What's new in the faust ecosystem and community?

Stéphane Letz, Romain Michon, Yann Orlarey

WHAT'S NEW IN THE FAUST ECOSYSTEM AND COMMUNITY?

Stéphane Letz
Univ Lyon, GRAME-CNCM, INSA Lyon, Inria, CITI, EA3720, 69621 Villeurbanne, France
letz@grame.fr

Romain Michon
Univ Lyon, Inria, INSA Lyon, CITI, EA3720, 69621 Villeurbanne, France
romain.michon@inria.fr

Yann Orlarey
Univ Lyon, GRAME-CNCM, INSA Lyon, Inria, CITI, EA3720, 69621 Villeurbanne, France
orlarey@grame.fr

ABSTRACT

This paper presents an overview all the new developments and contributions in the FAUST programming language since the 2020 International FAUST Conference. It shows a growing and dynamic community with artistic projects, plugins, standalone applications, integration in audio programming environments, development tools, research projects, embedded devices, Web applications, etc., produced by a large variety of contributors.

1. INTRODUCTION

The FAUST [1] community is growing steadily. This paper summarizes all the activity that have taken place since IFC 2020. Section 2 details new developments done in the compiler, architectures files, and various complementary tools. Section 3 explains efforts conducted to improve the documentation. Section 4 lists the various places and programs where FAUST is taught and part of the curriculum. Section 5 details new recently added libraries. Section 6 present the FAST (Fast Audio Signal-processing Technologies on FPGA) project. And finally, section 7 is about the tools that have been implemented to strengthen the FAUST community itself.

2. DEVELOPMENTS

2.1. New Backends

Three new backends have been developed. They allow us to use DSP programs in a larger set of targets and to reach new communities.

2.1.1. DLang + faust2dplug

D (Dlang) [1] is a general-purpose programming language with static typing, systems-level access, and C-like syntax. A backend and several architecture files have been contributed by Ethan Recker, with the help of GRAME.

The goal is to generate D code for Dplug, [2] an open-source audio plug-in framework existing since 2013 and allowing for the production of VT23, VST3, AUv2, AAX, and LV2 plugins for macOS, Windows, and Linux. A `faust2dplug` script has then been developed to simplify the production of Dplug projects starting from the FAUST DSP source code.

[1] `https://dlang.org`
[2] `https://dplug.org`

2.1.2. CSharp

C# is a general-purpose, multi-paradigm programming language used in some games engines. A backend and several architecture files have been contributed by Mike Oliphant, with the help of GRAME.

2.1.3. Julia

Julia [3] [2] is a general-purpose programming language for scientific computing that appeared in 2012. It uses an LLVM-based compiler, a dynamic type system with parameterized polymorphism and distributed parallel execution. It can be easily interfaced with existing languages like C, Fortran, or Python. For the programmer, it comes with a REPL (Real Eval Print Loop) model allowing for a simple and fast interaction with the system.

An integration of the libfaust compiler into Julia was first developed by Cora Johnson-Roberson. [4] A Julia backend was then added to the FAUST compiler. It allows us to generate Julia code on the fly from any FAUST DSP program to then compile it and run it directly in the Julia environment. A set of architecture files (`meta.jl`, `UI.jl`, `MapUI.jl`, `GTKUI.jl`, `OSCUI.j`) were implemented to:

- control the DSP with a graphical interface or an OSC interface
- interface it to an audio layer based on the PortAudio project (with the `audio.jl` and `portaudio.jl` files).

Test tools have also been developed:

- the `minimal.jl` file shows how the generated Julia code can be used in a minimal program that allocates and instantiates the DSP, and calls the compute function. The `MapUI.jl` file is used to eventually control the DSP. The command line [5] should be used to create a `foo.jl` file ready to be tested,
- the `faust2portaudiojulia` tool transforms a FAUST DSP program into a fully functional Julia source file that uses the PortAudio library for real-time audio rendering, and can be controlled with OSC messages. By default, it starts with the GTK-based GUI. It uses the `MapUI.jl`, `OSCUI.jl` and `GTKUI.jl` architecture files.

The Julia ecosystem contains over 4,000 packages that are registered in a general registry.[6] A `Faust.jl` package originally de-

[3] `https://julialang.org`
[4] `https://github.com/corajr/Faust.jl`
[5] `faust -lang julia -a julia/minimal.jl foo.dsp -o foo.jl`
[6] `https://juliahub.com`

veloped by Cora Johnson-Roberson has been extended.[7] It allows for the use of the libfaust library and the FAUST backend of Julia.

This groundwork will facilitate further developments in the area of machine learning applied to audio and DSP, where the Julia environment and language are increasingly used. The integration of libfaust in Julia allowed – for example – Cora Johnson-Roberson to do some experiments using neural networks [8]. We can imagine in the long run more advanced integration in this domain with the FAUST backend directly producing Julia code.

2.2. Modular Synthesis

Modular synthesizers are made of separate modules with different functions. Modules can be connected together using cables or a matrix connection system. The outputs (voltages) of the modules can function as (audio) signals, control voltages, or logic/time conditions. Typical modules are wave generators, effects, envelope generators, etc.

While modules were traditionally implemented using analog electronic circuits, digital modules are slowly becoming the norm. Such systems usually combine a processor (e.g., microcontroller, DSP chip, etc.) and an audio ADC/DAC. Their form factor and interface is standardized (following the Eurorack standard,[9] for example). They are then assembled in racks, and controlled by an external patching system.

Applications like VCV-Rack [10] allow a software emulation of this modular synthesis principle. The plugins are then developed in C++ with an SVG-based interface.

In this context, developments have been carried out to allow for the prototyping of plugins for the VCV-Rack format using the FAUST programming language.

2.2.1. The faust2vcvrack Tool

The `faust2vcvrack` tool compiles a FAUST DSP program into a folder containing (i) the C++ source code of the VCV-Rack plugin and (ii) a MakeFile to compile it. By default, the resulting C++ code is then compiled and installed in the VCV-Rack application.

The FAUST DSP code classically produces audio signals in the range [-1..1]. Since the VCVs expect audio signals in the range [-5v..5v], they are automatically converted in the architecture file. CV commands in the range [0v..10v] are assigned to the [min..max] range of the controllers.

Polyphonic modules can be created using the `-nvoices <n>` parameter up to 16 voices. The freq/gate/gain convention[11] can be used in the DSP code. VCV Rack follows the 1V/octave convention for MIDI pitch values, so MIDI signals are automatically converted to freq using this convention. Gain and gate signals (using the [0v..10v] range) are converted to [0..1] values.

Controllers (typically buttons, sliders, or bar graphs) are automatically transformed into GUI elements (like switches, buttons, or leds). But they can also be connected to CV inputs/outputs by using a `[CV:N]` metadata used in the input (typically sliders or nentry) or output (typically bargraphs) controllers to connect them to the CV signal instead of the GUI parameters.

[7] `https://github.com/sletz/Faust.jl`

[8] `https://github.com/corajr/faust_nn`

[9] `https://en.wikipedia.org/wiki/Eurorack`

[10] `https://vcvrack.com`

[11] `https://faustdoc.grame.fr/manual/midi/#midi-polyphony-support`

2.2.2. VCV Prototype

The VCV Prototype Module runs scripting languages for prototyping, learning, and live coding. It can currently be programmed using JavaScript, Lua, Vult, or PureData. A generic GUI with 6 inputs/outputs (either audio or CV signals), 6 knobs, 6 lights (RGB LEDs) or 6 switches (with RGB LEDs) is defined.

FAUST support has been added thanks to libfaust embedding the Interpreter backend. It allows us to edit/compile/execute DSP programs on the fly, with acceptable performances (even if using the LLVM JIT would allow us to generate faster code, but at the expense of a much more complicated installation procedure).

2.3. Embedded Platforms

2.3.1. The Daisy Board

Daisy[12] is an embedded platform for music developed by Electrosmith. It combines on a single board a low-consumption ARM embedded processor, SDRAM memory, as well as a stereo audio codec. Several boards hosting the Daisy Seed[13] have been developed on top of the same chip.

They are distributed with a software library which abstracts low-level embedded software development for the boards and provide support for a number of languages including C++, Arduino, and Max/MSP Gen~. Specific developments have been carried out to program this new device with FAUST.

2.3.2. The faust2daisy Tool

The `faust2daisy` tool compiles a Faust DSP program into a folder containing the C++ source code and a Makefile to compile it. Options to compile polyphonic DSP and add MIDI control are available. Specific architecture files have been written:

- `faust/gui/DaisyControlUI.h`: to be used with the DSP `buildUserInterface` method to implement button, checkbox, hslider, vslider controllers, and interpret the specific metadata used to describe the hardware,
- `faust/midi/daisymidi.h`: implements a `midi_handler` subclass to decode incoming MIDI events.

The current `faust2daisy` tool can only be used to program the POD[14]. On this board, the 2 switches and 2 knobs can be connected to UI controllers using metadata.

The programming model still needs to be extended to support more devices and better support their heterogenous memory model.

2.4. Exported Box and Signal API

The FAUST compiler can be used in applications or plugins using the libfaust library, embedding the LLVM backend and allowing DSP code to be dynamically compiled and executed. A set of C and C++ headers are available to access the API.

Work has been done to give access to other intermediate points in the compilation chain, so that new use-cases can be considered, like developing graphical language interfaces or connecting with machine learning models.

[12] `https://www.electro-smith.com/daisy`

[13] `https://www.electro-smith.com/daisy/daisy`

[14] `https://www.electro-smith.com/daisy/pod`

The compilation chain of the FAUST compiler is composed of several steps (see Figure 1):

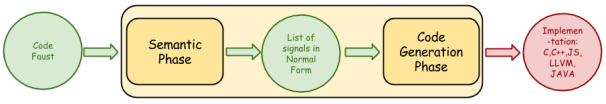

Figure 1: *The compilation chain.*

Starting from the DSP source code, the Semantic Phase produces signals as conceptually infinite streams of samples or control values. Those signals are then compiled in imperative code (C/C++, LLVM IR, WebAssembly, etc.) in the Code Generation Phase.

The Semantic Phase itself is composed of several steps (see Figure 2):

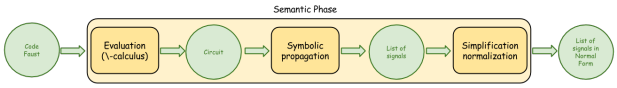

Figure 2: *The semantic phase.*

The initial DSP code using the Block Diagram Algebra (BDA) is translated in a flat circuit in normal form in the Evaluation, lambda-calculus step. The list of output signals is produced by the Symbolic Propagation step. Each output signal is then simplified and a set of optimizations are done (normal form computation and simplification, delay line sharing, typing, etc.) to finally produce a list of output signals in normal form.

The Code Generation Phase translates the signals in an intermediate representation named FIR (FAUST Imperative Representation) which is then converted to the final target language (C/C++, LLVM IR, WebAssembly, etc.) with a set of backends.

2.4.1. Accessing the Box Stage

A new intermediate public entry point has been created in the Semantic Phase, after the Evaluation, lambda-calculus step to allow the creation of a box expression, then beneficiate of all remaining parts of the compilation chain.

It means that a box expression can be *programmatically built* (using the box C++ API, or the C box API version), evalutated to the list of output signals, translated to the FIR format, and finally converted to the final target language, as a ready-to-use C++ class, LLVM or Interpreter factories, to be used with all existing architecture files.

2.4.2. Compiling Box Expressions

To use the box API, the following steps must be taken:

- creating a global compilation context using the `createLibContext` function,
- creating a box expression using the box API, progressively building more complex expressions by combining simpler ones,
- compiling the box expression using the `createCPPDSPFactoryFromBoxes` function to create a DSP factory (or some variants),
- finally destroying the compilation context using the `destroyLibContext` function.

DSP factories allow for the creation of DSP instances, to be used with audio and UI architecture files, outside of the compilation process itself. The DSP instances and factory have to be eventually deallocated when not used anymore.

Here is an example, creating and compiling the DSP code:

```
process = + ~ _;
```

With the box API code:

```
// Create global context
createLibContext();

// Create the box expression
Box box = boxRec(boxAdd(), boxWire());

// Compile it as a factory
string error_msg;
llvm_dsp_factory* factory =
    createDSPFactoryFromBoxes("FaustDSP",
    box, 0, nullptr, "", error_msg);

// Create a DSP instance
dsp* dsp = factory->createDSPInstance();

// Use dsp with all existing architecture
    files
...
// Delete dsp and factory
delete dsp;
deleteDSPFactory(factory);

// Destroy the global context
destroyLibContext();
```

2.4.3. Accessing the Signal Stage

A new intermediate public entry point has been created in the Semantic Phase allowing for the creation of a signal graph (as a list of output signals) then benefiting from all remaining parts of the compilation chain.

The signal C++ API (or the C signal API version) allows us to *programmatically build* the signal graph, then compile it to create a ready-to-use DSP as a C++ class (or LLVM Interpreter factories) to be used with all existing architecture files.

2.4.4. Compiling Signal Expressions

To use the signal API, the following steps must be taken:

- creating a global compilation context using the `createLibContext` function,
- creating signals outputs using the signal API, progressively building more complex expressions by combining simpler ones,

- compiling the list of outputs using the `createCPPDSPFactoryFromSignals` function to create a DSP factory (or some variants),
- finally destroying the compilation context using the `destroyLibContext` function.

The DSP factories allow for the creation of DSP instances, to be used with audio and UI architecture files, outside of the compilation process itself. The DSP instances and factory will finally have to be deallocated once used no more.

Here is an example, using the previous recursive DSP code and the signal API:

```
// Create global context
createLibContext();

// Create the signal expression
tvec signals;
Signal in1 = sigInput(0);
signals.push_back(sigRecursion(sigAdd(
    sigSelf(), in1)));

// Compile it as a factory
string error_msg;
llvm_dsp_factory* factory =
    createDSPFactoryFromSignals("FaustDSP",
    signals, 0, nullptr, "", error_msg);

// Create a DSP instance
dsp* dsp = factory->createDSPInstance();

// Use dsp with all existing architecture
    files
...
// Delete dsp and factory
delete dsp;
deleteDSPFactory(factory);

// Destroy the global context
destroyLibContext();
```

2.4.5. Creating a Language Based on Those APIs

Generating complex expressions by directly using the box or signal APIs can quickly become tricky and impracticable. So a language created on top of them is usually needed. This is exactly what the Block Diagram Algebra is all about, and the entire FAUST language itself.

But giving access to the box and signal APIs allows for completely *new audio languages* to be created, while taking advantage of the compiler infrastructure and existing architectures.

The Elementary audio language, [15] for instance, is built over signal language resembling the one previously described, and uses JavaScript as the upper layer language to help create complex signal graphs programmatically. A similar approach could be proposed with the FAUST signal API. Other approaches using graphical based tools could certainly be tested.

[15] `https://www.elementary.audio`

2.5. Debugging and Optimisation Tools

When FAUST DSP programs are used in demanding projects, the programmer may have to carefully check the behaviour of the generated code, and possibly optimize it as much as possible. Several tools have been developed to help with this.

2.5.1. Debugging the Code

The FIR (FAUST Imperative Representation) backend can possibly be used to look at a textual version of the intermediate imperative language. It displays various statistics, like the number of operations done in the generated `compute` method, or the DSP memory layout.

2.5.2. Using -ct and -cat Options

Using the `-ct` and `-cat` compilation options allows us to check table index range, by verifying that the actual signal range is compatible with the actual table size. Note that since the interval calculation is imperfect, you may see false positives especially when using recursive signals where the interval calculation system will typically produce [-inf, inf] range, which is not precise enough to correctly describe the real signal range.

2.5.3. Using -me Option

Starting with FAUST version 2.37.0, mathematical functions which have a finite domain (like `sqrt` defined for positive or null values, or `asin` defined for values in the [-1..1] range) are checked at compile time when they actually compute values at that time, and raise an error if the program tries to compute an out-of-domain value.

If those functions appear in the generated code, their domain of use can also be checked (using the interval computation system) and the `-me` option will display warnings if the domain of use is incorrect. Note that again, because of the imperfect interval computation system, false positives may appear and should be checked.

2.5.4. Optimizing the Code

Developments have been done to ease the deployment of C++ (or LLVM IR) generated code in real demanding environments.

2.5.5. Compiling for Multiple CPUs

On modern CPUs, compiling native code dedicated to the target processor is critical to obtain the best possible performances. When using the C++ backend, the same C++ file can be compiled with gcc of clang for each possible target CPU using the appropriate `-march=cpu` option.

When using the LLVM backend, the same LLVM IR code can be compiled into CPU specific machine code using the `dynamic-faust` tool. This step will typically be done using the best compilation options automatically found with the `faustbench` tool or `faustbench-llvm` tools. A specialized tool has been developed to combine all the possible options.

2.5.6. The faust2object Tool

The `faust2object` tool [16] either uses the standard C++ compiler or the LLVM dynamic compilation chain (the `dynamic-faust` tool) to compile a FAUST DSP to object code files (.o) and wrapper C++ header files for different CPUs.

The DSP name is used in the generated C++ and object code files, thus allowing to generate distinct versions of the code that can finally be linked together in a single binary.

3. TECHNICAL DOCUMENTATION

3.1. Documentation of the Architecture Files

The FAUST compiler produces the DSP processing code in the form of a module (e.g., a C++ class) that must then be connected to the outside world. The program will be integrated into the audio architecture of the target machine (i.e., a computer, a smartphone, or a web page) and used with a control interface – typically a graphical interface (possibly deported to another machine) – or in the form of gesture controls, if a smartphone with accelerometers or a gyroscope is used for example.

This connection to the outside world is made through what is called an architecture file. Initially developed for our own needs, the architecture files will have to be used by external developers who want to use FAUST in their projects.

An exhaustive documentation about the architecture files and their use has been written to facilitate their work.[17] It covers the classical requirements for deployment on computers, smartphones and tablets, web pages, as well as embedded hardware. The deployment of generators or monophonic effects, but also polyphonic instruments controllable by MIDI is described. The use of existing architecture files is explained, as well as the development of new custom files for specific needs.

3.2. Additional Resources

Additional pages/resources have been added progressively to describe:

- how to integrate the dynamic compiler (in the form of the libfaust library) into programs,[18]
- how to deploy Faust programs on the Web,[19]
- a Frequently Asked Questions (FAQ) page.[20]

4. LEARNING FAUST

FAUST is now taught at several places/institutions in the world.

4.1. Center for Computer Research in Music and Acoustics (CCRMA)

Several courses or tutorials around FAUST are given at CCRMA [21]:

- Julius Smith's FAUST tutorial,[22]
- Romain Michon's FAUST tutorials,[23]
- Music 250a (Physical Interaction Design for Music) course which hosts various tutorials on FAUST and hardware,[24]
- Music 320c (Audio Plugin Development in FAUST and C++),[25]
- Embedded DSP With FAUST Workshop. [26]

4.2. TU Berlin

The regular sound synthesis class at the Audio Communication Group, TU Berlin, makes use of FAUST for exploring the basics of different synthesis algorithms. Student projects based on FAUST include Eurorack modules, standalone drum machines and synthesizers, as well as data sonification approaches. The class is taught by Henrik von Coler, who is director of the Electronic Studio at the TU. HPI Potsdam

The class Data Sonification & Opportunities of Sound at Hasso Plattner Institute, University of Potsdam Potsdam, is an interdisciplinary format, exploring the use of sonification and sound synthesis in the context of design thinking, neurosience, and medical applications. The signal processing part is taught by Henrik von Coler.

4.3. Université Paris 8

A 24 hours introduction to FAUST is given by Alain Bonardi during the first semester to undergraduate students (L3, 3rd year after the french 'baccalauréat') as part of the "Programming Languages in Computer Music 1" course offered in the "Music creation with computers" minor.

4.4. Universidad Nacional de Quilmes

Faust / DSP courses in Spanish, prepared by Juan Ramos. They include the classes of the "Update Seminar on Sound, Science and Technology II," held at the National University of Quilmes (Argentina) with an intro video [27] and several classes. [28]

4.5. RIM & RAN Professional Masters Program

The RIM & RAN professional Masters programs aim at shaping young professionals in the fields of electronic and digital technologies applied to the arts in the prospect of becoming "Producer in Computer Music" (RIM - Réalisateur en Informatique Musicale) and in Digital Arts (RAN - Réalisateur en Arts Numériques).

These producers play an important role in musical and artistic productions, and work at the interface between software developers, applied computer scientists, composers, artists, etc. and all people likely to integrate video, image, and sound in their activities.

[16] `https://github.com/grame-cncm/faust/tree/master-dev/tools/benchmark#faust2object`
[17] `https://faustdoc.grame.fr/manual/architectures/`
[18] `https://faustdoc.grame.fr/manual/embedding/`
[19] `https://faustdoc.grame.fr/manual/deploying/`
[20] `https://faustdoc.grame.fr/manual/faq/`
[21] `https://ccrma.stanford.edu`
[22] `https://ccrma.stanford.edu/~jos/aspf/`
[23] `https://ccrma.stanford.edu/~rmichon/faustTutorials/`
[24] `https://ccrma.stanford.edu/courses/250a-winter-2022/`
[25] `https://ccrma.stanford.edu/courses/320c/`
[26] `https://ccrma.stanford.edu/workshops/faust-embedded-19/`
[27] `https://www.youtube.com/watch?v=DnBI7r273BE`
[28] `https://www.youtube.com/channel/UCD6aeS3GdkEmt86KUehr8LQ/videos`

Most of the courses about signal processing are given around through the FAUST language (M1 Romain Michon 12h / M2 Yann Orlarey 20h).

4.6. Aalborg University in Copenhagen

FAUST is taught by Romain Michon at Aalborg University in Copenhagen (Denmark) as part two one week workshops opened to Masters students of the Sound and Music Computing (SMC) masters program of the medialogy department. The first workshop typically happens in the Fall and focuses on the use of FAUST for programming embedded systems for real-time audio applications. The second workshop takes place in the Spring and is about physical modeling of musical instruments in FAUST.

5. LIBRARIES

5.1. Standard Libraries

Among the significant contributions received in the two last years on the Faust Libraries project, [29] we can mention:

- the `fds.lib` library developed by Riccardo Russo allowing the modelling of physical models by finite difference,
- the `wdmodels.lib` library developed by Dirk Roosenburg allowing WDF (Wave Digital Filter) modelling,
- the `aanl.lib` developed by Dario Sanfilippo. This library provides aliasing-suppressed nonlinearities through first-order and second-order approximations of continuous-time signals, functions, and convolution based on antiderivatives. This technique is particularly effective if combined with low-factor oversampling, for example, operating at 96 kHz or 192 kHz sample-rate,
- the `webaudio.lib` contributed by GRAME, implementing WebAudio filters using their C++ version as a starting point, taken from Mozilla Firefox implementation. This work was done to simplify porting audio effects written using the Web Audio API.

5.2. Community Contributions

Several contribution from the community have appeared in the last two years.

5.2.1. abclib library

The abclib library [30] is released by the CICM / MUSIDANSE (Centre de Recherches Informatique et Création Musicale, Paris 8 University) and is the result of 20 years of research, teaching, and creation in mixed music, expressed as a set of codes in the FAUST language.

The main topics addressed are: spatial sound processing and synthesis using ambisonics, multi-channel sound processing, utility objects for mixed music.

[29] `https://faustlibraries.grame.fr`
[30] `https://github.com/alainbonardi/abclib`

5.2.2. Edge of Chaos

This repository [31] contains libraries including some essential building blocks for the implementation of musical complex adaptive systems in FAUST programming.

It includes a set of time-domain algorithms, some of which are original, for the processing of low-level and high-level information as well as the processing of sound using standard and non-conventional techniques.

It also includes functions for the implementation of networks with different topologies, linear, and nonlinear mapping strategies to render positive and negative feedback relationships, and different kinds of energy-preserving techniques for the stability of self-oscillating systems.

5.2.3. realfaust

The realfaust library [32] contains a set of functions representing domain-limited versions of all FAUST primitives and math functions that can potentially generate INF or NaN values.

The goal of the library is to be able to implement DSP networks that, structurally, are free from INF and NaN values. Hence, the resulting programs should be rock-solid during real-time performance and virtually immune to crashes regardless of how mercilessly a network is modulated or how unstable a recursive system is made.

5.2.4. bitDSP-faust

BitDSP [33] is a set of FAUST library functions aimed to help explore and research artistic possibilities of bit-based algorithms.

BitDSP currently includes implementations of bit-based functions ranging from simple bit operations over classic delta-sigma modulations to more experimental approaches like cellular automata, recursive Boolean networks, and linear feedback shift registers.

A detailed overview of the functionality is in the paper "Creative use of bit-stream DSP in FAUST" presented at IFC 2020.

5.2.5. SEAM library

Sustained Electro-Acoustic Music is a project inspired by Alvise Vidolin and Nicola Bernardini. The SEAM libraries [34] have been developed for this project.

6. FAST: A FUNDED RESEARCH PROJECT AROUND FAUST

FAST[35] (Fast Audio Signal-processing Technologies on FPGA) is a research project funded by the Agence Nationale de la Recherche (ANR – the French National Research Agency). It gathers the strengths of GRAME-CNCM,[36] CITI Lab (INSA Lyon),[37] and LMFA (École Centrale Lyon)[38] towards two goals:

[31] `https://github.com/dariosanfilippo/edgeofchaos`
[32] `https://github.com/dariosanfilippo/realfaust`
[33] `https://github.com/rottingsounds/bitDSP-faust`
[34] `https://github.com/s-e-a-m/faust-libraries`
[35] `https://fast.grame.fr/`
[36] `https://www.grame.fr`
[37] `http://www.citi-lab.fr/`
[38] `http://lmfa.ec-lyon.fr/?lang=en`

- facilitate the design of ultra-low latency embedded systems for real-time audio signal processing,
- use such systems in the context of active control of acoustics.

FAUST plays a central role in this project by facilitating the programming of FPGAs for real-time audio signal processing applications. A "FAUST to FPGA" toolchain is in the process of being implemented. It already allows us[39] to run simple FAUST programs on Xilinx-based FPGA boards such as the Digilent Zybo Z7 (see Figure 3) and the Genesys 2, reaching unparalleled audio latency performances [3].

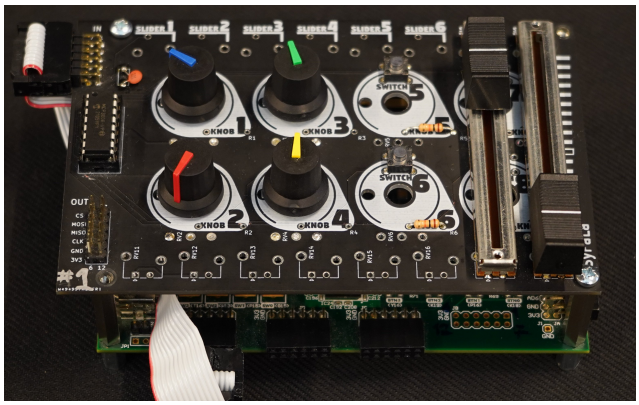

Figure 3: *Digilent Zybo Z7 board equipped with a custom-made control interface designed as part of FAST.*

7. THE FAUST COMMUNITY

7.1. Communication Channels

In addition to existing solutions such as the website, discussion lists and the compiler and applications GitHub development site, more modern communication tools have been put in place:

- a dedicated channel has been created on the Slack collaborative communication platform, organised in channels corresponding to many discussion topics. It currently gathers more than 300 developers (see Figure 4),
- the Audio Programmer website [40] animated by Joshua Hodge hosts a very large community of audio DSP developers. A FAUST channel has been created on this platform, which allows in particular to extend the visibility of the language and its ecosystem beyond the already interested or identified programmers (see Figure 5).

7.2. The "Powered by Faust" Page

A page listing all the significant "Powered with FAUST" projects is maintained: musical pieces or artistic projects, plugins, standalone applications, integration in audio programming environments, development tools, research projects, embedded devices, Web applications, etc are listed.

This page is regularly enriched and as of march 2022, more than 110 projects are described (see Figure 6).

[39] https://github.com/inria-emeraude/syfala

[40] https://theaudioprogrammer.com

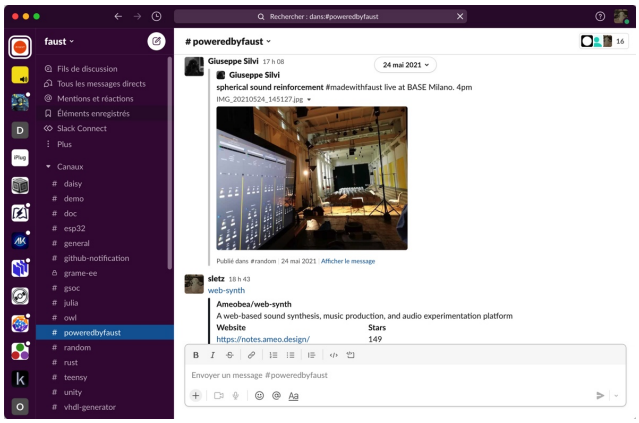

Figure 4: *Faust Slack channel.*

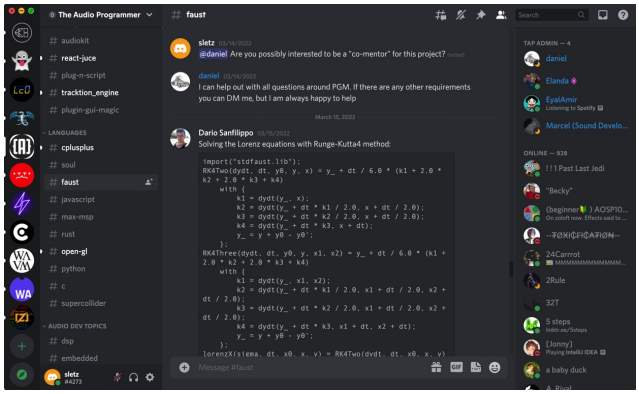

Figure 5: *Faust Discord channel.*

8. ACKNOWLEDGMENTS AND CONCLUSIONS

This paper reflects the richness and diversity of the contributions done in the last two years. Thanks to all contributors for all the different components and projects that have been described!

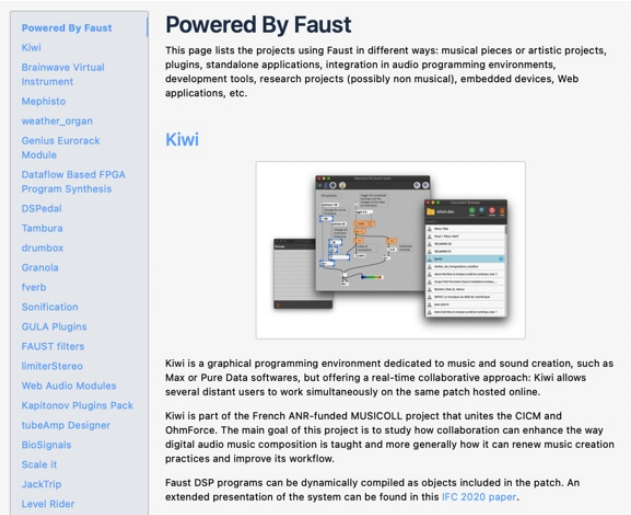

Figure 6: *Part of the "Powered by Faust" list*

9. REFERENCES

[1] Yann Orlarey, Dominique Fober, and Stéphane Letz, "Faust : an Efficient Functional Approach to DSP Programming," in *New Computational Paradigms for Computer Music*, Editions Delatour France, Ed., pp. 65–96. 2009.

[2] Jeff Bezanson, Stefan Karpinski, Viral B Shah, and Alan Edelman, "Julia: A fast dynamic language for technical computing," *ArXiv Preprint*, 2012.

[3] Maxime Popoff, Romain Michon, Tanguy Risset, Yann Orlarey, and Stéphane Letz, "Towards an fpga-based compilation flow for ultra-low latency audio signal processing," in *Proceedings of the 2022 Sound and Music Computing Conference (SMC-22)*, Saint-Étienne, France, 2022, *Paper accepted to the conference but not published yet.*