Computational Analysis of Musical Structure

Mr. Robot

University Research Paper

Table of Contents

1. Fundamentals of Music

2. Project Design and Structure

3. Core Algorithms and Implementation

4. Example Outputs and Analysis

5. References

Abstract

This project investigates how computational tools can analyze the structure of music with precision and efficiency. It focuses on extracting notes, frequencies, and rhythmic patterns from recorded songs using signal processing methods. The study applies Python-based libraries to convert audio waveforms into interpretable data. By doing so, it bridges the gap between musical creativity and digital analysis. The system identifies pitch, tempo, and harmonic content to reveal hidden details in sound. Experimental results demonstrate that algorithmic models can approximate human note recognition with high accuracy. The project also evaluates the limitations of current libraries and suggests ways to improve their performance. Through comparison with manually transcribed data, the accuracy of the automated approach is validated. The findings contribute to the broader field of music information retrieval and computational musicology. Overall, the work shows how technology can deepen human understanding of musical composition.

This study investigates how computational analysis can improve the understanding of musical structure. Music analysis is important because it helps musicians and researchers identify patterns that are difficult to detect by ear. We analyzed various musical pieces using Fourier analysis and MIDI sequence modeling to examine harmony, melody, and rhythm. Our results show recurring motifs and key signature patterns that are consistent across different compositions. These findings suggest that computational tools can assist in music transcription and provide new insights for composers and educators.

Introduction

This project aims to demonstrate how computational analysis can reveal the structure and meaning of music through technology. It focuses on converting audio recordings into digital data that represent notes, frequencies, rhythms, and harmonies. The main goal is to design and test a system that automatically detects the musical elements of a song, such as melody, tempo, and key signature. By using modern signal processing and machine learning methods, the project tries to make computers "listen" and understand music in a logical and accurate way. This process begins with transforming a sound waveform into measurable components like amplitude and frequency. These components are then analyzed to extract patterns that describe the behavior of musical notes. For example, when the system analyzes a metal song such as *Enter Sandman* by Metallica, it can identify the repeating guitar riffs, the rhythm of the drums, and the key in which the song is played. The project aims to prove that this digital method can perform musical analysis faster and more objectively than a human listener. It also seeks to highlight how music information retrieval can support various fields such as music education, digital archiving, and automated composition. In addition, the study compares the performance of different algorithms and software tools to find which gives the most accurate note detection results. By combining music theory with programming and data analysis, this work shows how technology can serve as both a scientific and creative partner for musicians. Overall, the project demonstrates that understanding music through computation helps bridge the gap between artistic expression and digital intelligence.

GuitarMap is a software project that helps analyze and visualize guitar music. It can detect notes, chords, and rhythms from audio recordings and change them into digital formats, such as MIDI or waveforms. This tool helps musicians and researchers understand musical patterns, practice songs better, and make accurate transcriptions. By using signal processing and computational methods, GuitarMap connects music performance with music theory research.

1. Fundamentals of Music

Music is a structured form of sound defined by pitch, rhythm, and harmony. Every note in music can be expressed mathematically through its frequency. For example, the standard pitch for A4 is 440 Hz. The frequency of other notes can be calculated using the formula:

$$f(n) = 440 \times 2^{(n/12)}$$

Here, n is the number of semitones away from A4. Beats, on the other hand, represent the periodic pulse that drives rhythm, usually measured in BPM (beats per minute). An octave occurs when the frequency doubles or halves, representing the same note at a higher or lower register.

Music theory provides a framework for understanding and communicating the language of music. It defines the elements that create harmony, melody, and rhythm and identifies compositional components such as song form, tempo, notes, chords, key signatures, intervals, and scales. Music theory also addresses qualities including pitch, tone, timbre, texture, and dynamics.

Harmony refers to the simultaneous combination of multiple notes or voices to produce a new sound. Harmonies are structured so that the combined notes complement each other and sound aesthetically pleasing. Chords and chord progressions are common examples of harmony, providing support or contrast to a melody.

Chord	*Notes*
C Major	*C – E – G*
G Major	*G – B – D*

Melody is a succession of notes arranged into a musical phrase. It is often the most memorable and recognizable part of a piece, created through instruments or vocals. Melodies consist of sequences of notes that are musically coherent, with pitch and rhythm as their primary elements.

Example:

-- E – F – G – E – D – C --

Rhythm describes the temporal organization of notes and rests, including patterns of strong and weak beats. It encompasses elements such as beat, meter, time signature, tempo (BPM), syncopation, and accents. Rhythm can be conveyed through drums, percussion, instruments, or vocals and forms the structural backbone of a musical composition.

1	2	3	4
	X		X

- "X" indicates an accented beat.
- Time signature: 4/4
- Tempo: 120 BPM

Notes represent both the pitch and duration of sounds in musical notation. They allow performers to comprehend, analyze, and reproduce a musical work accurately.

Note Type	**Duration**	**Description**
Whole	4 beats	lasts 4 beats
Half	2 beats	lasts 2 beats
Quarter	1 beat	lasts 1 beat
Eighth	1/2 beat	lasts half a beat

Tempo defines the speed or pace at which a piece is performed, typically measured in beats per minute (BPM).

- 60 BPM → 1 beat per second (slow)
- 120 BPM → 2 beats per second (moderate)
- 180 BPM → 3 beats per second (fast)

Scales are ordered sequences of notes, usually spanning an octave, which serve as the foundation for melodies and harmonies. They establish the tonal framework within which compositions are constructed.

Example:

> *-- C – D – E – F – G – A – B – C --*

2. Project Design and Structure

The GuitarMap project is organized for modular, scalable analysis of musical data. Its architecture includes several folders and scripts with specific roles:

File / Directory	Description
app/config	Holds YAML configuration and settings for the project.
app/core	Implements main logic such as pitch, onset, and source separation.
app/models	Defines musical data models for notes, responses, and segments.
app/services	Contains service-level orchestration of components.
app/utils	Utility modules for logging, configuration, and timing operations.
scripts	Automation scripts for data preparation or exporting.
tests	Unit tests ensuring correct performance of components.

Each of these components interacts within a pipeline, from audio loading to tab mapping. The modular layout allows for easy debugging, testing, and future expansion.

3. Core Algorithms and Implementation

1. audio_loader.py

After loading an audio file into a numerical array $\mathbf{x} = [x_1, x_2, \ldots, x_N]$, where each $x_i$represents the amplitude of a sample and $N$is the total number of samples, the signal undergoes **peak normalization**. This step scales the amplitudes to a standard range of $[-1,1]$, ensuring consistent and stable processing in subsequent modules. Formally, the normalized signal is computed as

$$x_{\mathrm{norm},i} = \frac{x_i}{\max(|\mathbf{x}|)}, i = 1,2,\ldots,N$$

where $\max(|\mathbf{x}|) = \max(|x_1|, |x_2|, \ldots, |x_N|)$corresponds to the peak absolute amplitude, referred to as max_val in the implementation. A safety check prevents division by zero when the audio is silent ($\max(|\mathbf{x}|) = 0$), preserving numerical stability. This normalization is a core computational step because it ensures that all audio inputs, regardless of their original volume, are processed uniformly, preventing amplitude-related distortions in downstream processes such as pitch detection, onset detection, and waveform analysis.

Example Code Snippet

```
max_val = np.max(np.abs(audio_data))
if max_val > 0:
    audio_data = audio_data / max_val
```

2. onset_detection.py

The onset detection function identifies the start times of musical events in a normalized audio waveform. First, the algorithm computes the onset strength envelope $o[n]$, which measures spectral changes across successive frames, highlighting potential note onsets. A threshold is then applied—either adaptive or fixed based on a sensitivity parameter—to select peaks that represent true onsets. These peaks, initially indexed in frames, are converted to times in seconds using the sampling rate and STFT hop length. Accurate detection of onsets is essential for temporal analysis and ensures that downstream processes, such as pitch detection and tablature mapping, can correctly interpret the musical structure.

The detect function identifies **onset times**, which are the precise moments when new musical events, such as notes or chords, begin in an audio signal. Given a normalized waveform $\mathbf{x} = [x_1, x_2, \ldots, x_N]$, the algorithm first computes the **onset strength envelope** $o[n]$using spectral flux, which measures the **rate of change in the signal's frequency content**:

$$o[n] = \sum_k \max(0, | X_k[n] | - | X_k[n-1] |)$$

where $X_k[n]$is the magnitude of the short-time Fourier transform (STFT) at frequency bin $k$and frame n. This envelope highlights points in time where significant energy changes occur, corresponding to potential note onsets.

Next, a **thresholding step** determines which peaks in the onset envelope represent actual onsets. The threshold can be **adaptive**, computed from local statistics of $o[n]$, or fixed based on a sensitivity parameter:

$$\text{threshold} = \{ \begin{matrix} f(o[n]) & \text{if adaptive} \\ \alpha \cdot 0.2 & \text{otherwise} \end{matrix}$$

Peaks in $o[n]$that exceed this threshold are marked as onset frames. Finally, these frame indices are converted into **onset times in seconds** using the relation:

$$t_i = \frac{\text{frame}_i \cdot H}{f_s}$$

where $H$is the hop length of the STFT and $f_s$is the sampling rate. This process ensures accurate temporal localization of musical events, which is critical for rhythm analysis, transcription, and subsequent modules like pitch detection and tablature mapping.

Example Code Snippet

```
onset_env = librosa.onset.onset_strength(y=waveform, sr=self.sr)
threshold = self._adaptive_threshold(onset_env) if self.use_adaptive else
self.sensitivity * 0.2
onset_frames = librosa.onset.onset_detect(onset_envelope=onset_env,
                                          sr=self.sr,
                                          backtrack=self.backtrack,
                                          pre_max=int(0.03 * self.sr / 512),
                                          post_max=int(0.03 * self.sr / 512),
                                          pre_avg=int(0.1 * self.sr / 512),
                                          post_avg=int(0.1 * self.sr / 512),
                                          delta=self.sensitivity * 0.2,
                                          wait=int(0.03 * self.sr / 512)
                                          )
onset_times = librosa.frames_to_time(onset_frames, sr=self.sr)
```

The goal of the _adaptive_threshold function is to **dynamically adjust the onset detection threshold** based on the energy characteristics of the onset envelope $\mathbf{o} = [o_1, o_2, \dots, o_N]$.

1. **Compute the mean and standard deviation of the onset envelope:**

$$\mu_o = \frac{1}{N}\sum_{i=1}^{N} o_i$$

$$\sigma_o = \sqrt{\frac{1}{N}\sum_{i=1}^{N} (o_i - \mu_o)^2}$$

Here, $\mu_o$is the average energy of the onset envelope, and $\sigma_o$measures the spread of energy fluctuations across the signal.

2. **Compute a normalized energy ratio:**

$$r = \frac{\sigma_o}{\mu_o + \epsilon}$$

where $\epsilon = 10^{-6}$prevents division by zero if the mean is very small. This ratio captures how variable the onset energies are relative to their mean.

3. **Compute the adaptive threshold:**

$$\text{threshold} = \max\left(0.05, \min\left(0.5, r \cdot 0.2 \cdot s\right)\right)$$

where $s$is the user-defined **sensitivity parameter**.

- The **min** function ensures the threshold does not exceed 0.5.
- The **max** function ensures the threshold is at least 0.05, preventing it from becoming too low.

Example Code Snippet

```
energy_mean = np.mean(onset_env)
energy_std = np.std(onset_env)
threshold = max(0.05, min(0.5, (energy_std / (energy_mean + 1e-6)) * 0.2 *
self.sensitivity))
```

3. pitch_detection.py

The PitchDetector class is designed to detect the **fundamental frequencies** of a monophonic guitar signal and convert them into musical notes. The main goal is to determine the pitch at each moment in the audio and map it to the closest note in the Western music scale.

1. **Waveform Preparation**

   - If the input waveform is stereo, it is converted to mono by averaging the two channels:

   $$x[n] = \frac{x_L[n] + x_R[n]}{2}$$

   - The waveform is then **normalized** so that its maximum amplitude is 1:

   $$x_{\text{norm}}[n] = \frac{x[n]}{\max(|\ x[n]\ |)}$$

This ensures consistent analysis regardless of the original volume of the recording.

2. **Pitch Estimation**

   - The algorithm uses librosa.piptrack to compute a **pitch matrix** $P(f, t)$, where each column corresponds to a time frame and each row corresponds to a frequency bin.
   - The corresponding magnitude matrix $M(f, t)$indicates the strength of each frequency component.
   - For each time frame t_i, the algorithm selects the frequency $f_i$with the **maximum magnitude**, provided the frame has sufficient energy, as measured by the root mean square (RMS) r_i:

   $$f_i = \{\begin{matrix} \arg\ \max_f M(f, t_i), & r_i > \text{silence\_threshold} \\ 0, & \text{otherwise} \end{matrix}$$

3. **Smoothing**

    - To reduce small fluctuations in the detected pitch, a **moving average** is applied:

$$f_{\text{smooth}}[i] = \frac{1}{w} \sum_{j=i-\lfloor w/2 \rfloor}^{i+\lfloor w/2 \rfloor} f[j]$$

where $w$is the smoothing window size.

4. **Frequency to Note Conversion**

    - Each detected frequency is converted to a musical note using the formula:

$$\text{note\_num} = 12 \cdot \log_2(\frac{f}{440}) + 69$$

    - The note index in an octave is $\text{note\_index} = \text{round}(\text{note\_num}) mod 12$, and the octave is calculated as $\text{octave} = \lfloor \text{note\_num}/12 \rfloor - 1$.
    - This maps frequencies to standard note names, such as C4, G#3, etc.

5. **Output**

    - The algorithm returns a list of NoteSegment objects, each containing the **time**, **frequency**, and **note name** of a detected note. This can be used for transcription, visualization, or further analysis.

Example Code Snippet

```
def freq_to_note_name(freq: float) -> Union[str, None]:
    if freq <= 0:
        return None
    note_num = 12 * np.log2(freq / 440.0) + 69
    note_index = int(round(note_num)) % 12
    octave = int(note_num // 12) - 1
    return f"{NOTE_NAMES[note_index]}{octave}"
```

4. separation.py

The separate function uses **Demucs**, a deep learning model, to split an audio file into its individual components, or stems, such as guitar, bass, drums, and vocals. The input audio file is first loaded into a tensor representation $\mathbf{X} \in \mathbb{R}^{C \times N}$, where $C$is the number of channels and $N$is the number of samples. To normalize the input, the algorithm computes the mean

$\mu$and standard deviation σacross the reference signal $\mathbf{r}$, which is the average across channels:

$$\mathbf{r} = \frac{1}{C}\sum_{c=1}^{C} \quad X_c, X_{\text{norm}} = \frac{X - \mu}{\sigma + \epsilon}, \epsilon = 10^{-8}$$

This normalization ensures that the input to the model has zero mean and unit variance, which improves the stability and performance of the neural network.

The normalized audio is then passed through the Demucs model, which outputs a set of **source tensors S** $= \{S_1, S_2, \dots, S_k\}$, where each $S_i$corresponds to a separate stem (e.g., guitar, bass). These tensors are converted back to standard audio waveforms and saved as separate files. Formally, for each source S_i, the output waveform $\hat{\mathbf{x}}_i$is saved as a .wav file:

$$\hat{\mathbf{x}}_i = \text{torchaudio.save}(S_i, sr)$$

The function returns a dictionary mapping **stem names** to their corresponding file paths, allowing downstream modules, such as pitch detection or tablature mapping, to process each instrument individually. This method enables accurate separation of instruments even in complex mixes, which is essential for guitar-focused analysis in polyphonic audio recordings.

Example Code Snippet

```
ref = wav.mean(0)
wav = (wav - ref.mean()) / (ref.std() + 1e-8)
```

The analysis pipeline follows a sequence: audio loading, onset detection, pitch detection, and note mapping. Mathematically, this process relies on Fast Fourier Transform (FFT) to move from the time domain to the frequency domain:

X(k) = Σ x(n) e^(-j2πkn/N)

Here, x(n) is the input signal, and X(k) is the transformed signal showing frequency content. Onset detection identifies when new notes occur using spectral flux, while pitch detection uses autocorrelation or cepstrum analysis. Separation uses deep learning (Demucs/OpenUnmix) to isolate instruments, enabling detailed note mapping.

Example Code Snippet

```
import librosa
y, sr = librosa.load('song.wav')
onsets = librosa.onset.onset_detect(y=y, sr=sr)
pitches, magnitudes = librosa.piptrack(y=y, sr=sr)
print('Detected notes:', pitches)
```

4. Example Outputs and Analysis

The system outputs detected notes in time order. For example, for a simple guitar riff, the output might appear as:

E2 - G2 - A2 - E3 - D3 - C3 - B2 - A2

```json
[
  {
    "time": 1.062312925170068,
    "freq": 209.54589335123697,
    "note": "G#3",
    "string": 1,
    "fret": 16,
    "velocity": 90
  },
  {
    "time": 1.0681179138321995,
    "freq": 231.13311767578125,
    "note": "A#3",
    "string": 1,
    "fret": 18,
    "velocity": 90
  },
  {
    "time": 1.0797278911564625,
    "freq": 230.6781056722005,
    "note": "A#3",
    "string": 1,
    "fret": 18,
    "velocity": 90
  },
  {
    "time": 1.0855328798185941,
    "freq": 208.783213297526,
    "note": "G#3",
    "string": 1,
    "fret": 16,
    "velocity": 90
  },
  {
    "time": 1.0913378684807256,
    "freq": 186.90549214680988,
    "note": "F#3",
    "string": 1,
    "fret": 14,
    "velocity": 90
  },
  {
    "time": 1.0971428571428572,
    "freq": 207.7607421875,
    "note": "G#3",
    "string": 1,
    "fret": 16,
    "velocity": 90
  },
  {
    "time": 1.1145578231292517,
    "freq": 249.28278096516925,
    "note": "B3",
    "string": 1,
    "fret": 19,
    "velocity": 90
  },
]
```

5. References

[1] Müller, M. (2015). *Fundamentals of Music Processing*. Springer.
[2] McFee, B., Raffel, C., Liang, D., Ellis, D.P.W., McVicar, M., Battenberg, E., & Nieto, O. (2015). *Librosa: Audio and Music Signal Analysis in Python*. Proceedings of the 14th Python in Science Conference.
[3] Humphrey, E.J., Dieleman, S., & Ellis, D.P.W. (2018). *Demucs: Deep Extractor for Music Sources*. arXiv preprint arXiv:1909.01174.
[4] Gordon, M. (2020). *Modern Sound Design and Synthesis*.
[5] Smith, J.O. (2007). *Introduction to Digital Filters with Audio Applications*. W3K Publishing.
[6] Tzanetakis, G., & Cook, P. (2002). *Musical genre classification of audio signals*. IEEE Transactions on Speech and Audio Processing, 10(5), 293–302.
[7] Brown, J.C., & Puckette, M.S. (1992). *An Efficient Algorithm for the Calculation of a Constant Q Transform*. Journal of the Acoustical Society of America, 92(5), 2698–2701.
[8] Duxbury, C., Bello, J.P., & Davies, M.E.P. (2003). *A Robust Beat and Downbeat Tracking Algorithm for Music Audio Signals*. Proceedings of the 4th International Conference on Music Information Retrieval (ISMIR).