Team Name: Impulse **Team ID: 24764**

Introduction:

This python module has been built and tested to perform Real-Time Beat Tracking on a Raspberry Pi 3B. The way it is able to output beats in real-time is by predicting future beats. The appropriate beat is shown visually through peripherals connected to an Arduino Mega 2560 in real-time from the predicted beat times. It uses maximum filter vibrato suppression for onset detection and performs beat tracking using dynamic programming. The main design goal was efficiently tracking beats with the lower computational resources of a Raspberry Pi. It runs smoothly on any device with better or equal resources.

Dependencies and Pre-requisites:

- Python (2.7)
- Numpy
- Scipy
- ffmpeg
- PyAudio
- CFFI (C Foreign Function Interface for Python)
- Six (Python 2 and Python 3 compatibility library)
- PortAudio Version 19
- serial (only required if paired with an Arduino)

Usage:

- First import the following packages:

```
>> from BeatTracker import BeatTracker
```

- Create a **BeatTracker** Instance.

```
>> proc = BeatTracker()
```

class **BeatTracker**_(output='terminal',serialName= 'None')_

Parameters: output: string

Set as 'terminal' (default) to print beat times in sync with input audio in the terminal window when using methods **wavStream** or **MicrophoneStream**.

Set as 'serial' to send predicted beat times to peripheral device via serial communication when using methods **wavStream** or **MicrophoneStream.** Port number of the peripheral must be specified as the second argument.

serialName: port number/ serial ID, string

The port number/ serial ID of the peripheral device connected via USART. Required if using multiple embedded devices in parallel.

Returns: **BeatTracker** instance

- Use any of the three methods to track beats in different modes.

  1. Using the _**Beats**_ method of the BeatTracker instance to read data from a wavfile sequentially.

BeatTracker.Beats_(InFile='sample.wav', OutFile='sample.txt')_

Parameters: InFile : Filename, string

Name of the file to read the data from. The audio file is read sequentially. Defaulted as 'sample.wav' which is an audio file provided for testing.

OutFile : Filename, string

Name of the file to write the predicted beat timestamps. Defaulted as 'sample.txt' for testing.

Returns: numpy array

Array containing the beat times of the predicted beats.

Note:

The method **Beats** reads the wavfile, sequentially feeds the input to the beat tracker and returns the predicted future beat times of the excerpt. The **Beats** method compared to the **wavStream** method, imports the wavfile instead of streaming it as frames. The data processing is first performed using the first 4s and is done in a sequential manner after each 1s (default) chunks which predict future beats of length 1.1s (default). The predicted beat times are also written to the txt file specified. The method has been designed for quick evaluation of the algorithm.

2. Using the ***wavStream*** method of the BeatTracker instance to open an audio stream (Non-Blocking) to acquire audio frames streamed as input from a .wav file, as frames of 2048 samples. This also prints **timestamps of the predicted beats in Real-Time** in the terminal window.

BeatTracker.wavStream*(InFile='sample.wav', OutFile='sample.txt')*

Parameters: InFile : Filename, string

Name of the file to read the data from. The audio file is streamed as an input. Defaulted as 'sample.wav' which is an audio file provided for testing.

OutFile : Filename, string

Name of the file to write the predicted beat timestamps to. Defaulted as 'sample.txt' for testing.

Note:

The method **wavStream** uses *PyAudio* to open a non-blocking stream of audio from the wavfile, reads the audio file in frames of 2048 samples at a rate of 44100 samples per second. It also plays the wavfile in parallel and prints the beat times of the excerpt in sync with the song if 'terminal' is specified as the first argument when initiating the class. The **wavStream** method has been designed to be able to test the real-time output along with the audio file.The predicted beat times which were synchronously displayed are also written to the txt file specified, for further evaluation.

3. Using the ***MicrophoneStream*** method of the BeatTracker instance to open an audio stream (Non-Blocking) to acquire audio using a microphone. This also prints **timestamps of the predicted beats in Real-Time** in the terminal window.

BeatTracker.MicrophoneStream*()*

Note:

The method **MicrophoneStream** uses *PyAudio* to open a non-blocking stream of input using the default input device, reads the audio file in frames of 2048 samples at a rate of 44100 samples per second. The **MicrophoneStream** method has been designed to perform audio acquisition using a microphone and output the predicted beat times in sync with the microphone input if 'terminal' is specified (see class **BeatTracker**).

Tutorials:

Run **test.py** to test the different modes of beat tracking. The terminal will prompt to press 'return' key to start streaming when using **MicrophoneStream** and **wavStream** methods.

Hardware Setup:

Device: Raspberry Pi – Model 3B for data acquisition and processing, connected with an Arduino MEGA via USART which controls the interfaces related to the creative visual output.

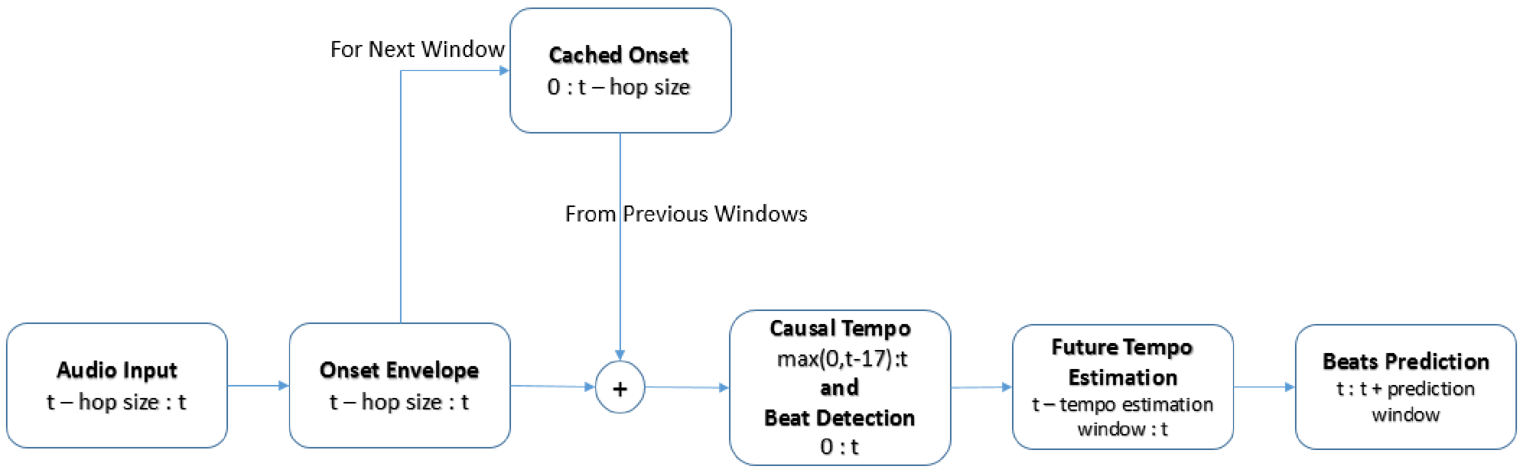
OS: Raspbian Jessie, Version: November 2016, Release Date: 2016-11-25

Input Device: MXL 990 Condenser Microphone, connected with an XLR to USB adapter (MXL USB Mic Mate Classic). The microphone was therefore connected to the RPi via USB. Note that PortAudio should be able to find the microphone, and it should be set as the default PortAudio device before using the module.

Note:

The Raspberry Pi is used to acquire audio and predict beat times in parallel. It sends the predicted beat times using serial communication to the Arduino MEGA. The clocks of the Arduino and Raspberry Pi are matched before starting audio acquisition. Furthermore all the methods of class **BeatTracker** has also been tested on **Ubuntu and Windows**.

Short Description of Algorithm:



Audio acquisition is continued till the first 4s are obtained. The onset envelope is then calculated and subsequently tempo estimation and beat tracking is performed. The beat positions are used to determine future tempo and predict beats. Subsequently, after each hop size (default 1s) time has passed the onset is evaluated in the last hop size data. This is appended with the previous onsets. Tempo and beats are again predicted. This method is repeated after each hop size data has been acquired. The beat positions over each prediction window are the beat positions that are written to output streams.

Challenges:

1. We have tried out different Recurrent Neural Network (LSTM/BLSTM) based Beat Tracking methods. All tested algorithms were initially developed on MATLAB. The algorithms were later converted to be compatible with Raspberry Pi. The main challenge with these algorithms were running them in real-time with limited computational resources. Multithreading routines were deployed but they were not good enough to produce satisfying results. We were not able to utilize the onboard GPU of the Raspberry Pi.

2. We have chosen Raspberry Pi as our preferred development board because it is cheap, well documented and widely available. This also makes deploying a Real-Time beat tracker in real life scenarios easier. The trade-off was the processing power limitations.

3. Onset Envelope calculation is the most computationally expensive part of the algorithm. So, Onset Memoization was performed to cache the onset calculation during previous segments. This reduces the overall evaluation time dramatically and enables Real-Time implementation on Raspberry Pi.

4. Hop size defines the gap (in seconds) between subsequent calculations. The large hop size and prediction window is to accommodate sufficient time to Raspberry Pi to estimate tempo and perform beat tracking over the largest possible segment. This ensures that the predictions of beat which are, in fact our outputs are optimized for the Raspberry Pi.

Citation:

1. The src.py file is built on top of modified LibROSA to adhere to our real-time requirements

   Brian McFee, . Matt McVicar, . Colin Raffel, . Dawen Liang, . Oriol Nieto, . Eric Battenberg, . Josh Moore, . Dan Ellis, . Ryuichi YAMAMOTO, . Rachel Bittner, . Douglas Repetto, . Petr Viktorin, . João Felipe Santos, and . Adrian Holovaty, “librosa: 0.4.1”. Zenodo, 17-Oct-2015.

2. Böck, Sebastian, and Gerhard Widmer. “Maximum filter vibrato suppression for onset detection.” 16th International Conference on Digital Audio Effects, Maynooth, Ireland. 2013.

3. Ellis, Daniel PW. “Beat tracking by dynamic programming.” Journal of New Music Research 36.1 (2007): 51-60. http://labrosa.ee.columbia.edu/projects/beattrack/