AN OPEN SOURCE SOLUTION FOR LIVE COMPUTATION AND BROADCASTING OF SOUND DESCRIPTORS

Valerio ORLANDINI (valerio.orlandini@stud-phd.consfi.it) (0000-0002-6857-5471)[1]

[1]**Conservatory of Music "Luigi Cherubini"**, Firenze, Italy

ABSTRACT

The Informer is a comprehensive open-source solution for real-time extraction and distribution of audio features. The system comprises a VST3/AU/LV2 plugin, a standalone application, a VCV Rack module, and Max/MSP externals. All these components compute a wide range of time- and frequency-domain sound descriptors. The results are broadcast via Open Sound Control (OSC) across all components, with the exception of the Max/MSP externals, which communicate directly with other objects. A detailed overview of the implemented descriptors, such as kurtosis, skewness, spectral centroid, flux, flatness, and more, is provided, along with their mathematical formulations and domain-specific interpretations. The accompanying C++ header-only library (with Python and JavaScript bindings) makes these computational algorithms accessible to developers, while a Max for Live device facilitates dynamic parameter modulation within Ableton Live. Built on open-source frameworks and distributed under free software licenses, *The Informer* bridges technical audio analysis with creative applications in music production, audiovisual performance, interactive installations, and scientific research.

1. INTRODUCTION

Audio descriptors, also referred to as sound descriptors, are numerical representations of specific characteristics of a sound signal. These descriptors have been systematically formalized with a shared terminology since the late 1990s, driven by developments such as the MPEG-7 standard [1] and significant advancements in the field of music information retrieval [2, 3].

Audio features play a crucial role in music analysis by capturing diverse characteristics of sound, such as timbre, rhythm, pitch, and dynamics, enabling their application across various domains. In music information retrieval (MIR), these features are used to classify genres, identify songs, or recommend music based on similarity [3, 4]. In musicology, they provide quantitative insights into the structural and expressive qualities of musical works [5], aiding scholars in understanding compositional techniques and stylistic evolution. Furthermore, sound descriptors can be employed in big data analysis, even on non-audio signals. Beyond these applications, audio features may also be used as a bridge between auditory and visual domains, facilitating the creation of artistic visual representations that are synchronized with sound, such as in multimedia installations or data-driven music visualizations.

In this paper, after a review of several low level sound descriptors, a series of plugins and tools to compute audio features and broadcast them across applications and a related multi-language library are presented.

Copyright: © 2026. This is an open-access article distributed under the terms of the Creative Commons Attribution 3.0 Unported License, which permits unrestricted use, distribution, and reproduction in any medium, provided the original author and source are credited.

1.1 Previous and similar works

The plugin was designed with this workflow in mind: analyse the incoming audio, compute the descriptors, and broadcast them through Open Sound Control protocol [6]. While numerous software packages already compute audio descriptors, most operate within distinct environments or formats. Notable examples include the Max/MSP packages *Zsa.Descriptors* [7], *FluCoMa* [8] and *MuBu* [9] (and its underlying API *PiPo* [10]), the Python library *librosa* [11], the C++ (with Python bindings) library *essentia* [12], and the standalone software *Sonic Visualiser* [13] and *Partiels* [14].

Sound Analyser [15] is a plugin that computes audio descriptors in real time and broadcasts them via OSC. Released in 2014, with its last update dating back to December 2015, this tool implements various analysis algorithms and allows users to extend its functionality by incorporating third-party functions written in C++. However, aside from appearing to no longer be actively maintained (and still being based on JUCE 4), it offers fewer descriptors (as it is not exclusively focused on them) and depends not only on JUCE but also on a few other libraries.

Conversely, *The Informer* mitigates these constraints by utilizing the latest JUCE framework and maintaining a self-contained, dependency-free architecture. Beyond the plugin itself, the project encompasses a broader ecosystem: a standalone C++ library implementing the same descriptor computations, featuring JavaScript and Python bindings, alongside compilation for heterogeneous environments such as Max/MSP and VCV Rack. While certain alternatives may offer greater algorithmic sophistication or computational efficiency, they lack this breadth of integration. *The Informer* thus aims to provide a highly comprehensive solution, balancing a lightweight plugin design with extensive cross-language and cross-platform interoperability.

2. IMPLEMENTED SOUND DESCRIPTORS

Before delving into the details of the implementation and use cases of the library and plugin, it is essential to outline the audio descriptors employed and explain their calculation methods. This step is particularly important because, while some descriptors have a single, well-defined implementation, others vary across sources in the literature, even though, most of the times, their theoretical definitions remain consistent in meaning.

The descriptors are categorized into time-domain and frequency-domain types. Some descriptors, such as peak, kurtosis, and skewness, can be defined in both domains.

2.1 Time domain descriptors

Given an audio signal $x(n)$ spanning over N samples, with a mean $\mu = \frac{1}{N}\sum_{n=1}^{N} x(n)$, the following descriptors can be defined:

Peak level, the sample with the highest absolute value of magnitude:

$$\mathrm{P}_n = \max_{1 \le n \le N} \{|x(n)|\} \tag{1}$$

Root mean square, the square root of the samples' mean square:

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

Variance, the 2nd statistical moment, calculated as the squared deviation from the mean:

$$\sigma_n^2 = \frac{1}{N}\sum_{n=1}^{N}(x(n)-\mu)^2 \tag{3}$$

Skewness, the 3rd statistical moment, measuring the asymmetry of the signal around its mean:

$$\mathrm{SK}_n = \frac{\frac{1}{N}\sum_{n=1}^{N}(x(n)-\mu)^3}{\left(\sqrt{\frac{1}{N}\sum_{n=1}^{N}(x(n)-\mu)^2}\right)^3} \tag{4}$$

Kurtosis, which is the 4th statistical moment and measures how much the distribution of the samples of the signal deviates from a normal distribution. The lower the kurtosis, the flatter the signal:

$$\mathrm{K}_n = \frac{\frac{1}{N}\sum_{n=1}^{N}(x(n)-\mu)^4}{\left(\sqrt{\frac{1}{N}\sum_{n=1}^{N}(x(n)-\mu)^2}\right)^4} - 3 \tag{5}$$

The value 3 is subtracted so that a normal distribution has zero kurtosis: in some texts this is referred as *excess kurtosis*.

Zero crossing rate, the rate at which a signal crosses the zero amplitude axis (i.e., transitions between positive and negative values). High values indicate a noisy signal or one rich in very high frequencies:

$$\mathrm{ZCR}_n = \frac{1}{N-1}\sum_{n=2}^{N} |\mathrm{sgn}(x(n)) - \mathrm{sgn}(x(n-1))| \tag{6}$$

Moreover, the fundamental frequency of the signal can be estimated in the time domain using the YIN algorithm [16], that computes a cumulative mean-normalized difference function of the signal and locates its first local minimum below a predefined threshold, thereby determining the period of the waveform while mitigating the octave errors common to standard autocorrelation-based methods.

2.2 Frequency domain descriptors

Given an array of magnitudes X derived from a short-time Fourier transform (STFT) across K samples, considering a sampling rate of f_s, and with $f(k)$ the centre frequency of the k-th bin, the following descriptors can be defined:

Peak frequency, the centre frequency of the bin with the maximum magnitude:

$$\mathrm{SPK}_n = \frac{argmax_k\{|X(k,n)|\}f_s}{K} \tag{7}$$

Crest factor, the peak-to-average ratio of the spectrum. High values indicate a spectrum concentrated in less bins [3]:

$$\mathrm{SCF}_n = \frac{\max_{0 \le k \le K/2-1}\{|X(k,n)|\}}{\sum_{k=0}^{K/2-1}|X(k,n)|} \tag{8}$$

Rolloff, the frequency below which a specified percentage κ of the signal is concentrated. Typical values for κ are 85% and 95% (corresponding to 0.85 or 0.95) [17]:

$$\mathrm{SRO}_n = \frac{if_s}{K} \text{ such that } \sum_{k=0}^{i}|X(k,n)| = \kappa \sum_{k=0}^{K/2-1}|X(k,n)| \tag{9}$$

Flatness, which indicates the uniformity of a signal's frequency distribution. A higher flatness value suggests a more noisy (or less tonal) signal [3]:

$$\mathrm{SFL}_n = \frac{\exp\left(\frac{\sum_{k=0}^{K/2-1} ln(|X(k,n)|)}{K/2}\right)}{\frac{\sum_{k=0}^{K/2-1}|X(k,n)|}{K/2}} \tag{10}$$

It is recommended to add a small delta to all magnitude values before computing their logarithm, to ensure numerical stability and avoid undefined operations. Simply discarding bins with zero magnitude is not a viable alternative, as it may lead to misleading results. For instance, a sinusoidal signal with energy concentrated in a single non-zero frequency bin would yield a spectral flatness of

1.0, which is inaccurate.

Slope, which quantifies the spectrum's overall energy trend. A positive slope indicates more energy at higher frequencies, while a negative slope indicates more energy at lower frequencies [3]:

$$\mathrm{SSL}_n = \frac{\sum_{k=0}^{K/2-1}(f(k)-\mu_f)\left(|X(k,n)|-\mu_X\right)}{\sum_{k=0}^{K/2-1}(f(k)-\mu_f)^2} \quad (11)$$

with μ_f being the mean frequency, given by $\frac{\sum_{k=0}^{K/2-1} f(k)}{K/2}$ and μ_X the mean magnitude $\frac{\sum_{k=0}^{K/2-1}|X(k,n)|}{K/2}$.

Decrease, indicating how steep is the decrease of the spectral envelope [2, 3, 18]:

$$\mathrm{SD}_n = \frac{\sum_{k=1}^{K/2-1}\frac{|X(k,n)|-|X(0,n)|}{k}}{\sum_{k=1}^{K/2-1}|X(k,n)|} \quad (12)$$

Centroid, the frequency corresponding to the centre of the mass of the spectrum [2, 3, 17]:

$$\mathrm{SC}_n = \frac{\sum_{k=0}^{K/2-1} f(k)\cdot|X(k,n)|}{\sum_{k=0}^{K/2-1}|X(k,n)|} \quad (13)$$

Spread, which measures how concentrated the spectrum is around the spectral centroid. High values indicate that the spectrum is more widely dispersed across frequency bins [2, 3, 17]:

$$\mathrm{SSP}_n = \sqrt{\frac{\sum_{k=0}^{K/2-1}(f(k)-\mathrm{SC}_n)^2|X(k,n)|}{\sum_{k=0}^{K/2-1}|X(k,n)|}} \quad (14)$$

Skewness, measuring the asymmetry of the spectrum around the centroid. A value of zero indicates a symmetric distribution, while negative values suggest more energy concentrated at higher frequencies, and positive values indicate more energy concentrated at lower frequencies [3]:

$$\mathrm{SSK}_n = \frac{\sum_{k=0}^{K/2-1}(f(k)-\mathrm{SC}_n)^3|X(k,n)|}{\mathrm{SSP}_n^3\sum_{k=0}^{K/2-1}|X(k,n)|} \quad (15)$$

Kurtosis, indicating how much the spectral magnitudes distribution is close to a Gaussian distribution [3]:

$$\mathrm{SKU}_n = \frac{\sum_{k=0}^{K/2-1}(f(k)-\mathrm{SC}_n)^4|X(k,n)|}{\mathrm{SSP}_n^4\sum_{k=0}^{K/2-1}|X(k,n)|} - 3 \quad (16)$$

Entropy, the degree of "disorder" of a signal, indicating how noisy and complex a signal is [17, 19]:

$$\mathrm{SH}_n = -\frac{\sum_{k=0}^{K/2-1} P_m log_2\left(P_m\right)}{log_2(K/2)} \quad (17)$$

with P_m being the probability distribution of the signal, given by $\frac{|X(k,n)|^2}{\sum_{k=0}^{K/2-1}|X(k,n)|^2}$.

Flux, which quantifies the average magnitude of change between consecutive spectral frames. It measures how much the spectrum varies over time, with higher values indicating more rapid spectral changes [3]:

$$\mathrm{SF}_n = \frac{\sqrt{\sum_{k=0}^{K/2-1}\left(|X(k,n)|-|X(k,n-1)|\right)^2}}{K/2} \quad (18)$$

Irregularity, which quantifies how unevenly the magnitude is distributed across frequency bins. A value of zero indicates a flat spectrum, a value approaching two may suggest a spectrum concentrated in one or a few bins, and a value around one signifies a spectrum with alternating peaks and valleys. There are various methods to compute irregularity, each focusing on either local or global patterns of unevenness [20, 21]. The one employed in *The Informer* is:

$$\mathrm{SI}_n = \frac{\sum_{k=1}^{K/2-1}|X(k,n)-X(k-1,n)|}{\sum_{k=0}^{K/2-1}|X(k,n)|} \quad (19)$$

Note that most of the spectral descriptors can be implemented using either the magnitude spectrum (i.e. $|X(k,n)|$) or the power one (i.e. $|X(k,n)|^2$). In all the reported formulas, the last frequency bin, corresponding to the Nyquist frequency, has not been considered due to its unreliability [3]. However, it can optionally be included by replacing every occurrence of $K/2-1$ with $K/2$.

In Table 1, the descriptors are summarized along with their respective boundaries. When no unit of measure is specified, the values are dimensionless. If one or both the boundaries are not fixed and depend on the characteristics of the signal or on the spectral magnitude normalization, they are marked as *not defined*. For time descriptors, it is assumed that sample values are inside [-1.0, 1.0] range.

3. FRAMEWORK OVERVIEW

The proposed framework was initially conceived as a plugin for Digital Audio Workstations (DAWs). However, its underlying rationale was subsequently implemented across a series of distinct software components, which are examined in depth in the following sections. The framework currently comprises:

- *Multi-platform plugin and standalone software*: A GPLv3-licensed VST3/LV2/AU plugin and standalone application, built using JUCE and available for Linux, macOS, and Windows. It computes descriptors in real-time and broadcasts them via Open Sound Control (OSC). A Max for Live receiver device is also available to facilitate the use of the descriptors as modulation sources within Ableton Live.

Name	Domain	Boundaries
Kurtosis	Time	*not defined*
Peak	Time	$[0.0, 1.0]$
Pitch[a]	Time	$[f_s/N, f_s/2]$ Hz
RMS	Time	$[0.0, 1.0]$
Skewness	Time	*not defined*
Variance	Time	$[0.0, 1.0]$
Zero crossing rate	Time	$[0.0, 1.0]$
Centroid	Frequency	$[0.0, f_s/2]$ Hz
Crest factor	Frequency	$[2/K, 1.0]$
Decrease	Frequency	$(n.\ d., \sum_{k=1}^{K/2-1} 1/k]$
Entropy	Frequency	$[0.0, 1.0]$
Flatness	Frequency	$[0.0, 1.0]$
Flux	Frequency	$[0.0, \textit{not defined})$
Irregularity	Frequency	$[0.0, 2.0]$
Kurtosis	Frequency	*not defined*
Peak	Frequency	$[0.0, f_s/2]$ Hz
Rolloff	Frequency	$[0.0, f_s/2]$ Hz
Skewness	Frequency	*not defined*
Slope	Frequency	*not defined*
Spread	Frequency	$[0.0, f_s/4]$ Hz

[a] Using YIN algorithm

Table 1. Descriptors and their boundaries

- *VCV Rack module*: A module for VCV Rack implementing the same functionalities as the plugin version; GPLv3-licensed.
- *Max/MSP externals*: C++ externals for Max/MSP designed to analyse either real-time audio streams or audio buffers; MIT-licensed.
- *C++ library*: A header-only, single-file C++ template library that implements the algorithms used in the plugin. The VCV Rack module and Max/MSP externals rely on this library for descriptor computation; MIT-licensed.
- *Python bindings*: MIT-licensed Python bindings for the C++ library, implementing both class-based and functional interfaces.
- *JavaScript bindings*: MIT-licensed JavaScript bindings for the C++ library, with C++ code compiled to WebAssembly.

4. PLUGIN IMPLEMENTATION

4.1 Implementation

C++ programming language and JUCE library were chosen for the development of the software. The build system is based on CMake, which relies on CPM to automatically download and build the JUCE library (the only required dependency) at compile time. Targets include Linux, Windows and macOS, with plugin formats VST3, LV2 and, for macOS only, AU. A standalone version for all platforms is also built.

The source code is available under the GNU General Public License (GPL) version 3 on GitHub at the following address: `https://github.com/valeriorlandini/theinformer`. Additionally, pre-compiled versions of the software can be found on the same page for user convenience.

The plugin operates by storing a segment of audio samples, which are used to compute time-domain descriptors and perform a Fast Fourier Transform (FFT). The result of the FFT is then used to calculate frequency-domain descriptors. The size of the sample chunk depends on the sample rate: 4096 samples are used for rates up to 48 kHz, while 8192 samples are used for higher rates, so that descriptors are updated around 20 times per second at common sample rates, considering that the chunks overlap by 50%. The FFT is performed using JUCE's internal FFT class, and the samples are processed with a Hann window function.

All descriptors are calculated using the formulas provided in the previous section. The spectral slope is scaled by a factor of 100 to account for its typically small magnitude.

Some audio descriptors are mathematically undefined for silent signals, particularly those that involve a magnitude or power sum in the denominator or within logarithmic functions. To ensure robustness and avoid computational errors, these descriptors are conventionally assigned a value of zero under such conditions. The same approach is applied to skewness and kurtosis when the denominator approaches or equals zero, preventing division by zero and maintaining numerical consistency in the output.

The plugin supports up to 64 channels, with descriptors calculated individually for each channel. To apply the same set of descriptors to the overall mix, the values from individual channels are averaged; for peak-related descriptors, the maximum value across channels is selected.

Moreover, in addition to the descriptors mentioned above, a simplified spectrum is provided, consisting of a user-specified number of equal-octave bands (ranging from 2 to 16). For each of these bands, the reported value corresponds to the square root of the highest magnitude among all the original frequency bands of the full spectrogram that fall within that equal-octave band. Equal-octave bands span 60–16000 Hz: any energy below 60 Hz is added to the first band, and any above 16000 Hz to the last.

The OSC messages are queued for delivery as soon as the descriptors are computed, which occurs at a rate of approximately 20 times per second. These messages are formatted according to the following schema: `/[root]/[channel]/[domain]/[descriptor]`, with `root` a custom identifier editable via the user interface (see next paragraph), `channel` that can be `mix` or a number between `1` and `64` and `domain` either `time` or `freq`.

The user interface has been designed to be as minimalistic as possible: as shown in Fig. 1, the adjustable parameters are the IP address, the port, the root string of the OSC messages, the activation of the normalization feature, to have all the parameters inside the [0.0, 1.0] interval, and the number of bands for the simplified spectrum. For descriptors without predefined theoretical limits, normalization relies on empirical estimates of their typical extreme

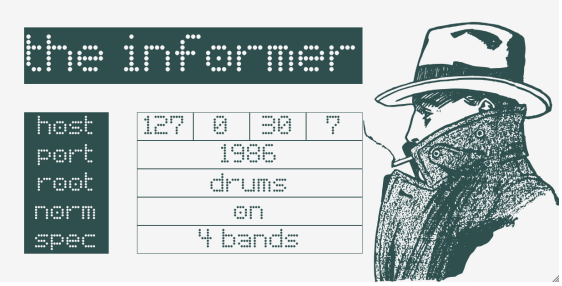


Figure 1. User interface of *The Informer* plugin.

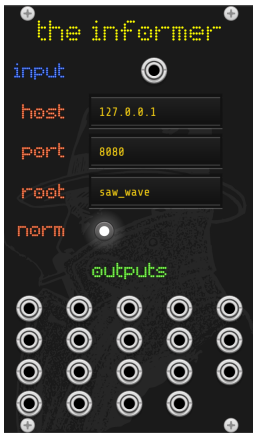


Figure 2. User interface of *The Informer* Rack module.

values. Consequently, this feature prioritizes artistic utility over rigorous measurement precision. When more than one instance of *The Informer* is opened, the default root string is automatically adjusted (using incremental numbers) to avoid name clashes.

4.2 VCV Rack module

After finalizing the initial version of the DAW plugin and standalone application, a module was developed for the open-source modular synthesizer platform *VCV Rack*. The concept underlying *The Informer* aligns well with the capabilities of *Rack*, as it offers both artistic and scientific value to extract audio descriptors from various stages of an audio synthesis or effects chain. This enables a dynamic representation of how parameters evolve throughout the signal processing pipeline.

Unlike the plugin, which currently implements the descriptors calculation functions directly, the Rack module uses the companion C++ library, whose architecture and rationale are discussed in the next section. For the Fourier transform, it uses the built-in DSP Rack library (which incorporates a fork of *pffft*), while for OSC message dispatching it uses the header-only, zlib licensed library *oscpkt* (with some modifications to make it work properly on Windows).

The module transmits the computed values via OSC, with an option to normalize them between 0 and 1 as in the plugin version, as well as through individual output ports for use within Rack (see Fig. 2). In this context, all descriptors are normalized and scaled to the range [0, 10] to conform to the voltage standards recommended by VCV.

The FFT size is fixed at 8192 samples with a 50% overlap between consecutive frames. Each instance of the module processes a mono signal.

4.3 Max/MSP externals

Built upon the same C++ library discussed in the next section and used in the VCV Rack module, a series of externals for Cycling '74 Max/MSP were also developed.

The *Informer for Max* package, created taking advantage of Cycling '74's Min-DevKit, comprises five externals:

- `informer.timedesc~`, `informer.timedesc`: Compute time-domain descriptors from a real-time audio stream. The tilde version outputs a selected descriptor as an audio signal, while the non-tilde version outputs all descriptors as messages, for routing or populating a dictionary. Block size and overlap factor can be set by the user.

- `informer.freqdesc~`, `informer.freqdesc`: Compute frequency-domain descriptors from a series of FFT magnitudes (e.g., within a `pfft~` subpatch) derived from a real-time audio stream. As above, the tilde version outputs a chosen descriptor as an audio signal, and the non-tilde version emits all descriptors as messages.

- `informer.bufferdesc`: Analyzes audio stored in a buffer to compute both time- and frequency-domain descriptors. It outputs messages suited to populate a `dict`, which can be later exported also as a JSON file. Fourier transform is performed using the public domain library *dj_fft*. Block size can be adjusted, from 32 to 32768 samples, while hop size is always half of the block size.

Each external is accompanied by a comprehensive help patch with complete usage examples.

4.4 Max for Live receiver device

A Max for Live device was developed to seamlessly integrate the plugin's functionality into Ableton Live. This device, called *The Case Officer*, acts as a bridge, receiving OSC messages from the plugin and exposing the descriptors as native modulation sources within the DAW. For instance, a user can map the spectral centroid to a synthesizer's filter cutoff, enabling timbre-driven, audio-reactive modulation that extends beyond traditional amplitude-based sidechaining. Similarly, mapping spectral flux to a delay line's feedback amount may allow spatial effects to naturally intensify during transient-rich or highly active musical passages. The interface allows users to selectively route specific parameters and define the modulation range for linked targets. Furthermore, it features an OSC daisy-chain functionality that forwards incoming messages to a secondary port. This practical design allows the same descriptor data to be shared across multiple device instances

or external applications, facilitating synchronized generative systems without duplicating the computational cost of the analysis.

4.5 Use cases

The plugin was initially developed for personal artistic projects that required real-time visual feedback of sound properties. However, the software was designed with versatility in mind, enabling it to support more scientific and digital signal analysis tasks as well. A summary of potential applications for *The Informer* is provided below.

Real-time visualization of audio properties This can be integrated into platforms like Jitter, TouchDesigner, or any other OSC-capable software. For artists and developers working with live audiovisual performances, this feature provides immediate visual feedback of sound characteristics. Examples of audiovisual interaction achieved through *The Informer* sending data from Ableton Live to TouchDesigner can be found at these links: `https://youtu.be/dkD22xAVwTQ` and `https://youtu.be/H_kckhTK-50`.

Dynamic control of plugin parameters Audio descriptors can serve as modulators to adjust parameters in other plugins. For instance, its messages can be used to automate the cutoff frequency of a filter, the thresholds of a multiband compressor, or even spatialization effects based on real-time audio analysis.

Real-time analysis of audio streams The plugin can analyse live audio streams and output data in various formats. This data can be used directly within compatible software for immediate visualization or recorded into text files, CSVs, or other text-based formats for later analysis. This makes it suitable for both live performance scenarios and offline scientific research.

Educational tools The plugin can be used in educational settings to teach students about audio descriptors and their sonic meaning, while the accompanying library (discussed in the next section) can be used to easily integrate descriptors computation into music analysis workflows.

Sound design and post-production Sound designers and post-production engineers can use the plugin to analyze and refine audio elements in film, video games, or music production. For example, it could help identify unwanted artifacts in a recording or guide the creation of adaptive soundscapes.

Biofeedback and experimental research Researchers exploring biofeedback or experimental psychology could use *The Informer* to map features of biosignals into visual or auditory feedback systems, creating new possibilities for studying human responses to stimuli.

5. C++ LIBRARY

Following the implementation of the plugin, all descriptor calculation functions, developed entirely from scratch without reliance on external dependencies (assuming the availability of audio samples and Fourier transform data), were systematically organized and encapsulated into a header-only library for ease of integration.

The library is structured to include both individual functions and a unified class. The class may be used as an interface to invoke the functions and, if desired, compute all descriptors simultaneously. This approach is recommended, as many descriptors rely on intermediate values that are shared across computations, thereby promoting efficiency and reducing redundancy. Moreover, there is a set of utility functions for data preparation and formatting. Built on C++ templates, the functions and class are designed to accept any type of floating-point data. The minimum required C++ standard is C++14, but the use of C++20 or higher is recommended. Any operating system with a modern C++ compiler is supported.

The library is organized in a single header file, `informer.h`, which features a namespace `Informer` with two inner namespaces, `Amplitude` and `Frequency`, to avoid confusion for functions with the same name in the two domains. Inside each inner namespace, the functions to compute the descriptors presented in the second section of this paper are implemented. For descriptors that need other descriptors' values to be computed (e.g. spectral spread or kurtosis), the functions allow the user to pass these values if already computed, otherwise they are calculated at runtime.

Inside `Informer` namespace there is also the class `Informer`, that allows to load a set of samples and their corresponding spectral representation and compute the various descriptors, that can be subsequently recalled. Once the class is instantiated with sample rate and optionally the data to analyse, the descriptors can be computed and retrieved. Usage examples of the library can be found in the GitHub repository.

The library is licensed under the MIT License, so it can also be integrated into closed-source software, provided that the license terms are met.

5.1 Python bindings

To enhance usability and streamline testing of the library, a set of Python bindings was developed using *pybind11*. The resulting interface closely mirrors the structure of the original C++ library and is organized into a Python module named `pyinformer`, which includes two submodules: `pyinformer.amplitude` and `pyinformer.frequency`. These submodules provide the same functionalities of their C++ counterparts. Additionally, a class `pyinformer.Informer` is provided, offering a Python interface that parallels the class-based API of the C++ implementation.

The data required for processing, such as buffer contents for amplitude descriptors or FFT magnitudes for frequency descriptors, can be provided either as a list of floats or as a one-dimensional NumPy array. Additionally, all helper functions present in the original C++ library have been included in the Python interface.

Like the C++ library, the Python version is released under the MIT License.

5.2 JavaScript bindings

From the C++ library also a JavaScript version was derived, using *Emscripten* [22] to build a WebAssembly binary and a JavaScript interface. The MIT-licensed library, called *informer.js*, like the Python version, mirrors the functions and the class implementation of its C++ counterpart. The only difference, in order to better comply with typical JavaScript conventions, is the use of camel case instead of snake case for all functions names.

A demo of how *informer.js* can be used is included in the repository [1]: it is a simple application where an audio file can be uploaded and analysed, resulting in a multi-row graph displaying all calculated descriptors alongside the audio waveform and spectrogram. The descriptors can be optionally stacked in the same graph, and a smoothing function can be applied. The audio file is processed and the FFT is performed using the Web Audio API within an `OfflineAudioContext`. It can be used to rapidly test the capabilities of the framework with any audio material.

6. CONCLUSIONS

This work presented *The Informer*, an open-source framework for the real-time computation and distribution of low-level audio descriptors across different software environments. Instead of providing a single-purpose tool, the solution encompasses a multi-platform plugin (VST3/AU/LV2), a standalone application, a VCV Rack module, Max/MSP externals, and a Max for Live receiver device, all unified by the same computational engine.

The descriptor algorithms were initially implemented in a multi-platform plugin and standalone application, and then encapsulated into a header-only C++ library forming the computational core of the entire ecosystem. This core is used across a VCV Rack module, a suite of Max/MSP externals, and to develop bindings for both Python and JavaScript (via WebAssembly). This structure ensures consistency of results across environments while promoting extensibility and long-term maintainability. The availability of bindings for multiple languages broadens accessibility, allowing the same descriptor definitions to be employed in real-time performance contexts, offline analysis pipelines, browser-based applications, and research workflows.

While there are different libraries and software packages providing audio feature extraction, existing solutions are typically confined to specific programming ecosystems, lack plugin-oriented real-time broadcasting capabilities, or are no longer actively maintained. In contrast, *The Informer* integrates real-time descriptor computation, cross-platform plugin deployment, OSC-based interoperability, and lightweight reusable code within a single coherent framework.

Initially conceived for audiovisual performance, the system's capabilities extend to scientific research, education, and sound design. Use cases range from dynamic parameter modulation in DAWs and interactive installations to biofeedback experiments and educational tools. The built-in normalization and OSC broadcasting features specifically facilitate interoperability with external systems such as video synthesis software, lighting rigs, and platforms for data visualization.

The open-source nature of the project encourages community contributions and adaptation to emerging creative and scientific contexts. By bridging technical audio analysis with creative applications, *The Informer* aims to ease the access to sound descriptors for artists, researchers, and developers alike.

[1] `https://valeriorlandini.github.io/theinformer/Informer.js/demo`

7. REFERENCES

[1] *Information technology — Multimedia content description interface — Part 4: Audio*, International Organization for Standardization, Geneva, Switzerland, 2002.

[2] G. Peeters *et al.*, "A large set of audio features for sound description (similarity and classification) in the CUIDADO project," *CUIDADO Ist Project Report*, vol. 54, no. 0, pp. 1–25, 2004.

[3] A. Lerch, *An introduction to audio content analysis*, 2nd ed. New York, NY: Wiley-IEEE Press, Nov. 2022.

[4] D. Grzywczak and G. Gwardys, "Audio features in music information retrieval," in *Active Media Technology*, D. Ślezak, G. Schaefer, S. T. Vuong, and Y.-S. Kim, Eds. Cham: Springer International Publishing, 2014, pp. 187–199.

[5] P. Couprie, "Designing sound representations for musicology," Jun. 2022.

[6] M. Wright and A. Freed, "Open Sound Control: A new protocol for communicating with sound synthesizers, greece," in *International Computer Music Conference 101*, vol. 104, 1997.

[7] M. Malt and E. Jourdan, "Zsa.descriptors: a library for real-time descriptors analysis." Zenodo, 2008.

[8] P. A. Tremblay, O. Green, G. Roma, J. Bradbury, T. Moore, J. Hart, and A. Harker, "The Fluid Corpus Manipulation toolbox," Jul. 2022.

[9] N. Schnell, A. Roebel, D. Schwarz, G. Peeters, and R. Borghesi, "MuBu and friends – assembling tools for content based real-time interactive audio processing in Max/MSP," *Proceedings of the International Computer Music Conference (ICMC 2009)*, 01 2009.

[10] N. Schnell, D. Schwarz, J. Larralde, and R. Borghesi, "PiPo, A Plugin Interface for Afferent Data Stream Processing Modules," in *International Symposium on Music Information Retrieval (ISMIR)*, Suzhou, China, Oct. 2017.

[11] Brian McFee, Colin Raffel, Dawen Liang, Daniel P.W. Ellis, Matt McVicar, Eric Battenberg, and Oriol Nieto, "librosa: Audio and Music Signal Analysis in Python," in *Proceedings of the 14th Python in Science*

Conference, Kathryn Huff and James Bergstra, Eds., 2015, pp. 18 – 24.

[12] D. Bogdanov, N. Wack, E. Gómez Gutiérrez, S. Gulati, H. Boyer, O. Mayor, G. Roma Trepat, J. Salamon, J. R. Zapata González, X. Serra *et al.*, "Essentia: An audio analysis library for music information retrieval," in *14th Conference of the International Society for Music Information Retrieval (ISMIR); 2013 Nov 4-8*. International Society for Music Information Retrieval (ISMIR), 2013.

[13] C. Cannam, C. Landone, and M. Sandler, "Sonic Visualiser: An open source application for viewing, analysing, and annotating music audio files," in *Proceedings of the ACM Multimedia 2010 International Conference*, Firenze, Italy, October 2010, pp. 1467–1468.

[14] P. Guillot, "Partiels - Exploring, Analyzing and Understanding Sounds," in *Proceedings of the 28-th Int. Conf. on Digital Audio Effects (DAFx25)*, L. Gabrielli and S. Cecchi, Eds., Sept 2025.

[15] A. Stark, "Sound Analyser: A plug-in for real-time audio analysis in live performances and installations," in *Proceedings of the International Conference on New Interfaces for Musical Expression*. London, United Kingdom: Goldsmiths, University of London, June 2014, pp. 183–186.

[16] A. de Cheveigné and H. Kawahara, "Yin, a fundamental frequency estimator for speech and music," *The Journal of the Acoustical Society of America*, vol. 111, no. 4, pp. 1917–1930, 04 2002. [Online]. Available: https://doi.org/10.1121/1.1458024

[17] T. Giannakopoulos and A. Pikrakis, *Introduction to Audio Analysis: A MATLAB Approach*, 1st ed. USA: Academic Press, Inc., 2014.

[18] G. Peeters, B. L. Giordano, P. Susini, N. Misdariis, and S. McAdams, "The Timbre Toolbox: Extracting audio descriptors from musical signals," *The Journal of the Acoustical Society of America*, vol. 130, no. 5, pp. 2902–2916, 11 2011.

[19] H. Misra, S. Ikbal, H. Bourlard, and H. Hermansky, "Spectral entropy based feature for robust ASR," in *2004 IEEE International Conference on Acoustics, Speech, and Signal Processing*, vol. 1, 2004, pp. I–193.

[20] K. Jensen and T. H. Andersen, "Real-time beat estimation using feature extraction," in *Computer Music Modeling and Retrieval*, U. K. Wiil, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 13–22.

[21] K. West and P. Lamere, "A model-based approach to constructing music similarity functions," *EURASIP J. Adv. Signal Process*, vol. 2007, no. 1, p. 149, Jan. 2007.

[22] A. Zakai, "Emscripten: an LLVM-to-JavaScript compiler," in *Proceedings of the ACM International Conference Companion on Object Oriented Programming Systems Languages and Applications Companion*, ser. OOPSLA '11. New York, NY, USA: Association for Computing Machinery, 2011, p. 301–312.