A REAL-TIME BEAT TRACKING SYSTEM BASED ON PREDOMINANT LOCAL PULSE INFORMATION

Peter Meier[1,2] **Gerhard Krump**[1] **Meinard Müller**[2]
[1] Deggendorf Institute of Technology, Germany
[2] International Audio Laboratories Erlangen, Germany
{peter.meier, gerhard.krump}@th-deg.de
{peter.meier, meinard.mueller}@audiolabs-erlangen.de

ABSTRACT

Beat tracking is a central topic within the MIR community and an active area of research [1–3]. The scientific progress in this area is mainly thanks to recent machine learning techniques. However, some machine learning approaches are black boxes, hard to understand, and often without direct control over the parameters to adjust the beat tracker. For this demo, we choose a model-based approach that is easy to understand, good for interactions, and well suited for educational purposes [4]. In particular, we present a system for real-time interactive beat tracking based on predominant local pulse (PLP) information, as first described in Grosche et al. [5]. In the first section, we show how the PLP-based algorithm can be transformed from an offline procedure to a real-time procedure. In the second section, we present the implementation of a system that uses this real-time procedure as the centerpiece of an interactive beat tracking application.

1. PLP-BASED ALGORITHM

Before discussing how to convert the PLP-based approach into a real-time procedure, we first look at the original offline procedure. All the essential steps for calculating the PLP are illustrated in Figure 1. First, the audio signal (Figure 1a) is converted into a spectrogram (Figure 1b) by computing a short-time Fourier transform (STFT) with a hop size H and a window length N. From the STFT, we compute a novelty function (Figure 1c) that measures spectral changes over time. The peak positions of this novelty function indicate possible note onset candidates. For beat tracking, there are two general assumptions. Firstly, beat positions go along with note onsets, and, secondly, beat positions are periodically spaced. These assumptions are exploited by comparing local sections of the novelty function with windowed sinusoidal kernels of a defined *kernel size*, as shown in Figure 1e. In particular, for each time position one chooses a kernel that optimally matches the local tempo structure of the signal within a given *tempo range*, as illustrated with colored dots in Figure 1d. To complete the original offline procedure, the last step consists in overlap-adding all optimal pulse kernels over time to form a global PLP function.

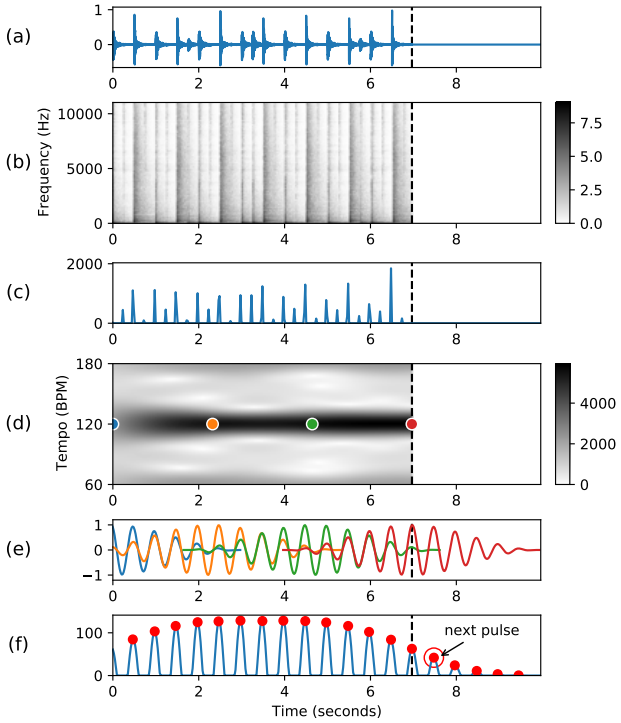

Figure 1. Illustration of the PLP computation pipeline and its real-time implementation. **(a)** Waveform. **(b)** Spectrogram. **(c)** Novelty. **(d)** Tempogram. **(e)** Kernels. **(f)** PLP.

This original offline procedure can now be converted into a real-time procedure, by using the superimposed pulse kernels to predict the next future pulse position, as illustrated in Figure 1f. Due to the centric nature of the local pulse kernels, there are always two halves of the kernel window to work with. The left half of the kernel window is used to calculate the pulse structure based on the current data. The right half of the kernel window is used to extrapolate this pulse structure to predict future pulse positions. In this real-time approach, the pulse kernels serve as local beat trackers and can be used as an engine to drive an interactive real-time system, as described in Section 2.

2. REAL-TIME SYSTEM

In Figure 2 we present our real-time system named *Beat Command Line Interface (beatcli.py)*. It is written in Python and based on the PLP algorithm described in Section 1.

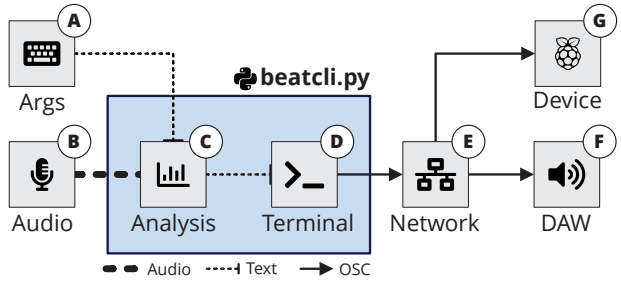

Figure 2. A block diagram of the beatcli.py terminal application. **(A)** Input arguments. **(B)** Audio input. **(C)** Audio analysis. **(D)** Terminal output. **(E)** Network output. **(F)** Receiving software. **(G)** Receiving hardware.

beatcli.py takes input arguments (A) to select a device/channel of the audio input (B) and to set several parameters needed for the audio analysis (C). The application keeps running a complete PLP computation pipeline to predict future pulse positions for every new block of audio from the real-time audio input. If there is a pulse prediction at the current time, there is both a terminal output (D) and a network output (E) with corresponding pulse information. Those network outputs can be received either by software clients (F) such as DAWs [1] or hardware devices (G) such as microcontrollers to apply the send-out pulse information.

```
(.myenv) ➜  applications git:(develop) python beatcli.py -h
usage: beatcli.py [-h] [--device ID] [--channel NUMBER]
                  [--samplerate FS] [--blocksize SAMPLES]
                  [--tempo LOW HIGH] [--lookahead FRAMES]
                  [--kernel SIZE] [--ip IP] [--port PORT]

Beat (C)ommand (L)ine (I)nterface.

optional arguments:
  -h, --help            show this help message and exit
  --device ID           (6) device id for sounddevice input
  --channel NUMBER      (10) channel number for sounddevice input
  --samplerate FS       (44100) samplerate for sounddevice
  --blocksize SAMPLES   (512) blocksize for sounddevice
  --tempo LOW HIGH      ([60, 180]) tempo range in BPM
  --lookahead FRAMES    (0) number of frames (samplerate / blocksize)
                        to lookahead in time and get the next beat
                        earlier to compensate for latency
  --kernel SIZE         (6) kernel size in seconds
  --ip IP               (0.0.0.0) ip address for OSC client
  --port PORT           (5005) port for OSC client
(.myenv) ➜  applications git:(develop)
```

Figure 3. The help function of the beatcli.py application with information about input arguments.

Figure 3 shows the help function of *beatcli.py* to explain the input arguments in more detail. The processing of the real-time audio input is based on the Python module *sounddevice* [2]. It receives the first four arguments from *beatcli.py* to set the sound device of choice. Each audio **device** has its own unique `ID` and a `NUMBER` of **channels** to choose from. Other important settings are the **samplerate**, given as frequency `FS` in Hz, and the **blocksize**, given in `SAMPLES`. The blocksize, however, does not only set the *buffersize* of the audio hardware, but also corresponds with both the hop size H and half the window length N of the spectrogram as shown in (1).

$$\text{blocksize} = \text{buffersize} = H = \tfrac{1}{2}N \qquad (1)$$

The next three arguments are needed to control the settings of the pulse analysis. The **tempo** range of the tempogram (see Figure 1d) is set with a `LOW` and a `HIGH` tempo value given in BPM. The argument **lookahead** takes a number of `FRAMES` to look ahead in time and get the next predicted pulse some frames earlier. This method is helpful to compensate for latency effects that might occur when sending and receiving pulse information over the network. The **kernel** `SIZE`, given in seconds, determines the duration of the local pulse kernels as illustrated in Figure 1e. Finally, there are two more arguments to set the parameters of the network output: *beatcli.py* acts as OSC [3] client to send messages with pulse information into the network. Another client with a specific `IP` address and a selected `PORT` number can receive those messages.

```
(.myenv) ➜  applications git:(develop) python beatcli.py
Beat (C)ommand (L)ine (I)nterface: {'device': 6, 'channel': 10, 'sampl
erate': 44100, 'blocksize': 512, 'tempo': [60, 180], 'lookahead': 0, '
kernel': 6, 'ip': '0.0.0.0', 'port': 5005}
OSC to 0.0.0.0:5005: time=13:40:59.998 tempo=60 stability=1.000
OSC to 0.0.0.0:5005: time=13:41:01.083 tempo=120 stability=1.000
OSC to 0.0.0.0:5005: time=13:41:01.582 tempo=120 stability=0.585
OSC to 0.0.0.0:5005: time=13:41:02.081 tempo=120 stability=1.000
OSC to 0.0.0.0:5005: time=13:41:02.580 tempo=120 stability=0.933
OSC to 0.0.0.0:5005: time=13:41:03.079 tempo=120 stability=1.000
OSC to 0.0.0.0:5005: time=13:41:03.579 tempo=120 stability=1.000
OSC to 0.0.0.0:5005: time=13:41:04.078 tempo=120 stability=1.000
OSC to 0.0.0.0:5005: time=13:41:04.577 tempo=120 stability=1.000
OSC to 0.0.0.0:5005: time=13:41:05.076 tempo=120 stability=1.000
^C
--- beat statistics ---
10 beats transmitted
tempo min/avg/max/stddev = 60.00/114.00/120.00/18.00
stability min/avg/max/stddev = 0.585/0.952/1.000/0.124
(.myenv) ➜  applications git:(develop)
```

Figure 4. The terminal output of the beatcli.py application showing the system in action.

Figure 4 shows the terminal output of *beatcli.py* to explain the send-out pulse information in more detail. By starting the application, you get an overview of all (default) settings it is running on. Below, you can find a table of detected pulses, where each row contains several different columns with pulse information. First, there is the `IP` address and `PORT` of the send-out OSC message. Next, there is a **time** value that gives a timestamp of when the pulse is detected. After that, there is a **tempo** value that shows the tempo of the current local pulse kernel, which corresponds with the last (red) dot in Figure 1d. Finally, there is a **stability** value to indicate how stable the beat estimation was over the last few seconds, specified by the duration of the kernel size. A value of 1.0 represents a beat with a steady tempo and maximum stability. Values close to zero represent a rather unstable tempo and beat structure. This wraps up our demo, where we introduced a real-time beat tracking system based on predominant local pulse information.

[1] Digital Audio Workstations

[2] https://pypi.org/project/sounddevice/

[3] Open Sound Control

3. ACKNOWLEDGMENTS

This work has been supported by the BayWISS Joint Academic Partnership "Digitalisation." The International Audio Laboratories Erlangen are a joint institution of the Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU) and Fraunhofer Institute for Integrated Circuits IIS. The icons used in Figure 2 are licensed under the Font Awesome Free License (https://fontawesome.com/license/free).

4. REFERENCES

[1] M. E. P. Davies and S. Böck, "Temporal convolutional networks for musical audio beat tracking," in *27th European Signal Processing Conference, EUSIPCO 2019, A Coruña, Spain, September 2-6, 2019*. IEEE, 2019, pp. 1–5. [Online]. Available: https://doi.org/10.23919/EUSIPCO.2019.8902578

[2] S. Böck and M. E. P. Davies, "Deconstruct, analyse, reconstruct: How to improve tempo, beat, and downbeat estimation," in *Proceedings of the 21th International Society for Music Information Retrieval Conference, ISMIR 2020, Montreal, Canada, October 11-16, 2020*, 2020, pp. 574–582. [Online]. Available: http://archives.ismir.net/ismir2020/paper/000223.pdf

[3] H. Schreiber, F. Zalkow, and M. Müller, "Modeling and estimating local tempo: A case study on Chopin's mazurkas," in *Proceedings of the International Society for Music Information Retrieval Conference (ISMIR)*, Montréal, Canada, 2020, pp. 773–779.

[4] M. Müller and F. Zalkow, "FMP Notebooks: Educational material for teaching and learning fundamentals of music processing," in *Proceedings of the International Society for Music Information Retrieval Conference (ISMIR)*, Delft, The Netherlands, 2019, pp. 573–580.

[5] P. Grosche and M. Müller, "Extracting predominant local pulse information from music recordings," *IEEE Transactions on Audio, Speech, and Language Processing*, vol. 19, no. 6, pp. 1688–1701, 2011.