Real Time Tempo Analysis of Drum Beats

Author: **Philip Hannant**, Supervisor: **Professor Steve Maybank**

Birkbeck, University of London
Department of Computer Science and Information Systems

Project Report
MSc Computer Science

September, 2016

Contents

Abbreviations

BPM	Beats Per Minute
DFT	Discrete Fourier Transform
DWT	Discrete Wavelet Transform
FFT	Fast Fourier Transform
JSON	Javacript Object Notification
SBT	Simple Build Tool
TDD	Test Driven Development

Definitions

Acoustic Drum Kit	A collection of drums and cymbals which do not have electronic amplification. Typically made up of a bass drum, snare drum, tom-toms, hi-hat and 1 or more cymbals.
Beat	For the purpose of this project a beat will be defined as the sequence of equally spaced pulses used to calculate the tempo being played by the drummer.
Drum Module	The device which serves as a central processing unit for an electronic drum kit, responsible for producing the sounds of the drum kit.
Electronic Drum Kit	An electrical device which is played like an acoustic drum kit, producing sounds from a stored library of instruments and samples.
MIDI	Musical Instrument Digital Interface is a protocol developed in the 1980's to allow electronic instruments and other digital musical tools to communicate with each other[1].
Tempo	The speed at which a piece of music is played [2] and counted in beats per minute (bpm).

1 Introduction and Background

This project report presents my aim to develop a real-time drum beat tempo analysis system using different beat detection algorithms which is able to record the performance of each method when an extensive set of drum samples, representing a real drummer's performance, is processed through the system.

1.1 Drumming Training Tools Background

Timing is the fundamental skill any good drummer should possess and is the staple by which they will be judged. For many years the only training tool available to a drummer to improve their timing was the metronome. An instrument used to mark musical tempo, erroneously attributed to Johann Nepomuk Maelzel in 1815 but was actually invented by a Dutchman, Dietrich Nikolaus Winkel a year earlier. The traditional metronome, based on Winkel's original design is a hand-wound clockwork instrument that uses a pendulum swung on a pivot to generate the ticking which depicts the desired tempo [3] is still used today by musicians, as seen in Figure 1.

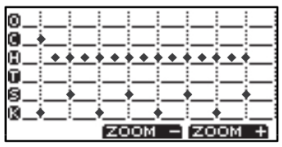
Figure 1: Traditional Metronome

For drummers however the electronic versions of the metronome are much more widely used, to the point that metronomes are now developed with functionality specifically tailored to a drummers training requirements. The Tama Rhythm Watch (Figure 2) was the first metronome designed specifically for drummers, providing enough volume to be used with real drums as well as allowing for the use of different time signatures[1] and preset set rhythm patterns to help improve performance.

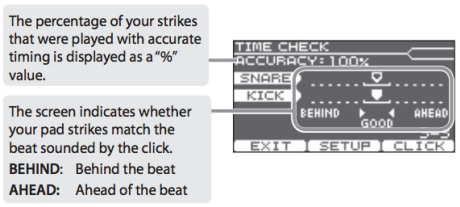
Figure 2: Tama Rhythm Watch

[1] A details explanantion of time signatures can be found in section 1.2.2

Following the development of the MIDI driven electronic drum kit came the development of more advanced training tools that were able to provide live feedback to drummer during any given performance. Today the leaders in this field are Roland, their V-Drums line provide a variety of tuition packages including the SCOPE and more recently the COACH system provided in the V-Drum modules. The SCOPE system provides live feedback to the drummer by plotting their performance on a grid representing the elements of the drum kit, Figure 3a. The COACH system offers an improved user experience by providing an accuracy figure relating to the number of hits which the drummer has played in time, Figure 3b.



(a) Screenshot of the SCOPE system



(b) Screenshot of the COACH system

Figure 3: SCOPE and COACH training tools

The V-Drum Rhythm Coach line is an advanced version of the traditional drummers practice pad (Figure 4a and the extensive DT-1 V-Drums tutor software package (Figure 4b. Roland have even

(a) The RMP-5 Rhythm coach practice pad

(b) Screenshot of the DT-1 V-Drums Tutor

Figure 4

used gamification within this field with their latest release, the V-Drums Friend Jam app. The application itself provides the player with live feedback and evaluates each performance in order to provide the player with a score which they can share over social media, a screenshot can be seen in Figure 5.

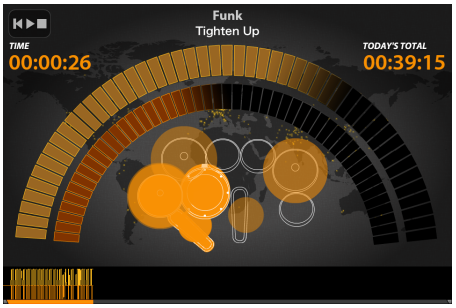


Figure 5: Example of the Roland Friend Jam software package

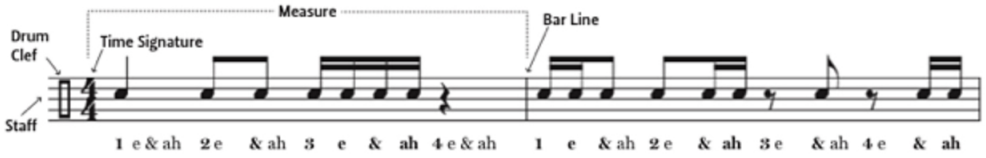


Figure 6: Example of a staff used in musical notation

Despite these developements there have been a limited number of tools available to be used with acoustic drums. A look at the the App Store[2] and Google Play[3], reveals a small number of general live bpm detectors an no applications which are focused towards a drummers training needs.

1.2 Drum Musical Theory

In order to understand the fundamentals of musical timing some theory needs to be examined but beforehand the concept of a drummer playing time must be considered. Time, in a drumming sense is an informal term used to describe the consistent rhythmic pattern that a drummer will play on the hi-hat or ride cymbal [4] and it can be considered one of the most important components of any drum beat.

1.2.1 Notation

Drum music notation is written on staff that is made up of five individual lines, the clef is found on the far left of the staff which indicates the pitch of the notes [2] and as percussion instruments are non-pitched they use the percussion-clef. On traditional musical notation the lines and spaces between represent a tonal where as for drum notation, notes written on lines or spaces indicate a certain drum or cymbal. The staff is separated into individual measures which are known as bars [5] and it is these bars that are the basis of musical time, an example of a staff can be seen in Figure 6. For the purpose of this project it is the count of these beats that will be used to calculate the tempo of a certain drum beat.

1.2.2 Time Signatures

Time signatures appear on the staff just after the clef and are written as a fraction where the top number indicates the number of beats that there are in a bar, as seen in Figure 6. With the bottom number representing the size of the note that makes up the duration of one beat. For example the straight time four four (4/4) or common time signature indicates four beats in each bar or measure where each beat is made up of one quarter note [5]. Within these bar lines beats can be further divided by using a technique known as subdivision, which is a method for reducing the pulse or rhythm pattern into smaller parts than those originally written, for example counting a four four (4/4) measure in eighth (1/8) or sixteenth (1/16) notes.

[2]Apple's application store - itunes.apple.com/uk/appstore
[3]Google's application store for Android devices - play.google.com/store

1.2.3 Notes

The notes used to represent what percussive instrument is to be played also provide the duration it should be played for. Notes come in different lengths and the key values are the whole (1/1), half (1/2), quarter (1/4), eighth (1/8) and sixteenth (1/16). For example two eighth notes represent the same time value as a single quarter note. It is also possible to divide note values by three instead of two, these notes are known as triplets. An eighth note triplet is played fifty percent faster than a normal eighth note, therefore for every two eighth notes there will be three eighth note triplets [5]. An example of the eighth-note triplets being used is in a twelve eight (12/8) jazz shuffle, the time element played on the ride cymbal or hi-hat is charaterised by playing the first and third triplet of an eighth-note triplet grouping [4].

1.2.4 Playing Basics

With the basics of drum theory covered it is now possible to discuss the key elements of a drum beat, typically for a straight four four (4/4) beat, the bass drum will be played on the first and third beat and the snare drum will be played on the second and forth beat both as quarter notes. This is more commonly known as a back beat [4]. This just leaves the time element which will usually be played on the ride cymbal or hi-hat, this too could be played using quarter notes on the first, second, third and forth beats. However, in order to make the drum pattern more dynamic the time element will usually be played using subdivisions, typically using eighth note subdivisions. The ride cymbal or hi-hat will therefore be played on the first, second, third and forth beats as well as the eighth notes in-between each quarter note. This can be demonstrated by counting the one-and-two-and-three-and-four-and, where the "and" represents the subdivided eighth note. Additionally to this technique a drummer will usually ensure that there is difference in the volume of the eighth notes being played on the quarter notes and those being played on the "and". This technique of emphasising certain beats is known as accenting.

1.3 Beat Detection Background

Most of the early work on beat detection was a by-product of research directed at other areas of musical understanding. The earliest work in this field can be attributed to H. C. Longuet-Higgins, who in 1976 while researching the psychological theory of how Western musicians perceive rhythmic and tonal relationships between notes. Produced an algorithm that was able to follow the beat of a performance and adjust the perceived tempo accordingly based on whether a note started earlier or later than expected [6]. Longuet-Higgins' work was built on the premise that in order to percieve the rhythmic structure of a melody it is first necessary to identify the time at which each beat occurs [7], otherwise know as onset detection. The onset of a note is the instant which marks the start of the variation in the frequency of a signal, a visualisation of this can be seen in Figure 7. Once detected it can then be used to measure the onset times of sonic events[4] within a piece of music[11]. These onset times are then used within a beat detection algorithm in order to calculate a piece of music's tempo.

Since Longuet-Higgins' first work the area of beat detection has expanded rapidly, in 2005 the first annual Music Information Retrieval Evaluation eXchange (MIREX) was held in 2005. MIREX includes a contest with the goal of comparing state-of-the-art algorithms for music information retrieval [14]. The topics to be evaluated are proposed by the participants. In the first year,

[4] A sonic event is a singular feature of a piece music which can be made up of one source or many[12], e.g. the hitting of a drum

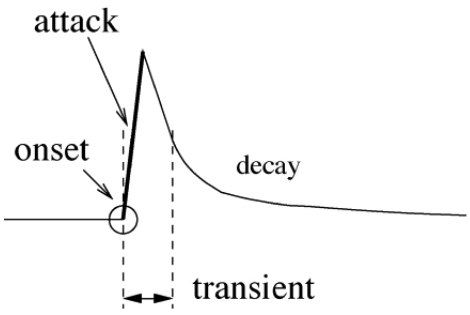


Figure 7: The onset of a note is the instant which marks start of the variation in the frequency of a signal (Image Source: [13])

three of the nine topics concerned beat detection, including audio onset detection and since it's first inclusion it has been an evalulated topic of all but one of the last twelve contests [11]. The beat detection algorithms proposed to perform the tempo analysis for this project, the Beatroot system developed by Simon Dixon [8] and a development on the original audio analysis using the Discrete Wavelet Transform (DWT) by Tzanetakis, Essel and Cook [15]. Are both former entrants to the MIREX contest, with the beatroot system receiving the highest score in the 2006 Audio Beat Tracking task [16]

1.4 Project Aims

The primary aim of this project is to investigation whether some of the currently available beat detection algorithms are accurate enough to form the basis for a training tool to be used by drummers practicing on an acoustic drum kit. In order to achieve this the developed software package, hereafter referred to as RTT_Analyser[5], will need to process enough live audio in form of sampled drum beats and record each of the chosen beat detection algorithms accuracy.

The original core features of the RTT_Analyser developed for this project are:

1. A live audio tempo analysis tool, that compares and records the performance of selected beat detection algorithms
2. The RTT_Analyser implements an adapted version of the beat tracking system Beatroot and Discrete Wavelet Transform algorithm described by Tzanetakis *et al*'s[15] which process live audio as opposed to originally designed off-line audio files.
3. RTT_Analyser implements a system capable of parallel processing, allowing for the same captured live audio data to be sent to the chosen beat detection algorithms in order for the tempo to be calculated simultaneously.
4. While processing live audio the RTT_Analyser stores a predetermined data set in order to allow for performance analysis of the beat detection algorithms.
5. The RTT_Analyser will provide the user with real time feedback of the most recent tempo calculation returned
6. An extensive sample set of drum beats will need to be created to ensure the system is tested sufficiently

[5]Where RTT stands for Real Time Tempo

2 RTT_Analyser Beat Detection Algorithms

The original proposed solution incorporated two beat detection algorithms, Beatroot system [8] and the DWT method [15]. However, the adaptation of the Beatroot system to be used with live audio took longer than the proposed time-frame. This meant that an alternative system needed to be found in order to mitigate this issue, conveniently the Beatroot software package also contained another beat detection system, the Performance Worm [10]. In order to ensure the project remained on track it was decided to substitute the Performance Worm for the Beatroot system. The intention was, if time allowed, to resume the adaptation of the Beatroot system to work with live audio once the project was ahead of schedule and this was successfully completed prior to the development of the user interface.

The project report will now discuss the beat detection algorithms incorporated in the RTT_Analyser as well as the system used to allow for simultaneous tempo calculation.

2.1 Beatroot

Beatroot is a beat detection software package created by Simon Dixon [8], which was originally designed extract musical expression information musical recordings[10]. Beatroot was included in the RTT_Analyser as the algorithm designed by Dixon was considered to be sufficiently fast enough to be implemented as part of a real-time system[17].

Beatroot works by first obtaining a time-frequency representation of the signal based on a Short Time Fourier Transform (STFT) using a Hamming window[9]. The STFT is a form of Fourier transform (FT), which can be used to find out how much of each frequency exists in a signal. Th A negative of the FT is that it is unable to provide any details of when a frequency component occurs in time for non-stationary signals[6]. A solution to this is to split a non-stationary signal up into a number of smaller segments using a window function, which effectively created a series of stationary[7] signals which the FT could then be applied to. By splitting the signal into smaller segments the STFT is able to apply the DFT to these segments and essentially express a signal as a linear combination of elementary signals that are easily manipulated. The DFT returns a spectrum that contains information about how the energy held within the signal is distributed in the time and frequency domains[20]. The use of a Hamming window function in the STFT employed by Beatroot is an attempt to reduce the amount of additional frequencies appearing in the returned DFT spectrum, known as spectral leakage, which is caused when the steep sloped of the a rectangular window causes the frequencies to become distorted. The Hamming window counteracts this by using a bell shape, which has the effect of reducing amplitudes of its sidelobes[19] and ensures the spectrum returned is less spread out and closer to the ideal theoretical result[20]. A visualisation of a Hamming window can be seen in Figure 8.

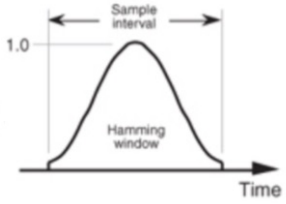


Figure 8: (Image Source: [19])

[6]Non-stationary signals are signals whose frequency contents changes over time[18]
[7]The frequency contents of a stationary signal does not change over time

Once the time-frequency representation is returned the next stage is to the spectral flux onset detection function, which is a method for measuring the change in magnitude of each returned frequency bin[9]. The onsets are then selected from the spectral flux onset detection function by a peak-picking algorithm which finds local maxima within the detection function. The next stage is to apply the tempo induction algorithm which is used to compute clusters of inter-onset intervals (IOI) by using the calculated onset times. Each cluster represents a hypothetical tempo, in seconds per beat [8]. The clustering algorithm works by assigning an IOI to a cluster if its difference from the cluster is less than 25ms. The cluster information is then combined by recognising the approximate integer relationships between clusters. An example of this can be seen in Figure 9 where cluster C2 is twice as long as C1 and C4 is twice that of C2. This information along with the number of IOIs within a cluster is then used to weight each cluster which is then returned as a ranked list of tempo hypotheses[17].

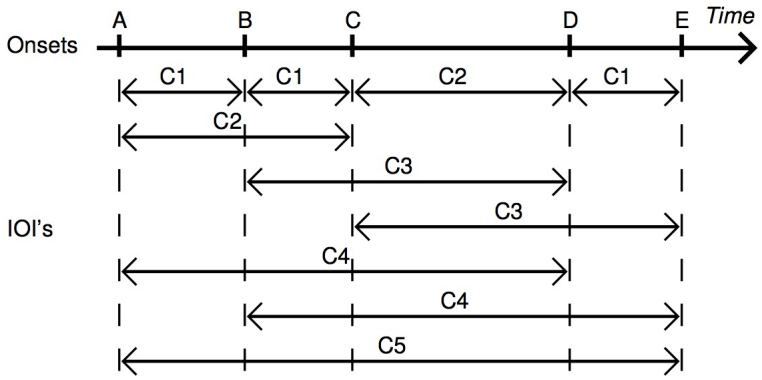


Figure 9: (Image Source: [17])

The multiple agent architecture of Beatroot's beat tracking subsystem is then employed to find the sequences of events that closest match the original tempo hypotheses, each of these sequences is then rated and the most likely set of beat times is determined. Each of the agents in initialised with a tempo or beat rate hypothesis and an onset time. Further beats are then predicted by the agent based on these parameters. Any onsets corresponding with the predicted beat times are taken as a beat time, while those that fall outside of the are considered not to be a beat time, although the possibility that the onset is not on the beat is considered. Then agents then rate themselves based on an evaluation function which looks at how evenly spaced the beat times are, the number of predicted beats which relate to actual events, and the salience[8] of the matched onsets. The agent with the highest score is then returned as the sequence of beats corresponding to the processed audio[17]. It is from the sequence of beats which this agent returns that overall tempo of the audio can be determined by using the "inter-beat intervals, measured in beats per second"[8] to calculate beats per minute (bpm) of the audio.

2.2 Discrete Wavelet Transform and Beat Detection Method

The first literature regarding the wavelet was provided by the mathematician Albert Haar in 1909 [18]. The wavelet transform is a technique for analysing signals which was developed as an alternative to the STFT[15]. Like the STFT, the DWT is able to provide time and frequency information, however, unlike the STFT the DWT is able to do this without the need for a window function. The DWT can essentially be considered to be a filter-bank, where a filter-bank is a system used

[8]The salience is a measure of the note duration, density, pitch and density[8], which is calculated from the spectral flux of the onset[17]

to separate sub-bands by using an array or bank of filters, where each of the filters corresponds to half frequency range of the closest centre higher frequency. Thus each filter will have half or twice the bandwidth of any of its adjacent filters.

In 2001, Tzanetakis *et al* described how the Discrete Wavelet Transform (DWT) could be used to extract information from non-speech audio[15]. Their beat detection algorithm was based on the detecting the most prominent signals which are repeated over a period of time within the analysed audio. The first stage is to process is to split the signal into a number of octave[9] frequency bands with the DWT. This allows for the time domain amplitude envelopes of each frequency band to be extracted separately. The extraction of these envelopes is completed in the following three steps:

1. Full Wave Rectification - process of converting the amplitude of each frequency band to one polarity[23], which can be either positive or negative. A visual representation can be seen in Figure 10.
2. Low Pass Filtering - Low pass filtering is a signal processing technique which is designed to allow frequencies below a cutoff frequency through but blocks any frequencies above the cutoff frequency[22].
3. Down-sampling - Due to the large periodicities that can occur in beat analysis, down-sampling the signal reduces the computation time of the autocorrelation stage without causing any negative effects on the performance or the algorithm[21]

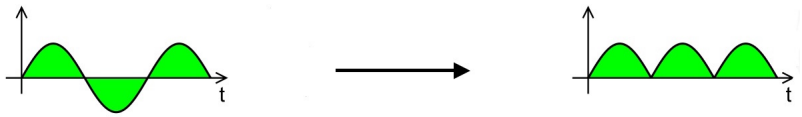


Figure 10: Visual representation of Full Wave Rectification (diagram adapted from https://en.wikipedia.org/wiki/Rectifier)

After these steps each frequency band is normalised through a method of mean removal in order to ensure the signal is centred at zero for the autocorrelation stage. The autocorrelation function is then applied to each frequency band and its peaks correspond to the various periodicities of that signal's envelope. The first five peaks of the function and their corresponding periodicities are then calculated in beats per minute and added to a histogram, this process is repeated while iterating over the signal. The estimated tempo of the audio signal is then retrieved from the periodicity that corresponds to the highest peak within the histogram[15].

Originally, it was intended that Discrete Wavelet Transform beat detection component of the RTT_Analyser would be implemented by the author. It was recognised in the project proposal that this could result in the rest of the project being delayed and therefore a mitigation to adapt the Matlab implementation of the Tzanetakis *et al*[15] algorithm by Eng Eder de Souza[32]. Due to the time spent in investigating how to adapt the Beatroot system to work with live audio a different solution was needed, this was provided by the Scala implementation by Marco Ziccardi[33].

2.3 Performance Worm

The final beat detection algorithm employed in the RTT_Analyser is the Performance Worm (PW) system, designed by Simon Dixon, Werner Goebl and Gerhard Widmer[24]. The PW is based on

[9] define an octave

For each new onset
    For IOI times t from 100ms to 2500ms in 10ms steps
        Find pairs of onsets which are t apart
        Sum the mean amplitude of these onset pairs
    Loop until all IOI time points are used
        For times t from 100ms to 2500ms in 10ms steps
            Calculate window size s as function of t
            Find average amplitude of IOIs in window $[t, t + s]$
            Store t which gives maximum average amplitude
        Create a cluster containing the stored maximum window
        Mark the IOIs in the cluster as used
    For each cluster
        Find related clusters (multiples or divisors)
        Combine related clusters using weighted average
    Match combined clusters to tempo hypotheses and update

Figure 11: Clustering algorithm used in the Performance Worm Multiple Tempo Tracking Subsystem[24]

a real time algorithm which is able to determine the tempo of the input raw audio, while keeping track of other possible tempo hypotheses that are rated and updated dynamically. Allowing for the most recently highest ranked tempo hypothesis to be returned to the user[24].

First the PW processes the raw audio which can either taken from a static recording or directly from a live input with a smoothing filter in order to obtain the RMS amplitude of the signal taken from a 40 ms window (explain RMS and smoothing filter). The note onsets are then calculated by the event detection module that finds the slope of the smoothed amplitude and then calculates the set of local peaks which are taken as the note onset times[24].

The signal is then processed by the multiple tempo tracking subsystem which uses a similar approach to the beatroot system. The time intervals (inter-onset intervals or IOIs) of event pairs are first calculated. A clustering algorithm (Figure 11) is then applied in order to determine the significant clusters of IOIs, which are subsequently assumed to be the musical units held within the signal. As in the beatroot system, these clusters are then used as the bases of the tempo hypotheses produced by the tempo tracking subsystem. While running the clustering algorithm keeps 5 seconds of onset times within memory and begins processing by determining all IOIs between the onsets in memory. The tempo inducer then exploits a property of most Western music where time intervals are related by small approximate integer ratios. As the times held by the clusters can be considered to represent related notes, e.g. quarter and eighth notes. The tempo inducer then adjusts cluster times and weightings according to the information held by the sets of related clusters. Two clusters are considered to be related if the ratio of their time intervals is close to an interger. The highest weighted clusters are their respective tempo hypothesis is then returned as the tempo output[24].

3 Solution Design and Architecture

The basic premise for the RTT_Analyser is to enable the user to play a live drum beat through the system and the tempo of the live audio is returned to user as well as being stored for future analysis. Initially, the RTT_Analyser opens the inbuilt microphone of the device upon which the software is being run. After establishing the live audio stream the RTT_Analyser beat detection

algorithms are sent the live audio data in the format of a byte array. These live audio bytes are then decoded according to the individual algorithm's requirements before being processed and the tempo calculated. A general schematic of the work-flow of the RTT_Analyser can be seen in Figure (add in).

3.1 Live Audio Processing

The live audio is processed using the Javax Sound package. In order to make the results reflective of the quality of microphone that any future applications might use it was decided that the RTT_Analyser was designed to be used with the built in to the device it was being executed on. In order to ensure the captured audio matches CD quality the Javax Sound AudioFormat class was designed with the following constructors:

- Encoding - This will be set to "PCM.signed", representing audio encoded to the native linear pulse code modulation, where pulse code modulation is the process of sampling and quantising the signal into discrete symbols for transmissions[25].
- Sample Rate - 44,100, set to match CD quality for the number of analog samples which will be analysed per second.
- Sample Size in Bits - 24, based on a sound card with a 24 bit sample depth.
- Channels - 2, audio will be captured using the built in microphone, typically stereo of the device the RTT_Analyser is being run on.
- Frame Size - 6, where the frame size is the number of bytes in a sample multiplied by the number of channels [17].
- Frame Rate - 44,100, same as sample rate.
- Big Endian (boolean) - false, as the project will be developed on an Intel core which uses a little-endian architecture[10].

The described audio format is then applied to the TargetDataLine, which provides access to the built in microphone of the device, if available. The TargetDataLine class is then attached to an instance of the Java AudioInputStream class which converts the captured audio into a stream of bytes for processing[26]. In order to process the captured bytes the RTT_Analyser then required the use of a concurrent system to allow for each of the three beat detection algorithms to be run in parallel, which was provided by the Akka Actors system.

3.2 Akka Actors

The Akka Actor Model is specifically designed to provide the ability to write concurrent systems with a high level of abstraction that alleviates the developer from the need to handle locking and individual thread management. Therefore, making it much easier to write concurrent and parallel systems when compared to the traditional approaches used in Java[27].

A fundamental construct of the Akka Actor system is that it strictly adheres too the Reactive Manifesto, which aims to ensure applications built under this are easier to develop and are amenable

[10]Endianess refers to the order of bytes which make up a digital word. Big endianess stores the most significant byte at a certain memory address and the remaining bytes being stored in the following higher memory addresses. The little-endian formate reverses the order storing the least significant at the lowest and most significant at the highest memory address [16].

to change, allowing for a higher level of tolerance to failure and the ability to meet any such failure elegantly as opposed to disaster[28]. The Reactive Manifesto requires applications to satisfy one or more of the following requirements[28]:

Responsive - The system responds in a timely manner, if possible. By providing usability and utility responsiveness allows for any problems to be detected and resolved effectively.

Resilient - The system should be able to remain responsive even in the event of a failure, which applies to all parts of the system not just the mission critical components. To achieve this each component needs to be isolated within the system, therefore allowing failed parts of the system to recover without effecting the responsiveness of the system as a whole.

Elastic - The system will be able to maintain the same level of responsiveness despite varying workloads, reacting to changes in the rate of input by adapting the levels of resources allocated to service these inputs accordingly.

Message Driven - Reactive systems rely on the use of asynchronous message passing in order to establish boundaries between components within the system, ensuring isolation, loose-coupling[11] and location transparency[12]. By utilising a non-blocking[13] message-passing it is possible to ensure message recipients only consume resources when active, leading to a much reduced system overhead.

3.2.1 The Actors

The actors within the actor model are objects which are used to encapsulate state and behaviour. They communicate exclusively through messages passed to a recipients mailbox and it can help to in fact think of them as group of people being assigned subtasks within an organisational structure. One of the key features of an actor system is that tasks are split up and delegated until they become small enough to be handled in one piece. This process not only ensures that the tasks being carried out by the actors are clearly defined but also the actors themselves can be designed in terms of the messages they are able to receive and how they should react to these messages and any failures that might occur. The actors within a system will naturally form a set hierarchy. For example, an actor system where an actor assigned the task of overseeing a certain function might split this function into a number of smaller tasks which it assigns some child actors that it supervises until the task is complete [29]. The actors created within the RTT_Analyser were a mix of stand alone actors and actors which were required to supervise subtasks carried out by child actors.

3.3 Design Methods

The design patterns used b

[11] add one
[12] add another
[13] another!

4 Implementation

The RTT_Analyser is written in Java and Scala, due to the Beatroot and Performance Worm both being written in Java, and the the author's familiarity with both languages and

the Scala specific build tool, sbt (simple build too), which is a build tool that creates a stable build platform that increases productivity by utilising some good ideas from other build tools like, minimal configuration and built-in tasks including test, compile and publish. It is also able to support a reactive development environment that is able to re-run all tests when source code is updated[30]. In terms of this project sbt was chosen as it supports mixed Scala/Java projects. Allowing for the Beatroot and Performance Worm systems written in Java to be easily incorporated with the Scala implementation of the Discrete Wavelet Transform and the Akka Actor system.

4.1 Live Audio Capture

To implement the RTT_Analyser's live audio capture system the Javax Sound package was used. Initially, the SoundCaptureImpl class was created in Java to be responsible for the live audio capture and holding of the raw audio in a circular buffer before being processed by the respective beat detection algorithms. The sound capturing components of this class were based on those described in the Java Sound Programmer Guide[31] and consisted of the code below:

```
private AudioFormat format;
private TargetDataLine input;
private DataLine dataLine;
...
    input = AudioSystem.getTargetDataLine(format);
    DataLine.Info info = new DataLine.Info(TargetDataLine.class, format);
    input = (TargetDataLine) AudioSystem.getLine(info);
    input.open(format);
```

The need for the above code however became defunct with the design decision to include the Performance Worm system within the RTT_Analyser as the Performance Worm was already set up to capture live audio. Therefore the captured audio now needed to be stored by the Performance Worm in a manner which allowed for all three beat detection algorithms to be able to process the same captured audio. This was achieved by adding the method *addBytes* (Figure 12) to the AudioWorm class, responsible for the capturing and processing of the live audio within the Performance Worm. The call to the *addBytes* was added just below the *.read* call of the AudioInputStream, the byte array was then passed to the *addBytes* method and stored within a ByteArrayOutputStream to be processed later. The difference in processing windows of the Performance Worm to initally th

4.2 DWT Beat Dectection Implementation

The implementation of Tzanetakis *et al* beat detection algorithm by Marco Ziccardi[33] was part of a larger system which offered a number of other beat detection methods. The RTT_Analyser only required the class which implemented the DWT beat detection algorithm, WaveletBPMDetector.

NEED SOME GLUE

```
public void addBytes(byte[] data){
    try{
        if(bytePosition == 525672){
            outputStream.write(data);
            byte[] out = outputStream.toByteArray();
            outputStream.close();
            outputStream = new ByteArrayOutputStream();
            bytePosition = 0;
            processingActor.tell(new ProcessBytes(out), processingActor);

        }
        outputStream.write(data);
    } catch (IOException e) {
        e.printStackTrace();
    }
}
```

Figure 12

The WaveletBPMDetector originally relied on a Scala version of the WavFile Java class created by Andrew Greensted[34], however as the RTT_Analyser was not required to decode WAV files this class was adapted to form the LiveAudioProcessor class (Figure 13). The class was based on three methods from Andrew Greensted's WavFile class[34], *readFrames*, an overloaded *readFrames* and the *getSample* method. Rather than being simply a Scala version of the Jave code, where possible these methods were written using Scala's pattern matching facility, which is considered to create "code that is both succinct and obviously correct"[35]. Where pattern matching is employed it is signified by the match keyword. Additionally, the Scala compiler also employs an optimiser which ensures that there are not any runtime overheads to be paid for using tail recursive[14] solutions. Writing tail recursive code often provides more elegant and concise solutions than when compared with loop-based solutions[36], therefore any loops in the WavFile class were converted into a tail recursive solution. The final addition to the LiveAudioProcessor class was the inclusion of the *addData* method that takes a byte array containing the raw live audio bytes to be processed and assigns them to the buffer variable which is processed in the *getSample* method. In order for the WaveletBPMDetector class to return a tempo result the a single window (matched to the size of the buffer byte array), the for loop was removed from the *bpm* method.

4.3 Tempo, Analyser, Stats and JSON

In order for the RTT_Analyser to be able to process the calculated tempo results the Tempo case class was implemented, comprising of the calculated and expected tempos, and difference between these values, an Option count of beats detected and the time at which the tempo result was calculated. For each tempo result calculates a Tempo object was created and added to a ListBuffer, chosen for its scalability, within the Analyser implementation corresponding to the relevant beat detection algorithm. The Analyser implementations also contained an Option Stats value, which enabled the Stats to be added to the Analyser after the instantiation of the case class. The Stats case class is used to hold the average and median values of the calculated tempos, as well as the difference between the expected and calculated tempo. To asses the efficiency of the algorithms the time taken to calculate a tempo within plus or minus one bpm of the expected value. The StatsCalculator case class carried out computation these which utilised Scala's pattern matching on the List data structure and .. MORE?!

The implementation of the data storage system of the RTT_Analyser used the JavaScript Object Notation (JSON) as the data generated was considered not large enough to warrant the creation and management of an SQL database. JSON's lightweight data-interchange format[38] ensured the the conversion from Scala case class to JSON object would be straightforward. The JSON was

[14] add one

```
def addData(data: Array[Byte]) = {
  buffer = data
}

def readFrames(sampleBuffer: Array[Int], numberOfFrames: Int): Int = {
  readFrames(sampleBuffer, 0, numberOfFrames)
}

def readFrames(sampleBuffer: Array[Int], offset: Int, numberOfFrames: Int): Int = {
  var pointer = offset

  for(i <- 0 until numberOfFrames){
    getSample(0)
    frameCounter = frameCounter + 1
  }

  def getSample(acc: Int): Int = {
    acc match{
      case x if x < numberOfChannels => {
        sampleBuffer(pointer) = readSample().toInt
        val newAcc = acc + 1
        pointer = pointer + 1
        getSample(newAcc)
      }
      case _ => acc
    }
  }
  numberOfFrames
}

def readSample(): Long = {

  @tailrec
  def sampleReader(value: Long, acc: Int): Long = {
    acc match {
      case x if x < bytesPerSample => {
        var v: Int = buffer(bufferPointer)
        if (acc < bytesPerSample - 1 || bytesPerSample == 1) v = v & 0xFF
        bufferPointer = bufferPointer + 1
        sampleReader(value + (v << (acc * 8)), acc + 1)
      }
      case x if x == bytesPerSample => value
      case _ => value
    }
  }

  sampleReader(0L, 0)
}
```

Figure 13: LiveAudioProcessor

generated the ScalaJSON library provided by the play framework[37], specifically the *Writes* object which converts an object into a JSON representation held encapsulated within a *JsValue* object. Three methods were written in order to account for the three implementations of the Analyser trait, an example of the code can be seen in Figure(add one). As JSON is written in a text format[38], the parsing of generated JSON was performed by simpling converting the JSON objects generated by the respective *writes* methods to a string which was then written to a JSON file using the *flush* method in Figureadd one, where the path was provided from the user interaction with the RTT_Analyser's user interface.

4.4 Adaption of Beatroot

Once the development of the RTT_Analyser was considered to be sufficiently on schedule the design decision was taken to attempt the adaption of the Beatroot system to work with live audio again. Much like with the WaveletBPMDetector, Beatroot would have to be set up to use the same audio bytes as those captured by the Performance Worm, to ensure all systems were using the exact same captured audio. As the *AudioWorm* was already set up to capture the enough bytes to work with

the WaveletBPMDetector's smallest working window size of 131072, the same figure was chosen to be processed by Beatroot.

With the correct design decisions now taken the live audio adaption of Beatroot consisted of only a few minor steps. First, the *processFile* method was amended to receive the byte array holding the audio data recorded by the Performance Worm. A ByteArrayInputStream was then instantiated to hold the data, allowing the *processFrame* and *getFrame* methods to work in the same manner as with a upload static audio file. After the captured audio was processed the ByteArrayInputStream was closed, a step necessary to ensure that no overlap existed between captured audio data sets sent to be processed.

4.5 The Actor System

The actor system employed within the RTT_Analyser is constructed of five different actors in total. The LiveAudioActor assumes the role of the head parent of the system, being responsible for starting and stopping the live audio capture, as well as the termination of the RTT_Analyser when appropriate.

The ProcessingActor, a child of the LiveAudioActor can be considered the coordinator of the storing and processing captured audio data, and the writing of the calculated tempos. In order to ensure the processing remains as isolated as possible the ProcessingActor employs two worker actors, BeatrootActor and DWTActor, carrying out the two tasks of sending the captured data and tempo calculation. The beat detection worker actors do not return any messages to the ProcessingActor, instead the immutable ActorRef handle which allows for the current instance of the respective actor is passed the respective beat detection system. Therefore, the calculated tempo values are then sent back directly to the ProcessingActor to be stored and subsequently saved to file when appropriate.

The final RTT_Analyser actor was a standalone actor on the same level as the LiveAudioActor within the actor hierarchy which was responsible for the timing of the beat detection test.

All of the RTT_Analyser's parent and child actors were all instantiated in the self contained Operator object. This ensured that potentially dangerous practice of declaring one actor and breaking actor encapsulation[27] was avoided.

4.6 RTT_Analyser User Interface and HTML Results Viewer

The RTT_Analyser's user interface was implemented using the ScalaFX, which sits on the top of the JavaFX API and uses the same declaration syntax as normal objects within Scala. Enabling developers to use the same operators and syntax to create and modify the scene graph[15][40]. It was designed with simplicity in mind. Before starting the user could begin live audio capture the expected bpm, file name and path were required. Once input, there are two modes available to the user. The first is to control the live audio beat detection manually using the start and stop buttons accordingly. The second, is the test mode which will runs the audio capture and beat detection for thirty seconds, before closing down audio inputs and storing the results. During execution the RTT_Analyser in both of these modes the three calculated tempos are displayed to the user as a bpm value rounded to two decimal places. On completion within both modes the results are stored within a JSON file automatically and also written to an HTML file so the results can be viewed in a web browser using the *Results* button.

[15] add one

The HTML tables were produced HtmlWriter object which used reflection where possible to obtain the relevant case class constructor field names, before using a number of tail recursive methods to write the field values and HTML tags to a StringBuilder, chosen for it's mutability. The HTML tables are subsequently saved to an HTML file which is opened and viewed via the default browser using the *Results* button.

5 Software Testing

During the development of the RTT_Analyser, Test Driven Development (TDD) was employed where possible. TDD is a development process which employs test-first development and refactoring. Test-first development involves writing a test before just enough production code is written to pass that test[41]. Refactoring is a process of making small changes to code, is used to improve the design of the code, thus making it easier to understand and modify[42].

The tests were written using the ScalaTest framework, which is one of the most dominant testing frameworks currently available. ScalaTest is designed with the intention of providing a test framework that makes testing more concise and uses the Scala language in a way to ensure testing is easy and fun[43].

TDD was used at any stage of the development process where a piece of code with an expected behaviour was being implemented, it was not used for all classes as when dealing with live audio it was not possible to produce a full suite of tests for all of the functionality. Although, where the requirement was simply to return a value of a certain type, basic TDD tests were written. The classes that were ideally suited to TDD and were developed completely using it were the Tempo, Analyser implementations, StatsCalculator, JSONParser and HTMLWriter.

The best example of TDD being applied during the development of the RTT_Analyser was during the creation of the StatsCalculator. Designed to carry out average and median calculations on the stored Tempo objects held within the Analyser implementations enabled TDD to be employed successfully.

When it was not possible to use TDD, particularly during the initial set-up and testing of the beat detection algorithms with live audio. A simpler testing approach was taken, consisting of running the system with a selection of sample drum beats of varying tempos, with console logs used to test that the system was performing the desired tempo calculations.

6 Drum Beat Sample Set and RTT_Analyser Beat Detection Testing

Once the RTT_Analyser was working with the initial test sample drum beats the full testing cycle could begin. In order to carry out this testing a set of drum beats were written using the Apple sequencer program, GarageBand[44]. The writing of the drum beats took a slight deviation from the project proposal in that rather than create multiple full drum beats in a varying number of styles, the drum beats were instead tested in layers. For example, a simple four four (4/4) bass drum beat with a back beat and the time element being played on the hi-hat cymbal would account for three samples. The reason for this decision stemmed from how a cymbal behaves when it is played at high amplitudes (loudness), the vibrations it create become chaotic and its clearly identifiable signals disappear and it effectively becomes noise[45]. Therefore in order to compare the drum sample with cymbals a baseline was required. Additionally, this change to the project proposal

was considered to be able to provide a much more diverse data set which would hopefully help the understanding of how the different elements of a drum kit, not just cymbals, can affect the accuracy of the beat detection algorithms being tested by the RTT_Analyser.

In total number twenty nine drum samples were produced and the full details of the elements which each drum beat was comprised of and style of the drum beat can be seen in Table 1.

The full beat detection tests were carried out in an environment with as little background noise as possible, where the drum beats were played on an external speaker set to the same volume for all tests. Each drum beat was tested for thirty seconds over the tempo range of 60-160 bpm, the tempo was increased in five bpm increments. Resulting in twenty one tests for being carried out for each drum beat and producing a sample set consisting of six hundred and nine (609) results files.

Table 1: List of Drum Beats Used in RTT_Analyser Tests

Drum Beat Code	**Elements**	**Style**
AMPEDUP	All	Heavy rock, GarageBand stock drum beat
HT	High tom-tom	Straight
HTLT	High tom-tom and low tom-tom	Straight
HTLT2	High tom-tom and low tom-tom	Straight
K	Bass drum	Straight
KS	Bass drum and snare drum	Straight
KS2	Bass drum and snare drum	Straight
KSCR	Bass drum, snare drum and crash cymbal	Straight
KSFTTF	Bass drum (four to the floor) and snare drum	Straight
KSH	Bass drum, snare drum and hi-hat cymbal	Straight
KSH2	Bass drum, snare drum and hi-hat cymbal	Straight
KSH2CR	Bass drum, snare drum, hi-hat and crash cymbals	Straight
KSHCR	Bass drum, snare drum, hi-hat and crash cymbals	Straight
KSHHTLT	Bass drum, snare drum, hi-hat cymbal, high tom-tom and low tom-tom	Straight
LT	Low tom-tom	Straight
MTREVIS	All	Mowtown swing, GarageBand stock drum beat
OFF-KSH	Bass drum, snare drum and hi-hat cymbal	Offbeat played on hi-hat
OFF-KSHCRFTTF	Bass drum (four to the floor), snare drum, hi-hat and crash cymbals	Offbeat played on hi-hat
OFF-KSHFTTF	Bass drum (four to the floor), snare drum, hi-hat cymbal	Offbeat played on hi-hat
OFF-KSHHCRFTTF	Bass drum (four to the floor), snare drum, hi-hat and crash cymbals	Offbeat played on hi-hat
S	Snare drum	Straight
SMASH	All	Punk rock (loud), GarageBand stock drum beat
SW-H	Hi-hat cymbal	Swing
SW-K	Bass drum	Swing
SW-KH	Bass drum and hi-hat cymbal	Swing
SW-KS	Bass drum and snare drum	Swing
SW-KSH	Bass drum, snare drum and hi-hat cymbal	Swing
SW-R	Ride cymbal	Swing
SW-SH	Snare drum and hi-hat cymbal	Swing

References

[1] *http://www.instructables.com/id/What-is-MIDI/*

[2] Alison Latham
The Oxford Companion to Music, 2002, Oxford University Press

[3] *https://www.britannica.com/art/metronome*

[4] Mick Berry and Jason Gianni
The Drummer's Bible: How to Play Every Drum Style from Afro-Cuban to Zydeco, Second Edition, 2004, See Sharp Press

[5] *http://www.drummagazine.com/lessons/post/drumkey/*

[6] Allen and Dannenberg
Tracking Musical Beats in Real Time, International Computer Music Conference, International Computer Music Association, 1990, pp. 140-143

[7] H. C Longuet-Higgins
Perception of melodies, Nature Vol. 263, 1976, pp. 646-653

[8] Simon Dixon
Automatic Extraction of Tempo and Beat from Expressive Performances. Journal of New Music Research, 30 (1), 2001, pp 39-58

[9] Simon Dixon
Onset Detection Revisited, Proceedings of the 9th International Conference on Digital Audio Effects, Montreal, September 2006, pp 133-137

[10] Simon Dixon
On the Analysis of Musical Expression in Audio Signals. Storage and Retrieval for Media Databases, SPIE-IS&T Electronic Imaging, SPIE Vol. 5021, 2003 pp 122-132

[11] *http://www.music-ir.org/mirex/wiki/2016:Audio_Onset_Detection*

[12] *http://www.ieor.berkeley.edu/ ieor170/sp15/files/Intro-to_Sonic_Events_Campion.pdf*

[13] Juan Pablo Bello, Laurent Daudet, Samer Abdallah, Chris Duxbury, Mike Davies, and Mark B. Sandler *A Tutorial on Onset Detection in Music Signals, IEEE TRANSACTIONS ON SPEECH AND AUDIO PROCESSING, VOL. 13, NO. 5, 2005, pp. 1035 - 1047*

[14] *http://www.music-ir.org/mirex/wiki/2005:Main_Page*

[15] George Tzanetakis, Georg Essl and Perry Cook
Audio Analysis using the Discrete Wavelet Transform, Proc. WSES International Conference on Acoustics and Music: Theory and Applications (AMTA), 2001

[16] *http://www.music-ir.org/mirex/wiki/2006:Audio_Beat_Tracking_Results*

[17] Simon Dixon
Evaluation of the Audio Beat Tracking System BeatRoot. Journal of New Music Research, 36, 1, 2007, pp 39-50

[18] *http://users.rowan.edu/ polikar/WAVELETS/WTpart2.html*

[19] Lyons
Understanding Digital Signal Processing

[20] Tao Li, Mitsunori Ogihara, George Tzanetakis
Music Data Mining, CRC Press, 2002, pp 45-53

[21] George Tzanetakis and Perry Cook
Musical Genre Classification of Audio Signals

[22] Steven W. Smith
The Scientist and Engineer's Guide to Digital Signal Processing, California Technical Publishing, 2011, Chapter 3

[23] Ramn Palls-Areny and John G. Webster
Analog Signal Processing, Wiley, 199, pp. 231

[24] Simon Dixon, Werner Goebl and Gerhard Widmer
The Performance Worm: Real Time Visualisation of Expression based on Langners Tempo-Loudness Animation, International Computer Music Conference, 16 - 21 September 2002, Gteborg, Sweden, pp 361-364.

[25] Bill Waggener
Pulse Code Modulation Techniques

[26] Oracle
https://docs.oracle.com/javase/tutorial/sound/

[27] Akka
http://doc.akka.io/docs/akka/2.4.9/scala/actors.html

[28] Jonas Bonr, Dave Farley, Roland Kuhn, and Martin Thompson
Reactive Manifesto, http://www.reactivemanifesto.org/

[29] Akka
http://doc.akka.io/docs/akka/2.4.9/general/actor-systems.html

[30] Josh Suereth and Matthew Farwell
SBT in Action: The simple Scala build tool, Manning Publications, 2015

[31] Oracle
https://docs.oracle.com/javase/8/docs/technotes/guides/sound/programmer_guide/contents.html

[32] *https://github.com/ederwander/Beat-Track*

[33] *https://github.com/mziccard/scala-audio-file*

[34] Andrew Greensted *http://www.labbookpages.co.uk/audio/javaWavFiles.html*

[35] Marius Eriksen
Effective Scala, http://twitter.github.io/effectivescala/

[36] Martin Odersky, Lex Spoon and Bill Venners
Programming in Scala, Second Edition, Artima Press, 2010

[37] *https://www.playframework.com/documentation/2.5.x/ScalaJson*

[38] *http://www.json.org/*

[39] *https://www.playframework.com/documentation/2.5.x/api/scala/index.html#play.api.libs.json.Writes*

[40] *http://www.scalafx.org/docs/home/*

[41] *http://www.agiledata.org/essays/tdd.html*

[42] *http://www.agiledata.org/essays/databaseRefactoring.html*

[43] Daniel Hinojosa *Testing in Scala, OReilly Media, 2013*

[44] *http://www.apple.com/uk/mac/garageband/*

[45] *http://www.soundonsound.com/sos/may02/articles/synthsecrets0502.asp*