Faust audio DSP language in the Web

Stéphane Letz, Sarah Denoux, Yann Orlarey, Dominique Fober

Faust audio DSP language in the Web

Stephane LETZ and **Sarah DENOUX** and **Yann ORLAREY** and **Dominique FOBER**
GRAME
11, cours de Verdun (GENSOUL)
69002 LYON,
FRANCE,
{letz, sdenoux, orlarey, fober}@grame.fr

Abstract

With the advent of both HTML5 and the Web Audio API (a high-level JavaScript API for audio processing and synthesis) interesting audio applications can now be developed for the Web. The Web Audio API offers a set of fast predefined audio nodes as well as customizable *ScriptProcessor* node, allowing developers to add their own javascript audio processing code.

Several projects are developing abstractions on top of the Web Audio API to extend its capabilities, and offer more complex unit generators, DSP effects libraries, or adapted syntax. This paper brings another approach based on the use of the FAUST audio DSP language to develop additional nodes to be used as basic audio DSP blocks in the Web Audio graph.

Different methods have been explored: going from an experimental version that embeds the complete FAUST native compilation chain (based on *libfaust + LLVM*) in the browser, to more portable solutions using JavaScript or the much more efficient *asm.js* version. Embedding the FAUST compiler itself as a pure JavaScript library (produced using *Emscripten*) will also be described.

The advantages and issues of each approach will be discussed and some benchmarks will be given.

Keywords

Web Audio API, FAUST, Domain Specific Language, DSP, real-time

1 Introduction

This paper demonstrates how an efficient compilation chain from FAUST to the Web Audio API can be done, allowing the available FAUST programs and libraries to be immediately used in a browser.

Section 2 describes the Web Audio API and how it can be extended and targeted by Domain Specific Languages. Section 3 describes the FAUST language and its mechanisms to be deployed on a large variety of platforms. Section 4 exposes the compilation chain and the multiple target languages available from a unique DSP specification. In the context of the Web Audio API, section 5 presents the different approaches experimented to deploy Faust DSP programs on the Web. Section 6 exposes some use cases, and finally some results and benchmarks are given in section 6.1.

2 Programming audio in the Web

2.1 Web Audio API

The Web Audio API [12] specification describes a high-level JavaScript API for processing and synthesizing audio in Web applications. The conception model is based on an audio routing graph, where a number of AudioNode objects are connected together to program the global audio computation.

The actual processing is executed in the underlying implementation [1] for native nodes, but direct JavaScript processing and synthesis is also supported using the *ScriptProcessorNode*.

2.2 Native nodes

The initial idea of the specification is to give developers a set of highly optimized *native* nodes, implementing the commonly needed functions: playing buffers, filtering, panning, convolution etc. The nodes are connected to create an audio graph, to be processed by the underlying audio real-time rendering layer.

2.3 JavaScript ScriptProcessorNode

The ScriptProcessorNode interface allows the generation, processing, or analyzing of audio using JavaScript. It is an AudioNode audio-processing module that is linked to two buffers, one containing the input audio data, one containing the processed output audio data.

An event, implementing the AudioProcessingEvent interface, is sent to the object each time the input buffer contains new data, and the event handler terminates when it has filled the output buffer with data.

[1] typically optimized assembly or C/C++ code

This is the *hook* given to developers to add new low level DSP processing capabilities to the system.

2.4 Programming over the Web Audio API

Various JavaScript DSP libraries or musical languages, have been developed over the years ([4], [6], [8], [10]) to extend, abstract and empower the capabilities of the official API. They offer users a richer set of audio DSP algorithms and sound models to be directly used in JavaScript code.

When following this path, developments have to be restarted from scratch, or by adapting already written code (often in more real-time friendly languages like C/C++) into JavaScript.

An interesting alternative has recently been developed by the Csound team [11]: by using the C/C++ to JavaScript Emscripten [3] compiler, the complete C written Csound runtime and DSP language (so including a large number of sound opcodes and DSP algorithms) is now available in the context of the Web Audio API. Using an automatic C/C++ to JavaScript compilation chain opens interesting possibilities to ease the deployment of well-known and mature code base on the Web.

3 FAUST language description

FAUST [Functional Audio Stream] [1] [2] is a functional, synchronous, domain-specific programming language specifically designed for real-time signal processing and synthesis. A unique feature of FAUST, compared to other existing music languages like Max[2], PureData, Supercollider, etc., is that programs are not interpreted, but fully compiled. FAUST provides a high-level alternative to hand-written C/C++ to implement efficient sample-level DSP algorithms.

One can think of FAUST as a *specification language*. It aims at providing the user with an adequate notation to describe *signal processors* from a mathematical point of view. This specification is free, as much as possible, from implementation details. It is the role of the FAUST compiler to automatically provide the best possible implementation. The compiler translates FAUST programs into equivalent C++ programs[3] taking care of generating the most efficient code. The compiler also offers various options to control the generated code, including options to do fully automatic parallelization and to take advantage of multicore architectures.

From a syntactic point of view FAUST is a textual language, but nevertheless block-diagram oriented. It actually combines two approaches: *functional programming* and *algebraic block-diagrams*. The key idea is to view block-diagram construction as function composition. For that purpose, FAUST relies on a *block-diagram algebra* of five composition operations (: , ~ <: :>) [1].

Here is an example of how to write a noise generator in FAUST:

```
random  = +(12345)~*(1103515245);
noise   = random/2147483647.0;
process = noise
        * vslider("Volume",0,0,1,0.1);
```

3.1 Language deployment

Being a specification language, the FAUST code tells nothing about the audio drivers or the GUI toolkit to be used. It is the role of the *architecture file* to describe how to relate the DSP code to the external world. Additional generic code is added to connect the DSP computation itself to audio inputs/outputs, and to control parameters, which could be buttons, sliders, num entries etc. in a standard user interface, or any kind of control using a remote protocol like OSC or HTTP.

This approach allows a single FAUST program to be easily deployed to a large variety of audio standards (Max-MSP externals, PD externals, VST plugins, CoreAudio or JACK standalone applications, etc.).

4 FAUST compilation chain

4.1 Static compilation chain

The current version of the FAUST compiler (*faust1*) produces DSP code as a C++ class, to be inserted in an architecture file. The resulting file is finally compiled with a regular C++ compiler to obtain an executable program or plug-in (Figure 1).

The produced application is structured as shown in Figure 2. The DSP becomes an audio computation module linked to the user interface and the audio driver.

[2]the *gen* object added in Max6 now creates compiled code from a patch-like representation, using the same LLVM based technology

[3]In faust1, faust2 branch allows to compile for different languages

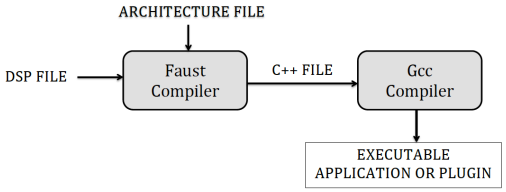

Figure 1: Steps of FAUST compilation chain

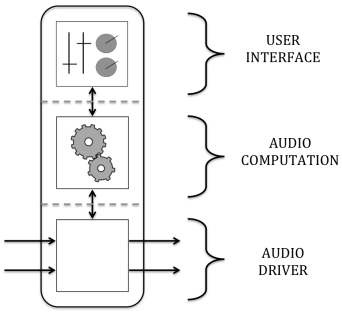

Figure 2: FAUST application structure

4.2 Multiple backends

Faust2 development branch uses an intermediate FIR representation (FAUST Imperative Representation), which can be translated to several output languages.

The FIR language describes the computation performed on the samples in a generic manner. It contains primitives to read and write variables and arrays, do arithmetic operations, and define the necessary control structures (*for* and *while* loops, *if* structure etc.). The *language of signals* (internal to the FAUST compiler) is now compiled in FIR intermediate language.

To generate various output languages, several backends have been developed: for C, C++, Java, JavaScript, asm.js, and LLVM IR (Figure 3). The native LLVM based compilation chain is particularly interesting: it provides direct compilation of a DSP source into executable code in memory, bypassing the external compiler requirement.

4.3 LLVM

LLVM (formerly Low Level Virtual Machine) is a compiler infrastructure, designed for compile-time, link-time, run-time optimization of programs written in arbitrary programming languages. Executable code is produced dynamically using a "Just In Time" compiler from a specific code representation, called LLVM IR. Clang, the "LLVM native" C/C++/Objective-C compiler is a front-end for LLVM Compiler.

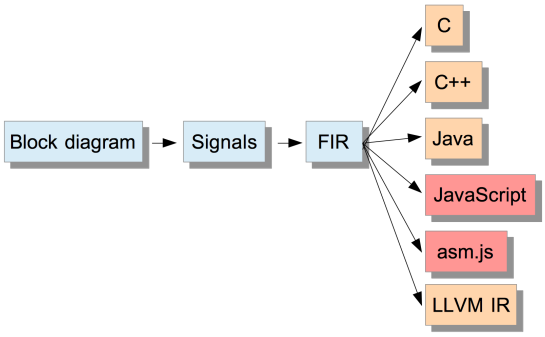

Figure 3: Faust2 compilation chain

It can, for instance, convert a C or C++ source file into LLVM IR code.

Domain-specific languages like FAUST can easily target the LLVM IR. This has been done by developing a special LLVM IR backend in the FAUST compiler.

4.4 Dynamic compilation chain

The complete chain goes from the DSP source code, compiled in LLVM IR using the LLVM back-end, to finally produce the executable code using the LLVM JIT [5]. All steps take place in memory, getting rid of the classical file based approaches. Pointers to executable functions can be retrieved from the resulting LLVM module and the code directly called with the appropriate parameters (Figure 4).

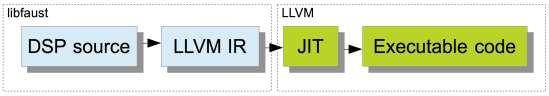

Figure 4: libfaust + LLVM dynamic compilation chain

4.5 FAUST compiler as a library

In the faust2 development branch, the FAUST compiler has been packaged as a library called *libfaust*, published with an associated API [5] that imitates the concept of oriented-object languages, like C++:

- given a FAUST source code (as a file or a string), calling the *createDSPFactory* function runs the compilation chain (FAUST + LLVM JIT) and generates the "prototype" of the class, as a *llvm-dsp-factory* pointer.
- next, the *createDSPInstance* function, corresponding to the *new className* of C++,

instantiates a *llvm-dsp* pointer, to be activated and controlled through its interface, and connected to the audio drivers.

Having the compiler available as a library opens new interesting possibilities explored in the FaustLive [9] application. DSP source code can be compiled on the fly and will run at native speed.

5 Using FAUST compiler in the Web

We have tested and implemented two different methods to use the FAUST compilation chain in the Web:

- the first one consists in embedding the *libfaust + LLVM* native compilation chain directly in the browser. Starting from the FAUST DSP source, a native WebAudio node will be compiled on the fly, to be used like any regular native node. The set of all control parameters will be exposed as WebAudio AudioParams objects.
- an alternative and more portable method purely stays at JavaScript level, using asm.js and Emscripten. Starting from the FAUST DSP source, a highly optimized asm.js based ScripProcessor node will be produced. The set of all control parameters will be exposed to control the DSP node.

Both approaches have advantages and issues that will be explained in detail in the following sections.

5.1 Native FAUST DSP Web Audio node

Embedding the *libfaust + LLVM* compilation chain has been experimented by "hacking" the WebKit open-source browser and by plugging the FAUST compiler in its Web Audio subproject.

A new native C++ *FaustNode* (sub-class of base class AudioNode) has been added to the set of native Web Audio nodes[4]. This node takes the DSP source code as a string parameter, compiles it on the fly to native executable code, and activates it:

```
var dsp
  = context.createFaustNode(code);
```

[4]This work was done in summer 2012 with the generous help of Chris Rogers, working at Google at that time.

As a native node, it can be used like any other regular native node and connected to other nodes in the graph:

```
dsp.connect(context.destination);
```

The FAUST source code usually contains an abstract description of its user interface, described in terms of buttons, sliders, bargraphs..., to be "interpreted" and displayed by an actual user interface builder component. This user interface can be obtained as a JSON description, that can be decoded to implement the UI themselves to control the node's parameters:

```
var json = dsp.json();
```

Internal control parameters of the DSP can be retrieved as a list of AudioParams, to be used like regular ones:

```
var num_params
  = dsp.numberOfAudioParams();
var audio_param
  = dsp.getAudioParam(0);
audio_param.setValue(0.5);
```

Instead of directly accessing the given parameter, another possibility is to use the following generic function, taking a complete access "path" to the parameter, and a given value:

```
dsp.setAudioParamValue("/wet",0.5);
```

Embedding the FAUST compiler in a browser is quite efficient, since the native executable code runs in the real-time audio thread that computes the audio graph rendering. But more general deployment and acceptance would require convincing the Web Audio community to embed a DSL language for audio processing in all browsers.

5.2 Compiling to JavaScript

More portable solutions have to use the ScriptProcessorNode node, directly producing JavaScript code to be executed in the node.

5.2.1 JavaScript backend

A pure JavaScript backend has been added to FAUST in 2012 to produce standard JavaScript code. The DSP class definition is then wrapped with a generic JavaScript file in order to get a fully working Web Audio ScriptProcessorNode.

5.2.2 Results

Two main problems have been discovered with this approach:

- for some of its computations, the FAUST compiler relies on pure 32 bits integral mathematical operations. Since JavaScript stores numbers as floating-point values according to the IEEE-754 Standard, this kind of computation can't produce the expected result. Thus, some DSP effects (like *noise* generation that uses a wrapping 32 bits integer division) do not work correctly.
- since standard JavaScript is not really suited to implement fast DSP code, the generated program is significantly slower compared to the native C/C++ or LLVM versions. The resulting audio nodes are usable only when the programmed DSP code is simple enough, but more demanding algorithms (like physical models) can usually not be used.

5.3 Compiling to asm.js JavaScript

Started in 2011 to facilitate the port of large C/C++ code base in JavaScript, Mozilla developers have started the *Emscripten* compiler project, based on LLVM technology, that generates JavaScript from C/C++ code.

Later on, they designed *asm.js*, a completely typed subset of JavaScript, statically compilable, garbage-collection free, that can be highly optimized by the compilation chain embedded in recent Web browsers. It is then possible to reach performances similar to pure native code[5]

Mainly designed to manipulate simple types like floating point or integer numbers, *asm.js* language is particularly of interest for audio code. Two successive developments have been carried out with this approach.

5.3.1 Using Emscripten compiler

Starting from the FAUST DSP generated C++ class, the Emscripten compiler translates it to JavaScript. Additional wrapping JavaScript code connects the Emscripten runtime memory manager and makes the generated code become a ScriptProcessorNode node to be used in the audio graph (Figure 5). This method has been successfully developed and demonstrated by Myles Boris [7].

Although this approach performs rather well, it requires the Emscripten tool chain to be installed on the user machine. A more integrated system has been later on developed.

[5]In the best cases, asm.js code is said to be only 2 or 3 times slower than pure native code, see `http://kripken.github.io/mloc_emscripten_talk`

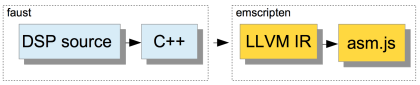

Figure 5: FAUST to asm.js (using Emscripten) static compilation chain

5.3.2 Developing a direct asm.js backend

A pure asm.js backend has been added to the faust2 branch, bypassing the Emscripten compilation chain (Figure 6).

The backend produces the asm.js module as well as some additional helper JavaScript functions, to be wrapped by generic JavaScript to become a completely usable Web Audio node. Heap memory code to be used with the asm.js module, and connection with compiled helper functions is managed by the wrapping code.

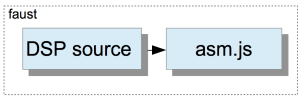

Figure 6: FAUST to asm.js (using FIR backend) static compilation chain

A new DSP instance is created using the following code, taking the Web Audio context and a given "buffer_size" as parameters:

```
var dsp
  = faust.karplus(context, buffer_size);
```

The user interface can be obtained as a JSON description, that can be decoded to implement the UI themselves to control the node's parameters:

```
var json = dsp.json();
```

The instance can be used with the following code:

```
dsp.start();
dsp.connect(context.destination);
dsp.setValue(path_to_control, val);
```

5.4 Embedding the JavaScript FAUST compiler in the browser

Thanks to the Emscripten compiler, the FAUST compiler itself can be compiled to asm.js JavaScript. This has been done by compiling the *libfaust* C++ library to the *libfaust.js* JavaScript library (Figure 7), that exports a unique entry point:

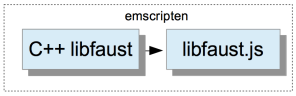

Figure 7: Compiling C++ libfaust to libfaust.js with Emscripten

- *createAsmCDSPFactoryFromString(...)* allows to create a DSP factory from a given DSP program as a source string and a set of compilations parameters, uses the asm.js backend, and produces the complete asm.js module and additional pure JavaScript methods as a string.
- then calling JavaScript "eval" function on this string *compiles* it in the browser. The dynamically created asm.js module and additional pure JavaScript methods (Figure 8) can then be used.

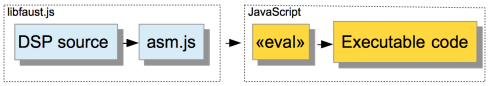

Figure 8: libfaust.js + asm.js dynamic compilation chain

This internal code in then wrapped with additional JavaScript code. A DSP "factory" will be created from the DSP source code with the following code:

```
var factory
  = faust.createDSPFactory(code);
```

A fully working DSP "instance" as a Web Audio node is then created with the code:

```
var dsp
  = faust.createDSPInstance(factory,
                            context,
                            buf_size);
```

The user interface can be retrieved as a JSON description:

```
var json = dsp.json();
```

The instance can be used with the following code:

```
dsp.start();
dsp.connect(context.destination);
dsp.setValue(path_to_control, val);
```

6 Use cases

Using the previously explained technologies, three different use cases have been experimented:

- compiling self-contained ready to use Web Audio nodes (see section 6.1)
- using FAUST static compilation chain to produce HTML pages with DSP code (see section 6.2)
- using the FAUST dynamic compilation chain to directly *program DSP* in the Web (see section 6.3).

6.1 Programming Web Audio nodes with FAUST

Self contained ready to use Web Audio nodes can be produced using the *faust2asmjs* script, using the static compilation chain explained in section 5.2. The script basically calls the FAUST compiler targeting the asm.js backend with the appropriate architecture file, that wraps the produced code with generic JavaScript to be usable in the Web Audio API context (Figure 9).

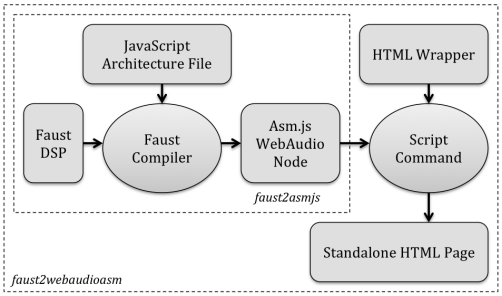

Figure 9: faust2amsjs and faust2webaudioasm compilation chains

6.2 Deploying FAUST DSP examples in the Web

Using the *faust2webaudioasm* script, a DSP source file can be compiled to a self-contained ready to run HTML page (Figure 10), using the static compilation chain (see section 5.2 and Figure 9).

The FAUST compiler targeting the asm.js backend with the appropriate architecture file is called. The asm.js + JavaScript WebAudio node is then wrapped in a more complex HTML code template, and the final HTML page is obtained. Adding the *-links* parameter to the script makes the HTML page also contains links

to the original DSP textual file, as well as the block-diagram SVG representation.

Thus it becomes quite simple to publish DSP algorithms, helping it wider usage of the FAUST DSL approach.

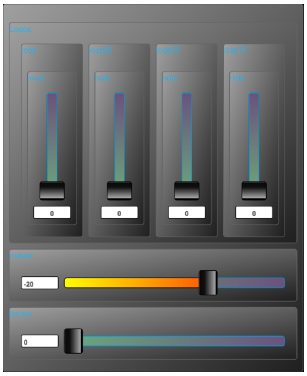

Figure 10: Example of SVG based user interface generated from the JSON description

6.3 Programming DSP in the Web

Having the FAUST compiler itself as a library in the browser opens interesting capabilities:

- "light" FAUST IDE allowing users to test the language can be easily developed on the Web, completing the more full featured FaustLive application [9].
- combining existing DSP sources published as HTML pages, to create new DSP programs to be directly tested and used in the Web, or possibly exported to any native platform supported by the FaustWeb external compilation service. This has been demonstrated by Sarah Denoux [13].

7 Tests and benchmarks

The three previously described approaches have been tested on a 4 cores MacBook Pro 2,3 GHz.

7.1 Benchmarks

The Web Audio API is still a fresh specification. Its implementation in different browsers on different platforms is not always complete or stable. Comparing the previously described approaches has been quite challenging, mainly because of slight differences of behavior or interaction with the underlying operating system.

The proposed benchmarks have been done by simply comparing the application CPU use with some heavy FAUST programs, using the "Activity Monitor" tool included in OSX. Three different DSP programs have been tested.

Since the various presented methods could not be developed in a same browser, we had to use two different ones. Native version is tested in the "hacked" WebKit application, JavaScript and asm.js using Firefox version 32.0.3.

Effect	native	JavaScript	asm.js
cubic_distortion	6.0 %	45 %	28 %
harpe	2.7 %	50 %	8 %
kisanaWD	4 %	over 100%	14 %

Table 1: Global CPU use of the application tested on a MacBook Pro 2,3 GHz

Even with this limited testing method, some interesting results emerge. The native chain (based on *libfaust + LLVM*) is clearly the fastest, the asm.js based one is usable in a lot of real world use cases. The JavaScript version performs poorly, and is even not usable because of CPU overuse in a lot of examples (like "kisanaWD" here).

7.2 Known issues and perspective

Although the previously described developments show some promising results, they are still several issues to be solved:

- code for pure JavaScript and asm.js generated nodes is executed in the main thread. So it may suffer from interferences with the UI computation or possibly garbage collection. Moreover latency is added since an additional buffer is used in the audio chain. Thus real-time guaranties may not be met typically resulting in audio glitches [6].
- a specific problem has been discovered when audio computation produces "denormal" float values: on Intel processors, CPU performances degrade a lot [7].
- on the contrary, the "native" version is much more stable, has less latency since the computation is done in the real-time thread with no added buffer, but is much more difficult to deploy and maintain [8].

[6] A possible solution to this problem by moving the ScriptProcessorNode code in audio worker threads has been recently discussed in the W3C Audio working list, see `http://webaudio.github.io/web-audio-api`

[7] The problem has been reported and should be solved at the JavaScript language definition level.

[8] A port in Firefox is in progress.

8 Conclusion

The FAUST audio DSP language can now be used to easily develop new audio nodes in the Web Audio model, and use them in an audio graph. Complete HTML pages with a working user interface can also be generated. Having the dynamic compilation chain (either in native or pure JavaScript mode) directly available in the browser is also interesting to further explore.

Even if the Web Audio approach starts to mature, there are still some problematic issues, for instance float samples denormalization problem, or non real-time guaranties while rendering the ScriptProcessorNode JavaScript code.

The recent discussion on the *Audio Workers* model opens perspectives for a better rendering scheme. Basically the JavaScipt audio code will be moved to the real-time audio thread, and communications to get/set parameter values will be done from/to the main thread.

It remains to be tested how the compilation of DSP to Web Audio nodes from a high-level DSL language like FAUST or Csound will benefit from it.

Acknowledgments

This work has been done under the FEEVER project [ANR-13-BS02-0008] supported by the "Agence Nationale pour la Recherche".

References

[1] Y. Orlarey, D. Fober, and S. Letz, "Syntactical and semantical aspects of Faust", *Soft Computing*, 8(9), 2004, pp. 623–632.

[2] S. Letz, Y. Orlarey and D. Fober, "Work Stealing Scheduler for Automatic Parallelization in Faust", *Linux Audio Conference*, 2010.

[3] A. Zakai, "Emscripten: an LLVM to JavaScript compiler", *In Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications, pages 301–312. ACM* , 2011.

[4] H. Choi, J.Berger, "Waax: Web Audio API extension", *In Proceedings of the Thirteenth New Interfaces for Musical Expression Conference.*, 2013.

[5] S. Letz, Y. Orlarey and D. Fober, "Comment embarquer le compilateur Faust dans vos applications ?", *Journees d'Informatique Musicale*, 2013.

[6] C. Roberts, G. Wakefield, and M. Wright, "The Web Browser as Synthesizer and Interface". *New Interfaces for Musical Expression conference (NIME)*, 2013.

[7] M. Borins, "From Faust to Web Audio: Compiling Faust to JavaScript using Emscripten", *Linux Audio Conference*, 2014.

[8] C. Clark, A. Tindale, "Flocking: a framework for declarative music-making on the Web", *International Computer Music Conference*, 2014.

[9] S. Denoux, S. Letz, Y. Orlarey and D. Fober, "FAUSTLIVE Just-In-Time Faust Compiler... and much more", *Linux Audio Conference*, 2014.

[10] J. Kalliokoski, "audiolib.js, a powerful toolkit for audio written in JS", `https://github.com/jussi-kalliokoski/audiolib.js/`

[11] V. Lazzarini, E. Costello, S. Yi and J. Fitch, "Csound on the Web", *Linux Audio Conference*, 2014.

[12] WebAudioAPI reference description, `http://webaudio.github.io/web-audio-api/`

[13] S. Denoux, Y. Orlarey, S. Letz, and D. Fober, "Compose with Faust in the Web", *Web Audio Conference*, IRCAM & Mozilla Paris, France 2015.