BPM & Time-Signature Detection Toolkit (Python)

This toolkit estimates **tempo (BPM)** and **time signature** from an audio MP3 file. The project is organized as small, composable modules and a single command-line entry point.

High-level pipeline

At a high level, the workflow is:

1. **Audio loading and preprocessing** (mono conversion and resampling to a target sample rate).
2. **BPM estimation** using one of two onset-driven pipelines:
   - a more robust "normal" mode, or
   - a faster "light" mode.
3. **Time-signature detection** using the BPM estimate:
   - `standard` mode produces one global meter label,
   - `varying` mode produces a time-aligned sequence of meter segments.

Modules overview

`beat_tracking_unified.py` (BPM estimation)

This module provides a single BPM interface with two modes:

- **Normal mode** (beat_tracking-style):
  - Computes an onset-strength (or novelty) curve from time–frequency energy changes.
  - Estimates the tempo by analyzing periodicities in the onset curve (autocorrelation / Fourier tempogram-style reasoning).
  - Optionally refines tempo by emphasizing stable periodic peaks and rejecting spurious ones.
  - Designed to be more reliable across a wide range of music, especially when attacks are clear.
- **Light mode** (optlibrosa lightweight-style):
  - Uses a more compact feature representation (typically mel/energy bands or a reduced-resolution STFT).
  - Derives a simpler onset envelope and performs a reduced-cost tempo search.
  - Trades some accuracy for speed and lower compute, useful for quick previews or low-power environments.

Both modes ultimately return a single BPM value. When the input has tempo drift, the returned BPM should be interpreted as an **average** over the analyzed excerpt.

`time_signature.py` (standard time signature detection)

This module estimates a **single** dominant time signature for a track (or excerpt), assuming the meter is mostly stable.

The detector works as follows:

1. **Beat-synchronous time grid.** Given an estimated BPM, it constructs a very fine-grained time–frequency representation with a window of 1/32 beat and a hop of 1/64 beat. This expresses time in units of “frames per beat” and allows testing candidate bar lengths at sub-beat resolution.
2. **Leading silence / noise trimming.** A simple first-note detector identifies the first frame that looks like “music onset” (as opposed to silence/noise), so the subsequent similarity analysis is not biased by empty leading frames.
3. **Audio Similarity Matrix (ASM).** The algorithm builds a frame-to-frame similarity matrix over the excerpt. Each matrix entry measures similarity between two spectrogram frames (distance in feature space mapped into a similarity value). Repeated musical structure (e.g., bars and phrases) creates recurring patterns in this matrix.
4. **Multi-resolution bar-length scoring.** Candidate bar lengths corresponding to 2–12 beats are evaluated. For a given candidate length, the method examines ASM diagonals offset by multiples of that bar length (Bar, 2·Bar, 3·Bar, …). Each diagonal is split into full bar-sized blocks and a remaining tail. Block RMS values are aggregated into a single similarity score **SM** for that candidate. Intuitively: if the candidate matches the true bar length, repeated bars align and produce consistently high diagonal similarity, raising the score.
5. **Label mapping.** The best beats-per-bar is rounded to an integer and mapped to a conventional label (e.g., 4/4, 3/4, 6/8, 7/8, …).

The output is a single time signature.

`new_time_signature.py` (varying time signatures over time)

This module extends the standard detector to handle **meter changes** and **local tempo variation** by applying the same bar-length scoring idea in a sliding-window fashion.

1. **Compute the beat-synchronous representation once.** As in the standard detector, it builds a beat-synchronous spectrogram. This ensures that windowing can be expressed naturally in beats.
2. **Sliding window analysis.** The track is split into overlapping windows measured in beats (for example 32-beat windows with an 8-beat hop). For each window:
   - a local similarity matrix is computed,
   - candidate beats-per-bar (2–12) are scored,
   - the best candidate and its score distribution are stored.
3. **Temporal smoothing (Viterbi-style decoding).** Window-by-window estimates can jitter (e.g., flip between 4 and 8, or 3 and 6). To obtain stable segments, the module applies a smoothing procedure that:
   - treats each beats-per-bar value as a “state”,
   - uses the per-window meter scores as emission strengths,
   - adds a penalty for switching states between adjacent windows.

The resulting sequence favors consistent meters unless there is strong evidence for a change.

4. **Segment extraction.** The smoothed state sequence is converted into contiguous time segments and reported as: start time, end time, and a time signature label.

The output is therefore **piecewise-constant meter**, suitable for songs with distinct sections in different meters.

`bpm_ts_cli.py` (command-line entry point)

This script connects BPM estimation to time signature estimation:

1. Load audio.
2. Estimate BPM (normal or light).
3. Run time signature (standard or varying) using the BPM.
4. Print results and optionally plot.

The plotting option shows:

- `standard` mode: the bar-length score curve versus candidate beats-per-bar.
- `varying` mode: beats-per-bar over time (raw window estimates and the smoothed sequence).

Installation and Usage

Python dependencies:

```
pip install numpy scipy matplotlib soundfile librosa audioread
```

Usage example:

```
python bpm_ts_cli.py INPUT_AUDIO --bpm-mode normal --ts-mode standard --plot
```