LFO – A Graph-based Modular Approach to the Processing of Data Streams
Matuszewski, Benjamin; Schnell, Norbert

Attribution-NonCommercial-NoDerivs 3.0 United States

For additional information about this publication click this link.
http://qmro.qmul.ac.uk/xmlui/handle/123456789/26126

Information about this research object was correct at the time of download; we occasionally make corrections to records, please therefore check the published record when citing. For more information contact scholarlycommunications@qmul.ac.uk
LFO – A Graph-based Modular Approach to the Processing of Data Streams

Benjamin Matuszewski
CIM/musidance EA1572, Université Paris 8, UMR STMS IRCAM-CNRS-UPMC
Paris, France
benjamin.matuszewski@ircam.fr

Norbert Schnell
UMR STMS IRCAM-CNRS-UPMC
Paris, France
norbert.schnell@ircam.fr

ABSTRACT
This paper introduces lfo — for Low Frequency Operators — a graph-based Javascript (ES2015) API for online and offline processing (i.e. analysis and transformation) of data streams such as audio and motion sensor data. The library is open-source and entirely based on web standards. The project aims at creating an ecosystem consisting of platform-independent stream operator modules such as filters and extractors as well as platform-specific source and sink modules such as audio i/o, motion sensor inputs, and file access. The modular approach of the API allows for using the library in virtually any Javascript environment. A first set of operators as well as basic source and sink modules for web browsers and Node.js are included in the distribution of the library. The paper introduces the underlying concepts, describes the implementation of the API, and reports on benchmarks of a set of operators. It concludes with the presentation of a set of example applications.

CCS Concepts
• Information systems → Multimedia and multimodal retrieval; • Software and its engineering → Real-time systems software; Software libraries and repositories;

Keywords
HTML5, Web Audio API, Digital Signal Processing, Javascript library

1. INTRODUCTION
In many domains, software environments provide the possibility to integrate components that are compliant to a specific API to extend them by particular functionalities. Such interfaces may be used to extend host environments such as operating systems, applications, and libraries. The interest of such interfaces is two fold. From the point of view of the host environment, they allow for extending its functionalities by components that can be developed and distributed independently. On the other hand, developers can focus on the implementation of specific extensions and significantly reduce their dependency on platform-specific issues. In some cases, such extensions are referred to as plugins. In case that multiple host environments implement a given plugin interface, a compliant plugin can be integrated in either of the environments.

Many sound editing and music performance applications implement one or multiple plugin interfaces (e.g. VST, AudioUnits, and LADSPA) for real-time and offline audio processing. The VAMP interface [1] initially has been created for the SonicVisualiser [2] audio editing and visualization environment. The interface allows for implementing plugins that extract information from audio streams and produce multi-dimensional data streams. Apart from its initial host environment, the interface has been implemented by several applications such as Audacity³ and Sonic Annotator.²

The PiPo interface [11] provides a more general formalization of the processing of multi-dimensional data streams. While other audio processing interfaces only consider audio streams, the PiPo interface allows modules to consume and to produce streams of different dimensions and rates. Apart from some experimental implementations, the interface has been integrated into the Max³ audiovisual programming environment and existing modules are dedicated to the processing of audio and motion data streams.

The recent evolution of web standards and Javascript APIs such as Web Audio [9] and DeviceOrientation [12] permitted the development of audio and music applications that were previously available as native applications only. In the past years, the web platform has integrated elaborated audio and music applications that could profit from the formalization of plugin interfaces such as, for example, audio tools [3, 5, 4] and collaborative audio editors [7]. Unfortunately, available audio processing libraries such as Meyda [8] and tone.js [5] are not always modular nor provide an interface to easily integrate additional components.

The lfo library defines a modular framework to develop processing modules that can be connected to graphs. The interface provides a general formalization of multidimensional streams and, abstractions (i.e. Javascript classes) for modules that transform these streams. As the PiPo interface, the design of lfo focuses on processing that transforms incoming data streams or extracts information from them to produce reduced output streams of relatively low rate. The available modules distributed with the library implement

---

1 http://www.audacityteam.org/
2 http://www.vamp-plugins.org/sonic-annotator/
3 https://cycling74.com/products/max/
filters, audio descriptor extractors, and platform dependent inputs and outputs.

2. CONCEPTS AND FORMALIZATION

The basic idea, lfo, shares with the PiPo library, is to create a framework for software modules that implement algorithms to be applied to different data streams in interactive systems. Typically, these data streams are generated by input peripherals such as sensors and other input devices or ready from files and databases. The algorithms used to process the data streams usually reduce their information rate and/or dimensionality in terms of operations such as filtering, mapping, or analysis. The processed data may drive the generation and modulation of audiovisual rendering, such as audio synthesis and visualizations, or may be used to control output peripherals such as actuators and motors. Especially for the purpose of simulation, debugging and batch processing, data streams may also be read from and stored into files and databases. While some transformations, such as simple filters or scaling, solely alter the values of incoming data frames, others may reduce their dimensionality (e.g. an RMS or a projection onto principal components) or even produce a data stream of a different temporality. A sample rate converter, for example, would transform an incoming stream of frames in a given rate into a stream of a different rate. An onset detector could receive a stream of a constant sample rate to produce a stream of aperiodic time-tagged markers.

Graphs of Modules

The lfo API implements a graph based approach. As shown in figure 1, a graph generally consists of a source module, one or more operator modules and/or a terminating sink module. While the output of one module can be connected to the inputs of multiple other modules, each module accepts only a single input.

A source module usually acts as an interface to an underlying — usually platform-dependent — API for data acquisition, such as device sensor APIs (e.g. DeviceMotion, DeviceOrientation), network reception, and file input. Other source modules, such as oscillators, may output data that is generated algorithmically. A source module defines the attributes of the source data stream, such as frame rate and frame size, and outputs a stream of frames into the modules connected to its output. The operator modules usually implement platform-independent algorithms that process an incoming data stream and output a transformed stream. An operator can accept any kind of stream at its input or be limited to a certain kind of input stream. While the output stream of some operators inherit the attributes of the input stream, other operators may output a different kind of stream defining new attributes. A sink module often acts as an interface to system outputs such as peripherals, network transmission, file or database storage. Moreover, the visual rendering of a data stream is usually implemented as a sink.

Insofar the platform-dependencies of a graph are limited to its source and sink modules, stream processing that is implemented by platform-independent operator modules can easily be applied across different platforms and within different configurations on the same platform. For example, the same stream processing (i.e. an operator or a chain of operators) that is applied to the accelerometer inputs of a mobile web-client to control the parameters of a Web Audio synthesis engine can be easily adapted to render the data on a graphical display (e.g. for debugging), or to transform a data stream read from a file by a Node.js application to control a synthesizer via a MIDI connection.

Streams of Frames

In the formalization of data streams underlying the lfo API, a stream is a succession of frames that consists of a time-tag (i.e. a timestamp associating each frame in a stream to a point in time regarding an arbitrary reference common to all modules of a graph) and data (i.e. the payload of the frame). Currently, the frame data can have one of three types:

- **vector**: an array of values corresponding to different dimensions (e.g. $x, y, z$ or $\text{mean}, \text{stddev}, \text{min}, \text{max}$)
- **signal**: an array of time-domain values corresponding to a fragment of a signal
- **scalar**: a single value that can be arbitrarily considered as a one-dimensional vector or as a signal fragment of one sample

The data stream produced by an lfo module is characterized by a set of stream parameters:

- **frame size**: number of values in the frame data (i.e. number of vector dimensions or signal samples).
- **frame rate**: number of frames per second for streams of regularly sampled frames (0 otherwise).
- **frame type**: type of the frame (i.e. vector, signal or scalar).
- **source sample rate**: sample rate of the source stream (if any) in samples per second
- **source sample count**: block size of the source stream
- **description**: an array of strings describing the output dimensions of vector and scalar frames (e.g. ['x', 'y', 'z'], ['mean', 'stddev', 'min', 'max'])

The output stream parameters of a module are propagated to its connected modules. An operator module can define its output stream parameters as a function of its incoming stream parameters.

3. IMPLEMENTATION

The distribution of the lfo library provides four different entry points (see example in figure 2):

- **waves-lfo/common** contains all platform-independent operators and a few source and sink modules which do not depend on platform specific functions (e.g. EventIn, Logger).
- **waves-lfo/client** additionally exposes source and sink modules specific to web-browsers, such as bindings with the Web Audio API and real-time canvas...
rendering of the output streams (e.g. AudioInNode, WaveformDisplay, VuMeterDisplay).

- waves-lfo/node includes bindings with the file system such as audio file loading and decoding as well as reading and writing of json and csv files.
- waves-lfo/core exposes the BaseLfo class and should be imported when creating extensions of the library.

```javascript
import * as lfo from 'waves-lfo/node';
```

Figure 2: A code example that shows how to import the waves-lfo/node entry point, giving access to source and sink modules specific to the Node.js platform.

The high-level API of the library is strongly inspired by the Web Audio API interface. To create a graph, modules are connected from source to sink (see example in figure 3) using the connect method inherited from the BaseLfo class. However, while an audio graph usually connects multiple inputs to a single output, lfo graphs are composed of a single source that can be connected to multiple outputs. Similarly to the Web Audio API, source modules expose start and stop methods that allow to control the state of the graph.

```javascript
import * as lfo from 'waves-lfo/client';
const audioInBuffer = new lfo.source.AudioInBuffer({
  audioBuffer: audioBuffer,
  frameSize: 512,
  frameType: 'audioBuffer'
});
const rms = new lfo.operator.Rms();
const logger = new lfo.sink.Logger({
  data: true
});
audioInBuffer.connect(rms);
logger.connect(rms);
audioInBuffer.start();
logger.start();
```

Figure 3: A code example that highlights the creation of a basic lfo graph that computes the Root Mean Square of signal frames of 512 samples and log the result in the console. The graph is composed of a source, the AudioInBuffer, an operator, the Rms, and a sink, the Logger.

The example in figure 3 also shows how a graph is initialized and started. When the init method is called, the source makes sure that each module of the graph is ready to handle incoming frames. Since the initialization of certain modules can be asynchronous, promises are used to synchronize the initialization of the graph. As part of the initialization, the source also propagates its stream parameters to the modules connected to its outlet. During this step, each module can define its own stream parameters and allocate additional memory. Stream parameters are propagated step by step through the graph down to the sink. When all modules of the graph finish their initialization — beginning from the the sink(s) and successively resolving the promises back to the source —, the source can safely start to produce frames and propagate them through the graph.

This initialization procedure allows for optimizing the performance of an lfo graph when processing an incoming data stream.

Independent of the data type (i.e. vector, signal or scalar), the data sent within the frames from one module to another is implemented as a Float32Array. The time-tag associated to each frame — a Number — is usually defined by the source of the graph using a high precision clock such as performance.now or process.hrtime.

Standalone Module API

As show in the example in figure 4, many operators can be used as standalone modules allowing for applying their algorithm to arbitrary data buffers without connecting the modules to an lfo graph. This alternative API allows, for example, to use the same implementation of a filter — including the initialization and state — as an lfo module or as a filter object that can be applied to any succession of values or arrays without explicitly using the lfo formalism. Moreover, compound lfo operators can use the standalone API of other operators. Our implementation of an Mfcc module, for example, uses the standalone API of the Fft, Mel and Dct operators.

```javascript
import * as lfo from 'waves-lfo/common';
const rms = new lfo.operator.Rms();
rms.initStream({ frameType: 'signal', frameSize: 1000 });
// process some data
const result = rms.inputSignal([...values]);
```

Figure 4: A code example that highlights the use of an operator, here the Rms operator, with the standalone API.

Bring Your Own Modules

An important motivation for the development of lfo was the idea to create an ecosystem of stream processing modules. While the development of applications can benefit from a rich set of modules implementing different functionalities, developers of processing algorithms may appreciate an environment to test and debug their code as well as to easily compare its performance with available implementations of similar features. In this sense, the design of the library aims at creating an easy-to-use API on both ends, the integration of existing modules into an application and the development and distribution of new modules.

As shown in the example of figure 5, an lfo module implements an ES2015 class that extends the BaseLfo class. The base class constructor must be called with the definition of the module’s parameters and the user defined options that override the default parameters. To handle incoming data stream of different types, a module has to implement pre-defined interfaces and follow a small set of conventions. To process incoming vector frames the derived class has to implement the processVector interface. Similarly, processing signal frames requires the derived class to implement the processSignal interface.

According to the type of frames output by the previous module, incoming frames will be routed to the appropriate processing method. If the module does not implement the required interface, an error is thrown when the source is started. To handle scalar frames — which can be considered as vector as well as signal frames — the module first looks for a processScalar method, but falls back on processVector or processSignal if processScalar is not implemented.

For consistency as well as for performance considerations, a module should comply with the following rules:

- Any memory allocation that depends on the modules configuration and/or its stream parameters — for example to create internal ring buffers — should be done in the processStreamParams method called during the

---

5 Parameters are defined using a small library that implements casting and constraints for a small set of types such as integer, float, enum — https://github.com/ircam-js-tools/parameters
The results are expressed in number of iterations per second which, given the nature of an iteration, easily allows to evaluate the real-time performance of the implemented algorithms. To get consistent and comparable results across different libraries, only the processing functions — using the standalone API of the lfo modules — are benchmarked. The measured algorithms, implemented by all tested libraries, are an Fft, an Rms, and Mfcc extraction.

The benchmarks have been performed on a MacBook Pro with a 2.8 GHz Intel Core i7 CPU and 16 GB of 1600 MHz DDR3 RAM under MacOS 10.11.6. The tested Javascript environments are Node.js (8.1.4), Google Chrome (59.0.3071.115, 64-bit), Firefox (54.0.1, 64-bit), and Safari (10.1). Detailed results are reported in table 1.

![Figure 5](https://benchmarkjs.com/)

### Table 1: Number of iterations per second for computing the FFT, RMS and Mfcc of a sound file of 1 seconds at different frame sizes with no overlapping. The benchmarks compare three libraries, PiPo, Meyda and lfo.

<table>
<thead>
<tr>
<th>Frame size</th>
<th>256</th>
<th>1024</th>
<th>4096</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>FFT</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PiPo (C/+++)</td>
<td>4.078</td>
<td>3.494</td>
<td>3.068</td>
</tr>
<tr>
<td>Node.js</td>
<td>330</td>
<td>285</td>
<td>266</td>
</tr>
<tr>
<td>Chrome</td>
<td>129</td>
<td>123</td>
<td>79.10</td>
</tr>
<tr>
<td>Firefox</td>
<td>542</td>
<td>503</td>
<td>446</td>
</tr>
<tr>
<td>Safari</td>
<td>673</td>
<td>623</td>
<td>562</td>
</tr>
<tr>
<td><strong>RMS</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PiPo (C/+++)</td>
<td>28.662</td>
<td>28.498</td>
<td>31.954</td>
</tr>
<tr>
<td>Node.js</td>
<td>11.640</td>
<td>13.844</td>
<td>15.328</td>
</tr>
<tr>
<td>Chrome</td>
<td>13.570</td>
<td>17.098</td>
<td>19.887</td>
</tr>
<tr>
<td>Firefox</td>
<td>9.547</td>
<td>18.774</td>
<td>27.317</td>
</tr>
<tr>
<td>Safari</td>
<td>28.284</td>
<td>21.691</td>
<td>15.472</td>
</tr>
<tr>
<td><strong>Mfcc</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PiPo (C/+++)</td>
<td>2.622</td>
<td>2.604</td>
<td>2.219</td>
</tr>
<tr>
<td>Node.js</td>
<td>301</td>
<td>279</td>
<td>268</td>
</tr>
<tr>
<td>Chrome</td>
<td>382</td>
<td>366</td>
<td>348</td>
</tr>
<tr>
<td>Firefox</td>
<td>449</td>
<td>473</td>
<td>444</td>
</tr>
<tr>
<td>Safari</td>
<td>543</td>
<td>538</td>
<td>508</td>
</tr>
</tbody>
</table>

The comparison of the lfo operators to the PiPo C/++ implementations shows that, the library performs 5.4 to 13.5 times slower than native code for the FFT, 1.1 times to 3 times slower for the RMS and 4.4 to 9.3 times slower for the Mfcc extraction. In all, these results show that the library performs well compared to native binaries, while leaving room for improvements. The comparison to Meyda shows that the lfo operators largely outperform the functions provided by the currently most popular Javascript library available. The particularly bad performance of Meyda’s RMS is certainly due to the fact that the

---

6 https://github.com/nickbrunn/hayai
7 https://benchmarkjs.com/
library calculates an FFT for all audio features even if the result of the FFT is not used by the requested feature.

The benchmarks show large differences between different environments. On average, Safari is 1.2 times faster and Firefox 1.4 times faster than Google Chrome. The relatively poor performance of environments based on the V8 Javascript engine (i.e. Google Chrome and especially Node.js) are particularly unexpected.

**Mobile Devices**

Since many of our envisaged use cases imply mobile devices, our benchmarks include mobile browsers on Android and iOS. The performance of the library was measured using the same test program on both platforms. The benchmarks on Android (6.0.1) use Google Chrome (59.0.3071.125) and Firefox (54.0.1) running on a Samsung A3 (2017) with a octa-core 1.6 GHz CPU and 2 GB RAM (referred to as device A). On iOS (10.3.2) we used Safari running on an iPhone 5C with a dual-core 1.3 GHz CPU and 1 GB RAM (referred to as device B). Detailed results are reported in table 2.

<table>
<thead>
<tr>
<th>Frame size</th>
<th>256</th>
<th>1024</th>
<th>4096</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>FFT</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Chrome (A)</td>
<td>37.95</td>
<td>32.80</td>
<td>27.85</td>
</tr>
<tr>
<td>Firefox (A)</td>
<td>41.21</td>
<td>36.25</td>
<td>30.98</td>
</tr>
<tr>
<td>Safari (B)</td>
<td>41.04</td>
<td>34.50</td>
<td>30.76</td>
</tr>
<tr>
<td><strong>RMS</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Chrome (A)</td>
<td>1,249</td>
<td>1,512</td>
<td>1,698</td>
</tr>
<tr>
<td>Firefox (A)</td>
<td>933</td>
<td>1,543</td>
<td>1,856</td>
</tr>
<tr>
<td>Safari (B)</td>
<td>887</td>
<td>1,612</td>
<td>1,748</td>
</tr>
<tr>
<td><strong>MFCC</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Chrome (A)</td>
<td>33.29</td>
<td>31.39</td>
<td>27.45</td>
</tr>
<tr>
<td>Firefox (A)</td>
<td>36.24</td>
<td>34.48</td>
<td>30.97</td>
</tr>
<tr>
<td>Safari (B)</td>
<td>35.24</td>
<td>33.37</td>
<td>30.65</td>
</tr>
</tbody>
</table>

**Table 2:** Number of iterations per second on mobile devices for computing the FFT, RMS, MFCC of a sound file of 1 seconds at different frame sizes with no overlapping.

As expected, measured results are drastically slower by an order of magnitude compared to laptop results, and however they show that the library remains usable even in this constrained context. As an example, the slowest measurement (i.e. MFCCs in *Google Chrome* with a frame size of 4096 samples, or about 93 ms) still performs 27 times faster than real-time with a processing time of approximately 3.4 ms for each frame.

**WebAssembly**

The [WebAssembly](https://github.com/kripken/emscripten) open standard [13], by enabling the usage of a binary format inside browsers, offers a new way to improve the performance of algorithms while remaining platform independent. The technology has been released as a minimum viable product in Mars 2017. Even if the WebAssembly implementations of some [lfo](https://cdn.rawgit.com/wavesjs/waves-lfo/main/examples/mosaicking/index.html) operators are still too experimental to be shipped with the library, they give an idea of the performance improvements enabled by this technology. For this purpose, we have developed an alternative implementation of the FFT [lfo](https://github.com/kripken/emscripten) operator based on a standard C implementation of the algorithm compiled to [WebAssembly (.wasm)](https://github.com/kripken/emscripten) using [emscripten].

Benchmarks were run using the same setup as in previous tests, using *Google Chrome* and *Firefox* on the MacBook Pro and the Android device. Results are shown in table 3.

**Table 3:** Number of iterations per second for computing the FFT of a sound file of 1 seconds at different frame sizes with no overlapping. For each tested browser, performance of C/C++ and vanilla Javascript are compared to the performance of the *WebAssembly* execution.

<table>
<thead>
<tr>
<th>Frame size</th>
<th>256</th>
<th>1024</th>
<th>4096</th>
</tr>
</thead>
<tbody>
<tr>
<td>PiPo (C/C++)</td>
<td>4,078</td>
<td>3,494</td>
<td>3,068</td>
</tr>
<tr>
<td>Chrome</td>
<td>js</td>
<td>wasm</td>
<td></td>
</tr>
<tr>
<td></td>
<td>455</td>
<td>388</td>
<td>358</td>
</tr>
<tr>
<td></td>
<td>826</td>
<td>720</td>
<td>673</td>
</tr>
<tr>
<td>Firefox</td>
<td>js</td>
<td>wasm</td>
<td></td>
</tr>
<tr>
<td></td>
<td>542</td>
<td>503</td>
<td>446</td>
</tr>
<tr>
<td></td>
<td>1,166</td>
<td>1,044</td>
<td>970</td>
</tr>
<tr>
<td>Chrome Android</td>
<td>js</td>
<td>wasm</td>
<td></td>
</tr>
<tr>
<td></td>
<td>37.95</td>
<td>32.80</td>
<td>27.45</td>
</tr>
<tr>
<td></td>
<td>67.84</td>
<td>60.20</td>
<td>50.66</td>
</tr>
<tr>
<td>Firefox Android</td>
<td>js</td>
<td>wasm</td>
<td></td>
</tr>
<tr>
<td></td>
<td>41.21</td>
<td>36.25</td>
<td>30.98</td>
</tr>
<tr>
<td></td>
<td>82.84</td>
<td>77.86</td>
<td>67.78</td>
</tr>
</tbody>
</table>

Measurements show an improvement of overall performances by a factor of 1.9 on average. This confirms that the library would greatly benefit from using small WebAssembly modules as algorithmic units inside operators.

**5. EXAMPLE APPLICATIONS**

To illustrate some use cases of the library and its integration into applications, we present a set of example applications.

**Mosaicking**

The example application shown in figure 7 implements interactive *audio mosaicking*. This synthesis technique refers to the process of recomposing the temporal evolution of a given *target* audio file from segments cut out of *source* audio materials [10]. In the application, the user can record an excerpt of speech that is analysed by an [lfo](https://cdn.rawgit.com/wavesjs/waves-lfo/main/examples/mosaicking/index.html) graph to extract the MFCCs of successive frames. The result of the analysis is used to control a concatenative synthesis process that recomposes the recorded excerpt (i.e. the *target*) using grains extracted from another audio file (i.e. the *source*). The source file is analysed server-side in *Node.js* using the same [lfo](https://cdn.rawgit.com/wavesjs/waves-lfo/main/examples/mosaicking/index.html) graph as on the client-side. The application highlights how a graph of operators can be reused across different environments by simply providing platform-specific sources and sinks.

![Figure 7: Screenshot of an example application that implements audio mosaicking, a concatenative synthesis technique based on MFCC descriptors of two audio files.](https://cdn.rawgit.com/wavesjs/waves-lfo/main/examples/mosaicking/index.html)

8 [https://github.com/kripken/emscripten](https://github.com/kripken/emscripten)
Segmentation

In the example application shown in figure 8, the audio stream captured from the microphone is sent into an lfo graph and analysed in real-time. The loudness and detected onsets are displayed on screen using simple visual elements. To illustrate offline abilities of the library, the user can also record few seconds from the audio stream. The recorded audio buffer is sent into a similar lfo graph using the same onset detection for offline segmentation. The extracted markers are displayed on top of the waveform of the recorded buffer using the waves-ui library [6].

Figure 8: Screenshot of an example application that features real-time and offline analysis and segmentation of the audio signal captured by the microphone.

The application has been published at the following URL: https://cdn.rawgit.com/wavesjs/waves-lfo/master/examples/realtime-offline-segmentation/index.html

Networked Graph

The last example application is a proof of concept that highlights the ability to create lfo graphs composed of several subgraphs distributed across multiple platforms over the network. The application consists of two different web-clients. The first client, running on a mobile device, contains the source of the graph and produces frames extracted from DeviceOrientation events (i.e. measuring the orientation of the mobile device using inertial sensors and compass). The frames are propagated through the server to the other web-client to be graphically displayed on screen. The networked lfo graph is composed of three sub-graphs — one on each client and one on the server — that are connected through WebSocket sources and sinks.

The source code of the application is available at the following URL: https://github.com/wavesjs/waves-lfo/tree/master/examples/networked-graph

6. CONCLUSIONS

We have presented the lfo library highlighting the underlying concepts and giving some details of its implementation. The benchmarks show that the implemented modules perform very well compared to existing Javascript libraries and comparable C/C++ implementations.

We hope that the lfo API and formalism will be picked up by other contributors and will foster the development of an ecosystem of modules implementing a large range of algorithms as well as new platform bindings. The library easily allows for developing modules that are distributed independently of the core of the library.9

Even if the library has been designed with interactive audio processing and music information retrieval in mind, the lfo formalism does not prevent the use of existing modules in other applications nor the integration of processing algorithms from other domains.

7. ACKNOWLEDGMENTS

The presented work has been developed in the framework of the CoStMa10 research project supported by the French National Research Agency (ANR-13-CORD-0010) and has received support from the Rapid-Mix Project (H2020-ICT-2014-1, Project ID 644862). We would like to thank our project partners and colleagues at IRCAM for their precious contributions to the project. We would especially like to acknowledge Victor Saiz who has collaborated on the very first version of the library of which the basic concepts have been presented at the 1st Web Audio Conference.

8. REFERENCES


9 While currently the distribution of the library includes a set of operators, sources and sinks, these modules may be separated from the core and distributed independently in the future.

10 http://cosima.ircam.fr/