Rhythmic Inside: A Beat Detection Project

Automated Beat Detection and BPM Analysis Framework in Octave/MATLAB

Sven Fuchs, Joshua Reichmann

07.03.2024

Contents

1 Introduction

Beat detection, a cornerstone of digital signal processing in music analysis, plays a pivotal role in a myriad of applications ranging from music production and DJing to fitness and dance. The ability to accurately identify 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

Despite its significance, beat detection remains a challenging problem due to the complexity and diversity of musical compositions. Variations in genre, tempo, and instrumentation, as well as 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. Thus, there is a pressing need for an automated solution that can robustly handle a wide range of music with high accuracy.

1.2 Significance of Beat Detection

Beat detection is fundamental to numerous applications, influencing how we interact with music and sound. In music production and DJing, it facilitates beat matching and mixing, enhancing the listener's experience. In fitness, accurate tempo detection can synchronize music with workout intensities, improving exercise engagement and effectiveness. Furthermore, in the realm of interactive gaming and dance, beat detection enables real-time music interaction, providing a more immersive and enjoyable experience for users.

1.3 Project Objectives and Scope

This project aims to address the challenges inherent in beat detection by developing a sophisticated algorithm capable of accurately and efficiently determining the BPM of any given music track. The objectives include:

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

The scope of this project encompasses the entire process of beat detection, from initial audio file preprocessing to the final BPM estimation, including the development of a comprehensive testing suite to ensure the algorithm's accuracy and reliability across different musical contexts.

In summary, this project seeks to advance the field of beat detection by offering an innovative solution that leverages signal processing techniques and testing methodologies to achieve high accuracy and efficiency. Through this work, we aim to contribute to the broader understanding and application of beat detection in various music-related domains.

2 System Requirements

To ensure the successful execution of the "Rhythmic Inside" beat detection project, the following system requirements must be met:

- **Operating System:** A Unix-compatible shell environment is required for running the scripts and commands associated with this project. This 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:** GNU Octave must be installed on the system. The project is developed and tested with Octave version 8.4.0, but it should be compatible with most recent versions. Octave can be installed from the official package repository of your Linux distribution or from the Octave website for macOS users.

  After installing Octave, ensure that the "signal" package is also installed. This can be done by running the following command in the Octave command line interface:

```
pkg install -forge signal
```

  Listing 1: Installing signal in MATLAB/Octave

  This package is essential for processing the audio signals during beat detection.
- **Audio File Format:** The primary input format supported is WAV files in PCM format. These files provide the uncompressed audio data necessary for accurate beat detection analysis.
- **FLAC to WAV Conversion:** For users with audio files in FLAC format, a script will be provided to convert these files to the required WAV format. This script also trims the audio files to the first 30 seconds to standardize the analysis duration and reduce processing time. Details on using this script will be discussed in the [where though?].

It is recommended to verify that all system requirements are met before proceeding with the installation and execution of the project to ensure optimal performance and reliability.

3 Project Structure and Implementation

The "Rhythmic Inside" project implements a sophisticated beat detection system using MATLAB/Octave, designed to analyze WAV audio files and estimate their Beats Per Minute (BPM). The implementation 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 describes the modular architecture and the role of each component within the system.

3.1 Folder Structure

The project is organized into three main directories, reflecting the separation of source code, documentation, and additional assets:

- **src:** This directory houses all the source code and shell scripts necessary for running 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 orchestrates the execution flow.

- **assets:** Contains graphical resources, sample WAV files for testing, and output PDFs generated from the `docs` directory. This folder aids in both testing and demonstrating the project's capabilities.
- **docs:** Stores the LaTeX source code for the project's documentation. It provides a comprehensive overview of the system, including its design, implementation details, and usage instructions.

3.2 Implementation Details

The source code within the **src** directory follows a modular design, with key components of the beat detection process implemented as separate functions:

- **preprocess.m:** Handles the initial preprocessing of the audio signal, including conversion from stereo to mono and application of a low-pass filter to mitigate high-frequency noise.
- **config.m:** A configuration script that defines global parameters used across various functions, such as filter specifications and threshold values.
- **calculateEnergy.m:** Computes the energy signal from the preprocessed audio, facilitating the detection of rhythmic patterns by emphasizing signal intensity variations.
- **autoCorrelation.m:** Applies an auto-correlation function to the energy signal to identify periodicities indicative of beats.
- **detectPeaks.m:** Analyzes the auto-correlated signal to detect significant peaks, representing potential beats within the audio.
- **calculateBPM.m:** Calculates the BPM based on the intervals between detected peaks, providing a numerical representation of the audio's rhythm.

The main script, **main.m**, integrates these components, orchestrating the entire beat detection process from file input through to BPM estimation. It checks for the necessary dependencies, reads the input WAV file, and sequentially invokes the aforementioned functions to process the signal and output the estimated BPM.

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 enable 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 get started with the "Rhythmic Inside" beat detection system, follow these steps to set up the environment and configure the application according to your needs.

4.1 Cloning the Repository

First, clone the project repository from GitHub and navigate to the `src` directory where the source code is located. Use the following commands in a Unix-compatible shell:

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

Listing 2: Cloning from GitHub

4.2 Editing the Configuration File

The project's behavior can be customized through the `config.m` file, which contains several predefined constants. While the default values are suitable for most cases, you may wish to adjust these settings to accommodate specific audio characteristics or analysis preferences. Open `config.m` in a text editor to modify any of the following variables:

- **CUTOFF_FREQUENCY** (*integer*, Hz): Defines the cutoff frequency for the low-pass filter. Acceptable values depend on the audio content, but it is typically set to 300 Hz to remove high-frequency noise not relevant to the beat detection process.
- **EXPECTED_BPM** (*integer*, BPM): Sets the expected Beats Per Minute, which influences the calculation of minimum samples between beats. This should 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 results in a sharper cutoff but may introduce phase distortion. Values typically 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

Once you have configured the project according to your preferences, you can run the beat detection analysis by executing the `main.m` script within a Unix-compatible shell environment. Navigate to the `src` directory of the project if you haven't already, and then use the following command to start the analysis:

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

Listing 3: Running the project

Replace `[path/to/wav]` with the actual path to the WAV file you wish to analyze. If you do not provide a path to a WAV file, Octave will open 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 based on the configurations set in `config.m`, performs beat detection, and outputs the estimated BPM to the console. This streamlined execution process facilitates quick and 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 pivotal in transforming the raw audio input into a form amenable for beat detection. This section details the preprocessing steps, including filtering and energy calculation, and explicates the rationale behind the choice of signal processing techniques.

5.1 Preprocessing the Audio File

5.1.1 Filtering

The preprocessing commences with applying a low-pass Butterworth filter to attenuate high-frequency components not pertinent to beat detection. This filter is favored for its flat passband, ensuring 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

Following filtration, the signal's energy is computed over short, overlapping windows, transforming the waveform to highlight 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 Discussion on Signal Processing Techniques

These preprocessing steps serve to distill the audio signal, emphasizing rhythmic beats while mitigating irrelevant frequencies and noise. The Butterworth filter's selection, with its characteristic smooth frequency response, ensures that the filtration process preserves the integrity of beat-related frequencies. The subsequent energy calculation and smoothing further refine the signal, setting the stage for effective beat detection by transforming the audio into a representation where beats manifest as discernible energy peaks.

Through these meticulous processing techniques, the project establishes a robust foundation for the reliable identification of beats across diverse music genres, underscoring its utility as a comprehensive beat detection tool.

6 Auto-correlation and Peak Detection

This section elaborates on the critical stages of auto-correlation and peak detection, instrumental in deducing the beats per minute (BPM) from the preprocessed audio signal.

6.1 Auto-correlation to Identify Periodicity

Auto-correlation is deployed to uncover the periodicity within the smoothed energy signal, a pivotal step for rhythm detection. It measures the similarity of the signal with itself at different time lags, highlighting the regular pattern of beats.

- **Implementation:** 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.
- **Algorithm:** The `xcorr` function internally employs a Fast Fourier Transform (FFT) based approach to compute correlations, offering a computationally efficient method to handle large datasets typical in audio processing.

6.2 Peak Detection for BPM Estimation

Upon establishing the periodicity, the project progresses to identify the significant peaks within the auto-correlation function, directly correlating with the beats' timing.

- **Implementation:** 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.
- **Algorithm:** `findpeaks` examines the signal gradient to locate local maxima, with criteria for peak prominence and distance ensuring the detection of distinct, rhythmically relevant peaks.
- **Rationale:** The choice of `findpeaks` and its underlying algorithm is predicated on its adeptness at handling varied signal patterns, enabling the discernment of true rhythmic peaks amidst potential signal irregularities.

6.3 Discussion

The synergy between auto-correlation and peak detection forms the cornerstone of the BPM estimation. Auto-correlation elucidates the signal's inherent periodicity, laying the groundwork for the precise identification of beats through peak detection. The utilization of `xcorr` and `findpeaks`, with their FFT and gradient-based algorithms respectively, underscores the methodological rigor in capturing the essence of rhythm. This approach not only enhances accuracy but also 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 comprehensive testing script is integral to this process, enabling the automated comparison of BPM estimates produced by our algorithm against those generated by the well-regarded external tool, `bpm-tools`.

7.1 Preparation of Test Audio Files

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

7.1.1 Trimming WAV Files

The cutWAVs.sh script trims WAV files to the first 30 seconds, standardizing 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 directory containing your WAV files and execute:

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

Listing 4: Trimming WAV Files

7.1.2 Converting FLAC to WAV

The flac2wav.sh script converts FLAC files to WAV format, preparing them for analysis. This conversion is crucial for maintaining a uniform input format for the beat detection algorithm. To convert FLAC files, execute:

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

Listing 5: Converting FLAC to WAV

7.2 Testing Script Usage

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

```
./testBulk.sh path/to/audio_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, saved as bland_altman_plot.png, provides a graphical representation of the agreement between the two sets of BPM values, highlighting any systematic bias or variability.

7.3 Test Cases and Methodology

The framework was applied to Tidal's Top 100 Germany tracks, offering 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 Integration into Development

The testing script is seamlessly integrated into the development process, enabling continuous validation as enhancements and modifications are made to the algorithm. This integration fosters a development cycle that is iterative and data-driven, ensuring that each modification is supported by empirical evidence of performance improvement.

7.5 Statistical Analysis

Following the execution of the testing suite on Tidal's Top 100 Germany, the results underwent statistical analysis to evaluate the algorithm's performance. The analysis revealed an average BPM of 133.49 from our algorithm, compared to 121.313 from `bpm-tools`, with median values of 140 and 122.846, respectively. The median percent difference stood at 3.96497%, indicating closely aligned performance with minor discrepancies likely due to inherent differences in algorithmic approaches to BPM detection.
This statistical comparison underscores the reliability and accuracy of our algorithm, closely mirroring that of established tools in the majority of test cases. Such analysis validates the algorithm's efficacy and highlights areas for further refinement and optimization.

7.6 Conclusion

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

8 Conclusion

This project represents a significant step forward in the realm of digital signal processing, particularly in the automated detection of beats within music tracks. Through the development of a sophisticated beat detection algorithm, this work has demonstrated the potential to accurately estimate the beats per minute (BPM) of a wide variety of musical genres.

8.1 Achievements

The core achievements of this project include the successful design and implementation of a signal processing pipeline that 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, showing a high degree of accuracy when compared to established tools in the field. Furthermore, the development of a robust testing framework has enabled continuous validation and refinement of the algorithm, ensuring its reliability and effectiveness.

8.2 Challenges Faced

Throughout the project, several challenges were encountered and overcome. These included the handling of diverse musical structures and tempos, which required a versatile approach to signal processing and the customization of filtering techniques. Additionally, optimizing the algorithm for computational efficiency while maintaining high accuracy posed a significant challenge, necessitating a careful balance between complexity and performance.

8.3 Future Work

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

8.4 Closing Remarks

In conclusion, this project has laid a solid foundation for advanced beat detection technologies, offering promising results and identifying clear paths for future enhancements. By addressing the challenges faced and exploring the potential avenues for improvement, there is a significant opportunity to push the boundaries of what is currently possible in beat detection and music analysis.

9 Appendices

9.1 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
  % 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

% 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

% Preprocess the audio file
[filteredSignal] = preprocess(signal, fs);

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

% 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);

% Calculate the BPM
bpm = calculateBPM(locations);

% Display the estimated BPM
fprintf('Estimated BPM: %d\n', round(bpm));
```

preprocess.m:

```
function [filteredSignal] = preprocess(signal, fs)

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

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

% Design and apply a low-pass filter
normalizedCutoff = CUTOFF_FREQUENCY / (fs / 2); % Normalize cutoff frequency to Nyquist frequency
[n, Wn] = buttord(normalizedCutoff, normalizedCutoff * 1.1, PASSBAND_RIPPLE, STOPBAND_ATTENUATION); % Calculate filter order and normalized cutoff
order = min(n, FILTER_ORDER_LIMIT); % Limit filter order to prevent instability
[b, a] = butter(order, Wn); % Design Butterworth filter
filteredSignal = filtfilt(b, a, signal); % Apply filter using zero-phase method

end
```

calculateEnergy.m:

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

% 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.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
    )

% 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.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
```

9.2 Appendix B: References

List of Figures

List of Tables