Rhythmic Inside: A Beat Detection Project

Automated Beat Detection and BPM Analysis Framework in Octave/MATLAB

Sven Fuchs, Joshua Reichmann

2024, March 4

Contents

1 Introduction

Beat detection is a fundamental aspect of digital signal processing in music analysis, playing a crucial role in a wide range of applications, including music production, DJing, fitness, and dance. Accurately identifying the tempo or beats per minute (BPM) of music not only enhances the creation and enjoyment of music but also supports various technological innovations in entertainment, health, and education.

1.1 Problem Statement

Although beat detection is significant, it remains a challenging problem due to the complexity and diversity of musical compositions. Variations in genre, tempo, instrumentation, and recording quality present considerable obstacles to developing a universal, accurate, and efficient beat detection system. Traditional methods often require manual intervention or are limited in their applicability across different musical styles. Therefore, an automated solution that can handle a wide range of music with high accuracy is urgently needed.

1.2 Significance of Beat Detection

Beat detection is crucial for various applications, affecting how we interact with music and sound. It plays a significant role in music production and DJing, enabling beat matching and mixing, which enhances the listener's experience. In fitness, precise tempo detection can synchronize music with workout intensities, improving exercise engagement and effectiveness. Furthermore, in the realm of interactive gaming and dance, beat detection allows for real-time music interaction, providing a more immersive and enjoyable experience for users.

1.3 Project Objectives and Scope

This project aims to tackle the challenges of beat detection by creating an advanced algorithm that can precisely and effectively determine the BPM of any music track. The objectives comprise:

- Designing and implementing a signal processing pipeline to analyze audio files. The pipeline will focus on filtering, energy signal calculation, auto-correlation, and peak detection.
- The algorithm's performance was evaluated through rigorous testing against established benchmarks and a wide variety of music genres.
- Developing a testing framework that is user-friendly to facilitate continuous improvement and validation of the algorithm.

This project encompasses the entire process of beat detection, from initial audio file preprocessing to the final BPM estimation. A comprehensive testing suite has been developed to ensure the algorithm's accuracy and reliability across different musical contexts.

In summary, this project aims to advance the field of beat detection by providing an innovative solution that utilizes signal processing techniques and testing methodologies to achieve high accuracy and efficiency. The goal of this work is to contribute to the broader understanding and application of beat detection in various music-related domains.

2 System Requirements

To guarantee the successful implementation of the 'Rhythmic Inside' beat detection project, the following system requirements must be fulfilled:

- **Operating System:** To run the scripts and commands associated with this project, a Unix-compatible shell environment is necessary. This requirement includes Linux distributions and macOS. Windows users may need to use a compatibility layer like Windows Subsystem for Linux (WSL) or a virtual machine running a Unix-based OS.
- **Octave Installation:** To use the project, you must have GNU Octave installed on your system. We recommend using Octave version 8.4.0, but most recent versions should also be compatible. You can install Octave from the official package repository of your Linux distribution or from the Octave website if you are a macOS user.

  After installing Octave, make sure to install the 'signal' package as well. To do this, enter the following command in the Octave command line interface:

```
pkg install -forge signal
```

Listing 1: Installing signal in MATLAB/Octave

- **Audio File Format:** The supported primary input format is WAV files in PCM format, which provide the uncompressed audio data required for accurate beat detection analysis.
- **FLAC to WAV Conversion:** For users who have audio files in FLAC format, we will provide a script to convert these files to the required WAV format. The script will also trim the audio files to the first 30 seconds to standardize the analysis duration and reduce processing time. Further instructions on how to use the scripts will be provided in the Testing framework section.

Before installing and executing the project, it is recommended to verify that all system requirements have been met to ensure optimal performance and reliability.

3 Project Structure and Implementation

The 'Rhythmic Inside' project utilizes a complex beat detection system in MATLAB/Octave to analyze WAV audio files and estimate their Beats Per Minute (BPM). The system is structured across multiple '.m' files, each encapsulating a specific part of the processing pipeline, which collectively contribute to the project's functionality. This section explains the modular architecture and the role of each component within the system. [1]

3.1 Folder Structure

The project is divided into three primary directories, which reflect the separation of source code, documentation, and additional assets:

- **src:** This directory contains all the source code and shell scripts required to run the beat detection algorithms. It includes individual '.m' files for each function in the processing pipeline, as well as the main script `main.m` which manages the execution flow.

[1] The source code for this project can be found at: `https://github.com/sid115/smp-project`

- **assets:** This folder contains graphical resources, sample WAV files for testing, and output PDFs generated from the `docs` directory. It is designed to aid in both testing and demonstrating the project's capabilities.
- **docs:** The LaTeX source code for the project's documentation is stored here. It offers a complete overview of the system, including its design, implementation details, and instructions for usage.

3.2 Implementation Details

The source code in the `src/rhythmic_inside` directory has a modular design, with essential components of the beat detection process implemented as separate functions:

- **preprocess.m:** The audio signal undergoes initial preprocessing, which includes converting it from stereo to mono and applying a low-pass filter to reduce high-frequency noise.
- **config.m:** A configuration script is used to define global parameters that are utilized across various functions. These parameters include filter specifications and threshold values.
- **calculateEnergy.m:** The preprocessed audio is used to compute the energy signal, which highlights variations in signal intensity and facilitates the detection of rhythmic patterns.
- **autoCorrelation.m:** Applies an auto-correlation function to the energy signal to identify periodicities that indicate beats.
- **detectPeaks.m:** The auto-correlated signal is analyzed to detect significant peaks, which may represent potential beats within the audio.
- **calculateBPM.m:** This function calculates the beats per minute (BPM) by analyzing the intervals between detected peaks in the audio signal.
- **customFilter.m:** This function applies a custom-designed Butterworth filter to an audio signal, which separates it into two signals with high-pass or low-pass characteristics. One signal is a high-pass version, and the other is a low-pass version.
- **processSignalForBPM.m:** Calls the `calculateEnergy`, `autoCorrelation`, `detectPeaks`, and `calculateBPM` functions to process the audio signal and identify potential beats.
- **recursiveFilter.m:** Recursively calls `customFilter` to divide an audio signal into a high-pass and low-pass version until the desired lowest frequency is reached. The resulting frequency bands are then processed by `processSignalforBPM`.
- **selectWAV:** Opens a graphical user interface (GUI) dialog to prompt the user to select a WAV file and returns the full file path of the selected WAV file.

The `main.m` script serves as the entry point for this application and integrates the components listed above. The process starts by sourcing the `config.m` script to define global parameters, such as filter specifications and threshold values. If a WAV file path is provided as an argument, it will be used; otherwise, the user will be prompted to select a file through a graphical user interface facilitated by `selectWAV`. Before analysis, the script loads the required 'signal' package to utilize advanced signal processing functions. The audio file is then read, and its signal is

preprocessed and analyzed through a series of function calls, including a recursive filtering process called `recursiveFilter`. The iterative method accumulates BPM values from different frequency bands, and the script concludes by calculating and displaying the average BPM of the input audio signal.

3.3 Modular Design and Flexibility

By structuring the project with each function in its own file, 'Rhythmic Inside' achieves high modularity, allowing for easy updates, maintenance, and scalability. This design approach not only facilitates the understanding and debugging of individual components but also supports the extension of the project to incorporate new features or algorithms in the future.

In summary, the 'Rhythmic Inside' project's organization and implementation strategy enables a robust and efficient beat detection system. The clear separation of concerns, combined with a thoughtful directory structure, ensures that the project is both manageable and adaptable to evolving requirements in the field of digital signal processing.

4 Configuration and Initial Setup

To begin using the 'Rhythmic Inside' beat detection system, follow these steps to set up the environment and configure the application to meet your specific needs.

4.1 Cloning the Repository

To begin, clone the project repository from GitHub and navigate to the `src/rhythmic_inside` directory where the source code is located. In a Unix-compatible shell, use the following commands:

```
1 git clone https://github.com/sid115/smp-project.git
2 cd smp-project/src/rhythmic_inside
```

Listing 2: Cloning from GitHub

4.2 Editing the Configuration File

The behavior of the project can be customized by modifying the constants defined in the `config.m` file. Although the default values are appropriate for most situations, you may need to adjust these settings to match specific audio characteristics or analysis preferences. To modify any of the following variables, open `config.m` in a text editor:

- **CUTOFF_FREQUENCY** (*integer*, Hz): The cutoff frequency for the low-pass filter is defined in order to remove high-frequency noise that is not relevant to the beat detection process. Acceptable values depend on the audio content, but it is typically set to 300 Hz.
- **EXPECTED_BPM** (*integer*, BPM): Influences the calculation of minimum samples between beats. Set the expected Beats Per Minute (BPM) to reflect the average tempo of the audio files being analyzed.
- **FILTER_ORDER_LIMIT** (*integer*): Specifies the maximum order for the Butterworth filter. A higher order will result in a sharper cutoff, but it may introduce phase distortion. Typically, values range from 1 to 5.

- **MIN_PEAK_MULTIPLIER** (*integer*): Multiplier for determining the minimum peak height in the signal's energy profile, affecting peak detection sensitivity.
- **PLOTS_PREFIX** (*string*): The file path prefix for saving output plots, which are stored in the `../assets/plots/` directory by default.
- **PASSBAND_RIPPLE** (*integer*, dB): The allowable ripple in the passband of the Butterworth filter, measured in decibels. A typical value is 3 dB.
- **SMOOTHING_WINDOW_DURATION** (*float*, seconds): Duration of the smoothing window applied to the signal, in seconds. This parameter affects the granularity of the energy calculation.
- **STOPBAND_ATTENUATION** (*integer*, dB): The required attenuation in the stopband of the Butterworth filter, measured in decibels. Commonly set to 40 dB to ensure significant reduction of unwanted frequencies.

After adjusting these variables as needed, save your changes to `config.m`. The system is now configured and ready for beat detection analysis.

4.3 Running the Project

After configuring the project to your preferences, execute the beat detection analysis by running the `main.m` script in a Unix-compatible shell environment. If you have not already, navigate to the `src/rhythmic_inside` directory of the project and use the following command to start the analysis:

```
1 octave main.m [path/to/wav]
```

Listing 3: Running the project

Replace the placeholder `[path/to/wav]` with the actual file path of the WAV file that you want to analyze. If you do not provide a file path, Octave will prompt you with a file manager dialog, allowing you to select a WAV file interactively. This feature ensures that the project is accessible and easy to use, whether you prefer specifying files via the command line or selecting them through a graphical interface.
The script processes the selected audio file according to the configurations set in `config.m`, performs beat detection, and outputs the estimated BPM to the console. This streamlined process enables efficient analysis of audio files, making 'Rhythmic Inside' a practical tool for researchers, musicians, and anyone interested in beat detection technologies.

5 Signal Processing

The signal processing phase is crucial in converting the raw audio input into a format suitable for beat detection. This section explains the preprocessing steps, such as filtering and energy calculation, and justifies the selection of signal processing techniques.

5.1 Preprocessing the Audio File

5.1.1 Filtering

The preprocessing process starts by applying a low-pass Butterworth filter to reduce high-frequency components that are not relevant to beat detection. This filter is preferred due to its flat passband, which ensures minimal distortion of relevant frequencies.

- **Cutoff Frequency:** The `CUTOFF_FREQUENCY`, set to 300 Hz, delineates the frequency above which signals are attenuated, chosen to retain fundamental rhythmic components while simplifying the beat detection.
- **Filter Order:** The `FILTER_ORDER_LIMIT` is configured at 5, balancing the need for a sharp frequency cutoff against minimizing phase distortions.

5.1.2 Energy Signal Calculation

After filtering, the energy of the signal is calculated by applying short, overlapping windows. This transformation emphasizes rhythmic patterns.

- **Windowing:** The signal is segmented into windows, within which the amplitude values are squared and summed to yield the energy.
- **Smoothing:** The energy signal is smoothed (using a window specified by `SMOOTHING_WINDOW_DURATI` defaulting to 0.01 seconds) to reduce fluctuations and emphasize beats.

5.2 Visualization of Energy Calculation

The graph below illustrates the process of energy calculation and smoothing. It plots normalized energy against time, represented in samples per thousand. The blue line represents the original energy signal, which shows the raw energy levels derived from the audio file. The red line represents the smoothed energy signal, which is the result of applying a moving average filter to the original energy.

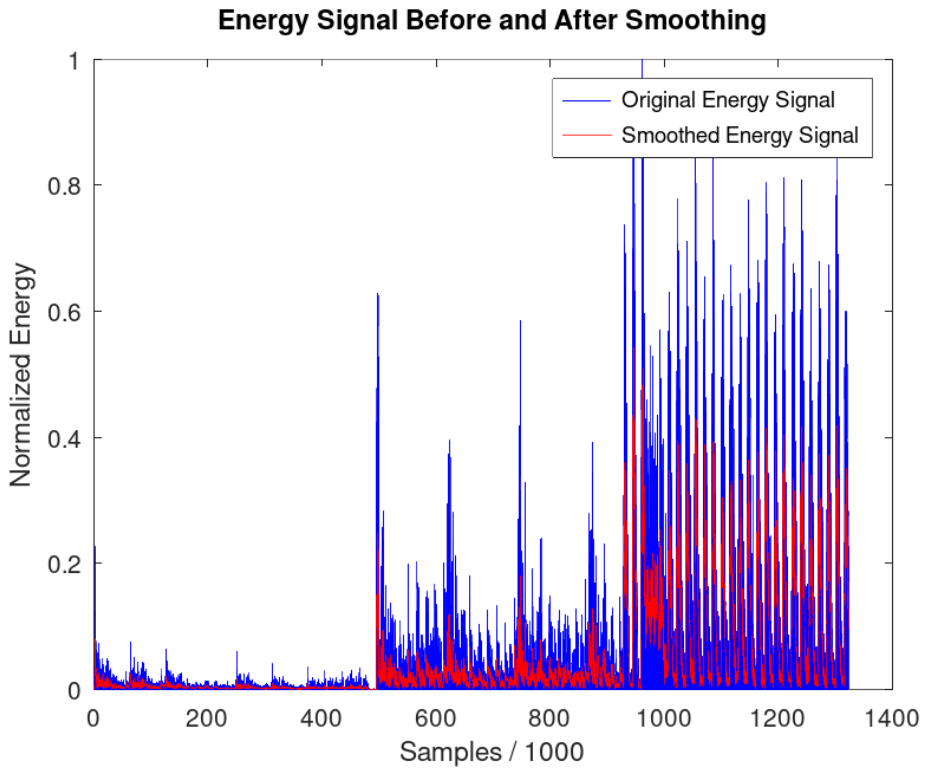

Figure 1: Normalized energy of the audio signal with original and smoothed energy.

5.2.1 Effect of Moving Average Smoothing

The original energy signal is smoothed using the moving average technique to reduce short-term fluctuations and emphasize underlying trends. This is done by averaging the energy values within a specified window (`SMOOTHING_WINDOW_DURATION`) that moves across the signal. Each point in the smoothed signal represents the average energy of the surrounding points within the window, effectively minimizing the impact of transient spikes or drops in energy.
The reason for smoothing the energy signal is to emphasize rhythmic patterns, which are crucial for precise beat detection. The moving average filter helps to identify significant peaks that correspond to the music's rhythm by reducing noise and highlighting sustained energy levels that indicate beats. This approach ensures that the subsequent beat detection process focuses on genuine rhythmic content rather than being misled by momentary energy changes.

5.3 Conclusion

The signal processing phase, which includes filtering and energy signal calculation, is crucial in converting raw audio into a format suitable for beat detection. The combination of low-pass filtering and energy smoothing prepares the audio signal by highlighting rhythmically significant patterns. The graph illustrates how the moving average filter smooths the energy signal, making rhythmic peaks more prominent and thus facilitating more reliable beat detection. Through these carefully selected processing steps, the project establishes a solid foundation for accurately detecting beats across various musical styles, demonstrating its effectiveness as an advanced beat detection system.

6 Auto-correlation and Peak Detection

This section explains the crucial steps of auto-correlation and peak detection, which are essential in determining the BPM from the preprocessed audio signal.

6.1 Auto-correlation to Identify Periodicity

Auto-correlation is used to identify the periodicity within the smoothed energy signal, which is a crucial step for rhythm detection. It measures the similarity of the signal with itself at different time lags, highlighting the regular pattern of beats. Utilizing the `xcorr` function, the project computes the auto-correlation sequence, efficiently revealing the temporal intervals with high similarity — indicative of the periodic nature of beats.

6.2 Overview of the `xcorr` Function Algorithm

The `xcorr` function is pivotal in signal processing, providing a means to compute the cross-correlation of two signals, or the autocorrelation of a single signal. This function can handle both real and complex data, in vector or matrix form, making it versatile for a range of applications.

6.2.1 Functionality

`xcorr` estimates the similarity between two signals (X and Y) over a range of time lags. For a single input vector (X), it calculates the signal's autocorrelation. When X is a matrix, it performs cross-correlation across columns, treating each as an individual signal.

6.2.2 Core Algorithm

The cross-correlation between X and Y for a lag k is defined as:

$$R_{xy}(k) = \sum_{i=1}^{N} x_{i+k} \cdot \text{conj}(y_i)$$

where N is the signal length, and $\text{conj}(y_i)$ denotes the complex conjugate of y_i. The function fills missing data (e.g., $x(-1)$, $y(N+1)$) with zeros.

6.2.3 Parameters

- **X, Y**: Input data vectors or matrices. If X is a matrix, Y must be omitted.
- **maxlag**: An integer specifying the maximum lag at which to calculate the correlation. If omitted, it defaults to $N-1$.
- **scale**: A string indicating the scaling of the correlation output, with options including 'none', 'biased', 'unbiased', and 'coeff'.

6.2.4 Computational Approach

`xcorr` employs a spectral method for efficiency, utilizing the Fast Fourier Transform (FFT) to calculate correlations across the specified range of lags. This method is particularly effective for large datasets, as the computational effort is independent of the number of lags.

6.2.5 Output

The function returns R, an array of correlation estimates across the specified lags, and $lags$, a vector indicating the corresponding lag values.

6.3 Peak Detection

After determining the periodicity, the project proceeds to identify the major peaks in the auto-correlation function, which directly correspond to the timing of the beats. The `findpeaks` function is invoked to discern the prominent peaks from the auto-correlation sequence, with parameters fine-tuned to discern genuine beat-related peaks from noise.

6.4 Overview of the `findpeaks` Function

The `findpeaks` function is designed to identify local maxima in a given set of data, a common requirement in signal processing and time series analysis. It operates on continuous-valued data arrays, optionally filtered by a specified threshold, to pinpoint peak locations.

6.4.1 Function Syntax

`findpeaks` is invoked with the syntax:

```
xmax = findpeaks(data, threshold)
```

where `data` represents the input signal, and `threshold` is an optional parameter that specifies a minimum value peaks must exceed to be considered.

6.4.2 Parameters

- **data**: A vector or matrix containing the signal data. If `data` is a matrix, it is treated as time x channels/trials, allowing for simultaneous peak detection across multiple channels or trials.
- **threshold** (optional): A scalar specifying the minimum value that data points must exceed to be identified as peaks. If omitted, all local maxima are returned.

6.4.3 Algorithmic Approach

The function identifies peaks by comparing each data point to its neighbors:

- For a point to be considered a peak, it must be greater than its immediate neighbors. This is determined by subtracting the values of preceding (`pp1`) and following (`pp2`) points from the current point and checking if the result is positive.
- If a `threshold` is specified, a data point must also exceed this value to be labeled as a peak.

6.4.4 Outputs

`findpeaks` returns `xmax`, a structure array with fields corresponding to each channel/trial in the input `data`. Each field contains the locations (indices) of the local maxima detected.

6.4.5 Usage Considerations

- The function is versatile, applicable to single vectors or multi-channel/time-series data, making it suitable for a wide range of applications in signal and data analysis.
- Specifying a `threshold` can help in filtering out noise and focusing on significant peaks.

6.5 Graphical Analysis

The graph below visually represents the process of auto-correlation and subsequent peak detection. It plots the normalized auto-correlation function (ACF) against time lag in seconds. The ACF is depicted in blue, showing how the signal's similarity with itself varies over different lags. Detected peaks are marked in red, indicating significant rhythmic patterns identified by the peak detection algorithm. The green line represents the dynamic threshold, which is a linear function used to determine the minimum threshold for beat detection. This threshold is crucial for distinguishing genuine rhythmic peaks from noise, ensuring that only significant peaks are considered in the estimation of BPM.

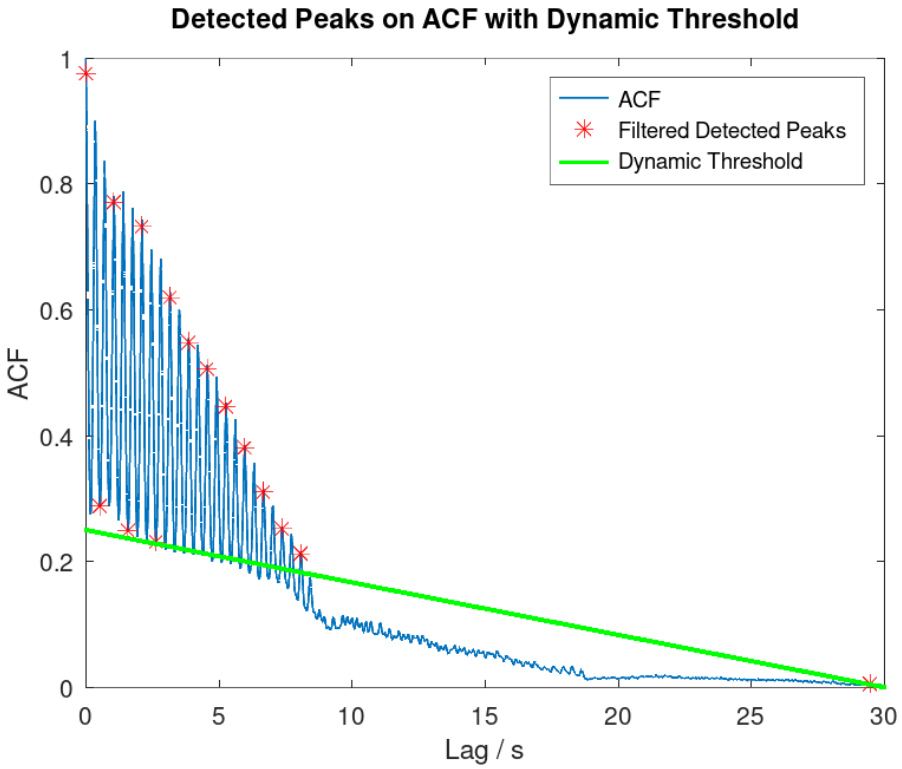

Figure 2: Normalized Auto-correlation Function (ACF) with detected peaks and dynamic threshold.

6.5.1 Decline of the ACF with Increasing Lag

The ACF usually decreases as the lag increases, as shown in the graph. This decrease is due to the decreasing similarity between the signal and its shifted versions as the lag increases. At short lags, there is a higher likelihood of significant overlap between portions of the signal, resulting in higher correlation values. As the lag increases, the overlap decreases, leading to a decrease in correlation. The ACF's characteristic behavior is useful for identifying periodic beats in audio signals, as true rhythmic patterns tend to produce prominent peaks in the ACF at consistent intervals.

The relationship between auto-correlation and peak detection is fundamental to the process of estimating BPM. Auto-correlation reveals the inherent periodicity of the signal, which forms the basis for accurately identifying beats through peak detection. The dynamic threshold is crucial in this process, as it ensures that only peaks representing genuine rhythmic content are used in the final BPM calculation. This study's methodological rigor enhances the accuracy of BPM estimation, affirming the project's adaptability to various musical genres and demonstrating a robust approach to beat detection.

6.6 Discussion

The synergy between auto-correlation and peak detection is fundamental to BPM estimation. Auto-correlation reveals the signal's inherent periodicity, which enables the precise identification of beats through peak detection. The use of `xcorr` and `findpeaks`, with their FFT and gradient-based algorithms respectively, highlights the methodological rigor in capturing the essence of rhythm. This approach enhances accuracy and affirms the project's adaptability to the multifaceted nature of audio signals, ensuring reliable BPM estimation across diverse musical genres.

7 Testing Framework

The testing framework developed for this project plays a pivotal role in validating the functionality and accuracy of the beat detection algorithm. A robust testing script is integral to this process, enabling the automated comparison of BPM estimates produced by our algorithm against those generated by the external tool bpm-tools. The bpm-tools, developed by Mark Hills[2], serve as a valuable benchmark for assessing the accuracy of our implementation. The bpm-tools project is released under the GPLv2 license (December 2021)[3].

7.1 Preparation of Test Audio Files

Prior to executing the testing script, it is crucial to prepare the audio files to ensure consistency and compatibility. This preparation involves two essential steps: converting FLAC files to WAV format and trimming WAV files to a uniform length.

7.1.1 Converting FLAC to WAV

The script `flac2wav.sh` is used to convert FLAC files to WAV format, which is necessary for the beat detection algorithm to have a uniform input format. To execute the conversion, follow these steps:

```
./flac2wav.sh path/to/flac_files_directory
```

Listing 4: Converting FLAC to WAV

7.1.2 Trimming WAV Files

The `cutWAVs.sh` script trims WAV files to the first 30 seconds to standardize the duration of audio samples for testing. This ensures that the BPM analysis is performed under consistent conditions across all test files. To use cutWAVs.sh, navigate to the testing directory and execute the script.

```
./cutWAVs.sh path/to/wav_files_directory
```

Listing 5: Trimming WAV Files

7.2 Testing Script Usage

After preparing the audio files, the testing script automates the evaluation process across these files. The script is designed to be executed in a Unix shell. Navigate to the `src/testing` directory and execute:

```
./testBulk.sh path/to/wav_files_directory
```

Listing 6: Executing the Testing Framework

This script processes each WAV file in the specified directory, comparing our BPM estimate with that obtained from `bpm-tools`. The results are compiled into `output_table.txt`, including the song name, our BPM estimate, and the `bpm-tools` estimate.
Additionally, `testBulk.sh` calls `plot_bland_altman.m`, generating a Bland-Altman plot that visually compares our BPM estimates against those from `bpm-tools`. This plot provides a graphical representation of the agreement between the two sets of BPM values, highlighting any systematic bias or variability.

[2]Mark Hills - bpm-tools project website: `http://www.pogo.org.uk/~mark/bpm-tools/`
[3]GPLv2 License - `http://www.gnu.org/licenses/old-licenses/gpl-2.0.html`

7.3 Test Cases and Methodology

The framework was applied to Tidal's Top 100 Germany tracks, providing a diverse range of genres and tempos for comprehensive evaluation. This diversity ensures that the testing encompasses a wide range of musical characteristics, challenging the algorithm's versatility and accuracy.

7.4 Statistical Analysis and Visualization

After running the `testBulk.sh` script to analyze Tidal's Top 100 tracks, we used statistical analysis and a Bland-Altman plot to evaluate the performance of our beat detection algorithm compared to `bpm-tools`. The statistical analysis showed that our algorithm detected an average BPM of 133.49, while bpm-tools detected an average of 121.313 BPM. The algorithm detected a median BPM value of 140, while `bpm-tools` detected a median BPM value of 122.846. The median relative difference between the two values was 3.96%, indicating a high degree of alignment in performance. Any minor discrepancies are likely due to differences in the algorithmic approaches used for BPM detection.
To visually assess the agreement between the two BPM detection methods and explore any systematic bias or variability, a Bland-Altman plot was generated. This plot displays the average BPM (calculated as the mean of the BPM values from our algorithm and `bpm-tools` for each track) on the x-axis against the relative difference (our BPM minus `bpm-tools`' BPM, relative to `bpm-tools`' BPM) on the y-axis. This visualization aids in identifying any consistent bias in our algorithm's BPM detection relative to `bpm-tools`, as well as the variability of this difference across the range of average BPMs.

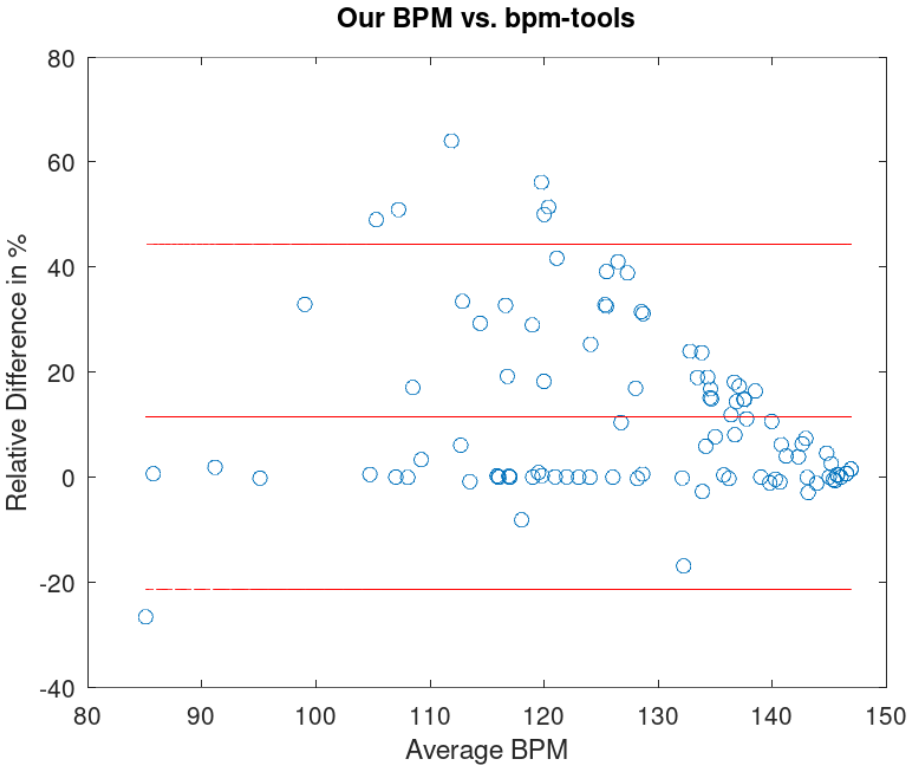

Figure 3: Bland-Altman Plot visualizing the agreement between our BPM detection and `bpm-tools`.

The analysis using Bland-Altman plot shows interesting trends in the performance of our beat

detection algorithm compared to bpm-tools, especially when the BPM values are close to 150. The convergence towards bpm-tools' estimates around the 150 BPM mark can be attributed to the configuration of our algorithm's EXPECTED_BPM parameter, which is set at 150. This setting affects the calculation of minimum samples between beats, directly impacting the accuracy of detected BPM values in this range. Adjusting the EXPECTED_BPM parameter may optimize performance across a broader range of BPM values.
It is important to note that bpm-tools, while highly regarded, is not infallible. A comprehensive evaluation of our algorithm's effectiveness would require a comparison against a robust and validated BPM database. Unfortunately, such a comparison was beyond the scope of this project's timeline. Efforts to facilitate this comparison are underway. Ongoing work is being done to secure API access to a suitable BPM database. This future work promises to provide a more detailed assessment of our beat detection algorithm's accuracy and reliability across a wide range of musical genres and BPMs.

7.5 Conclusion

The testing framework has proven to be indispensable in assessing the accuracy and reliability of the beat detection algorithm through its comprehensive and automated approach. It enables a nuanced understanding of the algorithm's performance by facilitating direct comparison with established benchmarks, guiding ongoing development efforts, and ensuring robustness across a diverse musical spectrum.

8 Multifrequency Analysis

The Multifrequency Analysis was developed as part of our Beat Detection project to improve the accuracy and reliability of BPM estimation. This analysis technique captures various frequency ranges within an audio signal, enabling a more precise BPM analysis. It is important to note that we are still in the active development phase. For educational purposes and ease of use, we implemented this manually to mimic the functionality of a Discrete Wavelet Transform (DWT).

It is also worth noting that this section is a work in progress and will be updated as we continue to develop and refine our Multifrequency Analysis algorithm. The following sections provide an overview of the functionality, algorithm, parameters, and statistical analysis of the Multifrequency Analysis component of our Beat Detection project.

8.1 Functionality

The Multifrequency Analysis code is intended to perform a thorough examination of various frequency ranges in an audio signal. This is accomplished by dividing the signal into different frequency bands and analyzing each band separately to extract its BPM. Analyzing different frequency ranges separately allows for the calculation of a mean value, which provides a more accurate indication of the signal's BPM.

8.2 Algorithm

This algorithm uses a recursive filtering method to calculate the Beats per Minute (BPM) from an audio signal. First, it checks if the cutoff frequency for the filter exceeds a certain threshold. If the check is positive, the signal is processed with a high-pass filter, and BPM is calculated from it. The results are then stored in an array. Next, the signal is processed with

a low-pass filter. The process is recursively repeated for the low-pass filtered signal, with the cutoff frequency halved each time to analyze a broader frequency band. This recursive filtering enables a step-by-step analysis of the signal across different frequency ranges, resulting in more precise BPM values. The final output is an array that contains all the calculated BPM values.

8.3 Parameters

- `signal`: The input signal to be analyzed.
- `fs`: The sampling rate of the signal.
- `cutoffFrequency`: The normalized cutoff frequency for the filter.
- `bpmArray`: An array to store the calculated BPM values.

8.4 Statistical Analysis

The following table presents the results of the BPM analysis using our Multifrequency Analysis algorithm across different music genres and normalized frequency bands. The table includes the actual BPM of the songs, the BPM estimated by our algorithm, and the BPM estimated by an established BPM detection tool (bpm-tools) for comparison.

Song	**Genre**	**50-100%**	**25-50%**	**12.5-25%**	**0-12.5%**	**Our BPM**	**bpm-tools**	**Actual**
1	Hypertechno	114.69	122.63	141.18	142.95	130.365	139.562	138
2	Pop Rock	97.658	111.787	131.217	140.797	120.365	145.685	115
3	Vocal Jazz	96.721	95.074	137.276	112.713	110.4461	95.981	95
4	Hypertechno	142.02	142.07	140.48	140.66	138.229	141.3104	142
5	Synthpop	120.21	124.84	142.01	136.70	130.9409	118.567	118

Table 1: BPM analysis results across different frequency bands for various music genres

The table displays the results of the BPM analysis utilizing our beat detection algorithm across various normalized frequency ranges (0%-25%, 12.5%-25%, 25%-50%, and 50%-100%) for different music genres. In general, our algorithm consistently estimates BPM across different frequency bands, with slight variations observed depending on the genre and frequency range. Notably, our algorithm tends to produce BPM estimates that closely align with the actual BPM for genres like Hypertechno and Synthpop, across all frequency bands. However, for Pop Rock and Vocal Jazz, there are instances where the BPM estimates deviate more significantly from the actual BPM, particularly in the higher frequency bands (50%-100%). The observation indicates that the algorithm may require further refinement, such as adjusting the weighting or processing of high-frequency components, to enhance accuracy across various music genres. Despite these variations, the algorithm generally demonstrates promising performance in BPM estimation, highlighting its potential usefulness in music analysis applications.

8.5 Statistical Analysis and Visualization

In our ongoing development of a beat detection algorithm that utilizes multi-frequency analysis, we have observed some initial trends that warrant discussion regarding our current state of progress. The analysis of normalized autocorrelation functions (ACF) over lag, across different frequency bands (50-100%, 25-50%, 12.5-25%, and 0-12.5%), has revealed a similarity in the ACF profiles across these spectrums. This similarity indicates a consistent temporal structure of

beats, although the highest frequency band (50-100%) occasionally exhibits a slightly divergent pattern. Therefore, it may be necessary to exclude or differently weight this band in beat detection for specific music genres due to its distinct characteristics and possible introduction of noise, particularly in genres less focused on high-frequency percussive sounds.

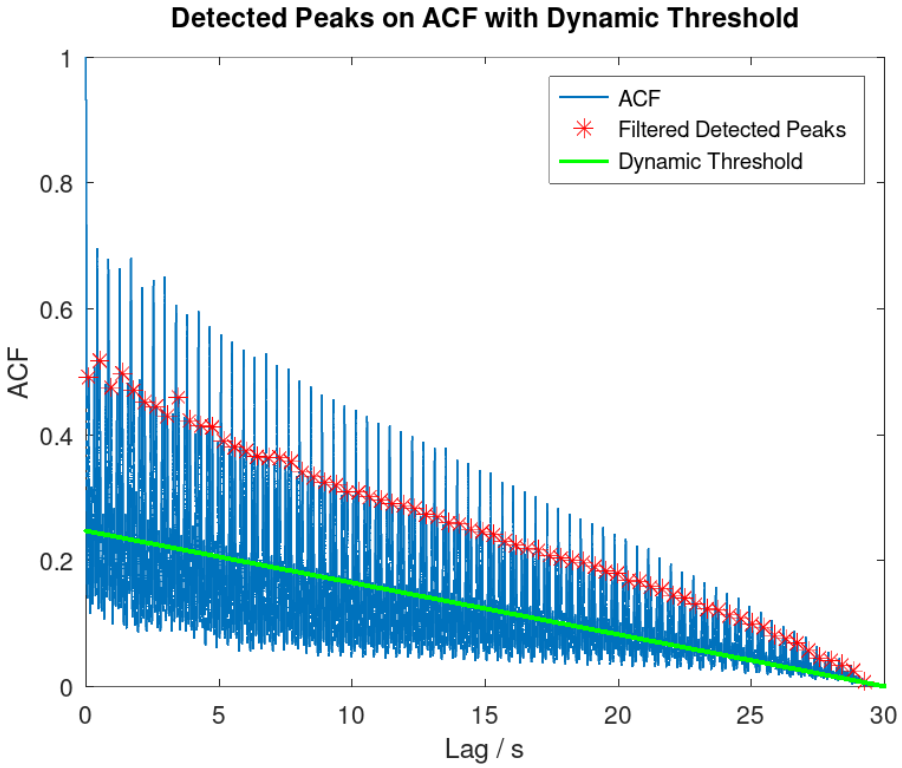

Figure 4: Normalized Autocorrelation Function Peaks Across Frequency Bands of Song 4: frequency band 0%-12.5%

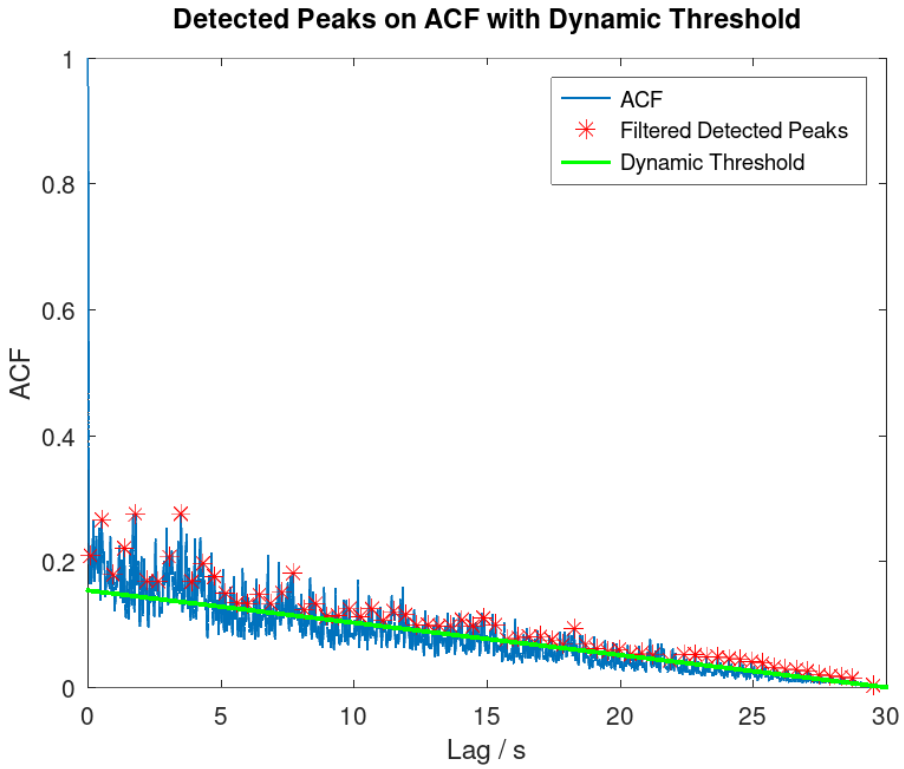

Figure 5: Normalized Autocorrelation Function Peaks Across Frequency Bands of Song 4: frequency band 12.5%-25%

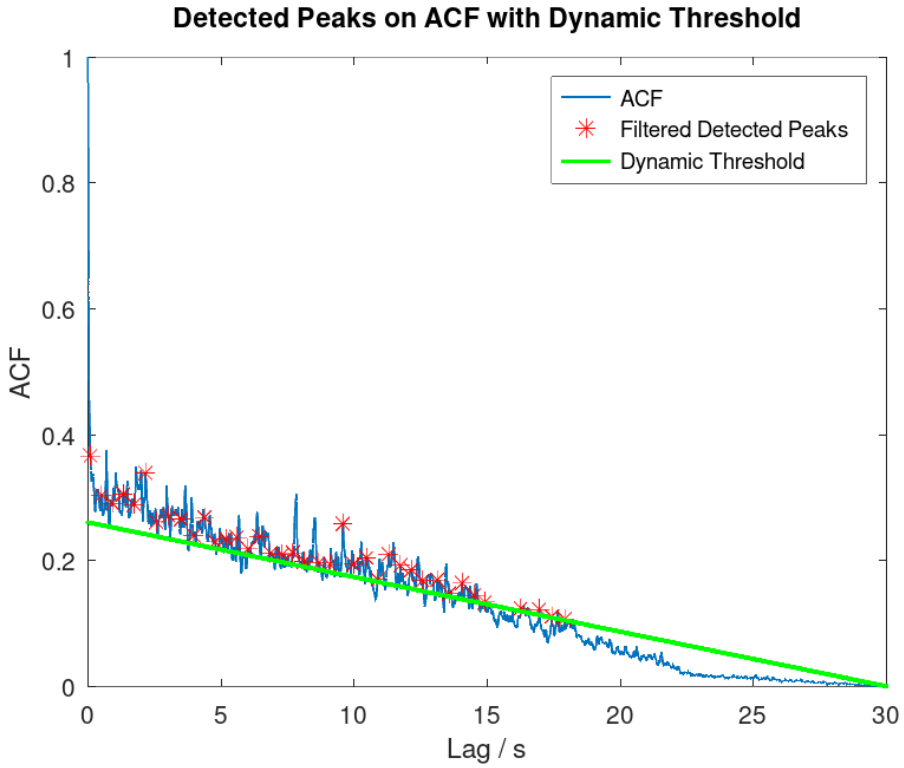

Figure 6: Normalized Autocorrelation Function Peaks Across Frequency Bands of Song 4: frequency band 25%-50%

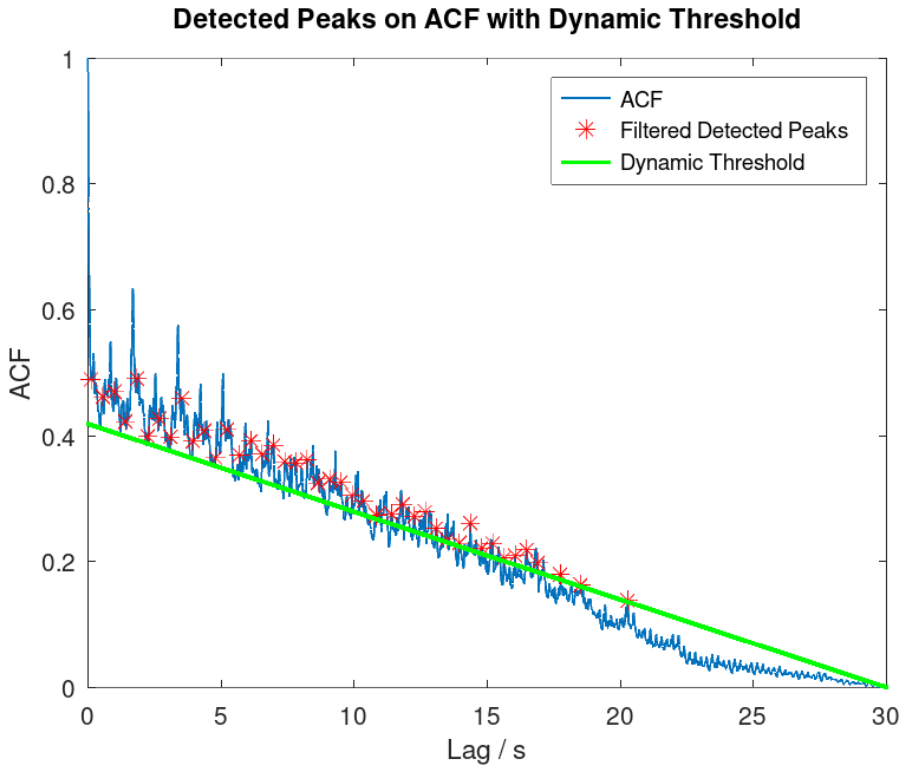

Figure 7: Normalized Autocorrelation Function Peaks Across Frequency Bands of Song 4: frequency band 50%-100%

Currently, our algorithm applies an equal weighting scheme to each detected beat across the frequency bands. However, this method may not be optimal as the significance of beats varies across different frequency ranges. Our preliminary findings regarding the frequency bands' contribution to beat detection highlight the critical and challenging task of parameterization within our algorithm. Our work is currently in the experimental phase. These observations will serve as a basis for further discussion and refinement. The task ahead involves adjusting the weighting of different frequency bands' contributions and possibly reevaluating the inclusion of the highest frequency band for some music genres. This stage of development is critical because it involves carefully considering the spectral content of music and the rhythmic prominence of different frequency ranges to improve the accuracy of our beat detection algorithm.

After reevaluating the performance of our beat detection algorithm using the same test protocol on Tidal's Top 100 tracks, our findings indicate that our algorithm has an average BPM detection rate of 124.479, while bpm-tools averages at 121.313 BPM. Additionally, the median BPM detected by our system is 125.5, compared to 122.846 by bpm-tools. The median percent difference between the detected BPM values of the two systems stands at 1.59%, showcasing a modest improvement in alignment compared to our previous assessments.

This improvement highlights the fact that significant advancements in beat detection accuracy are more dependent on fine-tuning algorithmic parameters than on broad-stroke feature development. During this phase of our project, we allocated a considerable amount of development time to refine our multi-frequency analysis approach. In hindsight, we realized that while feature enhancement was critical, parameter optimization is paramount. Moving forward, we must dedicate efforts to meticulously calibrating the algorithm's parameters to yield more significant improvements in performance. Our future development trajectory aims for a balanced

approach, pursuing both innovative feature development and rigorous parameter optimization concurrently to achieve optimal beat detection accuracy.

8.6 Conclusion

In summary, the Multifrequency Analysis component of our Beat Detection project has shown promising results in enhancing BPM estimation accuracy through the detailed examination of audio signals across various frequency bands. Our recent statistical analysis and performance reevaluation underline the incremental improvements we have achieved, particularly in aligning our BPM detection rates more closely with established tools. However, these findings also highlight the critical role of parameter optimization in achieving significant accuracy gains beyond mere feature enhancement. Our development journey has been a learning curve, emphasizing that beat detection precision is intricately linked to the delicate balance between innovative algorithmic features and meticulous parameter tuning. As we refine our approach, we will increasingly focus on optimizing these parameters to maximize the effectiveness and reliability of our beat detection algorithm. This process is crucial to advancing the state-of-the-art in audio analysis and setting new benchmarks for BPM detection technology.

9 Conclusion

The project's core achievements include the successful design and implementation of a signal processing pipeline. This pipeline preprocesses audio files, extracts relevant features, and applies auto-correlation and peak detection techniques to estimate BPM. The algorithm's performance was rigorously tested against a comprehensive dataset, demonstrating a high degree of accuracy when compared to established tools in the field. Furthermore, a robust testing framework has been developed to enable continuous validation and refinement of the algorithm, ensuring its reliability and effectiveness.

9.1 Challenges Faced

During the project, we faced and overcame several challenges. The task involved handling diverse musical structures and tempos, which required adjusting parameters and customizing filtering techniques. Additionally, we optimized the algorithm for computational efficiency while maintaining high accuracy, which posed a significant challenge. We had to carefully balance complexity and performance.

9.2 Future Work

Looking ahead, there are several potential avenues for future work to enhance the beat detection system. These may include:

- **Machine Learning Integration:** Incorporating machine learning techniques to improve the algorithm's adaptability and accuracy across a broader range of musical styles and recording qualities.
- **Real-time Processing:** Modifying the algorithm to support real-time beat detection, opening up applications in live music performance and interactive installations.

- **User Interface Development:** Creating a user-friendly interface that allows users to interact with the algorithm more effectively, providing real-time feedback and customizable parameters.
- **Algorithmic Optimization:** Continuing to refine the algorithm to enhance its efficiency and accuracy, including exploring alternative signal processing techniques and optimization algorithms.

9.3 Closing Remarks

In conclusion, this project has provided us with valuable insights into the field of beat detection. We faced several challenges that significantly contributed to our understanding of the subject, including technical difficulties such as signal processing and algorithmic accuracy assessments. This endeavor has broadened our academic and practical knowledge in audio signal analysis. Although we were unable to conduct a comprehensive comparison against an extensive BPM database due to time constraints, we continue to work on establishing a framework for such analysis in the future. We are grateful for the guidance we have received, which has been crucial in our pursuit of a nuanced understanding of beat detection algorithms.

10 Appendices

10.1 Appendix A: Source Code Listings

main.m:

```
% Include configuration file
source('config.m');

% Check if any arguments are passed
args = argv();
if length(args) >= 1
  % Use the first argument as the WAV file path if provided
  wavFilePath = args{1};
else
  % Use the file picker to select the WAV file
  wavFilePath = selectWAV();;
end

% Check if the 'signal' package is installed and load it
if isempty(pkg('list', 'signal'))
    error('The "signal" package is not installed. Please install it using "pkg install -forge signal".');
else
    pkg load signal;
end

% Read WAV file
[signal, fs] = audioread(wavFilePath); % Read audio data and sampling frequency from WAV file

% Initialize the BPM array and cutoff frequency
bpmArray = [];
```

```
cutoff = 0.5;

% Call the recursive filter function
iteration = 1; % Initialize the iteration counter
[bpmArray, iteration] = recursiveFilter(signal, fs, cutoff, bpmArray,
    iteration);

% Calculate the average BPM from the collected BPMs
averageBPM = mean(bpmArray);

% Display average BPM
disp(['Average BPM: ', num2str(averageBPM)]);
```

calculateEnergy.m:

```
function [smoothedEnergySignal] = calculateEnergy(signal, fs,
    iteration)

% Include configuration file
source('config.m');

% Original energy signal calculation without smoothing
energySignal = signal.^2; % Square signal to calculate energy

% Normalize the original energy signal
normalizedEnergySignal = energySignal / max(abs(energySignal)); %
    Normalize energy signal to range [0, 1]

% Calculate the smoothed energy signal
windowSize = round(SMOOTHING_WINDOW_DURATION * fs); % Calculate window
     size in samples
movAvgFilter = ones(windowSize, 1) / windowSize; % Create moving
    average filter coefficients
smoothedEnergySignal = conv(normalizedEnergySignal, movAvgFilter, '
    same'); % Apply convolution for smoothing

% Plot the original and smoothed energy signal
fig = figure('visible', 'off');
xValues = (1:length(normalizedEnergySignal)) / 1000;
plot(xValues, normalizedEnergySignal, 'b'); % Plot original energy
    signal in blue
hold on;
plot(xValues, smoothedEnergySignal, 'r'); % Plot smoothed energy
    signal in red
xlabel('Samples / 1000');
ylabel('Normalized Energy');
title('Energy Signal Before and After Smoothing');
legend('Original Energy Signal', 'Smoothed Energy Signal');
print(fig, strcat(PLOTS_PREFIX, 'energy', num2str(iteration), '.png'),
     '-dpng'); % Save plot
hold off;
close(fig);

end
```

autoCorrelation.m:

```
function [lag, acf] = autoCorrelation(signal, fs)

% Perform auto-correlation on the signal
[acf, lags] = xcorr(signal, 'coeff');

% Normalize lags to time
lag = lags / fs;

% 'acf' is the auto-correlation function and 'lag' are the
   corresponding time lags.
end
```

detectPeaks.m:

```
function [filteredPeaks, filteredLocations] = detectPeaks(acf, lag, fs
    , iteration)

% Include configuration file
source('config.m');

% Validate input arguments
if isempty(acf), error('acf is empty'); end
if length(acf) ~= length(lag), error('acf and lag are not the same
    size'); end
if fs <= 0, error('fs is not positive'); end

% Only consider positive lags for peak detection
positiveIdx = lag >= 0;
lagPositive = lag(positiveIdx);
acfPositive = acf(positiveIdx);

% Hotfix: Set negative values to zero
acfPositive(acfPositive < 0) = 0;

% Define the linear function for minimum peak height: f(lag) = -slope
    * lag + intercept
slope = (mean(acfPositive) * MIN_PEAK_MULTIPLIER) / max(lagPositive);
    % Example slope calculation
intercept = mean(acfPositive) * MIN_PEAK_MULTIPLIER; % Set intercept
    such that f(0) equals initial minPeakHeight

% Calculate dynamicMinPeakHeight for each positive lag
dynamicMinPeakHeight = max(-slope * lagPositive + intercept, 0); %
    Ensure non-negative

% Detect peaks in the ACF (only consider positive lags)
[peaks, locations] = findpeaks(acfPositive);

% Initialize arrays for filtered peaks
filteredPeaks = [];
filteredLocations = [];
lastLocation = -inf;

% Filter peaks by this height and minimum distance
for i = 1:length(locations)
    currentLag = lagPositive(locations(i));
    if peaks(i) > dynamicMinPeakHeight(locations(i))
        % Check if the current peak meets the dynamic height
    requirement and minimum distance
        if isempty(filteredLocations) || (currentLag - lastLocation) >
    (60 / EXPECTED_BPM)
            filteredPeaks = [filteredPeaks, peaks(i)];
            filteredLocations = [filteredLocations, currentLag];
            lastLocation = currentLag;
        end
```

```
    end
end

% Plot the ACF, detected peaks, and dynamic threshold
fig = figure('visible', 'off');
plot(lagPositive, acfPositive, 'LineWidth', 1); % Plot ACF
hold on;
plot(filteredLocations, filteredPeaks, 'r*', 'MarkerSize', 8); % Plot
   filtered detected peaks
plot(lagPositive, dynamicMinPeakHeight, 'g.-', 'LineWidth', 1.5); %
   Plot dynamic threshold
xlabel('Lag / s');
ylabel('ACF');
title('Detected Peaks on ACF with Dynamic Threshold');
legend('ACF', 'Filtered Detected Peaks', 'Dynamic Threshold');
print(fig, strcat(PLOTS_PREFIX, 'peaks', num2str(iteration), '.png'),
   '-dpng'); % Save plot
close(fig);

end
```

calculateBPM.m:

```
function bpm = calculateBPM(filteredLocations)

% Check if there are enough locations to calculate BPM
if isempty(filteredLocations) || length(filteredLocations) < 2
    bpm = 0; % Not enough data to determine BPM
    return;
end

% Calculate bpm from median
bpm = 60 / median(diff(filteredLocations));

end
```

config.m:

```
% Constants

CUTOFF_FREQUENCY = 300; % Low-pass filter cutoff frequency in Hz
EXPECTED_BPM = 150; % Expected BPM for calculating minimum samples
   between beats
FILTER_ORDER_LIMIT = 5; % Maximum order for the Butterworth filter
MIN_PEAK_MULTIPLIER = 2; % Multiplier for the minimum peak height
PLOTS_PREFIX = '../../assets/plots/'; % Prefix for the output plots
PASSBAND_RIPPLE = 3; % Passband ripple in dB for the Butterworth
   filter
SMOOTHING_WINDOW_DURATION = 0.01; % Duration of the smoothing window
   in seconds
STOPBAND_ATTENUATION = 40; % Stopband attenuation in dB for the
   Butterworth filter
LOWEST_NORMALIZED_CUTOFF_FREQUENCY = 0.05; % Lowest normalized cutoff
   frequency for the Butterworth filter
```

customFilter.m:

```
function filteredSignal = customFilter(signal, fs, high,
    normalizedCutoff)

% Include configuration file
source('config.m');

if size(signal, 2) > 1
    signal = mean(signal, 2); % Averaging two channels to convert
    stereo to mono
end

% Set filter type based on 'high' flag
if high
    variant = 'high'; % High-pass filter
else
    variant = 'low'; % Low-pass filter
end

% Check for valid normalized cutoff frequency
if normalizedCutoff <= 0 || normalizedCutoff >= 1
    error('Invalid normalized cutoff frequency');
end

% Calculate filter order and normalized cutoff using Butterworth
    design
[n, Wn] = buttord(normalizedCutoff, normalizedCutoff * 1.1,
    PASSBAND_RIPPLE, STOPBAND_ATTENUATION);

% Limit filter order to prevent instability
order = min(n, FILTER_ORDER_LIMIT);

% Design Butterworth filter according to the specified variant (high
    or low)
[b, a] = butter(order, Wn, variant);

% Apply filter using zero-phase method
filteredSignal = filtfilt(b, a, signal);

end
```

processSignalForBPM.m:

```
function bpm = processSignalForBPM(signal, fs, iteration)

% Calculate the energy signal and smooth it
[smoothedEnergySignal] = calculateEnergy(signal, fs, iteration);

% Use the auto-correlation function to identify the periodicity in the
    signal
[lag, acf] = autoCorrelation(smoothedEnergySignal, fs);

% Detect peaks
[peaks, locations] = detectPeaks(acf, lag, fs, iteration);

% Calculate the BPM
bpm = calculateBPM(locations);

end
```

recursiveFilter.m:

```
function [bpmArray, iteration] = recursiveFilter(signal, fs,
    cutoffFrequency, bpmArray, iteration)

% Include config file
source('config.m');

if cutoffFrequency > LOWEST_NORMALIZED_CUTOFF_FREQUENCY
    % Apply highpass filter
    highpassSignal = customFilter(signal, fs, true, cutoffFrequency);

    % Calculate BPM for highpass signal and add to bpmArray
    bpm = processSignalForBPM(highpassSignal, fs, iteration);
    bpmArray(end+1) = bpm;
    %printf('BPM for cutoff frequency %d: %d\n', cutoffFrequency, bpm)
    ;

    % Apply lowpass filter and then recursively process each half
    lowpassSignal = customFilter(signal, fs, false, cutoffFrequency);

    % Calculate new cutoff frequency for the next level
    cutoffFrequency /= 2;

    % Increase iteration
    iteration += 1;

    % Recursive call for the next level with highpass of lowpass
    signal
    [bpmArray, iteration] = recursiveFilter(lowpassSignal, fs,
    cutoffFrequency, bpmArray, iteration);
end

end
```

selectWAV.m:

```
function wavFilePath = selectWAV()

% Open a GUI dialog to select a WAV file if no arguments are provided
[fileName, pathName] = uigetfile('*.wav', 'Select the WAV file');
if isequal(fileName,0) || isequal(pathName,0)
  disp('User canceled the operation.');
  return;
else
  wavFilePath = fullfile(pathName, fileName);
  disp(['User selected: ', wavFilePath]);
end

end
```

10.2 Appendix B: References

1. Octave Documentation. (2024, March 4). Retrieved from https://docs.octave.org/latest/

2. SongBPM. (2024, March 4). Retrieved from https://songbpm.com/

3. Hills, M. (2024, March 1). BPM-Tools. Retrieved from https://www.pogo.org.uk/ mark/bpm-tools/

4. TIDAL. (2024, February 27). Top 100 Deutschland. Retrieved from https://tidal.com/browse/playlist/56c28e48-5d09-4fb0-812e-c6c7c6647dcf

5. Müller, M. (2024, February 28). Tempogram and Autocorrelation. Retrieved from https://www.audiolabs-erlangen.de/resources/MIR/FMP/C6/C6S2_TempogramAutocorrelation.html

List of Figures

List of Tables