae for audio EQ biquad filter coefficients" by Robert Bristow-Johnson. Filtered audio data are divided into windows of samples (whose dimension is the sample frequency, i.e. 1s). A peak detection algorithm is applied to each window and identifies as peaks all values above a certain threshold (defined as $C \cdot avg(window)$ where C is for the user to be configured, e.g. 0.95). For each peak the distance in number of samples between it and its neighbouring peaks is stored (10 neighbours are considered). For each distance between peaks the algorithm counts how many times it has been detected across all windows in the song. Once all windows are processed the algorithm has built a map `distanceHistogram` where for each possible distance its number of occurrences is stored:

```
distanceHistogram(distance) = count
```

For each distance a theoretical tempo can be computed according to the following equation: $$ theoreticalTempo = 60 / (distance / sampleRate) $$ Here the algorithm builds on the assumption that the actual tempo of the song lies in the interval [90, 180]. Any bigger theoretical tempo is divided by 2 until it falls in the interval. Similarly, any smaller tempo is multiplied by 2. By converting each distance to the corresponding theoretical tempo a map `tempoHistogram` is built where each tempo is associated the sum of occurrences of all distances that lead to it:

```
tempoHistogram(tempo) = count
```

The algorithm computes the `tempoHistogram` map and selects as the track's tempo the one with the highest count.

This second algorithm is also very fast and provides better results than the first one. To evaluate it, add the scala-audio-file library to your project and instantiate the `FilterBPMDetector` class as:

```
val audioFile = WavFile("filename.wav")
val filter = BiquadFilter (
      audioFile.sampleRate,
      audioFile.numChannels,
      FilterType.LowPass)
val detector = FilterBPMDetector(audioFile, filter)
val tempo = detector.bpm
```

Conclusion

To sum things up, both algorithms are very fast to implement and execute. The second one is a bit slower but provides better approximations of the actual tempo. More time consuming algorithms that compute almost exact results can be developed by exploiting the Fast Fourier Transform or the Discrete Wavelet Transform. I will discuss such algorithms in the next post.

Beat Detection Algorithms (Part 2)

12 Jun 2015

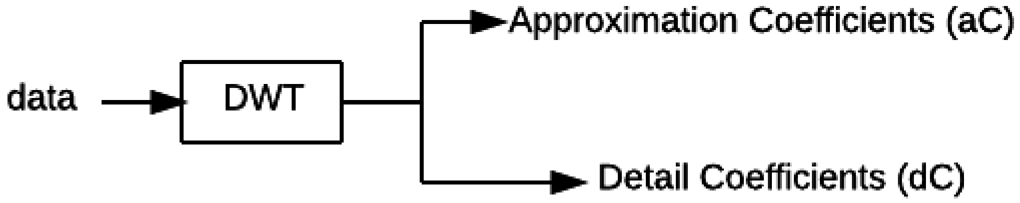
-

This post describes the second part of my journey in the land of beat detection algorithms. In the first part I presented two fast but rather inaccurate algorithms that could be used for beat and tempo detection when performance is much more important than precision. In this post I will present an algorithm (and its implementation) that is more complex (to understand, to code and to run) but that provides incredibly accurate results on every audio file I tested it on.

The algorithm was originally proposed by George Tzanetakis, Georg Essl and Perry Cook in their 2001 paper titled *Audio Analysis using the Discrete Wavelet Transform*. You can find an implementation of the algorithm in the class `WaveletBPMDetector` in the Scala scala-audio-file library.
Please notice that so far the library is only able to process WAV files.

Discrete Wavelet Transform

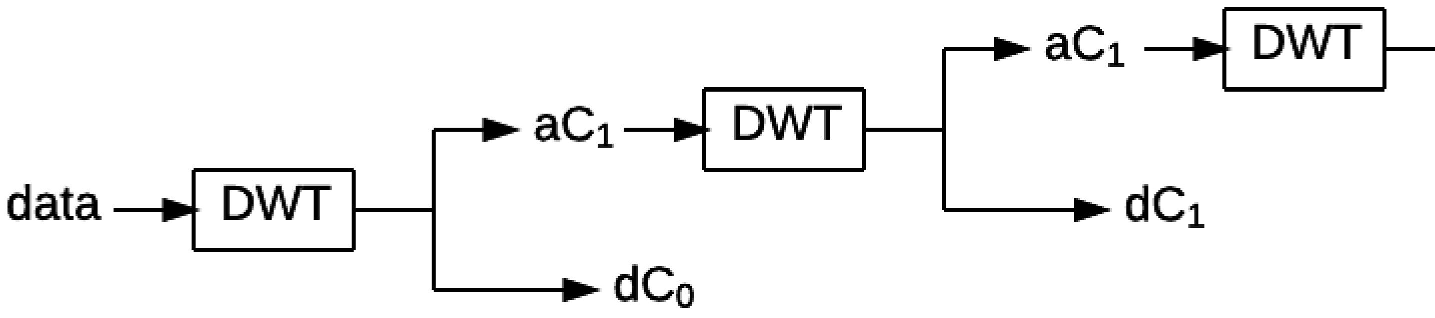
To detect the tempo of a song the algorithm uses the Discrete Wavelet Transform (DWT). A lot could be said on data transformations but this is a bit out of the scope of this post. In the field of audio processing, the DWT is used to transform data from the time domain to the frequency domain (and vice versa). When compared to the widely used Fast Fourier Transfor (FFT), the DWT allows to switch to the frequency domain while keeping information on the time location at several levels of granularity. Roughly speaking, the discrete wavelet transform applies combined low pass and high pass filters and produces some approximation and detail coefficients respectively.



Cascading application of DWT can be applied to increase the frequency resolution of the approximation coefficients thus isolating different frequency sub-bands. For a more correct/precise/detailed description of the discrete wavelet transform have a look at the Wikipedia page.

The Algorithm

The algorithm divides an audio file into windows of frames. The size of each window should correspond to 1-10 seconds of the original audio. For the sake of simplicity we consider a single-channel (mono) track, but the algorithm can be easily applied to stereo tracks as well. Audio data (`data`) of a window is processed through the discrete wavelet transform and divided into 4 frequency sub-bands (4 cascading applications of DWT).



For each frequency sub-band (i.e. its detail coefficients \(dC\)) an envelope is computed. Envelopes are obtained through the following operations:

1. **Full wave rectification**: take the absolute value of the coefficients \[dC'[j] = | dC[j] | ~~~\forall j\]
2. **Downsampling** \[dC''[j] = dC'[k \cdot j] ~~~\forall j\]
3. **Normalization**: subtract the mean \[dC'''i[j] = dC''[j] - mean(dC''[j]) ~~~\forall j\]

Envelopes are then summed together (in \(dCSum\)) and **autocorrelation** is applied to the just computed sum.

\[correl[k] = \sum _ j dCSum[j] \cdot dCSum[j+k] ~~~\forall k\]

A peak in the autocorrelated data corresponds to a peak in the signal envelope, that is, a peak in the original data. The maximum value in the autocorrelated data is therefore identified and from its position the approximated tempo of the whole window is computed and stored.

Once all windows are processed the tempo of the track in beats-per-minute is returned as the median of the windows values.

Implementation

The algorithm is implemented by the class `WaveletBPMDetector` in the Scala scala-audio-file library. Objects of the class can be constructed through the companion object by providing:

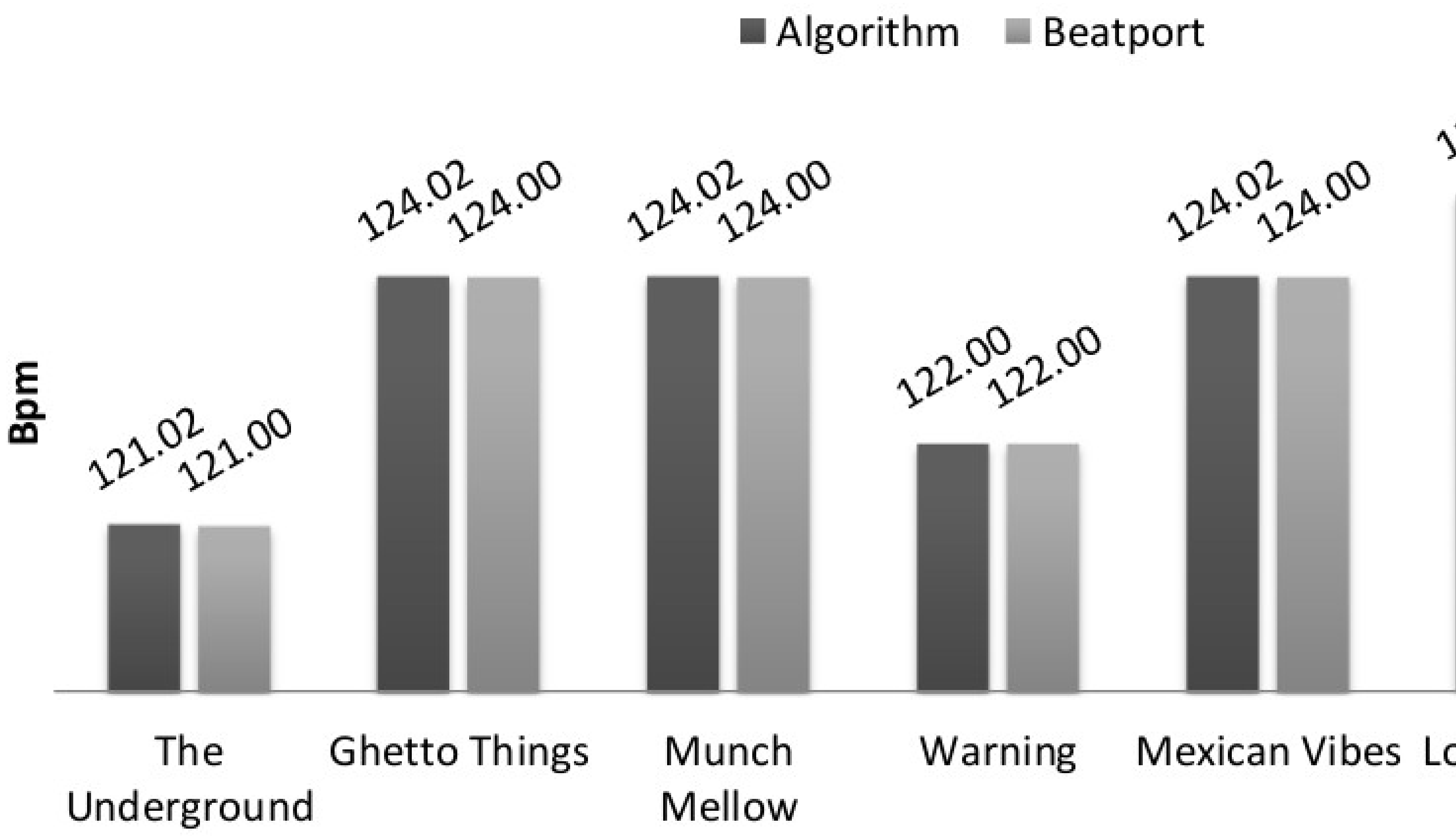
- An audio file
- The size of a window in number of frames
- The type of wavelet

So far only Haar and Daubechies4 wavelets are supported. To evaluate the algorithm add the scala-audio-file library to your project and instantiate the `WaveletBPMDetector` class as:

```
val file = WavFile("filename.wav")
val tempo = WaveletBPMDetector(
              file,
              131072,
              WaveletBPMDetector.Daubechies4).bpm
```

Evaluation

I gave a try to the algorithm/implementation on the release "The Seven Fluxes" that you can find on Beatport. I am developing these algorithms to integrate them in a music store so I expect the user not to be very interested in decimal digits. I take as a reference the tempos provided by Beatport. I am not actually expecting them to be the correct tempo of the track but due to the type of application that will use the algorithm I am satisfied of providing as accurate results as the N.1 store for electronic music. Only the first 30 seconds of each track are used to detect the tempo.



The results show that the algorithm provides the exact same results as Beatport, which is quite cool.

To be fair, the tracks are tech/house music and this makes the task of beat detection easier. A more thorough evaluation of the algorithm is given in the original paper.

Important Notes

- If necessary, the precision of the algorithm can be increased by dividing the data into more than 4 sub-bands (by using more cascading applications of DWT)
- Due to the way the DWT is implemented the size of a window in number of frames must be a power of two. In the example, *131072* approximately corresponds to 3 seconds of an audio track sampled at 44100Hz
- Algorithm implementation could be notably faster. Autocorrelation is in fact performed through brute force and has a \(O(n^2) \) running time. \(O(n \cdot log(n)) \) algorithms for autocorrelation exist as well, I will implement one of them as soon as possible
- The implementation works on stereo signal but detects only beats in the first channel. I still have to figure out how to exploit multiple channels data (max/avg/sum?), as soon as I identify a reasonable approach I will update the implementation (suggestions are appreciated)