# Differentiable DSP in Faust

Thomas Albert Rushton

# DIFFERENTIABLE DSP IN FAUST

*Thomas Albert Rushton*

Inria, INSA Lyon, CITI, EA3720
69621 Villeurbanne, France
thomas.rushton@inria.fr

## ABSTRACT

Differentiable Digital Signal Processing is the application of differentiable programming, whereby a computer program may be differentiated end-to-end, to audio tasks. Coupled with gradient-based optimisation methods, differentiable signal processors are central to a variety of audio problems and can be incorporated into machine learning architectures.

In this paper it is shown that, using the `environment` expression and pattern matching abstraction, it is possible to write FAUST code that is differentiable end-to-end. A system for writing FAUST programs that are automatically differentiable in the forward-mode is developed and a parameter optimisation example presented. Differentiable programming in FAUST could serve as a platform for native approaches to machine learning problems in the audio domain.

## 1. INTRODUCTION

Differentiable programming is a programming paradigm whereby a computer program can be differentiated end-to-end [1]. The sensitivity of a differentiable program's outputs to perturbations of its parameters can be computed via automatic differentiation (AD, autodiff) [2, 3], producing a partial derivative with respect to each input parameter. End-to-end differentiability is a desirable quality in the creation of computer programs that perform gradient-based optimisation, and differentiable programming via automatic differentiation is the foundation for contemporary approaches to machine learning [4].

Differentiable Digital Signal Processing (DDSP) is the application of differentiable programming to DSP operations [5]. The acronym *DDSP* was coined by Engel et al. [6], who used it to refer to the specific case of combining differentiable signal processors with a neural network architecture, but in principle any DSP system featuring recursive optimisation using gradients found as partial derivatives of a loss function fits this label [5], including work dating as far back as the late 1980's [7]. In addition to Engel et al.'s timbre transfer implementation via a differentiable spectral modelling synthesiser, DDSP has been applied to audio tasks such as source separation [8], filter optimisation [9], and echo cancellation [10] — see [5] for a comprehensive review.

This paper introduces the concept of a *differentiation arithmetic* [2] to FAUST, facilitating the creation of differentiable audio algorithms in the FAUST language. A framework for forward mode automatic differentiation is outlined and applied to a simple, but illustrative, parameter optimisation problem. The possibility of writing differentiable code in an audio domain specific language paves the way for novel approaches to problems at the intersection of DSP and machine learning.

## 2. ALGORITHMIC DIFFERENTIATION

Methods for computational differentiation are typically characterised as falling into one of three camps: *numerical*, *symbolic*, and *automatic*. Numerical differentiation produces numerical values for derivatives via approximation by finite differences, and will be familiar to those acquainted with finite-difference time-domain audio synthesis methods [11]. Symbolic differentiation takes a computational expression and generates the corresponding expression for its derivative; this approach may resonate with users of MATLAB's Symbolic Math Toolbox [12] or the Maple programming language [13]. Automatic differentiation describes an arithmetic for accumulating both the numerical output of a computational expression and the numerical value of its derivative; it is an arithmetic of this kind that underpins the current crop of Python libraries for machine learning [4].

A certain ambiguity abounds with regard to how automatic and symbolic differentiation relate to each other [14], and partisan views have been expressed over which is more efficient [15]. The ambiguity may be ascribed in part to the former's nature as "partly symbolic and partly numerical" [4], and perhaps also to the fact that programs composed symbolically may be differentiated automatically [16]. For our purposes, we shall defer to Rall, who, writing in the mid-1980's, before the waters were muddied by legions of machine learning researchers, observed (to paraphrase): *symbolic approaches produce formulas whereas automatic approaches produce numerical outputs* [2]. The latter do so, however, by implementing differentiation rules, symbolically, at the level, as we will see, of the primitive operations of the programming language upon which they are based.

### 2.1. The Arithmetic of Automatic Differentiation

Two principal *modes* of automatic differentiation are alluded to in scholarly works on the topic: forward (or *tangent*) mode, and reverse (or *adjoint*) mode [1, 3, 4]. These modes describe, in effect, two directions of derivative propagation through a computation graph; in forward mode, the computation of the undifferentiated, or *primal* output is accompanied by a tangent computation, with derivatives accumulated from inputs to outputs; in reverse mode, a forward primal pass is complemented by a reverse adjoint pass, during which derivatives accumulate from outputs to inputs (consult [3] for a detailed mathematical treatment of both). For a graph, $f : \mathbb{R}^N \to \mathbb{R}^M$, with input variables, $x_i$, and output variables, $y_j$, forward mode requires $N$ passes to compute the Jacobian matrix

$$\mathbf{J}_f = \begin{bmatrix} \frac{\partial y_1}{\partial x_1} & \dots & \frac{\partial y_1}{\partial x_N} \\ \vdots & \ddots & \vdots \\ \frac{\partial y_M}{\partial x_1} & \dots & \frac{\partial y_M}{\partial x_N} \end{bmatrix}, \tag{1}$$

whereas reverse mode calls for $M$ passes. Viewed through the lens of computational parsimony, this points at reverse mode being preferable for differentiating programs where input variables outnumber output variables, such as a digital audio synthesiser with many parameters and perhaps one or two output channels. In practice, reverse mode demands a bookkeeping strategy such as a *"tape"* mechanism [17, 18] to take account of the dependencies of each node in the graph during the forward pass, and thus ensure accurate derivative accumulation during the reverse pass. This need for a structured overview of the computational graph introduces a degree of complexity that forward mode does not impose.

#### *2.1.1. Dual Number Arithmetic*

An interesting property of forward mode automatic differentiation lies in the possibility of formulating differentiation in a manner similar to the complex numbers [2, 19]. Where complex arithmetic uses the imaginary unit $i$ to designate the imaginary part of a complex number $z = x + iy$, with $i^2 = -1$, differentiation arithmetic uses the *nilpotent symbol*, $\varepsilon$, for which $\varepsilon^2 = 0$, and $\varepsilon \neq 0$ [19, 3]:

$$U = u + \varepsilon u' \ , \tag{2}$$

where $u' = \frac{du}{dx}$ is the derivative of $u$ with respect to some input variable $x$.

Using this arithmetic, the sum of two functions, *and* the addition rule of differentiation, emerge quite naturally as the sum of $U$ and $V = v + \varepsilon v'$,

$$(u + \varepsilon u') + (v + \varepsilon v') = u + v + \varepsilon(u' + v') \ . \tag{3}$$

Similarly, the product rule, via simple polynomial expansion

$$\begin{aligned}(u + \varepsilon u')(v + \varepsilon v') &= uv + u\varepsilon v' + v\varepsilon u' + \varepsilon^2 u'v' \\ &= uv + \varepsilon(uv' + vu') \ ,\end{aligned} \tag{4}$$

the final term cancelling due to the presence of $\varepsilon^2$.

This arithmetic may be more conveniently expressed, for computational purposes, as one of *ordered pairs* [2, 20] or *dual numbers* [19, 21, 22],

$$U = \langle u, u' \rangle \ . \tag{5}$$

The addition and product rules now take the following forms:

$$U + V = \langle u, u' \rangle + \langle v, v' \rangle = \langle u + v, u' + v' \rangle \tag{6}$$

$$UV = \langle u, u' \rangle \langle v, v' \rangle = \langle uv, uv' + vu' \rangle \ . \tag{7}$$

The first component of each dual number is the rule for evaluation of the operation, the second is the rule for differentiation [2]; these are the primal and tangent respectively [3, 18]. In dual number differentiation arithmetic, an independent variable can be expressed as $X = \langle x, \frac{dx}{dx} \rangle = \langle x, 1 \rangle$, and a constant $C = \langle c, \frac{dc}{dx} \rangle = \langle c, 0 \rangle$. If we wish, for example, to compute the numerical values for the primal and tangent of a polynomial $(x + 1)(x - 2)$ at $x = 2$, we set $X = \langle 2, 1 \rangle$ and supply appropriate values for the constants:

$$(\langle 2, 1 \rangle + \langle 1, 0 \rangle)(\langle 2, 1 \rangle - \langle 2, 0 \rangle) = \langle 3, 1 \rangle \langle 0, 1 \rangle = \langle 0, 3 \rangle \ .$$

We find that our arithmetic produces the expected numerical results *automatically* via composition of the fundamental operations characterised by equations (6) and (7).

Differentiation arithmetic of this sort is a special case of a more general *gradient arithmetic* [2], which comes into effect when multiple variables are present, and thus multiple partial derivatives must be calculated:

$$U = \langle u, \nabla u \rangle, \quad \nabla u = \frac{\partial u}{\partial \mathbf{x}} = \begin{bmatrix} \frac{\partial u}{\partial x_1} \\ \vdots \\ \frac{\partial u}{\partial x_N} \end{bmatrix} \ . \tag{8}$$

### 2.2. Extending FAUST's Arithmetic

One quality generally possessed by automatic differentiation implementations is that of allowing the programmer to write differentiable programs with minimal changes to the syntax of their code. This is typically achieved either by source code transformation or operator overloading [19, 3] — neither of which is available in the FAUST language.[1] We could take an approach similar to Wengert's 1964 demonstration of the composition of differentiable functions in Fortran [23] and define differentiable functions `diffAdd`, `diffMul`, etc. each accepting and returning dual numbers; thanks to FAUST's pattern matching abstraction, however, we can go one better, achieving something akin to, albeit slightly more verbose than, operator overloading.

Pattern matching has been used extensively in the FAUST libraries. The `basics.lib` library, for example, provides a function with a recursive pattern matching definition for taking an element from a list:

```
// Take the first element, the head;
take(1, (head, rest)) = head;
// Take the only element;
take(1, head)         = head;
// Take the n-1th element from the rest.
take(n, (head, rest)) = take(n-1, rest);
```

Listing 1: *Definition of the* `take` *function from* FAUST*'s* `basics.lib` *library.*

The `physmodels.lib` library provides `pm.chain` for creating chains of bidirectional signal blocks [24]. `chain(A)` simply returns the signal block `A`; `chain(A:As)` creates a recursive structure containing `A` and `chain(As)`. Similarly, `wdmodels.lib` facilitates the creation of wave digital filter models via primitive elements, resistors, capacitors, etc. [25], whose behaviour is defined via pattern matching syntax. Both libraries extend FAUST's arithmetic with their own rules, with the aim of achieving a particular goal within the syntax provided by FAUST.

If one's particular goal was to be able to compose reciprocal expressions, one could create an arithmetic of the following form, using pattern matching to avoid division by zero:

```
import("stdfaust.lib");
recip(0) = recip(ma.EPSILON);
recip(0.0) = recip(0);
recip(expr) = 1,expr : /;
a = log(1);
b = _,2 : ^;
c = -;
```

[1] Transformations *are* in fact possible at the level of the FAUST compiler, and automatic differentiation could occur as a compilation step. For reasons of scope, this paper focuses solely on the topic of automatic differentiation in the FAUST language itself.

```
process = recip(a),recip(b) : recip(c);
```

Listing 2: *Definition of a simple scheme for computing automatic reciprocals in* FAUST *via pattern-matching.*

Wrapping our expressions in `recip` gives us automatic reciprocals without affecting the fundamental composability of FAUST's primitives. Note that if we were to insert the expression `recip(-) = +;` at the top of the program, the final operation, rather than returning $1/(a-b)$ would be *overridden*, in a manner of speaking, returning $a+b$ instead. An approach along these lines will form the basis for the creation of differentiable FAUST primitives.

## 3. DIFFERENTIABLE PROGRAMMING IN FAUST

As described in section 2.1, in forward mode, primal and tangent outputs are found during a forward pass through the computation graph, which fits neatly with the left-to-right propagation of signals through a FAUST block diagram. Reverse propagation of signals is entirely possible in FAUST (indeed forward mode demands it as a final, backpropagation step — see section 3.3), and it is by way of nested recursive composition that the `physmodels.lib` library accomplishes simulated bidirectional wave propagation; the structures underpinning `physmodels.lib` are purely linear, however, and, at the time of writing, no general scheme for the creation of *branching* bidirectional structures, such as reverse mode requires, has been found.[2]

Consequently, this section is concerned with the description of an approach to differentiable programming in FAUST based on forward mode automatic differentiation. End-to-end differentiability is predicated on the availability of derivative expressions for the primitive operations of the language; presented in the subsections that follow is an approach to defining FAUST primitives that are differentiable in forward mode.

### 3.1. Defining a Differentiable Primitive

As a basic starting point, consider the addition primitive; in FAUST one can write:

```
process = +;
```

which yields the diagrammatic representation:

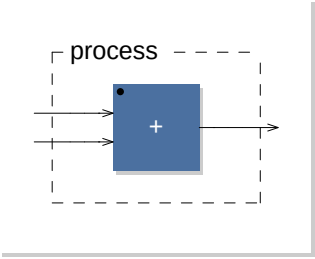


Figure 1: *Block diagram of a* FAUST *program consisting of a lone addition primitive.*

[2]Reverse mode does not entirely elude the capabilities of FAUST, but it does not generalise easily. For a small reverse mode example, see `https://gist.github.com/hatchjaw/8b3eb17aae27e91d0927ac8cb3eba9cd#file-reverse_multivariate-dsp`.

FAUST primitives, and block diagrams constructed from them, are *signal processors*. A semantic distinction is drawn, however, between a block diagram, $D$, and the signal processor represented by that block diagram, notated $[\![D]\!]$ [26]. We can think of $D$ as a symbolic expression, in FAUST syntax, and $[\![D]\!]$ as a processor that acts upon a vector of input signals and, in turn, produces a vector of output signals. A signal is a discrete function of time, and a member of the set $\mathbb{S}$ of all signals; the value of a signal at time $n$ is analogous to a numerical output. The semantic scheme for the addition operator is described as [26]

$$\begin{aligned} [\![+]\!] &: \mathbb{S}^2 \to \mathbb{S} \\ [\![+]\!](s_1, s_2) &= (y) \\ y[n] &= s_1[n] + s_2[n] \ . \end{aligned} \tag{9}$$

Note that FAUST's addition primitive has no special knowledge of its arguments, their history, provenance, etc., it just consumes them and returns their sum. In FAUST's arithmetic, the addition of two signals is simply well-defined.

Suppose that the block diagram, $Y = +$, is dependent on some variable $x$, and that we wish to know how sensitive $Y$ is to perturbations in $x$. We can produce an analytic expression for this sensitivity by differentiating $Y$ with respect to $x$. Recall, from equation (6), that a dual-number addition takes the form of two additions in parallel; in FAUST, that could be expressed as:

```
diffAdd = +,+;
process = diffAdd;
```

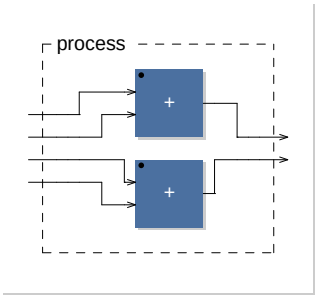


Figure 2: *Block diagram of a* FAUST *program representing a naive implementation of a dual signal differentiable addition primitive, consisting of two parallel additions.*

We can think of this as consisting of two block diagrams in parallel; interpreted as a signal processor, we can refer to its output as a *dual signal*, $\langle y, y' \rangle$.

Just as the addition primitive has no special knowledge of its input signals, nor does `diffAdd`, but at this stage the notion of differentiable addition is not well-defined. In order for this new primitive to behave as it should, it is necessary to define an accompanying semantic scheme. First, we denote $\mathbb{S}_d$ to be the set of all dual signals: $\mathbb{S}_d = \mathbb{S}^2$; differentiable addition can then be defined as

$$\begin{aligned} [\![\texttt{diffAdd}]\!] &: \mathbb{S}_d^2 \to \mathbb{S}_d \\ [\![\texttt{diffAdd}]\!](\langle s_1, s_1' \rangle, \langle s_2, s_2' \rangle) &= (\langle y, y' \rangle) \\ \langle y[n], y'[n] \rangle &= \langle s_1[n] + s_2[n], s_1'[n] + s_2'[n] \rangle \ . \end{aligned} \tag{10}$$

In its form in figure 2, `diffAdd` is not consistent with the scheme that we have just defined; this can be remedied with FAUST's `route` primitive:

```
diffAdd = route(4, 4,
    (1, 1), (2, 3), (3, 2), (4, 4)) : +,+;
process = diffAdd;
```

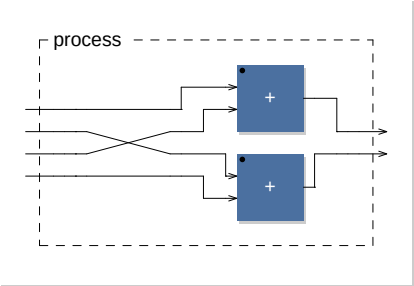


Figure 3: *Block diagram of a* FAUST *program representing a dual signal differentiable addition primitive. Via appropriate signal routing, valid dual signal output is produced.*

`diffAdd` is now semantically sound, implementing equation (6), and describing a dual-signal differentiable addition primitive. As alluded to in section 2, this primitive implements its differentiation rule symbolically, via appropriate signal routing, and will produce the correct numerical output automatically; it is self-contained, well-defined, and composable with other similarly well-defined primitives.

### 3.2. Multivariate Differentiable Primitives

The above holds for single-variable differentiation arithmetic, but what if a program features more than one dependent variable? Consider the following (non-differentiable) example consisting of a DC offset and a gain control applied to an input signal:

```
x1 = hslider("gain", .5, 0, 1, .1);
x2 = hslider("dc", 0, -1, 1, .1);
process = _,x1 : *,x2 : +;
```

Listing 3: *A* FAUST *program that applies gain and DC offset parameters to an input signal.*

The general case of gradient arithmetic (see equation (8)) demands a redefinition of the set of dual signals, $\mathbb{S}_d = \mathbb{S}^{N+1}$, where $N$ is the number of variables, $x_i$, with respect to which partial derivatives must be found. One way to implement a multivariate differentiable addition primitive in FAUST could be to define `diffAdd` as a function receiving $N$ as a parameter; using FAUST's `environment` expression, however it is possible to address the problem in a more general, and syntactically succinct fashion.

#### *3.2.1. A Differentiable Environment*

First, we can define a function for collecting, counting, and retrieving variables and their partial derivatives:

```
vars(V) = environment {
  // Count the variables.
  N = outputs(V);
  // Retrieve a variable by index i.
  var(i) = ba.take(i, V),pds(N, i)
  with {
    // Compute partial derivatives of
    // variable x_i.
    pds(N, i) = par(j, N, i-1==j);
  };
};
```

Listing 4: *A* FAUST *function for defining an environment of differentiable variables.*

`vars` receives a list of variables, expressed via parallel composition, e.g. `X = vars((gain,dc));`, where `gain` and `dc` are defined as `hslider` instances; the $i^{\text{th}}$ differentiable variable is defined semantically as

$$\begin{aligned} [\![\texttt{X.var}(i)]\!] &: \mathbb{S}^0 \to \mathbb{S}_d \\ [\![\texttt{X.var}(i)]\!]() &= (\langle y, \nabla y \rangle) \\ \langle y[n], \nabla y[n] \rangle &= \langle x_i[n], \nabla x_i[n] \rangle \\ &= \left\langle x_i[n], \left[0 \cdots 1 \cdots 0\right]^T \right\rangle . \end{aligned} \tag{11}$$

Next, we can define a function that takes the variable environment produced by `vars` as its sole argument, and returns a *differentiable environment*, containing a collection of multivariate differentiable primitives. As a further improvement, we can use FAUST's pattern matching syntax to simplify the nomenclature of the differentiable primitives; instead of exposing the name `diffAdd`, for example, the differentiable addition primitive can be named `diff(+)`:

```
env(vars) = environment {
  diff(+) = diffAdd with {
    diffAdd = route(nIN, nOUT,
      (s1, 1), (s2, 2), // s1 + s2
      par(i, vars.N,
        // ds1/dx_i + ds2/dx_i
        (s1+i+1, dx), (s2+i+1, dx+1)
        with {
          // Start of derivatives wrt x_i
          dx = 2*i+3;
        }
      )
    ) with {
      nIN = 2+2*vars.N;
      nOUT = nIN;
      s1 = 1;
      s2 = s1+vars.N+1;
    }
    : +,par(i, vars.N, +);
  };

  // ...definitions of other
  // differentiable primitives...
};
```

Listing 5: *Extract from the definition of a* FAUST *environment for differentiable programming.*

As before, the primal signal output of the differentiable addition primitive is the sum of the primal inputs, $s_1[n] + s_2[n]$; now, however, the differentiable primitive produces `vars.N` tangent outputs, each corresponding to a derivative with respect to $x_i$. Once again,

the `route` primitive ensures that incoming signals are delivered to the parallel additions in the correct order.

Encapsulating listings 4 and 5 as a library in a file named `diff.lib`, and defining a differentiable audio input, which, since it does not depend on $x_i$, has the semantic representation

$$\begin{aligned} [\![\texttt{input}]\!] &: \mathbb{S} \to \mathbb{S}_d \\ [\![\texttt{input}]\!](s) &= (\langle y, \nabla y \rangle) \\ \langle y[n], \nabla y[n] \rangle &= \langle s[n], \mathbf{0} \rangle \ , \end{aligned} \tag{12}$$

we can use `vars` and `env` to write a differentiable version of the gain-plus-DC-offset program encountered earlier:

```
df = library("diff.lib");

X = df.vars((gain,dc)) with {
  gain = hslider("gain", .5, 0, 1, .01);
  dc = hslider("dc", 0, -1, 1, .01);
};

d = df.env(X);

process = d.input,X.var(1)
  : d.diff(*),X.var(2)
  : d.diff(+);
```

Listing 6: *Differentiable counterpart to the* FAUST *program described in listing 3.*

See figure 4 for the block diagram of this program. Note that while the routing for the arithmetic primitives — particularly `diff(*)` — may be quite complex (and would only become more involved with the addition of further variables) the `df` library abstracts this complexity away. Note also that partial derivatives of the program are found automatically via application of the chain rule of differentiation, imposed by the definition of semantically-consistent dual-signal primitives.

### 3.3. Parameter Optimisation via Gradient Descent

Armed with the means to write end-to-end differentiable FAUST programs, it is possible, with a few modifications (and additions to `diff.lib`), to combine the code in listings 3 and 6, to create a demonstrative parameter optimisation algorithm. An algorithm of this kind consists of a target output, governed by parameters that are *hidden* with respect to some *estimated* output, which itself depends on parameters that we wish to optimise.

The algorithm in listing 3 is dependent on hidden parameters $\mathbf{x}$ and produces a ground truth output signal $y[n]$; we assign this algorithm to a variable named `target`. Its differentiable equivalent in listing 6 is dependent on estimated parameters $\hat{\mathbf{x}}$ and produces the dual output signal, $\langle \hat{y}[n], \nabla \hat{y}[n] \rangle$; we assign this to a variable named `estimate`. The output signal produced by `target` and the primal output signal of `estimate` can now be compared by way of a loss function; to this end, we can employ time-domain L1-norm loss of the form

$$\mathcal{L}(y, \hat{y})[n] = ||\hat{y}[n] - y[n]|| \ . \tag{13}$$

Our aim is to minimise the value returned by the loss function, i.e. to reach the point at which $y[n]$ and $\hat{y}[n]$ (and by extension $\mathbf{x}$ and $\hat{\mathbf{x}}$) most closely approximate one-another. The sensitivity of $\mathcal{L}$ to perturbations in $\hat{\mathbf{x}}$ can again be found by automatic differentiation, subject to the provision of a differentiable absolute value function in `df.env`, with the following semantic definition:

$$\begin{aligned} [\![\texttt{diff(abs)}]\!] &: \mathbb{S}_d \to \mathbb{S}_d \\ [\![\texttt{diff(abs)}]\!](\langle s, \nabla s \rangle) &= (\langle y, \nabla y \rangle) \\ \langle y[n], \nabla y[n] \rangle &= \left\langle |s[n]|, \frac{s[n] \nabla s[n]}{|s[n]|} \right\rangle \ . \end{aligned} \tag{14}$$

The loss function can then be implemented as follows:

```
env(vars) = environment {
  // ...
  lossL1(learningRate, y, yHat) =
    error,par(i, vars.N, _)
    : diff(abs)
    : _,scaleGrads
  with {
    error = yHat,y : -;
    scaleGrads = par(i, vars.N,
      _,learningRate : *);
  };
  // ...
```

Listing 7: *Implementation of a differentiable loss function (L1-norm) using differentiable* `abs` *primitive.*

The loss function's partial derivatives are the *gradients* associated with each variable in $\hat{\mathbf{x}}$. In our two-parameter example, $\mathcal{L}$ will describe a three-dimensional surface, and $\frac{\partial \mathcal{L}}{\partial x_i}$ its slope relative to $x_i$. Values at time $n+1$ are found by scaling gradients by a learning rate, $\alpha$, and subtracting the result from values at time $n$:

$$\hat{\mathbf{x}}[n+1] = \hat{\mathbf{x}}[n] - \alpha \frac{\partial \mathcal{L}}{\partial \hat{\mathbf{x}}}[n] \ . \tag{15}$$

In FAUST, this can be achieved via recursion, and by changing the definition of the variables delivered to `df.env`. Since values will be updated automatically via gradient descent, the sliders used in listing 6 are no longer appropriate; instead, a bargraph instance can be used to display the value of each variable, with a recursive subtraction accumulating each parameter's incoming scaled gradient.

```
// diff.lib
var(meter) = -~_ <: attach(meter);

// gain_dc_AD.dsp
X = df.vars((gain,dc)) with {
  gain = df.var(hbargraph("Gain", 0, 1));
  dc = df.var(hbargraph("DC", -1, 1));
};
```

Listing 8: *Definition of differentiable variables encapsulating recursive gradient descent.*

Finally, we can create a program that takes white noise as input, encapsulates `target`, `estimate`, and the loss function, and recurses gradients produced by the latter back to `estimate`.

```
process = no.noise <: (
    route(nvars+nInputs, nvars+nInputs,
      // Route gradients to estimate.
      par(n, nvars, (n+1, n+1+nInputs)),
```

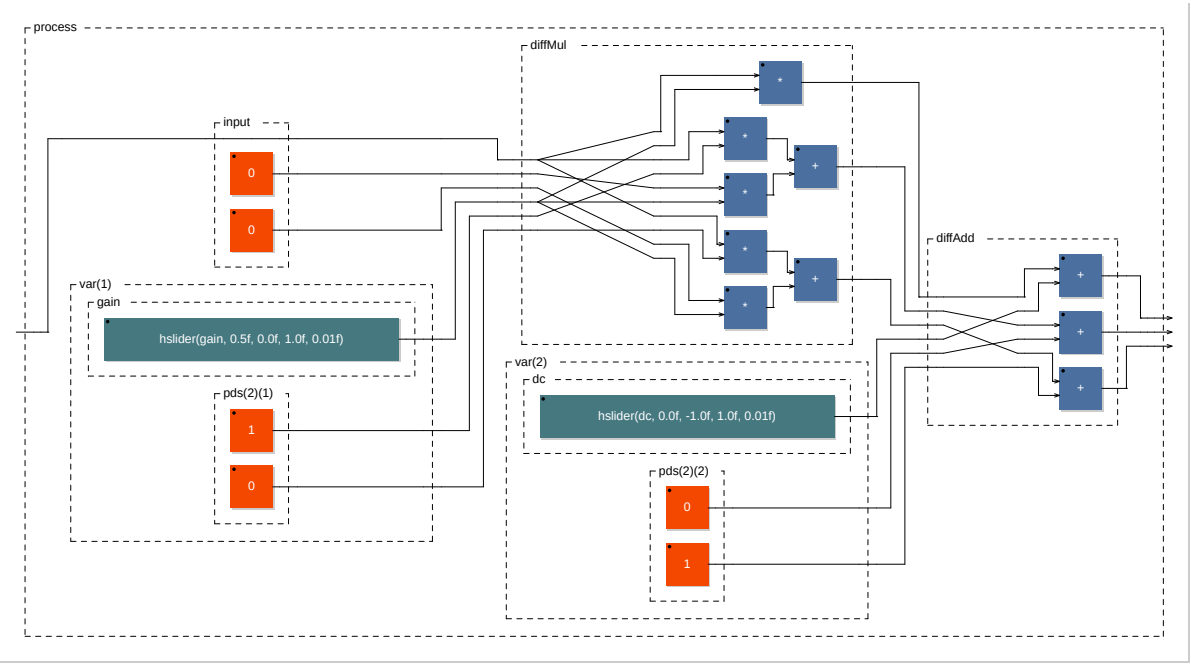


Figure 4: *Block diagram of a differentiable* FAUST *program consisting of differentiable gain and DC parameters applied to an input signal (see listing 6).*

```
      // Route input to target & estimate.
      par(n, nInputs, (nvars+1+n, n+1))
    )
    : target,estimate : d.lossL1(1e-3)
  // Recurse the gradients.
  ) ~ (!,si.bus(nvars))
  // Block the gradients, post-recursion.
  : _,si.block(nvars)
with {
  nvars = inputs(estimate),
    inputs(target) : -;
  nInputs = inputs(target),2 : *;
  // ...
```

Listing 9: *Excerpt from a parameter optimisation algorithm. See listings 3, 6 and 7 respectively for definitions of* `target`, `estimate`, *and* `d.lossL1`.

This delivery of gradients from the outputs of the program back to its inputs is commonly (and particularly in material from the field of machine learning) referred to as *backpropagation* [3, 10, 27]. Whereas in reverse mode gradients arrive at the inputs inevitably as a consequence of the reverse adjoint pass through the graph, in forward mode a recursion such as that described in listing 9 is required; consult figure 5 for the corresponding top-level block diagram.

Running this program[3] reveals a user interface which includes slider elements for the parameters of the `target` algorithm, and bargraphs that report the values of the parameters of `estimate`. Moving a slider results in an increase in the value returned by the loss function, which is then minimised via gradient descent, the estimated parameter values tracking the values of the target.

[3] A full code example, adapted from the excerpts in this paper, can be found at `https://gist.github.com/hatchjaw/59f35d0cde7aba218d785d31f26d2d83`.

## 4. DISCUSSION

The previous section presented an approach to differentiable programming in FAUST, but the scheme under consideration is not free from disadvantages. Considered in the following subsections are some limitations of the suggested automatic differentiation strategy, plus a selection of ideas for future development.

### 4.1. Primitives With Poorly-Defined Derivatives

To provide comprehensive support for differentiable programming, the formative library presented here should of course comprise differentiable equivalents to all of FAUST's primitives. That would include, however, the likes of `floor` and `ceil`, whose primal outputs are discontinuous. Indeed, in implementation, the differentiable `abs` function used in the loss function in listing 7 takes the liberty of avoiding division by zero by dividing by whichever of $|s[n]|$ and `ma.EPSILON` is greater; it may prove preferable to replace `abs`, and similarly problematic functions, with smooth approximations [28].

Another class of FAUST primitives not addressed here are those relating to delays. As demonstrated by Shynk [7], IIR filters, and thus fixed delays (including recursive delays), are differentiable in terms of their coefficients; whether a variable delay (FAUST's `@` primitive) is differentiable with respect to the length of the delay line, stands as a topic for future research.

### 4.2. Frequency-Domain Loss

The parameter optimisation example given in section 3.3 works, but with a couple of significant caveats, the first of these being that the

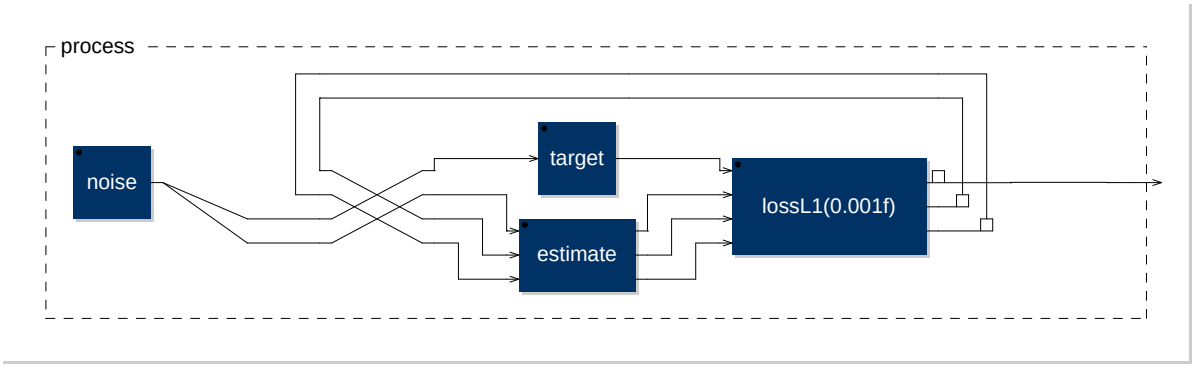


Figure 5: *Top-level block diagram for the full* FAUST *program of which an excerpt appears in listing 9.* `target` *algorithm, with* hidden *parameters, and* `estimate` *algorithm, with two* optimisable *parameters, process a noise signal. Their outputs are compared via a loss function; scaled derivatives of the loss function are backpropagated such that estimated parameter values can be updated. Note that the only output signal produced by this example program is the primal output signal produced by the loss function; to hear the primal signal produced by* `estimate`*, one could perform additional signal routing prior to the loss function.*

`target` and `estimate` algorithms receive identical signals as input, and the second being that loss is calculated sample-by-sample in the time domain. If decorrelated noise signals were used instead, it is vanishingly unlikely that good parameter estimates would be found. Some improvement may be achieved by comparing $y$ and $\hat{y}$ after a short-duration application of `ba.slidingMean`, but not if oscillators of different frequencies, or unaligned phase, were employed instead.

One way to combat problems of this sort would be to calculate loss in the frequency domain; indeed it is typically by way of perceptually-informed *spectral loss* that optimisation is conducted in a DDSP setting [5]. Various functions exist in FAUST's `analyzers.lib` library that could be used to this end; being based on an FFT implementation that is restricted, however, to a single sample hop-size, at the time of writing computational expense places limits on the calculation of magnitude spectrograms, particularly in a real-time setting.

### 4.3. Computational Efficiency

On that note, and as alluded to in section 2.1, forward mode is not, on paper at least, the most efficient choice for automatically differentiating programs with more inputs than outputs. Figure 4 shows two sets of tangent calculations accompanying each primitive's primal operation, and a number of zero signal paths (partial derivatives of the input signal, for example). Not pictured in figure 4, the most egregious proliferation of zeros is caused by differentiable numerical constants; a constant $c$, is, in dual-signal form, $\langle c, \nabla c\rangle = \langle c, \mathbf{0}\rangle$, or in FAUST:

```
diff(c) = c,par(i, vars.N, 0);
```

Of course, a constant may well be followed, for instance, by a trigonometric function — there is no guarantee that $\nabla c$ will not contribute to a non-zero signal path, thus no scope for optimisation.

That being said, the FAUST compiler is designed with this sort of optimisation in mind, applying various rewriting rules after its symbolic propagation phase to simplify expressions and avoid redundancy [29]. In effect, the compiler will attempt to produce the most efficient possible FAUST *Imperative Representation* for any given FAUST program. Nevertheless, the creation of a generalisable approach to reverse mode should be explored, and this too would benefit from compile-time optimisations. Moreover, forward and reverse mode can be thought of as extremes on a continuum of derivative propagation options; a combination of these modes (dubbed *cross-country mode*), tailored to the structure of the program being differentiated, would be ideal, though finding the optimal ordering is deemed a challenge [30].

### 4.4. A General Pattern-Matching Syntax

Although the differentiable algorithm in listing 6 bears the same compositional structure as its undifferentiated sibling (listing 3),[4] the use of the differentiable environment, coupled with pattern matching, inevitably leads to the necessity of wrapping primitives in `d.diff(...)` notation. `d.input` too is an unsatisfactory solution to the problem of there being no simple way, syntactically speaking, of distinguishing an input signal from an identity function, which, since they have different derivatives, is a necessity.

Ideally, an approach similar to that taken in `physmodels.lib` (as described in section 2.2), whereby expressions are recursively *decomposed*, with a base case to handle the desired transformations, should be employed. In that instance, it would be possible to define `estimate` in listing 9 via syntax along the lines of:

```
estimate = forwardAD(target);
```

In addition to abstracting away calls to `diff()`, this could permit identifying input signals by counting the number of inputs to the circuit passed to `forwardAD`. Standing in the way of this idea, however, are limitations in FAUST's pattern matching system at the time of writing, the principal issue being the impossibility of pattern-matching user interface elements in the general case; i.e., to match a `hslider` one needs to match its label exactly, plus the values provided for `init`, `min`, `max` and `step`. By way of an alternative, a strategy based on FAUST's *widget modulation* syntax may help circumvent this problem.

[4] This is thanks to the quality of the parallel and sequential composition operators of having no arithmetical influence on the output of the program — they are analogous to application of the identity function. This is not the case for merge composition, which, having the effect of summation, would, in a comprehensive implementation, require a differentiable transformation.

### 4.5. Application to Machine Learning

Further to the caveats mentioned in 4.2, the success of the parameter optimisation example presented in this paper is contingent on the provision of deterministic input data; essentially the example constitutes an extreme example of *overfitting*, solving one specific problem, on one particular set of input data, very well, but possessing no capability to generalise. Nevertheless, it shares its basis, in the form of differentiable programming, with more sophisticated applications of mathematical optimisation, chief amongst these being machine learning.

Using differentiable FAUST primitives it is straightforward to create differentiable loss functions; activation functions and artificial neurons (the latter being based on simple linear algebra principles) could follow without much trouble. Neural network structures, which support the training of models capable of generalising to unseen input data, would require significant effort to implement in a composable, extensible fashion, but it is unlikely that they lie beyond the capabilities of the language.

## 5. CONCLUSION

In this paper, it has been shown that differentiable programming is possible in the FAUST language, and thus that FAUST can be used to tackle audio problems based on principles of mathematical optimisation. The presence of a comprehensive automatic differentiation framework in FAUST would lend the language to a multitude of DDSP problems and applications that currently lie unexplored by FAUST programmers; in turn, the ability to tackle such problems in a domain specific language could foster innovation in what is a vibrant research area.

Aims for further investigation should be the implementation of differentiation rules for all of the primitives of the language, the development of a less intrusive syntax for automatic differentiation transformations, and perhaps enhancements to pattern matching at the level of the FAUST compiler. An FFT implementation of greater efficiency, or a novel approach to perceptually-informed loss computation, would also be of great benefit.

## 6. REFERENCES


[1] Mathieu Blondel and Vincent Roulet, "The Elements of Differentiable Programming," Mar. 2024, arXiv:2403.14606.

[2] Louis B. Rall, "The Arithmetic of Differentiation," *Mathematics Magazine*, vol. 59, no. 5, pp. 275–282, Dec. 1986.

[3] Atılım Gunes Baydin, Barak A Pearlmutter, Alexey Andreyevich Radul, and Jeffrey Mark Siskind, "Automatic Differentiation in Machine Learning: A Survey," *Journal of Machine Learning Research*, vol. 18, pp. 1–43, 2018.

[4] Davan Harrison, "A Brief Introduction to Automatic Differentiation for Machine Learning," Oct. 2021, arXiv:2110.06209.

[5] Ben Hayes, Jordie Shier, György Fazekas, Andrew McPherson, and Charalampos Saitis, "A Review of Differentiable Digital Signal Processing for Music & Speech Synthesis," Aug. 2023, arXiv:2308.15422.

[6] Jesse Engel, Lamtharn Hantrakul, Chenjie Gu, and Adam Roberts, "Differentiable Digital Signal Processing," in *Proceedings of the Eighth International Conference on Learning Representations*, Online, 2020.

[7] John J. Shynk, "Adaptive IIR filtering," *IEEE ASSP Magazine*, vol. 6, no. 2, pp. 4–21, Apr. 1989.

[8] Kilian Schulze-Forster, Gaël Richard, Liam Kelley, Clement S. J. Doire, and Roland Badeau, "Unsupervised Music Source Separation Using Differentiable Parametric Source Models," *IEEE/ACM Transactions on Audio, Speech, and Language Processing*, vol. 31, pp. 1276–1289, 2023.

[9] Boris Kuznetsov, Julian D Parker, and Fabián Esqueda, "Differentiable IIR filters for machine learning applications," in *Proceedings of the 23rd International Conference on Digital Audio Effects (DAFx-20)*, Vienna, Austria, 2020.

[10] Jonah Casebeer, Nicholas J. Bryan, and Paris Smaragdis, "Auto-DSP: Learning to Optimize Acoustic Echo Cancellers," Oct. 2021, arXiv:2110.04284.

[11] Stefan Bilbao, *Numerical Sound Synthesis: Finite Difference Schemes and Simulation in Musical Acoustics*, Wiley, Oct. 2009.

[12] Cleve Moler and Peter J. Costa, *Symbolic Math Toolbox User's Guide Version 2.0*, 1997.

[13] Michael B. Monagan and Walter M. Neuenschwander, "GRADIENT: Algorithmic differentiation in Maple," in *Proceedings of the 1993 International Symposium on Symbolic and Algebraic Computation - ISSAC '93*, Kiev, Ukraine, 1993, pp. 68–76, ACM Press.

[14] Soeren Laue, "On the Equivalence of Automatic and Symbolic Differentiation," Dec. 2022, arXiv:1904.02990.

[15] Brian Guenter, "Efficient Symbolic Differentiation for Graphics Applications," *ACM Transactions on Graphics*, vol. 26, no. 3, 2007.

[16] Dominique Villard and Michael B. Monagan, "ADrien: An implementation of automatic differentiation in Maple," in *Proceedings of the 1999 International Symposium on Symbolic and Algebraic Computation (ISSAC '99)*, New York, NY, USA, July 1999, pp. 221–228, Association for Computing Machinery.

[17] Barak A. Pearlmutter and Jeffrey Mark Siskind, "Reverse-mode AD in a functional framework: Lambda the ultimate backpropagator," *ACM Transactions on Programming Languages and Systems*, vol. 30, no. 2, pp. 7:1–7:36, Mar. 2008.

[18] Alexey Radul, Adam Paszke, Roy Frostig, Matthew J. Johnson, and Dougal Maclaurin, "You Only Linearize Once: Tangents Transpose to Gradients," *Proceedings of the ACM on Programming Languages*, vol. 7, pp. 43:1246–43:1274, Jan. 2023.

[19] Wenbin Yu and Maxwell Blair, "DNAD, a simple tool for automatic differentiation of Fortran codes using dual numbers," *Computer Physics Communications*, vol. 184, no. 5, pp. 1446–1452, May 2013.

[20] Dan Kalman, "Doubly Recursive Multivariate Automatic Differentiation," *Mathematics Magazine*, vol. 75, no. 3, 2002.

[21] Jesse Sigal, "Automatic Differentiation via Effects and Handlers: An Implementation in Frank," Jan. 2021, arXiv:2101.08095.

[22] Jarrett Revels, Miles Lubin, and Theodore Papamarkou, "Forward-Mode Automatic Differentiation in Julia," July 2016, arXiv:1607.07892.

[23] Robert E. Wengert, "A simple automatic derivative evaluation program," *Communications of the ACM*, vol. 7, no. 8, pp. 463–464, Aug. 1964.

[24] Romain Michon, Julius Smith, Chris Chafe, Ge Wang, and Matthew Wright, "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.

[25] Dirk Roosenburg and Romain Michon, "A Wave Digital Filter Modeling Library for the Faust Programming Language," in *Proceedings of the 18th Sound and Music Computing Conference*, Online, 2021.

[26] Yann Orlarey, Dominique Fober, and Stéphane Letz, "Syntactical and Semantical Aspects of Faust," *Soft Computing*, vol. 8, no. 9, pp. 623–632, 2004.

[27] Aloïs Brunel, Damiano Mazza, and Michele Pagani, "Backpropagation in the simply typed lambda-calculus with linear negation," *Proceedings of the ACM on Programming Languages*, vol. 4, pp. 1–27, Jan. 2020.

[28] Rómer Rosales, Mark Schmidt, and Glenn Fung, "Fast Optimization Methods for L1 Regularization: A Comparative Study and Two New Approaches," in *Proceedings of the 18th European Conference on Machine Learning*, Warsaw, Poland, Sept. 2007, pp. 286–297.

[29] Yann Orlarey, Dominique Fober, and Stéphane Letz, "FAUST: An Efficient Functional Approach to DSP Programming," *New computational paradigms for computer music*, pp. 65–96, 2009.

[30] Sören Laue, Matthias Mitterreiter, and Joachim Giesen, "A Simple and Efficient Tensor Calculus," in *Proceedings of the AAAI Conference on Artificial Intelligence*, New York, NY, USA, Apr. 2020, vol. 34, pp. 4527–4534.