A Javascript Library for Flexible Visualization of Audio Descriptors

Gerard Roma
CeReNeM
University of Huddersfield
g.roma@hud.ac.uk

Anna Xambó
Centre for Digital Music
Queen Mary University of London
a.xambo@qmul.ac.uk

Owen Green
CeReNeM
University of Huddersfield
o.green@hud.ac.uk

Pierre Alexandre Tremblay
CeReNeM
University of Huddersfield
p.a.tremblay@hud.ac.uk

ABSTRACT

Research in audio analysis has provided a large number of ways to describe audio recordings, which can be used for enhancing their visual representation in web applications. In this paper we present *fav.js*, a Javascript library for flexible visualization of audio descriptors. We explain the proposed design and demonstrate its potential for web audio applications through several visualization examples.

1. INTRODUCTION

Decades of research on automatic speech recognition, environmental sound recognition, and particularly music information retrieval (MIR) have contributed to establish the notion of audio descriptors [13]. Audio descriptors can be generally seen as metadata related to a given audio recording. While descriptors can be obtained in many different ways, a large effort has been devoted to the automatic extraction of descriptors using signal processing techniques. Such descriptors (often known as acoustic features) are typically inspired, albeit sometimes loosely, by current understanding of human perception of sound, or by established concepts in music theory. In this sense, it is common to distinguish between low-level, mid-level and high-level descriptors, depending on how close they are to common language. Many toolboxes are available for automatic extraction [11].

While descriptors produced by automatic analysis have been extensively used in machine learning research, their use for visualization of sound in interactive applications is a promising direction, as evidenced by early work [6]. This direction has unfortunately received relatively little attention. Among other issues, some descriptors may be difficult to understand, or may need further processing or scaling. Some of them may not be relevant in the absence of musical sound or in noisy conditions.

Web Audio Conference WAC-2018, September 19–21, 2018, Berlin, Germany.

In this paper, we propose a flexible framework for visualization of audio descriptors in web applications. The framework is implemented in a Javascript library that allows processing and combining audio descriptors and drawing them in different styles. Our focus is not real-time visualization but displaying time series of descriptors obtained from existing recordings. While we are interested in the representation of sound collections, we do not focus on layout algorithms for positioning multiple sounds. Our hope is to allow web developers and researchers to experiment with currently available methods for obtaining sound descriptors (both client- and server-side), and use them for visualization in novel web audio prototypes. Potential application areas include audio content creation and distribution, as well as education.

The rest of the paper is organized as follows. In the next section we review existing research related with web-based audio visualization and visualization of audio descriptors in general. We then describe the design of the framework. Finally, we show some examples of visualization with fav.js, and reflect on future work.

2. RELATED WORK

Method chaining is a popular technique for designing concise object oriented APIs. This pattern can be used to build embedded Domain Specific Laguages (DSL) [8]. This design was popularized in the domain of web-based data visualization by the D3 library [3] and has been followed by many other libraries. A notable example related to our work is the DataToMusic API [16]. Among other things, this library implements several transformations of time series with a focus on sonification.

Visualization of audio and other time series data in web applications has been implemented in the WavesJS library.[1] An early version [15] was based on D3, and thus followed the method chaining approach. In [10], the data model was extended to integrate audio playback and interaction. The general library includes many useful components for web audio development, whilst *waves-ui* and *waves-blocks* expose the functionality for interactive visualizations. The semantics of these components mostly focus on configuration of layers including interaction. Like in D3, WavesJS visualizations

[1] https://github.com/wavesjs

leverage Scalable Vector Graphics (SVG) and the Document Object Model (DOM), which allow for complex application-oriented data models and abstractions. Our approach is different in that we focus on the specific task of rendering time series, which allows us to use the HTML canvas. At the same time, our system aims to help visualizing arbitrary audio descriptors, and thus offers a variety of drawing functions. As an example, two-dimensional features such as spectrograms can be rendered as bitmap images in the canvas.

Outside the web platform, the visualization of audio descriptors has been a common topic of research within the area of MIR, as discussed in [12], where a range of techniques for feature extraction (from either symbolic or audio representations) is presented, and common issues (e.g. forms, formats, dimensions of music) are discussed. Analyzing audio recordings has become an important task in musicology, where automatic audio analysis has helped to gain a better aural understanding [5].

One of the best known desktop applications for visualizing audio descriptors is Sonic Visualiser [4]. This program allows configuring several panes and layers with waveform and spectrogram displays. Many audio descriptors can be displayed in different layers thanks to vamp plugins.[2] EAnalysis [7] is a similar application that provides several pre-configured templates for laying out the visualization. Descriptors can also be obtained using vamp plugins. Both programs enable the use of audio descriptors in specific applications such as musicological analysis. By providing an open-ended Javascript library, as opposed to a user-level application, we hope to enable their use in new applications related to audio and music.

Computing the descriptors themselves is, however, beyond the scope of our library. Our goal is to leverage existing and future efforts for this task. At the time of writing, two libraries have been presented for client-based feature extraction in Javascript: JS-Xtract [9] and Meyda [14]. C++ libraries such as Essentia [2] can be compiled to Javascript via Emscripten. We used JS-Xtract in our examples. Given the computational cost, another possibility is to compute the descriptors on the server. For example, the Freesound API[3], provides access to descriptors computed for any sound in the Freesound database. While the default setting returns statistics of the descriptor time series, obtaining the full series is also possible.

3. DESIGN

The design of the library stems from our research on manipulating large collections of audio for music creation. In this context, we hope that improving on existing tools for interactive audio analysis can provide new opportunities for creative segmentation and manipulation of sounds. At the same time, visualization of audio descriptors has potential for applications to browsing audio in a broader sense. We hope to facilitate making available results from signal processing research to musicians, creative coders and web developers who may all have varying dispositions towards technology. In this sense, our design priority is ease of use, as well as easy integration with other tools. Thus, we focused on a lightweight codebase with no dependencies.

[2] https://www.vamp-plugins.org
[3] https://freesound.org/docs/api

Table 1: Summary of functionality

Signal			Display
Unary ops.		Binary ops.	Drawing functions
threshold	slice	add	wave
norm	offset	subtract	line
log	square	multiply	fill
pow	exp	over	range
sqrt	abs	and	image
scale	diff	or	errorbar
delay	smooth	xor	
schmitt	sample		
draw	slide		
reflect			

The library implements an internal DSL through method chaining. The main goal is to offer a concise way to express a route from a `Signal` object to a `Display` object (Figure 1). A `Signal` is basically a wrapper around one or several `Float32Array` objects, along with a sample rate and a type identifier. The type identifier distinguishes between binary, integer or real signals, but for simplicity these are always encoded as floats. Each `Float32Array` is assumed to vary in time according to the sample rate, and two-dimensional signals are represented by an `Array` of `Float32Array` objects (e.g. frequency channels). Operators modify each element in the signal, as is common in array languages and scientific computing environments. A summary of currently implemented functionality can be seen in Table 1. While the system could be used for visualization of other kinds of time series, the choice of operators and drawing functions is motivated by our focus on audio descriptors.

We define a number of unary operators to allow transformations of a signal, such as scaling, smoothing, thresholding, slicing or applying simple mathematical operations. A special unary operator is "sample", which upsamples or downsamples the `Signal`. This is obviously an important part in the process of visualization, as the desired width of the display will only rarely coincide with the length of the signal. The resampling process does not follow common audio resampling techniques, as the goal is to efficiently produce a visual representation. In order to accommodate non-integer ratios, the descriptor is split into subsequences of potentially different sizes. Then several statistics (e.g. mean, median, standard deviation) can be computed for each sequence. The same statistics are available for smoothing without resampling. For upsampling, the mean statistic is used, so the signal is linarly interpolated.

Binary operators allow combining two `Signals` by sample-wise arithmetic operations, including boolean arithmetic. This may be useful when one descriptor is not useful in parts of the sequence. For instance, a measure of pitch confidence, or an amplitude measure, can be used to select when a pitch descriptor is displayed. Descriptors from different channels of a recording can be combined to visualize the spatial image. Binary operators are mostly thought for combining descriptors of the same sound, so two `Signals` of the same length are required.

Finally, a `Display` is a container for several `Layer` objects. For an efficient notation, `Layers` can be accessed using array operators. A `Display` owns a DOM container element, and attaches an HTML canvas for each `Layer`. All `Layers` in a

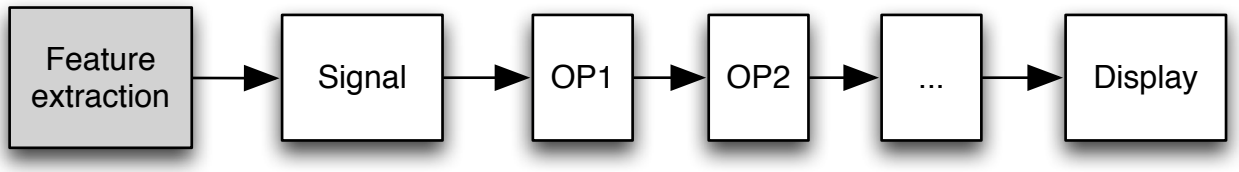


Figure 1: Visualization process

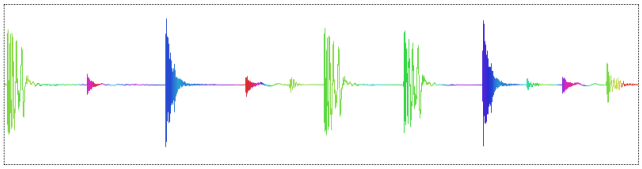
`Display` share the same position and dimensions, which are specified in the constructor. If the width is not specified it will be determined for all `Layers` the first time a `Signal` is drawn. Otherwise, the `sample` operator is used internally to scale the `Signal` to the desired width. Different `Layer` types can be used for different kinds of drawings, some restricted to particular dimensions. Available `Layer` types are "wave", for drawing waveforms; "line", "errorbar" and "fill", for unipolar time series; and "image", for two-dimensional signals such as spectrograms. For one-dimensional signals, a second signal can be provided to control the color. In this case, as well as with two-dimensional signals, the color mapping can be controlled using hue, saturation and lightness (HSL), which is available for the HTML5 canvas in all major browsers. Each of the three parameters allows for intuitive visual mappings, however for simplicity we focus on lightness, allowing the user to specify the hue. Drawing is triggered by an operator on the `Signal`, which potentially results from several operations. `Display` objects offer minimal interaction capabilities, allowing the developer to attach a function to click and drag events. Unlike other frameworks like WavesJS, the goal for fav.js is to focus on transformation and visualization. This obviously does not preclude the development of interactions such as zooming or scrolling.

4. EXAMPLES

In this section we show some examples of the utilization of the library. All examples assume an "audio" array, that has been obtained by decoding a buffer, and a "getSignal" function that returns a `Signal` object with some descriptor. Descriptors were computed using JS-Xtract, but as long as `Float32Array` and a sample rate can be provided, they can be obtained in many other ways. The sample rate for the loaded audio is obtained from the `AudioContext`.

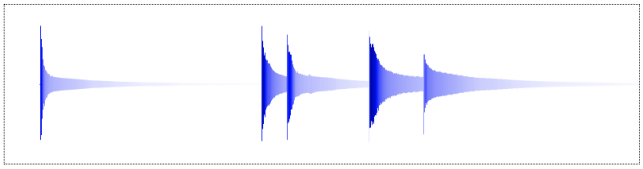
One example of descriptor-based visualization is provided by the Freesound project [1], where the waveform is colored by the spectral centroid. The idea can be traced back to *Timbregrams* described in [6]. With the proposed framework, any descriptor (or combination of descriptors) can be used to color waveforms. Figure 2 shows an example using the spectral centroid with a drum pattern. The descriptor is mapped to the hue, which has a range of 360 degrees. As a result, each drum instrument gets a different color. Another combination is shown in Figure 3. Here the root mean square (RMS) amplitude descriptor is used to control the lightness. The information is a bit redundant with the waveform, so the visual effect reinforces the decay of the notes.

Another common application is segmentation. Thresholding the RMS signal yields a binary signal that can be used to visualy identify sound objects. In Figure 4, this technique is applied to color a grayscale spectrogram. On the other hand, the ability to transform and combine descriptors makes it possible to use the library interactively to develop more complex forms of object selection in order to obtain better accuracy. An example is shown in Figure 5. We show only part of the code for saving space, the full example can be obtained with the library code. Here, the `slice` operator is used to zoom into the signal in a second display, and observe the effect of the different operations. The original RMS is shown in the pale blue shade, and the first order derivative in the dark blue line. The smoothed



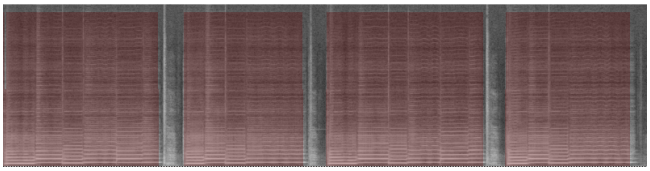
```
let sc = getSignal(audio, "spec_centroid");
let wave = new fav.Signal(audio, sampleRate);
let display = new fav.Display("container",
              "wave", 800, 200);
wave.draw(display,
  [sc.smooth(20)
     .normalize()
     .scale(360),
    70, 50
  ]);
```

Figure 2: Waveform coloring with Spectral Centroid



```
let rms = getSignal(audio, "rms");
let wave = new fav.Signal(audio, sampleRate);
let display = new fav.Display("container",
              "wave", 800, 200);
wave.draw(display,
  [237, 100, rms.normalize()
               .reflect()
               .scale(70)
               .offset(30)
  ]);
```

Figure 3: Waveform luminance coloring using RMS



```
let spgm = getSignal(audio, "spectrum");
let rms = getSignal(audio, "rms");
let display = new fav.Display("container",
              "image", 800, 200);
display.addLayer("fill");
spgm.log()
    .normalize()
    .draw(display[0]);
rms.smooth(20)
   .threshold(0.05)
   .draw(display[1],"rgba(100,0,0,0.3)");
```

Figure 4: Spectrogram with basic RMS thresholding

derivative is thresholded then combined with the original RMS and thresholded again. The resulting signal is able to segment the decay of a snare drum sound.

Finally, Figure 6 shows an experimental example that uses errorbar layers to illustrate a selection of related, yet eclectic, sounds of bowed cardboard. Several descriptors (spectral centroid, zero-crossing rate, spectral skew and RMS energy) are used as a visual fingerprint for assessing the relatedness of somewhat disparate sounds from the same corpus. It is interesting to see that features which are strongly correlated in one sound may not be in another.

5. CONCLUSIONS

The combination of existing signal processing techniques for description of sound and music with the visualization power of current web technologies creates a great opportunity for interactive web audio applications. In this paper we have proposed a framework to make this possible, implemented in a lightweight Javascript library. We have shown some examples of potential applications. We plan to continue this work experimenting with other sources of audio descriptors. Also, since the Javascript language is also available in the Max/MSP environment for user interface graphics, we plan to adapt our library to this environment. The library can be obtained from https://github.com/flucoma/fav.js.

6. ACKNOWLEDGEMENT

This research was part of the Fluid Corpus Manipulation project (FluCoMa), which has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No 725899).

7. REFERENCES

[1] V. Akkermans, F. Font Corbera, J. Funollet, B. De Jong, G. Roma Trepat, S. Togias, and X. Serra. Freesound 2: An improved platform for sharing audio clips. In *Proceedings of the 12th Conference of the International Society for Music Information Retrieval (ISMIR)*, 2011.

[2] D. Bogdanov, N. Wack, E. Gómez Gutiérrez, S. Gulati, P. Herrera Boyer, O. Mayor, G. Roma Trepat, J. Salamon, J. R. Zapata González, and X. Serra. Essentia: An Audio Analysis Library for Music Information Retrieval. In *Proceedings of the 14th Conference of the International Society for Music Information Retrieval (ISMIR)*, 2013.

[3] M. Bostock, V. Ogievetsky, and J. Heer. D^3 Data-Driven Documents. *IEEE transactions on visualization and computer graphics*, 17(12):2301–2309, 2011.

[4] 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 18th ACM international conference on Multimedia*, 2010.

[5] N. Cook. Methods for Analysing Recordings. In *The Cambridge Companion to Recorded Music*, pages 221–245. Cambridge University Press, 2009.

[6] M. Cooper, J. Foote, E. Pampalk, and G. Tzanetakis. Visualization in Audio-Based Music Information Retrieval. *Computer Music Journal*, 30(2):42–62, 2006.

[7] P. Couprie. EAnalysis : Aide à l'Analyse de la Musique Électroacoustique. In *Actes des Journées d'Informatique Musicale*, 2012.

[8] M. Fowler. *Domain-Specific languages*. Addison-Wesley Professional, 2010.

[9] N. Jillings, J. Bullock, and R. Stables. JS-Xtract: A Realtime Audio Feature Extraction Library for the Web. In *Proceedings of the 17th Conference of the International Society for Music Information Retrieval (ISMIR)*, 2016.

[10] B. Matuszewski, N. Schnell, and S. Goldszmidt. Interactive Audiovisual Rendering of Recorded Audio and Related Data with the WavesJS Building Blocks. In *Proceedings of the 2nd Web Audio Conference (WAC)*, 2016.

[11] D. Moffat, D. Ronan, J. D. Reiss, et al. An Evaluation of Audio Feature Extraction Toolboxes. In *Proceedings of the 18th International Conference on Digital Audio Effects*, 2015.

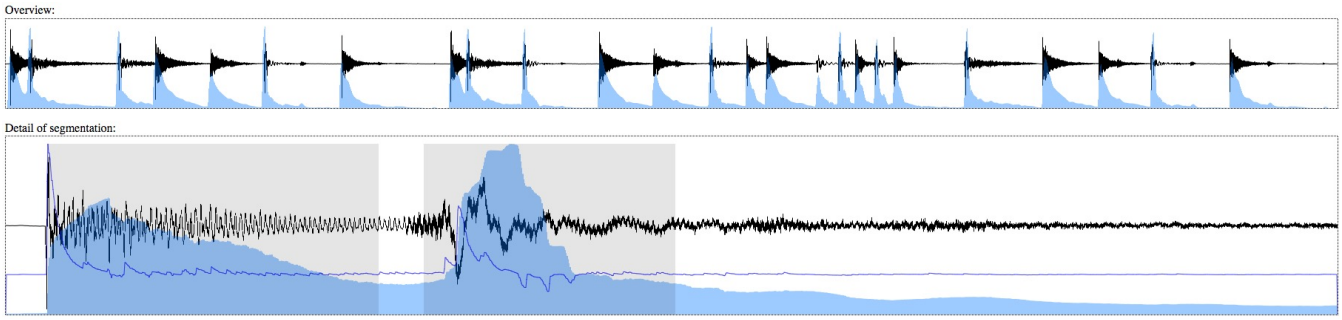
[12] N. Orio. Music Retrieval: A Tutorial and Review. *Foundations and Trends® in Information Retrieval*, 1(1):1–90, 2006.

[13] G. Peeters. A large set of audio features for sound description (similarity and classification) in the cuidado project. Technical report, IRCAM, 2004.

[14] H. Rawlinson, N. Segal, and J. Fiala. Meyda: An Audio Feature Extraction Library for the Web Audio API. In *Proceedings of the 1st Web Audio Conference (WAC)*, 2015.

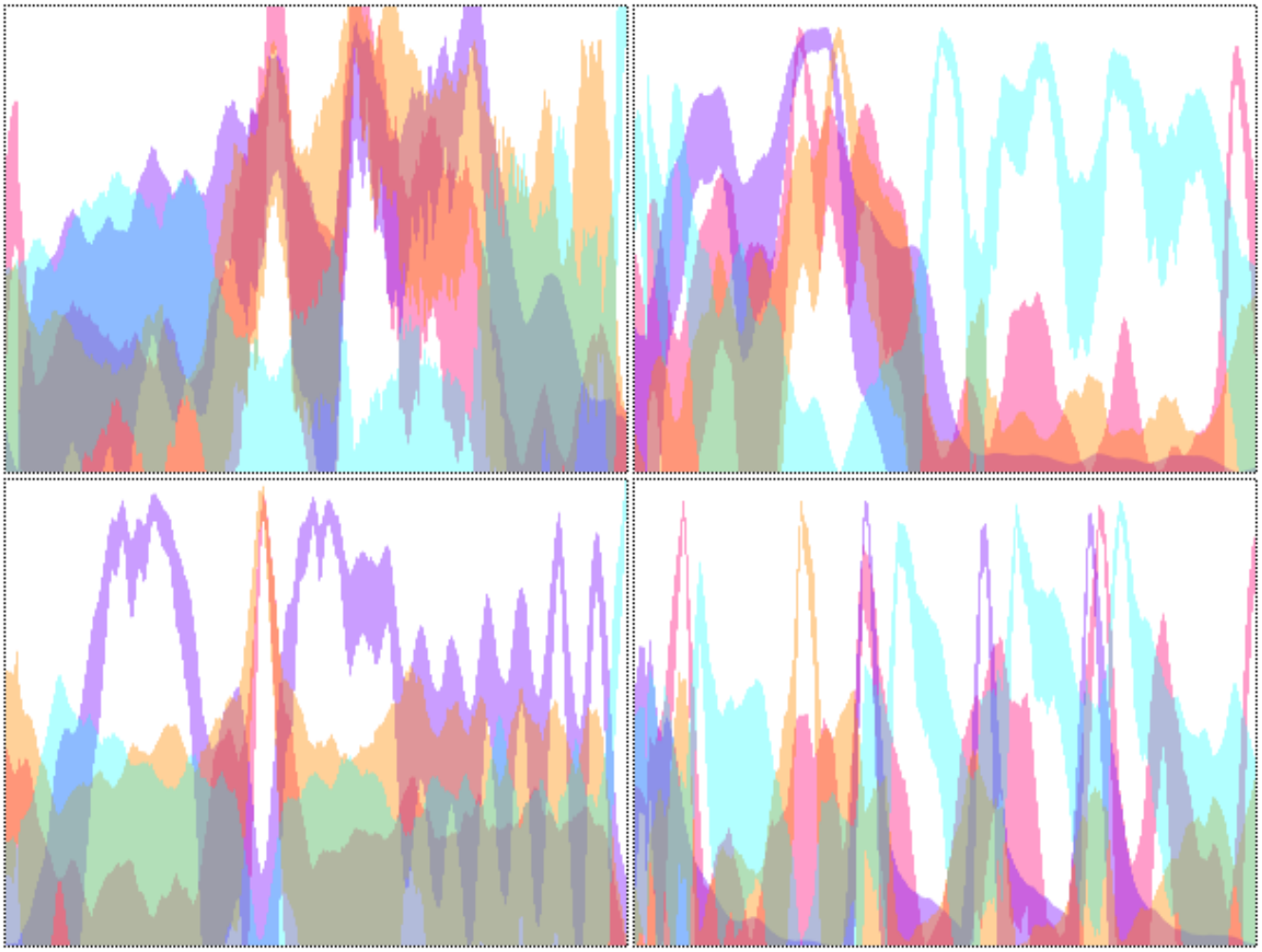
[15] V. Saiz, B. Matuszewski, and S. Goldszmidt. Audio Oriented UI Components for the Web Platform. In *Proceedings of the 1st Web Audio Conference (WAC)*, 2015.

[16] T. Tsuchiya, J. Freeman, and L. W. Lerner. Data-to-Music API: Real-time Data-Agnostic Sonification with Musical Structure Models. In *Proceedings of The 21st International Conference on Auditory Display (ICAD)*, 2015.




```
let part = wave.slice(0,0.5,"seconds")
part.draw(displayZoom,"black")
let partRMS = rmsM.slice(0,0.5,"seconds")
partRMS
.draw(displayZoom,"rgba(9, 123, 255, 0.4)", 1)
.diff().normalize().slide(1,10)
.draw(displayZoom, "rgba(0, 0, 255, 0.7)",2)
.schmitt(0.6,0.3).or(partRMS.slide(1,10).threshold(thresh))
.draw(displayZoom, "rgba(0, 0, 0, 0.1)",3)
```

Figure 5: Custom RMS-based segmentation



```
let zcr = getSignal(audio,"zcr");
let rms = getSignal(audio,"rms");
let centroid = getSignal(audio,"spectral_centroid");
let skew = getSignal(audio,"spectral_skewness");
let wave = new fav.Signal(audio, sampleRate);
let display = new fav.Display(container,"errorbar", 300, 225);
for(i = 0; i < 3; i++) display.addLayer("errorbar");
zcr.draw(display[0],"rgba(255,9,123,0.4)");
rms.draw(display[1],"rgba(123,9,255,0.4)");
centroid.draw(display[2], "rgba(255,140,0,0.4)");
skew.draw(display[3], "rgba(0,255,255,0.3)");
```

Figure 6: Multiple errorbar patterns