THE FAUST ECOSYSTEM ONION: EXPLORING THE LAYERS OF AN AUDIO DSP LANGUAGE IN THE AGE OF AI

Stéphane Letz

GRAME - INRIA LYON

IFC, Cannes, June 5th 2026

AGENDA

Let's revisit Faust as a layered ecosystem for audio DSP, in the age of AI

Why DSLs still matter in the age of LLMs

Connecting Faust to AI agents: MCP, real-time testing and library APIs

Solidifying the ecosystem: tests, CI, documentation and maintenance

faust-rs: experimental port of the compiler to Rust with AI based workflows

Differentiable DSP and algebraic compilation perspectives

Conclusions

FAUST IN THE AGE OF AI

Faust is a layered ecosystem: language, compiler, libraries, architectures, deployment tools and applications

Developed over a period of more of 20 years

The key question: are DSLs still relevant when LLMs can write code?

WHY A DSP DSL STILL MATTERS

Faust can be seen as an adequate notation for DSP, both for humans and for AI-assisted workflows

A compact, composable and declarative notation to describe signal-processing

A better semantic signal-to-syntactic noise ratio

Less tokens, so a better fit with model context windows

Libraries encode domain knowledge that LLMs can reuse instead of reinventing

CONNECTING FAUST TO LLMS

MCP can connect an LLM to real-time Faust compilation and execution, and "hear the program"

Debug probes from debug.lib can expose internal signals to the agent

Faust library functions can be exposed through structured JSON APIs

A **virtuous loop** emerges: generate Faust code, compile it, run it, analyze it, iterate

THREE MCP EXAMPLES

faustforge and faustcode

faustbrowser-mcp

AI-ASSISTED FAUST IN PRACTICE

Chaos Audio and Pluginmaker.ai use dynamic Faust/WebAssembly then export C++ workflows

Magda DAW with integrated Faust + IA

Michel Buffa and Jérôme Lebrun: guitar models and Web version

Community examples such as ai-faust-dsp-effects show an emerging pattern

OPEN RESEARCH QUESTIONS

How do we measure and improve the adequacy of a DSL for audio DSP?

Can local LLMs be efficiently used ?

Thomas Mandolini: Multi-Model AI Development Assistant to be tested

SOLIDIFYING THE ECOSYSTEM WITH AGENTS

Maintenance of the compiler, CI, documentation sites

Library hardening: more than 1100 tests with reference responses

```
// #### Test
// ```
// de = library("delays.lib");
// os = library("oscillators.lib");
// fdelay_test = os.osc(440) : de.fdelay(44100, 22050.5);
// ```
```

FAUST COMPILER: CURRENT STATE

Stable production compiler: official release 2.85.5

Mature multi-backend toolchain, with 24 years of layered C++ evolution

On a development branch: **ondemand**, **upsampling** and **downsampling** primitives, currently in scalar mode

Recent experiments: FIR/IIR (aka LTI) automatic reconstruction and graph scheduling

Main challenge: preserve compatibility while making the core easier to test, reason about and extend

FAUST-RS: AN EXPERIMENTAL PORT OF THE FAUST COMPILER TO RUST

A modern low-level language with stronger memory-safety guarantees

Integrated tooling with **cargo** : compiling, formatting, tests, documentation and dependency management

Simplify the compilation of WebAssembly libfaust

Allow libfaust with a C/C++ API

COMPILATION CHAIN

Parser

Boxes: evaluation and propagation

Signals: typing, normalization and transformation

FIR: signal to FIR conversion

Backends: C, C++, Interp, Cranelift, WebAssembly

PORTING STRATEGY (1)

Analyse the C++ codebase with Claude and produce several porting documents

Look for critical points: parsing, tree memory representation

Keep explicit compilation boundaries: parser, eval, propagate, signals, FIR, backends

Treat the supported Faust subset as a living document, updated when the compiler grows

PORTING STRATEGY (2)

Start with a simplified compilation chain and a growing DSP subset

Validate FIR before developing and trusting multiple backends

Use the interpreter backend to test and debug small reduced cases

Generate everything at audio rate first, then reintroduce optimizations

HOW AGENTS HELP PORTING

Take a DSP that does not compile, reduce the failing case and test hypotheses

Compare interpreter traces and generated code behavior

Unit-test added everywhere

Human review remains necessary to reject local "ad hoc" fixes

FAUST-RS AFTER TWO MONTHS

About 92 kLoc of Rust code, including unit tests

Five backends: C, C++, Interp, Cranelift and WebAssembly

libfaust built tested with existing C/C++ programs (using the new **createCraneliftDSPFactoryFromFile** function)

WebAssembly compilation connected to faustwasm and the Faust IDE

All tests pass on the current corpus !

FAUST-RS: FIRST IMPRESSIONS

The generated scalar code is globally comparable to the C++ Faust compiler

Compilation time is longer and still needs analysis

The compiler structure is similar, but data structures and algorithms differ

The code is better documented

DEVELOPMENT METHODOLOGY WITH AI (1)

Write explicit design plans before asking the agent to code

Specify the goal at the right abstraction level: behavior before implementation

Properly define and maintain invariants

Keep a porting log: assumptions, decisions, failed attempts and open questions

DEVELOPMENT METHODOLOGY WITH AI (2)

Experiment with two agents: Claude Code and ChatGPT Codex, first in VS Code, then as standalone tools

Use one agent to review, reduce or challenge the work of the other

Build the best possible feedback loop by guiding agents to create testing tools, such as the FIR module validation tool

RISKS AND FRAGILITIES

New forms of fragility: wrong fixes, lost context, missing persistent memory

Keeping a consistent project memory over weeks of work remains difficult

Git history (going back...), regression tests and human validation remain essential

Growing dependency on AI tools: availability, cost and sustainability...

DIFFERENTIABLE DSP IN FAUST, DONE IN FAUST-RS

Context: DDSP paper showed how DSP structure can be integrated into machine-learning systems

The usual model is: have manually coded DDSP blocks (Pytorch, JAX etc.), do machine learning, recode for real-time use-cases

The new feature is the **integration of differentiable DSP directly into the Faust language**

Use cases: real-time optimization, machine learning, ODE models

DIFFERENTIABLE DSP IN FAUST (2)

Previous work: JAX backend (David Braun)

Ongoing research: Thomas Rushton, from IFC 2024 to AES 2025 Faust Autodiff work, GSoC 2023 FAD and faust-ddsp

New approach: **first-class forward and reverse autodiff primitives** in the language

NEW AUTODIFF PRIMITIVES

FAD (Forward Automatic Differentiation): derivatives propagate forward with the audio signal

Causal and adapted to real-time, efficient with few input parameters

RAD (Reverse Automatic Differentiation): forward pass stores computations, backward pass distributes gradients

Well suited to optimization with many parameters and machine learning workflows (backpropagation)

FAD PRIMITIVE

Add **fad(exp, seed)** langage primitive

Example: **fad(one, (p1, p2, p3))** returns the primal signal and 3 gradients

So **(one, d(one)/d(p1), d(one)/d(p2), d(one)/d(p3))**: the primal signal and 3 gradients

Forward mode uses dual numbers and transports primal values plus derivatives

Operators and mathematical functions receive dual signals

FAD: DIFFERENTIATING STATELESS CIRCUITS

DSP differentiation is applied directly to time-varying signals with no internal state

Example: **z[t] = sin(x[t]) + y[t]** (for **process(x,y) = sin(x) + cox(y);**)

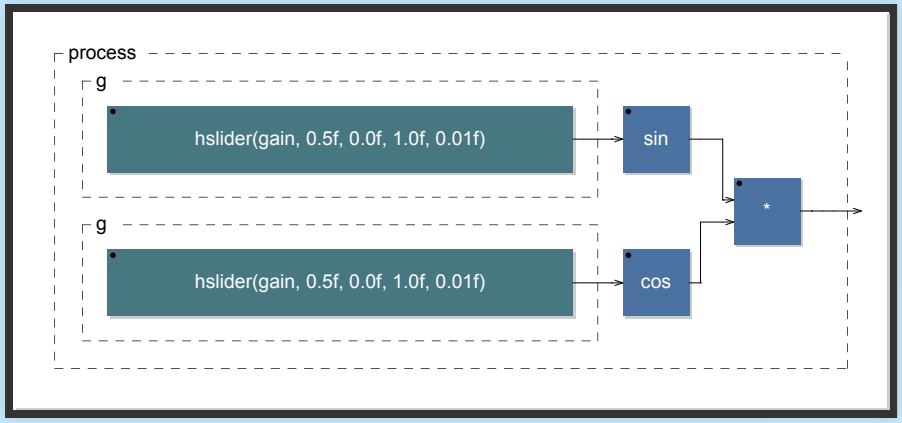
With respect to **x**: **dz[t]/dx[t] = cos(x[t])**

The derivative is computed pointwise at the signal level

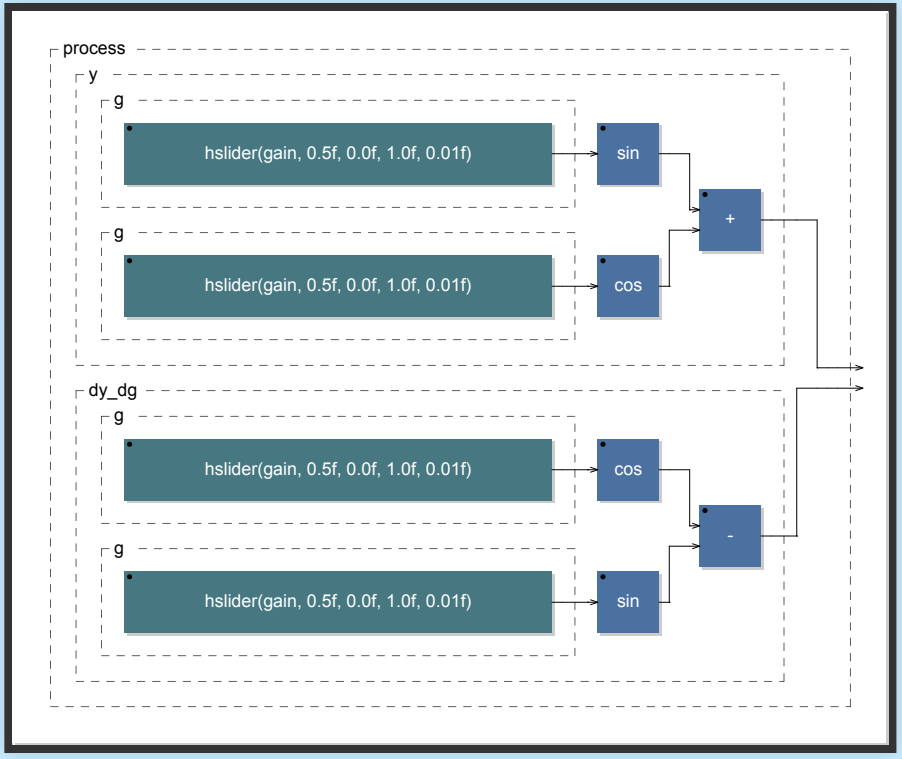
FAD: BLOCK DIAGRAMS

Differentiated DSP = fad(sin(g) + cos(g),g)

Original DSP




```
g = hslider("gain", 0.5, 0, 1, 0.01);
process = sin(g) + cos(g);
```



FAD: DIFFERENTIATING STATEFUL CIRCUITS (1)

DSP differentiation must handle time-varying signals, with delays and recursions

Example: **y[t] = x[t] + g * y[t-1]** (for **process = + ~ *(g);**)

The derivative (relative to **g**) follows the recurrence:

y'[t] = g * y'[t-1] + y[t-1]

FAD: DIFFERENTIATING STATEFUL CIRCUITS (2)

Variable delays require the multivariable chain rule, based on the slope of the delayed signal

For **v[t] = u[t - d[t]]**

The derivative with respect to a parameter **p** is **dv[t]/dp = du[t - d[t]]/dp - dd[t]/dp * slope(u)[t - d[t]]**

FAD: LEARNING A GAIN (1)

Learn a target gain applied to an input signal using gradient descent

Define a loss between the target signal and the learned signal using MSE: **(true - learned)^2**

Use **fad(loss, prev_gain)** to compute **grad = d(loss)/d(prev_gain)**

Express the update **next_gain = prev_gain - rate * grad** as a recursive circuit

FAD: LEARNING A GAIN (2)

```
import("stdfaust.lib");

// 1. Target and signal
target_gain = hslider("gain", 0.5, 0, 1, 0.01);

input = os.osc(440);

true_value = input * target_gain;

// 2. The learning loop
// This loop uses gradient descent on the MSE loss:
// FAD computes d(loss)/d(prev_gain), then the gain is updated
// with prev_gain - rate * grad to converge toward target_gain.
learned_gain = loop ~ _
with {

    // Loop variable representing the current gain estimate
    loop(prev_gain) = next_gain
    with {

        // Learning rate
        rate = 0.01;

        // Current gain estimate applied to the input signal
        learned_value = input * prev_gain;

        // Loss computation (MSE)
        square(x) = x * x;
        loss = square(true_value - learned_value);

        // FAD computes the derivative with respect to prev_gain
        grad = fad(loss, prev_gain) : !, _;
```

Test code

FAD: LEARNING A BIQUAD: FI.TF2(B0,B1,B2,A0,A1)

```
declare name "Biquad Auto-Apprenant (5D FAD)";
declare description "Modifiez les sliders cibles, l'IA les poursuivra en direct.";

import("stdfaust.lib");
import("optimizers.lib");

// ================================================================
// 1. THE DSP MODEL
// ================================================================
// fi.tf2 implements the direct biquad form:
// y(n) = b0*x(n) + b1*x(n-1) + b2*x(n-2) - a1*y(n-1) - a2*y(n-2)
modele_biquad(b0, b1, b2, a1, a2, audio) = fi.tf2(b0, b1, b2, a1, a2, audio);

// ================================================================
// 2. THE USER INTERFACE (alphabetical sorting fix)
// ================================================================
grp_cibles(x) = hgroup("1_CIBLES SECRETES (Humain)", x);
grp_ia(x)     = hgroup("2_APPRENTISSAGE (IA)", x);

// Sliders with numeric prefixes to force b0->a2 ordering
t_b0 = grp_cibles(vslider("[1] Cible b0", 0.1, -2.0, 2.0, 0.001)) : si.smooth(0.99);
t_b1 = grp_cibles(vslider("[2] Cible b1", 0.2, -2.0, 2.0, 0.001)) : si.smooth(0.99);
t_b2 = grp_cibles(vslider("[3] Cible b2", 0.1, -2.0, 2.0, 0.001)) : si.smooth(0.99);

// Tighter bounds for a1 and a2 to guarantee stability (stability triangle)
t_a1 = grp_cibles(vslider("[4] Cible a1", -1.0, -1.90, 1.90, 0.001)) : si.smooth(0.99);
t_a2 = grp_cibles(vslider("[5] Cible a2", 0.4, -0.90, 0.90, 0.001)) : si.smooth(0.99);

// ================================================================
// 3. THE SCENE AND SYSTEM IDENTIFICATION
// ================================================================
process = b0_vis, b1_vis, b2_vis, a1_vis, a2_vis, rendu
with {
    bruit = no.pink_noise; // Pink noise for balanced spectral coverage

    cible_dynamique = modele_biquad(t_b0, t_b1, t_b2, t_a1, t_a2, bruit);

    // Two learning speeds: fast for gain, slow for the poles
    moteur_rapide = rmsprop(0.002);
    moteur_stable = rmsprop(0.0005);

    opts = optimize_5D(
        modele_biquad,
        moteur_rapide, moteur_rapide, moteur_rapide, // b0, b1, b2
        moteur_stable, moteur_stable,                // a1, a2 (slower)
        -2.0, 2.0,
        -2.0, 2.0,
        -2.0, 2.0,
        -1.92, 1.92,    // Increased safety on a1
        -0.92, 0.92,    // Increased safety on a2
        cible_dynamique,
        bruit
    );

    // Extract the 5D bus through direct routing optimized for faust-rs
    b0_opt = opts : _, !, !, !, !;
    b1_opt = opts : !, _, !, !, !;
    b2_opt = opts : !, !, _, !, !;
    a1_opt = opts : !, !, !, _, !;
    a2_opt = opts : !, !, !, !, _;
```

Test code

FAD USE CASES

Newton-Raphson solvers: zero-delay feedback (ZDF) analog modeling.

Real-time adaptive effects: gradient descent for automated sound matching and smart EQs.

Sensitivity analysis: visualization of parameter influence

RAD PRIMITIVE

Add **rad(exp, seed)** langage primitive

Runs differentiation in reverse mode

The forward pass computes the primal DSP and records the dependency graph

The reverse pass propagates adjoints from the loss back to the selected parameters

Cost scales with the number of outputs or losses, not with the number of parameters

RAD AND BPTT

For stateful DSP, reverse mode becomes
Back(P)ropagation Through Time

Delays and recursions require a time horizon or a bounded tape

RAD AS GRAPH TRANSPOSITION

Reverse AD can be understood as transposing the computation graph

A fan-out in the primal graph becomes adjoint accumulation in the reverse graph

Linear DSP subgraphs connect naturally to system transposition and LTI analysis

WHERE RAD FITS IN FAUST

Offline training: many coefficients optimized from one loss signal

Limited online adaptation: short horizons and bounded memory

Hybrid mode: FAD for causal local derivatives, RAD for larger parameter sets

Main compiler questions: tapes, memory, and controllable latency

FAD AND RAD IMPLEMENTED AT PROPAGATION AND FIR STAGES

Faust compilation chain: **source -> boxes -> signals -> FIR -> backends**

fad and **rad** are language primitives carried in the box algebra

During symbolic propagation, the compiler expands them into primal and derivative signal graphs

The generated derivative code then benefits from the same signal optimizations as ordinary Faust code

PERSPECTIVES: FAUST AS A DSP-AWARE ALGEBRAIC COMPILER ?

Move beyond "transpilation" from Faust source to target code

Detect DSP structure: LTI and LTV subgraphs, state-space forms...

Apply algebraic transformations such as Z-domain factorization and filter decomposition

Generate numerically better structures, such as cascades of biquads

CONCLUSIONS

Hybridation between DSL and AIs: where does it make sense ?

Faust DSL remains useful as a compact, testable and deployable DSP notation

AIs and agents can help solidify and extend the ecosystem

faust-rs as en experimentation to solidify the core while opening new compiler experiments

Differentiable DSP shows one possible next step: programs that can be compiled, optimized and learned from

But growing dependency on AI tools: availability, cost and sustainability has to be questioned