Exercise Track
Lepatriinu

Audio and Music Processing

June 23, 2013

Wolfgang Küllinger (0955711)
Fabian Jordan (0855941)

Contents

1 Architecture **3**

1.1 `Analyzer` . . . 4

1.2 `OnsetDetector` . . . 4

1.2.1 `SimpleOnsetDetector` . . . 5

1.2.2 `SpectralFluxOnsetDetector` . . . 5

1.2.3 `HighFrequencyOnsetDetector` . . . 5

1.2.4 `GroundTruthPicker` . . . 5

1.3 `TempoExtractor` . . . 5

1.3.1 `InterOnsetTempoExtractor` . . . 5

1.4 `BeatDetector` . . . 6

1.4.1 `AutoCorrelationBeatDetector` . . . 6

1.4.2 Best results . . . 6

2 User interface **7**

2.1 Purpose . . . 7

2.2 Precaustionary Measures . . . 7

2.3 Structure . . . 7

2.3.1 Files . . . 7

2.3.2 Outputs . . . 7

1 Architecture

The project was separated into several classes. The given framework simple calls methods which are defined in the package `at.jku.amp.lepatriinu`.

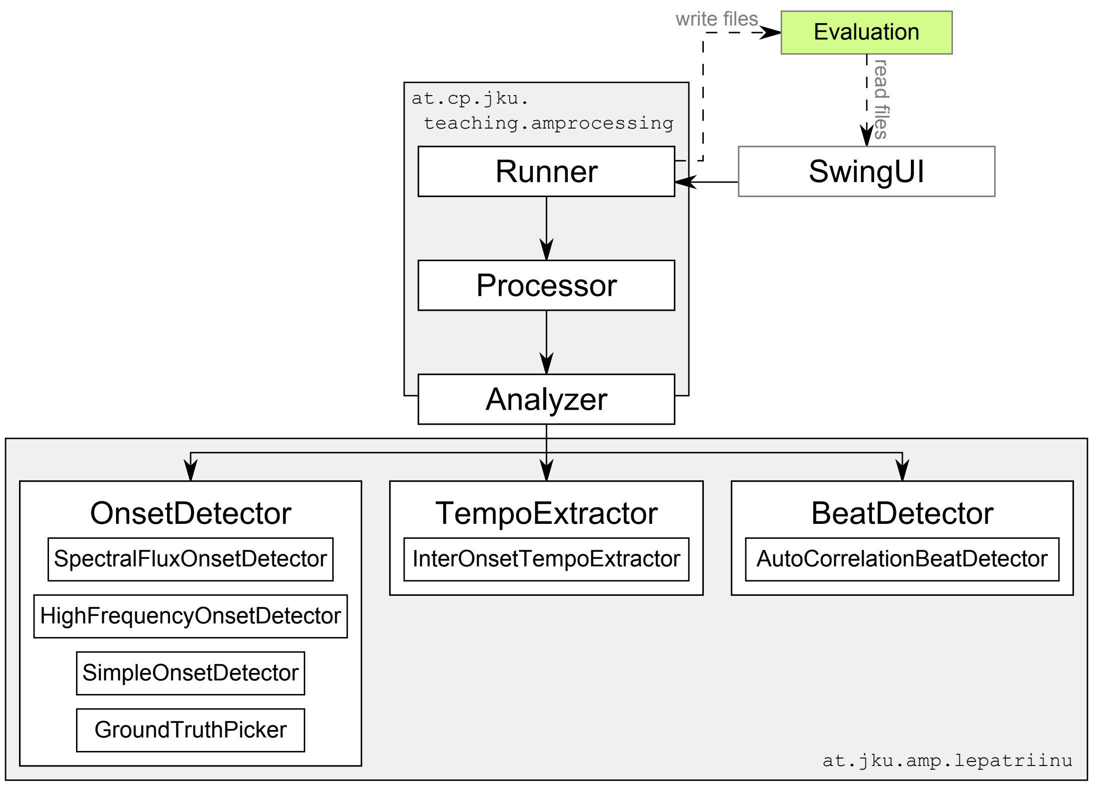

Figure 1.1: Class diagram

1.1 `Analyzer`

The *Analyzer* class defines all the constants that are needed for any algorithm in the project to succeed. In order to make central configurations possible, the constants are collected in this single class.

Furthermore it provides the central interface to the `at.cp.jku.teaching.amprocessing` project. It is initialized with a pre-processed (e.g. FFT) audio file of type `Audiofile`. The order of usage is important. In the first place, onset detection can be done. This information can directly be retrieved from the audio file. The found onsets are the base for tempo extraction. Therefore the onset list has to be provided as parameter of the tempo extraction function. Finally beat detection can be performed. In order to be able to use the best algorithms both the onset list as well as the calculated tempo should be provided.

1. Onset detection: `onsets = performOnsetDetection()`
2. Tempo extraction: `tempo = performTempoExtraction(onsets)`
3. Beat Detection: `beats = performBeatDetection(onsets, tempo)`

1.2 `OnsetDetector`

OnsetDetector is the abstract superclass of all onset detection algorithm's classes. It provides two different peak picking methods which can be used by any sub class.

1. The first peak picking method is taken from lecture slides (5.40) and called *Adaptive Thresholding.*
2. The second one is a slight modification of the adaptive thresholding algorithm. Instead of a threshold it simply isolates the highest magnitudes by zeroing everything but the peak as well as close neighbors that are lower. As a result a clear list of onsets remains. We called this method *mountain climbing.*

Constants

- `USE_MOUNTAIN_PEAKPICK` defines which peakpicking method to use. If constant is `true`, *mountain peakpicking* is chosen. Otherwise *adaptive thresholding* is performed.
- `THRESHOLD_RANGE` is used to specify the `int` size of the shifted slice used in *adaptive thresholding* as well as the range of zoroed neighbors in *mountain climbing.* The

best results were produced by a range of `5`.

- `PEAKPICK_USE_MEAN` defines whether to use mean (`true`) or median (`false`) in order to select the threshold that is later on applied to all the onset calculations of *adaptive thresholding.*
- `THRESHOLD` is used to define a fixed `int` threshold for choosing the magnitude peaks when using *mountain climbing.* The value that prouced the best results in our experiments was `13`.

1.2.1 `SimpleOnsetDetector`

1.2.2 `SpectralFluxOnsetDetector`

1.2.3 `HighFrequencyOnsetDetector`

1.2.4 `GroundTruthPicker`

1.3 `TempoExtractor`

TempoExtractor is the abstract superclass of which all tempo extractors should be derived.

1.3.1 `InterOnsetTempoExtractor`

As the name suggests, *InterOnsetTempoExtractor* performs the Inter Onset Intervals as described in the lecture slides 5.10 through 5.14. The method calculates the most common gap size between the onsets inside a set of onsets.

Because of the inaccuracy that evolves when adding vague `double` values and the necessarity to provide some sort of categorization and clusterization an indirectional mapping from the `double` value of the gap size to the `int` value of the number of occurencies was established. In order to reach this goal actually two mappings are done:

1. `double` $\rightarrow$ `int`: specified by the constant `TEMPO_KEY_TOLERANCE`, given distances are separated into different categories which are consecutively numbered using the `int` values. To be more precise, if `distance` $\pm$ `TEMPO_KEY_TOLERANCE` meets one key in the map, it is put into the same category. Otherwise it is put into a new one.

2. `int` $\rightarrow$ `int`: The *value* of the mapping in 1 is the *key* of the second mapping, which provides the number of occurencies.

Of course this is to be done if the distance is within the range of usual tempi. Therefore distances that are smaller than `MIN_TEMPO` or greater than `MAX_TEMPO` are not taken into account.

The tempo then is returned in *beats per minute* (bpm).

1.4 `BeatDetector`

Also *BeatDetector* is an abstract class, thats purpose is to be the superclass of all beat detection algorithms.

1.4.1 `AutoCorrelationBeatDetector`

This subclass implements beat detection based on auto correlation. It accommodates an enumeration, named *Mode*, which allows to switch between three different ideas and implementations:

1. `AUTO_TEMPO_CORRELATION` simply implements the idea of the pulse train, described on slides 5.15 through 5.19. To tackle the inaccuracy the list of onsets will always carry along we use a threshold, called `AUTO_PHASE_TOLERANCE` which, after excessive testing we found best to be set around 0.21 and pushed it up to 48% (average over all 18 provided data sets).
2. `AUTO_ONSET_CORRELATION` represents the implementation of the IOI idea presented on lecture slides 5.10 through 5.14.
3. `AUTO_SPECTRAL_CORRELATION` was the attempt to improve the onset correlation by trying to reinterprete the formulas as to be the spectral data instead of the onsets. It didn't work out that well (less than 4% over 18 data sets).

1.4.2 Best results

As onset correlation did a rather poor job and spectral correlation an even worse, we stuck with tempo correlation as the algorithm of choice.

2 User interface

2.1 Purpose

As the pure command line interface provided too few possibilities to see the overall performance of our algorithms and lacks a simple way to execute batches of runs, we decided to develop a small GUI in Swing to match these two requirements.

2.2 Precaustionary Measures

In order to prevent the program from shutting down at any occuring error we had to edit the `Runner` a little bit and replace the `System.exit(0)` calls for simple runtime exceptions.

2.3 Structure

2.3.1 Files

Starting the program will allow you to select one or multiple files, which should be located in the "`./data/`" folder. On the bottom of the window three checkboxes let you choose, whether the output files should be generated. The "Run" buttons on the top are pretty self-explanatory.

2.3.2 Outputs

The outputs are displayed seperately for Onsets, Tempo and Beat, to keep a straight interface. Each of the three output tabs has the same layout:

- In the upper half, each output file will be displayed. For onsets and beat, only the respective "`*.eval`" files appear, whereas for the tempo the "`*.bpms`" as well as the "`*.bpms.eval`" file.

- In the bottom half, the selected file will be displayed as well as the summary of this set of evaluations. To make differences easy to spot at first look, the summaries are color coded pretty straight forward: The greener, the better.