COMPILING DIFFERENTIABLE AUDIO GRAPHS TO REAL-TIME DSP

*Facundo Franchino**

Massachusetts Institute of Technology
Cambridge, MA
facundof@mit.edu

Sebastian J. Schlecht

Artificial Audio, Multimedia Communications and Signal Processing
Friedrich-Alexander-Universität Erlangen-Nürnberg
Erlangen, Germany
sebastian.schlecht@fau.de

ABSTRACT

Differentiable audio processors are habitually designed and optimised in machine-learning frameworks, but deploying them as real-time audio effects still often requires non-automatic implementation in a dedicated digital signal processing language. The translation is error-prone, demands an onerous verification process, and detaches research prototypes from usable production tools. That being so, we present ADAC, a compiler that lowers a trained model to a framework-agnostic intermediate representation and emits efficient FAUST code whose impulse response matches the source model to within floating-point arithmetic noise, direct paths included. The optimisation loop is made audible by replacing the model in a running plugin after each gradient step. The exported processor carries a small set of macro-controls that leave its stability intact. A stability certificate computed from the shipped parameters is checked before the plugin is built. At the demonstration, a feedback delay network is trained and exported to a working plugin.

1. INTRODUCTION

Differentiable audio frameworks such as FLAMO [1] permit the optimisation of audio processors by gradient descent. The optimised model encodes a complete processor, yet it remains confined to its training framework. FAUST produces efficient real-time code for various targets through a maintained compilation pathway [2], but a trained model may only reach FAUST by hand, through a rewriting in said dedicated DSP language that must be repeated whenever the model changes. Comparable deployment difficulties come to light in differentiable audio systems such as DDSP [3].

We close this gap with the Automatic Differentiable Audio Compiler (ADAC). The pipeline (Fig. 2) traverses a trained model's computational graph, extracts the learned parameters, lowers the processor to a JSON intermediate representation, and emits equivalent FAUST [4] code, which existing toolchains compile to a variety of hardware and software targets. This representation captures topology (series, parallel, and recursive composition) together with numerical parameters, so source traversal and target emission remain decoupled.

This demonstration uses feedback delay networks (FDNs) [5, 6] as the case study, since they exercise every composition operator and carry the richest parameter structure; the same path also compiles a scattering delay network [7] (Sec. 6). The contribution is not solely code generation, but the workflow around it, in three parts. (i) Training is made audible, the model replaced in a running plugin after every gradient step. (ii) The macro-controls hold the processor stable at any setting. (iii) The exported plugin carries a stability certificate, verified before it is built.

2. THE COMPILER

2.1. From model to representation

An N-line FDN is defined by a delay vector $\mathbf{m} \in \mathbb{N}^N$, a feedback matrix $\mathbf{A} \in \mathbb{R}^{N\times N}$, input and output gains $\mathbf{B} \in \mathbb{R}^{N\times N_{\text{in}}}$, $\mathbf{C} \in \mathbb{R}^{N_{\text{out}}\times N}$, and a direct path $D \in \mathbb{R}^{N_{\text{out}}\times N_{\text{in}}}$, with transfer function

$$H(z) = \mathbf{C}\big(\mathbf{\Delta}(z)^{-1} - \mathbf{A}\big)^{-1}\mathbf{B} + D\,, \tag{1}$$

where $\mathbf{\Delta}(z) = \operatorname{diag}(z^{-m_1}, \ldots, z^{-m_N})$ [6], so that the shortest path through the recursive branch traverses at least $\min_i m_i$ samples.

The extractor walks the model tree and serialises each node into the representation (Fig. 1). Many trained modules hold their parameters in a parametrisation space rather than an audio space. An orthogonal feedback matrix [8] is stored as skew-symmetric weights and exponentiated on every forward pass, and a Householder matrix as a single vector $\mathbf{u}$ from which $\mathbf{I} - 2\mathbf{u}\mathbf{u}^\top$ is reconstructed. The extractor accordingly serialises the *effective* parameter, that is, the value the model applies to the signal, whilst retaining the raw weights so that the source model may be reconstructed from the representation without loss.

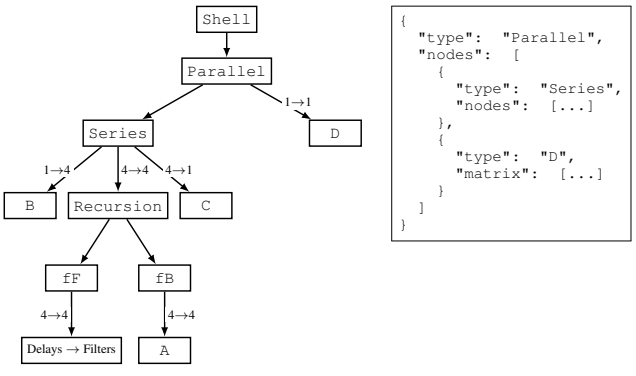

Figure 1: FDN topology. Left: The representation visualised as a tree mapping to state-space formulation matrices. Right: A structural snippet of the corresponding JSON intermediate representation.

* Corresponding author.

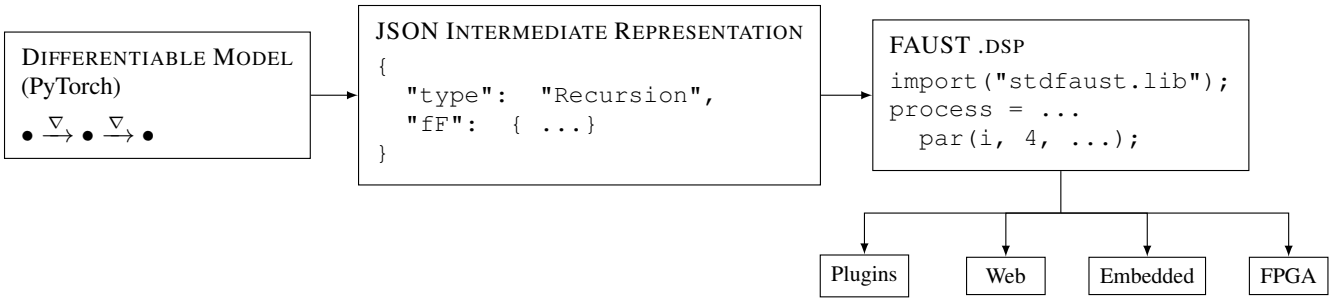

Figure 2: Pipeline overview. The differentiable audio graph is extracted from the host framework (PyTorch) into a framework-agnostic JSON intermediate representation, lowered into functional FAUST DSP code, and from there to any FAUST target: audio plugins (VST3, AU, CLAP), the web (WebAssembly), embedded boards (Bela, Daisy, ESP32), and FPGA (VHDL).

2.2. From representation to FAUST

Each structural node maps to a FAUST composition operator, with Series becoming `:`, Parallel a shared-input split, and Recursion `~`. Leaves emit the corresponding primitives, `@(`n`)` for delays, $\star$`(`g`)` for gains, and `fi.tf2` cascades for filters, whilst mixing matrices are hoisted as named functions with explicit sum-of-products arithmetic. FAUST's recursion operator introduces one sample of implicit delay. The emitter therefore writes $z^{-(m_i-1)}$ for each in-loop line, so that with this implicit delay the round-trip is

$$z^{-1}\, z^{-(m_i-1)} = z^{-m_i}\,, \tag{2}$$

the exact loop period, and a single z^{-1} on the recursion output, outside the loop, restores the absolute arrival time that the in-loop shortening would otherwise advance. When the model's delays are already integer samples, as in our evaluation, the emitted graph matches the source model sample-for-sample up to the single-precision arithmetic of the audio path. Fractional delays, when present, are rounded to the nearest sample, an error of at most $1/(2f_\mathrm{s})$ seconds per line, where f_s is the sample rate.

2.3. Correctness

The feedback matrix $\mathbf{A}$ is the weighted adjacency matrix of a directed graph over the delay lines, and entry (i, j) of $\mathbf{A}^k$ sums the weight products over all length-k walks between them [9]. Equivalently, the FDN is a signal flow graph whose transfer function is a sum over its paths in the sense of Mason [10]. Expanding Eq. (1) as a Neumann series exhibits $H(z)$ as a sum over such walks, so $H(z)$ is determined entirely by $(\mathbf{A}, \mathbf{m}, \mathbf{B}, \mathbf{C}, D)$. The pipeline preserves each of these quantities, and therefore the transfer function. Across the full input-to-output matrix of a stereo FDN, and for a mono FDN with a direct path, the impulse responses of the compiled FAUST agree with the source model to within 7×10^{-5} of peak with no alignment applied (Fig. 3), a residual at the level of single-precision arithmetic rather than structural error. The implementation is exercised by 201 unit tests and an integration suite comparing impulse responses end-to-end.

2.4. Cost

For a dense N-line FDN with S second-order sections per line, the emitted code costs $\Theta(N^2 + NS + N_\mathrm{in}N + N_\mathrm{out}N)$ arithmetic operations per sample and $\Theta(\sum_i m_i + NS)$ state; the matrix emitter drops zero entries, so the N^2 term falls to $\mathrm{nnz}(\mathbf{A})$ for sparse feedback. Benchmarked with `faust2bench` on a single core of an Apple M2 at 48 kHz (Fig. 4), the measured load follows this $\Theta(N^2)$ prediction in the practical range and rises above it once the feedback matrix outgrows the cache. A 32-line FDN, representative of a practical reverberator, runs at roughly ninety times real time, and a 64-line one at fourteen, so even large networks leave ample headroom on commodity hardware.

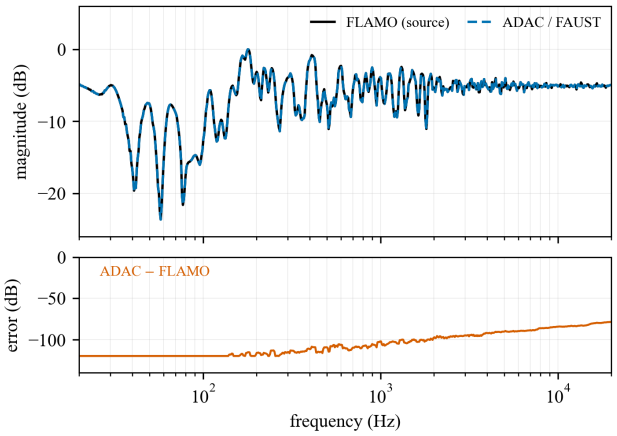

Figure 3: Magnitude response of a compiled four-line FDN, one input-to-output path. Top, the FLAMO source (black) and the emitted FAUST (blue, dashed), $1/12$-octave smoothed; the two coincide across the audible band. Bottom, their difference, at or below -80 dB of the peak, the level of single-precision arithmetic.

3. MACRO-CONTROLS

The raw trained parameters are unsuitable as user controls, being jointly optimised and resident in parametrisation spaces, so that moving any one of them in isolation can destabilise the processor. The compiler instead offers a fixed vocabulary of macro-controls layered onto the generated code, namely reverberation time, dry/wet balance, and pre-delay. The reverberation-time control follows Jot's [5] homogeneous-decay construction. Writing RT for the reverberation time, the interval over which the energy decays by 60 dB, each delay line of length m_i receives the gain

$$g_i = 10^{-3m_i/(f_\mathrm{s}\,\mathrm{RT})}\,, \tag{3}$$

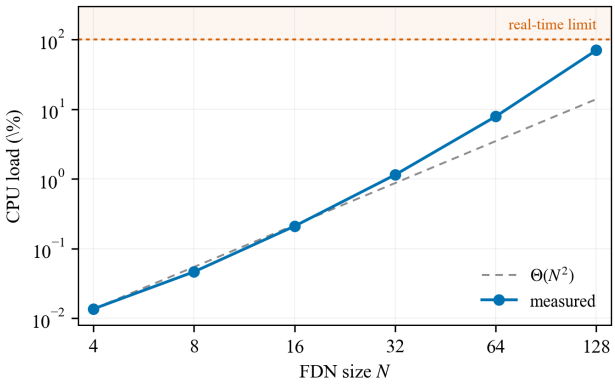

Figure 4: Single-core CPU load of the emitted FAUST against FDN size N, on an Apple M2 at 48 kHz. The measured cost tracks the $\Theta(N^2)$ reference until the feedback matrix exceeds the cache. The shaded region marks loads above real time.

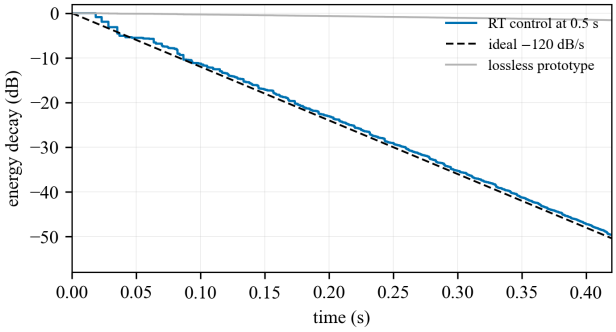

Figure 5: Compiled-plugin energy decay for 0.5 s reverberation time, compared with the ideal $-120\,\mathrm{dB\,s^{-1}}$ line and lossless prototype.

so a single slider, calibrated in seconds, imposes the same decay rate on every line at any sample rate. Measured by Schroeder integration [11] on the compiled plugin, a setting of 0.5 s yields a reverberation time of 0.5000 s (Fig. 5).

4. STABILITY CERTIFICATES

Before anything is shipped, the representation is analysed for stability. The criterion is a small-gain argument [12]. For a loop of elements $E_1, \ldots, E_K$ in series, with $E_k(e^{j\omega})$ the frequency response of element k and $\sigma_{\max}(\cdot)$ the largest singular value of a matrix, the closed loop is stable if

$$\sup_{\omega} \prod_{k=1}^{K} \sigma_{\max}\big(E_k(e^{j\omega})\big) < 1\,, \tag{4}$$

evaluated over a frequency grid, with pole checks on every recursive filter section besides. The spectral radius is reported but is not the criterion, since it says nothing once filters sit in the loop and, for non-normal matrices, permits arbitrarily large transient growth.

The analysis runs on the parameter values exactly as emitted, that is, after decimal formatting and the single-precision cast of the audio path. A trained orthogonal matrix in our evaluation measures $\sigma_{\max} = 1.000000170$ after this chain, marginally above unity, and an analysis of the pristine double-precision values would therefore misdescribe the artefact that ships. A lossless prototype is classified marginally stable; with the reverberation-time control present, the loop is attenuated at every slider position and the export is certified stable outright. The certificate is written as JSON next to the generated code, and the plugin exporter refuses to build a model whose verdict is unstable or unproven unless explicitly overridden.

5. LIVE TRAINING AND DEPLOYMENT

In the demonstration an FDN of $N = 4$ delay lines with learnable output gains is optimised with Adam (learning rate 0.05) for 200 steps at 48 kHz, minimising the mean-squared error (MSE) between its frequency response and a target. The training loop is made audible by a callback that re-emits the model after each optimiser step and atomically rewrites a watched `.dsp` file. A resident plugin built on the FAUST interpreter [2] reloads the file when it changes, recompiling the emitted network in under 10 ms on an Apple M2, whilst ADAC itself re-emits the model in 0.2 ms, so a step becomes audible within a few audio buffers of the write. Slider values survive each reload by address, so the reverberation-time control may be adjusted whilst the optimiser works. Publishing is deduplicated and rate-limited, so a converged or paused optimisation causes no reloads. The host plugin is a CLAP effect [2], wrapped as VST3 and AU so that the same workflow runs in hosts without native CLAP support.

Deployment is a single call comprising code generation, certification, JUCE project generation, release compilation, and installation of validated VST3 and AU binaries into the user's plugin folders, with the macro-controls exposed as automatable parameters in every format. From trained model to installed plugin is at most a few minutes of compilation.

Because the generated code is standard FAUST, it can be pushed to fixed-point targets through the Syfala flow [13]; these requantise the coefficients, and the certificate is recomputed at the target word length.

6. CONCLUSIONS

The case study is an FDN, but ADAC is not restricted to one. Any audio graph built from series, parallel, and recursive composition with parameterised leaves passes through the same traversal and emission unchanged, and a new leaf type requires only a new emitter. As a larger instance, a scattering delay network (SDN) for a six-wall room compiles through the same path; its thirty wall-to-wall delay lines, block-diagonal scattering [7], and per-wall input and output routing reproduce the FLAMO source to within the same single-precision agreement reported for the FDN. What remains is breadth of leaf coverage and a stable specification of the representation against which other frameworks and tools can be written.

A sharper limit is the linear, time-invariant (LTI) setting itself. The emitter extends readily to nonlinear and time-varying processors, since FAUST expresses waveshapers, envelope followers, and modulation directly, and a frontend for a framework that carries such modules, for example dasp [14] or differentiable time-varying filters [15], would bring them into the same representation. The equivalence and stability arguments are tied to that linear setting, so carrying the guarantees across alongside the emitter is the

natural next step. Source code is available on GitHub.[1]

7. ACKNOWLEDGMENTS

The authors thank Gloria Dal Santo for the FLAMO framework and the FAUST team at GRAME for the compiler infrastructure.

8. REFERENCES

[1] G. Dal Santo, G. M. De Bortoli, K. Prawda, S. J. Schlecht, and V. Välimäki, "FLAMO: An open-source library for frequency-domain differentiable audio processing," in *Proceedings of the 2025 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, Hyderabad, India, Apr. 2025.

[2] F. Franchino, S. Letz, and J. Chowdhury, "A dual-mode Faust-to-CLAP compilation system," in *Proceedings of the International Faust Conference (IFC)*, Lyon, France, 2026.

[3] J. Engel, L. Hantrakul, C. Gu, and A. Roberts, "DDSP: Differentiable digital signal processing," in *Proceedings of the International Conference on Learning Representations (ICLR)*, Addis Ababa, Ethiopia, 2020.

[4] Y. Orlarey, D. Fober, and S. Letz, "FAUST: An efficient functional approach to DSP programming," in *New Computational Paradigms for Computer Music*. Paris, France: Delatour, 2009.

[5] J.-M. Jot and A. Chaigne, "Digital delay networks for designing artificial reverberators," in *Proc. 90th Audio Eng. Soc. Convention*, Paris, France, 1991.

[6] S. J. Schlecht and E. A. P. Habets, "On lossless feedback delay networks," *IEEE Transactions on Signal Processing*, vol. 65, no. 6, pp. 1554–1564, Mar. 2017.

[7] ——, "Scattering in feedback delay networks," *IEEE/ACM Transactions on Audio, Speech, and Language Processing*, vol. 28, pp. 1915–1924, 2020.

[8] D. Rocchesso and J. O. Smith, "Circulant and elliptic feedback delay networks for artificial reverberation," *IEEE Transactions on Speech and Audio Processing*, vol. 5, no. 1, pp. 51–63, Jan. 1997.

[9] C. Godsil and G. Royle, *Algebraic Graph Theory*, ser. Graduate Texts in Mathematics. New York, NY: Springer, 2001, vol. 207.

[10] S. J. Mason, "Feedback theory: Some properties of signal flow graphs," *Proceedings of the IRE*, vol. 41, no. 9, pp. 1144–1156, Sep. 1953.

[11] M. R. Schroeder, "New method of measuring reverberation time," *Journal of the Acoustical Society of America*, vol. 37, no. 3, pp. 409–412, 1965.

[12] G. Zames, "On the input-output stability of time-varying nonlinear feedback systems—part I: Conditions derived using concepts of loop gain, conicity, and positivity," *IEEE Transactions on Automatic Control*, vol. 11, no. 2, pp. 228–238, 1966.

[13] P. Cochard, M. Popoff, R. Michon, and T. Risset, "Programming FPGA platforms for real-time audio signal processing in C++," in *Proceedings of the Sound and Music Computing Conference (SMC-24)*, Porto, Portugal, 2024.

[14] C. J. Steinmetz, N. J. Bryan, and J. D. Reiss, "Style transfer of audio effects with differentiable signal processing," *arXiv preprint arXiv:2207.08759*, 2022.

[15] C.-Y. Yu, C. Mitcheltree, A. Carson, S. Bilbao, J. D. Reiss, and G. Fazekas, "Differentiable all-pole filters for time-varying audio systems," in *Proceedings of the International Conference on Digital Audio Effects (DAFx)*, Guildford, UK, 2024, pp. 345–352.

[1] Code: `https://github.com/cucuwritescode/adac`. Documentation: `https://adac.readthedocs.io`.