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

Abstract

This project report presents an invetigation into the suitability of three beat detection algorithms to be used in a drumming training tool. The system captures live audio and passes it to the beat detection algorithms for tempo calcualtion. The result of which is then passed back to the user.

An extensive sample set of drum beats were created in order to replicate a drummers performance and processed through the system. The tempo calculation results were recorded by the system and then analysed in the attempt to draw conclusions regarding, if any of the three beat detection algorithms could form the basis for a future training tool.

Contents

Abbreviations

BPM	Beats Per Minute
DFT	Discrete Fourier Transform
DWT	Discrete Wavelet Transform
JSON	JavaScript Object Notification
PW	Performance Worm
SBT	Simple Build Tool
STFT	Short Time Fourier Transform
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 is 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 details the aim to develop a real-time drum beat tempo analysis system. The system will use different beat detection algorithms and be 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 are judged. For many years the only training tool available to a drummer to improve their timing was the metronome, which is an instrument used to mark musical tempo. The metronome was erroneously attributed to Johann Nepomuk Maelzel in 1815 but was 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] and is still used today by musicians, as seen in Figure 1.

Figure 1: Traditional Metronome

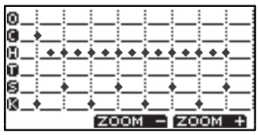
Today drummers commonly use electronic versions of the metronome are much more widely used, which now developed with functions specifically tailored to a drummers training requirements. The Tama Rhythm Watch (Figure 2) was the first metronome designed specifically for drummers, providing enough volume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

Following the development of the MIDI driven electronic drum kit came the development of more advanced training tools, able to provide live feedback to a 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 their drum modules. The SCOPE system provides live feedback to a drummer by plotting their performance on a grid representing the elements of the drum kit, Figure 3a.

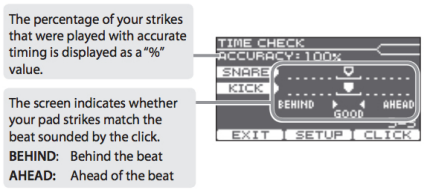
[1] A details explanation of time signatures can be found in section 1.2.3

Figure 2: Tama Rhythm Watch

The COACH system offers an improved user experience by providing an accuracy figure relating to the number of hits that the drummer has played in time, Figure 3b.



(a) Screen shot of the SCOPE system



(b) Screen shot of the COACH system

Figure 3: SCOPE and COACH training tools

The Roland catelogue also includes the DT-1 tutor software package, which provides drummers with an interactive experience helping to improve their rhythm, coordination and sight reading (Figure 4).

Roland have even started to design games within this field. Their latest release, the V-Drums Friend Jam app, provides the player with live feedback and evaluates each performance to provide the player with a score, which can shared over social media (see Figure 5).

Despite these developments 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. Firstly, the concept of a drummer playing time must be considered. Time, in a

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

Figure 4: Screen shot of the DT-1 V-Drums Tutor

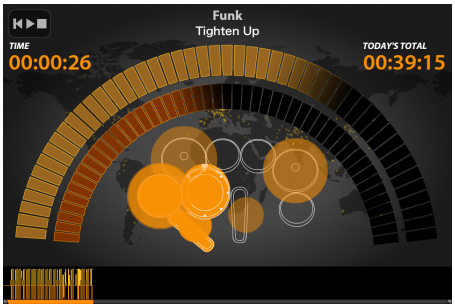


Figure 5: Example of the Roland Friend Jam software package

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, which is made up of five individual lines (Figure 6). The clef found on the far left of the staff 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 them represent a tone to be played. However, as drums are not considered tonal, the notes written on the staff 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. A bar of drum music typically consists of different notes representing which instruments to be played and when. The number of notes within a bar

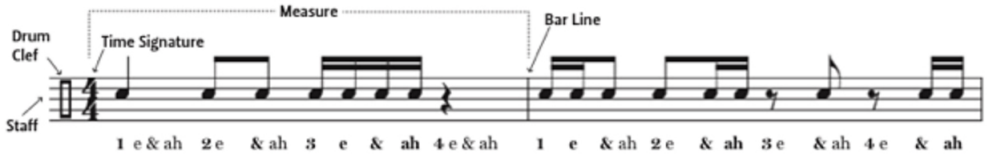


Figure 6: Example of a staff used in musical notation

may also vary depending on their length.

1.2.2 Notes

The notes used in drum notation not only represent the percussive instrument to be played but also 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 are 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]. For the purpose of this project it is the count of these beats that are used to calculate the tempo of a certain drum beat.

1.2.3 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.

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 is played on the first and third beat and the snare drum is played on the second and fourth beat both as quarter notes. This is more commonly known as a back beat [4]. This just leaves the

time element which is usually 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 is 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. Playing time on the "and" note is also known as playing on the off beat.

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. Some of the first work in this field can be attributed to H. C. Longuet-Higgins. In 1976, whilst studying the psychological theory of how Western musicians perceive rhythmic and tonal relationships between musical notes, he produced an algorithm capable of following the beat of a performance and adjust the perceived tempo accordingly[6]. Longuet-Higgins' work was built on the premise that in order to perceive 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.

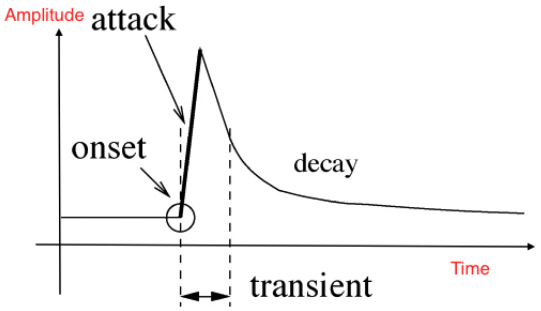


Figure 7: The onset of a note is the instant which marks start of the variation in the frequency of a signal (Image Source: [13])

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

[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

held. MIREX includes a contest with the goal of comparing state-of-the-art algorithms for music information retrieval [14] and the topics to be evaluated are proposed by the participants. In the first year, three of the nine topics concerned beat detection, including audio onset detection and since it's first inclusion it has been an evaluated 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 be used in a training tool for 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 the DWT algorithm described by Tzanetakis *et al* [15].
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

2 RTT_Analyser Beat Detection Algorithms

The original proposed solution incorporated two beat detection algorithms, the 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

[5]Where RTT stands for Real Time Tempo

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]. To ensure the project remained on track it was decided to substitute the Performance Worm for the Beatroot system. With the intention, once the project was back on schedule, to resume the adaptation of the Beatroot system to work with live audio. This was successfully completed prior to the development of the user interface.

The project report will now discuss the three beat detection algorithms incorporated in the RTT_Analyser.

2.1 Beatroot

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

Beatroot is comprised of two major components, the tempo induction and the beat tracking subsystems. Before tempo induction can take place, first a list of onsets needs to be retrieved[17]. This is achieved by obtaining a time-frequency representation with a Short Time Fourier Transform (STFT). The STFT is based on the Discrete Fourier Transform (DFT), which can be used to determine how much of a frequency exists within a signal. However, the DFT 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 into a number of smaller segments using a window function, which effectively creates a series of stationary[7] signals which the DFT can 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 containing information about how the energy held within the signal is distributed in the time and frequency domains [20].

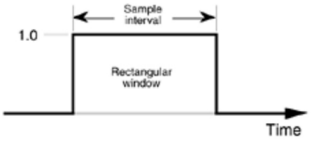
The window function used by Beatroot within the STFT is known as a Hamming window, which can help to reduce the amount of additional frequencies appearing in the returned DFT spectrum, known as spectral leakage. This is caused when the steep slope of a rectangular window (Figure 8a) causes the frequencies to become distorted. The Hamming window counteracts this by using a bell shape, which reduces the 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 8b.

The STFT used by Beatroot is defined as follows:

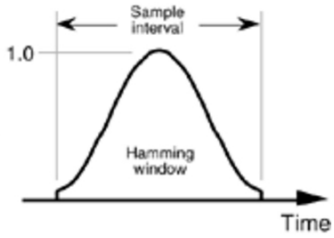
$$X(n,k) = \sum_{m=-\frac{N}{2}}^{\frac{N}{2}-1} x(hn+m)\ \ w(m)\ \ e^{-}\frac{2jmk}{N} \tag{1}$$

[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



(a) Rectangular window



(b) Hamming window

Figure 8: (Image Source: [19])

Where a Hamming window $w(m)$ is calculated at a frame rate of 100 Hz, N is the size of the window, and n represents the frame number being processed. The index of frequency-domain component is represented by k, and h is the hop-size or number of samples between the start times of adjacent frames.

The spectral flux function is then used as a method for measuring the change in magnitude of each returned frequency-domain sample [9] and is provided by:

$$SF(n) = \sum_{k=-\frac{N}{2}}^{\frac{N}{2}-1} H(|X(n,k)| - |X(n-1,k)|) \tag{2}$$

Where SF is the spectral flux and H is the half-wave rectifier function given by the $H = \frac{x+|x|}{2}$, which is used to remove any negative values representing a falling signal. An example of the peaks returned by a spectral flux function can be seen in Figure 9.

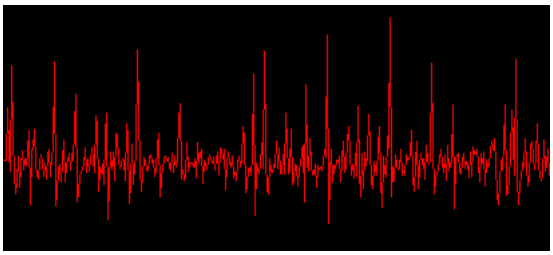


Figure 9: (Image Source: [57])

The onsets detected are then selected by a peak-picking algorithm which finds local maxima within the detection function. The tempo induction algorithm is then 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 10 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].

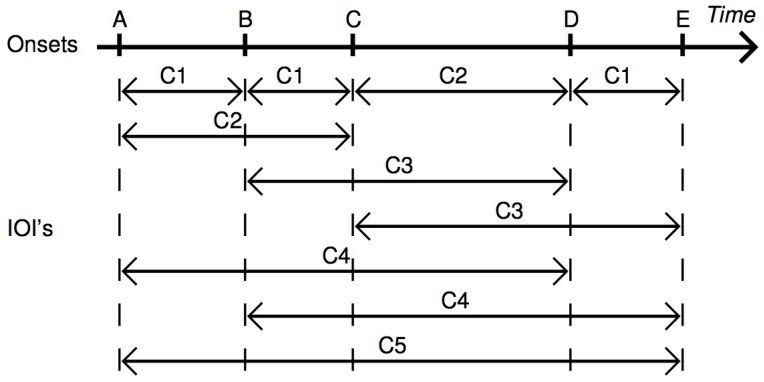


Figure 10: (Image Source: [17])

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, these sequences are rated and the most likely set of beat times is determined. Each of the agents are 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, those falling outside are not considered to be a beat time. The agents then rate themselves based on 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], which is then used to determine the overall tempo of the audio 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 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. Instead, the DWT uses a variable time-frequency resolution that allows for low frequencies to be detected precisely but not accurately placed in the time, and high frequencies to be accurately placed in time but not detected as precisely. The most common type of DWT can be viewed as a filter-bank, where each of the filters corresponds to a certain frequency range, usually representing an octave in frequency[9][20].

In 2001, Tzanetakis *et al* described how the DWT could be used to extract information

[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]

[9]Acoustically, an octave above a note is the note which has twice the frequency of the original [2].

from non-speech audio [15]. Their beat detection algorithm was based on detecting the most prominent signals which are repeated over a period of time within the analysed audio.

The first stage is to split the signal into a number of octave frequency bands using the DWT. This allows for the time domain amplitude envelopes of each frequency band to be extracted separately, using the following three steps. Where $x[n]$ is the input signal and $y[n]$ the output:

1. Full Wave Rectification:

$$y[n] = |x[n]| \tag{3}$$

   a process of converting the amplitude of each frequency band to one polarity [23], which can be either positive or negative. Performed in order to extract only the time-related envelope of the signal as opposed to the time domain signal itself [21].

2. Low Pass Filtering:

$$y[n] = (1 - \alpha)x[n] + \alpha y[n - 1] \tag{4}$$

   for a single frequency point where α has a value of 0.99. Designed to allow frequencies below a cutoff frequency through but blocks any frequencies above the cutoff frequency [22]. Used in this context as a way to smooth the signal within envelope [21].

3. Down-sampling:

$$y[n] = x[kn] \tag{5}$$

   where k was set to a value of 16. Due to the large periodicities in beat analysis, the signal is down-sampled in order to reduce the computation time of the autocorrelation stage without causing any negative effects on the performance of the algorithm [21].

Each frequency band is then normalised through a method of mean removal:

$$y[n] = x[n] - E[x[n]] \tag{6}$$

where E represents the expected mean value. Mean removal is applied to ensure the signal is centred at zero for the autocorrelation stage [21]. The autocorrelation function is then applied to each frequency band:

$$y[n] = \frac{1}{N} \sum_{n} x[n]x[-k] \tag{7}$$

where N represents the sample size. The resulting peaks produced by the autocorrelation function 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]. The DWT beat histogram flow diagram can be seen in Figure 11.

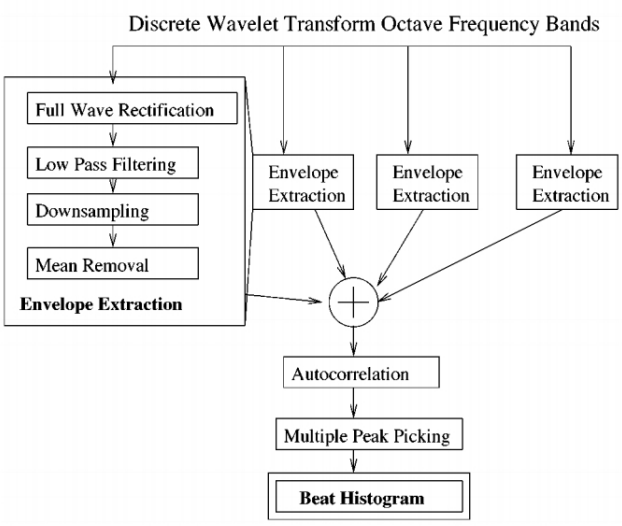


Figure 11: Flow diagram of the beat histogram method described by Tzanetakis et al [21]

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 a real time algorithm which is able to determine the tempo of input raw audio, while keeping track of other possible tempo hypotheses that are rated and updated dynamically. This allows 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 be extracted from a static recording or directly from a live input with a smoothing filter[10] in order to obtain the RMS amplitude[11] of the signal taken from a 40ms window. 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. A clustering algorithm (Figure 12), first calculates the time intervals (inter-onset intervals or IOIs) between the pairs of onsets, which it holds in memory (approximately five seconds of audio). The intervals are then weighted by the mean of the amplitudes and each interval's IOIs are clustered using the best-first algorithm (shown in Figure 12). The significant clusters of IOIs determined by the best-first algorithm 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

[10]Process which attempts to reduce any noise found within a signal. A smoothing filter does this by reducing any points in the signal that are higher than adjacent points and reducing those points that are lower than adjacent point [46]

[11]RMS stands for root mean square and is the process of squaring all of the amplitude values, then taking the average (mean) of the squared values, and then calculating the square root of the average value [47].

Inter-onset interval calculation
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

    Best-first algorithm
    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

    Cluster Combining
    For each cluster
        Find related clusters (multiples or divisors)
        Combine related clusters using weighted average
    Match combined clusters to tempo hypotheses and update

Figure 12: Clustering algorithm used in the Performance Worm Multiple Tempo Tracking Subsystem [24]

produced by the tempo tracking subsystem. 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 integer. The highest weighted clusters and their respective tempo hypothesis are 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, the tempo of the live audio is then returned to user while 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.

3.1 Live Audio Processing

The live audio is processed using the Java Sound API, which contains two key types; mixers and lines. Mixers are a representation of the audio devices available on a certain system and a line is an element of the digital audio pipeline responsible for moving audio in and out of the system. Typically, audio capture is started or stopped using a TargetDataLine that provides a path to the systems audio input device. Access to this device is provided by the AudioSystem class, which acts a clearing house for audio components. In order to ensure the bytes of audio data passed through the TargetDataLine are interpreted correctly the AudioFormat object enables the audio format to be defined by the list attributes below [31], which includes the values chosen to ensure that the audio capture by the RTT_Analyser is comparable to CD quality:

- Encoding - 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 analysed per second.
- Sample Size in Bits - 24, based on a sound card with a 24 bit sample depth.
- Channels - 2, audio is 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 [58].
- Frame Rate - 44,100, same as the sample rate.
- Big Endian (boolean) - false, as the machine used to develop the RTT_Analyser has Intel cores, which use a little-endian architecture[12].

In order to process the captured bytes the RTT_Analyser required a concurrent system capable of running the three beat detection algorithms in parallel. This 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 removes the need for the developer to handle locking and individual thread management. Making it much easier to write concurrent and parallel systems than with the traditional approaches used in Java [27].

[12]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 [51].

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 to change. This allows for a higher level of tolerance to failure and the ability to meet any such failure elegantly [28]. The Reactive Manifesto requires applications to satisfy one or more of the requirements listed below [28] and visualisation of how they interact can be seen in Figure 13.

Responsive - The system responds in a timely manner, if possible. By providing usability and utility responsiveness, it is possible to detect and resolve problems 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 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[13] and location transparency[14]. By utilising non-blocking[15] message-passing it is possible to ensure message recipients only consume resources when active, leading to a much reduced system overhead.

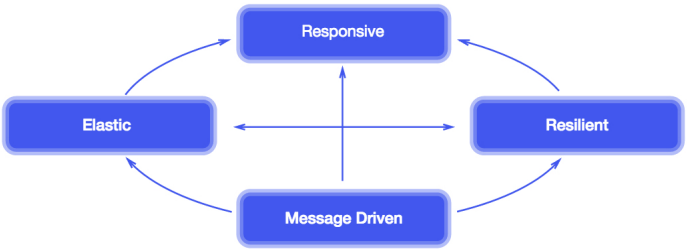


Figure 13: Visualisation of the requirements of the Reactive Manifesto interact [28]

[13]Loose coupling is a design approach of distributed systems which emphasises agility and ability to adapt to changes [48]

[14]The decoupling of the runtime instances from their references [28]

[15]In concurrent programming algorithms which do not have their execution indefinitely postponed when competing for a resource is said to be non-blocking [28].

3.2.1 The Actors

The actors within the actor model are objects that are used to encapsulate state and behaviour. They communicate exclusively through messages passed to a recipients mailbox and it can help to 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 ensures that the tasks being carried out by the actors are clearly defined. It also allows for the actors to 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 assigned the task of overseeing a certain function might split this function into a number of smaller tasks. It will then assign these tasks to some child actors, which it will 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 Pattern

The RTT_Analyser was developed using the Model View Controller (MVC) design pattern, as the MVC pattern is the tried-and-tested approach when designing applications with a graphical user interface [50]. The MVC pattern is made up of the following components and a visualisation of the MVC can be seen in Figure 14.

- **Model** - is represented by the data and associated applications [50], within the RTT_Analyser the model is provided by the beat detection algorithms, the live audio capturing and processing components.

- **View** - is the graphical interface displayed to the user which updates itself automatically [50]. For the RTT_Analyser the view is represented by the user interface

- **Controller** - is the part of the application that responds to the user interactions, liasing with both the model and the view components[50]. The controller for the RTT_Analyser is the actor system, which is responsible for message passing between the model and view components.

A high level class diagram of the final version of RTT_Analyser can be seen in Figure 15

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 author's familiarity with both languages. Using

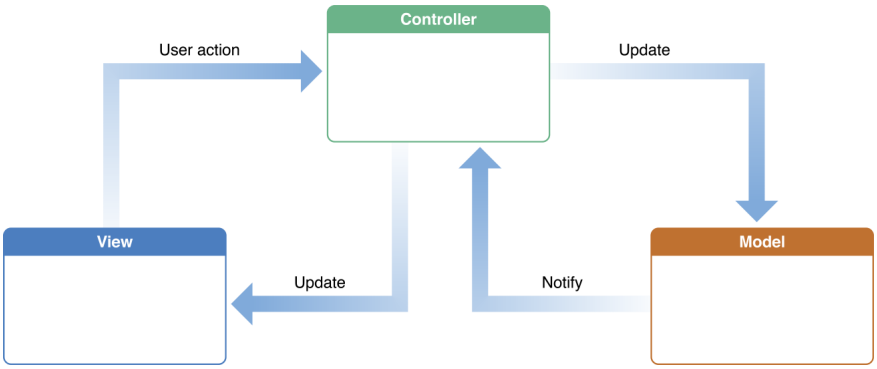


Figure 14: The three components which make up Model View Controller design pattern [49]

Figure 15: Image to be added

Scala also ensured the twin constructs of pattern matching and case classes were able to be utilised where appropriate. Pattern matching is used to implement type switches and is considered to create "code that is both succinct and obviously correct" [35]. Case classes are Scala's way of allowing pattern matching to be performed on objects without the need for additional boilerplate code. Using Scala, also allows for more elegant and concise tail recursive[16] solutions to be developed over loop-based solutions. As the Scala compiler provides an optimiser which ensures that there are not any runtime overheads to be paid for employing tail recursion [36].

The RTT_Analyser was developed in the Scala specific build tool, sbt (simple build too). The sbt is a build tool that creates a stable build platform that supports a reactive development environment able to re-run all tests when source code is updated [30]. In terms of this project sbt was chosen primarily for it's support of mixed Scala/Java projects. Allowing for the Java Beatroot and Performance Worm systems to be easily incorporated with the Scala implementation of the DWT beat detection algorithm and the Akka Actor system.

4.1 The Actor System

Although the actors were not the first element of the RTT_Analysert to be implemented, introducing them first helps to provide some context for the how the other components communicate.

The actor system is constructed of five different actors and a diagram of the actor hierachy can be seen in Figure 16.

The messages permitted to be passed between the RTT_Analyser's actors were all implemented as case classes within a sealed trait (Figure 17). A sealed trait ensures that the

[16] A recursive solution is considered to be tail recursive if the last operation carried out is to calculate the return value, it has the advantage over recursive calls of not building a stack trace and instead performing each operation in a single stack frame [36]

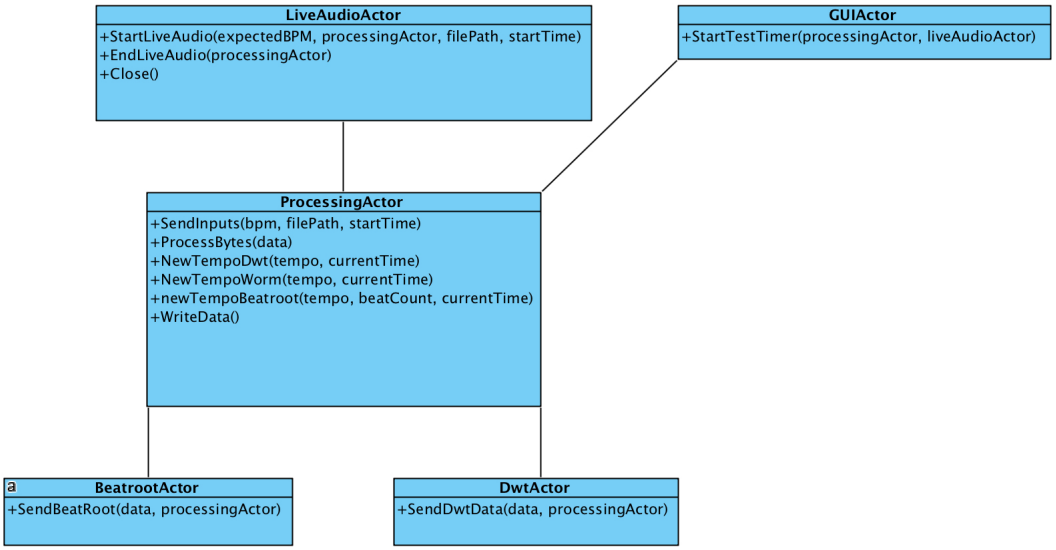


Figure 16: UML drawing of the RTT_Analyser's actor hierachy, operations within each actor represent the messages they can recieve

only classes extending the trait are in the same file [36], which works well with the actor's pattern matching recieve method.

```
sealed trait Messages
case class StartLiveAudio(expectedBPM: Double, processingActor: ActorRef, filePath: String, startTime: Long) extends Messages
case class EndLiveAudio(processingActor: ActorRef) extends Messages
case class ProcessBytes(data: Array[Byte]) extends Messages
case class NewTempoWorm(tempo: Double, currentTime: Long) extends Messages
case class NewTempoDwt(tempo: Double, currentTime: Long) extends Messages
case class NewTempoBeatroot(tempo: Double, beatCount: Double, currentTime: Long) extends Messages
case class WriteData() extends Messages
case class SendInputs(bpm: Double, filePath: String, startTime: Long) extends Messages
case class SendBeatRoot(data: Array[Byte], processingActor: ActorRef) extends Messages
case class SendDwt(data: Array[Byte], processingActor: ActorRef) extends Messages
case class Close() extends Messages
case class StartTestTimer(processingActor: ActorRef, liveAudioActor: ActorRef) extends Messages
case class Reset() extends Messages
```

Figure 17: The message case classes used by the RTT_Analyser's actor system

The LiveAudioActor (Figure 18) assumes the role of the overseer 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 (Figure 19), 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 task of iniating the tempo calculation. The beat detection worker actors do not return any messages to the ProcessingActor, instead a reference to the ProcessingActor, in the form of an ActorRef handle is passed to the respective beat detection system. The calculated tempo

```
class LiveAudioActor extends Actor with ActorLogging{

  val w: Worm = new Worm()

   def receive = {
     case StartLiveAudio(expectedBPM, processingActor, filePath, startTime) =>
       w.setActor(processingActor)
       w.play()
       processingActor ! SendInputs(expectedBPM, filePath, startTime)
     case EndLiveAudio(processingActor) =>
       w.stop()
       Thread.sleep(2000)//allow for any processing to finish
       processingActor ! WriteData
     case Close =>
       context.system.terminate()
       System.exit(0)
     case _ =>
       println("Error")
   }
}
```

Figure 18: LiveAudioActor class

values are then sent back directly to the ProcessingActor to be stored and subsequently saved to file when appropriate.

The final RTT_Analyser actor, the GUIActor, is a standalone actor on the same level as the LiveAudioActor within the actor hierarchy. the GUIActor is responsible for the timing of the beat detection test process.

All of the RTT_Analyser's parent and child actors were all instantiated in the self contained Operator object (Figure 20. This ensured that potentially dangerous practice of declaring one actor and breaking actor encapsulation [27] was avoided.

4.2 Live Audio Capture

Originally, the RTT_Analyser's live audio capture was handled by the SoundCaptureImpl class, where the raw audio was captured using the audio format described in section 3.1. The audio is then converted into a stream of bytes by the Java AudioInputStream [26], before being passed for processing by the 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 in Figure 21.

The need for the code in Figure 21 became defunct with the design decision to include the Performance Worm system within the RTT_Analyser. As the Performance Worm is already set up to capture live audio. The captured audio now needed to be stored by the PW 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 22) to the AudioWorm class, responsible for the capturing and processing of the live audio within the PW. The addBytes method took a byte array, which was stored within a ByteArrayOutputStream. Once the sufficient number of bytes had been amassed the ProcessingActor was sent the ProcessBytes message, which initiated the processing of the

```
def receive = {
  case SendInputs(bpm, filePath, startTime) =>
    expectedBpm(bpm)
    path(filePath)
    timeAtStart(startTime)
  case ProcessBytes(data) =>
    beatrootWorker ! SendBeatRoot(data, self)
    dwtWorker ! SendDwt(data, self)
  case NewTempoDwt(tempo, currentTime) =>
    val t = Tempo(tempo, expectedBpm, None, currentTime - timeAtStart)
    gui.updateDwt(tempo)
    dwtStatsBuffer += t
    dWtAnalyser.addTempo(t)
  case NewTempoWorm(tempo, currentTime) =>
    val t = Tempo(tempo, expectedBpm, None, currentTime - timeAtStart)
    if (count == 5) {
      gui.updateWorm(tempo)//worm element of gui updated 5 time a second
      count = 0
    }
    count = count + 1
    wormStatsuffer += t
    wormAnalyser.addTempo(t)
  case NewTempoBeatroot(tempo, beatCount, currentTime) =>
    val t = Tempo(tempo, expectedBpm, Some(beatCount), currentTime - timeAtStart)
    gui.updatebrt(tempo, beatCount)
    beatStatsBpmBuffer += t
    beatrootAnalyser.addTempo(t)
  case WriteData =>
    addStats()
    jsonParser.writeAll(wormAnalyser, dWtAnalyser, beatrootAnalyser)
    jsonParser.flushFull(path + "full.json")
    htmlWriter.writeHtml(List(wormAnalyser, dWtAnalyser, beatrootAnalyser))
    htmlWriter.flush(path + "stats.html")
}
```

Figure 19: Recieve method of the ProcessingActor class

```
val system = ActorSystem("liveAudioSystem")
val liveAudioActor = system.actorOf(Props[LiveAudioActor], "liveAudioActor")
val beatrootActor = system.actorOf(Props[BeatrootActor], "beatrootActor")
val dwtActor = system.actorOf(Props[DwtActor], "dwtActor")
val processingActor = system.actorOf(Props(new ProcessingActor(beatrootActor, dwtActor)))
val guiActor = system.actorOf(Props[GUIActor], "guiActor")
```

Figure 20: Declaring of the RTT_Analyser's actors

```
def startCapture: Int = {

  try {

    input = AudioSystem.getTargetDataLine(inputFormat)
    val info: DataLine.Info = new DataLine.Info(classOf[TargetDataLine], inputFormat)
    input = AudioSystem.getLine(info).asInstanceOf[TargetDataLine]
    input.open(inputFormat)
    outputStream = new ByteArrayOutputStream
    input.start()
```

Figure 21: Example of the basis of live audio capture method in Scala

audio data by the Beatroot and DWT beat detection systems.

```
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 22

4.3 DWT Beat Detection Implementation

The DWT beat detection component of the RTT_Analyser was originally proposed to be implemented by the author. However, due to the time spent 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].

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, which 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 23). 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 Java code, where possible these methods were written using Scala's pattern matching facility and any loops converted to a tail recursive solution.

The final addition to the LiveAudioProcessor class was the inclusion of the addData method, which assigns the byte array containing the raw audio to the buffer variable processed in the getSample method. In order for the WaveletBPMDetector class to return a tempo result for a single window of audio (matched to the size of the buffer byte array), the loop was removed from the bpm method. Once a tempo was calculated this was sent back to the ProcessingActor using the NewTempoDWT message.

4.4 Tempo, Analyser and Stats

The Tempo object, implemented as a method of encapsulating the RTT_Analyser's calculated tempo results. Is Comprised of the calculated and expected tempos, and difference

```
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 23: LiveAudioProcessor

between these values. Additonally, an Option[17] count of beats detected and the time at which the tempo result was calculated is included.

The Analyser object was designed to hold the calculated Tempo objects within a ListBuffer, chosen for its scalability. The Analyser trait was extended to create three case classes corresponding to the relevant beat detection algorithms. The Analyser implementations also contained an Option Stats value.

The Stats object is an object used to hold the average and median values of the calculated tempos, difference between the expected and calculated tempos. The time taken to calculate a tempo within plus or minus one bpm of the expected value was also recorded in order

[17]The Option type is Scala's alternative to Java's null value, the Option type is in fact a container which either holds a value or is empty. It provides a much safer alternative to the error prone null value [35].

to assess the efficiency of the beat detection algorithms. The StatsCalculator was a helper class responsible for carrying out computation of the values held with the Stats object and was implemented using the code seen in Figure 24.

4.5 JSON Parsing

The implementation of the data storage system of the RTT_Analyser used the JavaScript Object Notation (JSON). JSON's lightweight data-interchange format [38] ensured the conversion from a Scala case class to a JSON object is straightforward. The JSON is generated by the ScalaJSON library provided by the play framework [37]. Specifically the *Writes* object, which converts an object into a JSON representation. 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 25.

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 write methods to a string. This was then written to a JSON file where the path was provided from the user interaction with the RTT_Analyser's user interface.

4.6 Adaption of Beatroot

Once the development of the RTT_Analyser was considered to be sufficiently on schedule, the adaption of the Beatroot system to work with live audio was attempted again. As with the WaveletBPMDetector, Beatroot would have to be set up to use the same audio bytes as those captured by the PW. As the PW's AudioWorm class 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 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.

4.7 RTT_Analyser User Interface and HTML Results Viewer

The RTT_Analyser's user interface (Figure 26) was designed with simplicity in mind and was implemented using the ScalaFX[40]. Sitting on the top of the JavaFX API and using the same declaration syntax as normal objects within Scala, ScalaFX enables developers to use the same syntax to create and modify the user interface's scene graph[18] [40].

[18] A hierarchy of tree nodes what represent the visual components of an application's user interface[52]

```
def getMedian(listBuff: ListBuffer[Tempo], identifer: String): Double =
  identifer match {
    case "tempo" =>  val list = getTempos(listBuff.toList); median(list)
    case "diffs" => val list = getDiffs(listBuff.toList); median(list)
}

def median(list: List[Double]): Double = {
  //adapted from the sample at Rosetta code
  val (lower, upper) = list.sortWith(_<_).splitAt(list.size / 2)
  if (list.size % 2 == 0) (lower.last + upper.head) / 2.0 else upper.head
}

def getAverage(listBuff: ListBuffer[Tempo], identifer: String): Double =
  identifer match {
    case "tempo" =>  val list = getTempos(listBuff.toList); average(list)
    case "diffs" => val list = getDiffs(listBuff.toList); average(list)
  }


def average(list: List[Double]): Double = {

  @tailrec
  def averageTR(list: List[Double], size: Double, total: Double): Double = list match {
    case Nil => println(total); println(size); total/size
    case x :: xs => averageTR(xs, size + 1, total + x)

  }

  averageTR(list, 0, 0)
}

def getTotal(listBuff: ListBuffer[Tempo]): Double = {
  val list = getBeatCounts(listBuff.toList); totalbeatCount(list)
}

def totalbeatCount(list: List[Double]): Double = {

  @tailrec
  def totalHelper(lst: List[Double], total: Double): Double =
    lst match {
      case Nil => total
      case x :: xs => totalHelper(xs, x)
    }

  totalHelper(list, 0)
}


def getResponseTime(list: List[Tempo]): Long = list match {
  case (x :: xs) => {
    if (x.difference < 1.0 && x.difference > -1.0) {
      x.timeElapsed
    }
    else getResponseTime(xs)
  }
  case Nil => 0
}


def getTempos(lst: List[Tempo]): List[Double] =
   lst match {
     case x :: xs => x.tempo :: getTempos(xs)
     case Nil => Nil
}

def getDiffs(lst: List[Tempo]): List[Double] =
  lst match {
    case x :: xs => x.difference :: getDiffs(xs)
    case Nil => Nil
  }

def getBeatCounts(lst: List[Tempo]): List[Double] =
  lst match {
    case x :: xs => unwrapOption(x.beatCount) :: getBeatCounts(xs)
    case Nil => Nil
  }

def unwrapOption(head: Option[Double]): Double =
  head match {
    case Some(value) => value
    case None => 0
  }
```

Figure 24: Stats calculation methods used by the StatsCalculator

```
implicit val tempoWrites = new Writes[Tempo] {
  def writes(tempo: Tempo) = Json.obj(
    "tempo" -> tempo.tempo,
    "expectedTempo" -> tempo.baseTempo,
    "difference" -> tempo.difference,
    "elapsedTime" -> tempo.timeElapsed
  )
}

implicit val statsWrites = new Writes[Stats] {
  def writes(stats: Stats) = Json.obj(
    "averageTempo" -> stats.averageTempo,
    "medianTempo" -> stats.medianDiff,
    "averageDiff" -> stats.averageDiff,
    "medianDiff" -> stats.medianDiff
  )
}

implicit val wormAnalyzerWrites = new Writes[WormAnalyser] {
  def writes(analyser: WormAnalyser) = Json.obj(
    "name" -> analyser.name,
    "buffer" -> analyser.buffer,
    "stats" -> analyser.stats

  )
}
```

Figure 25: Example of the write method used to convert the Tempo and Stats object values to JSON

Before 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 that runs the audio capture and beat detection for thirty seconds, before closing down audio inputs and storing the results. During execution in both modes the RTT_Analyser displays the calculated tempo values 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. The results can then be viewed in a web browser using the *Results* button. The HTML tables were produced in HtmlWriter object which used reflection where possible to obtain the relevant field names, before using a number of tail recursive methods to write the field values and HTML tags to a StringBuilder object, chosen for it's mutability. The HTML tables are subsequently saved to an HTML file which is opened and viewed via the default browser. An example of the displayed results can be seen in Figure 27.

5 Software Testing

During the development of the RTT_Analyser, Test Driven Development (TDD) was employed where possible. TDD is a development process that uses 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 the process of making small changes to code and 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

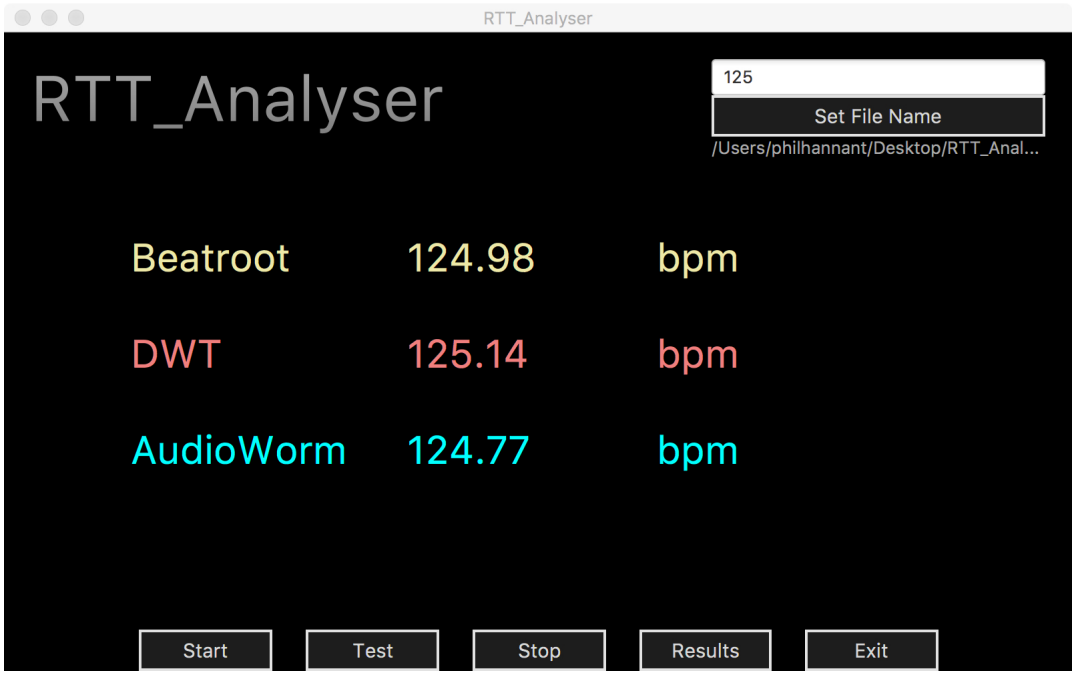


Figure 26: RTT_Analyser's User Interface

name	stats
worm: 2016-25-26, 09:25:10	
averageTempo	118.87383409364367
medianTempo	119.8527280210139
averageDiff	-58.873834093643325
medianDiff	-59.8527280210139
totalBeatCount	0.0
responseTime	0.0

tempo	baseTempo	difference	beatCount	timeEl
75.0	60.0	-15.0	None	276
148.32203293831626	60.0	-88.32203293831626	None	799
115.2256641406328	60.0	-55.2256641406328	None	930
117.97814954783244	60.0	-57.978149547832444	None	1323
119.01958295391182	60.0	-59.01958295391182	None	1840
119.50166377695483	60.0	-59.50166377695483	None	2099
119.7252361528647	60.0	-59.725236152864696	None	2359
119.53294273830856	60.0	-59.53294273830856	None	2620
119.67926220986494	60.0	-59.679262209864945	None	2880
119.54280558557888	60.0	-59.542805585578876	None	3140
119.22672502022178	60.0	-59.22672502022178	None	3375

Figure 27: Example of the RTT_Analyser's HTML results view

Scala 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 know the exact data that would be recorded due to the presence of background noise. 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.

A good 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 RTT_Analyser Beat Detection Testing

6.1 Drum Beat Sample Set

Once the software testing of the RTT_Analyser was complete, the full testing process could begin. The sample 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, 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 four samples. 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, effectively becoming noise [45]. Therefore, to compare the drum samples with cymbals a baseline was required. This change was also considered to help provide a much more diverse data set, hopefully helping 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.

Additonally, three Garageband stock samples were added to the sample set. These samples were included in an attempt to provide insight as to whether the beat detection algorithms are able to work efficiently, even when drum beat being tested is fairly complex.

In total, twenty eight 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. A full visualisation of the drum beats used can be found in Appendix A.

6.2 Real Time Tempo Tests

The real time tempo detection tests were carried out in an environment with as little background noise as possible. 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, where the tempo was increased in five bpm increments. Resulting in twenty one tests being carried out for each drum beat, producing a sample set consisting of 588 results files.

During the real time tempo tests it became apparent that at lower tempos the RTT_Analyser was returning values which were roughly twice that of the expected tempo. A possible explanantion for this was considered to be aliasing. Resulting in it not being possible to distinguish between the values from sampled audio as they contain periodic replications of the original spectrum[19]. As this behaviour was being displayed by all three of the beat detection algorithms. It was decided that all of the drum samples would be reprocessed through the RTT_Analyser. However, this time a low-pass filter would applied to the point at which the audio is captured. The low-pass filter was provided by the source_code.biz Java dsp collection and the cutoff frequency was set to 100 Hz.

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

7 Results

The twenty eight sample beats processed by the RTT_Analyser yielded 137,180 separate tempo records. A breakdown of the number of tempo records recorded per beat detection algorithm can be seen in Figure 28. The separate JSON files were then consolidated into one file, converted into a CSV file and connected to the data visualisation software package, Tableau[53].

TotalRecords

Beat Detection Name	Count of Tempos
beatroot	11,818
dwt	11,818
worm	113,544
Grand Total	137,180

Figure 28: Total number of tempo values recorded by the RTT_Analyser

7.1 Overall Average Difference

The difference between the expected tempo and the calculated tempo was determined by subtracting the calculated tempo from the expected tempo (8). If the difference is negative then the calculated tempo was greater than expected and if a positive difference was returned the tempo was less than the expected value.

$$difference = expectedtempo - calculatedtempo \tag{8}$$

The average difference for each of the three beat detection algorithms, split by style can be seen in Figure 29. The best performing beat detection method can be seen to be the DWT based algorithm by Tzanetakis *et al*'s [15], for the samples without a filter in a Swing style. As mentioned previously, during the testing process some of the tempo results returned were double that of the expected bpm value, possibly due to aliasing. A solution to the aliasing issue, hereafter referred to as doubling, was to add a low-pass filter and looking at the results on the whole it can be considered to have worked to a certain degree. For six out of the nine sets of results, those samples processed with a low-pass filter produced a lower average difference than those without a filter. Although, the margin of error between the expected tempo and the calculated tempo are still suitably large.

A flag was then applied to the results set to highlight any tempo result approximately double the expected value. A record was flagged, if the calculated tempo when divided by the expected tempo returned a result greater or equal to 1.95 (9). The flagged results were then filtered out of the data set and the updated results can be see in Figure 30. With the

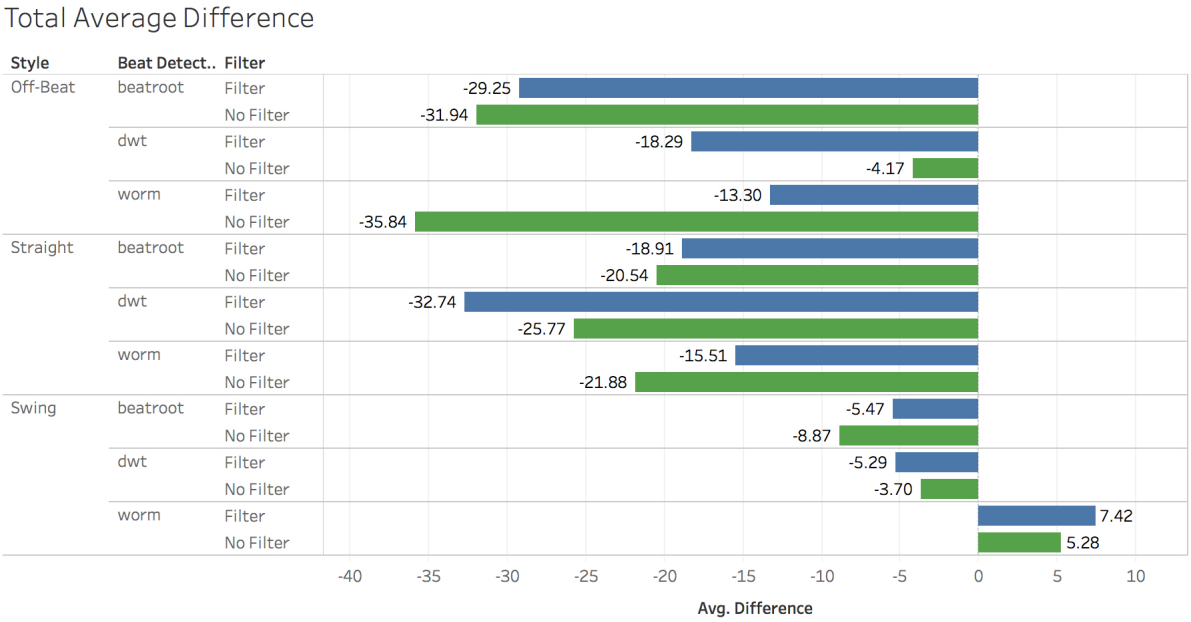


Figure 29: Total average difference results

flagged results removed the average difference value for all of the beat detection algorithms was greatly reduced except for the swing samples processed by the PW.

$$CalculatedTempo/ExpectedTempo >= 1.95 \tag{9}$$

Rather than just removing these results altogether, the next step was to attempt to correct any of the flagged tempo results that when divided by two, were within plus or minus five percent of the expected tempo result. With this adaption applied to the data set (seen in Figure 31), it is clear that overall the Tzanetakis *et al*'s [15] DWT method is the most accurate. The different musical styles within the samples also affect the accuracy of the calculated tempos. With the swing samples producing the least accurate results for all three algorithms.

7.2 Straight Sample Results

The results recorded for the straight sample set clearly show that doubling is an issue, demonstrated by all but one of the results returning a negative value (Figure 32). A large majority of the average difference values are also significantly large (greater than five). It is the Tzanetakis *et al*'s [15] DWT method which exhibits the poorest level of accuracy within this data set. Regularly producing average difference values of greater than twenty.

When the corrected difference values are applied to the data set the accuracy of the Tzanetakis *et al*'s [15] DWT method increases significantly (Figure 33). Although, it does return

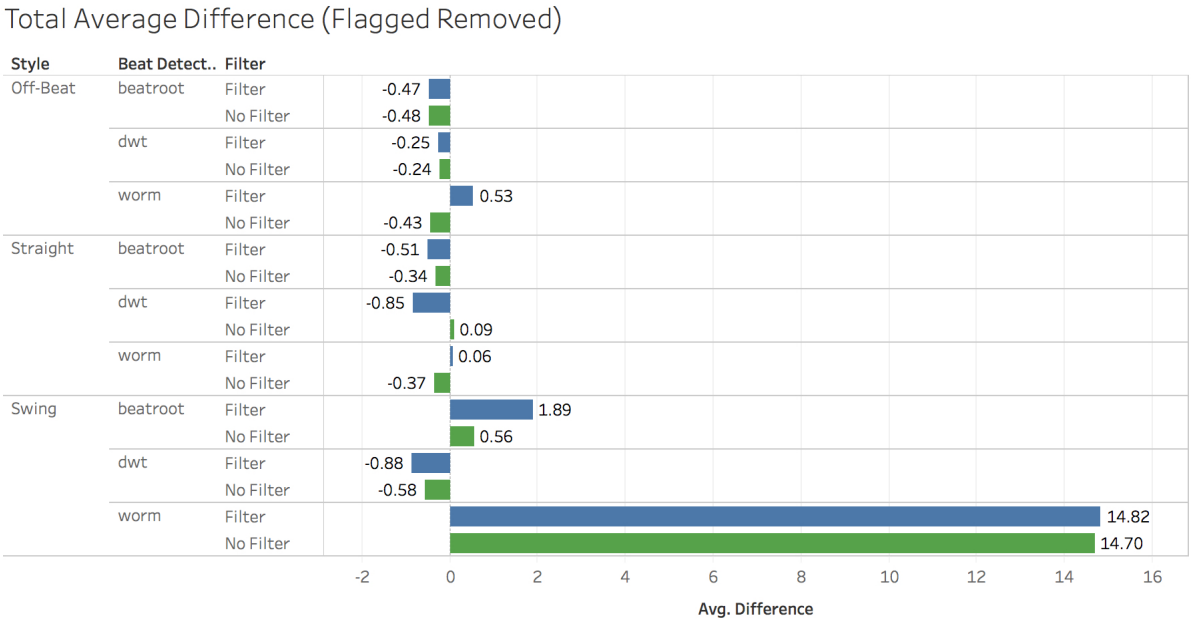


Figure 30: Total average difference results with flagged values removed

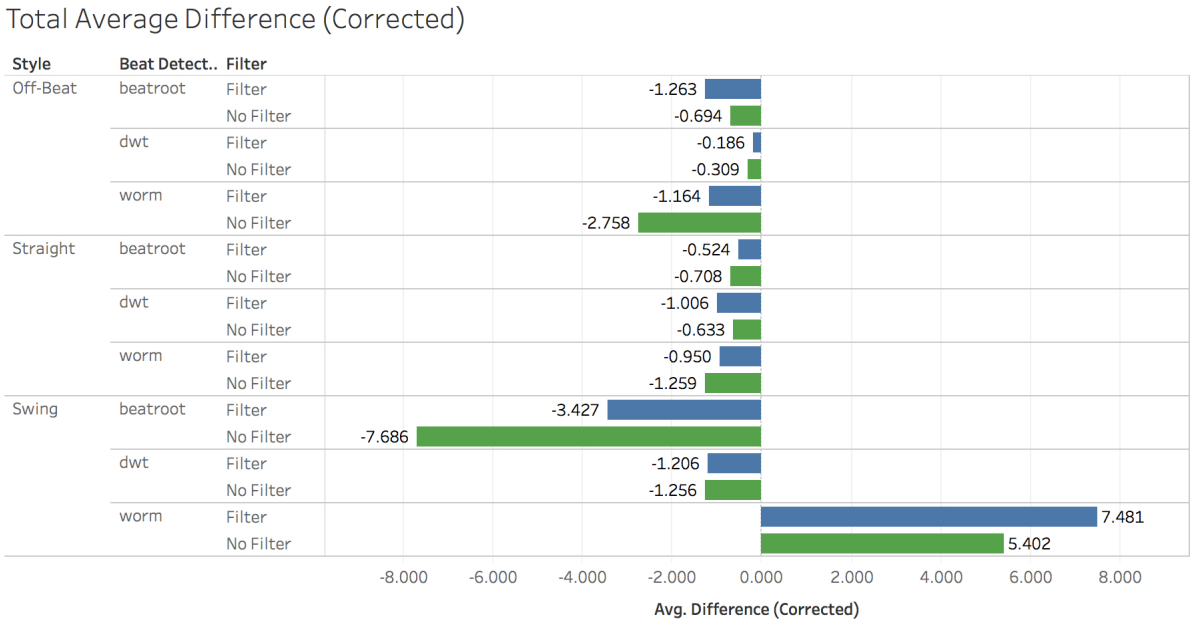


Figure 31: Total average difference results with correction applied

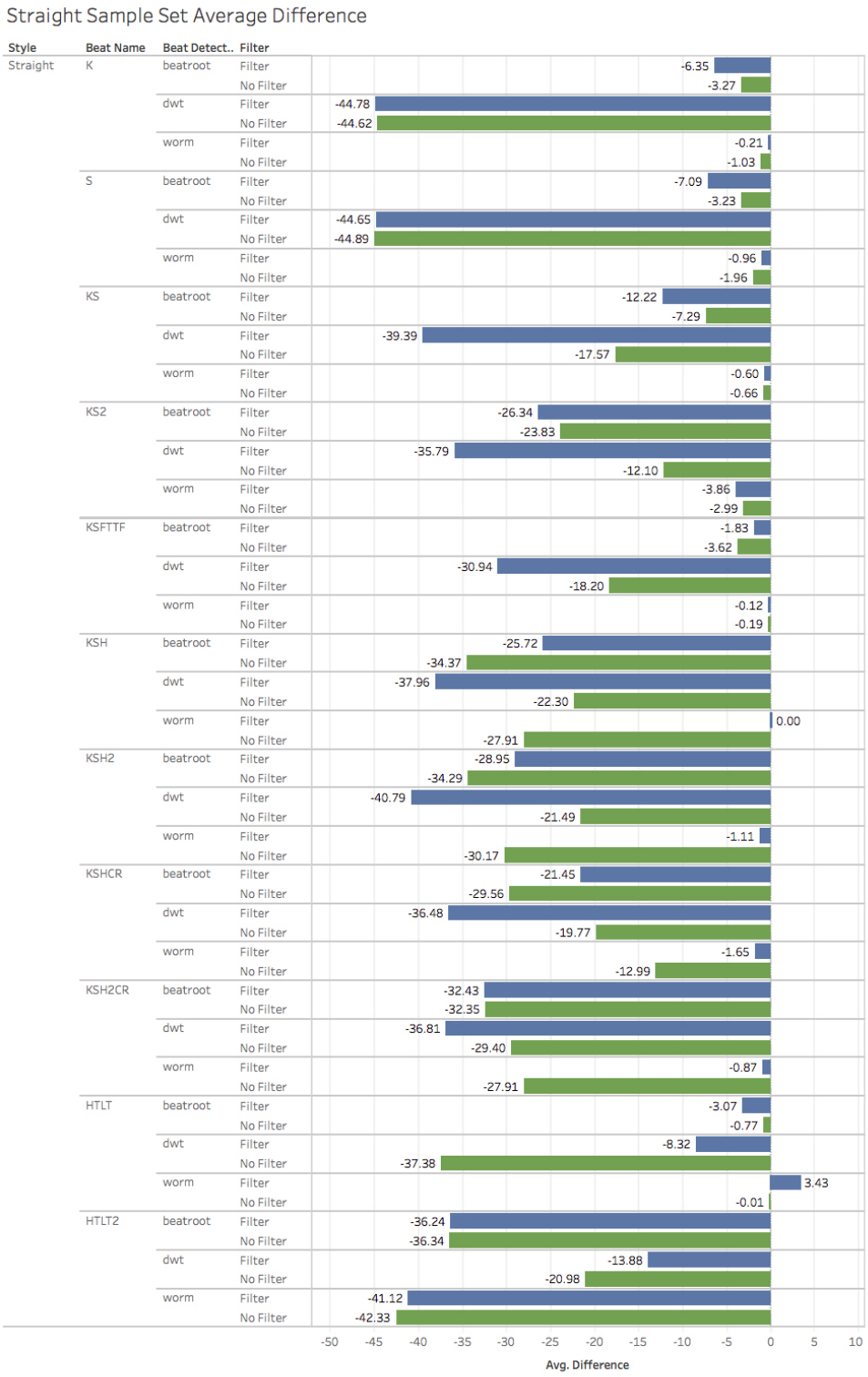


Figure 32: Straight sample set average difference results

its least accurate results for the samples consisting of a crash cymbal and the high and low tom-toms. The affect of the crash cymbal on the accuracy of the tempo recorded confirms the suspicion that loud cymbals will adversely affect the accuracy of the beat detection algorithms. The low accuracy recorded by the DWT method for the HTLT2 sample set where the time element is played on the low tom-tom. Suggesting, the possibility that this method will not respond as well when there are not distinct frequencies playing each part of the drum beat.

Also, within this data set the results produced for the single drum samples can be observed to actually have some of lowest levels of accuracy. This is improved once the corrected values are introduced, suggesting that the issue of doubling was particularly prevalent for these samples.

7.3 Swing Sample Results

The doubling of calculated tempos was not as prevalent within the Swing sample set. Both the original (Figure 34) and the corrected (Figure 35) difference values were significantly high. Although, the Tzanetakis *et al*'s [15] DWT method's accuracy is again improved by the updated difference values, the Beatroot and PW's performance remains very similar across both data sets.

The single drum elements within this data set were expected to cause some problems, due to their use of triplets. This can be seen for both the R and H sample sets within Figure 34. Once the corrected values are applied the DWT method does return a higher level of accuracy. However, the accuracy of both Beatroot and the PW are relatively unaffected by the introduction of the corrected values.

7.4 Off-Beat Sample Results

The last group of style results utilise a time element being played on the off-beat. The rest of the beat is considered to be straight. It is to be expected then that the results exhibit a similar amount of potential doubling of the tempo values as with the straight data set (Figure 36).

When the corrected difference values are introduced the accuracy is again improved (Figure 37). For the louder KSHHCRFTTF sample, the Beatroot and PW both recorded a lower level of accuracy with the low-pass filter than without. It is expected that the low-pass filter would help to remove some of the noise produced by the cymbals in this sample. However, this result suggests that by removing the higher frequencies produced by cymbals can adversely affect the accuracy of these two tempo calculation methods.

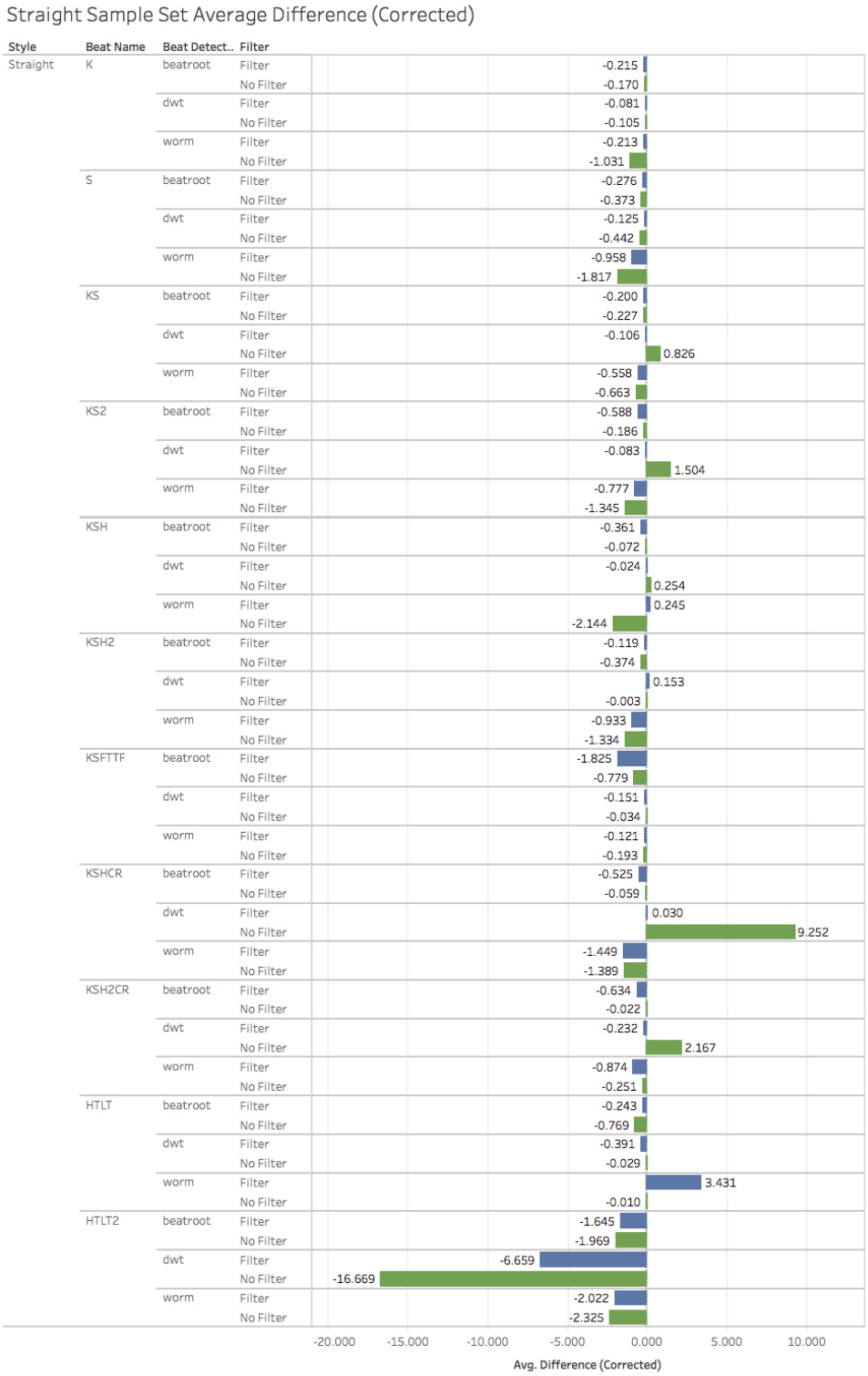


Figure 33: Straight sample set average difference results with correction applied

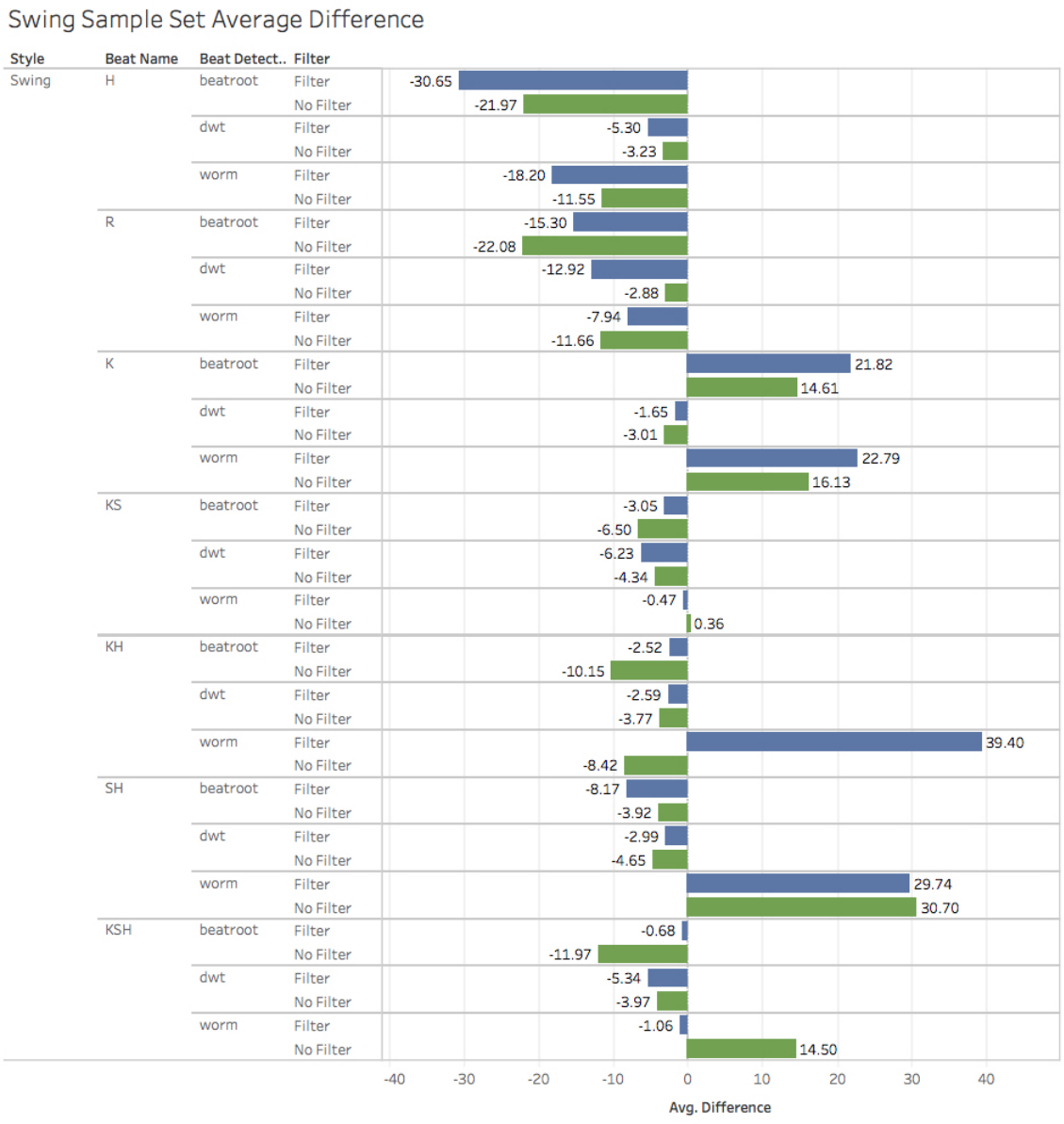


Figure 34: Swing average difference results

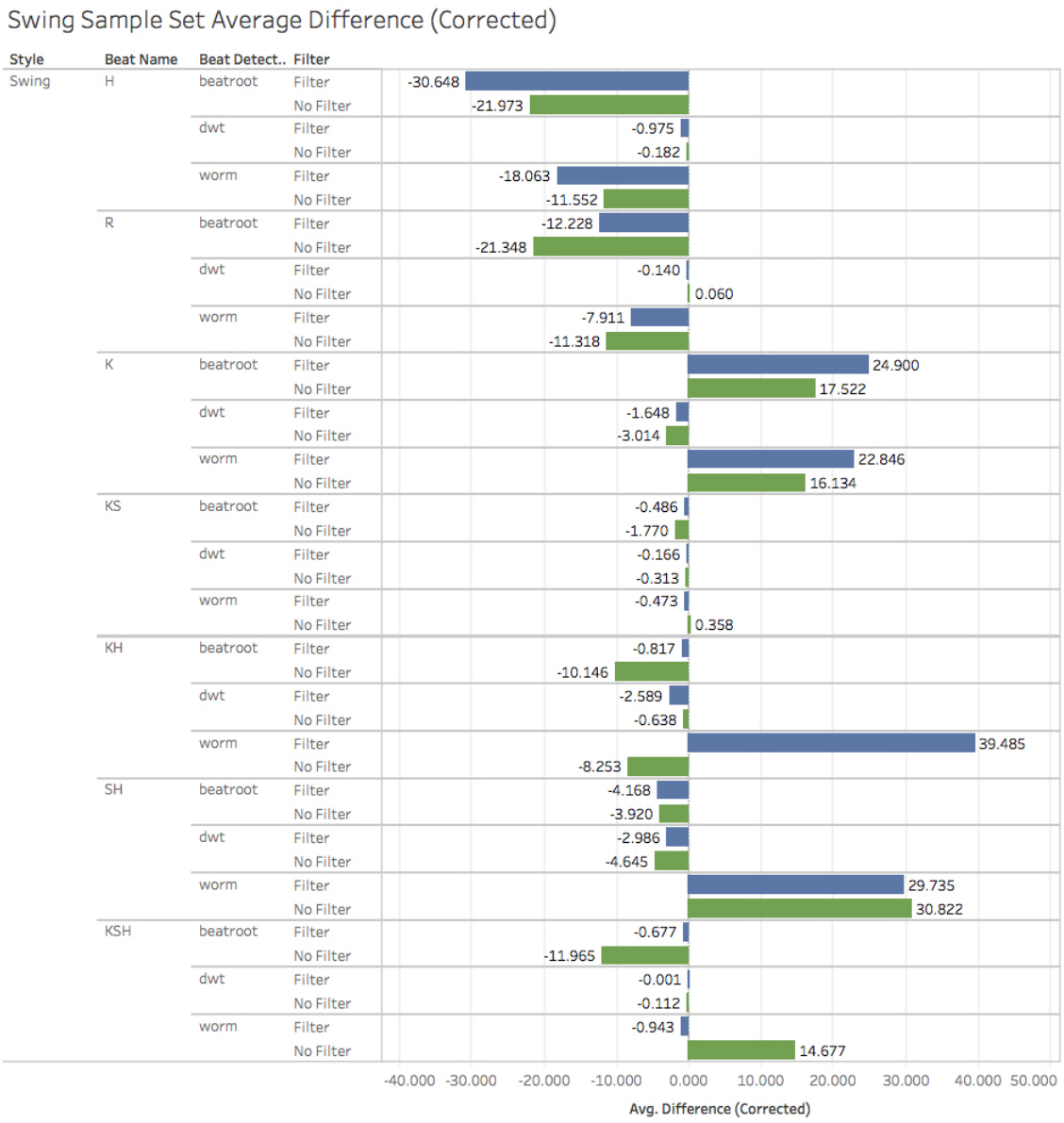


Figure 35: Swing average difference results with correction applied

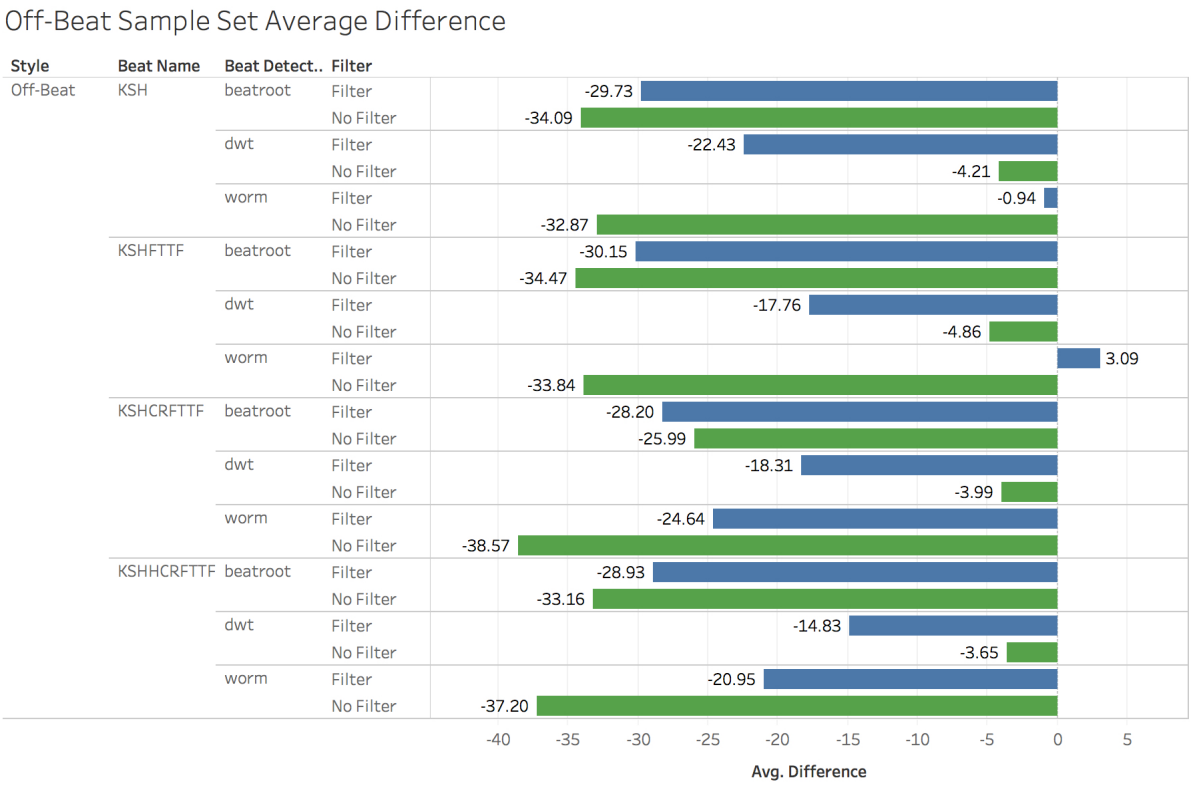


Figure 36: Off-Beat average difference results

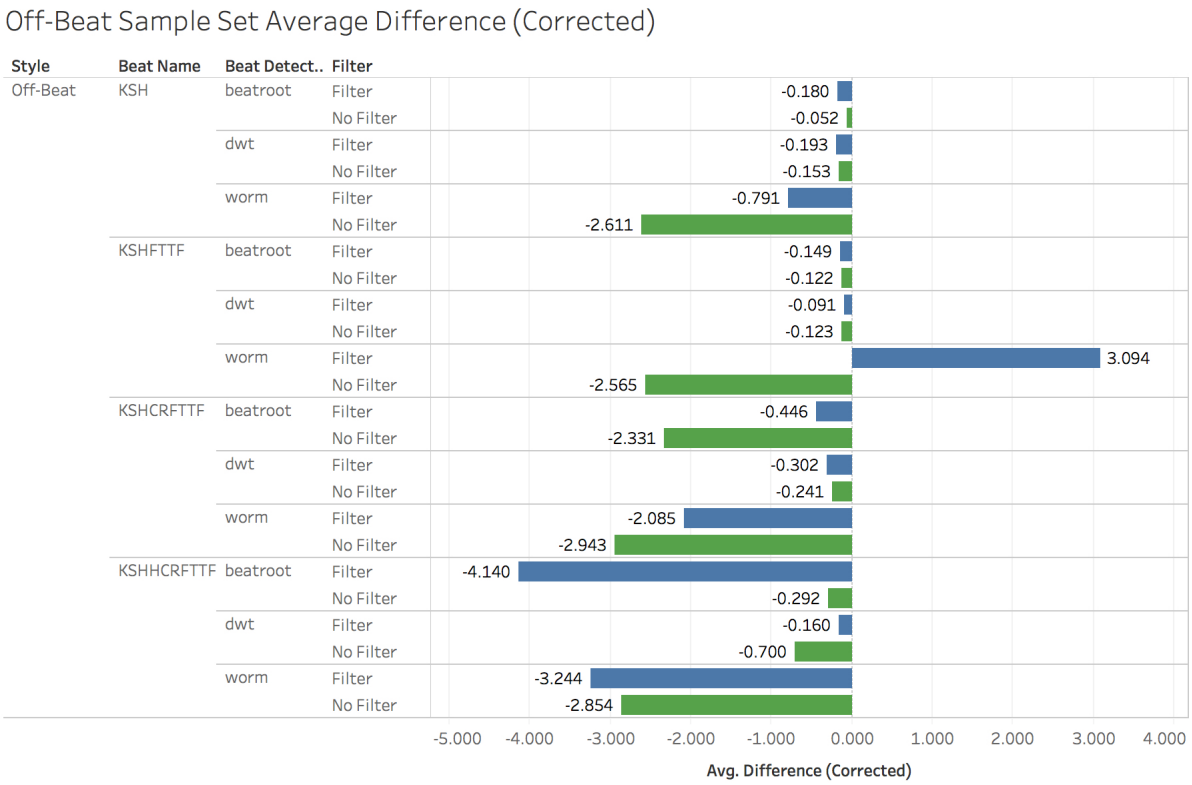


Figure 37: Off-Beat average difference results with correction applied

7.5 Low Tempo Comparison

During the testing process it was noted that the result doubling seemed to be prominent for samples being played at lower tempos. Usually anything with a bpm of one hundred and below. To investigate if the algorithms performed better with higher tempos regardless of style, a doubling count was applied to the data set. Using the same error margin applied previously.

The results viewed in Figure 38, confirm that the all three of the beat detection algorithms produce significantly less doubling when processing drum beats with higher tempos. If the style grouping is introduced (see Figure 39, it is apparent that the doubling is most prevalent with the straight sample set. Out of the three algorithms, the PW returns the highest count of doubled results. Although, this is to be expected as the PW returns a larger number of tempo values per sample.

7.6 Beat Detection Response Time

Due to the smaller window size utilised by the PW it is expected to perform the best within this comparison. The response time was recorded by the RTT_Analyser when the first returned tempo was plus or minus one bpm from the expected tempo result.

Before this could be calculated the all of the tempo results produced by the PW for expected tempos of seventy five were removed. As the PW's first result returned regardless of expected tempo was always seventy five. Despite this and as expected the PW returned the fastest tempo value (see Figure 40). Returning a tempo within the margin of error for the straight KSH2 beat within 0.675 seconds when processing through the low-pass filter. Then without a filter a tempo was produced in 0.660 seconds.

7.7 Overall Tempo Accuracy

The overall tempo accuracy was calculated by counting the number of tempo results that were plus or minus one bpm away from the expected tempo. The results for the three beat detection methods with and without a filter were then compared. A corrected value was also included to show the possible level of accuracy if the issue of doubling was not present (Figure 41).

Unsurprisingly, the Tzanetakis *et al*'s [15] DWT method performed with the highest level of accuracy. Interestingly, without the corrected values all three systems produced a consistent level of accuracy when using the low-pass filter. Without the filter though, the DWT method produced the highest overall accuracy of 71.32% and the PW produced the least of 57.12%.

The corrected results show that there is a clear distinction between the accuracy of the three methods. With the DWT method exhibiting the most potential followed by the Beatroot system and then the PW. The effectiveness of the low-pass filter is hard to ascertain

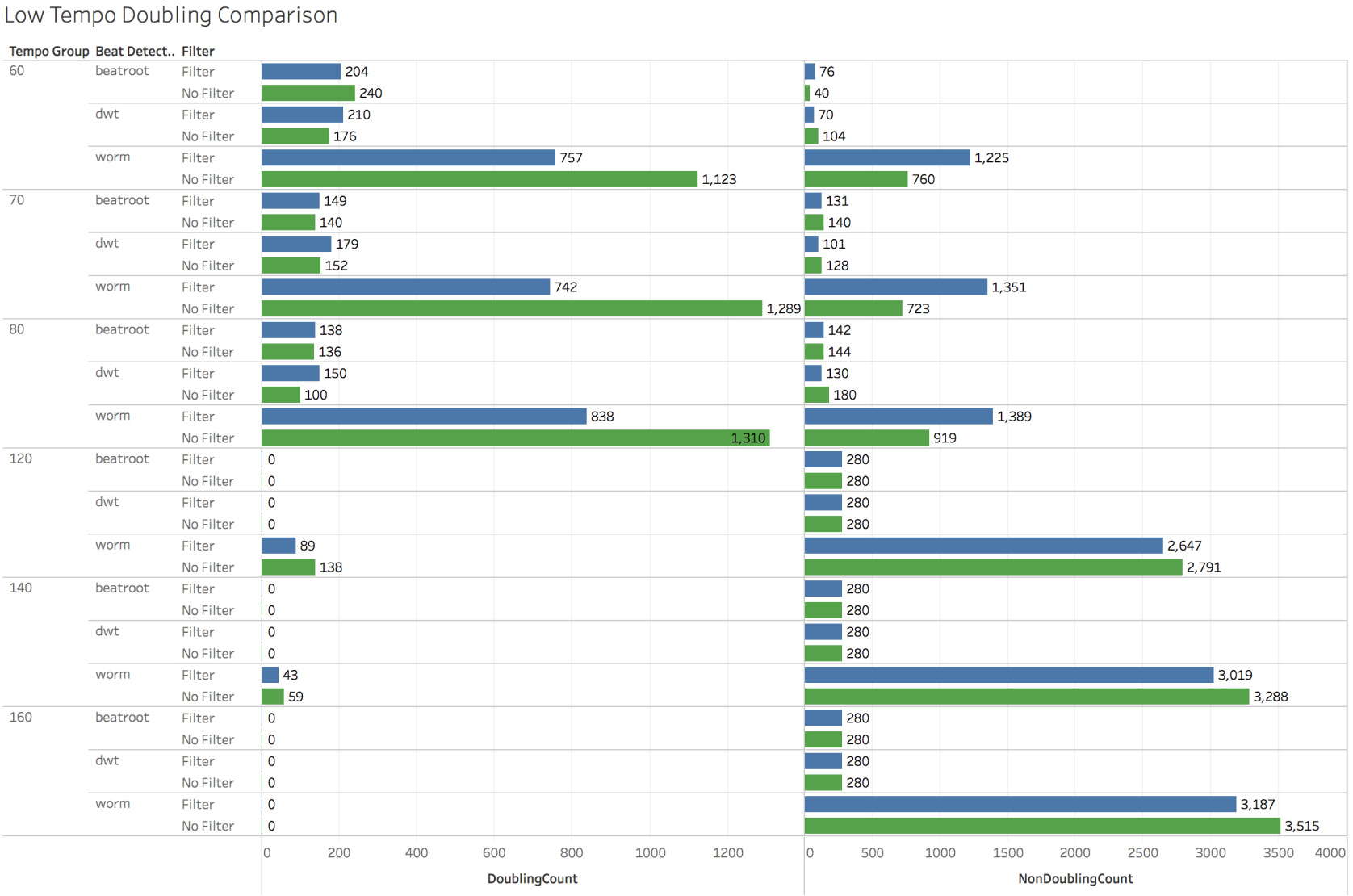


Figure 38: Low tempo comparison

Figure 39: Low tempo comparison with style groupings

Response Times

Beat Detection Name	Filter	Response Time (seconds)
beatroot	F	3.337
	NF	3.293
dwt	F	4.248
	NF	4.23
worm	F	0.675
	NF	0.66

Figure 40: Fastest response times recorded by the RTT_Analyser

Tempo Accuracy

Beat Detection Name	Filter	Tempo Accuracy	Tempo Accuracy (Corrected)
beatroot	Filter	64.22%	83.80%
	No Filter	64.94%	85.99%
dwt	Filter	64.96%	92.33%
	No Filter	71.32%	91.00%
worm	Filter	62.56%	74.04%
	No Filter	57.14%	76.51%

Figure 41: Overall tempo accuracy

from these results alone. However, based on the five percent increase to the overall tempo accuracy, it can be said that the PW benefited the most from the inclusion of the filter.

7.8 Filter/Non-Filter Comparison

To ascertain the effectiveness of the low-pass filter applied to the RTT_Analyser, the number of uncorrected tempo records within one bpm were collated for the sample sets with and without a filter. The number of records that fell within this accuracy margin and the percentage of the total number of tempo records can be seen in Figure 42.

The results show a slightly higher level of accuracy is provided for the sample set which used a low-pass filter but it is very marginal. The overall comparison of the total number of records can be seen in Figure 43, which further highlights the slight increase in accuracy offered by the introduction of a low-pass filter.

7.9 Analysis

the PW may be less accurate when processing samples played with a swing style. dwt hihat worse Demonstrating that if any future drum training tools require instant feedback then the PW needs to be considered. dwt high at low tempos suggesting a possible flaw in how the beat inducing algorithm converts peaks in the signal to the returned tempo result.

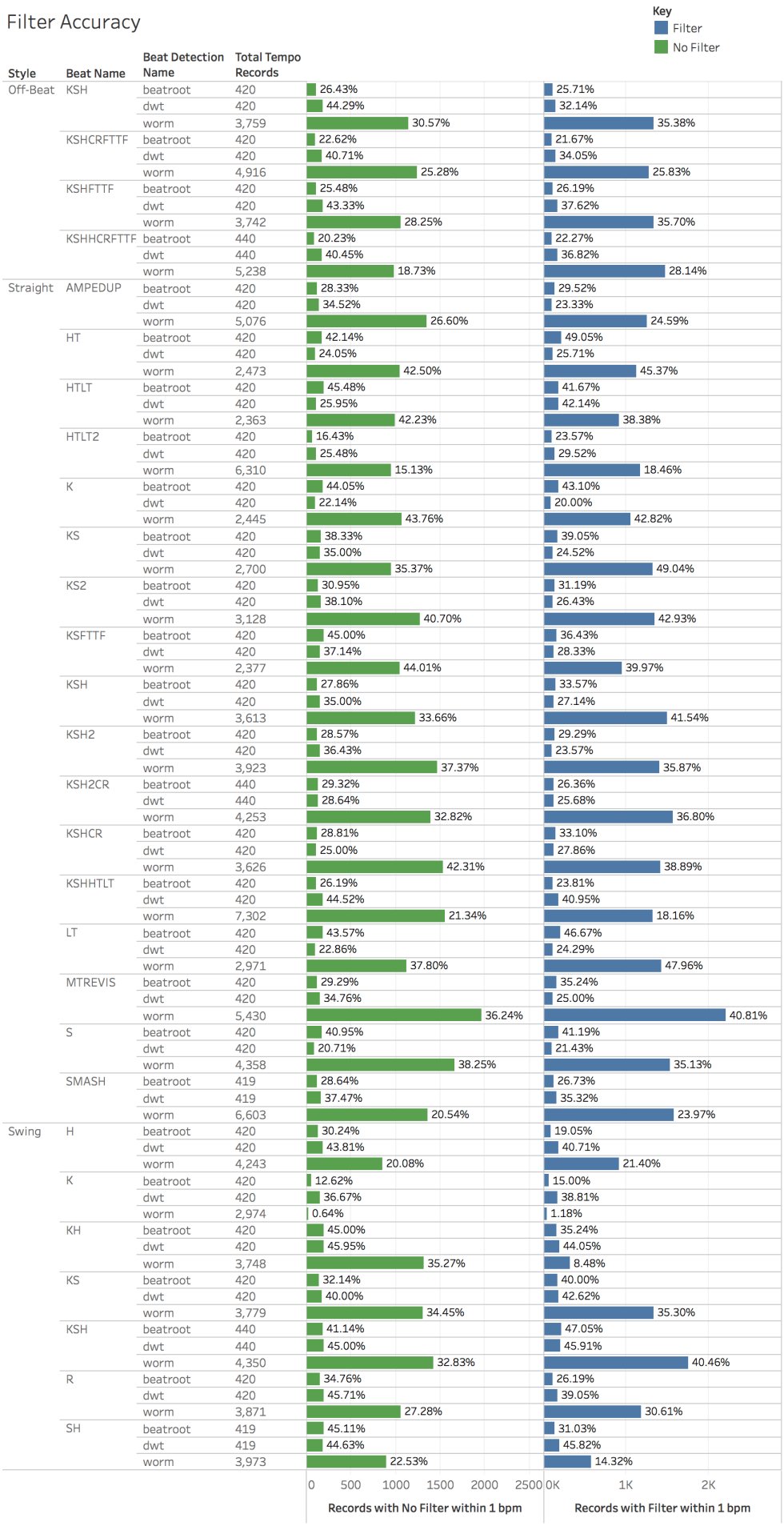


Figure 42: Filter/No Filter Accuracy Comparison, split by style and sample set

Overall Filter/No Filter Accuracy Comparison

Total Records	Records Filter within 1 bpm	Filter Accurcacy %	Records No Filter within 1 bpm	No Filter Accuracy %
137,180	42,221	30.78%	41,339	30.13%

Figure 43: Filter/No Filter Accuracy Comparison of total records

8 Evaluation and Discussion

8.1 Project Successes and Challenges

During the live audio testing phase of the project, the RTT_Analyser processed just under ten hours of audio without an issue, producing well over 130,000 different tempo records. This provided a sufficiently large data set to confidently assess whether any of the three beat detection algorithms were suitable to be implemented as part of a training tool for drummers. From the data set, it is clear that the Tzanetakis *et al*'s [15] DWT method offers the highest level of accuracy, with the potential of even higher if the issue of doubling is resolved. However, the response time of the Tzanetakis *et al*'s [15] DWT method might be an issue.

One challenge experienced during the development of the RTT_Analyser, was that the adaption of the Beatroot system to work with live audio took longer than anticipated. This was mainly due to the difficulty of working with a large pre-existing code base. To overcome this, the mitigation protocol set out in the project proposal was followed and an alternative system was sourced; the Performance Worm. This ensured that the project did not fall any further behind schedule.

The delay in development caused by this issue, led to insufficient time for the author to implement the Tzanetakis *et al*'s [15] beat detection algorithm. Although, there was a mitigation protocol in place, there was insufficent time to implement this, so a different course of action was chosen. A preexisting Scala implementation of the Tzanetakis *et al*'s [15] DWT method by Marco Ziccard[33] was used. Using this implementation, allowed for the regaining of the time lost during the initial failed attempt to adapt the Beatroot system. This ensured the project not only remained on schedule, but there was also sufficient time to try adapting the Beatroot system again. This was completed successfully on the second attempt.

One unforeseen challenge was presented by the original decision to use JSON files for data storage. Due to the inclusion of the PW beat detection algorithm in the RTT_Analyser and the decision to run tests with a low-pass filter and without, the number of records recorded by the RTT_Analyser increased significantly. The original estimate was for approximately 700 files each holding approximately ten results per beat detection method. The final number of results collection was 130,000 plus. JSON is more than adequate for holding this much data, an issue arose when the data visualisation software package, Tableau[53] was used. As Tableau does not work with JSON files, the data would need to be converted to another format. Due to limited time the data was converted to an Excel[55] file. This required a large amount of data manipulation to be carried out. A number of Visual Basic for Applications (VBA) macros were written to perform the data manipulation, and once the data was in the required format it was connected to Tableau where any further manipulation and analysis of the data could be performed much more efficiently.

8.2 Future Work

There are a number of possible routes to consider with regard to the future development of the RTT_Analyser, including, further investigations into the doubling issue, adaption of the data storage method, and converting the RTT_Analyser into a mobile training tool application.

In order to investigate the doubling issue presented during this project, a function to log the frequencies detected by the beat detection algorithm during live audio capture could be added to the RTT_Analyser. This could be achieved by logging the data produced by the parts of the Beatroot and Tzanetakis *et al*'s [15] DWT method that perform the original signal processing. This data can be examined to see if any particular frequencies are the cause of this issue.

Alternatively, the analysis could be performed on the drum beat samples before they are processed by the RTT_Analyser. The goal of this analysis would be to produce a frequency map of the samples. This data could then be compared with the output of the RTT_Analyser in an attempt to discover if there are particular frequencies for which doubling occurs.

If further drum beat testing is to be carried out, then adapting the data storage method of the RTT_Analyser from JSON files to a relational database should be considered. One option is to set up a MongoDB[56] database to hold the RTT_Analyser's data. An advantage of this is that the data structure used by MongoDB is very similar to structure of JSON objects. This would enable the existing data structures used by the RTT_Analyser to be maintained. Alternatively, the RTT_Analyser could be adapted to work with an SQL database, which could be the easier option, due to the author's familiarity with SQL.

Although, the accuracy of the beat detection algorithms used in the RTT_Analyser is not on the same level as the midi training tools, they could still provide a useful tool for a drummer. One possible route would be to develop an application for mobile devices which combines a metronome with the RTT_Analyser. A drummer could then use this application to aid their practice session by using the metronome to set the tempo. After a predetermined amount of time the metronome would stop and the drummer continues to play. While doing so, the application would report as to whether they have remained at the same tempo as the metronome or if their tempo has fluctuated. Such a tool would definitely provide a useful training aid to any drummer wanting to work on their timing.

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] Paul E. Allen and Roger B. 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] Richard G. Lyons
Understanding Digital Signal Processing, Third Edition, Prentice Hall, 2010

[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, IEEE TRANSACTIONS ON SPEECH AND AUDIO PROCESSING, VOL. 10, NO. 5, JULY 2002

[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*

[46] Tom O'Haver
A Pragmatic Introduction to Signal Processing with applications in scientific measurement, https://terpconnect.umd.edu/ toh/spectrum/, last update 2016

[47] *https://csd.wisc.edu/vcd202/rms.html*

[48] Doug Kaye
Loosely Coupled: The Missing Pieces of Web Services, RDS Press, 2003

[49] *https://developer.apple.com/library/mac/documentation/General/Conceptual/DevPedia-CocoaCore/MVC.html*

[50] Tony Bevis *Java Design Pattern Essentials, Ability First, 2012*

[51] *http://www.cs.umd.edu/class/sum2003/cmsc311/Notes/Data/endian.html*

[52] *http://docs.oracle.com/javase/8/javafx/getstartedtutorial/jfx-architecture.htm#A1106328*

[53] *http://www.tableau.com/*

[54] B. S. Everitt and A. Skrondal *The Cambridge Dictionary of Statistics, Fourth Edition, Cambridge University Press, 2010*

[55] *https://products.office.com/en-gb/excel*

[56] *https://www.mongodb.com/*

[57] *http://www.badlogicgames.com/wordpress/?p=161*

[58] *http://www.jsresources.org/faq_audio.html#frame_size*

A Drum Sample Beat Patterns

To be added