Audio Engineering Society

Conference Paper 28

Presented at the AES International Conference on Machine Learning and Artificial Intelligence for Audio
2025 September 8–10, London, UK

This paper was peer-reviewed as a complete manuscript for presentation at this conference. This paper is available in the AES E-Library (http://www.aes.org/e-lib), all rights reserved. Reproduction of this paper, or any portion thereof, is not permitted without direct permission from the Journal of the Audio Engineering Society.

Faust Autodiff: Towards Audio Domain-Specific Machine Learning

Thomas Rushton[1], Yann Orlarey[1], Romain Michon[1], Tanguy Risset[1], and Stéphane Letz[2]

[1]*Inria, INSA Lyon, CITI, EA3720 , 69621 Villeurbanne, France*
[2]*Grame CNCM, CITI, EA3720 , 69621 Villeurbanne, France*

Correspondence should be addressed to Thomas Rushton (thomas.rushton@inria.fr)

ABSTRACT

Differentiable programming via automatic differentiation (AD) is the foundation for gradient-based optimisation techniques, and forms the basis for many current approaches to machine learning. Though well catered-for in general-purpose programming languages, the availability of AD in a domain-specific language (DSL) could offer novel perspectives on optimisation problems in the field of audio. We present a general scheme for differentiable programming in the Faust programming language, a high-performance DSL tailored to audio synthesis and signal processing. Faust's rich ecosystem, coupled with a comprehensive AD implementation, can provide support for audio optimisation and machine learning applications on a multitude of platforms, from FPGAs to the web.

1 Introduction

Gradient-based methods are ubiquitous in tackling a wide variety of mathematical optimisation and machine learning problems, and the availability of such methods hinges on the possibility of computing the derivatives of computer programs. In the field of machine learning in particular, automatic differentiation (AD) [1] has emerged as the dominant differentiable programming paradigm. In AD, the primitive operations of a programming language are defined differentiably, thus programs composed from those operations are in turn differentiable end-to-end with respect to their input variables. In a gradient-based optimisation scenario, the partial derivatives of an algorithm terminating in a loss function constitute a gradient vector, which can be backpropagated, causing the algorithm's input variables, weights in a neural network perhaps, to approach optimal values. Techniques of this sort employed in the audio domain are referred to as *differentiable digital signal processing* (DDSP) [2], and find applications from least mean squares (LMS) filtering [3] to differentiable audio synthesis [4].

Building on previous work on AD in the Faust programming language [5], here we exploit Faust's pattern matching abstraction to extend to a more general AD syntax, demonstrating that arbitrary compositions of signal processing blocks can be differentiated, and to higher order if desired. With AD in Faust, it is possible to create differentiable infinite impulse response (IIR) filters and neural network building blocks. Our work provides the basis for an audio-focused ecosystem for the development of solutions to problems based on machine learning.

2 The Faust Programming Language

Faust is a functional paradigm, domain specific language for audio digital signal processing [6]. It is built atop a domain-agnostic *block diagram algebra* [7, 8] that describes primitive operations connected into graphs via a selection of *composition operators* (see Figure 2). A textual language, Faust can, for example, define a diagram consisting of two blocks, A and B, composed in sequence, with code of the form

```
process = A : B;
```

Lst. 1: Sequential composition of two Faust blocks.

Conceptually, a given block diagram, D, may undergo semantic transformation, notated $[\![\cdot]\!]$, into a signal processor [8]:[1]

$$\begin{gathered} [\![D]\!] \,:\, \mathbb{S}^n \to \mathbb{S}^m \\ [\![D]\!](s_1,\ldots,s_n) \to (y_1,\ldots,y_m) \\ \mathbf{y}(t) = f(\mathbf{s}(t)) \,, \end{gathered} \tag{1}$$

where $\mathbb{S}$ is the set of all signals, f is a mathematical function analogous to the primitive operations and compositions of D, and t is time. The signal processor receives a vector of input signals, $\mathbf{s}$, and produces a vector of output signals, $\mathbf{y}$.

2.1 The Faust Ecosystem

Faust is a highly optimised, compiled language [9], with C++ as its default compilation target. One of the great strengths of the Faust ecosystem is its *write once, run anywhere* model; a variety of additional backends are supported, including LLVM [10], WebAssembly [11], and Rust [12], plus numerous software and hardware platforms. Faust code can be compiled for use in VST plugins [13], for embedded systems [14] and FPGAs [15]; it is used by artists and digital luthiers [16, 17, 18], and features sophisticated libraries for wave digital filter modelling [19] and physical modelling synthesis [20, 21], amongst others.

[1] The colon symbol on the first line of Equation (1) introduces the *type* of the signal processor $[\![D]\!]$ and is not to be confused with Faust's sequential composition operator.

2.2 Differentiable Programming in Faust

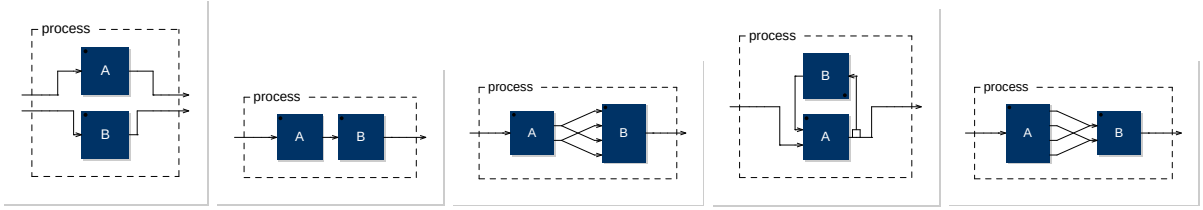
In light of Faust's versatility, efficiency, and strengths as a DSL in tackling audio problems it may come as little surprise that it has been turned to machine learning problems in the audio domain, via *DawDreamer* [22], which uses a Python backend to transpile Faust code for gradient-based optimisation with the JAX Python library. JAX, and other well-known machine learning frameworks, such as TensorFlow and PyTorch, enable gradient-based optimisation approaches by way of *automatic differentiation*, a technique for computing the numerical outputs of both a mathematical expression (or, more commonly, a computer program) and its derivatives in parallel [23, 1, 24].

Differentiable programming [25] is, broadly speaking, the creation of computer programs using primitive operations which are themselves differentiable; a computer program consisting of differentiable primitives may be differentiated automatically, producing the desired *primal* and *gradient* numerical outputs for use in a gradient-based optimisation algorithm. In this vein, a Faust-based LMS algorithm has been used for real-time acoustic control [3], though the DDSP aspect of this work was limited to the particular application. The feasibility of a more general approach to DDSP in the Faust language has been demonstrated via the implementation of forward-mode AD primitives [5].

One way of framing forward-mode AD is as an arithmetic of dual numbers [23, 26] of the form $\langle u, \nabla u \rangle$, where u is the aforementioned primal, and $\nabla u = \begin{bmatrix} \frac{\partial u}{\partial x_1} & \cdots & \frac{\partial u}{\partial x_N} \end{bmatrix}^T$, the gradient of u, its partial derivatives with respect to a vector of N input variables, $\mathbf{x}$. A dual number can be passed into a forward-mode differentiable primitive, f, yielding an output dual number, $\langle f(u), \nabla u f'(u) \rangle$. Faust is a language of *signals* and signal processors; consequently we will refer to the *dual signal*, $\langle s, \nabla s \rangle \in \mathbb{S}_d$, where $\mathbb{S}_d = \mathbb{S}^{N+1}$. The signal processor based on the differentiated block diagram D has the following semantic scheme:

$$\begin{gathered} \Delta_N[\![D]\!] \,:\, \mathbb{S}_d^M \to \mathbb{S}_d^P \\ \Delta_N[\![D]\!](\langle s_1, \nabla s_1 \rangle, \ldots, \langle s_M, \nabla s_M \rangle) \\ \to (\langle y_1, \nabla y_1 \rangle, \ldots, \langle y_P, \nabla y_P \rangle) \\ \langle \mathbf{y}(t),\, \mathbf{J}_\mathbf{y}(t) \rangle = \langle f(\mathbf{s}(t)),\, \mathbf{J}_f(t)\mathbf{J}_\mathbf{s}(t) \rangle \,, \end{gathered} \tag{2}$$

where $\Delta_N[\![\cdot]\!]$ indicates derivative transformation with respect to N input variables, and $\mathbf{J}_\mathbf{y}$, $\mathbf{J}_f = \nabla f(\mathbf{s})$, and



Parallel, A, B. Sequential, $A : B$. Split, $A <: B$. Recursive, $A \sim B$. Merge, $A :> B$.

Fig. 2: Example compositions of Faust blocks.

$\mathbf{J_s} = \nabla \mathbf{s}$ are, respectively, $P \times N$, $P \times M$ and $M \times N$ Jacobian matrices.

AD primitives have been demonstrated in Faust using the language's pattern matching abstraction [5]. Pattern matching is a source code transformation mechanism, and exploits the fact that a Faust function may receive, as a parameter, a programmatic expression. For example, a differentiable addition primitive, acting on dual signals $\langle s_1, \nabla s_1 \rangle, \langle s_2, \nabla s_2 \rangle$, may be created by defining a function that accepts the addition operator as a parameter, and produces, in parallel, the addition primitive's primal output, $s_1 + s_2$, and its gradients, $\nabla s_1 + \nabla s_2$. A

```
env(vars) = environment {
  // ...
  diff(+) = interleave(vars.N + 1, 2)
    : +,par(i, vars.N, +);
  // ...
};
```

Lst. 2: Definition of a differentiable addition primitive in the Faust language.

dual-signal addition operator implementation is given in Listing 2; the operator is defined within a *differentiable environment*, which receives and enumerates a vector of input variables, ensuring that all derivative transformations produce the appropriate number of partial derivatives; the function `interleave` delivers the primal signals and their respective partial derivatives to the addition operations in the appropriate order: $s_1, s_2, \frac{\partial s_1}{\partial x_1}, \frac{\partial s_2}{\partial x_1}, \ldots, \frac{\partial s_1}{\partial x_N}, \frac{\partial s_2}{\partial x_N}$, as illustrated in Figure 3. One addition primitive is created for the primal operation, and a further N additions in parallel, which we can notate $\|_{l=1}^{N} +$, for computing partial derivatives.

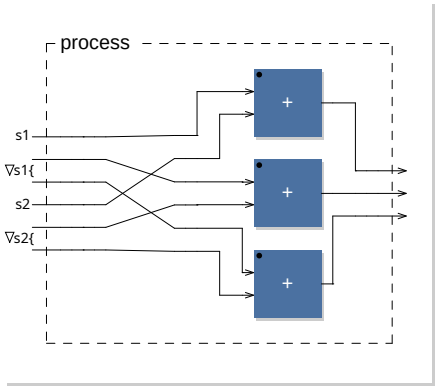


Fig. 3: Differentiable addition primitive applied to dual signals $\langle s_1, \nabla s_1 \rangle, \langle s_2, \nabla s_2 \rangle$; implicit input variable vector $\mathbf{x}$ of length $N = 2$.

2.3 Motivation

Considering the plethora of existing approaches to DDSP [2], why pursue an implementation in Faust? The availability of the Faust-to-JAX transpiler would appear to obviate such an initiative, but for the fact that the AD transformations facilitated by JAX lie the wrong side of the transpilation step. A comprehensive differentiable programming framework in the Faust language itself would allow DDSP algorithms to be written once, and run anywhere. A domain-specific approach to audio-oriented machine learning is not without precedent; indeed, a suite of ML-centric tools, with an emphasis on human computer interaction, exists for the ChucK programming language [27]. Faust, however, with its functional approach, and cross-platform capabilities, offers a unique point of vantage.

The observation has been made that the relative structures of neural networks and digital filters bear striking similarities [28]. There is a very real sense in which

machine learning problems, and gradient-based optimisation problems in general, are signal processing problems. The possibility of turning a signal processing DSL to problems of this sort paves the way to novel approaches and fresh perspectives on this dynamic field of research.

3 A General Scheme for Differentiating Faust Programs

In Faust's block diagram algebra, a given block diagram, D, is comprised of a primitive block, P, or the composition of two block diagrams via a composition operator according to the syntactic rule

$$D = P \mid (D_1 \diamond D_2) \,, \tag{13}$$

where $\diamond$ is one of the sequential (:), parallel (,), split (<:), merge (:>), or recursive ($\sim$) composition operators. The creation of differentiable Faust programs has been shown to be possible by *undifferentiated* composition of differentiated primitive blocks [5], the syntactic rule being

$$\Delta_N[\![D]\!] = \Delta_N[\![P_1]\!] \diamond \Delta_N[\![P_2]\!] \diamond \cdots \diamond \Delta_N[\![P_n]\!] \,. \tag{14}$$

While effective in facilitating differentiable programming, this syntax imposed an undesirable verbosity. Our preferred syntactic rule should be a direct transformation of Equation (13):

$$\Delta_N[\![D]\!] = \Delta_N[\![P]\!] \mid \Delta_N[\![D_1 \diamond D_2]\!] \,. \tag{15}$$

The ideal outcome is to permit the programmer to subject an entire Faust program to automatic differentiation in a single expression, minimising the amount of re-writing required to ready their code for transformation. This would render a great proportion of existing Faust code differentiable, and, in doing so, prepare that wealth of material for prospective use in optimisation problems, and machine learning applications. In order to achieve such an outcome, it is necessary to provide differentiation rules for the five composition operators, taking into account their arithmetical significance, if any, in a block diagram to which they contribute.

3.1 Differentiable Block-Diagram Compositions

To define syntactic rules for the derivatives of Faust's composition operators entails analysing them as if they were themselves primitive blocks. To do this, we must consider them in tandem with rules for some of Faust's fundamental primitive operators; these will assist us in achieving differentiable compositions.

3.1.1 Identity Primitive

Faust's identity function, notated _, describes a connection between two blocks. Treated as a signal processor, it adheres to the following semantic scheme [8]:

$$\begin{aligned} [\![_]\!] &: \mathbb{S} \to \mathbb{S} \\ [\![_]\!](s) &\to (s) \,. \end{aligned} \tag{16}$$

It receives one signal and returns that same signal unchanged, analogous to multiplying it by a constant signal of value 1. In dual signal arithmetic, this is

$$\langle s, \nabla s \rangle \langle 1, \nabla 1 \rangle = \langle 1s, 1\nabla s + s\nabla 1 \rangle = \langle s, \nabla s \rangle \,, \tag{17}$$

and the semantic rule for the derivative of the identity function is thus

$$\begin{aligned} \Delta_N[\![_]\!] &: \mathbb{S}_d \to \mathbb{S}_d \\ \Delta_N[\![_]\!](\langle s, \nabla s \rangle) &\to (\langle s, \nabla s \rangle) \,. \end{aligned} \tag{18}$$

The dual signal $\langle s, \nabla s \rangle$ being composed of one primal signal and N gradient signals, we can further state that

$$\Delta_N[\![_]\!] = \|_{l=1}^{N+1} _ \,. \tag{19}$$

3.1.2 Parallel Composition

Blocks composed in parallel (see Figure 1a) do not influence one-another, arithmetically-speaking. The outputs of the program A, B, $y_1, \ldots, y_{o_A}$ and $y_{o_A+1}, \ldots, y_{o_A+o_B}$, are essentially the outputs of separate block diagrams; thus $\Delta_N[\![A,B]\!] = \Delta_N[\![A]\!], \Delta_N[\![B]\!]$, and, more generally

$$\Delta_N \left[\!\!\left[\|_{l=1}^{L} D_l \right]\!\!\right] = \|_{l=1}^{L} \Delta_N[\![D_l]\!] \,. \tag{20}$$

3.1.3 Sequential Composition

Implicit in Equations (1) and (2) are the input and output signals of the Faust block diagram, D, and the following equivalence applies:

$$D = \|_{l=1}^{i_D} _ : D \tag{21}$$

$$D = D : \|_{l=1}^{o_D} _ \,, \tag{22}$$

where i_D and o_D are, respectively, the number of input and output signals of D. Applying derivative transformation to Equation (21), and bringing Equations (19) and (20) to bear,

$$\begin{aligned} \Delta_N[\![D]\!] &= \Delta_N[\![\|_{l=1}^{i_D} _ : D]\!] \\ &= \|_{l=1}^{i_D} \|_{m=1}^{N+1} _ : \Delta_N[\![D]\!] \\ &= \|_{l=1}^{i_D} \Delta_N[\![_]\!] : \Delta_N[\![D]\!] \,. \end{aligned} \tag{23}$$

$$\Delta_N[\![A,B]\!] = \Delta_N[\![A]\!],\Delta_N[\![B]\!] \quad (3)$$

$$\Delta_N[\![A : B]\!] = \Delta_N[\![A]\!] : \Delta_N[\![B]\!] \quad (4)$$

$$\Delta_N[\![A <: B]\!] = \Delta_N[\![A]\!] <: \Delta_N[\![B]\!] \quad (5)$$

$$\Delta_N[\![A \sim B]\!] = \Delta_N[\![A]\!] \sim \Delta_N[\![B]\!] \quad (6)$$

$$\Delta_N[\![A :> B]\!] = \Delta_N[\![A : \chi(i_B,k)]\!] : F : \Delta_N[\![B]\!] \quad (7)$$

$$F = \|_{l=1}^{i_B} \left(\|_{l=1}^{k} \Delta_N[\![_]\!] \; :> \; \Delta_N[\![_]\!] \right)$$

$$k = \frac{o_A}{i_B}$$

Parallel iteration, `par(l, L, D)`:

$$\|_{l=1}^{L} D_l = D_1, \ldots, D_L \quad (8)$$

Signal interleaving function:

$$\chi(r,c) = \|_{l=1-}^{r\times c} <: \|_{m=0}^{r-1} \|_{n=1}^{c} \left(\|_{p=1}^{m} !, _, \|_{q=1}^{r-m-1} ! \right) \quad (9)$$

$$\Delta_N[\![_]\!] = \|_{l=1-}^{N+1} \quad (10)$$

$$\Delta_N[\![!]\!] = \|_{l=1}^{N+1} ! \quad (11)$$

$$\Delta_N[\![\texttt{mem}]\!] = \|_{l=1}^{N+1} \texttt{mem} \quad (12)$$

Fig. 4: Automatic differentiation rules for Faust's composition operators, plus supporting primitive operators and routing expressions. Use of $\Delta_N[\![\texttt{mem}]\!]$ is implicit in the definition of $\Delta_N[\![A \sim B]\!]$. N is the number of input variables with respect to which partial derivatives are calculated.

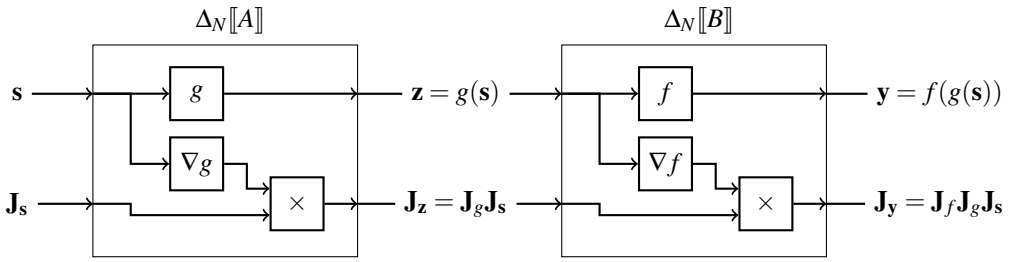


Fig. 5: Derivative transformation of the sequential composition of two Faust blocks, $\Delta_N[\![A : B]\!]$, is analogous to composition of two mathematical functions of a vector of dual signals, $f(g(\langle \mathbf{s}, \mathbf{J_s} \rangle))$, and is thus equivalent to the sequential composition of the blocks following separate derivative transformation.

The arithmetical transparency of sequential composition holds in the general case; expanding on the semantic scheme in Equation (2), $\Delta_N[\![A : B]\!]$ is analogous to composing two functions, g and f:

$$\begin{aligned} &\Delta_N[\![A : B]\!](\langle \mathbf{s}, \mathbf{J_s} \rangle) \to (\langle \mathbf{y}, \mathbf{J_y} \rangle) \\ &\langle \mathbf{y}(t), \mathbf{J_y}(t) \rangle = f(g(\langle \mathbf{s}(t), \mathbf{J_s}(t) \rangle)) \\ &\qquad = \left\langle f(g(\mathbf{s}(t))), \mathbf{J}_f(t)\mathbf{J}_g(t)\mathbf{J_s}(t) \right\rangle . \end{aligned} \quad (24)$$

As illustrated in Figure 5, the composition of functions of dual signals has the effect of primal function composition on the one hand, and application of the chain rule of differentiation on the other, and works so long as $o_A = i_B$,[2] thus

$$\Delta_N[\![A : B]\!] = \Delta_N[\![A]\!] : \Delta_N[\![B]\!] . \quad (25)$$

[2]Though not the case in Faust's early days [8], the equality of o_A and i_B is long-established as an absolute condition of sequential composability.

3.1.4 Split Composition

Composing A and B, $i_B = ko_A$, with the split composition operator (see Figure 1c) duplicates the output signals of A, k times, passing each group of duplicates to B, in order:

$$A <: B = A \; : \; \|_{l=1-}^{o_A} <: \; \|_{m=1}^{k} \|_{l=1-}^{o_A} \; : \; B . \quad (26)$$

Signal order is maintained within each group of duplicates, so, applying derivative transformation to both sides of Equation (26), and recalling Equation (23),

$$\begin{aligned} &\Delta_N[\![A <: B]\!] \\ &= \Delta_N[\![A]\!] : \|_{l=1}^{o_A} \Delta_N[\![_]\!] <: \|_{m=1}^{k} \|_{l=1}^{o_A} \Delta_N[\![_]\!] : \Delta_N[\![B]\!] \\ &= \Delta_N[\![A]\!] <: \Delta_N[\![B]\!] . \end{aligned} \quad (27)$$

3.1.5 Cut Primitive

The cut primitive, notated !, terminates a signal, and combinations of the cut and identity functions can

be used to create complex signal routings. For instance, to interleave two groups of two signals, such that $s_{1,1}, s_{1,2}, s_{2,1}, s_{2,2}$ becomes $s_{1,1}, s_{2,1}, s_{1,2}, s_{2,2}$, one may write

$$_,_,_,_ <: _,!,_,!,!,_,!,_ \,. \tag{28}$$

To differentiate an expression of this kind, the appropriate accompaniment to Equations (19) and (27) for the derivative of the cut primitive is

$$\Delta_N[\![!]\!] = \|_{l=1}^{N+1} ! \,. \tag{29}$$

A general interleaving function, as used in Listing 2, can be written using the cut and identity primitives, plus parallel and split composition (see Listing 3). Comprising only differentiable elements, it too is differentiable.

```
interleave(r, c) = par(l, r * c, _)
  <: par(m, r,
       par(n, c, cutpq(m, r - m - 1)))
with {
  cutpq(0, q) = _,par(l, q, !);
  cutpq(p, 0) = par(l, p, !),_;
  cutpq(p, q) =
    par(l, p, !),cutpq(0, q);
};
```

Lst. 3: A Faust function, $\chi(r,c)$ (see Equation (9) for its mathematical definition), that interleaves c groups of r signals into r groups of c signals.

3.1.6 Delay Primitive

Faust provides two primitives for delaying signals; here we will focus on the `mem` primitive, which, when composed in sequence with a signal, applies a one-sample delay. In general, a fixed delay can be represented as the convolution of a signal, $s(x,t)$, with a time-shifted impulse [29],

$$s(x,t-\tau) = s(x,t) * h(t) \,, \quad h(t) = \delta(t-\tau) \,. \tag{30}$$

Taking the temporal-parametric Fourier transform of the derivative of the convolution with respect to x, then using the convolution and differentiation properties of the transform [29], yields

$$\begin{aligned} &\mathscr{F}\left[\frac{\partial}{\partial x}(s(x,t) * h(t))\right] \\ &= j\omega_x S(j\omega_x, j\omega_t) H(j\omega_t) \qquad (31) \\ &= \mathscr{F}\left[\frac{\partial}{\partial x} s(x,t)\right] \cdot \mathscr{F}\left[h(t)\right] \,, \qquad (32) \end{aligned}$$

which demonstrates that differentiation is time-invariant, and allows us to deduce the convolution rule of differentiation — to differentiate a convolution we need only differentiate one of the functions being convolved. Applied to dual numbers,

$$\langle s, \nabla s \rangle * \langle h, \nabla h \rangle = \langle s * h, \nabla s * h \rangle \,, \tag{33}$$

giving us the following semantic scheme:

$$\begin{aligned} \Delta_N[\![\texttt{mem}]\!] &: \mathbb{S}_d \to \mathbb{S}_d \\ \Delta_N[\![\texttt{mem}]\!](\langle s, \nabla s \rangle) &\to (\langle y, \nabla y \rangle) \\ \langle y(t), \nabla y(t) \rangle &= \langle s(t-1), \nabla s(t-1) \rangle \,. \end{aligned} \tag{34}$$

3.1.7 Recursive Composition

The differentiability of recursive filters is well-established [30] — in short, the derivative of a recursive expression is itself recursive. In the specific case of Faust's recursive composition operator (see Figure 1d), to establish a rule for derivative transformation we can take a similar approach to Equation (24), making g an inner function that acts upon a vector of past output signals:

$$\begin{aligned} \langle \mathbf{y}(t), \mathbf{J}_\mathbf{y}(t) \rangle = \Big\langle & f(\mathbf{s}(t), g(\mathbf{y}(t-1))), \\ & \frac{\partial f}{\partial \mathbf{s}(t)} \mathbf{J}_\mathbf{s}(t) + \frac{\partial f}{\partial g(\mathbf{y}(t-1))} \mathbf{J}_g(t-1) \mathbf{J}_\mathbf{y}(t-1) \Big\rangle \,. \end{aligned} \tag{35}$$

In the absence of a feedforward component (notwithstanding the fact that such a system would produce constant output signals of value zero), the result simplifies to

$$\langle f(g(\mathbf{y}(t-1))), \, \mathbf{J}_f(t-1) \mathbf{J}_g(t-1) \mathbf{J}_\mathbf{y}(t-1) \rangle \,, \tag{36}$$

which, comparing with Equation (24), tells us that recursive composition is simply *sequential composition through time*, and points to the implicit, and trivially-differentiable, one-sample delay introduced by Faust's recursive composition operator. We may conclude that

$$\Delta_N[\![A \sim B]\!] = \Delta_N[\![A]\!] \sim \Delta_N[\![B]\!] \,. \tag{37}$$

3.1.8 Merge Composition

Merge composition (Figure 1e) entails summation [8], thus the addition rule of differentiation must be obeyed when merging dual signals. A merge $A :> B$, can be viewed as an interleaving of the outputs of A into i_B groups of $k = o_A/i_B$ signals, followed by a merging of each group into one signal path:

$$A :> B = A : \chi(i_B, k) : \|_{l=1}^{i_B} \left(\|_{m=1}^{k} _ :> _ \right) : B \,. \quad (38)$$

Clearly $\Delta_N[\![A :> B]\!]$ cannot be defined recursively, however. Note that for $i_B = 1$ the right hand side of Equation (38) reduces to $A :> _$, which is essentially the term in parenthesis, and can be handled thanks to the definitions in Equations (19) and (20),

$$\Delta_N[\![\|_{m=1}^{k} _ :> _]\!] = \|_{m=1}^{k} \Delta_N[\![_]\!] :> \Delta_N[\![_]\!] \,. \quad (39)$$

The order of signals is maintained, so the resulting dual signal is guaranteed to be of the desired form $\left\langle \sum_{m=1}^{k} s_m, \sum_{l=m}^{k} \nabla s_m \right\rangle$. For the full expression for differentiable merge composition, see Equation (7).

4 Discussion

The rules for differentiating Faust's composition operators are collected in Figure 4. With the implementation of these rules via pattern matching, Faust programs can be recursively decomposed according to their composition operators, and re-composed in differentiable fashion, satisfying the syntactic rule in Equation (15). For example:

$$\begin{aligned} \Delta_N[\![(A, C) : B]\!] &= \Delta_N[\![A, C]\!] \ : \ \Delta_N[\![B]\!] \\ &= \Delta_N[\![A]\!], \Delta_N[\![C]\!] \ : \ \Delta_N[\![B]\!] \,. \end{aligned} \quad (40)$$

An artificial neuron of the form $y = \sigma(\mathbf{w}\mathbf{x} + b)$, such as could be used in a binary classification problem, can be created as per Listing 4. The code describes a neuron and sigmoid activation function in primal form, in a library file, `diff.lib`; the `dot` function (line 14) makes use of the familiar `interleave` function to compute the dot product, **wx**; in `neuron.dsp`, differentiable weights and biases, `wb`, are created according to a network layer specification (line 27) — one pair of signals indicating one layer containing one neuron with `NW` weights; weights and biases are used to create a differentiable environment (`d`, line 28; described in Section 2.2 and Listing 2); the wrapper-function `diffInN`

```
1  // diff.lib
2  neuron(NINPUTS, activationFunction) =
3    WX + b : activationFunction
4  with {
5    WX = linalg.dot(NINPUTS);
6    b = _;
7  };
8
9  activations = environment {
10   sigmoid = *(-1) : 1 / (1 + exp);
11 }
12
13 linalg = environment {
14   dot(N) = interleave(N, 2)
15     : par(i, N, *) :> _;
16 };
17
18 // neuron.dsp
19 NW = 3; // Number of weights.
20
21 // Send inputs, and weights/biases,
22 // to the differentiated neuron.
23 process = par(i, NW, _),
24   par(i, NW + 1, wb.var(i + 1))
25   : d.diffInN(neuron, NW)
26 with {
27   wb = weightsAndBiases((1, NW));
28   d = df.env(wb);
29   neuron = d.neuron(NW,
30     d.activations.sigmoid);
31 };
```

Lst. 4: Excerpt from a differentiable artificial neuron implementation. See Section 4 for detail.

(line 25) enacts the recursive derivative transformation; its first `NW` inputs are considered input signals, **s**, the rest as input variables, **x**.

Worthy of reflection is that algorithms of this sort, consisting of differentiable primitives, plus our now-differentiable composition operators, can be automatically differentiated to arbitrary order, $\Delta_N[\![\Delta_N[\![\cdots \Delta_N[\![\cdot]\!] \cdots]\!]\!]$.

4.1 Limitations

Although the scheme presented here renders a great deal of existing Faust code differentiable, certain primitives and language constructs cannot, at present, be subjected to AD.

Faust's `route` primitive cannot be pattern-matched, which precludes the derivative transformation of arbitrary signal routing expressions based on that primitive, such as those in the `routes.lib` library.[3] As described in Section 3.1.5, however, such expressions can be recreated with the identity and cut primitives; common routing tasks like interleaving can be achieved differentiably with code such as in Listing 3.

User interface primitives (sliders and buttons, for example) are matchable in principle, but in practice, `hslider("x1", 0, -1, 1, .01)`, whose parameters are a string identifier, initial value, minimum, maximum and step, must be matched verbatim; clearly, in the general case, this is not practical. The obvious derivative transformation of a user interface element is to an input variable, i.e. a parameter for optimisation such as a weight in a neural network. Input variables must be identifiable so as to create the appropriate corresponding dual signal, $\left\langle x_i, \begin{bmatrix} 0 & \cdots & \frac{\partial x_i}{\partial x_i} & \cdots & 0 \end{bmatrix}^T \right\rangle$. The approach proposed is to move input variables out of the general AD transformation; a wrapper function is used to indicate the number of input signals (whose gradients are zero vectors) and input variables received by the main body of the differentiated block diagram (see Listings 4 and 5). Though imposing a certain coding convention, we consider this comparable with the particular requirements imposed by PyTorch, TensorFlow, and JAX, with their respective dedicated datatypes for learnable parameters.

Though not a frequently-employed feature of the language, unapplied lambda abstractions appear in some key definitions, the `iir` function in `filters.lib`, for example. Implemented in transposed-direct-form [31], `iir` is defined with the abstraction `ma.sub`, which accepts two arguments, `x` and `y`, and returns the result `y-x`. Attempting $\Delta_N[\![\texttt{sub}]\!]$, the abstraction is essentially treated as a numerical constant, and given a vector of zeros as its gradient. Fortunately, constructions of this kind — in this case reversing signal order before applying some primitive operation — are effectively matters of signal-routing and, as described in Section 3.1.5, can be recreated using differentiable primitives.

4.2 Applications & Outlook

Despite the above limitations, and beyond the example given in Listing 4, the AD system described here can be put to practical use. A differentiable transposed-direct-form IIR filter implementation is given in Listing 5; though reliant on some careful signal routing, particularly for directing coefficients to the feedback (`fir(N)`, line 7) and feedforward (`fir(N + 1)`, line 9) portions of the filter, it overcomes the obstacles to differentiability posed by its counterpart in `filters.lib`. Structurally, it is similar to the neuron implementation in Listing 4; parameters, in this case learnable filter coefficients, are defined outside of the differentiable environment, and passed, along with an input signal, to the wrapper function `diffInN`. An expanded version of this example (plus that given in Listing 4) is available online[4] and can be run in Faust's web IDE.[5] Comparing, via a differentiable loss function, the output of the filter against that of an equivalent filter with fixed, *hidden* parameters, and backpropagating the resulting gradients, learnable parameters can be optimised in real time. This generalises to filters of higher order and those described by metaparameters such as cut-

```
1  N = 2; // filter order.
2
3  // ...Set up AD vars & environment...
4
5  // Differentiable TDF IIR filter.
6  iir = ((sub,par(l, N, _))
7    ~ (fir(N))),par(l, N + 1, _)
8    : _,par(l, N, !),par(l, N + 1, _)
9    : fir(N + 1)
10 with {
11   sub = _,_ <: !,_,_,! : -;
12   fir(n) =
13     (_ <: par(l, n, seq(m, l, mem))),
14       par(l, n, _)
15     : linalg.dot(n);
16 };
17
18 // Send a signal input, and coeffs,
19 // to the differentiated filter.
20 process = _,coeffs : d.diffInN(iir, 1)
21 with {
22     coeffs = par(i, 2 * N + 1,
23                  vars.var(i + 1));
24 };
```

Lst. 5: Excerpt from a differentiable IIR filter implementation.

[3] https://github.com/grame-cncm/faustlibraries/blob/65594336a3476d9a82a4df71e88c92f3e4b94821/routes.lib

[4] https://codeberg.org/hatchjaw/aes-faust-ddsp

[5] https://faustide.grame.fr/

off frequency, and the length of multiple hidden delay lines can be found by optimisation against a differentiable FIR filterbank; these examples point the way to a variety of real-time adaptive processing applications.

Though demonstrated with *known* algorithms, in principle the above can be extended to optimisation against a target *black box* filter, granted that both target and estimate receive the same input signal. This hints, however, at another limitation of the current approach — Faust is primarily concerned with time-domain operations, thus loss is computed in the time domain on a per-sample basis (albeit with the possibility of taking a running-average of recent sample values, for example). For phase-invariant and perceptually-informed optimisation scenarios, frequency-domain loss computation is required; the recent introduction of multi-rate processing to the Faust language promises to enable spectral signal analysis. Developments of this sort will improve the feasibility of taking on larger-scale, typically deferred-time problems such neural audio synthesis. These call for generalisation to a corpus of training data (rather than optimisation to a small number of input variables); with that end in mind, the example in Listing 4 should be extended to a general neural network structure, and a supporting infrastructure created for delivering training data and capturing weights and biases for real-time inference.

5 Conclusion

In this paper we have described a general scheme for differentiating Faust programs, moving an important step toward providing the means with which to tackle audio optimisation and machine learning problems with the domain-specific and cross-platform benefits of the Faust language. A variety of time-domain, gradient-based optimisation problems can be tackled in Faust, with results produced in real-time.

Many challenges and opportunities remain as impetus for further development of the work presented here. Having only been put to informal use to-date, the proposed AD approach should be assessed for its efficacy and efficiency. Beyond the correctness of the semantic scheme, being based on forward-mode AD, which, on paper, compares unfavourably with reverse-mode in terms of computational expense, its combination with the optimisations of the Faust compiler should be explored. At the level of the Faust compiler, changes that would facilitate the transformation of elements of the language that currently elude pattern matching would be of benefit here, as would further development of Faust's adaptability to frequency domain processing.

References

[1] Baydin, A. G. et al. "Automatic Differentiation in Machine Learning: A Survey". In: *Journal of Machine Learning Research* 18 (2018).

[2] Hayes, B. et al. "A Review of Differentiable Digital Signal Processing for Music and Speech Synthesis". In: *Frontiers in Signal Processing* 3 (Jan. 11, 2024).

[3] Alexandre, L., Lecomte, P., and Galland, M.-A. "Experimental Active Noise Control Using Faust On FPGA: Comparison Between A Multi-Point and Spherical Harmonics Method". In: *Forum Acusticum 2023 - 10th Convention of the European Acoustics Association*. Torino, Italy, Sept. 2023.

[4] Engel, J. et al. "Differentiable Digital Signal Processing". In: *Proceedings of the Eighth International Conference on Learning Representations*. Online, 2020.

[5] Rushton, T. A. "Differentiable DSP in Faust". In: *Proceedings of the International Faust Conference (IFC-24)*. Turin, Italy, Nov. 2024.

[6] Orlarey, Y., Fober, D., and Letz, S. "FAUST: An Efficient Functional Approach to DSP Programming". In: *New computational paradigms for computer music* (2009).

[7] Orlarey, Y., Fober, D., and Letz, S. "An Algebra for Block Diagram Languages". In: *Proceedings of International Computer Music Conference*. Gothenburg, Sweden, 2002.

[8] Orlarey, Y., Fober, D., and Letz, S. "Syntactical and Semantical Aspects of Faust". In: *Soft Computing* 8.9 (2004).

[9] Orlarey, Y. et al. "A New Intermediate Representation for Compiling and Optimizing Faust Code". In: *Proceedings of the 2nd International Faust Conference (IFC-20)*. Paris, France, Dec. 1, 2020.

[10] Letz, S., Orlarey, Y., and Fober, D. "An Overview of the FAUST Developer Ecosystem". In: *Proceedings of the 1st International Faust Conference (IFC-18)*. Mainz, Germany, July 17, 2018.

[11] Letz, S., Orlarey, Y., and Fober, D. “Compiling Faust Audio DSP Code to WebAssembly”. In: *Web Audio Conference*. London, United Kingdom, Aug. 2017.

[12] Gaster, B. R. and Cole, M. “Audio Anywhere with Faust”. In: *Proceedings of the 2nd International Faust Conference (IFC-20)*. Paris, France, 2020.

[13] Albouy, A. and Letz, S. “Faust Audio DSP Language for JUCE”. In: *Proceedings of the Linux Audio Conference*. Saint-Etienne, France, May 18, 2017.

[14] Michon, R. et al. “Embedded Real-Time Audio Signal Processing With Faust”. In: *Proceedings of the 2nd International Faust Conference (IFC-20)*. Paris, France, Dec. 2020.

[15] Cochard, P. et al. “Programming FPGA Platforms for Real-Time Audio Signal Processing In C++”. In: *Proceedings of the 2024 Sound and Music Computing Conference (SMC-24)*. Porto, Portugal, July 2024.

[16] Michon, R. et al. “The Faust Programming Language As a Platform for Creating Hybrid Acoustical and Digital Musical Instruments”. In: *Forum Acusticum 2020 Proceedings*. Lyon, France, Dec. 2020.

[17] Caren, M., Michon, R., and Wright, M. “The KeyWI: An Expressive and Accessible Electronic Wind Instrument”. In: *Proceedings of The International Conference on New Interfaces for Musical Expression (NIME ’20)*. Birmingham, UK, July 21, 2020.

[18] Hansen, M. “Weather Organ: A Sparse Stochastic Synthesizer in Faust”. In: *Proceedings of the 1st International Faust Conference (IFC-18)*. Mainz, Germany, July 17, 2018.

[19] Roosenburg, D. and Michon, R. “A Wave Digital Filter Modeling Library for the Faust Programming Language”. In: *Proceedings of the 18th Sound and Music Computing Conference*. Online, June 29, 2021.

[20] Russo, R. et al. “Introducing Finite Difference Schemes Synthesis in Faust: A Cellular Automata Approach”. In: *Proceedings of the 18th Sound and Music Computing Conference*. Online, June 29, 2021.

[21] Michon, R. et al. “The Faust Physical Modeling Library: A Modular Playground for the Digital Luthier”. In: *Proceedings of the 1st International Faust Conference (IFC-18)*. Mainz, Germany, 2018.

[22] Braun, D. *DawDreamer: Bridging the Gap Between Digital Audio Workstations and Python Interfaces*. Nov. 18, 2021. arXiv: 2111.09931. Pre-published.

[23] Rall, L. B. “The Arithmetic of Differentiation”. In: *Mathematics Magazine* 59.5 (Dec. 1986).

[24] Harrison, D. *A Brief Introduction to Automatic Differentiation for Machine Learning*. Oct. 14, 2021. arXiv: 2110.06209. Pre-published.

[25] Wang, F. et al. “Demystifying Differentiable Programming: Shift/Reset the Penultimate Backpropagator”. In: *Proceedings of the ACM on Programming Languages* 3 (ICFP July 26, 2019).

[26] Yu, W. and Blair, M. “DNAD, a Simple Tool for Automatic Differentiation of Fortran Codes Using Dual Numbers”. In: *Computer Physics Communications* 184.5 (May 2013).

[27] Li, Y. and Wang, G. “ChAI => Interactive AI Tools in ChucK”. In: *The International Conference on New Interfaces for Musical Expression (NIME ’24)*. Utrecht, The Netherlands, Sept. 4, 2024.

[28] Kuznetsov, B., Parker, J. D., and Esqueda, F. “Differentiable IIR Filters for Machine Learning Applications”. In: *Proceedings of the 23rd International Conference on Digital Audio Effects (DAFx-20)*. Vienna, Austria, 2020.

[29] McClellan, J. H., Schafer, R. W., and Yoder, M. A. *Signal Processing First*. Pearson, 2003.

[30] Shynk, J. J. “Adaptive IIR Filtering”. In: *IEEE ASSP Magazine* 6.2 (Apr. 1989).

[31] Smith, J. O. *Introduction to Digital Filters: With Audio Applications*. W3K Publishing, 2008.