Revista de Informática Teórica e Aplicada - RITA - ISSN 2175-2745
Vol. 27, Num. 04 (2020) 127-138

RESEARCH ARTICLE

Iracema: a Python library for audio content analysis

Iracema: biblioteca Python para a análise de conteúdo em áudio

Tairone N. Magalhães[1]*, Felippe B. Barros[1], Mauricio A. Loureiro[1]

Abstract: *Iracema* is a Python library that aims to provide models for the extraction of meaningful information from recordings of monophonic pieces of music, for purposes of research in music performance. With this objective in mind, we propose an architecture that will provide to users an abstraction level that simplifies the manipulation of different kinds of time series, as well as the extraction of segments from them. In this paper we: (1) introduce some key concepts at the core of the proposed architecture; (2) describe the current functionalities of the package; (3) give some examples of the application programming interface; and (4) give some brief examples of audio analysis using the system.
Keywords: Music Expressiveness — Music Information Retrieval — Software Systems and Languages for Sound and Music

Resumo: *Iracema* é uma biblioteca Python que visa fornecer modelos para a extração de informações de gravações de peças musicais monofônicas, para fins de pesquisa em performance musical. Com este objetivo em mente, nós propomos uma arquitetura que fornece aos usuários um nível de abstração que simplifica a manipulação de diferentes tipos de séries temporais, bem como a extração de segmentos destas. Nesse paper nós: (1) introduzimos alguns conceitos essenciais do núcleo da arquitetura proposta; (2) descrevemos as funcionalidades atuais do pacote; (3) apresentamos alguns exemplos da interface de programação do aplicativo; e (4) oferecemos breve exemplos de análise de áudio utilizando o sistema.
Palavras-Chave: Expressividade musical — Recuperação de informações musicais — Sistemas de Software e Linguagens para Som e Música

[1] *Universidade Federal de Minas Gerais, Belo Horizonte - UFMG, MG, Brazil*
*__Corresponding author__: tairone@ufmg.br
DOI: http://dx.doi.org/10.22456/2175-2745.107202 • **Received:** 02/09/2020 • **Accepted:** 29/11/2020

1. Introduction

Iracema is a Python package for audio content analysis aimed at the empirical research on music performance. It aims to provide to researchers in the field of music performance analysis tools for extracting patterns of manipulation of duration, energy, and spectral content from monophonic audio, especially for instruments such as clarinet, flute, trumpet, and trombone. Its development was motivated by research projects conducted at CEGeME[1], and was strongly inspired by a previous Matlab tool developed by the group, called Expan [1], which was not released for public use. Iracema is licensed under the GNU General Public License v3.0, and its source code can be freely obtained at ⟨https://github.com/cegeme/iracema⟩. To obtain more detailed information about the library, like usage examples, more information about the feature extractors available, library modules, and extensive documentation of the API, check the online documentation, which is available at ⟨https://cegeme.github.io/iracema⟩.

Iracema uses NumPy arrays to store and manipulate data, providing a new level of abstraction on top of such objects [2]. It also wraps some functionalities from SciPy [3] to provide a more natural interface for audio content extraction.

1.1 Empirical study of Music Performance

People frequently choose to listen to a piece of music not just because of the composition itself, but for the different possibilities of rendition of the piece. Sometimes they seek a specific interpretation by their favorite musician or band, maybe from one particular concert or recording session. That is a consequence of the fact that every music performance is unique[2]. There is a significant amount of information in a performance that does not depend only on the underlying structure of the piece, but instead, on several other factors, like individual playing style, performance traditions, the mood of the performer, or expressive choices made by him. Other factors are the environment where the performance took place, the instrument played, random manipulations that might occur

[1] ⟨http://musica.ufmg.br/cegeme/⟩

by chance, etc. In the face of such idiosyncrasies, listeners establish their individual preferences, often favoring a specific rendition of a piece as more compelling, for some reason, than others. This scenario gives rise to many questions regarding the complex process by which a performer shapes a musical composition into its actual rendition.

Despite the fact that music performance is an intrinsic element to nearly every culture, its empirical study is relatively recent, with the seminal works dating back to the turn of the twentieth century. As stated by 5, p. 77, "only once methods had been developed to record either the sounds of performance, or the actions of instruments, was any kind of detailed [empirical] study possible — and so the piano roll, record, magnetic tape, and computer have all played their part at different stages in the short history of empirical studies of performance". Undoubtedly, the twentieth century's technological developments progressively extended our capacity to measure relevant information from the performance. 6, 7, 8, and 9 provide excellent surveys on the topic that indicate the field's large growth in the last few decades. The availability of new tools and technologies that enable the extraction of information from performances have played a pivotal role in this surge. The contributions in audio analysis by researchers from the Music Information Retrieval community were greatly beneficial to the field, as 9, p. 1-2 points out, even though most of them were not motivated by the interest in conducting research in music performance, but by other tasks, such as retrieving information from large music databases.

Since data acquisition is a fundamental step in the study of music performance, we believe that the continuous development of purpose-oriented and specialized tools to extract information from audio recordings of performances will be of crucial importance for the research on the field. Equally important will be the development of better techniques for obtaining more meaningful representations of musical content, like higher-level descriptors of musical expressiveness in the performance, for example.

1.2 Monophonic instruments

In many musical instruments, the excitation that produces sound happens only during a short interval at the beginning of a note (i.e., the plucking of strings in a guitar or a hammer hitting the strings of a piano). Contrastingly, in woodwind and brass instruments, the player continuously feeds energy into the system, employing high-pressure air from his lungs. Therefore, due to the dynamic control that the player has over the instrument's acoustic properties, a single note will contain over its duration a substantial amount of expressive information, e.g., timbral manipulations or dynamic intensity variations. It is harder to extract this kind of information from polyphonic music signals, such as a full orchestral recording, than from signals of a single source, especially when one cannot afford to lose any relevant expressive information pertaining to individual instruments/performers. For example, to analyze the timbral manipulations that a single clarinetist performs in some specific notes on a full orchestral recording, it would be necessary to isolate the information pertinent to that single clarinet. This approach would probably depend on a highly sophisticated source separation algorithm, which would have to be absolutely precise in retrieving only the data belonging to that clarinetist from a highly complex stream containing all the other instruments. This level of precision is still unfeasible, taking into account that any loss would impact the subsequent timbral analysis.

While there are several examples of scientific works that use polyphonic recordings as a source of investigation [10, 11, 12], for the reasons aforementioned, Iracema's scope has been intentionally limited to analyzing monophonic recordings[3].

2. Architecture

This section will discuss some aspects of Iracema's architecture and offer an overview of the elements that compose the main functionalities of the library.

Audio content analysis systems rely on the manipulation of dynamic data, i.e., data that represent an attribute's changes over time. Thus, the *time series* is a fundamental element in Iracema's architecture. The starting point for any task performed by the system is the *audio time series*, from which other kinds of time-related data will be extracted. The transformation of time series into other time series to obtain more meaningful representations of the underlying audio is a common behavior of audio content analysis systems, usually called *feature extraction*. The implementation of such extractors usually depends on some recurrent types of operations, like applying sliding windows to a series of data, for example. In Iracema, these operations are called *aggregation* methods.

Sometimes it will be necessary to deal with a specific excerpt of a time series, such as a musical phrase or a note. Another important element in the architecture, called *segment*, can be used to delimit such excerpts. A user may sometimes label the start and end of such segments manually; however, most of the time, users will expect the system to identify such limits by itself, a common kind of task in audio content extraction, known as *segmentation*. Since segments are delimited by two *points*, this is another important element in the architecture.

Some aforementioned elements, like audio, time series, points, and segments were implemented as classes, since they have intrinsic attributes (e.g., the samples of the time series, and the start/end of the segments) and behavior (e.g., generating time vectors in time series or calculating indexes

[2] 4, p. 239 mentions that "[n]o two performances of the same work are exactly alike, and this is often true even for repeated renditions of the same piece."

[3] Notice that the scope is limited to monophonic recordings, but it is still possible to study polyphonic compositions, as long as the individual voices that constitute are recorded and analyzed separately. Although this simplification could be questioned from the perspective of gestalt, or criticized for relying on sonic material that lacks the context of a real performance situation, it enables a more precise investigation of attributes related to individual instruments/performers.

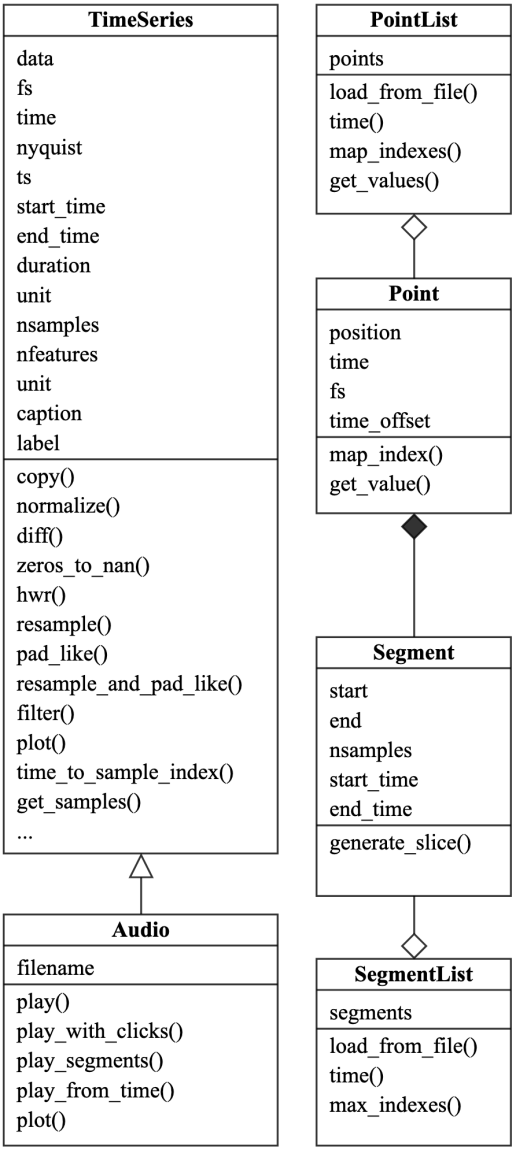

Figure 1. Diagram showing the core classes of Iracema.

in segments). Figure 1 shows those classes in a diagram. The class `Audio` inherits the functionalities from `TimeSeries`, and add some specific behaviors (such as loading wave files). The classes `Point`, `Segment`, `PointList` and `SegmentList` provide a handy way to extract corresponding excerpts from time series of different sampling rates, since it performs all the necessary index conversion operations to extract data that coincide with the same time interval.

Other elements have been implemented as methods that take objects of those classes as input and output another object. For example, the method `fft` takes as input an `audio` object, a `window_size`, and a `hop_size`, and generates a time series in which each sample contains all the bins of the FFT (Fast Fourier Transform) for the interval corresponding to `hop_size`. Another example, the method `spectral_flux` will take a time series containing the result of an FFT operation as input and generate another time series containing the calculated spectral flux. Figure 2 shows a diagram that illustrates the typical workflow for performing basic feature extraction from audio files.

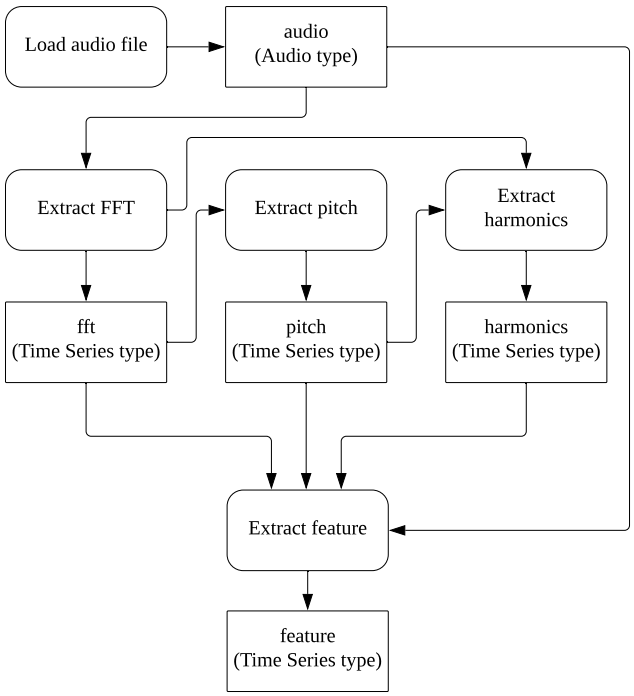

Figure 2. Extracting features from an audio file.

Segmentation methods will usually take `time_series` objects as input to output a list of segments (Figure 3). Then, these segments can be used to extract excerpts from time series objects easily (Figure 4), using square brackets (the same operator used in Python to perform indexing/slicing operations).

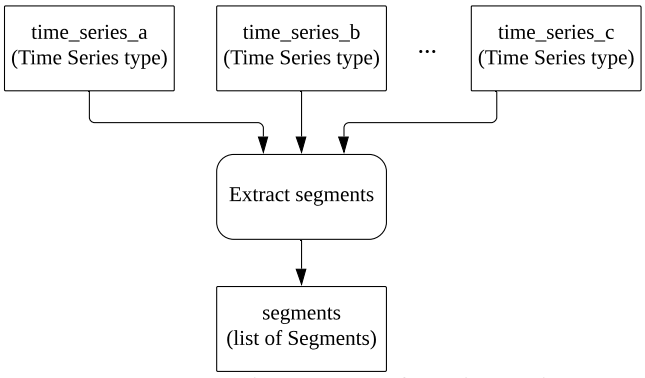

Figure 3. Extracting segments from time series.

3. Modules and functionalities

These are the modules that compose Iracema, and their respective functionalities:

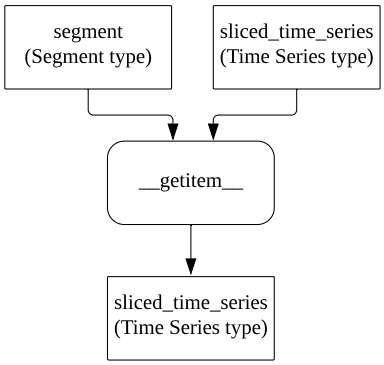

Figure 4. Using a segment to slice a time series.

- `core.timeseries`: contains the definition of the class `TimeSeries`;
- `core.audio`: contains the definition of the class `Audio`.
- `core.segment`: contains the definition of the classes `Segment` and `SegmentList`.
- `core.point`: contains the definition of the classes `Point` and `PointList`.
- `spectral`: contains methods for frequency domain analysis (currently the FFT);
- `pitch`: a few different models for pitch detection.
- `harmonics`: a model for extracting harmonic components from audio.
- `features`: contains methods with the implementation of several classic feature extractors.
- `segmentation`: methods for automatic audio segmentation.
- `plot`: contains several different methods for plotting time series data.
- `aggregation`: contains some common aggregation methods that can be useful for implementing feature extractors.
- `io`: subpackage containing IO methods, for loading / writing files, playing audio, etc.
- `util`: subpackage containing some useful modules for unit conversion, DSP, windowing operations,etc.

3.1 Core modules

Contains modules that implement the core classes of the library. To make these classes easily available for users, they are all already imported into the namespace `iracema`. For instance, the class `Audio` is available in the module `iracema.core.audio`, but it can be accessed using the shorter name `iracema.Audio`.

3.2 Pitch detection

The module `pitch` contains pitch detection methods. At the time this paper was finished, two methods had been implemented, as well as an extra method that wraps a model from an external library.

3.2.1 Harmonic Product Spectrum

Measures the maximum coincidence for harmonics, based on successive down sampling operations on the frequency spectrum of the signal [13]. It is based on successive downsampling operations on the frequency spectrum of the signal. If the signal contains harmonic components, then it should contain energy in the frequency positions corresponding to the integer multiples of the fundamental frequency. So by down-sampling the spectrum by increasing integer factors $(1,2,3,...,R)$ it is possible to align the energy of its harmonic components with the fundamental frequency of the signal.

Then the original magnitudes of the spectrum are multiplied by its downsampled versions. This operation will make a strong peak appear in a position that corresponds to the fundamental frequency. The HPS calculates the maximum coincidence for harmonics, according to the equation:

$$Y(\omega) = \prod_{r=1}^{R} |X(\omega r)| \tag{1}$$

where $X(\omega r)$ represents one spectral frame and R is the number of harmonics to be considered in the calculation. After this calculation a simple peak detection algorithm is used to obtain the fundamental frequency of the frame.

This implementation modifies this approach adding an offset of 1 to the magnitude spectrum of the signal before applying the product shown in the equation above. This makes the algorithm more reliable in situations where some harmonics have very little or no energy at all. Also, alternatively to the original approach, it is possible to choose between different interpolation methods, using the argument 'decimation'.

3.2.2 Expan Pitch

Based on the algorithm implemented in Expan [1]. The method consists in choosing the n highest local peaks in the spectrum of a signal (above a certain minimum relative threshold) as potential candidates, and then calculating the expected position of their theoretical harmonics. A local search for peaks is performed around this theoretical position, within a tolerance interval. This step is important to account for some inharmonicity, which is quite frequent in higher harmonic components. After performing this procedure for all the candidates, their harmonic energies are calculated. The candidate with the highest harmonic energy is chosen as the pitch values.

3.2.3 CREPE

This method is based on a deep convolutional neural network operating directly on the time-domain waveform [14]. It was developed by the Music and Audio Research Laboratory, at

the New York University, and is available in iracema as a wrapper over their published Python package. It uses six convolutional layers connected to a densely connected output layer. The output produced by this layer is a 360-dimensional vector, from which the pitch estimate is calculated deterministically. The frequencies that correspond to the output follow a logarithmic scale, and cover six octaves with 20-cents intervals. The model was trained on two large datasets containing synthesized audio, for which it is possible to have very precise target annotations. The method produces excelent pitch estimations (according to the authors, state-of-the-art as of 2018), but it is highly demanding in terms of computational power when set to its maximum capacity. Lowering the capacity of the model (it is one of its input arguments) it is possible to find compromise between accuracy and processing time.

3.3 Feature extractors

These are the methods available in the module `features`:

3.3.1 Peak Envelope

Extracts the envelope of the waveform by extracting the peaks in the amplitude for each analysis window.

$$\mathrm{PE} = max(|x(n)|), 1 <= n <= L \tag{2}$$

Where L is the length of the window.

3.3.2 RMS

Calculate the root-mean-square of a time-series. This is usually a better choice for extracting the envelope of an audio signal, since it is more closely related to our perception of intensity that the peak envelope.

$$\mathrm{RMS} = \sqrt{\frac{1}{L}\sum_{n=1}^{L} x(n)^2} \tag{3}$$

Where $x(n)$ is the n-th sample of a window of length L.

3.3.3 Zero-crossing

The zero crossing is a measure of how many time-series a signal crosses the zero axis in one second. It gives some insight on the noisiness character of a sound. In noisy / unvoiced signals, the zero-crossing rate tends to reach higher values than in periodic / voiced signals.

$$\mathrm{ZC} = \frac{1}{2L}\sum_{n=1}^{L} |\mathrm{sgn}\,[x(n)] - \mathrm{sgn}\,[x(n-1)]| \tag{4}$$

Where

$$\mathrm{sgn}\,[x(n)] = \begin{cases} 1, & x(n) \geq 0 \\ -1, & x(n) < 0 \end{cases} \tag{5}$$

3.3.4 Spectral Flatness

Gives an estimation of the noisiness / sinusoidality of an audio signal. It might be used to determine voiced / unvoiced parts of a signal [15]. It is defined as the ratio between the 'geometric mean' and the 'arithmetic mean' of the energy spectrum:

$$\mathrm{SFM} = 10log_{10}\left(\frac{\left(\prod_{k=1}^{N}|X(k)|\right)^{\frac{1}{N}}}{\frac{1}{N}\sum_{k=1}^{N}|X(k)|}\right) \tag{6}$$

Where $X(k)$ is the result of the FFT for the k-th frequency bin.

3.3.5 HFC

Measures of the amount of high frequency content of a time-series spectrum. It produces sharp peaks during attacks transients [16] and might be a good choice for detecting onsets in percussive sounds.

$$\mathrm{HFC} = \sum_{k=1}^{N}|X(k)|^2 \cdot k \tag{7}$$

Where $X(k)$ is the result of the FFT for the k-th frequency bin.

3.3.6 Spectral Centroid

The spectral centroid is a well known timbral feature that is used to describe the brightness of a sound. It represents the center of gravity of the frequency components of a signal [17].

$$\mathrm{SC} = \frac{\sum_{k=1}^{N}|X(k)| \cdot f_k}{\sum_{k=1}^{N}|X(k)|} \tag{8}$$

Where $X(k)$ is the result of the FFT for the k-th frequency bin.

3.3.7 Spectral Spread

Gives an estimation of the spread of the spectral energy around the spectral centroid [15].

$$\mathrm{SSp} = \sqrt{\frac{\sum_{k=1}^{N}|X(k)| \cdot (f_k - SC)^2}{\sum_{k=1}^{N}|X(k)|}} \tag{9}$$

Where $X(k)$ is the result of the FFT for the k-th frequency bin and SC is the spectral centroid for the frame.

3.3.8 Spectral Flux

Measures the amount of change between adjacent spectral frames [18].

$$\mathrm{SF} = \sum_{k=1}^{N} H(|X(t,k)| - |X(t-1,k)|) \tag{10}$$

Where $H(x) = \frac{x+|x|}{2}$ is the half-wave rectifier function, and t is the temporal index of the frame.

3.3.9 Spectral Skewness

Measures how symmetric is the distribution of the values for the spectral magnitudes around their arithmetic mean [19].

$$\mathrm{SSk} = \frac{2 \cdot \sum_{k=1}^{N} \left(|X(k)| - \mu_{|X|}\right)^3}{N \cdot \sigma_{|X|}^3} \tag{11}$$

Where $X(k)$ is the result of the FFT for the k-th frequency bin, $\mu_{|X|}$ is the mean value of the magnitude spectrum and $\sigma_{|X|}$ its standard deviation.

3.3.10 Spectral Kurtosis

Measures if the distribution of the spectral magnitude values is shaped like a Gaussian distribution or not [19].

$$\mathrm{SKu} = \frac{2 \cdot \sum_{k=1}^{N} \left(|X(k)| - \mu_{|X|}\right)^4}{N \cdot \sigma_{|X|}^4} \tag{12}$$

Where $X(k)$ is the result of the FFT for the $k-th$ frequency bin, $\mu_{|X|}$ is the mean value of the magnitude spectrum and $\sigma_{|X|}$ its standard deviation.

3.3.11 Spectral Rolloff

The spectral rolloff is a measure of the bandwidth of the spectrum [19]. It is defined as the point in the spectrum bellow which a percentage k of the spectral energy is contained.

3.3.12 Spectral Irregularity

The spectral irregularity measures the irregularity between consecutive FFT frames, based on the differences of amplitude between the current FFT frame and the average between the current, previous and next frames.

$$\mathrm{SI} = \sum_{k=1}^{N-1} |20\log(X[k]) - [20\log(X[k-1]) + 20\log(X[k]) + 20\log(X[k+1])]/3| \tag{13}$$

3.3.13 Spectral Entropy

The spectral entropy is based on the concept of information entropy from Shannon's information theory. It measures the unpredictability of the given state of a spectral distribution [20].

$$\mathrm{SEpy} = -\sum_{k}^{N} P(k) \cdot \log_2 P(k) \tag{14}$$

Where

$$P(i) = \frac{|X(i)|^2}{\sum_j^N |X(j)|^2} \tag{15}$$

3.3.14 Spectral Energy

The total energy of a frame of the spectrum.

$$\mathrm{SE} = \sum_{k=1}^{N} |X(k)^2| \tag{16}$$

Where $X(k)$ is the result of the FFT for the $k-th$ frequency bin.

3.3.15 Harmonic Energy

The total energy of the harmonic partials of a time-series.

$$\mathrm{HE} = \sum_{k=1}^{H} A(k)^2 \tag{17}$$

Where $A(k)$ represents the amplitude of each harmonic partial.

3.3.16 Inharmonicity

Determines the divergence of the time-series spectral components from an ideal harmonic spectrum.

$$\mathrm{Inh} = k f_0 \sqrt{1 + \lambda(k^2 - 1)} \tag{18}$$

Where k represents the index of the partial and λ is the inharmonicity factor.

3.3.17 Noisiness

The ratio of the noise energy to the total energy of a signal. Represents how noisy a signal is (values closer to 1), as opposed to harmonic (values close to 0) [15].

$$\mathrm{Ns} = \frac{\mathrm{SE} - \mathrm{HE}}{\mathrm{SE}} \tag{19}$$

3.3.18 Odd-to-Even Ratio

It is the ratio between the energy of the odd and even energy harmonics of a signal.

$$\mathrm{OER} = \frac{\sum_{h=1}^{H/2} A(2k-1)^2}{\sum_{h=1}^{H/2} A(2k)^2} \tag{20}$$

Where $A(k)$ represents the amplitude of the k-th harmonic partial.

3.4 Segmentation

The module `segmentation` contains the implementation of methods of onset detection, and in the future will also include methods for other segmentation tasks. The current implemented methods are based on pitch and RMS. Other implementations are currently under development.

3.4.1 Adaptative RMS

This method was proposed by [21], and consists on the calculation of two RMS curves: one with a short window lenght, and another with a large window length. These curves will intersect each other along the audio signal, and the onsets will tend to occur in the valleys of the interval between two intersections, when the values for the RMS calculated with the shorter window are smaller than the values calculated for the other curve. So the difference between these curves can be calculated and used as an onset detection function (ODF). The peaks of this curve are extracted to obtain the points corresponding to the note onsets.

3.4.2 Pitch change

This method is based on the detection of changes in the pitch values for adjacent frames. It calculates the ratio between adjacent frames to generate an ODF. The peaks in this curve will correspond to instants with fast changes in pitch. It is necessary to establish a threshold to consider a peak in the ODF as an onset. The cons of this method are: (1) it is highly dependent on a good pitch detection method, possibly with a good post processing method for smoothing the pitch curve; and (2) it usually fails to detect consecutive notes of the same pitch.

3.4.3 Derivative of the RMS

Since note onsets usually result in energy increases in the audio signal, the derivative of the RMS can be used to estimate the note onsets. It can be used as the ODF, since the peaks in this will correspond to instants of rapid variation in the energy of the signal. This method tends to produce false negatives in legatto phrases, since there might be little or no energy variation in the audio signal.

4. Examples

This section introduces some basic code examples for using iracema. All the examples will assume that the package iracema has been imported using the convention shown bellow.

```
1 import iracema as ir
```

Listing 1. Importing iracema

The audio files used in the next code examples can be found in the repository `iracema-audio`[4]. The examples will assume that the current working directory (CWD) set in the Python interpreter contains those audio files.

4.1 Loading audio files into audio objects

Iracema uses the package *audioread* [22] to load audio files, which can handle a wide range of different audio formats. This package wraps a few different backend libraries to load the audio files, so the supported file types will depend on which backend the user has installed. Depending on the operating system of the computer and the format of the file to be loaded, it might be necessary to install one of the backends listed in the audioread website[5]. However, the default audio libraries available in most contemporary operating systems will probably provide native support for reading a few uncompressed audio formats, such as *wave*, *aifc* or *sunau*, and possibly for a few compressed formats too.

To load an audio file, the user must provide a string containing the location where it is stored. Iracema accepts file system paths to load files stored locally or HTTP URLs to download them from remote locations. The class method `ir.Audio.load()` can be used to load the content of an audio file into a newly instantiated audio object.

```
1  flute = ir.Audio.load(
2      '00 - Flute - Iracema.wav'
3  )
4
5  # playing audio
6  flute.play()
7
8  # resampling audio to 16KHz
9  flute_resampled = flute.resample(16000)
10
11 # chaining methods to create a pipeline
12 flute_processed = (
13     flute.filter(
14         160, filter_type='highpass'
15     )
16     .normalize()
17     .resample(22050)
18 )
```

Listing 2. This example demonstrates how to load an audio file into an audio object, and the functionalities implemented in some of its methods label

The example shown above loads the audio and assigns the instantiated audio object to the variable `flute` (lines 1 through 3). This object is equipped with methods for things like playing the audio, plotting its waveform, resampling and filtering the audio, among several other functionalities.

The instance method `play()`, in line 6, will play the audio on the computer that is running the Python interpreter, using the package *sounddevice* [23]. However, if the code is being executed in a Jupyter Notebook or IPython session, iracema will display an embedded HTML audio player, which allows users to play audio even when the kernel is running on a remote computer.

Next, the method `resample()` will downsample the audio to the specified sampling rate (16KHz), using the package *resampy*[24], and return a new audio object. There are other methods that return new objects of the same class. Those methods can be chained, creating a sequential pipeline of data processing. An example of three chained methods follows in lines 12 through 18: `filter()`, `normalize()`, and

[4] ⟨https://github.com/cegeme/iracema-audio⟩

[5] ⟨https://github.com/beetbox/audioread⟩

`resample()`. The resulting audio object is attributed to the variable `flute_processed`.

4.2 Plotting data, extracting RMS and calculating the FFT

The next example will demonstrate the extraction of some features and the calculation of the FFT for an audio object.

```
1  clarinet = ir.Audio.load(
2      '02 - Clarinet - Long Notes.wav'
3  )
4
5  window = 2048
6  hop = 512
7
8  clarinet_rms = ir.features.rms(
9      clarinet,
10     window,
11     hop,
12 )
13
14 clarinet.plot();
15 clarinet_rms.plot();
16
17 clarinet_fft = ir.spectral.fft(
18     clarinet,
19     window,
20     hop,
21     fft_len=4096,
22 )
23
24 ir.plot.spectrogram(
25     clarinet_fft,
26     logfft=False,
27     fftlim=(0, 10000,),
28 );
```

Listing 3. Extracting the RMS and plotting data

In the first lines of example shown above, a different audio file is loaded into the variable `clarinet`. Then, in lines 5 and 6, two variables are set: `window` and `hop`. Both will be passed as arguments to feature extraction methods, to ajust the parameters of the sliding window applied to the audio signal. The first variable will define the length of the sliding window, and the second the number of samples to be skipped between successive windows. The method `rms()`, from the module `features`, will apply a sliding window to the `clarinet` object and calculate the Root Mean Square values for each window, returning a time series with a lower sampling frequency.

The `plot()` method of both objects is called in lines 14 and 15, to generate a line plot for the time series. The resulting plots are shown in Figures 5 and 6. This method automatically sets some basic plot parameters, such as axis labels, legend and title, by using metadata from the audio time series.

In lines 17 through 22 , the method `fft`, from the module `spectral` will apply a sliding window to the signal

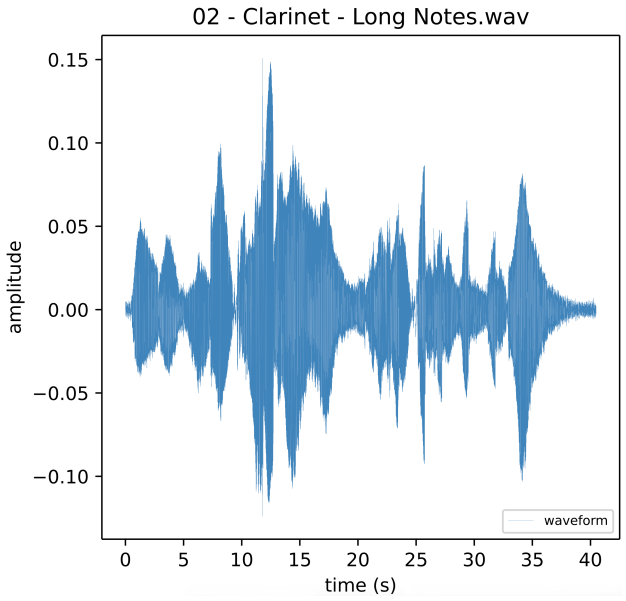

Figure 5. Audio waveform plotted for the object `clarinet`.

and calculate the FFT for each window. It will generate the time series object `clarinet_fft`, which will contain multiple values per sample, one corresponding to each bin of the FFT. Subsequently, the object will be passed to the method `ir.plot.spectrogram()`, to obtain the visualization shown in Figure 7.

4.3 Extracting pitch and harmonics

This next example will demonstrate how to extract pitch, harmonics and a few other features from an audio signal. A shorter audio file containing a faster passage with alternating ascending and descending melodies will be used for this purpose.

```
1  clari_fast = ir.Audio.load(
2      '03 - Clarinet - Fast Excerpt.wav'
3  )
4
5  window = 1024
6  hop = 512
7
8  clari_fast_fft = ir.spectral.fft(
9      clari_fast,
10     window,
11     hop,
12     fft_len=4096,
13 )
14
15 args = {
16     'minf0': 140,
17     'maxf0': 3000,
18 }
19
20 # extracting data
21 clari_fast_pitch = ir.pitch.expan(
```

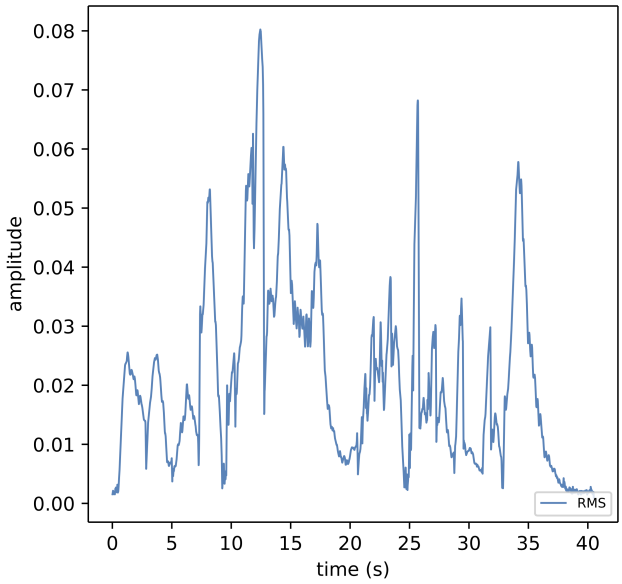

Figure 6. Plotted values for the object `clarinet_rms`.

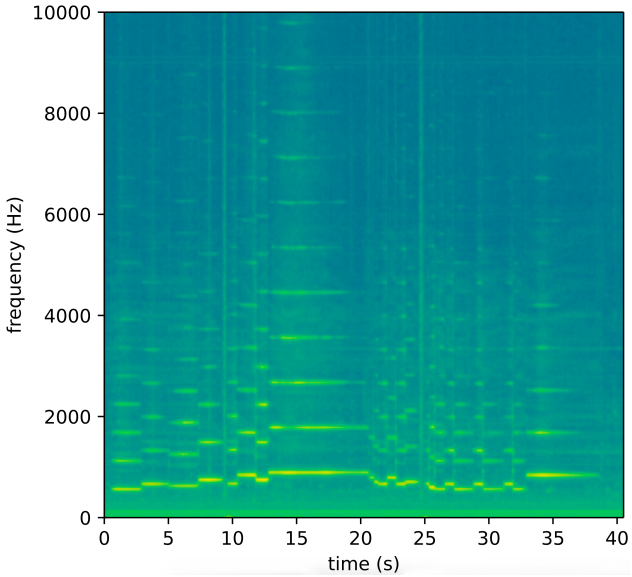

Figure 7. The resulting spectrogram.

```
22      clari_fast_fft,
23      **args,
24  )
25
26  clari_fast_harm = ir.harmonics.extract(
27      clari_fast_fft,
28      clari_fast_pitch,
29      **args,
30  )
31
32  sflux = ir.features.spectral_flux(
33      clari_fast_fft
34  )
35  sflat = ir.features.spectral_flatness(
36      clari_fast_fft
37  )
38  sc = ir.features.spectral_centroid(
39      clari_fast_fft
40  )
41  no = ir.features.noisiness(
42      clari_fast_fft,
43      clari_fast_harm['magnitude'])
44  hfc = ir.features.hfc(clari_fast_fft)
45
46  # plotting the extracted data
47  ir.plot.waveform_spectrogram_pitch(
48      clari_fast,
49      clari_fast_fft,
50      clari_fast_pitch,
51      fftlim=(0, 10000),
52  );
53  ir.plot.waveform_spectrogram_harmonics(
54      clari_fast,
55      clari_fast_fft,
56      clari_fast_pitch,
57      clari_fast_harm['frequency'],
58  );
59  ir.plot.waveform_trio_and_features(
60      clari_fast,
61      features=(sflux, sflat, sc, no, hfc),
62  );
```

Listing 4. Extracting pitch, harmonics and other features

First, the audio file is loaded and its FFT is calculated. In lines 21 through 24, the method `expan()` from the module `pitch` is employed to extract the pitch for the melody. Then, to extract the harmonics of the signal, the method `extract()`, from the module `harmonics` is used. The method for extraction of harmonics will extract 16 harmonics by default, but a different number could be specified, using the optional argument `nharm`. Both of these methods receive the arguments `minf0` and `maxf0`, which represent the frequency interval in which the methods will look for a fundamental frequency.

Then, in lines 32 through 44, five different spectral features will be calculated for the whole audio signal: spectral flux, spectral flatness, spectral centroid, noisiness and high frequency content (HFC).

The last lines of the example (47 through 62) contain the methods used to plot the extracted data. The resulting plots are shown in Figures 8, 9, 10.

4.4 Extracting onsets

This next example will demonstrate the extraction of note onsets using a method called adaptative RMS. The same audio excerpt from the previous example will be used.

```
1  onsets = ir.segmentation.onsets\
2      .adaptative_rms(
3          clari_fast,
4          min_time = 0.01,
5          display_plot = True,
```

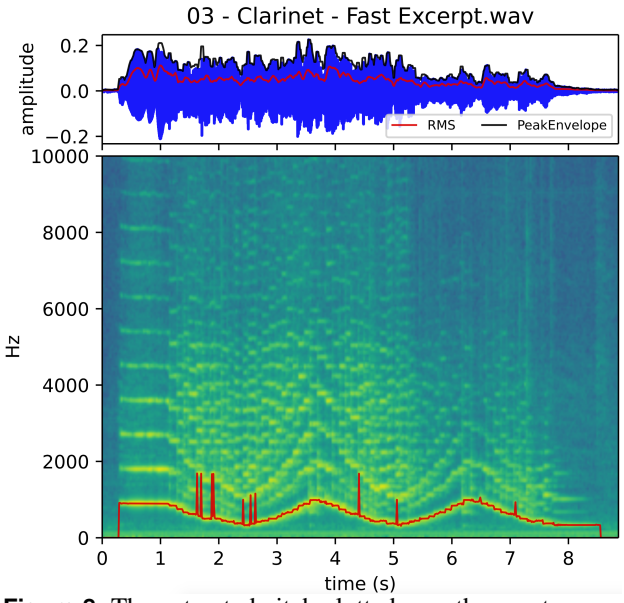

Figure 8. The extracted pitch plotted over the spectrogram for a fast excerpt played on the clarinet.

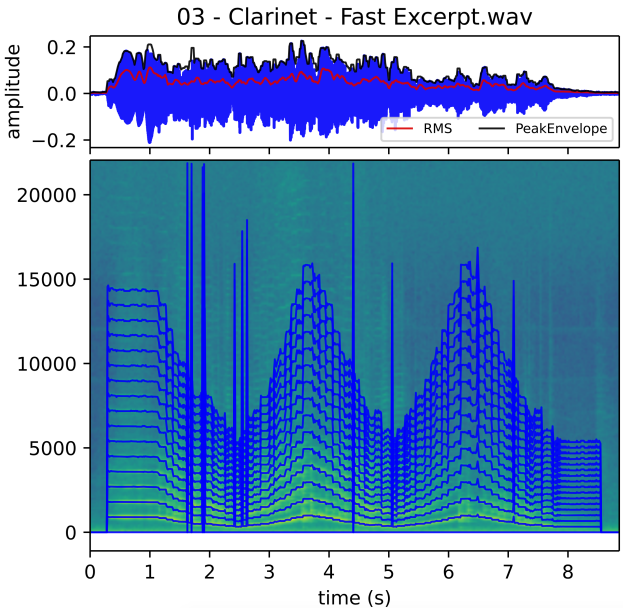

Figure 9. The extracted frequencies of the harmonics plotted over the spectrogram.

```
6       )
7
8   clari_fast.play_with_clicks(onsets)
```

Listing 5. Note onset detection using the method adaptative RMS.

The method `adaptive_rms` (lines 1 through 6), from the module `segmentation.onsets`, receives only one positional argument, which is the audio object for which the note onsets will be extracted. There are a few other optional arguments, but the only being used in this example are `min_time`, which is the minimum time in seconds between two consecutive onsets and `display_plot`, which controls if the plot of the onset detection function used should be displayed or not. Figure 11 shows the resulting graph. In the last line of this example, the method `play_with_clicks()` from the object `clari_fast` is called, receiving as argument the variable `onsets`, which is a list of points, each corresponding to an instant in the audio object. This method will play the audio mixed with click sounds in the instants corresponding to the extracted onsets.

5. Future perspectives

The library's functionalities will move towards feature extractors that can provide more meaningful representations of the information from music performance, from a musical point of view. The architecture proposed for Iracema and the feature extractors mentioned in this article form the basis for the development of such representations, which will be included in the future stages of development of the tool.

Good models for note segmentation are essential for audio content extraction, so this is our primary concern at the time. Although we have already implemented some basic models that use pitch and energy information to detect note onsets, they sometimes produce false positives. Therefore, we are working on a better model, using machine learning techniques, which should be included in a future release. Such robust note segmentation is essential for obtaining good articulation descriptors, and we have already developed a legato index descriptor that relies on such robustness. We also plan to include, in a future version of Iracema, a vibrato descriptor, which was previously proposed and described in [25].

6. Acknowledgement

The development of this work has been supported by CAPES / Brazil (Coordenação de Aperfeiçoamento de Pessoal de Nível Superior) and CNPq/Brazil (Conselho Nacional de Desenvolvimento Científico e Tecnológico).

Author contributions

Tairone N. Magalhães contributed writing the paper and as the main developer of the library *Iracema*. Felippe B. Barros contributed structuring some sections of the paper and as a developer of the library *Iracema*. Mauricio A. Loureiro was responsible for the supervision of the research project, also contributed writing and proofreading the paper.

References

[1] CAMPOLINA, T. A. M.; MOTA, D. A.; LOUREIRO, M. A. Expan: a tool for musical expressiveness analysis. In: *Proceedings of the 2 nd International Conference of Students of Systematic Musicology*. Ghent, Belgium: IPEM, 2009. p. 24–27.

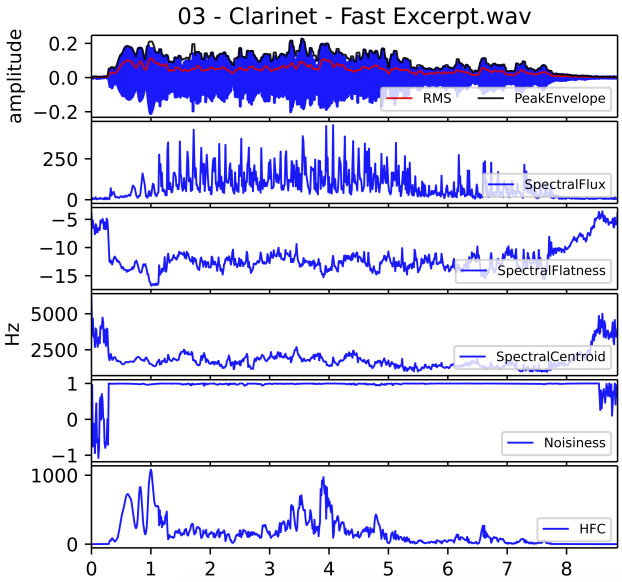

Figure 10. Five different spectral features plotted along with the audio waveform.

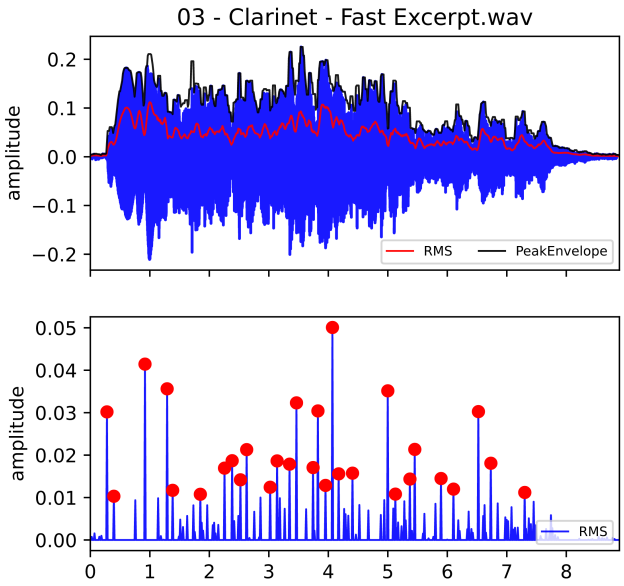

Figure 11. Onset detection function based on RMS and the extracted onsets (points in red).

[2] WALT, S. van der; COLBERT, S. C.; VAROQUAUX, G. The NumPy array: a structure for efficient numerical computation. *Computing in Science & Engineering*, IEEE, v. 13, n. 2, p. 22–30, 2011.

[3] JONES, E. et al. *SciPy: Open source scientific tools for Python*. 2001. Disponível em: ⟨http://www.scipy.org/⟩.

[4] REPP, B. H. Individual differences in the expressive shaping of a musical phrase: the openin of Chopin's Etude in E major. In: *Music, Mind and Science*. Seoul, Korea: Seoul National University Press, 1999. p. 239–270.

[5] CLARKE, E. Empirical methods in the study of performance. In: *Empirical musicology: Aims, methods, prospects*. [S.l.: s.n.], 2004. p. 77–102.

[6] PALMER, C. Music performance. *Annual review of psychology*, v. 48, p. 115–38, 1997.

[7] GABRIELSSON, A. Music Performance Research at the Millennium. *Psychology of Music*, v. 31, n. 3, p. 221–272, jul 2003. Disponível em: ⟨http://pom.sagepub.com/cgi/doi/10.1177/03057356030313002⟩.

[8] GOEBL, W.; DIXON, S.; POLI, G. D. Sense in expressive music performance: Data acquisition, computational studies, and models. *Sound to Sense – Sense to Sound: A State of the Art in Sound and Music Computing*, p. 195–242, 2005.

[9] LERCH, A. et al. Music Performance Analysis: A Survey. 2019.

[10] REPP, B. H. Diversity and commonality in music performance: an analysis of timing microstructure in Schumann's "Träumerei". *The Journal of the Acoustical Society of America*, v. 92, p. 227–260, 1992.

[11] BERNAYS, M.; TRAUBE, C. Investigating pianists' individuality in the performance of five timbral nuances through patterns of articulation, touch, dynamics, and pedaling. *Frontiers in Psychology*, v. 5, n. MAR, p. 1–19, 2014.

[12] KOREN, R.; GINGRAS, B. Perceiving individuality in harpsichord performance. *Frontiers in Psychology*, v. 5, n. FEB, p. 1–13, 2014.

[13] CUADRA, P. D. L. Efficient pitch detection techniques for interactive music. In: *ICMC*. [S.l.: s.n.], 2001. p. 403–406.

[14] KIM, J. W. et al. CREPE: A Convolutional Representation for Pitch Estimation. In: *Proceedings of the IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP), 2018*. [S.l.: s.n.], 2018.

[15] PEETERS, G. et al. The timbre toolbox: extracting audio descriptors from musical signals. *The Journal of the Acoustical Society of America*, v. 130, n. 5, p. 2902–2916, 2011.

[16] BELLO, J. P. et al. A tutorial on onset detection in music signals. *IEEE Transactions on Speech and Audio Processing*, v. 13, n. 5, p. 1035–1046, 2005.

[17] PARK, T. H. *Introduction to digital signal processing: Computer musically speaking*. [S.l.]: World Scientific Publishing Co. Pte. Ltd., 2010. 429 p.

[18] DIXON, S. Onset Detection Revisited. In: *9th International Conference on Digital Audio Effects*. Montreal, Canada: [s.n.], 2006. p. 133–137.

[19] LERCH, A. *An introduction to audio content analysis: Applications in signal processing and music informatics*. [S.l.: s.n.], 2012. 1–248 p.

[20] TOH, A. M.; TOGNERI, R.; NORDHOLM, S. Spectral entropy as speech features for speech recognition. *Computer Engineering*, n. 1, p. 22–25, 2005.

[21] De Poli, G.; MION, L. From audio to content. In: *Unpublished book*. [S.l.: s.n.], 2006. cap. 5.

[22] SAMPSON, A. *audioread*. 2020. Disponível em: ⟨https://github.com/beetbox/audioread⟩.

[23] GEIER, M. *sounddevice*. Spatial Audio, 2020. Disponível em: ⟨https://github.com/spatialaudio/python-sounddevice⟩.

[24] MCFEE, B. *resampy*. 2019. Disponível em: ⟨https://github.com/bmcfee/resampy⟩.

[25] MAGALHÃES, T. N. et al. Análise do vibrato e bending na guitarra elétrica. In: *Anais do XV Simpósio Brasileiro de Computação Musical*. [s.n.], 2015. p. 36–47. Disponível em: ⟨http://compmus.ime.usp.br/sbcm/2015/papers/sbcm-2015-3.pdf⟩.