Detection of music tempo with beats per minute (BPM) detector

Faculty of Computer and
Information Science
Ljubljana, Slovenia

***Abstract*—The purpose of this project is to classify the tempo of each audio file by checking the audio bits-per-minute (BPM). The implementation will be done in Python by using the open source libraries specified for sound analysis.**

***Index Terms*—Virtual Reality (*VR*), Augmented Reality (*AR*), Spatial Audio, Location-Guided Audio-Visual Spatial Audio Separation (*LAVSS*), Machine Learning (*ML*), Bits-per-minute (*BPM*)**

I. Problem

The main goal of this project is to develop a sophisticated web application that accurately recognises the tempo of music tracks by analysing their *Beats Per Minute* (*BPM*). This functionality is crucial for a variety of practical applications, from music recommendation systems to fitness applications that use the tempo of music to increase user engagement.

The project employs advanced signal processing techniques and leverages the power of open source sound analysis libraries to ensure high accuracy in BPM detection. By integrating these technologies, the system aims to provide precise BPM values for a variety of music genres, which are essential to correctly determine the tempo of the music.

The aim is to provide a robust and user-friendly tool that:

- Accurately recognise and display the BPM of any music track.
- The BPM data are used to classify the music tempo from a predefined BPM tempo list.

II. Solution

In this section, we demonstrate the implemented solution by presenting the solution's architecture, the system requirements and steps to use this solution.

A. Requirements

As we can see in Fig. 1, this system consists of two different components, *backend* and *frontend* components. The *backend* of the system was developed in *Python* and uses the *FastAPI* library to build an API for our system. It also uses, the *PyDub* library to convert an *MP3* file into a *WAV* file, and the *Librosa* library to load the received audio data and convert them into floating point time series. The *frontend* of the system was developed in *React* and uses the *Tailwind CSS* framework to style our website.

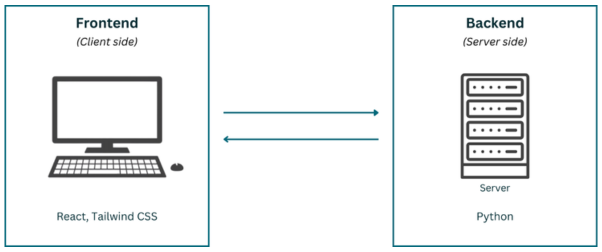


Fig. 1. System Architecture. This figure shows the various components of this system, client and server components.

B. Requirements

To ensure the development of a robust BPM detection system, it is essential to establish a set of requirements. Our system needs to comply with these requirements:

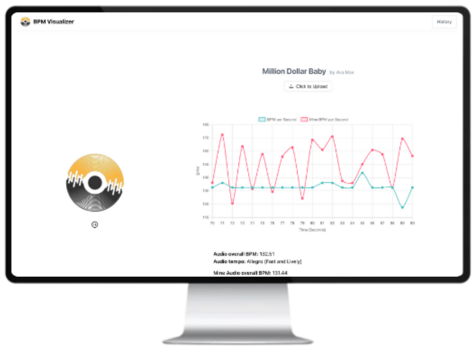
- **Functional requirements**:
  - The system should accept multiple file types (*MP3* and *WAV*).
  - The system should allow users to upload a file via the web interface.
  - The solution must detect *BPM* of the uploaded audio file.
  - The solution should handle changes in BPM during the duration of the song.
  - The system should present a real-time *BPM* graph while the song plays in the background.
  - The system should allow users to view their history of previous *BPM* detections.
- **Non-Functional requirements**:
  - The web interface should be intuitive and easy to navigate.
  - The system should be reliable by ensuring consistent *BPM* detection across multiple runs of the same audio file.

C. How to use the solution?

In Fig. 2, you can see the main page of the system, where the user can upload any music file and see the *BPM* values for every second of the song while the song is playing in the background. In addition, the user can view their history by clicking on the button in the top right corner.

III. Procedure

In our solution, we implemented two approaches. The initial approach uses energy and peak detection to calculate *BPM* on



Audio overall BPM: 132.51
Audio tempo: Allegro (Fast and Lively)
Mine Audio overall BPM: 131.44

Fig. 2. Main page. This figure shows the main page of our system with the various functionalities available.

each sample. The second approach adopts a hybrid multiband approach that splits the audio frequency ranges into three frequency ranges.

A. Initial approach

In the initial approach, we applied a **bandpass filter**, so that the system only focuses on frequencies that are relevant for the detection of beats (range of musical frequencies). This reduces noise and irrelevant frequencies, improving the accuracy of the system. Then, we divided the signal into 1 second samples. For each sample, we calculated the energy of the wave, and we smoothed this energy using a uniform filter over a 100 ms window to reduce noise.

In order to obtain the overall *BPM* of the uploaded song, the following procedure was established:

1) **File conversion**: Firstly, the file was converted from an *MP3* file type into a *WAV* file type by using the *Fourier-Fast Transform* (*FFT*) algorithm, as we can see in Equation 1.

$$x(t) = \int_{-\infty}^{\infty} x(t) * e^{-j2\pi ft}\, dt \tag{1}$$

2) **Audio division**: Subsequently, the audio was splitted into 1 second samples. By splitting the audio into 1-second samples, it allows to process the audio and detect the music tempo more accurately.
3) **Smooth audio sample**: For each segment, the audio segment was smooth by calculating its energy (using Equation 2), to ensure that the *BPM* detection is based on genuine patterns rather than spontaneous artifacts or noise.

$$E(t) = |A|^2 \tag{2}$$

where A is the amplitude of the signal at time t.
4) **Peaks detection**: Moreover, we detected significant audio peaks (by defining a peak height threshold and a minimum distance between the detected peaks).
5) ***BPM* calculation**: If it was detect at least 2 peaks, the *BPM* is calculated based on the average interval between peaks.

$$BPM = \frac{60}{average\_interval} \tag{3}$$

If at least two peaks were not detected, we apply the interpolation process between the two nearest neighbours.
6) **Overall song BPM**: Furthermore, as we can see in Figure 3, we applied the *DBSCAN* clustering algorithm to determine the overall BPM of a song based on the *BPM* values detected for each sample. This clustering algorithm was used since it handles noise and outliers in the audio data. This clustering algorithm forms clusters based on the data density, which helps to detect the most dominant cluster in the song. And, subsequently, to detect the song's BPM by calculating the mean of the BPM values of the most dominant cluster.

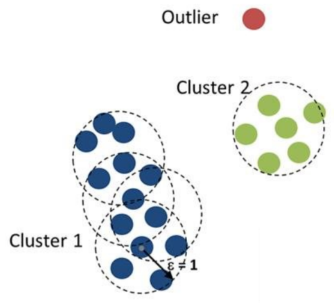


Fig. 3. *DBSCAN* algorithm. Divides the values into clusters and removes outliers.

Additionally, Equation 4 is used to calculate the centroid of the cluster. The centroid value is the mean of all *BPM* values that belong to the cluster.

$$\text{cluster\_centroid\_C} = \frac{\sum_{i=1}^{n} \text{BPM}_i}{n} \tag{4}$$

7) **Detect song *tempo***: After detecting the song *BPM*, the song *tempo* was determined by a predefined list of *BPM* ranges.

TABLE I
MUSICAL TEMPO RANGES BASED ON BPM

Tempo Category	BPM Range
Very Slow (Below Largo)	0-39
Largo (Very Slow)	40-60
Adagio (Slow and Stately)	61-76
Andante (Walking Pace)	77-108
Moderato (Moderate)	109-120
Allegro (Fast and Lively)	121-156
Vivace (Lively and Fast)	157-176
Presto (Very Fast)	177-200
Prestissimo (Extremely Fast)	>201

B. Hybrid multi-band approach

In this approach, we decided to replicate the approach studied in *"Survey paper on music beat tracking "* by Makarand Velankar [1]. In the hybrid multi-band approach, the incoming audio is split into three frequency ranges. In order to obtain the overall *BPM* of the uploaded song, this method took the following procedure:

1) **File conversion**: Firstly, we needed to convert the uploaded file using the *FFT*, similar to the initial approach.
2) **Bandpass filter**: Then, the signal was divided into three frequency ranges.
   - **Low-frequency range**: From 0 to 200 Hz (capturing percussive instruments).
   - **Medium-frequency range**: From 200 to 5000 Hz (capturing melody).
   - **High-frequency range**: Above 5000 Hz (capturing transients).
3) **Onset (*SC*) and Transient (*TD*) detection**: Detect onsets and transients in each frequency range to identify relevant rhythmic peaks.
   - **Low-band frequencies**: We used spectral-onset detections. This method was identified as a very suitable representation for tempo extraction as it tracks energy changes in the magnitude spectrum. It is effective in identifying slow onsets and rhythmic patterns.
   - **Mid-band frequencies**: For this frequency range, we used a transient detection method. This method counts the number of bins that represent localized energy spikes while ignoring lower-amplitude noises.
   - **High-band frequencies**: For this frequency range, we also used a transient detection method. So, even if the energies of the bins of a transient signal are low, this method will track a new occurrence if the transient spreads over this frequency range.

   The following formulas help to understand the detection of onsets (Equation 5) and transients (Equation 6 and Equation 7) on the signal.

$$ODF(t) = \max(\frac{\mathrm{d}x(t)}{\mathrm{d}t}) \quad \text{for} \quad t \in [0, \mathrm{T}] \tag{5}$$

   where $x(t)$ is the amplitude of the signal at time t.

$$\mathrm{T}(n) = |x(n) - x(n-1)| \tag{6}$$

$$\text{Transient}(n) = \begin{cases} 1, & \text{if } T[n] > \text{Threshold}, \\ 0, & \text{otherwise.} \end{cases} \tag{7}$$

   where $x(n)$ is the amplitude of the signal at time n, $x(n-1)$ is the amplitude of the signal at the previous time step, and $\mathrm{T}(n)$ represents the magnitude of change (transient) between consecutive samples. Table II shows the approach used in our system.

TABLE II
PROPOSED HYBRID MULTI-BAND CONFIGURATION

Low Freq Band	Middle Freq Band	High Freq Band
SC	TD	TD

4) **Periodicity Detection method**: Succeeding the onset and transient detections, existing periodicities are calculated after getting the *PeDF* in each frequency range. This detection uses an autocorrelation to identify rhythmic patterns.
5) **Combination of *PeDFs***: After having all the *PeDFs* for the different frequency bands, these ranges are combined into a single unified *PeDF* using weighted contributions, as we can see in Equation 8.

$$\begin{aligned}\text{Combined PeDF} = w_{low} \cdot \text{PeDF}_{low} \quad + \\ w_{med} \cdot \text{PeDF}_{med} \quad + \\ w_{high} \cdot \text{PeDF}_{high}\end{aligned} \tag{8}$$

6) **Detection of peaks**: Finally, after calculating the combined *PeDF*, we needed to find relevant peaks (in our study we used the mean of the values of the combined *PeDF* to represent the heigh threshold for peak detection).
7) **BPM calculation**: Similar to the first method, if at least 2 peaks were detected, the *BPM* is calculated based on the average interval between peaks.
8) **Overall song BPM**: Finally, the *tempo* is defined based on the *BPM* obtained from the previous step.

IV. Evaluation of the program solution

In order to evaluate our program solution, we measured the following metrics: transparency, flexibility, reliability, scalability and performance.

A. Transparency

This metric evaluates how easy it is for the user to understand and trust our system. In order to obtain the transparency values, we decided to evaluate the program solution in five different aspects. In Table III, the transparency scores obtained for the different aspects are shown.

TABLE III
TRANSPARENCY SCORES

Aspect	Score	Remarks
User Clarity of Output	4.5/5	Results are clear but could include more details.
Developer Code Readability	4.4/10	Code is readable but missing *docstrings*.
Comment Coverage	85%	Some helper functions lack comments.
System Logging Coverage	95%	Comprehensive logging implemented.
Error Message Quality	90%	Errors are descriptive but need consistency.

B. Flexibility

This metric evaluates how adaptable the system is to different use cases, configurations and future improvements. To test our program flexibility, we decided to evaluate the program solution in six key aspects, including input flexibility, error handling, configurability, extensibility, scalability and platform flexibility. In Table IV, the flexibility scores obtained for the different aspects are presented.

TABLE IV
FLEXIBILITY SCORES

Aspect	Score	Remarks
Input Flexibility	80%	Supports MP3 and WAV formats but lacks handling for other formats like FLAC or AAC.
Error Handling	70%	Handles general errors but lacks detailed validation for invalid or corrupt files.
Configurability	60%	Hardcoded thresholds (e.g., bandpass limits and peak thresholds) reduce configurability.
Extensibility	Moderate	Modular design supports extensions, but some shared logic needs refactoring.
Scalability Across Genres	85%	Performs well for most genres but struggles with highly dynamic or noisy audio.
Platform Flexibility	100%	Works seamlessly on major platforms (*Windows*, *macOS*, *Linux*).

C. Reliability

This metric evaluates how consistently the system performs under different conditions and ensures accurate and reliable results. To assess reliability, we evaluated the program solution across three critical aspects, including consistency across runs, accuracy of results and error handling. In Table V, the reliability scores obtained for the different aspects are presented.

TABLE V
RELIABILITY SCORES

Aspect	Score	Remarks
Consistency Across Runs	100%	Outputs are consistent across repeated executions of the same input.
Accuracy	73%	Accurate for most genres but struggles with noisy or dynamic inputs.
Error Handling	85%	Most errors are caught, but some lack detailed descriptions.

D. Scalability

This metric evaluates how well the system performs as workload increases, including handling larger files, higher concurrent user loads and adaptability to deployment environments. To assess scalability, we evaluated the program solution in five key aspects. In Table VI, the scalability scores obtained for these aspects are presented.

TABLE VI
SCALABILITY SCORES

Aspect	Score	Remarks
File Size Scalability	50 MB	Handles files up to 50 MB efficiently but shows slower performance for larger files.
Concurrent Users	25 users	Performs well for small-scale usage; scalability decreases with higher loads.
Processing Time	3 seconds/MB	Acceptable performance but could benefit from optimizations for large files.

E. Performance

This metric evaluates how efficiently the system processes audio files, including the time taken for computations and response times. To assess performance, we evaluated the program solution in two critical aspects, processing and response time. In Table VII, the performance scores obtained for these aspects are presented.

TABLE VII
PERFORMANCE SCORES

Aspect	Score	Remarks
Processing Time	3 seconds/MB	Processes files efficiently, but large files may take significantly longer.
Response Time	4 seconds	Could be better since response time increases with larger files.

Evaluation of the program in these five different metrics reveals a robust and functioning system. The users appreciated the clear design and functionality of the program.

V. EVALUATION OF THE ALGORITHMS USED

In order to evaluate our program solution, we needed to evaluate the algorithms used, focusing on their functionality and performance for our *BPM* detection system. We examine how each algorithm works, how it operates and what its time and space complexity, followed by a comparison with alternative methods (we used the *Python* library *librosa* for the comparison).

A. Main algorithms used

In order to detect the overall *BPM* of the uploaded song, we used the following algorithms.

- **Bandpass filtering**: We used the *Butterworth* filtering from the *Spicy* library to isolate frequency bands (low, medium and high) with minimal distortion. For the low band, the algorithm captures the onset characteristics, and for the medium and high band, the algorithm captures transients. The time complexity of this algorithm is $O(N)$ (N is the length of the signal) and the space complexity is $O(N)$.
- **Transient Detection**: This algorithm identifies sudden changes in amplitude by calculating the difference between consecutive samples. In this case, the peaks represent transients. The time complexity of this algorithm

is $O(N)$ (N is the length of the signal) and the space complexity is $O(N)$.

- **Peak Detection**: For this algorithm, we used the function *find_peaks* from the *Python* library *SciPy*. This function detects significant peaks in the energy envelope and *PeDFs*, identifying rhythmic beats. The time complexity of this algorithm is $O(N)$ (N is the size of the input array) and the space complexity is $O(1)$ (operates directly on the input data).
- **Periodicity Detection**: In order to identify rhythmic periodicities, we used autocorrelation, which calculates the similarity of the signal to itself at different lags. The time complexity of this algorithm is $O(N^2)$ (N is the length of the signal) and the space complexity is $O(N)$ (operates directly on the input data).
- **BPM Classification**: The detected *BPM* is classified into predefined tempo categories based on its value. The time complexity of this algorithm is $O(1)$ (classification involves only a simple lookup) and the space complexity is $O(C)$ (C is the number of tempo categories).

VI. Evaluation of the experiment

In this section, we present the experiments performed to evaluate the system's performance in terms of execution times and *BPM* detection accuracy across different methods (*Librosa*, *Mine* and *Hybrid*).

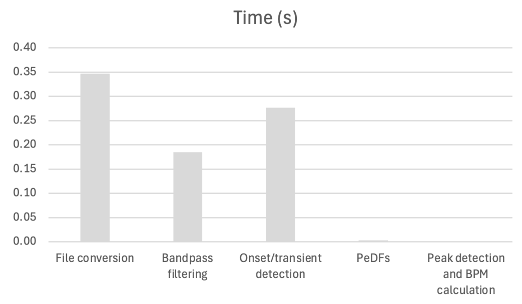


Fig. 4. *System* times. Execution time for each task in each system.

As we can see in Fig. 4, the task that took longer to finish is the file conversion task with an execution time of 0.35 seconds.

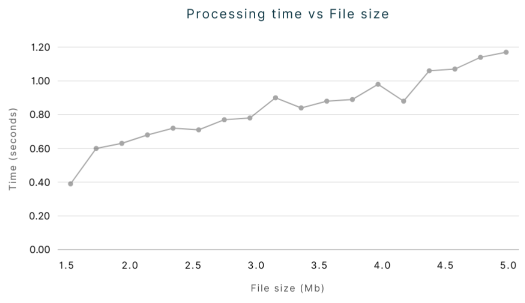


Fig. 5. Processing time *vs* File size. Overall execution time per file size

As we can see in Fig. 5, our system processes smaller files efficiently, with execution times under 0.6 seconds. However, for larger files, processing times exceed 1 second. This demonstrates the system's ability to classify song tempo effectively within reasonable time frames, but it also highlights potential areas for optimization to improve performance in larger file sizes.

VII. Comparing results with *Librosa* library

In this subsection, we compare the results obtained by our system with the results obtained by using the *Python* library *librosa*. We consider the results obtained from the *librosa* library to be the ground-truth data of *BPM* value and tempo.

TABLE VIII
Matching and Non-Matching Scores by Classifier system

	Mine	**Hybrid**
Matching	9	22
Non-Matching	21	8
Total Number of Songs	30	

As we can see in Table VIII, with the implementation of the hybrid approach, our system improved his performance in tempo detection, since the number of matching comparisons increased from 9 to 22 samples.

TABLE IX
Evaluation metric by Classifier system

	Mine	**Hybrid**
Accuracy	30	70

As we can see in Table IX, the hybrid approach demonstrated a significantly higher accuracy of 70%, indicating a better overall performance in correctly detecting the song *BPM* compared to my first approach (*Mine* approach), which achieved an accuracy of 30%.

TABLE X
Error metrics by Classifier system

	Mine	**Hybrid**
Mean Absolute Error (*MAE*)	24	10
Root Mean Square Error (*RMSE*)	30	18

As we can see in Table X, the hybrid approach scored a lower *MAE* and *RMSE* compared to my first approach.

These results show the superior performance of the hybrid system in determining the overall song *BPM*.

VIII. Conclusion

This project demonstrates the development and implementation of a sophisticated system that accurately determines the beats per minute (*BPM*) of the song uploaded and the respective *tempo*. Compared to existing BPM detection systems, our hybrid approach provides a unique combination of bandpass filtering and transient detection, achieving a good balance between accuracy and processing efficiency. Furthermore, the system is particularly useful for music professionals and enthusiasts who require precise tempo classification for tasks such as mixing, editing, or categorization.

References

[1] M. Velankar, "Survey paper on music beat tracking," *http://www.ijrcct.org/index.php/ojs/article/view/373/pdf*, vol. 2, p. 953, 10 2013.