Proceedings of the 4th International Faust Conference

Romain Michon, Stéphane Letz, Francesco Mulassano

Proceedings of the

4th International Faust Conference

November 21st -22nd, 2024 – Turin (Italy)

SoundMit // Inria // GRAME-CNCM

IFC 2024

Proceedings of the 4th International Faust Conference

November 21st-22nd, 2024

Romain Michon, Stéphane Letz, and Francesco Mulassano, eds.

4th International Faust Conference
November 21st-22nd, 2024

IFC-24 is organized by:

With the support of:

Proceedings of the 4th International Faust Conference
Romain Michon, Stéphane Letz, and Francesco Mulassano, Editors
ISBN: 978-2-9597911-0-9
EAN: 9782959791109
Website: `https://ifc24.soundmit.com/`

Bibtex:

```
@proceedings{24IFCConf,
Editor = {Romain Michon, Stéphane Letz, and Francesco Mulassano},
Organization = {SoundMit, Inria, and GRAME-CNCM},
Publisher = {GRAME-CNCM},
Title = {Proceedings of the 4th International Faust Conference},
Year = {2024}}
```


Typeseting by Romain Michon using LaTeX.

Table of Contents

IFC-24 Preface

By Romain Michon, Stéphane Letz, and Francesco Mulassano.

Welcome to the 4th International Faust Conference (IFC-24)! IFC-24 gathers members of the Faust community every two years. It is a platform to present new and ongoing work around the Faust programming language (`https://faust.grame.fr`) through talks, demos, workshops, short performances, etc. IFC-24 also hosts roundtables to collectively brainstorm on the future of the Faust ecosystem.

IFC-24 is taking place in Turin on November 21-22, 2024 at "Toolbox Co-Working" under the auspices of Fase Lunare and AlphaLab - Laboratorio Elettro Musicale and with the support of Inria (French National Institute for Research in Digital Science and Technology) and GRAME - Centre National de Création Musicale, the birthplace of Faust.

One of the main novelty of this edition of IFC is its... published proceedings. While papers were presented at previous editions of IFC, they were only made available through the conference website and they were not "officially" published. Hence, the present proceedings hosts a total of nine papers and also provides a brief description of the two workshops and of the four demos that were given at IFC-24.

In a time when it is so easy to remotely work collectively on a project such as Faust, we do believe that hosting in-person events like IFC helps strengthen our community by providing what technology can't offer yet: the serendipity of direct social interactions. The time we spend talking with each other during coffee breaks, lunches, and dinners is essential for provoking new ideas, collaborations, etc.

We look forward to seeing you at the next edition of IFC!

Organizational Committee, Authors, and Presenters

Organizational Committee
Francesco Mulassano (SoundMit, Italy) – General Chair
Romain Michon (Inria, France) – Scientific Co-Chair
Stéphane Letz (GRAME - Centre National de Création Musicale, France) – Scientific Co-Chair

Authors
Alessandro Anatrini (Hochschule für Musik und Theater, Hamburg, Germany)
Christian Blöcher (IEM, University of Music and Performing Arts Graz, Austria)
Alain Bonardi (CICM, Paris 8 University, France)
Michel Buffa (Université Côte d'Azur, France)
Domenico Cipriani (Pharo Association, Italy)
Paul Goutmann (CICM, Paris 8 University, France)
Felix Holzmüller (IEM, University of Music and Performing Arts Graz, Austria)
Pierre, Lecomte (Ecole Centrale de Lyon, France)
Stéphane Letz (GRAME - Centre National de Création Musicale, France)
Tomoya Matsuura (Tokyo University of the Arts, Japan)
Romain Michon (Inria, France)
José Miguel Fernandez (IRCAM, France)
Sebastian Jordan Montaño (Université de Lille and Inria, France)
Yann Orlarey (Inria, France)
Alois Sontacchi (IEM, University of Music and Performing Arts Graz, Austria)
Thomas Rushton (Inria, France)

Workshops, Demos, and Tutorials Presenters
Martin Bartlett (Developer, Creator, UK)
Alain Bonardi (CICM, Paris 8 University, France)
Domenico Cipriani (Pharo Association, Italy)
Leon Gnaedinger (obsoleszenz)
Paul Goutmann (CICM, Paris 8 University, France)
Christophe Lebreton (LiSiLoG, France)
Landon McCoy (Chaos Audio, USA)
Ruolun Allen Weng (Shanghai Conservatory of Music, China)

IFC-24 Paper Session 1

SPATIAL GRANULAR SYNTHESIS WITH AMBITOOLS AND ANTESCOLLIDER

Pierre Lecomte [*]

Ecole Centrale de Lyon, CNRS, Universite Claude Bernard Lyon 1, INSA Lyon, LMFA, UMR5509, 69130, Ecully, France
Lyon, France
pierre.lecomte@ec-lyon.fr

José Miguel Fernandez [†]

IRCAM, STMS, UMR9912, 75004, Paris, France
jose.miguel.fernandez@ircam.fr

ABSTRACT

This paper presents a spatial granular synthesis tool developed in FAUST and part of the Ambitools v1.3 library. This tool generates a swarm of spatialized sound grains in a spherical shell sector using the Higher Order Ambisonic format. The grains can be played from pre-recorded sound files or from a circular buffer of the input signal for live use. This tool is then integrated into the AntesCollider library to offer fine control over the evolution of the grain swarm as well as its visualization.

1. INTRODUCTION

Granular synthesis, originally based on the pioneering work of Gabor [1] and Xenakis [2], is a sound synthesis technique that breaks down an audio signal into small segments called "grains". Building on the general concepts developed by Truax [3] and Roads [4], granular synthesis has been extensively used by composers and musicians to create new sounds from pre-recorded audio (whether concrete or synthesized). This technique allows for the creation of complex sounds, rich textures, and evolving sounds by manipulating various parameters such as grain duration, overlap time, envelope type, and playback position within the audio file. The use of granular synthesis in spatial audio, or granular spatialization, is a natural evolution of this concept. Beyond its original capabilities, it enables the creation of diverse spatial sound morphologies. For instance, one can easily generate sound masses that move not only in timbre but also in space, such as a sound point exploding into the surrounding space, multiple sounds transforming and converging towards a specific spatial point, or using random positions to generate an immersive spatial sound. Although spatial granulation is not a new concept [5, 6], the implementation we propose, based on the Ambitools library [7], allows for the generation of many grains in a Higher Order Ambisonics (HOA) format of any order. This approach enables the dynamic creation of various granular synthesis modules in real-time from an audio file or live input. The spatial granular synthesizer being written in FAUST, many plug-ins formats are available which can be integrated in multiple environments. We present here an integration of this tool in AntesCollider [8], which provides an interface for precise control over the evolution of a swarm of grains within the 3D spatial environment. AntesCollider is a library dedicated to electronic music composition and temporal sequencing, enabling, among other things, synchronization with a musician through score follower or gesture follower. It also allows for 3D visualizations through the integration of the OpenFrameworks library. The paper is organized as follows: Technical details on the FAUST implementation of the spatial granular synthesis engine is given in Sec. 2. Then, the integration to AntesCollider is presented in Sec. 3. Conclusions and future works are given in Sec. 4.

[*] https://sekisushai.net/ambitools

[†] https://josemiguel-fernandez.com/

2. THE GRANULATOR

The spatial granulation tool is written in the FAUST language [9]. It is part of the ambitools plug-in suite [7] v1.3 as the "Granulator" (`granulator.dsp`) [1]. The tool generates N^2 parallel signal streams of sound "grains", spatialize each grain individually in a swarm whose geometry define a spherical shell sector. An example of Graphical User Inteface (GUI) for the Granulator is shown in Fig. 1.

This section detail the construction of a stream of grains, present extra parameters for each grain and their spatialization. A discussion of the FAUST compilation time is carried out.

2.1. Grain Stream Generation

Each of the N streams of grains is a signal made of concatenated grains: once a grain is played, a new one is generated and so on. The grains are read from a buffer which can be either a sound file (by using the `soundfile` primitive), or a circular buffer fed with an input signal (by using the `rwtable` primitive). The user can choose which sound source to use at runtime. For both cases, a "read index" signal, which gives the samples index over time, is needed and constructed according the following grain parameters:

- duration,
- reading speed,
- starting index in the buffer,
- reading direction: forward or reverse.

An example of read index signal as well as the duration, starting sample and reading speed parameters for a grain stream is shown in Fig. 2

For each grain of each stream, the parameters are set randomly using decorrelated random noise signals using `no.rnoises` primitive [3]. The random signals are scaled and shifted according to the parameters ranges which can are tuned using sliders in the User Interface (see Fig. 1). They are fed into a sample and hold function. The latter is denoted `trig` and a block diagram is shown in Fig. 3. The `trigger` signal in Fig. 3 is an impulse which is

[1] https://sekisushai.net/ambitools/docs/granulator.html

[2] $N \in \mathbb{N}$ is set at compilation time.

[3] Note that we don't use of `no.noises` primitive here as it would produce the same grains sequences at each plugin initialization because the noises generator seeds are the same.

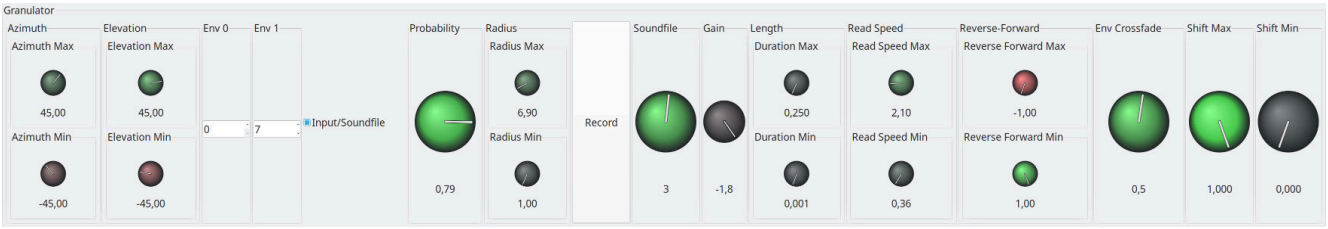


Figure 1: *GUI of the Granulator compiled with* `faust2jack` *script.*

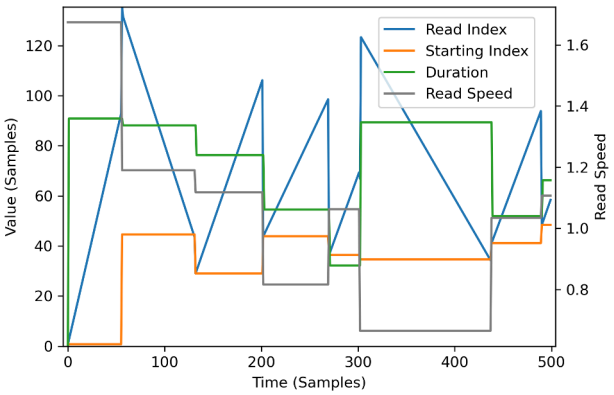


Figure 2: *The read index signal for a stream of grains (in blue): the slope is proportional to the read speed. Its sign gives the read direction. Once the read index minus the starting sample equals the duration, a new set of parameters is randomly chosen within the parameter ranges.*

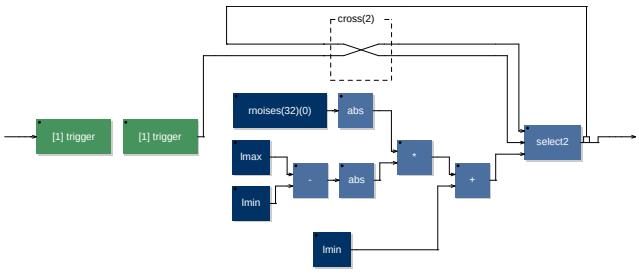


Figure 3: *Block diagram of the* `trig` *function, i.e., a sample and hold function triggered by a impulse signal. Here the 0-th noise among 32 noises signals (*`rnoises(32)(0)`*) is scaled and shifted to give a random signal between* `lmin` *and* `lmax`*. A value of this signal is hold until an impulse is received (*`trigger`*).*

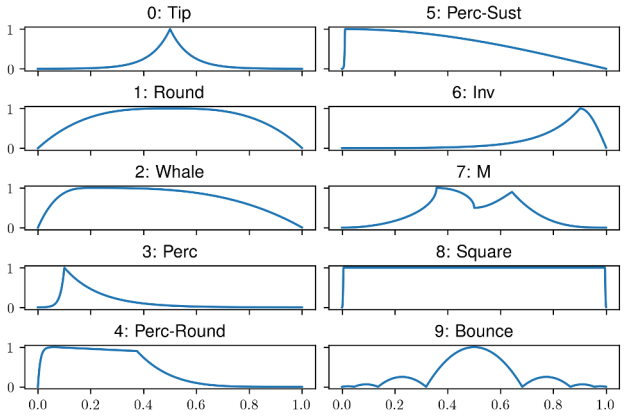


Figure 4: *The various envelopes used on a grain stream. A cross-fade allows interpolation between two envelopes among this bank at runtime.*

non-zero only when the read index value minus the starting index equals the grain duration. At this instant the `trig` function of Fig. 3 samples the current value of the random signal and hold it until the next impulse. Thus, a new set of grain parameters is randomly set at the end of each grain.

2.2. Extra Parameters

Grain Envelopes Since the reading index signal exhibits discontinuities (see Fig. 3), audible clicks may occur at each grain change. To prevent this phenomenon, but also to provide control over grain dynamics, an envelope starting and ending at zero is applied to each grain. The same envelope is used in the N streams. This envelope is constructed at runtime with a cross-fade between two envelopes taken from a bank. The envelopes bank can be seen in Fig. 4.

Grain Probability To control the density of grains played simultaneously in the N streams, a `Probability` slider (see Fig. 1) is used (between 0 and 100%). The `trig` function of Fig. 3 is used with `lmin = 0` and `lmax = 1` and this value is compared with the `Probability` slider value. If the value is higher, the gain grain is set to 0 and the grain is not played.

Markers The starting index in Fig. 3 is by default chosen randomly in the buffer. However, in the case of sound file as the sound source, it is possible to use "markers" given as a list of samples index (using the `waveform` primitive). In this case, the starting

index of each grain is chosen randomly among these markers. This feature helps the composer to select time instants in the soundfile where the sound grains are of interest. An example of such markers in a sound sample are shown in Fig. 5:

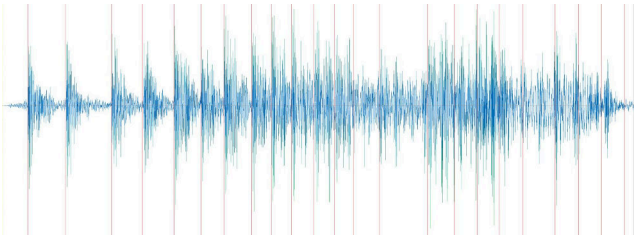


Figure 5: *Markers in red are used to supervise the starting index choice for each grain in the sound sample.*

2.3. Grain Spatialization

For each grain, the spatialization stage is performed in HOA format as a source point with the ambitools encoder (`encoder.dsp`)[4]. The HOA order $L \in \mathbb{N}$ is set at compilation time. The spherical coordinates are randomly picked within intervals set in the user interface at runtime (see Fig. 1). To do so, the `trig` function of Fig. 3 is used. In this way, the radius, azimuth and elevation ranges define a spherical shell sector in which the grain swarm evolves. Optionnally, the grain coordinates and amplitudes signals are forwarded into `bargraphs` to transmit these values through Open Sound Control (OSC) or as output signals in SuperCollider, using the `faust2supercollider` script, unlocking the swarm visualization in AntesCollider (see Sec. 3.2).

2.4. FAUST Compilation

The Granulator in its current version uses 8 decorrelated random noises signals[5] for each of the N stream. In addition, each of the N monophonic stream is encoded into $(L+1)^2$ HOA signals. Moreover, to switch between recorded buffer or sound file at run time, both signal are computed at runtime, as well as 20 signals per stream for the grains envelopes. Finally, there are $28N + (L+1)^2$ audio rate signals to compute at runtime. As L and N increase the FAUST compiler takes a rapidly increasing time to evaluate and progagate the code and produce the binary output, if at all. This can be seen in Fig. 6 for increasing N and L. Therefore, we suggest compiling the Granulator keeping the value of N low and launching several instances of the plug-in in the host software. Note that use of the `no.rnoises` primitive in the code is therefore essential to use different seeds for the noise generators of the various plug-in instances.

[4] `https://sekisushai.net/ambitools/docs/encoder.html`

[5] One random signal for each of the following parameters: duration, reading speed, starting index, reading direction, probability, radius, azimuth and elevation.

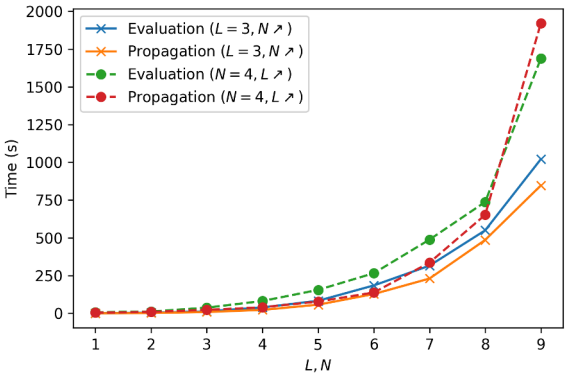


Figure 6: *The time spent by the* FAUST *on the evaluation and propagation steps for: A constant HOA order $L = 3$ and increasing number of grain streams N; A constant grain streams number $N = 4$ and increasing HOA order. The values are obtained using `faust -time -t 0 granulator.dsp` on a conventional laptop.*

3. ANTESCOLLIDER INTEGRATION

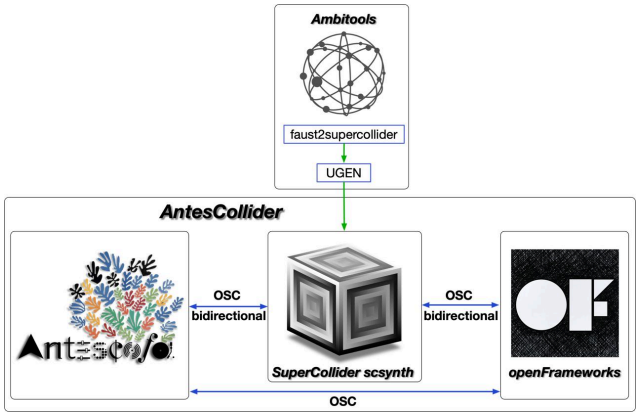


Figure 7: *Interaction between Ambitools and Antescollider.*

AntesCollider [8] is a library for composing and writing electronic music, created using the Antescofo [10] programming language. It consists of two elements: an audio rendering engine, the SuperCollider [11] servers (scsynth), and a synchronous programming language to control them, Antescofo. The goal of this integration is to dynamically create real-time audio processing chains with fine control over parameters over time. The expressiveness of the Antescofo language and its temporal control allow for efficient and concise creation and on-the-fly restructuring of audio processing, simplifying and adding flexibility to synthesis control. The main motivations behind the creation of this library are both musical and compositional. They aim towards the conception of an environment that can extend the composer's palette and imagination in order to create music with a strong component of interactivity, thanks to score following and data processing derived from performance, such as audio signal analysis and gesture tracking of performers. This interactivity can also be easily extended to other

media, such as video, through the integration of communication protocols like OSC. AntesCollider integrates advanced functions:

- dynamic audio chain creation,
- preset saving and loading,
- graphical monitoring,
- spatial composition thanks to the integration of the Ambitools [12, 7] library,
- algorithmic control (with examples of physical models (boids, mass-spring), KNN in parameter spaces, etc.).

3.1. Granulator Unit Generator

The Granulator tool of Sec. 2 is integrated within Antescollider as a Unit Generator (UGEN) in SuperCollider (see Fig. 7). To do so, the `granulator.dsp` code is compiled using the `faust2supercollider` script. An example of use within AntesCollider is shown in the code of Fig. 9.

3.2. 3D Visualization

To visualize the grain swarm from the Granulator, we use the OpenFrameworks [6] C++ library for real-time image and video synthesis. To receive data from the Granulator UGEN, an OSC connection is used with the SuperCollider scsynth (see Fig. 7). The UGEN sends the spherical coordinates as well as the amplitude in dB of each grain in real-time at the audio rate. It is then converted into OSC messages via the `SendReply.ar` command in a SynthDef in SuperCollider. This implementation allows for real-time visualization of the 3D position of each grain in space, as well as their amplitude: the grain dynamically changes size based on the amplitude[7]. In parallel with this implementation, a 3D Ambisonic energy visualizer is used to display energy on a spherical surface. The UGEN for this energy visualizer uses a sampling decoder from the `sampling_decoder.dsp`[8] tool of the Ambitools library on a 974-node Lededev grid [13]. This 3D Visualization tool is shown in Fig. 8:

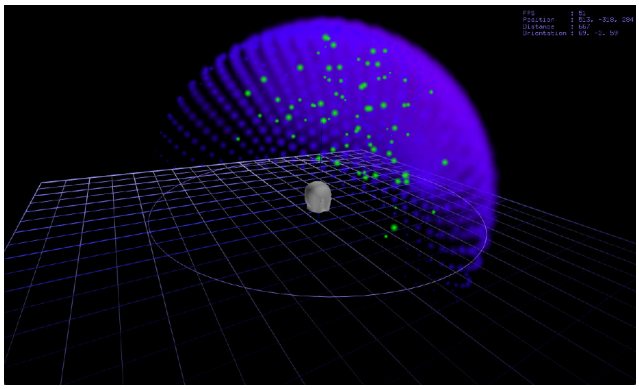


Figure 8: *The 3D visualization of the grain swarm. Here $N = 30$ grain streams are used. The grains are represented in green, their size is proportional to they amplitude. The acoustic energy of the resulting HOA scene is shown in purple. A dummy head facing the front direction is placed at origin and represented in grey.*

4. CONCLUSIONS

We have developed a new spatial granular synthesis tool, the "Granulator", integrated into the Ambitools v1.3 library and implemented in the AntesCollider library. The "Granulator" is the result of research and creation at GRAME[9]. The development and addition of various parameters were carried out by considering elements of spatial and musical perception, with the aim of using it in real-time, both with audio files and live input. Resource optimization through programming in FAUST and its deployment in SuperCollider (using the `faust2supercollider` script) allows for a versatile, dynamic, HOA, multi-grain granulator where all parameters can be modified in real-time, providing great flexibility and richness both in timbre and spatial impession. Its implementation in the AntesCollider library, thanks to the Antescofo synchronous programming language, enables the creation of intuitive spatial morphologies in direct relation to instrumental performance and/or with fine control of all parameters in a multimodal way. Future developments include the possibility of creating spatial zones based on timbre, through the use of audio descriptors. This involves the idea of creating and recreating spatial soundscapes based on characteristics such as pitch, spectrum, dynamics, roughness, and more. The Granulator will be extensively used for the creation of a new piece for trumpet and live electronics, "Gnomon", commissioned by GRAME and to be premiered in June 2025 in Lyon, France.

5. ACKNOWLEDGMENTS

The authors would like to thank GRAME for hosting them during artistic residencies in 2024. Most of this work was carried out at that time. In particular we would like to thank Stéphane Letz for his technical support on FAUST and `faust2supercollider` script. Part of this work has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme. ERC REACH: Raising Co-Creativity in Cyber-Human Musicianship, Grant agreement #883313

[6] `https://openframeworks.cc/`

[7] Note that if the grain amplitude is 0 (as when its probability is 0), it disappears in the visualization.

[8] `https://sekisushai.net/ambitools/docs/sampling_decoder.html`

[9] `https://www.grame.fr/`

```
// Example of creating a track in AntesCollider and initialize a Granulator:

// First of all, we instantiate a 'mix_group' (a group of tracks) in ambisonic,
// in this case, in 4th order on the scsynth server 'server1'
obj::mix_group_HOA(''group_hoa1'', ''server1'', ''HOA_GRAME_ADTN3D'', 4)

// Then, instantiate the track 'granulator_track' in the group 'group_hoa1'.
// All tracks instantiated in this group will inherit the order of the group (in
    this case, 4th order).
obj::crea_track_HOA(''granulator_track'', ''group_hoa1'', fade_in = 1, amp = -6,
    encoder = false, doppler = 0)

// Retrieve the audio bus and assign it to the Antescofo variable '$hoa_bus'.
$hoa_bus := $tracks(''granulator_track'').$hoa_inter_bus

// Add the module (SuperCollider SynthDef) 'Granulator8_4' to the track '
    granulator_track', a UGEN with 8 grain streams in
// 4th order and set the initialization parameters for the granular synthesis module
     'Granulator8_4'
$tracks(''granulator_track'').mod_add([''Granulator8_4'', ''globTBus'', $hoa_bus,
    radius_max, 1, radius_min, 0.1, azimuth_max, 180, azimuth_min, -180,
    elevation_max, 45.0, elevation_min, -45.0, read_speed_max, 1.5,read_speed_min,
    0.5, duration_max, 0.51, duration_min, 0.001, shift_min, 0.0, shift_max, 0.5,
    sound, 4, env_0, 8, env_1, 0, env_crossfade, 0])

// Set the initialization parameters for the granular synthesis module '
    Granulator8_4'.
$tracks(''granulator_track'').set(''Granulator8_4'', [radius_max, 1, radius_min,
    0.1, azimuth_max, 180, azimuth_min, -180,
elevation_max, 45.0, elevation_min, -45.0, read_speed_max, 1.5, read_speed_min, 0.5,
     duration_max, 0.51, lduration_min, 0.001, shift_min, 0.0, shift_max, 0.5, sound
    , 4, env_0, 8, env_1, 0, env_crossfade, 0])

// Change parameters in real-time (live coding)
$tracks(''granulator_track'').set(''Granulator8_4'', [record_input, 0])
$tracks(''granulator_track'').set(''Granulator8_4'', [grains_probability, 0.05])
$tracks(''granulator_track'').set(''Granulator8_4'', [reverse_forward_max, 1])
$tracks(''granulator_track'').set(''Granulator8_4'', [reverse_forward_min, -1])
$tracks(''granulator_track'').set(''Granulator8_4'', [record, 0])
$tracks(''granulator_track'').set(''Granulator8_4'', [azimuth_min, -30])
$tracks(''granulator_track'').set(''Granulator8_4'', [azimuth_max, 30])
$tracks(''granulator_track'').set(''Granulator8_4'', [elevation_min, -20])
$tracks(''granulator_track'').set(''Granulator8_4'', [elevation_max, 40])
$tracks(''granulator_track'').set(''Granulator8_4'', [radius_max, 10])
$tracks(''granulator_track'').set(''Granulator8_4'', [radius_min, 1])

//Move parameters with continuous controls, in this case at different speeds using
    random LFOs.
$tracks(''granulator_track'').rand_lfo(''Granulator8_4'', azimuth_min, 0, 360, 0, ''
    linear'', 120)
$tracks(''granulator_track'').rand_lfo(''Granulator8_4'', azimuth_max, 0, 360, 0, ''
    linear'', 90)
$tracks(''granulator_track'').rand_lfo(''Granulator8_4'', elevation_min, -30, 0, 0,
    ''linear'', 30)
$tracks(''granulator_track'').rand_lfo(''Granulator8_4'', elevation_max, 0, 90, 0, '
    'linear'', 55)
$tracks(''granulator_track'').rand_lfo(''Granulator8_4'', radius_min, 0.1, 2, 0.1, '
    'linear'', 160)
$tracks(''granulator_track'').rand_lfo(''Granulator8_4'', radius_max, 1, 10, 1, ''
    linear'', 40)
```

Figure 9: *Example of the Granulator usage in AntesCollider.*

6. REFERENCES

[1] Dennis Gabor, "Acoustical quanta and the theory of hearing," *Nature*, vol. 159, pp. 591–594, 1947.

[2] Iannis Xenakis, "Formalized music. bloomington, indi-ana," 1971.

[3] Barry Truax, "Real-time granular synthesis with a digital signal processor," *Computer Music Journal*, vol. 12, no. 2, pp. 14–26, 1988.

[4] Curtis Roads, *The computer music tutorial*, MIT press, 1996.

[5] Scott Wilson, "Spatial swarm granulation," in *ICMC*, 2008.

[6] Nicholas Mariette, "Ambigrainer-a higher order ambisonic granulator in pd," in *Ambisonics symposium*, 2009.

[7] Pierre Lecomte, "Ambitools: Tools for Sound Field Synthesis with Higher Order Ambisonics - V1.0," in *International Faust Conference*, Mainz, 2018, pp. 1–9.

[8] José Miguel Fernandez, Jean-Louis Giavitto, and Pierre Donat-Bouillud, "Antescollider: Control and signal processing in the same score," in *ICMC 2019-International Computer Music Conference*, 2019.

[9] Yann Orlarey, Dominique Fober, and Stéphane Letz, "FAUST: An efficient functional approach to DSP programming," *New Computational Paradigms for Computer Music*, vol. 290, 2009.

[10] Arshia Cont, "Antescofo: Anticipatory synchronization and control of interactive parameters in computer music.," in *International Computer Music Conference (ICMC)*, 2008, pp. 33–40.

[11] James McCartney, "Rethinking the computer music language: Super collider," *Computer Music Journal*, vol. 26, no. 4, pp. 61–68, 2002.

[12] Florian Grond and Pierre Lecomte, "Higher order ambisonics for supercollider," in *Linux audio conference*, 2017.

[13] Vyacheslav Ivanovich Lebedev and AL Skorokhodov, "Quadrature formulas of orders 41, 47, and 53 for the sphere," in *Russian Acad. Sci. Dokl. Math*, 1992, vol. 45, pp. 587–592.

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*, ε, 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 ε^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:

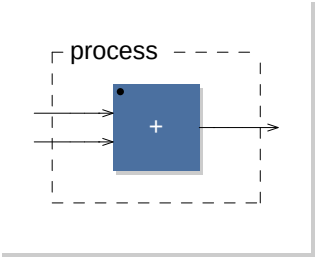


Figure 1: *Block diagram of a* FAUST *program consisting of a lone addition primitive.*

[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;
```

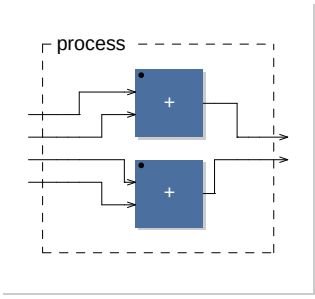


Figure 2: *Block diagram of a* FAUST *program representing a naive implementation of a dual signal differentiable addition primitive, consisting of two parallel additions.*

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;
```

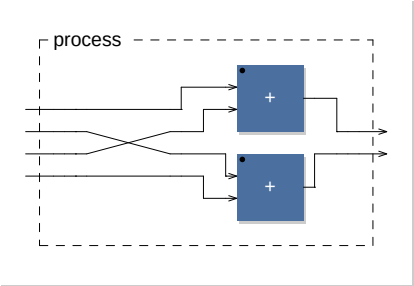


Figure 3: *Block diagram of a* FAUST *program representing a dual signal differentiable addition primitive. Via appropriate signal routing, valid dual signal output is produced.*

`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^{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, α, 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)),
```

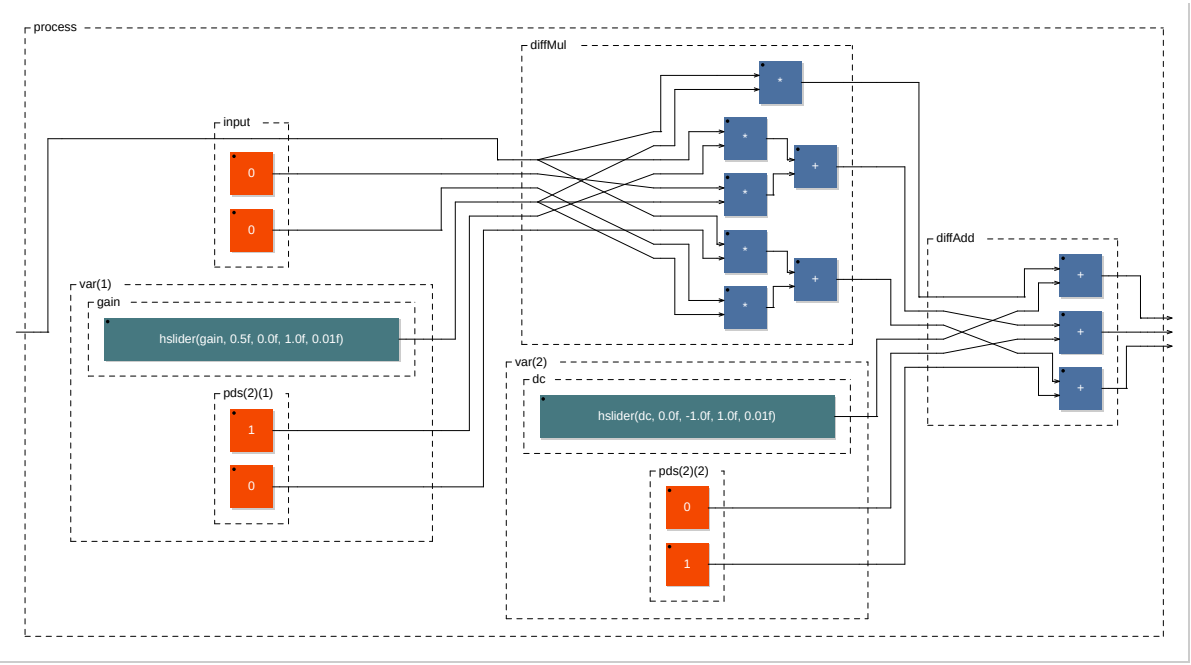


Figure 4: *Block diagram of a differentiable* FAUST *program consisting of differentiable gain and DC parameters applied to an input signal (see listing 6).*

```
      // 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

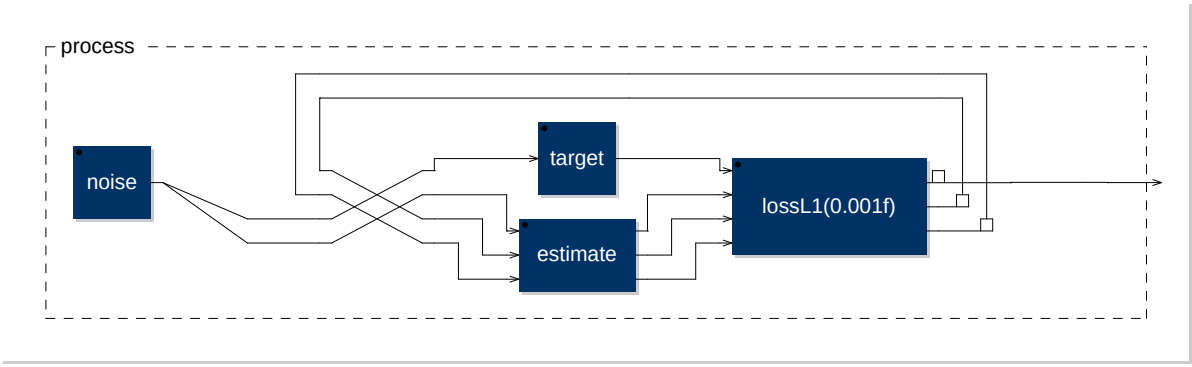


Figure 5: *Top-level block diagram for the full* FAUST *program of which an excerpt appears in listing 9.* `target` *algorithm, with* hidden *parameters, and* `estimate` *algorithm, with two* optimisable *parameters, process a noise signal. Their outputs are compared via a loss function; scaled derivatives 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 ∇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.

LAMBDA-MMM: THE INTERMEDIATE REPRESENTATION FOR SYNCHRONOUS SIGNAL PROCESSING LANGUAGE BASED ON LAMBDA CALCULUS

Tomoya Matsuura

Tokyo University of the Arts
Tokyo, Japan
me@matsuuratomoya.com

ABSTRACT

This paper proposes λ_{mmm}, a call-by-value, simply typed lambda calculus-based intermediate representation for a music programming language that handles synchronous signal processing and introduces a virtual machine and instruction set to execute λ_{mmm}. Digital signal processing is represented by a syntax that incorporates the internal states of delay and feedback into the lambda calculus. λ_{mmm} extends the lambda calculus, allowing users to construct generative signal processing graphs and execute them with consistent semantics. However, a challenge arises when handling higher-order functions because users must determine whether execution occurs within the global environment or during DSP execution. This issue can potentially be resolved through multi-stage computation.

1. INTRODUCTION

Many programming languages have been developed for sound and music; however, only a few possess strongly formalized semantics. A language that is both rigorously formalized and practical is Faust [1]; it combines blocks with inputs and outputs with five primitive operations: parallel, sequential, split, merge, and recursive connection. Almost any type of signal processing can be written in Faust by providing basic arithmetic, conditionals, and delays as primitive blocks. In a later extension, a macro based on a term rewriting system was introduced that allowed users to parameterize blocks with an arbitrary number of inputs and outputs [2].

This strong abstraction capability through formalization enables Faust to be translated into various backends, such as C, C++, Rust, and LLVM IR. On the other hand, Faust's Block Diagram Algebra (BDA) lacks theoretical and practical compatibility with common programming languages. Although it is possible to call external C functions in Faust, these functions are assumed to be pure functions that do not have internal states. Therefore, while it is easy to embed Faust in another language, it is not easy to call another language from Faust.

In addition, a macro for Faust is an independent term rewriting system that generates a BDA based on pattern matching. Consequently, the numeric arguments for pattern matching are implicitly required to be integers, which can sometimes lead to compile-time errors despite the fact that BDA does not distinguish between real and integer types. However, the implicit typing rules are not intuitive for novice users.

Proposing a computational model for signal processing based on more generic computational models such as lambda calculus has the potential to enable interoperability between many different general-purpose languages and facilitate the appropriation of existing optimization methods and the implementation of compilers and runtimes.

It has been demonstrated that BDA can be converted into a general-purpose functional language using an arrow, which is a higher-level abstraction of monads [3]. However, higher-order functions in general-purpose functional languages are often implemented using dynamic memory allocation and deallocation, making them difficult to use in host languages designed for real-time signal processing.

In addition, Kronos [4] and W-calculus [5] are examples of lambda calculus-based abstractions influenced by Faust. Kronos is based on the theoretical foundation of the System-$F\omega$, a variation of lambda calculus in which the types themselves can be abstracted (i.e., a function that takes a type as input and returns a new type can be defined). In Kronos, type calculations correspond to signal graph generation, whereas the value calculations correspond to actual processing. Delay is the only special primitive operation in Kronos, and feedback routing can be represented as a recursive function application in type calculations.

W-calculus includes feedback as a primitive operation, along with the ability to access the value of a variable from the past (i.e., delay). W-calculus restricts systems to those that can represent linear-time-invariant processes, such as filters and reverberators, and defines more formal semantics, aiming for automatic proofs of linearity and the identity of graph topologies.

Previously, the author designed the music programming language *mimium* [6]. By incorporating basic operations such as delay and feedback into the lambda calculus, signal processing can be concisely expressed while maintaining a syntax similar to that of general-purpose programming languages. Notably, *mimium*'s syntax was designed to resemble the Rust programming language.

An earlier issue with *mimium* was its inability to compile code that contained combinations of recursive or higher-order functions with stateful functions involving delay or feedback because the compiler could not determine the data size of the internal state used in signal processing.

In this paper, I propose the syntax and semantics of λ_{mmm}, an extended call-by-value simply typed lambda calculus, as a computational model intended to serve as an intermediate representation for *mimium*[1]. In addition, I propose a virtual machine and its instruction set, based on Lua's VM, to execute this computational model in practice. Finally, I discuss both the challenges and potential of the current λ_{mmm} model, one of which is that users must differentiate whether a calculation occurs in a global context or during actual signal processing; the other is that runtime interoperability with other programming languages could be easier than in existing DSP languages.

[1] The newer version of mimium compiler and VM based on the model presented in this paper is on the GitHub. https://github.com/tomoyanonymous/mimium-rs

```
fn onepole(x,g){
  x*(1.0-g) + self*g
}
```

Listing 1: *Example of the code of one-pole filter in mimium.*

2. SYNTAX

$$\begin{array}{lll} \tau_p ::= R & [real] & v_p ::= r \quad r \in \mathbb{R} \\ \quad | \ N & [nat] & \quad | \ n \quad n \in \mathbb{N} \\ \tau ::= \tau_p & & v ::= v_p \\ \quad | \ \tau \to \tau & [function] & \quad | \ cls(\lambda\, x.e, E) \end{array}$$

Types Values

$$\begin{array}{ll} e ::= x & x \in v_p \ [value] \\ \quad | \ \lambda x.e & [lambda] \\ \quad | \ let\ x = e_1\ in\ e_2 & [let] \\ \quad | \ fix\ x.e & [fixpoint] \\ \quad | \ e_1\ e_2 & [app] \\ \quad | \ if\ (e_c)\ e_t\ else\ e_e & [if] \\ \quad | \ delay\ n\ e_1\ e_2 & n \in \mathbb{N}\ [delay] \\ \quad | \ feed\ x.e & [feed] \\ \quad | ... & \end{array}$$

Terms

Figure 1: *Definition of Types, Values and Terms of the λ_{mmm}(Basic arithmetics are omitted).*

The types and terms of λ_{mmm} are presented in Figure 1.

Two terms are introduced in addition to the standard simply typed lambda calculus: $delay\ n\ e_1\ e_2$, which refers to the previous value of e_1 by e_2 samples (with a maximum delay of n to limit memory usage to a finite size), and $feed\ x.e$, an abstraction that allows the user to refer to the result of evaluating e from one time unit earlier as x during the evaluation of e itself.

2.1. Syntactic Sugar of the Feedback Expression in *mimium*

The programming language *mimium*, developed by the author, includes a keyword $self$ that can be used in function definitions to refer to the previous return value of the function. An example of a simple one-pole filter function, which mixes the input and last output signals such that the sum of the input and feedback gains is 1, is shown in Listing 1. This code can be expressed in λ_{mmm} as illustrated in Figure 2.

$$\begin{array}{l} let\ onepole = \\ \quad \lambda x.\lambda g.\ feed\ y.\ x * (1.0 - g) + y * g\ in\ ... \end{array}$$

Figure 2: *Equivalent expression to Listing 1 in λ_{mmm}.*

2.2. Typing Rules

$$\frac{\Gamma, x : \tau_a \vdash e : \tau_b}{\Gamma \vdash \lambda x.e : \tau_a \to \tau_b}\ \text{T-LAMBDA} \qquad \frac{\Gamma \vdash n{:}N\ \Gamma \vdash e1{:}\tau \quad \Gamma \vdash e_2{:}R}{\Gamma \vdash delay\, n\, e_1\, e_2 : \tau}\ \text{T-DELAY}$$

$$\frac{\Gamma, x : \tau_p \vdash e : \tau_p}{\Gamma \vdash feed\, x.e : \tau_p}\ \text{T-FEED} \qquad \frac{\Gamma \vdash e_c : R\ \Gamma \vdash e_t : \tau\ \Gamma \vdash e_e : \tau}{\Gamma \vdash if\ (e_c)\ e_t\ e_e\ : \tau}\ \text{T-IF}$$

Figure 3: *Excerpt of the typing rules for λ_{mmm}.*

Additional typing rules for typical simply typed lambda calculus are shown in Figure 3.

The primitive types include a real number type, used in most signal processing, and a natural number type, used for the indices of delay.

In W-calculus, which directly inspired the design of λ_{mmm}, the function types can only take tuples of real numbers and return tuples of real numbers. This restriction prevents the definition of higher-order functions. While this limitation is reasonable for a signal processing language—since higher-order functions require data structures such as closures that depend on dynamic memory allocation—it also reduces the generality of lambda calculus.

In λ_{mmm}, the problem of memory allocation for closures is delegated to runtime implementation (see Section 4), which allows the use of higher-order functions. However, $feed$ abstraction does not permit function types to be either input or output. Allowing function types in the $feed$ abstraction enables the definition of functions whose behavior could change over time. While this is theoretically interesting, there are no practical examples in real-world signal processing, and such a feature would likely further complicate the implementation.

3. SEMANTICS

An excerpt of the operational semantics for λ_{mmm} is shown in Figure 4. This big-step semantics conceptually explains the evaluation process. When the current time is n, the evaluation environment from t prior samples can be referred to as E^{n-t}. If the time is less than 0, any term is evaluated as the default value of its type (0 for numeric types).

Naturally, if we attempt to execute these semantics directly, we would need to recalculate from time 0 to the current time for every sample, saving all the variable environments at each step. However, in practice, a virtual machine is defined to account for the internal memory space used by $delay$ and $feed$, and λ_{mmm} terms are compiled into instructions for this machine before execution.

4. VM MODEL AND INSTRUCTION SET

The virtual machine (VM) model and its instruction set for running λ_{mmm} are based on Lua version 5 VM [7].

A key challenge when executing a computational model based on lambda calculus is handling the data structure, which is known as a closure. A closure captures the variable environment in which the inner function is defined, allowing it to refer to the variables

$$\frac{E^n \vdash e_2 \Downarrow v_d \; n > v_d \; E^{n-v_d} \vdash e_1 \Downarrow v}{E^n \vdash \; delay \; n \; e_1 \; e_2 \Downarrow v}$$

E-DELAY

$$\frac{}{E^n \vdash \; \lambda x.e \Downarrow cls(\lambda x.e, E^n)}$$

E-LAM

$$\frac{E^{n-1} \vdash e \Downarrow v_f \; E^n, x \mapsto v_f \vdash e \Downarrow v}{E^n \vdash \; feed \; x.e \Downarrow v}$$

E-FEED

$$\frac{E^n \vdash e_c \Downarrow n \quad n > 0 \; E^n \vdash e_t \; \Downarrow v}{E^n \vdash \; if(e_c) \; e_t \; else \; e_e \Downarrow v}$$

E-IFTRUE

$$\frac{E^n \vdash e_c \Downarrow n \quad n \leqq 0 \; E^n \vdash e_e \; \Downarrow v}{E^n \vdash \; if(e_c) \; e_t \; else \; e_e \Downarrow v}$$

E-IFFALSE

$$\frac{E^n \vdash e_1 \Downarrow cls(\lambda x_c.e_c, E_c^n) E^n \vdash e_2 \Downarrow v_2 \; E_c^n, \; x_c \mapsto v_2 \vdash e_c \Downarrow v}{E^n \vdash \; e_1 \; e_2 \Downarrow v}$$

E-APP

Figure 4: *Excerpt of the big-step semantics of* λ_{mmm}.

from the outer function's context. If the inner function is paired with a dictionary of variable names and values, the compiler (or interpreter) implementation is straightforward; however, the runtime performance is limited.

Conversely, the runtime performance can be improved using a process called closure conversion (or lambda lifting). This process analyzes all the outer variables referenced by the inner function and transforms the inner function by adding arguments; thus, the outer variables can be referred to explicitly. However, the implementation of this transformation in the compiler is relatively complex.

The Lua VM adopts a middle-ground approach between these two methods by adding the VM instructions `GETUPVALUE` and `SETUPVALUE`, which allow the outer variables to be dynamically referenced at runtime. The implementation of the compiler and VM using *upvalues* is simpler than full closure conversion while still avoiding significant performance degradation. In this approach, the outer variables are accessed via the call stack rather than the heap memory unless the closure escapes the context of the original function [8].

In addition, *upvalues* facilitate interoperability with other programming languages. Lua can be easily embedded through its C API, and when implementing external libraries in C, programmers can access the upvalues of the Lua runtime, not just the stack values available via the C API.

4.1. Instruction Set

The VM instructions for λ_{mmm} differ from those for the Lua VM in the following aspects:

1. Since mimium is a statically typed language, unlike Lua, instructions for basic arithmetic operations are provided for each type[2].
2. The call operation is split into normal function calls and closure calls owing to the static typing and to manage higher-order stateful functions (see 4.2 for details).
3. Conditional statements are implemented using a combination of two instructions, `JMP` and `JMPIFNEG`, whereas the Lua VM employs a dedicated `TEST` instruction.
4. Instructions related to for-loops, the `SELF` instruction used in object-oriented programming, and the `TABLE`-related instructions for metadata references to variables are omitted in mimium as they are unnecessary.
5. Instructions related to list-like data structures are also excluded from this paper, as the implementation of data structures such as tuples and arrays is outside the scope of the λ_{mmm} description here.

The VM for λ_{mmm} operates as a register machine similar to the Lua VM (post version 5). However, unlike traditional register machines, it does not employ physical registers. Instead, the register number simply refers to an offset index on the call stack relative to the base pointer during VM execution. The first operand of most instructions specifies the register number where the result of the operation is stored.

A list of instructions is presented in Figure 5 (basic arithmetic operations are partially omitted). The notation for the instructions follows the format outlined in the Lua VM documentation [7, p.13]. The operation name, list of operands, and pseudocode of the operation are displayed from left to right. When each of the three operands is used as an unsigned 8-bit integer, it is represented as `A B C`. If an operand is used as a signed integer, then it is prefixed with `s`. When the two operand fields are combined into a 16-bit value, the suffix `x` is added. For example, when `B` and `C` are merged and treated as a signed 16-bit value, they are represented as `sBx`.

In the pseudocode, `R(A)` denotes the data being moved in and out of the register (or call stack) at the base pointer + `A` for the current function. `K(A)` refers to the `A`-th entry in the static variable section of the compiled program, and `U(A)` accesses the `A`-th upvalue of the current function.

In addition to Lua's upvalue operations, four new operations—`GETSTATE`, `SETSTATE`, `SHIFTSTATE`, and `DELAY`—have been introduced to handle the compilation of the $delay$ and $feed$ expressions in λ_{mmm}.

4.2. Overview of the VM Structure

The overall structure of the virtual machine program, and instantiated closures for λ_{mmm} is depicted in Figure 6. In addition to the usual call stack, the VM has a dedicated storage area (a flat array) to manage the internal state data for feedback and delay.

[2] In the actual implementation, instructions such as `MOVE` include an additional operand to specify the word size of values, particularly for handling aggregate types like tuples.

```
MOVE          A B     R(A) := R(B)
MOVECONST     A B     R(A) := K(B)
GETUPVALUE    A B     R(A) := U(B)
(SETUPVALUE does not exist)
GETSTATE*     A       R(A) := SPtr[SPos]
SETSTATE*     A       SPtr[SPos] := R(A)
SHIFTSTATE*   sAx     SPos += sAx
DELAY*        A B C   R(A) := update_ringbuffer(SPtr[SPos],R(B),R(C))
 *(SPos,SPtr)= vm.closures[vm.statepos_stack.top()].state
  (if vm.statepos_stack is empty, use global state storage.)
JMP           sAx     PC +=sAx
JMPIFNEG      A sBx   if (R(A)<0) then PC += sBx
CALL          A B C   R(A),...,R(A+C-2) := program.functions[R(A)](R(A+1),...,R(A+B-1))
CALLCLS       A B C   vm.statepos_stack.push(R(A))
                      R(A),...,R(A+C-2) := vm.closures[R(A)].fnproto(R(A+1),...,R(A+B-1))
                      vm.statepos_stack.pop()
CLOSURE       A Bx    vm.closures.push(closure(program.functions[R(Bx)]))
                      R(A) := vm.closures.length - 1
CLOSE         A       close stack variables up to R(A)
RETURN        A B     return R(A), R(A+1)...,R(A+B-2)
ADDF          A B C   R(A) := R(B) as float + R(C) as float
SUBF          A B C   R(A) := R(B) as float - R(C) as float
MULF          A B C   R(A) := R(B) as float * R(C) as float
DIVF          A B C   R(A) := R(B) as float / R(C) as float
ADDI          A B C   R(A) := R(B) as int + R(C) as int
              ...Other basic arithmetic continues for each primitive types...
```

Figure 5: *Instruction sets for VM to run λ_{mmm}.*

This storage area is accompanied by pointers that indicate the positions from which the internal state data are retrieved via the `GETSTATE` and `SETSTATE` instructions. These positions are shifted forward or backward using the `SHIFTSTATE` instruction. The actual data layout in the state storage memory is statically determined during compilation by analyzing function calls involving references to `self`, `delay`, and other stateful functions, including those that recursively invoke such functions. The `DELAY` operation takes two inputs: `B`, representing the input value, and `C`, representing the delay time in the samples.

However, for higher-order functions—functions that take another function as an argument or return one—the internal state layout of the passed function is unknown at compile time. Consequently, a separate internal state storage area is allocated to each instantiated closure, which is distinct from the global storage area maintained by the VM instance. The VM also uses an additional stack to keep track of the pointers in the state storage of instantiated closures. Each time a `CALLCLS` operation is executed, the VM pushes the pointer from the state storage of the closure onto the state stack. Upon completing the closure call, the VM pops the state pointer off the stack.

Instantiated closures also maintain their own storage areas for upvalues. Until a closure exits the context of its parent function (known as an "Open Closure"), its upvalues hold a negative offset that references the current execution's stack. This offset is determined at compile time and stored in the function's prototype in the program. Furthermore, an upvalue may reference not only local variables but also upvalues from the parent function (a situation that arises when at least three functions are nested). Thus, the array of upvalue indices in the function prototype stores a pair of values: a tag indicating whether the value is a local stack variable or an upvalue from a parent function and the corresponding index (either the negative stack offset or the parent function's upvalue index).

For example, consider a scenario where the upvalue indices in the program are specified as `[upvalue(1), local(3)]`. In this case, the instruction `GETUPVALUE 6 1` indicates that the value located at index `3` from the upvalue list (referenced by `upvalue(1)`) should be retrieved from `R(-3)` relative to the base pointer, and the result should be stored in `R(6)`.

When a closure escapes its original function context through the `RETURN` instruction, the inserted `CLOSE` instruction moves the active upvalues from the stack to heap memory. These upvalues may be referenced from multiple locations, particularly in cases involving nested closures. Thus, a garbage collection mechanism is required to free memory once these upvalues are no longer in use.

In λ_{mmm}'s VM, since the paradigm is call-by-value and there is no reassignment expression, the `SETUPVALUE` instruction is omitted. If reassignment is allowed, open upvalues would need to be implemented as shared memory cells, as the values might be accessed by multiple closures that could trigger a `CLOSE` operation.

4.3. Compilation to the VM Instructions

Listing 2 shows a basic example of how the mimium code in Listing 1 is compiled into VM bytecode. When `self` is referenced, the value is retrieved using the `GETSTATE` instruction, and the internal state is updated by storing the return value with the `SETSTATE` instruction before returning it via the `RETURN` instruction. In this case, the actual return value is obtained using the

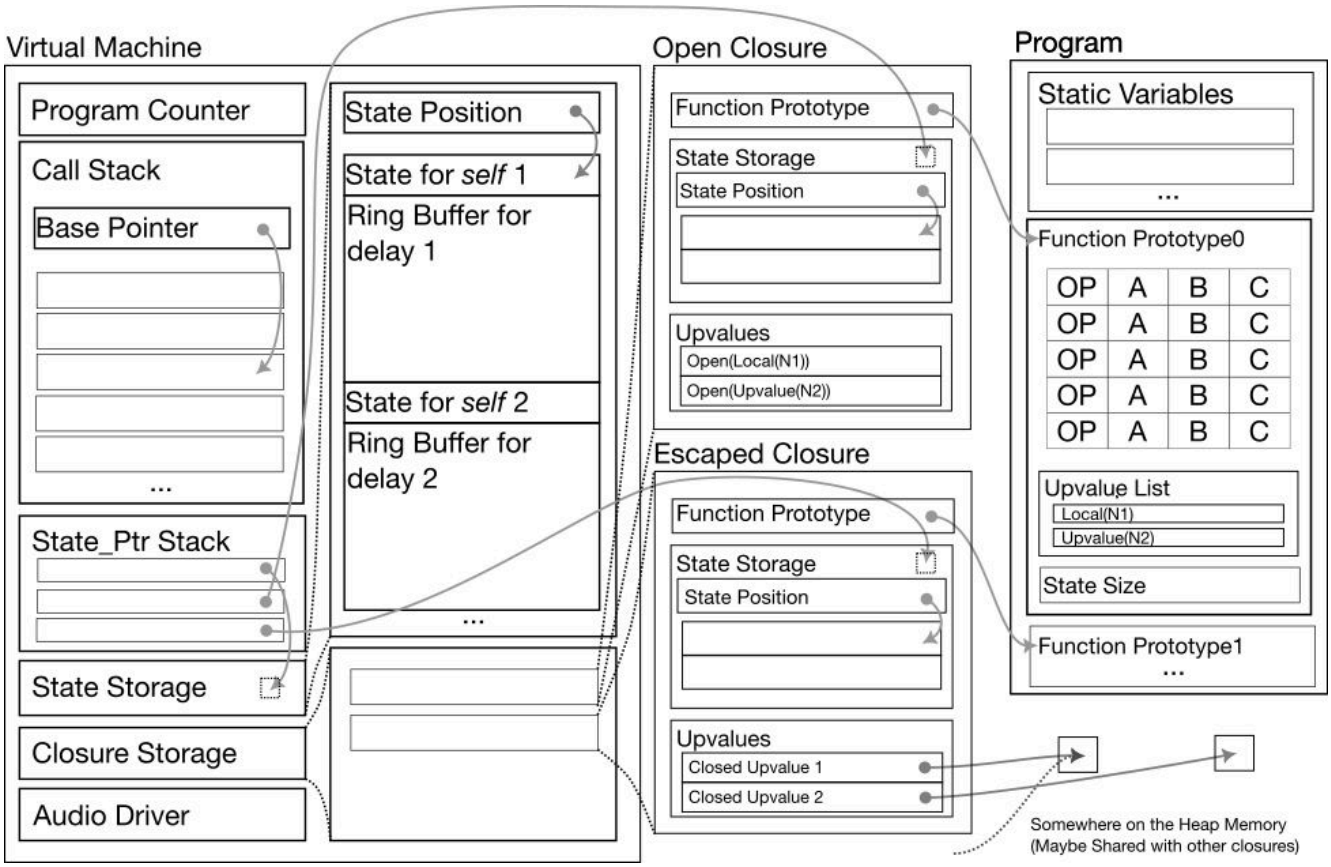


Figure 6: *Overview of the virtual machine, program and instantiated closures for* λ_{mmm}.

```
CONSTANTS:[1.0]
fn onepole(x,g) state_size:1
MOVECONST 2 0   // load 1.0
MOVE      3 1   // load g
SUBF      2 2 3 // 1.0 - g
MOVE      3 0   // load x
MULF      2 2 3 // x * (1.0-g)
GETSTATE  3     // load self
MOVE      4 1   // load g
MULF      3 3 4 // self * g
ADDF      2 2 3 // compute result
GETSTATE  3     // prepare return value
SETSTATE  2     // store to self
RETURN    3 1
```

Listing 2: *Compiled VM instructions of one-pole filter example in Listing 1*

second `GETSTATE` instruction, which ensures that the initial state value is returned at time = 0.

For example, if a time counter is written as $feedx.x + 1$, the decision on whether the return value at time = 0 should be 0 or 1 is left to the compiler design. Although returning 1 does not strictly follow the semantics of E-FEED in Figure 4, if the compiler is designed to return 1 at time = 0, the second `GETSTATE` instruction can be omitted, and the value for the `RETURN` instruction should be `R(2)`.

A more complex example, along with its expected bytecode instructions, is shown in Listings 3 and 4. The code defines a delay with feedback as `fbdelay`, while another function, `twodelay`, uses two feedback delays with different parameters. Finally, `dsp` uses two `twodelay` functions.

After each reference to `self` through the `GETSTATE` instruction or after calling another stateful function, the `SHIFTSTATE` instruction is inserted to advance the state storage position in preparation for the next non-closure function call. Before the function exits, the state position is reset to where it was at the beginning of the current function context using the `SHIFTSTATE` instruction. The total operand value for `SHIFTSTATE` within a function must always sum to 0. Figure 7 illustrates how the state position shifts with the `SHIFTSTATE` operations during the execution of the `twodelay` function. The argument for the `SHIFTSTATE` operation is a word size (a number of 64 bit values) and the word size for delay is `maximum delay time + 3` since the read index,

```
fn fbdelay(x,fb,dtime){
    x + delay(1000,self,dtime)*fb
}
fn twodelay(x,dtime){
    fbdelay(x,dtime,0.7)
      +fbdelay(x,dtime*2,0.8)
}
fn dsp(x){
    twodelay(x,400)+twodelay(x,800)
}
```

Listing 3: *Example code that combines self and delay without closure call.*

write index and the length of the ring buffer are added.

The state data can be stored as a flat array by representing the internal state as a relative position within the state storage, thereby simplifying compiler implementation; this avoids the need to generate a tree structure from the root, which was required in the previous implementation of mimium. This approach is similar to how upvalues simplify the compiler implementation by treating free variables as relative positions on the call stack.

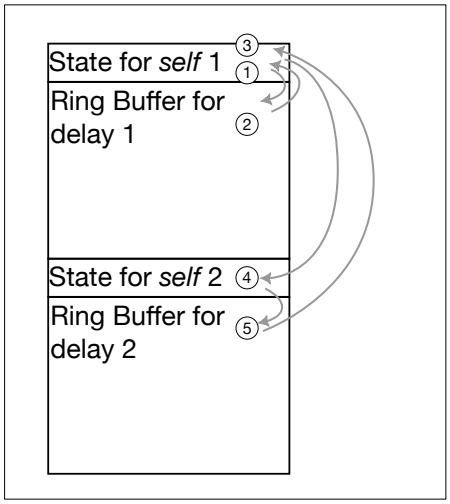


Figure 7: *Image of how the state position moves while executing* `twodelay` *function in Listing 4.*

Listing 5 shows an example of a higher-order function `filterbank`, which takes another function `filter`—accepting an input and a frequency as arguments—duplicates `n` instances of `filter` and adds them together[3].

The previous mimium compiler was unable to compile code that took a function with an internal state as an argument because the entire tree of internal states had to be statically determined at compile time. However, the VM in λ_{mmm} can handle this dynamically. Listing 6 shows the translated VM instructions for this code. Recursive calls on the first line of `filterbank`, as well as calls to functions passed as arguments or obtained through

[3]In the previous specification of mimium [6], the syntax for the variable binding and destructive assignment was the same (`x = a`). However, in the current syntax, variable binding uses the `let` keyword.

```
CONSTANTS:[0.7,2,0.8,400,800,0,1]
fn fbdelay(x,fb,dtime) state_size:1004
MOVE        3 0   //load x
GETSTATE    4     //load self
SHIFTSTATE 1      //shift Spos
DELAY       4 4 2 //delay(_,_,_)
MOVE        5 1   // load fb
MULF        4 4 5 //delayed val *fb
ADDF        3 3 4 // x+
SHIFTSTATE -1     //reset SPos
GETSTATE    4     //prepare result
SETSTATE    3     //store to self
RETURN      4 1   //return previous self

fn twodelay(x,dtime) state_size:2008
MOVECONST  2 5 //load "fbdelay"
    prototype
MOVE        3 0
MOVE        4 1
MOVECONST  5 0 //load 0.7
CALL        2 3 1
SHIFTSTATE 1004   //1004=state_size of
    fbdelay
MOVECONST  3 5 //load "fbdelay"
    prototype
MOVE        4 0
MOVECONST  5 1 //load 2
MULF        4 4 5
MOVECONST  5 0 //load 0.7
CALL        3 3 1
ADDF        3 3 4
SHIFTSTATE -1004
RETURN      3 1

fn dsp (x)
MOVECONST  1 6 //load "twodelay"
    prototype
MOVE        2 0
MOVECONST  3 3 //load 400
CALL        1 2 1
SHIFTSTATE 2008
MOVECONST  2 6 //load "twodelay"
    prototype
MOVE        2 3
MOVE        3 0
MOVECONST  3 4 //load 400
CALL        2 2 1
ADD         1 1 2
SHIFTSTATE -2008
RETURN      1 1
```

Listing 4: *Compiled VM instructions of feedback delay example in Listing 3*

```
fn bandpass(x,freq){
  //...
}
fn filterbank(n,filter_factory:()->(float,
    float)->float){
  if (n>0){
    let filter = filter_factory()
    let next = filterbank(n-1,
        filter_factory)
    |x,freq| filter(x,freq+n*100)
           + next(x,freq)
  }else{
    |x,freq| 0
  }
}
let myfilter = filterbank(3,| | bandpass)
fn dsp(){
      myfilter(x,1000)
}
```

Listing 5: *Example code that duplicates filter parametrically using a recursive function and closure.*

upvalues (like `filter`), are executed using the `CALLCLS` instruction rather than the `CALL` instruction. The `GETSTATE` and `SETSTATE` instructions are not used in this function because the internal state storage is switched dynamically when the `CALLCLS` instruction is interpreted.

5. DISCUSSION

As demonstrated in the example of the filterbank, in λ_{mmm}, a signal graph can be parametrically generated during the evaluation of the global context, whereas Faust uses a term-rewriting macro and Kronos employs type-level computation, as shown in Table 1.

The ability to describe both the generation of parametric signal processing and its execution content within single semantics makes it easier for novice users to understand the mechanics of the language. In addition, unified semantics may simplify runtime interoperability with other general-purpose languages.

However, there is a drawback: unified semantics can cause λ_{mmm} to deviate from the behavior typically expected in standard lambda calculus.

	Parametric Signal Graph	Actual DSP
Faust	Term Rewriting Macro	BDA
Kronos	Type-level Computation	Value Evaluation
λ_{mmm}	Evaluation in Global Context	Evaluation of `dsp` Function

Table 1: *Comparison of the way of signal graph generation and actual signal processing between Faust, Kronos and λ_{mmm}.*

5.1. Different Behaviour Depending on the Location of Let Binding

By using functions with internal states that change over time in mimium, there is counterintuitive behavior when higher-order func-

```
    CONSTANTS[100,1,0,2]
    fn inner_then(x,freq)
    //upvalue:[local(4),local(3),local(2),
        local(1)]
    GETUPVALUE 3 2 //load filter
    MOVE       4 0
    MOVE       5 1
    GETUPVALUE 6 1 //load n
    ADDD       5 5 6
    MOVECONST  6 0
    MULF       5 5 6
    CALLCLS    3 2 1  //call filter
    GETUPVALUE 4 4 //load next
    MOVE       5 0
    MOVE       6 1
    CALLCLS    4 2 1 //call next
    ADDF       3 3 4
    RETURN     3 1

    fn inner_else(x,freq)
    MOVECONST  2 2
    RETURN     2 1

    fn filterbank(n,filter)
    MOVE      2 0 //load n
    MOVECONST 3 2 //load 0
    SUBF      2 2 3
    JMPIFNEG  2 12
    MOVE      2 1 //load filter_factory
    CALL      2 2 0 //get filter
    MOVECONST 3 1 //load itself
    MOVE      4 0 //load n
    MOVECONST 5 1 //load 1
    SUBF      4 4 5
    MOVECONST 5 2 //load inner_then
    CALLCLS   3 2 1 //recursive call
    MOVECONST 4 2 //load inner_then
    CLOSURE   4 4 //load inner_lambda
    JMP       2
    MOVECONST 4 3 //load inner_else
    CLOSURE   4 4
    CLOSE     4
    RETURN    4 1
```

Listing 6: *Compiled VM instructions filterbank example in Listing 5*

```
fn filterbank(n,filter){
  if (n>0){
    |x,freq| filter(x,freq+n*100)
    + filterbank(n-1,filter)(x,freq)
  }else{
    |x,freq| 0
  }
}
fn dsp(){
  filterbank(3,bandpass)(x,1000)
}
```

Listing 7: *Wrong example of the code that duplicate filter parametrically.*

```
fn filterbank(n,filter:&(float,float)->
    float)->&(float,float)->float{
  .< if (n>0){
    |x,freq| ~filter(x,freq+n*100)
      + ~filterbank(n-1,filter)(x,freq)
  }else{
    |x,freq| 0
  } >.
}
fn dsp(){
  ~filterbank(3,.<bandpass>.)(x,1000)
}
```

Listing 8: *Example of filterbank function using multi-stage computation in a future specification of mimium.*

tions are used compared to general functional programming languages.

Listing 7 presents an example of incorrect code that is slightly modified from the filterbank example in Listing 5. The main difference between Listing 7 and Listing 5 is whether the recursive calls in the `filterbank` function are written directly or bound using a `let` expression outside the inner function. Similarly, in the `dsp` function, which is called by the audio driver in mimium, the difference lies in whether the `filterbank` function is executed within `dsp` or bound with `let` once in the global context.

In a typical functional programming language, if none of the functions in the composition involve destructive assignments, the calculation process remains unchanged even if the variable bound by `let` is replaced with its term (via beta reduction), as seen in the transformation from Listing 7 to Listing 5.

However, in mimium, there are two distinct stages of evaluation. 0: The code is first evaluated in a global environment (where the signal-processing graph is concretized). 1: The `dsp` function is executed repeatedly (handling the actual signal processing) and may involve implicit updates to the internal states.

Although the code contains no destructive assignments, the recursive execution of the `filterbank` function occurs only once in Listing 5 during the global environment evaluation. Conversely, in Listing 7, the recursive function is executed, and a closure is generated each time the `dsp` function runs on every sample. Because the internal state of the closure is initialized at the time of closure allocation, in the example of Listing 7, the internal state of the closure is reset at each time step, following the evaluation of `filterbank`.

This implies that major compiler optimization techniques, such as constant folding and function inlining, cannot be directly applied to mimium. These optimizations must be performed after global context evaluation and before the evaluation of the `dsp` function.

To address this issue, it is necessary to introduce a distinction in the type system to indicate whether a term should be used during global context evaluation (stage 0) or actual signal processing (stage 1). This can be achieved with Multi-Stage Computation [9]. Listing 8 provides an example of the `filterbank` code using BER MetaOCaml's syntax: `.<term>.`, which generates a program to be used in the next stage, and `~term`, which embeds the terms evaluated in the previous stage [10].

The `filterbank` function is evaluated in stage 0 while embedding itself with `~`. In contrast to Faust and Kronos, this multi-stage computation code retains the same semantics for both the generation of the signal processing graph and the execution of signal processing.

5.2. A Possibility of the Foreign Stateful Function Call

The closure data structure in λ_{mmm} combines functions with the internal states, as shown in Figure 3. The fact that `filterbank` samples do not require special handling for internal states means that external signal processors (Unit Generators: UGens), such as oscillators and filters written in C or C++, can be called from mimium, just like normal closure calls. Additionally, it is possible to parameterize, duplicate, and combine external UGens[4]. This capability is difficult to implement in Faust and similar languages but is easily achievable in the λ_{mmm} paradigm.

However, mimium currently uses sample-by-sample processing and cannot handle buffer-by-buffer value passing. Because most native unit generators process data on a buffer-by-buffer basis, there are few practical cases where external UGens are currently used. Nonetheless, in λ_{mmm}, only $feed$ terms require sample-by-sample processing. Therefore, it is possible to differentiate the functions that can process only one sample at a time from those that can process concurrently at the type level. As the multi-rate specification is being considered in Faust [11], it may be possible to facilitate buffer-based processing between an external Unit Generator by having the compiler automatically determine the parts that can be processed buffer-by-buffer.

6. CONCLUSION

This paper proposed λ_{mmm}, an intermediate representation for programming languages for music and signal processing, along with a virtual machine and an instruction set to execute it. λ_{mmm} enables the description of generative signal graphs and their contents within a unified syntax and semantics. However, users are responsible for ensuring that their code does not create escapable closures during the iterative execution of a DSP, which can be challenging for novice users to grasp.

In this paper, the translation of λ_{mmm} terms into VM instructions was illustrated by showing examples of code and the corre-

[4]In fact, in the actual implementation of mimium, an interoperation between VM and audio driver is realized by passing and calling Rust's closure.

sponding expected bytecode alongside pseudocode to describe the behavior of the VM. More formal semantics and a detailed translation process should be considered, particularly with the introduction of multi-stage computation.

I hope that this research will contribute to more general representations of music and sound on digital computers and foster deeper connections between the theory of languages for music and the broader field of programming language theory.

7. ACKNOWLEDGMENTS

This study was supported by JSPS KAKENHI (Grant No. 23K12059). I would also like to thank the many anonymous reviewers.

8. REFERENCES

[1] Yann Orlarey, Dominique Fober, and Stephane Letz, "Syntactical and semantical aspects of Faust," *Soft Computing*, vol. 8, no. 9, pp. 623–632, 2004.

[2] Albert Gräf, "Term Rewriting Extension for the Faust Programming Language," in *International Linux Audio Conference*, 2010.

[3] Benedict R Gaster, Nathan Renney, and Tom Mitchell, "OUTSIDE THE BLOCK SYNDICATE: TRANSLATING FAUST'S ALGEBRA OF BLOCKS TO THE ARROWS FRAMEWORK," in *Proceedings of the 1st International Faust Conference*, 2018.

[4] Vesa Norilo, "Kronos: A Declarative Metaprogramming Language for Digital Signal Processing," *Computer Music Journal*, vol. 39, no. 4, pp. 30–48, 2015.

[5] Emilio Jesús Gallego Arias, Pierre Jouvelot, Sylvain Ribstein, and Dorian Desblancs, "The W-calculus: A Synchronous Framework for the Verified Modelling of Digital Signal Processing Algorithms," in *Proceedings of the 9th ACM SIGPLAN International Workshop on Functional Art, Music, Modelling, and Design*. 2021, vol. 12, pp. 35–46, Association for Computing Machinery.

[6] Tomoya Matsuura and Kazuhiro Jo, "Mimium: A self-extensible programming language for sound and music," in *Proceedings of the 9th ACM SIGPLAN International Workshop on Functional Art, Music, Modelling, and Design*, New York, NY, USA, Aug. 2021, FARM 2021, pp. 1–12, Association for Computing Machinery.

[7] Roberto Ierusalimschy, Luiz Henrique de Figueiredo, and Waldemar Celes, "The Implementation of Lua 5.0," *JUCS - Journal of Universal Computer Science*, vol. 11, no. 7, pp. 1159–1176, July 2005.

[8] Robert Nystrom, *Crafting Interpreters*, Genever Benning, Daryaganj Delhi, July 2021.

[9] Walid Taha and Tim Sheard, "Multi-Stage Programming with Explicit Annotations," *SIGPLAN Notices (ACM Special Interest Group on Programming Languages)*, vol. 32, no. 12, pp. 203–214, Dec. 1997.

[10] Oleg Kiselyov, "The Design and Implementation of BER MetaOCaml," in *Proceedings of the 12th International Symposium on Functional and Logic Programming*, Michael Codish and Eijiro Sumii, Eds., Cham, 2014, pp. 86–102, Springer International Publishing.

[11] Pierre Jouvelot and Yann Orlarey, "Dependent vector types for data structuring in multirate Faust," *Computer Languages, Systems & Structures*, vol. 37, no. 3, pp. 113–131, 2011.

IFC-24 Paper Session 2

WHAT'S NEW IN THE FAUST ECOSYSTEM IN 2024?

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

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

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

ABSTRACT

This paper provides an overview of the developments in the FAUST programming language and ecosystem since the 2022 International FAUST Conference. Over the past two years, the FAUST community has made significant progress in compiler enhancements, backend integrations, web-based tools, a new Widget Modulation language extension, and FPGA support for audio DSP compilation.

The Emeraude team, a collaboration between Inria, GRAME-CNCM, and INSA Lyon, started its work in March 2022 and has strengthened FAUST's development and application in academic and industrial contexts.

Additionally, a reflective process and proposed consortium aim to empower the user community in guiding FAUST's future direction.

The paper also explores several industrial applications, highlighting the practical impact and versatility of the FAUST ecosystem.

1. INTRODUCTION

The FAUST programming language and its ecosystem are key technological components used by the Emeraude team, particularly in the Syfala project, as discussed in §3.4.1.

Furthermore, a reflective process, presented in §2, has been initiated to strengthen the FAUST project, with plans to establish a FAUST consortium, aiming to provide the user community with a significant role in shaping the future of the project.

FAUST has made significant strides in compiler developments, backends integration, and community projects. Highlights in §3 include:

- New Backends: integration within JAX, JSFX, Cmajor, and RNBO, enhancing FAUST's versatility across various DSP contexts.
- Widget Modulation: enabling developers to effortlessly implement voltage control type modulation to existing Faust circuits.
- Web Developments: introduction of the `faustwasm` and `faust-web-component` packages, modernization of the FAUST IDE, Editor, and Playground for easier web-based DSP integration and update WAM 2.0 plugin model.
- FPGA Support: the Emeraude team's work on providing an audio DSP compilation flow for FPGA platforms, Linux support for Syfala, and development of multichannel audio boards.

Finally, several of the main industrial applications of the FAUST ecosystem are presented in §4.

2. THE FAUST COMMUNITY

2.1. FAUST Consortium

The FAUST project is an open-source initiative hosted on GitHub [1] and freely accessible to the public. While the community has significantly enriched the ecosystem with architecture files, libraries, and more, contributions to the language design and the compiler itself have been minimal so far.

The aim of the FAUST consortium is to give the FAUST user community a stake in the future of FAUST, by giving them the opportunity to see how the language will evolve, and to take an active part in the decision-making process, in particular the development of the roadmap.

Another goal of the FAUST consortium is to gather financial resources to ensure the maintenance and development of FAUST in the years to come.

Here's the current state of our thinking and the resulting proposals, bearing in mind that this is an ongoing endeavour that still needs some work.

2.1.1. Consortium Members

The FAUST Consortium is made up of several categories of members, according to their financial contribution to the Consortium, which determines their level of Membership, and consequently their rights in the running of the Consortium. The different categories are as follows:

- Guest member
- Paying member: platinum, gold, silver

Their rights and obligations will be defined in a "FAUST Consortium Contract" document proposed by InriaSoft [2].

2.1.2. Consortium Organization

The consortium is supported by two governing bodies:

- the Annual General Meeting (AGM)
- the Scientific and Technical Committee (STC).

2.1.3. Annual General Meeting

The General Meeting is the governing body that ensures the smooth running of the FAUST consortium. At its annual meeting:

[1] `https://github.com/grame-cncm/faust`

[2] `https://www.inria.fr/fr/inriasoft-pour-la\-diffusion-des-logiciels-open-source`

- It examines the state of the ecosystem and makes recommendations on future directions and work priorities;
- It examines the consortium's financial situation and approves the annual budget;
- It sets work priorities for the various elements of the roadmap drawn up by the Scientific and Technical Committee;
- It coordinates communication and promotional activities, such as the International FAUST Conference (IFC).

2.1.4. Scientific and Technical Committee

The Scientific and Technical Committee defines the roadmap for the evolution of the compiler and the various tools that make up its ecosystem:

- It is responsible for the official specification of the FAUST language and its evolution;
- It proposes a reference implementation of the compiler, compliant with this specification, and regularly publishes this implementation officially;
- It issues certificates of conformity for any third-party implementations of the compiler to this specification;
- It defines and maintains the standard FAUST libraries, as well as various development tools that are part of the FAUST ecosystem;
- It maintains a set of basic architecture files;
- It develops the language's official documentation and teaching resources;
- It manages the language's official websites and related Git repositories.

The STC is made up of a technical manager appointed by Inria, members of the Emeraude team working on FAUST, and possibly one or two representatives of the members. Consortium members are invited to suggest topics for the agenda of STC meetings.

2.1.5. Drawing Up the Roadmap

The roadmap defines short and medium-term developments for the language, the compiler and the various tools in the FAUST ecosystem. A first, fairly broad version is drawn up by the STC, based on proposals from the FAUST community and consortium members. In particular, the STC assesses the feasibility and labor costs of the various points, and proposes an initial ranking according to their importance and dependence.

The General Assembly selects and prioritizes the items to be included in the roadmap from the STC's proposals. It ensures that the necessary workload does not exceed 50% of the consortium's available human resources. The roadmap is then officially adopted by the General Assembly in a vote in which each member has a number of votes corresponding to its level of membership.

2.2. Communication Channels

The now automomous FAUST Discord channel [3] is an active and dynamic online community space dedicated to users, developers. This platform serves as a hub for real-time communication, collaboration, and support, fostering a sense of community among members with varying levels of expertise (see Figure 1).

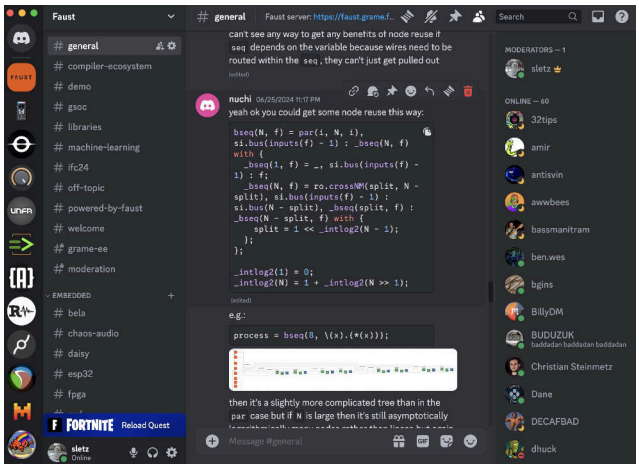


Figure 1: *The Faust Discord channel.*

2.3. The "Powered by FAUST" Page

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

This page is regularly enriched and as of July 2024, more than 250 projects are described (see Figure 2).

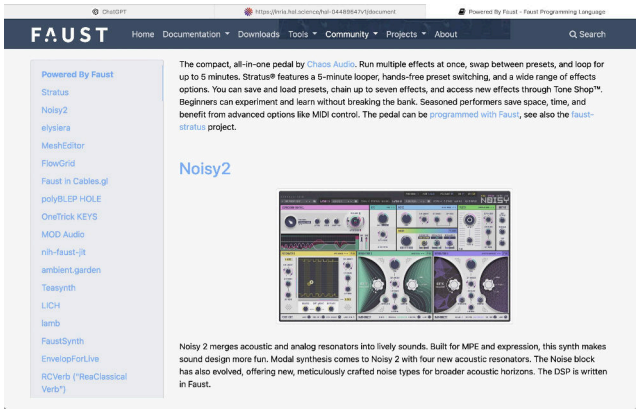


Figure 2: *Excerpt of the "Powered by Faust" list.*

3. DEVELOPMENTS

3.1. New Backends

Four new backends have been developed. They allow us to use FAUST DSP programs with a larger set of targets in new applications, or to reach new communities.

[3] https://faust.grame.fr/community/help/#faust-on-discord

3.1.1. JAX

The introduction of the JAX backend opens up a new domain of exploration for FAUST, significantly expanding its capabilities and potential applications, especially in the field of machine learning.

JAX is a library for high-performance numerical computing, particularly popular in the machine learning research community for its flexibility in model development and its capability to handle complex mathematical operations efficiently. It extends the capabilities of NumPy by enabling automatic differentiation, allowing us to compute gradients of functions with respect to their inputs. JAX also supports just-in-time (JIT) compilation to highly optimized machine code, which significantly improves performance for numerical routines. It supports the creation and training of neural networks through libraries built on top of it, like Flax and Haiku.

With the assistance of GRAME, David Braun has contributed a JAX backend allowing for the direct creation of differentiable FAUST programs for potential uses in machine learning applications. It is designed to be used within DawDreamer [4], an audio-processing Python framework that supports core DAW features and more, also developed by David Braun.

Several Flax examples with a learnable low-pass filter, a differentiable subtractive synthesizer, a differentiable polyphonic wavetable synthesizer whose wavetables are learnable, as well as a parametric equalizer written in FAUST to a QDax environment, have been explored [5].

3.1.2. JSFX

Developed by Cockos, the creators of Reaper, JSFX [6] is a scripting language which allows users to extend the capabilities of the DAW with custom audio and MIDI processing scripts adapted to their specific needs.

With the assistance of GRAME, Johann Philippe has contributed a backend enabling the creation of synthesizers and effects with MIDI control, as well as polyphonic MIDI-controllable audio plugins. Following the standard JSFX file structure, several `@init`, `@block`, `@slider` and `@sample` blocks are filled with the appropriate part of the generated FAUST code. The result is self-contained, with the architecture part directly inserted by the backend (even the voices allocation and MIDI control in polyphonic mode), allowing it to be loaded and executed in Reaper seamlessly. A comprehensive tutorial has been written. [7]

3.1.3. Cmajor

Cmajor is a C like procedural high-performance language specifically designed for audio processing providing a runtime with dynamic LLVM JIT based compilation. Cmajor is intended to complete other technologies like C++, JUCE, and CLAP, and also support Web export. DSP is deployed with Cmajor as a patch, which includes a description of the plugin, the source code for the DSP, and an associated GUI implemented in JavaScript. The language supports a signal flow through a graph structure with nodes containing implementations of specific DSP building blocks.

A Cmajor backend has been written to generate a processor from a FAUST DSP program. Parameters such as sliders, buttons, and bar graphs correspond to Cmajor's concept of input and output events. The actual sample computation code is generated within the essential `run()` function.

A `faust2cmajor` script enables the creation of ready-to-use Cmajor patches, which can be directly executed with the `cmaj` runtime, or possibly exported as C++, or JUCE, CLAP, Web plugins. The script supports both regular DSP programs and polyphonic MIDI-controllable programs. A comprehensive tutorial has been written. [8]

3.1.4. RNBO

RNBO is a library and toolchain that can take Max-like patches, export them as portable code, and directly compile that code to targets like a VST, a Max External, or a Raspberry Pi.

FAUST programs can be compiled to the internal `codebox~` sample level scripting language. In this model, several sections are generated for parameter definitions, DSP state construction, initialization, the `control` function (called once per block), and finally, the `compute` function (called at audio rate).

The `faust2rnbo` tool transforms a FAUST DSP program into a RNBO patch containing a `rnbo~` object and including the codebox code (generated using the codebox backend) as a subpatch. Needed audio inputs/outputs and parameters (with the proper name, default, min and max values) are automatically added in the patch. Additional options allow us to generate a specific version of the RNBO patch used in the testing infrastructure. The code is written in Python and uses the very powerful `py2max` [9] library to generate the maxpat JSON format.

The script supports both regular DSP programs and polyphonic MIDI-controllable programs. A comprehensive tutorial has been written. [10]

3.2. Widget Modulation

An extension to the FAUST programming language, *Widget Modulation*, has been recently developed. Inspired by *Modular Synthesizer*, this extension enables developers to effortlessly implement *voltage control type* modulation to existing Faust circuits.

Although signal modulation can traditionally be achieved by writing the necessary code during circuit development, *Widget Modulation* expressions enable it *a posteriori*, after the circuit has been developed and without modifying its code. This feature allows for direct *reuse and customization* without prior planning by the original circuit designer. It offers a new level of expressivity and flexibility in FAUST circuit design. A separate paper on the subject has been proposed to IFC 2024 [7].

3.3. Web-Related Developments

The development of the FAUST to Web glue code started in 2014, initially as a collection of JavaScript files. To streamline maintenance and facilitate future development, the compiler's C++ export layer has been modernized using the Embind model, [11] which is

[4] `https://github.com/DBraun/DawDreamer`

[5] `https://github.com/DBraun/DawDreamer/tree/main/examples`

[6] `https://www.reaper.fm/sdk/js/js.php`

[7] `https://faustdoc.grame.fr/tutorials/jsfx/`

[8] `https://faustdoc.grame.fr/tutorials/cmajor/`

[9] `https://github.com/shakfu/py2max`

[10] `https://faustdoc.grame.fr/tutorials/rnbo/`

[11] `https://emscripten.org/docs/porting/connecting_cpp_and_javascript/embind.html`

part of the Emscripten compiler.[12] Additionally, the FAUST to WebAudio glue code has been completely restructured and rewritten in TypeScript, developed and distributed as a separated npm package. Several projects have been developed using this new framework, demonstrating its robust performance and ease of integration.

3.3.1. The faustwasm Package

The FaustWasm library [13] provides a user-friendly high-level API that wraps around the FAUST compiler. While the interface is primarily tailored for TypeScript, it also includes API descriptions and documentation for pure JavaScript users. This WebAssembly version of the FAUST Compiler, suitable for both Node.js and web browsers, has been compiled with Emscripten 3.1.31.

The library enables the compilation of FAUST DSP code into WebAssembly, allowing it to be used as WebAudio nodes within a standard WebAudio node graph. It also supports offline rendering scenarios. Additionally, tools are available for generating SVGs from FAUST DSP programs.

Users can create "mono" synthesizer and effect nodes, as well as polyphonic nodes. MIDI support is automatically activated when MIDI metadata is included in the DSP code for mono nodes, and is always enabled in polyphonic mode.

Sensors (accelerometer and gyroscope) are supported, as well as the Progressive Web Application model, so playable instruments to be used on smartphones and tablets can be easily deployed.

3.3.2. Modernized Faust IDE, Faust Editor and FaustPlayground

As the main outcomes of Ian Clester's Google Summer of Code 2023 projects, [14] the three FAUST IDE, FAUST Editor and FaustPlayground projects have been modernized with updated build tools, and the use of the `faustwasm` package.

3.3.3. faust-web-component

Another outcome of Ian Clester's Google Summer of Code project is the `faust-web-component` [15] package which provides two web components for embedding interactive FAUST snippets in web pages:

- `<faust-editor>` displays an editor (using CodeMirror 6) with executable, editable FAUST code, along with some bells & whistles (controls, block diagram, plots) in a side pane. This component is ideal for demonstrating some code in FAUST and allowing the reader to try it out without having to leave the page.
- `<faust-widget>` just shows the controls and does not allow editing, so it serves simply as a way to embed interactive DSP, and can been tested here.

These components are built on top of `faustwasm` and `faust-ui` [16] packages and are released as an npm package.

3.3.4. Web Audio Module (WAM) 2.0

In 2015, Jari Kleimola and Olivier Larkin proposed Web Audio Modules (WAM), a standard for Web Audio plugins and DAWs. The 2.0 version [1], released in 2021, was a collaborative effort involving many contributors, resulting in multiple open source and free software plugins and hosts. WAM 2.0 includes an SDK, an abstract API, numerous open source repositories with various plugins, tutorials, and several hosts demonstrating WAM capabilities. The design of WAM 2.0 aimed to support diverse development workflows, from web developers using plain JavaScript, React developers, to C/C++ developers cross-compiling code to WebAssembly.

WAM 2.0 [2] plugins can be developed using FAUST and easily generated using the FAUST IDE [17], with the adapted targets [18].

3.4. Emeraude Team Projects

The Emeraude team is continuing its work on the FAST ANR project [19], initiated in 2021. This project aims to facilitate high-level programming of FPGA-based platforms for multichannel ultra-low-latency audio processing using FAUST.

3.4.1. Syfala: Compilation of Audio DSP on FPGA

The team has been actively extending the Syfala toolchain, first released in 2022 [3]. It is meant to be a powerful audio to FPGA compilation toolchain. All the possible use of the compilation toolchain have been combined in a single software suite. This section describes the extensions that have been added to Syfala in 2023.

When compiling FAUST programs to FPGA, Syfala relies on the High Level Synthesis (HLS) tool provided by Xilinx, which takes a C++ program as an input. Hence, FAUST generates C++ code from a FAUST program and Syfala feeds it to HLS. The topology of the C++ code provided to HLS has a huge impact on the performances of the generated Intellectual Property (IP). In 2023, a study has been conducted aiming at understanding the kind of optimizations that can be carried out on C++ code in the context of the high-level synthesis of real-time audio DSP programs.

Thanks to this work, the applications generated by Syfala have been significantly optimized, allowing for much more complex audio DSP algorithms to be run on the FPGA. While these findings haven't been integrated to the FAUST Syfala backend, they can be used with the new Syfala C++ support. Indeed, a new mode in Syfala allowing for C++ code to be used as a substitute for FAUST has been added. This, combined with an exhaustive public documentation of the aforementioned optimizations will help increasing the attractivity of Syfala.

3.4.2. Linux Support for Syfala

Most modern FPGA boards host a CPU SoC tightly coupled to the FPGA. Real-time audio DSP applications running on such boards can leverage the CPU of the board to carry out control computations or to provide high-level functionalities (i.e., user interface,

[12] `https://emscripten.org/index.html`
[13] `https://github.com/grame-cncm/faustwasm`
[14] `https://ijc8.me/2023/08/27/gsoc-faust/`
[15] `https://github.com/grame-cncm/faust-web-component`
[16] `https://github.com/Fr0stbyteR/faust-ui`
[17] `http://www.webaudiomodules.com/docs/usage/generate-with-faustide`
[18] `https://faustdoc.grame.fr/manual/deploying/#exporting-wam-20-plugins`
[19] `https://fast.grame.fr`

external controllers, etc.) [3]. Up to now, the CPU portion of applications generated by Syfala was implemented as a bare-metal kernel. In 2023, the possibility to run Alpine Linux on the CPU of the Zybo board while carrying out audio DSP operations on the FPGA has been added, taking a hardware accelerator approach. This enables the compilation of complete audio applications involving various control protocols and approaches such as OSC (Open Sound Control) through Ethernet or Wi-Fi, MIDI, web interfaces running on an HTTPD server, etc. It also opens the door to the integration of hardware accelerators in high-level computer music programming environments such as Pure Data, SuperCollider, etc.

This work led to a publication at the 2023 Sound and Music Computing conference (SMC-23)[4].

3.4.3. Syfala PipeWire Support

During the work on applications for Syfala requiring the handling of a large number of audio channels in parallel for spatial audio, a way to send and receive audio streams in parallel between a laptop computer and our FPGA board has been developed. For this, an open standard named PipeWire, which allows for the transmission of digital audio streams in real-time over an ethernet connection has been chosen. PipeWire was implemented in the Linux layer of Syfala and is now perfectly integrated to the toolchain. It will allow us to significantly expand the scope of the various spatial audio systems that has been working on in the context of the PLASMA project.

3.4.4. Multichannel Audio Boards for FPGA

Two audio FPGA sister boards aiming various kinds of spatial audio applications have been developed:

- One targets the Digilent Zybo Z7 (10 or 20) board and is designed to be cost-efficient, accessible, and easily reproducible. It provides 32 amplified (3W) audio outputs to which small speakers can be directly connected. Its goal is to provide an affordable way to deal with a large number of audio outputs in the context of spatial audio.
- The other board that has been developed is meant to be connected to a Digilent Genesys board and targets high-end spatial audio applications with a strong focus on active control. It provides 32 ultra-low latency (10us) balanced inputs and outputs. It is currently used as part of the FAST ANR project for implementing FxLMS algorithms for active control.

This work has been published at the 2024 at the Sound and Music Computing conference (SMC-24) [5].

3.4.5. FAUST *to VHDL Backend*

Syfala uses HLS for compiling C++ code down to VHDL, the C++ code being itself generated from FAUST . However, FAUST , as a functional language, exhibits all the parallelism of the audio application. The code is sequentialized in the C++ code and then re-parallelized by the viti_hls tool for the FPGA.

An interesting alternative is to translate directly FAUST down to VHDL. FAUST programs can be represented as audio circuits connected together and hence provides a natural equivalence with VHDL structural representation of such circuits. The VHDL program is just a translation of the data-flow graph of the audio application.

However, for an efficient implementation on FPGA, this data-flow graph must be retimed. Retiming is an old classical transformation that adds registers in a digital circuit without changing its functional behaviour but allowing for a much faster clock rate.

A first `Faust2VHDL` translator prototype was issued in 2022 generating a fully combinatorial data path on the FPGA. In 2023 the first real `Faust2FPGA` compiler which includes retiming and fixed point computations has been released.

Preliminary results shows that the IP generated by our `Faust2FPGA` compiler are twice smaller than the IP generated by viti_hls. However, the use of HLS is still preferred because many features are not included in the `Faust2FPGA` compiler (i.e., control from the ARM processor or use of the external RAM).

3.4.6. Fixed-Point Extension for the FAUST *Programming Language*

This recent paper [6] addresses the challenge of efficiently utilizing fixed-point arithmetic in FAUST. Instead of the standard floats format, fixed-point arithmetic can be more resource-efficient and faster than floating-point arithmetic, particularly on FPGAs where the required circuitry can be precisely configured. However, it implies the careful determination of number formats at each step of the computation tree.

The need to reconsider the representation of real numbers in this context is highlighted, where fixed-point numbers, represented as scaled integers, can offer significant efficiency improvements. The introduced key concept is "pseudo-injectivity" which ensures that output values of each function in the language retain the necessary precision. The method extends the previously existing interval range analysis to determine the range of values variables can take and error analysis to manage rounding and ensure precision.

Enhancements to the FAUST compiler to facilitate automatic fixed-point format determination have been done. The precision constraints are propagated through the signal graph to maintain pseudo-injectivity. Additionally, when generating C++ code, a `sfx_t` macro is added in the generated code at each step of the computation, to represent fixed-point formats with the most significant bit (MSB) and the least significant bit (LSB) values.

Results from testing on FAUST programs, such as sine wave generation and the Karplus-Strong string synthesis algorithm, indicate that the method can maintain high audio quality, though inferred formats tend to be wider than necessary. Future improvements will include backward propagation of precision constraints and targeted optimizations to further refine the fixed-point format determination.

3.5. PLASMA: Pushing the Limits of Audio Spatialization with eMerging Architectures

Plasma (Pushing the Limits of Audio Spatialization with eMerging Architectures) is an associate research team gathering the strength of Emeraude and of the Center for Computer Research in Music and Acoustics (CCRMA) at Stanford University. [20]
The two main objectives of Plasma are:

[20] `https://team.inria.fr/emeraude/plasma/`

- Exploring various approaches based on embedded systems towards the implementation of modular audio signal processing systems involving a large number of output channels (and hence speakers) in the context of spatial audio.
- Making these systems easily programmable to create an open and accessible system for spatial audio where the number of output channels is not an issue anymore.

Two approaches are being considered in parallel:

- Distributed using cheap simple embedded audio systems (i.e., Teensy, etc.),
- Centralized using an FPGA-based (Field-Programmable Gate Array) solution based on the multichannel interfaces presented in §3.4.4.

The focus is on enhancing the hardware and computational capabilities of current spatial audio systems, rather than on the DSP algorithms for spatial audio themselves. FAUST plays a central role in this project by allowing us to deploy spatial audio and virtual acoustics programs from the same source in a generic way.

4. INDUSTRIAL APPLICATIONS

Here is a non-exhaustive list of some recent industrial applications of FAUST.

4.1. Expressive E

Expressive E [21] is a French company that creates innovative musical instruments and software designed to enhance expressive performance. Their products include the Osmose, a standalone expressive synthesizer, and Touché, a device that adds tactile control to existing synthesizers.

They also offer a range of software instruments and sound libraries, such as Noisy and Imagine, which are especially designed to be used with the Osmose and Touché devices, but are versatile enough be used with other MIDI controllers and digital audio workstations (DAWs).

Noisy 1 and 2 products were largely created using FAUST, and benefited from a close collaboration between Expressive E's development team and GRAME, in particular in developing performance measurement and optimisation tools.

Figure 3: *The Noisy2 plugin interface.*

[21] https://www.expressivee.com

4.2. Punk Labs

Punk Labs LLC is a tiny studio of just two people creating apps, games, music, and even social networks. They develop for desktops and game consoles, mobile and embedded devices. Four plugins have been developed using the Rust NIH-plug framework and FAUST for DSP[22], which allows us to develop and export VST3 and CLAP format, as well as a standalone module:

- OneTrick KEYS: a physically modeled piano synth with a lo-fi sound.
- OneTrick URCHIN: an hybrid drum synth that models the gritty lo-fi sound of beats from vintage records without sampling.
- OneTrick CRYPTID: whispers of a drum machine with the cold clanging heart of a DX7 in the fearsome frame of a TR-808 echo in dusty backrooms of backstreet recording studios.
- OneTrick SIMIAN: crash into the 80s with an open source drum synth inspired by hexagonal classics like the Simmons SDS-V.

4.3. Joué Play

The Joué Play [23] is a system that combines an expressive multi-instrument, an intuitive app and interactive content, with a range of musical instruments that use touch-sensitive technology to create a unique playing experience. These instruments are designed to be highly expressive, allowing musicians to play with greater nuance and emotion. Part of the audio effects are coded in FAUST.

5. ACKNOWLEDGMENT AND CONCLUSION

This paper reflects the richness and diversity of the contributions carried out during the past two years. The significant advancements detailed herein are the result of the collaborative efforts of numerous individuals and teams.

Thanks to all contributors for all the different components and projects that have been described, and especially this time: Johann Philippe, David Braun, Shihong Ren, Michel Buffa, Ian Clester, and the Emeraude team: Tanguy Risset, Romain Michon, Pierre Cochard, Florent de Dinechin, Anastasia Volkova, Thomas Rushton, Maxime Popov and Agathe Herrou.

6. REFERENCES

[1] Michel Buffa, Shihong Ren, Owell Campbell, Tom Burns, Steven Yi, Jari Kleimola, and Oliver Larkin, "Web Audio Modules 2.0: An Open Web Audio Plugin Standard," in *Companion Proceedings of the Web Conference 2022 (WWW 22)*, Lyon, France, April 2022.

[2] Shihong Ren, Stéphane Letz, Yann Orlarey, Dominique Fober, Romain Michon, Michel Buffa, and Laurent Pottier, "Modernized Toolchains to Create JSPatcher Objects and WebAudioModules from Faust Code," in *Proceedings of the Web Audio Conference (WAC-22)*, Cannes, France, July 2022.

[22] https://github.com/robbert-vdh/nih-plug
[23] https://jouemusic.com/en

[3] Maxime Popoff, Romain Michon, Tanguy Risset, Yann Orlarey, and Stéphane Letz, "Towards an FPGA-Based Compilation Flow for Ultra-Low Latency Audio Signal Processing," in *Proceedings of the Sound and Music Computing Conference (SMC-22)*, Saint Etienne, France, June 2024.

[4] Pierre Cochard, Maxime Popoff, Antoine Fraboulet, Tanguy Risset, Stéphane Letz, and Romain Michon, "A Programmable Linux-Based FPGA Platform for Audio DSP," in *Proceedings of the Sound and Music Computing Conference*, Stockholm, Sweden, 2023.

[5] Maxime Popoff, Romain Michon, and Tanguy Risset, "Enabling Affordable and Scalable Audio Spatialization With Multichannel Audio Expansion Boards for FPGA," in *Proceedings of the 2024 Sound and Music Computing Conference*, Porto, Portugal, July 2024.

[6] Agathe Herrou, Florent de Dinechin, Stéphane Letz, Yann Orlarey, and Anastasia Volkova, "Towards Fixed-Point Formats Determination for Faust Programs," in *Proceedings of the Journées d'Informatique Musicale (JIM-24)*, Marseille, France, May 2024.

[7] Yann Orlarey, Stéphane Letz, Romain Michon, and the Emeraude team, "Widget Modulation in Faust," in *Proceedings of the International Faust Conference (IFC-24)*, Turin, Italy, November 2024.

PHAUSTO: EMBEDDING THE FAUST COMPILER IN THE PHARO WORLD

Domenico Cipriani *

Pharo Association - Lille, France
mspgate@gmail.com

Alessandro Anatrini

Hochschule für Musik und Theater (HfMT), Hamburg - Germany
Conservatorio Statale di Musica J. Tomadini - Udine, Italia
alessandro@anatrini.com

Sebastian Jordan Montaño

Univ. Lille, Inria, CNRS, Centrale Lille, UMR 9189 CRIStAL F-59000 Lille, France
sebastian.jordan@inria.fr

ABSTRACT

Phausto is a lightweight, open-source library for live-coding music, enabling sound generation and Digital Signal Processing (DSP) programming. Developed in the Pharo programming language, it incorporates the Faust compiler for robust audio capabilities, using Foreign Function Interface (FFI) calls for seamless integration. Phausto connects with platform-specific audio layers through PortAudio, offering a consistent API across operating systems. Designed for educational settings, it targets users interested in DSP, musicians, and sound artists with limited technical skills. Phausto addresses two main challenges: generating audio in Pharo applications and providing an accessible environment for programming digital musical instruments. It is easy to install and supports the latest Pharo versions, with instructions available on its GitHub repository.

1. INTRODUCTION

Phausto is a library for live-coding music. It enables sound generation and DSP (Digital Signal Processing) programming. Phausto is free and open source, lightweight (only 10 MB including the Faust libraries), and an accessible tool developed primarily to provide developers with a fast and easy way to integrate sound into their programs and applications. Phausto is well-suited for educational use, particularly in environments that emphasize hands-on, exploratory learning. It targets users interested in learning DSP programming, musicians, and sound artists with limited computer proficiency who want to learn about computer music.

Phausto is implemented in the Pharo programming language. It has an embedded Faust [3] compiler for producing the sound. We chose Faust because it offers incredible audio-programming capabilities [2]. The interaction between Pharo and Faust is enabled through Foreign Function Interface (FFI) calls to a dynamic library allowing seamless integration with Faust libraries and the Box-API. Phausto also manages the connections to platform-specific audio layers via PortAudio, a cross-platform audio library that provides a consistent API for audio input and output across different operating systems. From educational and artistic perspectives, Phausto aims to serve as a functional alternative to Faust and as an introductory tool for users needing more advanced DSP or custom solutions.

Phausto is easy to install. It operates on the latest stable version of Pharo, ensuring backward compatibility up to Pharo 10. Detailed installation instructions for Phausto can be found in the Phausto GitHub repository: `https://github.com/lucretiomsp/phausto`.

* This work was supported by the Pharo Association

The Phausto Library addresses two key challenges:

1. generating audio in Pharo applications;
2. providing an accessible environment for sound artists with limited computer literacy to program digital musical instruments (DMIs).

2. THE PHARO PROGRAMMING LANGUAGE

We chose Pharo as our implementation platform because it has an easy-to-read-and-learn syntax with only seven reserved words. Pharo [4, 5] is a pure object-oriented programming language that is dynamically typed. Pharo is a modern implementation of Smalltalk [6, 1] that started in 2008. It is multi-platform and has a vibrant community worldwide, welcoming coders of all experience levels.

Its simple syntax makes Pharo resemble a pidgin language[1] [7]. Pharo is also an integrated development environment (IDE) that offers a live coding environment where programmers can modify their code during execution. At the same time, GUI widgets can be opened or easily created and used in real-time development.

3. INSIDE PHAUSTO

The communication between Faust and Pharo depends on three technologies: the Faust dynamic engine, Pharo's unified Foreign Function Interface (uFFI) , and Faust Box API. Figure 1 provides an overview of the Phausto's architecture. In the following section we will discuss the implementations detail of Phausto. These implementation details require an advanced understanding of Pharo and Faust, which is only relevant within the context of this paper. The final user of Phausto does not need to know about the details of how Unit Generators are initialised and converted to `PhBox`. To combine Unit Generators and create DSP, users only need to know how to set their instance variables and how they can be patched together. All this information is contained in the class comments. This concept of *abstraction* is fundamental within the object oriented paradigm.

[1] In computing, a "pidgin language" refers to a programming language with a simplified syntax and minimalistic design, akin to a pidgin language in linguistics. Just as a pidgin language simplifies communication between speakers of different native languages, a pidgin programming language simplifies code writing and reading by reducing complexity and removing extraneous features. This approach aims to make the language easier to learn and use.

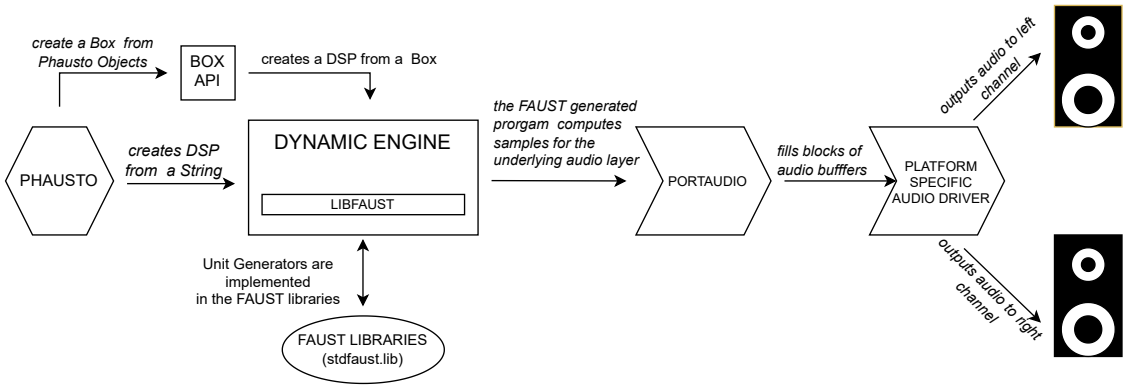


Figure 1: *A simplified diagram illustrating Phausto's framework architecture. The dynamic engine libraries use* libfaust *and the* Faust libraries *to transform into DSP programs the strings of code or the combinations of boxes written inside Pharo. The computed samples feeds the PortAudio stream that is finally rendered into sound by the platform specific audio driver.*

3.1. The Dynamic Engine

The dynamic engine is a Faust DSP architecture developed by Stéphane Letz [2]. Its C API details how to create Faust objects, initialize them with a sample rate and a buffer size and start and stop their operation. This dynamic engine can be packaged with an interpreter backend and a basic *WaveReader* for reading audio files, instead of the default LLVM compiler backend. To meet our design goals, we opted for the interpreter backend and the WaveReader due to their lack of external dependencies. Additionally, we selected PortAudio for its cross-platform compatibility. While we acknowledge that the interpreter backend is slower compared to the LLVM compiler, we do not anticipate this will impact Phausto's target audience.

New DSP objects can be instantiated from a string containing a Faust program using the following function:

```
dsp* createDsp(const char* name_app, const
    char* dsp_content, int argc, const char*
     argv[], const char* target, int
    opt_level);
```

This function is called sending the `create: aString` message to the DSP class. It can be considered the easiest way for a Faust programmer to create DSP in Phausto and it was our first choice to test the functioning of our framework.

Alternatively, we can create a DSP object from a box or a combination of boxes[3]. This approach is more flexible because it allows the user define the connections between boxes in Phausto code, taking advantage of Pharo syntax highlighting. The following C function is called when the `asDsp` message is sent to a `PhBox`[4]

```
dsp* createDspFromBoxes(const char*
    name_app, Box box, int argc, const char*
     argv[], const char* target, int
    opt_level);
```

Both methods return a pointer to the created DSP objects on success, or a null pointer on a failure; if a failure occurs, a call to:

```
{const char* getLastError();
```

returns the error.

3.2. Pharo Unified Foreign Function Interface (UFFI)

A Foreign Function Interface (FFI) is a programming mechanism that enables the use of functions and data structures written and compiled in a different language [8]. Typically these "foreign" resources are shared libraries, such as `.dll`, `.so`, and `.dylib` files on Windows, Linux, and macOS respectively.

The Pharo uFFI API framework allows us to use implementations written in `faust-dynamic-engine.h`, in `libfaust-c.h` and in `libfaust-box-c.h`. We have implemented all functions from `faust-dynamic-engine.h` as FFi calls in Pharo. Below is an example of the FFI call used to create a DSP from boxes:

[2] `https://github.com/grame-cncm/faust/blob/master-dev/architecture/faust/dsp/faust-dynamic-engine.h`

[3] A Box is an intermediate representation of a Faust primitive, or of a DSP. Boxes expression can be created and combined though a C/C++ . Boxes enable precise and granular manipulation of DSPs (Digital Signal Processing units) through our higher-level Phausto API, allowing for detailed tuning and adjustment of their operational parameters in Pharo code.

[4] The `PhBox` class is a subclass of Pharo `FFIOpaqueObject` that is essentially a pointer to a Faust box.

```
PhaustoDynamicEngine >> #createDspFromBoxes
    : aFaustBox

    ^ self ffiCall:
    #( DSP * createDspFromBoxes #( const
        char * 'MyApp', #PhBox * aFaustBox,
        int 0, void * 0, const char * ' ',
        int -1 ) )
```

3.3. The Box API

The Faust Box API serves as an intermediate public entry point in the *Semantic Phase*[5] of Faust's compilation process. It facilitates the programmatic construction of a box expression, which is subsequently used to instantiate a DSP object. Boxes can be created by invoking a specific function defined in `libfaust-box-c.h`. For example, to create a Checkbox:

```
Box CboxCheckbox(const char * label);
```

Or from a string containing a Faust program:

```
Box CDSPToBoxes(const char* name_app, const
     char* dsp_content, int argc, const char
    * argv[], int* inputs, int* outputs,
    char* error_msg);
```

In Phausto, we have created a subclass of `FFILibrary` named `BoxAPI` to handle all the bindings to `libfaust-box-c.h`. The `BoxAPI` provides the interface to the Faust Box API, which, in turn, enables us to define a custom API for constructing DSP objects. This design allows Pharo users to develop their own DSP without needing to understand or use Faust syntax directly.

Currently 45 functions from the Faust Box-API have been implemented as Pharo methods. This includes the five binary composition operations, most of the C-equivalent primitives, the *Wire* and the *Cut* boxes, as well as all UI primitives. The following example demonstrates the Pharo implementation of the FFI call to the `CboxHSlider` function:

```
BoxAPI >> #boxHslider: aLabel init: initBox
     min: minBox max: maxBox step: stepBox

    self createLibContext.

    self ffiCall:
    #( #PhBox * CboxHSlider #( const char *
         aLabel , #PhBox * initBox, #PhBox *
         minBox , #PhBox * maxBox , #PhBox *
         stepBox ) )
```

All functions from `libfaust-box-c.h` are implemented in Pharo as methods (FFI calls) within the `BoxAPI` class, and are available to use by instances of the PhBox class (see next subsection). Each method in the `BoxAPI` class first invokes `createLibContext` method to ensure that a global compilation context exists; if not, `createLibContext` will create one. This compilation context will be automatically destroyed when the `asDsp` message is sent to a Phausto Box.

[5]The Semantic Phase is the initial step in the Faust compilation chain and consists of multiple stages. It takes Faust code as input and produces a list of signals in Normal Form as output. This list of signals in Normal Form is then passed to the Code Generation Phase, which compiles it into imperative code (C++, LLVM IR, WebAssembly, etc.

3.3.1. Integrating Unit Generators with Phausto

The concept of Unit Generators was first introduced by Max Mathews and Johan E. Miller for the Music III program in 1960 [9]. These components serve as the foundational building blocks of signal processing algorithms. Essentially, Unit Generators are subroutines that generate an output signal and may also process an input signal. Each Unit Generator is designed to perform a specific task, such as producing sound waves, applying filters, or controlling audio parameters. They function as modular elements within a synthesis framework.

As a functional language, Faust does not use any hierarchical organisation of Unit Generators, which is possible in object-oriented languages like Pharo, ChucK or SuperCollider through inheritance and abstraction. At the same time, Faust provides hundreds of DSP functions for synthesis and audio processing within the Faust Libraries, which yield the same output of our Unit Generators. The `UnitGenerator` class is a subclass of the `PhBox` class. Instances of the PhBox class are FFIOpaqueObjects, which correspond to pointers to Faust Boxes. UnitGenerators exists only in the Pharo environment and become PhBoxes when they receive the `asBox` message. To understand this mechanism, consider a simplified implementation of the `LFOTriPos` class, corresponding to Faust's `os.lf_trianglepos`. First, let's look at its initialise method:

```
LFOTriPos >> #initialize

faustCode := 'import("stdfaust.lib");
    process = os.lf_squarewavepos;'

freq := PhHSlider new
    label: self label , 'Freq'
    init: 440
    min: 1
    max: 2000
    step: 0.001.

amount := PhHSlider new
    label: self label , 'Amount'
    init: 1
    min: 0
    max: 28000
    step: 1.

offset := PhHSlider new
    label: self label , 'Offset'
    init: 0
    min: 0
    max: 800
    step: 0.01.
```

Next, the `asBox` method converts the UnitGenerator into an instance of its superclass, PhBox. :

```
LFOTriPos >> #asBox

    | intermediateBox finalBox |

    BoxAPI uniqueInstance createLibContext.
    intermediateBox := BoxAPI
        uniqueInstance
        boxFromString: self faustCode
        inputs: self inputs
        outputs: self outputs
        buffer: self errorBuffer.
    finalBox := freq asBox connectTo:
        intermediateBox.
    ^ finalBox * self amount asBox + self
        offset asBox
```

This final steps is fundamental as it enables the creation of a DSP object from a combination of boxes. Instances of the `PhBox` class serve as our entry point into the Faust compilation chain of the backend interpreter.

The Phausto API converts the majority of Faust libraries into Unit Generators. This design choice aligns with the object-oriented principle of inheritance while preventing the bloat of a single class with hundreds of methods.[6] The organisation of these units into subclasses and their interconnection capabilities are heavily inspired by the design principles of the ChucK programming language [10]. Indeed, in ChucK, there is no distinction between Unit Generators that operate at audio rate and those operating at control rate [11].

All Phausto Unit Generators are provided with initialised instance variables for the parameters specified in the corresponding Faust function. When possible (i.e. the argument is neither a constant value nor another function) they are initialised to a Faust `hslider` or `button` primitive, enabling both the parameter control and the on-the-fly creation of UI elements for the given parameter. If the Unit Generator's label has not been changed via the `label:` message, all UI elements use the class name as prefix. For example the `PulseOsc` has two controls: `'PulseOscFreq'` for its frequency and `'PulseOscDuty'` for its duty cycle.

4. SYNTAX IN A NUTSHELL

To create a DSP in Phausto, simply send the `asDsp` message to a Unit Generator or a combination of them. The `stereo` message converts it into a stereo DSP. Next, the DSP must be initialised and started to produce sound. A slider can be opened in the Pharo window to control a parameter of the DSP, and finally, the sound can be stopped.

```
sqr := SquareOsc new.
dsp := sqr stereo asDsp.
dsp init.
dsp start.
dsp openSliderFor: 'SquareOscFreq'.
dsp stop.
```

[6] Integrating over 200 methods directly into the DSP class would have been impractical. Instead, we organised these methods into Unit Generators, which enhances modularity and readability. Similarly, the Faust programming language efficiently manages large numbers of functions by structuring them within environments.

At need, we can combine UnitGenerators by assign them to instance variables, and we can change the value of a parameter with the keyword message `setValue: parameter:`:

```
pulse := PulseOsc new duty: LFOTriPos new.
dsp := pulse stereo asDsp.
dsp init.
dsp start.
dsp setValue: (Random new
    nextIntegerBetween: 50 and: 800)
    parameter: 'PulseOscFreq'
dsp stop.
```

The binary operator `=>` from the ChucK programming language was adopted to simplify the connection between UnitGenerators. This approach abstracts the connections while adhering to the principles of modular synthesis patching.For example a simple synth could look like this:

```
synth := SawOsc new => ADSREnv new =>
    ResonLp new => SatRev new;
```

Due to the double dispatch mechanism in the Phausto implementation of the `=>` message, its meaning depends on the argument provided. If the argument is an envelope, it performs signal multiplication; if it is a filter or a reverb, it connects the input(s) of a Unit Generator or a combination of them.

4.1. Dynamic Control of DSPs with Pharo Processes

Once our DSPs have been created, initialised and started, Pharo's syntax enables us to create algorithmic compositions. This is achieved by defining a process that repeatedly executes a `BlockClosure` for a number of times. Within this `BlockClosure`, time advancement is managed by sending the `wait` message to an instance of the `Delay` class. The process is then initiated by sending the `fork` message. This approach allows forked processes to run concurrently.

```
djembe := Djembe new.
dsp :=  djembe stereo asDsp.
dsp init.
dsp start.

pos := 0.
[128 timesRepeat:
    [ dsp setValue: (Random new
        nextIntegerBetween: 200 and: 900)
        parameter: 'DjembeFreq'.
     dsp setValue: (pos  \% 1) parameter: '
         DjembeStrikePos'.
     dsp trig: 'DjembeGate'.
     pos := pos + 0.1.
     (Delay forSeconds: 0.2) wait ] ] fork.
```

5. THE TOOLKIT AND TURBOPHAUSTO

In Phausto, we have implemented two sets of classes to simplify DSP programming and to provide musicians an ensemble of instruments effects for programming music on-the-fly without the need to use external audio generators.

5.1. The ToolKit

The ToolKit is a collection of synthesisers, effects and utilities included in the Phausto package. The name *ToolKit* pays tribute to Perry Cook's and Gary Scavone's *Synthesis ToolKit* [12]. Within the ToolKit, one can find utilities as an *incrementer*, an LFO that outputs a pseudo random signal, a *reader* and a *resetter* for reading sound files, and a basic *SamplePlayer* for playing back .wav files in Pharo applications.

5.2. TurboPhausto

TurboPhausto is a collection of synthesisers and effects designed to be the counterpart of SuperCollider's *SuperDirt* engine, intended for use with Coypu, the package that has been developed over the past three years for programming music on-the-fly with Pharo. Currently, 4 instruments and 2 effects are ready in *TurboPhausto*:

- *TpSampler*- a monophonic multisample player, that looks for all the sample in a specified folder (maximum 256 files);
- *Fm2Op* - a monophonic FM synth with 2 operators;
- *PsgPlus* - a monophonic chirpy synth inspired by Sega Master System Programmable Sound Generator (PSG);
- *Chordy* - a pseudo polyphonic virtual analog synth, in which different chords can be selected with the `mode:` message;
- *DelayMonoFB* - a smoothed mono delay with resonant feedback and dry/wet control;
- *GreyHoleDW* - a mono version of Faust's GreyHole reverb with dry/wet control.

All *TurboPhausto* synthesisers come with an AR/ADSR envelope and optionally with filters and effects on their output. All the effects are provided with dry/wet control. Here is a brief example of an extract of a live performance using Coypu and *TurboPhausto*:

```
"create, initialize the DSP"
dsp := (TpSampler new + PsgPlus new + Fm2Op
    new) stereo asDsp.
dsp init.
dsp start.

"initialize the Performance and assign the
   DSP"
p := PerformanceRecorder uniqueInstance .
p performer: PerformerPhaust new.
p freq: 143 bpm.
p activeDSP: dsp.

"assign TurboPhausto instruments to
   Performance sequencers"
16 downbeats index: '1' to: #TpSample.
16 quavers notes: '38 41 45 50' to: #
   PsgPlus .
16 rumba to: #Fm2Op.
p playFor: 32 bars.
```

6. CONCLUSIONS AND FUTURE WORK

After a year of development, Phausto provides a comprehensive solution for Pharo programmers to integrate sound synthesis into their applications. It includes sample players, basic oscillators, envelopes, various physical models, resonant filters, reverbs, and delays. The extensive array of Unit Generators features a streamlined API for parameter manipulation, which has been well-received by Pharo programmers. This was highlighted at the ESUG 2024 conference[7] in Lille, where Phausto earned 3rd place in the *Innovation Technology Awards*. Additionally, TurboPhausto's synthesizers and effects were showcased in a 30-minute live performance titled *Riding the MoofLod*.

In the coming months, the primary goal will be to port all functions from the Faust libraries and the Box-API. Subsequently, the focus will shift to MIDI and polyphonic support, which may necessitate the implementation of a MIDI handler using PortMIDI, an open-source library for which FFI bindings are already available in the Pharo-Sound package. Additionally, classes and methods will be provided to export DSPs into different architectures directly from Phausto and to display their signal flow along with a default GUI. Finally, an exhaustive and robust ensemble of instruments and effects for TurboPhausto will be designed and implemented.

7. ACKNOWLEDGMENTS

Thanks to Stéphane Letz for the never ending advice on Faust and its ecosystem, and to Yann Orlarey for being enthusiast about Phausto. Thanks Stéphane Ducasse, Guillermo Polito, Esteban Lorenzano, Nahuel Palumbo, for their invaluable support, push and assistance throughout the development of Phausto. Thanks to the Pharo Association for the support.

8. REFERENCES

[1] Alan C. Kay, "The early history of smalltalk," *ACM Sigplan Notices*, vol. 28, no. 3, pp. 69–95, 1993.

[2] Giorgio C. Buttazzo, *Hard Real-Time Computing Systems. Predictable Scheduling Algorithms and Applications*, Springer, Berlin, Germany, 2011.

[3] Stéphane Letz, Yann Orlarey, and Dominique Fober, "An overview of the faust developer ecosystem," in *Proceedings of International Faust Conference (IFC18)*, Mainz, Germany, July 2018.

[4] Stéphane Ducasse, Gordana Rakic, Sebastijan Kaplar, Quentin Ducasse Originally written by A. Black, S. Ducasse, O. Nierstrasz, D. Pollet with D. Cassou, and M. Denker, *Pharo 9 by Example*, Book on Demand – Keepers of the lighthouse, 2022.

[5] Stéphane Ducasse, *Pharo with Style*, Creative Commons, 2022, `http://books.pharo.org/booklet-WithStyle/pdf/WithStyle.pdf`.

[6] Adele Goldberg and David Robson, *Smalltalk-80: the language and its implementation*, Addison-Wesley Longman Publishing Co., Inc., 1983.

[7] Trudcill P. Dell Hymes, "Pidginization and creolization of languages," *Journal of Linguistics*, vol. 9, no. 1, pp. 193–195, 1971.

[7] ESUG stands for European Smalltalk User Group.

[8] Guillermo Polito, Stéphane Ducasse, Pablo Tesone, and Ted Brunzie, *Unified FFI - Calling Foreign Functions from Pharo*, Creative Commons, 2020, `http://books.pharo.org/booklet-uffi/pdf/2020-02-12-uFFI-V1.0.1`.

[9] Julius O. Smith III, "Viewpoints on the history of digital synthesis," in *Proceedings of the International Computer Music Conference (ICMC91)*, Montréal, Canada, October 1991.

[10] Ge Wang and Perry R. Cook, "Chuck: A concurrent, on-the-fly, audio programming language," in *Proceedings of the International Computer Music Conference (ICMC03)*, Singapore, September 2003.

[11] Ge Wang, *The ChucK Audio Programming Language. "A Strongly-timed and On-the-fly Environ/mentality"*, Ph.D. thesis, Princeton University, 2008.

[12] Perry R. Cook and Gary P. Scavone, "The synthesis toolkit (stk)," in *Proceedings of the International Computer Music Conference (ICMC99)*, Beijing, China, October 1999.

FAUST PLUGINS IN (SOMETIMES UNEXPECTED) WEB-BASED HOSTS

Michel Buffa, Dorian Girard, Samuel Demont, Quentin Escobar, Ayoub Hofr *

University Côte d'Azur
Nice, France
michel.buffa@univ-cotedazur.fr

ABSTRACT

In this short paper we will present the use of FAUST based Web Audio Modules plugins in two hosts: an open source DAW and in an collaborative, immersive, WebXR application in the Musical Metaverse.

1. INTRODUCTION

WAM Studio is an online Digital Audio Workstation (DAW) for creating audio projects, designed as multi-track musical compositions [1, 2]. It has been designed around the standard for Web Audio plugins and hosts called "Web Audio Modules" (or "WAM") [3]. Each track represents a different layer of content that can be recorded, edited, played back or integrated with audio files. Some tracks can control virtual instruments, containing only the notes to be played and metadata. Users can add or delete tracks, play them individually or together, and arm them for recording (Figure 1). The integration of FAUST based plugins is detailed in section 2.

The Musical Metaverse (MM) [4] is an immersive virtual space dedicated to musical activities, expanding on the broader concept of the Metaverse. It has recently gained popularity, reigniting interest in shared virtual environments within the realm of computer music. Applications include virtual concerts, educational tools, and collaborative musical performances. Transposing user experiences into conventional 2D applications generally does not work in an immersive environment, and new ergonomic and sensory approaches need to be employed [5]. The WAM application presented in this paper focuses on a persistent, real-time multi-participant immersive world for shared music creation, exploiting recent W3C web standards such as Web Audio, Web MIDI, WebXR, WebGL, WebGPU, WebAssembly, WebSockets, now implemented in the web browsers of the most common VR/XR headsets available on the market. In this application, users can build music installations by connecting Web Audio Modules plugins in a graph. Most of these WAM plugins are made with FAUST. An original approach has been developed for integrating existing WAMs in the 3D world, with a GUI generated on the fly, providing a user experience adapted for VR headset controllers.

2. FAUST BASED WEB AUDIO MODULES IN THE WAM STUDIO DAW

2.1. General Features

WAM-Studio is a powerful online Digital Audio Workstation (DAW) that utilizes cutting-edge technology to enable users to playback, record audio and MIDI tracks, employ high-quality plugins (effects and virtual instruments), manage latency, and perform offline rendering[1]. The source code is readily available (mono repository with front-end and back-end, including a simple Docker image for deployment) and the application is available online [1].

Figure 1: WAM Studio typical view, showing audio and MIDI tracks with some parameter automation curves and the plugin chain associated with the selected track.

The project is developed in TypeScript, deliberately avoiding external frameworks, with the aim of making the code accessible to a wider audience and ensuring its long-term viability (minimal build tools). For those with a keen interest, WAM-Studio is like an "alarm clock to be disassembled» and will reveal the secrets of its design and implementation to the most curious, potentially uncovering insights into tasks that are not well-documented within the Web Audio and Web MIDI communities. It has been designed around the standard for Web Audio plugins and hosts called "Web Audio Modules" (or "WAM") [3] and serves as a compelling demonstration of its vast potential.

Each track represents a different layer of content that can be recorded, edited, played back or integrated with audio files. Some tracks can control virtual instruments, containing only the notes to be played and metadata. Users can add or delete tracks, play them individually or together, and arm them for recording. During recording, all other tracks play simultaneously, while armed tracks record new content.

In WAM-Studio, each track is a container for audio or MIDI related data, accompanied by an interactive representation of this data, editing and processing functions, and a few default parameters such as the track's volume and left/right panning. Figure 1 shows some audio tracks in WAM Studio with the associated audio buffer region (waveforms) and MIDI regions (squares) displayed

* This work has been supported by the French government, through the France 2030 investment plan managed by the Agence Nationale de la Recherche, as part of the "CA DS4H project, reference ANR-17-EURE-0004

[1] https://wam-studio.i3s.univ-cotedazur.fr/, source code: https://github.com/Brotherta/wam-studio

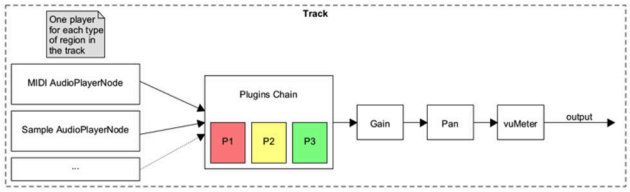


Figure 2: Audio graph of a track implementation.

and the default track controls/parameters on the left (mute/solo, record arming, volume, stereo panning, automation curve display, effects plugins). As many tracks can be displayed, scrolled during playback, zoomed and edited, we used the pixi.js library to efficiently manage drawing and interaction within an HTML canvas. This library uses GPU-accelerated WebGL rendering and offers many features for managing multiple layers on a single HTML5 canvas.

Figure 2 shows the audio graph corresponding to the processing chain of an audio track. The sound goes from left to right: first the "track player/recorder/editor" is implemented as an AudioWorklet node, using custom code to render an audio buffer or a MIDI region, then the sound goes through a chain of WAM plugins for adding audio effects, then the output signal has its gain and stereo pan adjusted, then we have another AudioWorklet node for rendering volume in a canvas (VU-meter).

Plugin chains are managed using a special WAM plugin that also acts as a "mini host" (Figure 3 and 4). We call it the WAM-bank (or the "WAM pedalboard") [6]. It connects to one or more plugin servers, which return(s) the list of available plugins as a JSON array of URIs (a WAM plugin can be loaded simply using a dynamic import and its URI, see [2]). From this list of URIs, WAM plugin descriptors are retrieved, which contain metadata about the plugins: name, version, provider, thumbnail URI, type (effect or instrument), available inputs and outputs etc. When the pedalboard plugin is displayed in the DAW, the chain of active plugins is empty, and plugins can be added to the processing chain, deleted, reordered and their parameters set.

Any configuration can be saved as a named preset (e.g. "crunch guitar sound 1", “Synthesizer with ambient audio effects”). Presets can be organized into sound banks ("rock", "funk", "blues"). Managing the organization and naming of banks and presets is the responsibility of the WAM-bank plugin. The parameters exposed by this plugin correspond to all the parameters of the active preset (i.e. the sum of the parameters of the preset’s plugins in the chain) and can be automated by the DAW.

2.2. Rapid Development of WAM Plugins Using the Online IDE

All WAM plugins have a URI and can be dynamically imported into hosts using JavaScript dynamic import statements. The DAW uses JSON configuration files that contain a list of WAM plugin URIS. When one clicks on the FX icon of a track, an instance of the WAM-bank plugin manager is created, and acts as a Mini-Host for handling the chain of plugins (instruments, effects) that will be associated with the track.

Most instruments and audio effects in the current version of the DAW have been developed and exported using the FAUST IDE directly, without any further modifications [7]. A step by step tu-

Figure 3: WAM Bank is a special WAM that manages the chain of plugins associated with a track.

Figure 4: WAM plugins associated with a track

torial about how to build WAM plugins with the FAUST IDE is available online [2]

In addition, many WAMs developed by the community of developers have been made available through the "WAM Community REST API", an endpoint from which WAMs can be requested online [8]. As of October 2024, dozens of WAM plugins are available.

Figure 6 shows the MIDI flute instrument in the GUI Builder of the FAUST IDE (the FAUST code comes from the example menu of the IDE). From this GUI Builder, an optional custom GUI can

[2]Create your own Web Audio Plugins with the FAUST IDE: `http://tinyurl.com/yckdyax4`http://tinyurl.com/yckdyax4

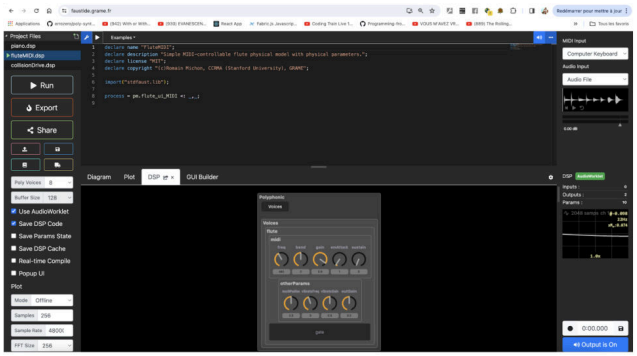


Figure 5: Auto-generated CSS based GUI of the MIDI flute instrument.

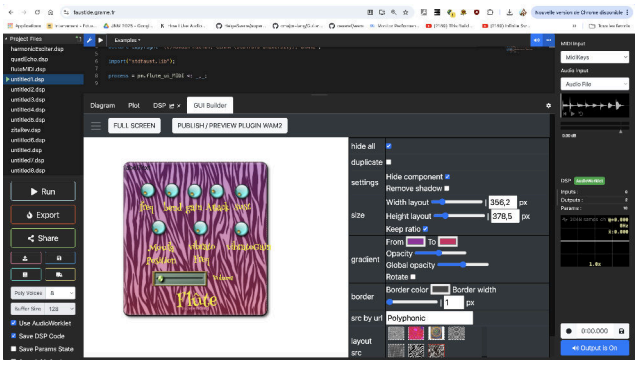


Figure 6: MIDI flute virtual instrument in the GUI Builder of the FAUST IDE

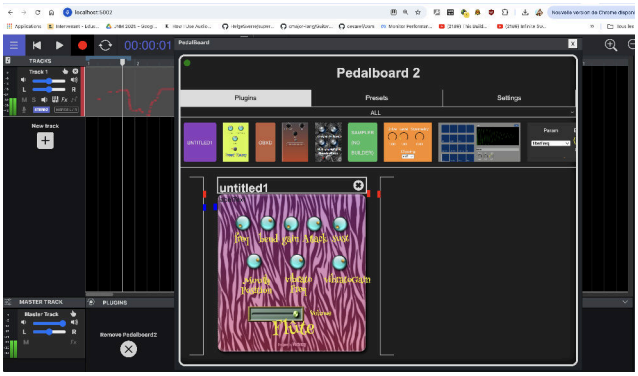


Figure 7: The Midi flute WAM in WAM Studio.

be designed (changing the position, sizes, look and feel of the buttons, changing the fonts used etc.), and by pressing the "Publish / preview WAM2" button, a WAM plugin is generated and published on a remote server. It can be tested directly from the IDE or downloaded as a ZIP file. Once published, its URI can be directly used in any WAM host. Figure 7 shows it in the WAM Studio DAW. The whole operation (compiling the source code in the IDE, making a custom GUI, exporting it and publishing it on a remote server) took less than two minutes. Then, making it available in the DAW is just a matter of adding one line in a configuration file.

The DAW includes a wide range of audio effects created with FAUST, such as a recreation of Eventide's famous Blackhole effect pedal, or Electro-Harmonix's Big Muff fuzz and Stone Phaser, for example, as well as numerous original effects covering the most classic needs: reverbs, modulation effects, stereo enhancers, distortions, etc. Several instruments of various types have also been integrated (flute, djembe, guitar, synthesizers).

2.3. FAUST WAMs for Optimal Performances in Host-Plugin Communications

Based on the faustwasm module, the FAUST distribution introduces a new script called faust2wam, a JavaScript tool that can generate self-contained FAUST WAMs within the Node.js environment or dynamically within browsers. Additionally, support has been added for polyphonic instruments and FAUST based spectral processors [8]. These new generation targets (web/wam2-ts, wam2-poly-ts, and wam2-fft-ts) are now available in the FAUST IDE Export window and are also used by the GUI Builder presented in the previous section.

During the WAM export and publishing process, the original FAUST code is compiled to WebAssembly, and the generated GUI is packaged as a Web Component. All generated files are placed in a single folder, which can be published online and downloaded. By default, this includes two different GUIs: the auto-generated default GUI (Figure 5) and the custom GUI, designed using the GUI Builder and leveraging widgets from the webaudiocontrols library (Figure 6). Two URIs are provided: one for the default GUI and the other for the custom GUI-based WAM.

More interesting in terms of performance is that the WebAssembly code runs inside an Audio Worklet, enabling custom DSP code execution. Audio Worklets were added to the Web Audio API in 2018, and FAUST was one of the first DSLs to support them as a target [9]. An Audio Worklet consists of two main parts: 1) the Audio Worklet Processor, where the core custom audio processing occurs. It's a class that extends AudioWorkletProcessor and processes audio in small chunks called frames this is where the FAUST WebAssembly code runs. 2) the Audio Worklet Node, which serves as the interface connecting the Audio Worklet Processor to the Web Audio API's audio graph. It bridges custom audio processing in the Worklet with the Web Audio API context.

More specifically, the faust2wam script uses WAMProcessor and WAMNode classes from the WAM SDK, which inherit from AudioWorkletProcessor and AudioWorkletNode, providing additional features. In particular, WAMProcessor supports high performance communication with a WAM host using Shared Array Buffers [3].

This approach eliminates the need to create events or cross the thread boundary between the GUI thread and the audio thread when a host needs to communicate with a WAM plugin, for parameter automation at the sample rate or for MIDI communication

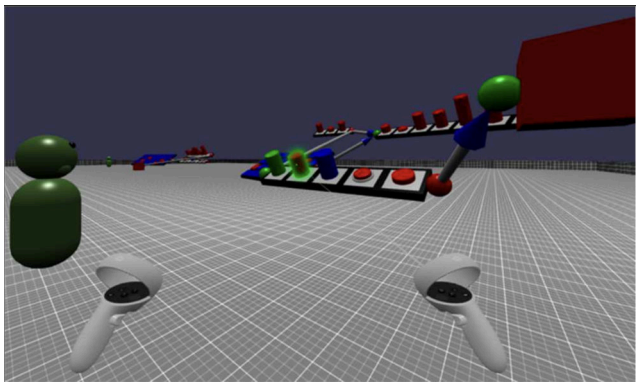


Figure 8: Multiple participants assemble 3D WAMs to build spatialized music installations.

with a virtual instrument since both the host and plugin, based on Audio Worklets, have their processing parts running in the high-priority audio thread.

3. FAUST BASED WEB AUDIO MODULES IN THE MUSICAL METAVERSE

3.1. WAM Jam Party, Making Music in an Immersive, Collaborative Environment

WAM Jam Party [10] is an immersive, collaborative application that runs in the Web Browser of VR headsets. This WAM-based application focuses on a persistent, real-time multi-participant immersive world for shared music creation, exploiting recent W3C web standards such as Web Audio, Web MIDI, WebXR, WebGL, WebGPU, WebAssembly, WebSockets, now implemented in the web browsers of the most common VR/XR headsets available on the market.

As stated in section 1, dozens of WAM plugins are now available through the WAM Community endpoint, comparable in quality and complexity to native applications [8]. Available WAMs include note generators such as: a piano roll, a programmable step sequencer, random note generators, chord generators, virtual instruments: synthesizers samplers, physical modeling of instruments (flute, clarinet, brass, djembe), audio effects (including all classics: flanger, chorus, reverb, distortion, fuzz, overdrive, etc.). The majority of the effects and instruments have been generated with the FAUST IDE.

With WAM Jam Party, users can connect to a URL using the VR headset web browser, and start building musical installation in the immersive world, by adding and connecting together note generators, virtual instruments and audio effects (Figure 8). Each of these component is an existing Web Audio Module.

The main inspiration was Sequencer Party [8], a collaborative online audio and MIDI editor, also based on WAMs. WAM Jam Party is a 3D adaptation of Sequencer Party concepts.

The first prototype of WAM Jam Party uses WAM introspection to build 3D interactive GUIs from existing 2D WAM plugins, BabylonJS for the 3D rendering of the immersive world, and a CRDT algorithm is used synchronizing incrementally the states of all the elements between clients. While client states are updated 3 times/s, the server saves seamlessly the ongoing session into a file every second, making the world persistent. Special care has been

Figure 9: Big Muff fuzz, a FAUST-based WAM, and its 3D GUI.

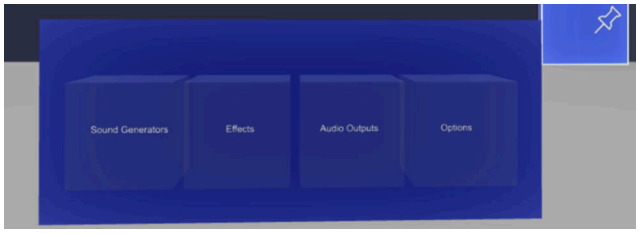


Figure 10: Main menu for adding WAMs in the 3D scene.

taken for generating the 3D representation of WAM plugins, using an optional JSON file for the mapping of the internal parameters into the 3D world. Each WAM is represented by default as a box with draggable cylinders, boxes or push buttons for its parameters (Figure 9).

A dynamic menu made with the Mixed Reality Tool Kit (MRTK) available in BabylonJS, proposes a large set of WAMs enabling users to add elements in the 3D world (Figure 10). In addition, a green and a right sphere are located on the sides of the box, enabling users to connect WAMs together to form an audio graph (Figure 11). Connect a step sequencer to a synthesizer to some audio effects and you have a first basic music installation. Any element can be moved, oriented, or have its parameters adjusted using different controllers (see Figure 12).

In a multi participant session, each user can perform the same interactions (add, remove, move, rotate WAMs, connect and disconnect elements, adjust the parameters, move around), and mod-

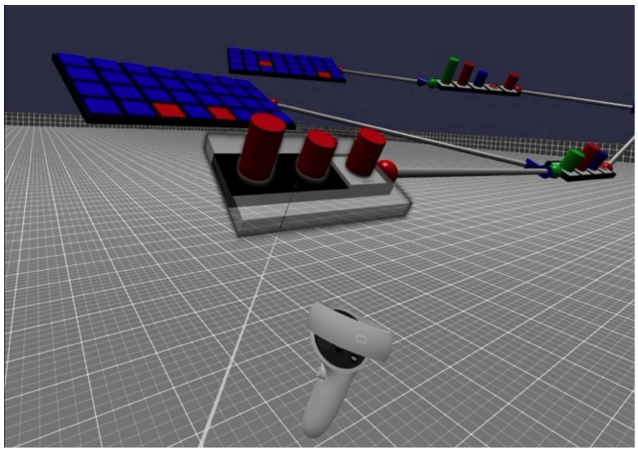


Figure 11: Example of a graph made of two note generators (step sequencers on the left), two instruments and audio effects.

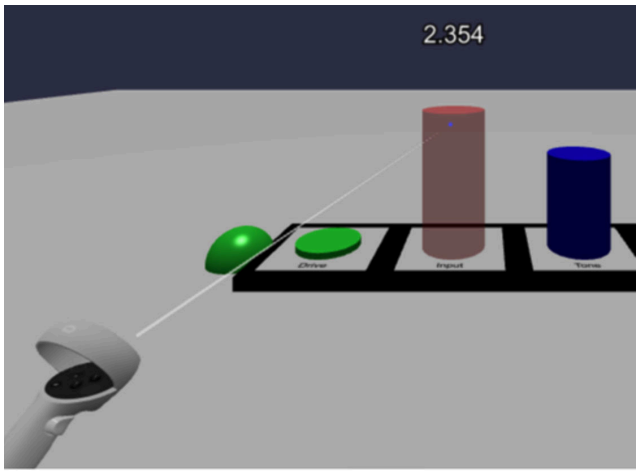


Figure 12: Parameters can be adjusted by clicking the controller trigger while aiming at a cylinder, and dragging it vertically.

ify other users' creations. The sound produced by each installation is spatialized and changes as you move your avatar in the 3D world.

Each user in the virtual environment is represented by a simple avatar consisting of a body, head, and two hands. The avatar's head orientation is directly mapped to the user's VR headset orientation, ensuring that the avatar's gaze direction aligns with the user's actual view within the virtual environment. The avatar's hands are positioned and oriented to match the user's controller movements. This visual representation provides a clear indication of the user's current actions, such as selecting WAMs or manipulating parameters.

3.2. How FAUST WAMs's 3D GUI Is Generated

Details about the UX design and interactions can be found in [10]. More interesting is the history of the first empiric tests conducted. FAUST allows us to declare basic user interface (UI) elements to

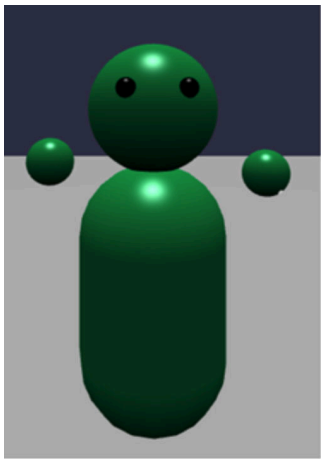


Figure 13: Avatar of a player.

Figure 14: A FAUST 2D GUI and its 3D version using 3D models of 2D widgets.

control the parameters of a FAUST object [3]. After the compilation of a FAUST source code an abstract representation of the UI can be obtained for generating target dependant code. For example, the WAM GUI Builder uses it for generating an HTML based GUI. A first series of tests consisted in using this abstract representation for generating a 3D GUI, as shown in Figure 14. Unfortunately, this kind of 3D interfaces were not suited for a 3D manipulation in VR headsets. Users found many difficulties operating 3D knobs, sliders or switches with VR headset controllers. Also, for plugins with too many parameters, the 3D rendered view could be confusing with small labels and widgets.

Finally, the adopted solution that gave good results during user testing was to use ad hoc 3D shapes with interactions adapted to the VR usages. For example, 2D knobs and sliders become draggable cylinders or boxes, switches become a push button, etc. It also became rapidly obvious that as all original 2D elements could not be used in 3D (i.e a synthesizer with 60 parameters controlled by 2D knobs), the original abstract UI definition provided by FAUST was no more useful. Instead, the list and type of parameters provided by the WAM API of the generated plugin was sufficient for generating a 3D GUI on the fly. For example, the Big Muff pedal from Figure 9 has four parameters (volume, tone and sustain, plus the on/off switch), three of type float, and one of

[3] `https://faustdoc.grame.fr/manual/quick-start/#building-a-simple-user-interface` Building a Simple User Interface - Faust Documentation.

type boolean, so we can generate a 3D GUI with three draggable cylinders and a push button.

A configuration file also helped filtering unwanted parameter 3D controllers and customizing colours, shapes, labels of the wanted 3D parameters. A "convention over configuration" approach was used: a minimal configuration file with just the URI of a WAM would lead to generating all parameters 3D controllers using default values, i.e red draggable cylinders for float parameters. But it is also possible to indicate different colors, labels, to hide some parameters by editing this configuration file. Furthermore, this parameter based generation works with all kind of WAMs, not only with FAUST based ones.

An interactive editor in which developers can enter an existing WAM URI and preview their 2D and 3D GUI (using a mouse of a VR headset) while customizing the different options (show/hide a parameter control, etc.) is under development, and could be integrated in the future in the FAUST GUI Builder.

4. CONCLUSIONS

This paper presents two innovative music applications utilizing Web Audio Modules plugins (WAMs), most of which are written in FAUST. We believe that FAUST and its online IDE offer one of the best approaches to developing Web Audio Modules. FAUST WAMs are at the heart of the applications presented: a web-based DAW and a 3D immersive application. While the first project shows that it is possible to recreate some of the most complex audio software on the Web (see "Why You Shouldn't Write a DAW" [4]), and plugins written in FAUST are a very important part of this, the second project opens up new possibilities and will certainly lead to the short-term availability of an editor for configuring and interactively generating reusable FAUST based 3D components.

5. ACKNOWLEDGMENTS

We would like to thank Antoine Vidal-Mazuy for his investment in WAM-Studio over the course of 2023 (he was its main coder and designer), and the team behind the Web Audio Modules, without which this DAW would never have existed. This work has been supported by the French government, through the France 2030 investment plan managed by the Agence Nationale de la Recherche, as part of the "UCA DS4H" project, reference ANR-17-EURE-0004.

6. REFERENCES

[1] Michel Buffa and Antoine Vidal-Mazuy, "Wam-studio, a digital audio workstation (daw) for the web," in *Companion Proceedings of the ACM Web Conference 2023*, 2023, pp. 543–548.

[2] Michel Buffa and Samuel Demont, "Can you DAW it Online?," in *IS2 2024 - IEEE International Symposium on the Internet of Sounds 2024 / 1st IEEE International Workshop on the Musical Metaverse (IEEE IWMM)*, Erlangen, Germany, Sept. 2024.

[4] David Rowland - ADC23, `https://www.youtube.com/watch?v=GMlnh6_9aTc`.

[3] Michel Buffa, Shihong Ren, Owen Campbell, Tom Burns, Steven Yi, Jari Kleimola, and Oliver Larkin, "Web audio modules 2.0: An open web audio plugin standard," in *Companion Proceedings of the Web Conference 2022*, 2022, pp. 364–369.

[4] Luca Turchet, "Musical metaverse: vision, opportunities, and challenges," *Personal and Ubiquitous Computing*, vol. 27, no. 5, pp. 1811–1827, 2023.

[5] Alberto Boem, Matteo Tomasetti, Alessio Gabriele, Agostino Di Scipio, and Luca Turchet, "User needs in the musical metaverse: a case study with electroacoustic musicians," 2024.

[6] Michel Buffa, Pierre Kouyoumdjian, Quentin Beauchet, Yann Forner, and Michael Marynowic, "Making a guitar rack plugin-webaudio modules 2.0," in *Web Audio Conference 2022*, 2022.

[7] Shihong Ren, Stephane Letz, Yann Orlarey, Romain Michon, Dominique Fober, Michel Buffa, and Jerome Lebrun, "Using faust dsl to develop custom, sample accurate dsp code and audio plugins for the web browser," *Journal of the Audio Engineering Society*, vol. 68, no. 10, pp. 703–716, 2020.

[8] Michel Buffa, Shihong Ren, Tom Burns, Antoine Vidal-Mazuy, and Stéphane Letz, "Evolution of the web audio modules ecosystem," in *Web Audio Conference 2024*. Zenodo, 2024.

[9] Stéphane Letz, Yann Orlarey, and Dominique Fober, "Compiling faust audio dsp code to webassembly," in *Web Audio Conference*, 2017.

[10] Michel Buffa, Ayoub Hofr, and Dorian Girard, "Using Web Audio Modules for Immersive Audio Collaboration in the Musical Metaverse," in *IS2 2024 - IEEE International Symposium on the Internet of Sounds 2024*, Erlangen, Germany, Sept. 2024.

[11] Yann Orlarey, Stéphane Letz, and Dominique Fober, *New Computational Paradigms for Computer Music*, chapter "Faust: an Efficient Functional Approach to DSP Programming", Delatour, Paris, France, 2009.

[12] Julius O. Smith, "Signal processing libraries for Faust," in *Proceedings of Linux Audio Conference (LAC-12)*, Stanford, USA, May 2012.

[13] Albert Gräf, "pd-faust: An integrated environment for running Faust objects in Pd," in *Proceedings of the Linux Audio Conference (LAC-12)*, Stanford, USA, April 2012.

[14] Romain Michon and Julius O. Smith, "Faust-STK: a set of linear and nonlinear physical models for the Faust programming language," in *Proceedings of the 14th International Conference on Digital Audio Effects (DAFx-11)*, Paris, France, September 2011.

IFC-24 Paper Session 3

FUNCTIONAL AMBISONIC GRANULATOR

David Fierro

CICM-MUSIDANSE
Université Paris 8
davidfierro@gmail.com

Alain Bonardi

CICM-MUSIDANSE
Université Paris 8
alain.bonardi@univ-paris8.fr

ABSTRACT

This paper presents a functional ambisonic granulator developed using the Faust language, designed to generate diverse sound outputs from minimal inputs. Suitable for mixed music performances, it features configurable envelopes, distributed internal parameters, and feedback loops. The granulator's architecture includes input filtering, grain generation, modulation, transposition, and spatialization. Two spatialization approaches are implemented: "spatial sound transformation" and "point source to diffuse field".

1. INTRODUCTION

This paper introduces a functional ambisonic granulator, combining granular synthesis with ambisonic spatialization using Faust's functional programming paradigm. Developed for the BBDMI project,[1] it aims to generate diverse sound outputs from minimal inputs, particularly suited for performances using electrophysiological signals. Its key features include:

- Configurable envelopes for grain shaping.
- Distributed internal parameters across channels.
- Strategic feedback loop positioning.
- Two distinct ambisonic spatialization techniques.
- Comprehensive user-controllable variables.

The granulator's architecture enables a wide spectrum of sound outputs. This paper details each component's implementation, explores the granulator's MaxMSP integration, and discusses future developments to enhance user accessibility and control. In general, the specificity of the proposed granulator lies in the highly configurable envelopes, the distribution of internal parameters for each instantiated channel, and the way feedback loops are positioned.

All the codes related to this paper can be found in our Gitlab repository.[2]

2. CONTEXT

The development of this granulator is proposed as a solution for generating a mixed music sound effect that operates with minimal input while producing a broad spectrum of diverse sound outputs. This type of sound effect is essential for our BBDMI project, which involves the performance of mixed music using EMG and EEG signals. Typically, the small number of extracted features from EEG signals and the limited number of channels in EMG devices result in a constrained set of variables for user interaction.

The concept of granular synthesis has a rich history that informs its current applications. Granular synthesis was initially conceptualized by Iannis Xenakis in the early 1950s, who explored the idea of sound as composed of small grains or particles, each contributing to a complex acoustic texture [1]. Xenakis's theoretical groundwork laid the foundation for later developments in the field.

In the 1970s and 1980s, Curtis Roads advanced granular synthesis by formalizing its methods and exploring its potential in digital sound processing. His work, particularly in Microsound, has been instrumental in defining the techniques used in granular synthesis today [2]. Roads highlighted the ability of granular synthesis to create new soundscapes by manipulating tiny fragments of sound, which is a core principle in the design of our ambisonic granulator.

Further contributions to the field were made by Horacio Vaggione, whose work on the articulation of microtime has significantly influenced the musical applications of granular synthesis. In his article "Articulating Microtime", Vaggione explores how the manipulation of microtemporal structures can be used to create complex sound forms that transcend traditional time scales [3]. His approach to granular synthesis emphasizes the importance of controlling the temporal evolution of grains, allowing composers to shape intricate sound textures at a microstructural level. This concept is reflected in the design of our granulator, which leverages feedback loops and delays, creating detailed and evolving soundscapes.

Barry Truax's work on real-time granular synthesis has also been influential, particularly in the context of live performance. Truax's approach to real-time processing aligns closely with the requirements of mixed music performances, where sound effects need to be generated dynamically based on live input [4]. This is directly applicable to the functionality of our granulator, which is designed for live use, incorporating feedback loops and real-time spatialization.

The design of this granulator was significantly influenced by the functional logic of the Faust programming language. The concept of a continuous signal flow, inherent to Faust, inspired the development of a flow of grains within the granulator. Through the incorporation of feedback loops, the granulator facilitates the creation of intricate sound textures. The development of this granulator was also inspired by the work conducted at the CICM lab [5]. By incorporating a final layer of spatialization, the granulator is capable of generating spatially diffuse sound fields and creating spatial sound transformations.

3. A LIVE MUSIC GRANULATOR

Following Truax's model of real-time granulation, this mixed music granulator requires an external sound input to generate audio.

[1] BODY BRAIN DIGITAL MUSIC INSTRUMENTS. ANR-21-CE-38-00018. https://bbdmi.nakala.fr/

[2] https://gitlab.huma-num.fr/bbdmi/bbdmi

This operational approach makes it particularly suitable for live mixed music performances. The granulator can be configured in two distinct modes: as a multichannel sound effect with ambisonic output, or as an ambisonic sound effect when positioned between an ambisonic encoder and decoder.

4. MAXMSP WRAPPING

Throughout the development of our project, we have created numerous objects within the MaxMSP environment. To maintain workflow continuity, we developed a Max/MSP wrapper that facilitates interaction with the granulator within this development environment. The wrapper was initially designed to enable dynamic invocation of any Faust object by simply adjusting the number of channels within a bpatcher.[3] The Figure 1 illustrates the variables within the wrapper, which include three additional options not originally present in the Faust code.

The first wrapper-specific option is the "channels" variable. By default, dynamically changing the number of channels for a Faust object within Max/MSP is not feasible. This variable, however, allows us to replace the Faust object by selecting the desired number of channels, thus offering greater flexibility in the development process.

The wrapper includes two additional variables, "randomize" and "dump", each providing distinct functionalities. The "randomize" option assigns random values to all internal variables within their respective minimum and maximum ranges. This feature is crucial for exploring the full range of the granulator's capabilities, especially given the complexity introduced by having 17 internal variables.

The "dump" option allows users to export the current configuration of the granulator for future retrieval with a single click. The interplay between the "randomize" and "dump" functions enables users to experiment with and save various configurations, facilitating both exploration and preservation of interesting settings.

5. ARCHITECTURE

The granulator's behavior is determined by seven distinct components:

- Input filtering.
- Feedback loop of the input signal with delay.
- Grains generation.
- Modulation.
- Grains transposition.
- Spatial sound transformations.
- Feedback loop with transposition.

The Figure 2 represents these 7 modules. Subsequent sections will provide a detailed discussion of each module. It is crucial to note that two variables significantly influence the behavior of all other parameters in the granulator: the "variability" and "indexdistr" parameters (explained in Section 5.6.3). These variables introduce varying degrees of randomness into each of the other parameters, thereby impacting their behavior and interaction. The "variability" parameter modulates the extent of randomness applied, while the

[3] A MaxMSP object that modularizes a patcher or subpatcher, displaying only specified visual elements in other patchers

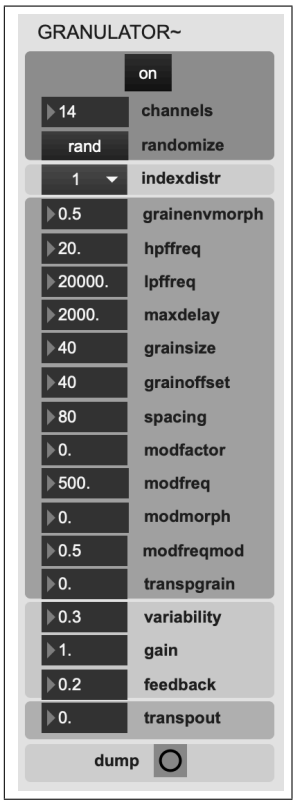


Figure 1: *Max bpatcher.*

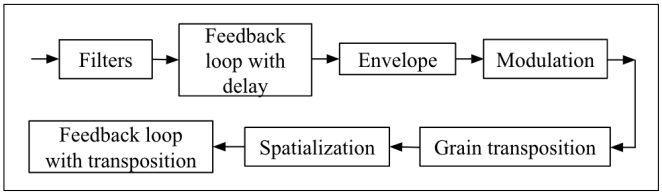


Figure 2: *Simplified diagram.*

"indexdistr" parameter governs how this randomness is distributed across the different channels or harmonics. Together, these variables play a central role in shaping the overall sonic output by dynamically altering the relationships between the granulator's internal variables.

5.1. Input Filtering

The first effect applied to the input signal is a combination of low-pass and high-pass filtering. By default, the granulator does not apply any filtering to the input signal; however, it provides the flexibility to dynamically adjust the input filtering as needed.

5.2. Delayed Feedback Loop

The primary objective of this module is to blend the input signal with a delayed version of itself to create a more complex sound texture before segmenting it into grains. By adjusting the "maxdelay" variable, a delay is introduced to the input signal, which is

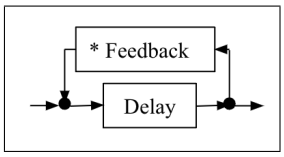


Figure 3: *Input delayed feedback loop.*

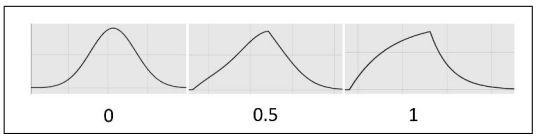


Figure 4: *Envelope design.*

then added to the input after being scaled by the "feedback" variable. Figure 3 illustrates the structure of this module.

These two variables allow the user to either process the clean input signal with no delay and zero feedback or to achieve a more intricate effect by mixing the input signal with its delayed version, with delays extending up to 10 seconds.

5.3. Grains Generation

This module is responsible for segmenting the processed input signal into grains. Two primary properties of the grain are addressed: the size of the grain and the grain envelope. The grain size determines the duration of each grain, while the grain envelope controls the temporal shape of the grain's amplitude profile.

5.3.1. Grain Size

Instead of employing a rarefaction logic, the granulator operates based on grain size and spacing. The grain size is calculated by adding the "grainoffset" variable to the product of the "grainsize" and the "variability" parameter. This configuration enables the generation of grains with a uniform size or with sizes varying within a specified range. The "spacing" variable multiplied by the "variability" parameter will determine the spacing between grains.

5.3.2. Grain Envelope

After evaluating various envelope shapes, we determined through experimental validation that a combination of Gaussian and logarithmic waveforms provides optimal results. The "grainenvmorph" variable enables dynamic adjustment of the envelope shape, ranging from zero to one. A value of zero corresponds to a Gaussian envelope, while a value of one corresponds to a logarithmic envelope. Figure 4 illustrates the envelope shapes for different values of the "grainenvmorph" variable. Figure 5 illustrates the creation of grains while the "grainenvmorph" and "grainsize" variable are changing.

5.4. Envelope Amplitude Modulation

Following the creation of each grain, this module enables the application of amplitude modulation to each grain through a variable modulation factor, "modfactor". Figure 6 depicts the grain envelope under three conditions: without modulation, with 50% modulation, and with 100% modulation. The variable "modfreq" sets

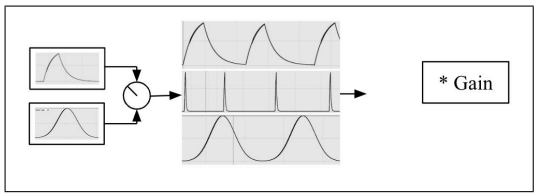


Figure 5: *Morphing between envelopes and different grain sizes.*

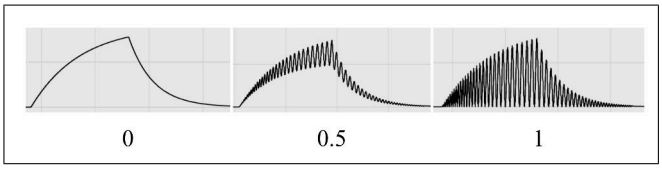


Figure 6: *Envelope amplitude modulation.*

the frequency of the amplitude modulation, ranging from 1 Hz to 15 kHz. Figure 7 demonstrates the effects of envelope modulation at three different frequencies: 10 Hz, 20 Hz, and 40 Hz.

Initially, the module utilized a single modulating signal for all grains. This approach led to the perception of a single, continuous tone after listening to a stream of grains, which did not align with our objectives. To address this issue, we implemented phase-controlled modulation for each grain. This modification ensures complete decorrelation between the shapes of the modulated envelopes of the grains. By employing an ad hoc modulating signal, the granulator is able to produce grains that share the same modulation characteristics without creating a cohesive tonal effect.

5.4.1. Modulating Signal Morphing

The ability to morph the waveform of the modulating signal in an amplitude modulation (AM) system plays a crucial role in shaping the sound of the resulting signal. For instance, using a square wave as the modulation signal tends to produce a bright, aggressive sound, whereas a sine wave results in a smoother, softer tone. The granulator's "modmorph" feature acts as an interpolation controller between four waveform signals: sine, sawtooth, square, and sine again. Figure 8 illustrates the envelopes generated by varying values of the "modmorph" variable. It is important to note that integer values correspond to pure waveforms, while decimal values result in a morphed shape that blends two adjacent waveforms.

The design choice to position a sine wave at both the beginning and end of the modulation cycle allows the user to create any conceivable morph between the three basic waveforms. This capability enables the creation of interesting effects through waveform morphing. By gradually transitioning from one waveform shape to another, either rapidly or slowly, users can achieve smooth or abrupt changes in sound texture. Figure 9 illustrates the process of

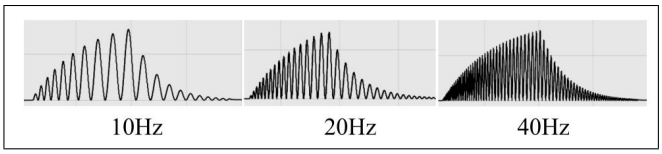


Figure 7: *Variable "modfreq".*

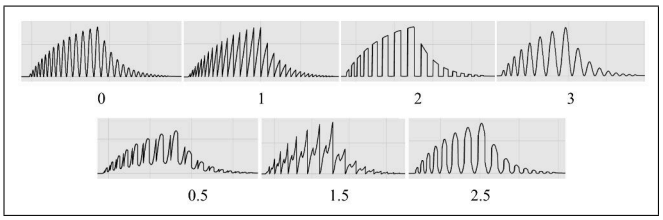


Figure 8: *Modulating signal morphing from sine to square.*

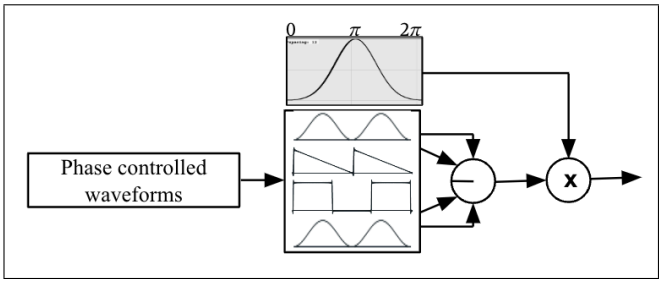


Figure 9: *Mix of different modulating signals.*

wave morphing applied to the grain envelope.

5.4.2. Modulating the Frequency of the Amplitude Modulating Signal

In addition to modulating the amplitude of each grain's envelope, the granulator also offers a variable called "modfreqmod". This variable controls the degree of frequency modulation applied to the amplitude modulation of the envelope. Figure 10 illustrates the impact of varying the "modfreqmod" variable from zero to one. In this context, a value of 0.5 represents the absence of frequency modulation, while values closer to zero or one introduce increasing levels of frequency modulation to the amplitude modulation envelope. This transformation enables the creation of envelopes where the amplitude modulation can dynamically shift from a high frequency to a low frequency, or vice versa, with each new grain. This modulation process is applied every time a grain is generated, allowing for varied and evolving sound textures.

5.5. Grains Transposition

At this stage of the process, the grains are transposed within a range of -24 to +24 semitones.

5.6. Spatialisation

Given that one of CICM laboratory's primary focus on sound spatialization and our prior advancements in the field of ambisonic spatialization, we decided to integrate an ambisonic spatialization layer into the granulator. There are various approaches to constructing this spatialization layer, and we chose to implement two distinct methods, each designed to produce different sonic outcomes. Currently, we have developed two versions of the granulator, each catering to different musical requirements. The version presented in this paper differs from the second version, particularly in its approach to spatialization. Here we discuss this two approaches :

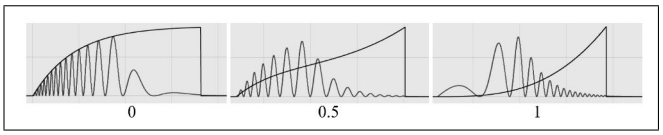


Figure 10: *Frequency modulation of the amplitude modulating signal.*

5.6.1. Spatial Sound Transformation

The version of the granulator presented in this paper employs a method we refer to as "spatial sound transformation". Unlike traditional spatialization approaches that focus on placing sound at a specific point in space, this approach leverages spatialization algorithms to create new and dynamic sound effects [6].

In this configuration, the granulator could function as a multichannel effect, with each channel acting independently, but in order to achieve a more intricate spatial result, the granulator is positioned between an ambisonic encoder and decoder. This approach contrasts with the traditional method of granulating multiple channels, placing them on the space and then projecting them through a sound system. By using the granulator as an ambisonic effect between the encoder and decoder, we modify the spatial representation sent from the encoder to the decoder, effectively altering how the spatial audio is perceived.

This "spatial sound transformation" enables a wide array of sound effects, one example is spatial decorrelation [7]. By applying different delays to each ambisonic harmonic channel, we can alter the reconstruction of the final "sound image", leading to unique spatial effects. In Section 5.6.3, we will explore how each variable of the granulator is configured to respond to different distributions among the ambisonic harmonics, further enhancing the complexity and richness of the spatial effects.

5.6.2. From Point Source to Diffuse Field

The second implementation of the granulator adheres to a more conventional approach to spatialization. In this version, each grain is positioned randomly within the 2D space before entering the final feedback loop, that will be discussed in Section 5.7.

Since each grain is spatialized using an ambisonic encoder, an ambisonic decoder of the same order is required for proper interpretation. What makes this implementation particularly interesting is that the initial "image" of the spatialized sound is precisely positioned in a specific location within the space. However, as the sound is reintroduced through the feedback loop, these subsequent "images" become increasingly diffuse, ultimately creating a more complex and enveloping diffuse sound field. This approach leverages the spatialization algorithms to transition from a clearly defined spatial sound to a more ambient and immersive auditory experience.

5.6.3. Ambisonic Distribution

When the granulator is used between an ambisonic encoder and decoder, every channel of the granulator is modifying a specific spherical (for 3D) or circular harmonics (for 2D). As every harmonic adds complementary information to the final reconstruction of the spatialised sound, the question of how to alter the parameters of each harmonic becomes relevant. As each channel of the

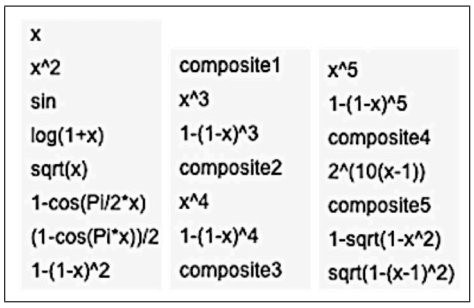


Figure 11: *Ambisonic distributions.*

granulator is modifying a specific harmonic, the way each internal parameter of every channel of the granulator relates to one another determines the characteristics of the final ambisonic reconstruction. There are numerous possibilities for distributing the granulator's internal parameters across the spherical or circular harmonics, ranging from high-level control parameters to specific adjustments for each harmonic. Our proposition for distributing how each internal parameter of each channel changes is based on the transfer functions proposed by Alain Bonardi and Paul Goutmann in the Faust "HOA" library. With this method the variables of each channel will change taking in account the selected transfer function and the number of the harmonic. The "indexdistr" variable enables users to select from various distribution patterns. Figure 11 illustrates the potential distributions as mathematical functions. The first distribution is "x" which represents a linear distribution. Upon selecting a specific ambisonic distribution, the internal parameters of each channel in the granulator are adjusted according to the "variability" parameter and the corresponding value from the selected transfer function. For instance, consider the impact on the first low-pass and high-pass filters. For each harmonic, or granulator channel, the frequency of these filters is calculated by taking the base frequency value and adding an additional component. This component is the product of the base frequency and a scaling factor ranging from zero to one, which is distributed across all the harmonics. For a linear transfer function, or "x", the resulting sonic effect is that the higher the harmonic level, the more pronounced the effect of each parameter becomes. This configuration creates a gradient of effects, where parameters such as filter frequency progressively intensify with the level of the harmonic, contributing to a more dynamic and evolving sound texture.

5.7. Grains Feedback Loop and Transposition

At this stage, a grain with specific spatial properties and a modified spectrum, thanks to filtering and modulation, has already been created. The final step in the granulator's process involves the collection of these grains through a feedback loop. This final loop enables the granulator to produce a continuous stream of grains rather than discrete, isolated grains at the output. As detailed earlier in this paper, parameters such as grain size, spacing, and other characteristics are adjusted before each grain is generated. The outcome is a flux of grains with varying sizes and spectral contents, which are accumulated within a time window specified by the "maxdelay" variable.

Figure 12 illustrates the configuration of this final feedback loop. A critical aspect of this module is the transposition within the feedback loop. As shown in Figure 12, each grain is initially

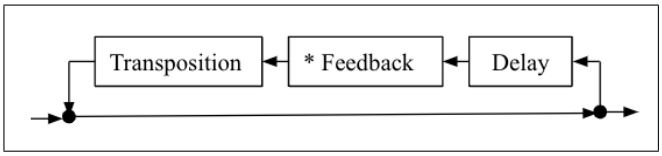


Figure 12: *Feedback loop and transposition of grains.*

outputted without any transposition. However, subsequent feedback signal of the same grain are increasingly transposed. This setup produces an interesting sonic effect, as the echoes of each grain begin to "escape" into either very low or very high frequency spectrums. The variable "transpout" controls the degree of transposition applied, allowing the user to shape the extent of this spectral shift.

5.8. Smoothing

A 1024-sample transition *buffer* is employed for all input variables to smooth the input before it is applied. This smoothing mechanism mitigates abrupt changes that could otherwise result in audible clicks.

6. MACRO CONTROLS

The granulator has 17 internal variables, which exceeds what a performer can feasibly adjust simultaneously, especially when working with EMG/EEG signals that offer limited channels for control. To maximize the granulator's potential, we have discovered that employing advanced mapping algorithms, such as regression models and neural networks, can yield highly effective results. These approaches allow for the complex and dynamic manipulation of the granulator's parameters, enabling performers to explore a broader range of sonic possibilities despite the constraints of limited input channels.

6.1. Regression

By employing the regression algorithm "RapidMax",[4] we successfully recorded various configurations of the granulator, enabling them to evolve in response to inputs provided to the regression model. This approach offers users the flexibility not only to explore the extensive possibilities of the granulator but also to record these configurations and seamlessly morph between them. This dynamic capability enhances the creative potential of the granulator, allowing for the development of evolving soundscapes and complex transitions between different sonic states.

6.2. One Layer Perceptron

A comparable approach to the regression model was adopted by developing a single-layer perceptron neural network using Faust. This implementation relies on vector and matrix multiplications to evaluate the perceptron. However, as of now, we have not developed a training algorithm, so this method remains in the experimental stage. Currently, we can only test random weights and biases until an interesting outcome is achieved. The lack of a systematic training process limits the full potential of this method,

[4] `https://github.com/samparkewolfe/RapidMax`

but it represents a promising direction for future development in generating and exploring novel sound configurations.

7. USER CASES AND FUTURE DEVELOPMENTS

Within the context of the BBDMI project, we have had the opportunity to test the granulator in various settings. This experience has provided valuable insights into both the possibilities and limitations of the granulator. In concert settings, we received positive feedback regarding the granulator's sound, which has garnered interest from composers and musicians. However, despite this initial enthusiasm, many users eventually discontinue using the granulator due to its complexity.

This issue highlights the need for further development of the internal mapping algorithms to facilitate more accessible and intuitive macro controls directly within the Faust-compiled object. Simplifying the user interface while maintaining the granulator's powerful capabilities is crucial for broader adoption.

8. ACKNOWLEDGMENTS

The research presented in this paper has been supported by the French national research agency (ANR-21-CE38-0018).

9. REFERENCES

[1] Iannis Xenakis, *Formalized music: Thought and mathematics in composition*, Bloomington : Indiana University Press, 1971.

[2] Curtis Roads, *Microsound*, The MIT Press. MIT Press, London, England, Aug. 2004.

[3] Horacio Vaggione, "Articulating microtime," *Computer Music Journal*, vol. 20, no. 2, pp. 33–38, 1996.

[4] Barry Truax, "Real-time granular synthesis with a digital signal processor," *Computer Music Journal*, vol. 12, no. 2, pp. 14–26, 1988.

[5] Alain Bonardi, "La librairie abclib : un ensemble de codes Faust rassemblant 20 ans de recherche, enseignement et création en musique mixte," in *Proceedings of the Journées d'Informatique Musicale 2021*, videoconferencing due to Covid, July 2021, AFIM.

[6] Paul Goutmann and Alain Bonardi, "Approaching spatial audio processing by means of decorrelation and ring modulation in ambisonics," in *Proceedings of the 19th Sound and Music Computing Conference, june 5- 12th, 2022, Saint-Etienne (France)*. 6 2022, Zenodo.

[7] Anne Sèdes, "Approche musicale de la décorrélation microtemporelle dans la bibliothèque HOA," in *Journées d'Informatique Musicale 2015*, université de Montréal, Ed., Montréal, Canada, May 2015, Université de Montréal.

WIDGET MODULATION IN FAUST

Yann Orlarey
Univ Lyon, Inria, INSA Lyon, CITI, EA3720, 69621 Villeurbanne, France
yann.orlarey@inria.fr

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

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

ABSTRACT

This article presents a novel extension to the FAUST programming language called *Widget Modulation*. Inspired by *Modular Synthesizer*, this high order operation enables developers to effortlessly implement *voltage control type* modulation to existing FAUST circuits.

Although signal modulation can easily be achieved by writing the necessary code during circuit development, *Widget Modulation* expressions enable it *a posteriori*, after the circuit has been developed and without modifying its code. This feature allows for easy *reuse and customization* without prior planning by the original circuit designer, offering a new level of expressivity and flexibility in FAUST circuit design.

1. INTRODUCTION

The development of voltage control in analog sound synthesis, beginning in the mid-20th century, marks a milestone in the history of electronic music. This technique involves using electrical voltages to modulate various synthesizer parameters, such as the pitch or amplitude of an oscillator or the cutoff frequency of a filter. This capability allows synthesis parameters to evolve, imparting timbral richness, expressiveness, and dynamism to the sounds produced.

Key figures in the development of voltage control for analog sound synthesis include Hugh Le Caine, Robert Moog, and Don Buchla. Building on Hugh Le Caine's concept of the voltage-controlled oscillator (VCO), Robert Moog established a crucial standard for modular synthesizers known as 1V/oct, where an increase of one volt corresponds to a pitch change of one octave [1].

This standard enables a form of recursivity within the synthesizer itself: the sound signals produced by one module can control the parameters of other modules, including itself. This recursive capability is a key factor in the richness and complexity of the sounds generated by modular synthesizers. Let's quote Suzanne Ciani in [2]: *"What we all love is the hands-on experience of patching and tweaking ... the way it engages both our brains and our bodies, the freedom of choice it offers, the individualism, the uniqueness."*

Implementing *voltage control* principals, à la Modular Synthesizer, in FAUST [3] had always been straightforward. All we need to do is add an audio input and implement a *modulation operation* that describes how to combine this additional input signal with that of the widget we want to modulate. The modulation operation can be as simple as an addition or multiplication.

As an example, let's start with a simple oscillator: `myosc`, with a frequency control, but no modulation possibility:

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

myosc = vslider("freq[style:knob][scale:log
    ]", 440, 20, 20000,0.1)
      : os.osc;

process = myosc;
```

Let's now look at how to transform `myosc` to create a frequency modulation (FM) circuit [4]. To do this, we need to modulate the frequency of the oscillator by introducing the influence of another oscillator. Specifically, we will achieve this by adding the output signal of the second oscillator (`mymod`) to the frequency control (the "freq" widget) of the first oscillator, as in the following code:

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

myosc = +(vslider("freq[style:knob][scale:
    log]", 440, 20, 20000,0.1))
      : os.osc;

mymod = hslider("fmod[style:knob][scale:log
    ]", 110, 20, 1000,0.01)
      : os.osc * hslider("amod[style:knob
          ]", 25, 0, 1000, 0.01);

process = mymod : myosc;
```

The modification was minimal. All we had to do was add an input signal to `myosc` and sum it with the "freq" widget of the oscillator. However, we could do this because we had access to the source code of `myosc`. If `myosc` had been defined in a library, we would have had to either modify the library or duplicate the `myosc` code in our program.

As we will see, *Widget Modulation* allows us to do the same kind of transformation but without modifying the source code! It, therefore, makes FAUST's code reuse mechanisms, `library()` and `component()` even more useful.

2. EXAMPLES OF WIDGET MODULATION

Before a more formal description of *Widget Modulation*, let's consider some very simple examples using `dm.freeverb_demo` from the standard library.

Without inspecting the code, just by looking at the user interface of `dm.freeverb_demo` (figure 1) we can see the names of the various widgets that are involved and that could possibly be modulated.

Here we are interested in the `"Wet"` slider that controls the balance between the wet (reverbered) and dry (unprocessed) signals.

In order to modulate the `"Wet"` slider, we write:

Figure 1: *Freeverb user interface.*

```
["Wet" -> dm.freeverb_demo]
```

As we can see, the syntax of a *Widget Modulation* deliberately resembles that of a *lambda-abstraction*, although the semantics are quite different and the two should not be confused. Here the string `"Wet"` identifies the target of the modulation, the `vslider("Wet",...)` in the definition of `dm.freeverb_demo`. Because we didn't specify a modulation circuit, the modulation circuit is implicitly assumed to be a multiplication.

The resulting circuit has now three inputs: the new input for the modulation signal, and the original left and right inputs of the reverb. Moreover the signal delivered by the `"Wet"` slider is multiplied by the input modulation signal everywhere in the reverb circuit.

This extra input can now be connected to an oscillator to modulate the `Wet` parameter as in:

```
1+os.osc(0.1)/4,_,_: ["Wet" -> dm.
    freeverb_demo];
```

Here the modulation signal `1+os.osc(0.1)/4` is an oscillator with a frequency of 0.1 Hz, and an amplitude of 0.25. The `1+` is used to ensure that the modulation signal is between `0.75` and `1.25`.

Modulation Circuit. In the previous example, we didn't indicate a modulation circuit. To do so, we use the symbol `':'` followed by the modulation circuit. For example `"Wet":+` indicates the use of an addition as a modulation circuit. It means that our previous example is equivalent to:

```
1+os.osc(0.1)/4,_,_: ["Wet":* -> dm.
    freeverb_demo];
```

By writing `"Wet":*` we explicitly stated to use a multiplication `*` as a modulation circuit. Please note that the `:` symbol in `"Wet":*` is used to separate the widget name from the modulation circuit and should not be confused with the sequential composition operator `:`, even if it also suggests an idea of connection.

We'll come back to this later, but a modulation circuit can be of three types. A circuit with *two inputs and one output*, like `*` or `+`; a circuit with *one input and one output*, like `*(2)`; or a circuit with *no input and one output*, like `0.75`.

1. Only a modulation circuit with two inputs, like `+` or `*` creates an external modulation input. Its first input is connected to the widget, and the second one becomes the modulation input.
2. Another possibility is to describe the entire modulation circuit in a single expression, in which case there is no need for an additional input, as in the following example:

   ```
   ["Wet":*(1+os.osc(0.1)/4) -> dm.
       freeverb_demo];
   ```

3. Finally, we can completely replace the target widget with a modulation circuit that has no inputs, for example:

   ```
   ["Wet":0.75 -> dm.freeverb_demo];
   ```

   Then the slider will be removed from the user interface and replaced by a constant value of 0.75, with potential speed up of the computation.

   Instead of replacing a widget with a constant, we can replace it with another widget, for example to change its name, style, range, etc.:

   ```
   ["Wet":vslider("WetDry", 0.25, 0, 1,
       0.01) -> dm.freeverb_demo];
   ```

   In this case, all occurrence of `vslider("Wet", 0.25, 0, 1, 0.01)` in `dm.freeverb_demo` is replaced by `vslider("WetDry", 0.25, 0, 1, 0.01)`.

Multiple Targets. In the previous examples, we only had one target widget. We can specify more than one by separating them with commas as in the following example:

```
["Wet", "Damp", "RoomSize" -> dm.
    freeverb_demo]
```

The resulting circuit has five inputs, three modulation inputs and two reverb inputs. The first input modulates the `"Wet"` widget, the second the `"Damp"` widget, and the third the `"RoomSize"` widget. These three inputs are followed by the two inputs of the reverb.

Please note that the above expression is equivalent to the "curryfied" version:

```
["Wet" -> ["Damp" -> ["RoomSize" -> dm.
    freeverb_demo]]]
```

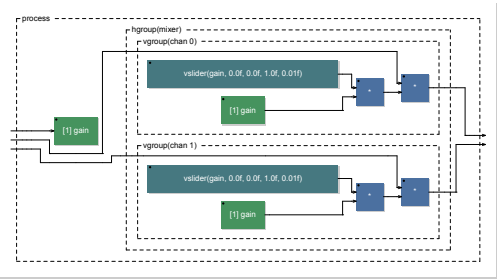


Figure 2: *All gain controls are modulated by the same input.*

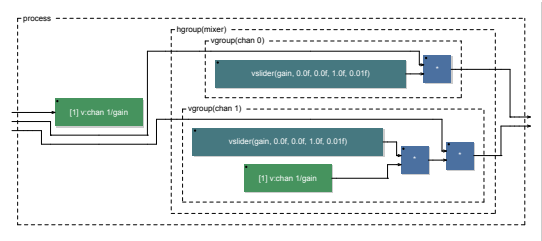


Figure 3: *Only the gain of channel 2 is modulated.*

Multiple Matches. It might happen that the same name matches multiple widgets in different groups. In this case, all the matched widgets will be modulated by the same audio input.

In the following example we have a kind of two voices mixer:

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

mixer = hgroup("mixer",
          par(i,2,
            vgroup("chan %2i",
              *(vslider("gain", 0, 0, 1,
                  0.01))
            )
          )
        );

process = ["gain" -> mixer];
```

Since in both channels we have “gain” widget, the modulation will affect both channels as we can see on the bloc-diagram figure 2.

For a more specific selection of the target widget, we can include the names of some or all of the enclosing groups of the target widget, as in `["v:chan 1/gain" -> mixer]`. Here, only the gain of channel number 1 will be modulated (see figure 3).

3. SYNTAX OF *WIDGET MODULATION*

In the preceding examples, we have provided an informal overview of *Widget Modulation*, aiming to offer a relatively intuitive understanding. Now, we will present a more formal description using syntactic rules in Backus-Naur Form (BNF), starting from the *Widget Modulation* expression itself:

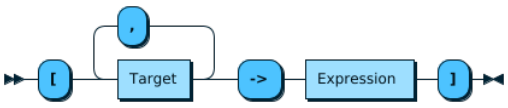


Figure 4: *WidgetModulationExpression.*

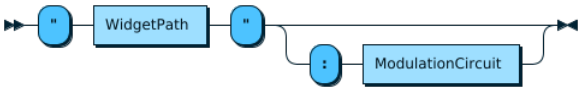


Figure 5: *Target.*

3.1. WidgetModulationExpression

```
WidgetModulationExpression
          ::= '[' Target ( ',' Target )*
              '->' Expression ']'
```

A *Widget Modulation* expression is composed of a list of target widgets and a modulated expression in which the target widgets are presumably used. The targets are separated by a comma sign (`,`). There must be at least one target. A *Widget Modulation* without targets is not allowed. The targets and the modulated expression are separated by the sign `->`, and the whole *Widget Modulation* expression is enclosed in square brackets.

3.2. Target

```
Target    ::= '"' WidgetPath '"'   ( ':'
    ModulationCircuit )?
```

A *Target* is composed of a *WidgetPath* that identifies the widget to modulate, and an optional *ModulationCircuit* that indicates how to combine the signal delivered by the widget with the modulation signal. If no ModulationCircuit is provided, the default is multiplication.

3.3. WidgetPath

```
WidgetPath
          ::= ( ( 'h:' | 'v:' | 't:' )
              GroupLabel '/' )* WidgetLabel
```

The *WidgetPath* is used to identify widgets in a modulated expression. It is a string composed of a widget label, optionally preceded by a sequence of group labels separated by slashes. The widget label is matched after removing any metadata. Group labels are used to disambiguate the widget to match, but they do not have to be consecutive.

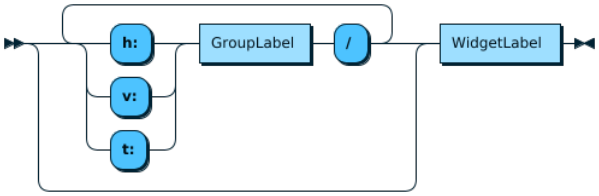


Figure 6: *WidgetPath.*

3.4. ModulationCircuit

The *ModulationCircuit* describe how to modulate the signal produced by the target widget. It can be any FAUST circuit with up to two inputs and one output. Three cases have to be considered:

- $(2 \rightarrow 1)$: A binary circuit with two inputs and one output, for example `+` as in `"Wet":+`. In this case, and only in this case, an additional input is created and the value of the widget is combined into the binary circuit before being used in the modulated expression.
- $(1 \rightarrow 1)$: A unary circuit with one input and one output, for example `*(lfo(10, 0.5))` as in `"Wet":*(lfo(10, 0.5))`. In this case, the value of the widget is routed into the unitary circuit before being used in the modulated expression, and no additional input is created.
- $(0 \rightarrow 1)$: A constant circuit with no input and one output, for example `0.75` as in `"Wet":0.75`. In this case all the occurrences of widget are simply removed and replaced by the constant circuit in the modulated expression. This is convenient to simplify a rich interface when some widgets that are not needed. In this case, no additional input is created.

3.5. Compilation

Widget Modulation is a circuit transformation primitive that is handled in the first phase of the compilation process. During this phase the FAUST program is evaluated to produce a circuit in *normal form* (a *flat* circuit composed solely of interconnected primitives).

Let's illustrate this process on our previous example:

```
["Wet":*(1+os.osc(0.1)/4) -> dm.
    freeverb_demo]
```

The compiler first evaluates `*(1+os.osc(0.1)/4)` and `dm.freeverb_demo` into their respective normal forms `c1` and `c2`. It then computes the normal form of `["Wet":*(1+os.osc(0.1)/4)-> dm.freeverb_demo]` by replacing every widget `w` labeled `"Wet"` occurring in `c2` with `w:c1`.

As a circuit transformation, *Widget Modulation* represents a new type of operation for FAUST, distinct from circuit composition operations (`:`, `,`, `\~`, `<:`, `:>`), which assemble existing circuits without transforming them. Despite this distinction, *Widget Modulation* fully aligns with the philosophy of a programming language dedicated to the description and implementation of audio circuits.

4. EXAMPLES OF MODULATION CIRCUITS

The ability to specify our own modulation circuits provides a lot of flexibility and expressiveness to *Widget Modulation*. Here, we give some examples of modulation circuits, some of which exploit the fact that it is possible to know the minimum and maximum values of a signal using the primitives `lowest` and `highest`, thereby ensuring that the signal after modulation remains within the widget's limits.

4.1. Frequency Modulation

Let's start with a simple example of frequency modulation showing the usage of simple additive and multiplicative modulations.

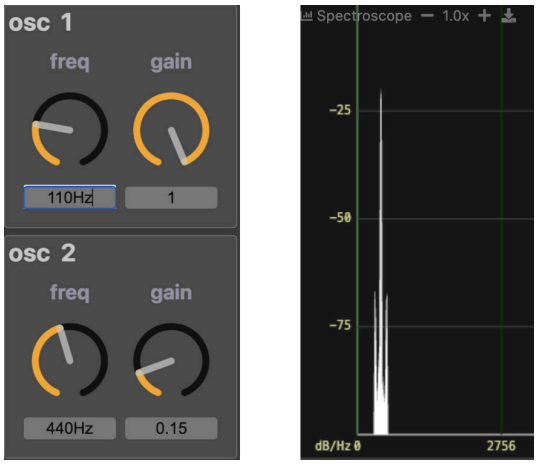


Figure 7: *Simple frequency modulation.*

We first define an oscillator with its own user interface consisting of two widgets, one controlling its frequency and the other its amplitude.

```
osc(n) = hgroup("osc %2n", os.osc(f) * g
    with {
     f = vslider("freq[scale:log][style:knob
         ][unit:Hz]",440,0.25,20000,1);
     g = vslider("gain[style:knob]", 0, 0,
         1, 0.01);
});
```

The `n` parameter is used in the group label to distinguish oscillators, so that we can use more than one. The minimal value for the frequency, 0.25 Hz, is deliberately outside the audible range in order to use the oscillator also as a LFO.

Let's look at a first example of frequency modulation using an addition as the modulation circuit:

```
process = osc(1) : ["freq":+ -> osc(2)];
```

The user interface and resulting spectrum are shown in figure 7. We recognize a FM spectrum, but the amplitude of the modulation oscillator is not high enough to obtain a rich spectrum. We can fix the problem by amplifying the modulation signal:

```
process = osc(1)*500 : ["freq":+ -> osc(2)
    ];
```

This gives us the spectrum figure 8.

Now let's add a third modulation stage, to modulate the "gain" of oscillator 1 and obtain a periodic variation in the spectrum:

```
process = osc(0)+1 : ["gain" -> osc(1)*500]
     : ["freq":+ -> osc(2)];
```

The resulting program can be try on line here

4.2. Advanced Modulation Circuits

In the previous examples, we did not account for the possibility of the modulated widget signal exceeding the limits of the initial widget. However, there are scenarios where this is important. Therefore, we now present several more advanced modulation circuits that ensure that the output value respects the range of values of

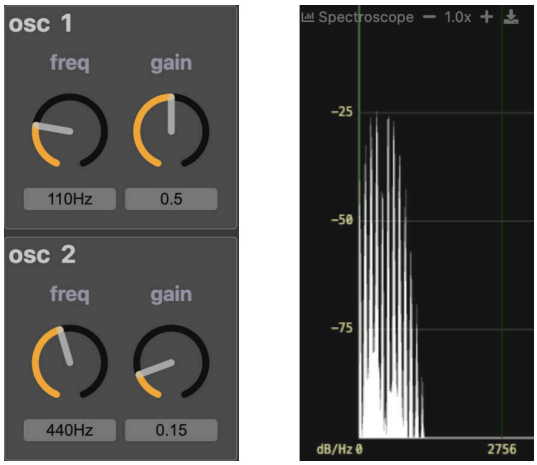


Figure 8: *Improved FM circuit.*

the widget `w`. As a reminder, we can determine the minimum and maximum values of a signal using the `lowest` and `highest` primitives.

`addp` and `add` modulations. The principle here is to add a value taken between two limits to the widget. A distinction is made between two cases, depending on the nature of the modulation signal. It could be a positive modulation signal, between 0 and +1, such as the signal from an envelope follower. Alternatively, it could be an audio signal, between -1 and +1, such as the signal from an oscillator. The obtained value is then clipped to the limits of the widget.

The first function, `addp`, adds to the widget `w` a value between `v1` and `v2` based on a positive modulation signal `m` ranging from 0 to +1:

```
addp(v1,v2,w,m) = max(lo, min(hi, w + v))
  with {
    lo = lowest(w);
    hi = highest(w);
    v = v1+m*(v2-v1);
  };
```

The second function, `add`, adds to the widget `w` a value between `v1` and `v2` based on an audio modulation signal `m` ranging from -1 to +1.

```
add(v1,v2,w,m) = addp(v1,v2,w,(m+1)/2);
```

`mulp` and `mul` modulations. The first function `mulp` multiply the widget value `w` by a factor between `f1` and `f2` based on a positive modulation signal `m` ranging from 0 to +1.

```
mulp(f1,f2,w,m) = max(lo, min(hi, w * f))
  with {
    lo = lowest(w);
    hi = highest(w);
    f = f1+m*(f2-f1);
  };
```

The second one `mulp` multiply the widget value `w` by a factor between `f1` and `f2` according to an audio modulation signal `m` ranging from -1 to +1.

```
mul(f1,f2,w,m) = mulp(f1,f2,w,(m+1)/2);
```

`mapp` and `map` modulations. These last two functions allow you to completely replace a widget (causing it to disappear from the user interface) with a value that varies between two bounds controlled by a modulation signal. The first function `mapp` replaces the widget by a value between `v1` and `v2` based on a positive modulation signal `p` ranging from 0 to +1.

```
mapp(v1,v2,w,p) = v1 + p*(v2-v1);
```

4.3. Revisiting the Frequency Modulation Example

We can apply these new functions to revisit our frequency modulation example. Let's start by defining an `md` environment containing all our modulation functions:

```
md = environment {
    addp(v1,v2,w,m) = max(lowest(w), min(
        highest(w), w + v))
    with {
        v = v1+m*(v2-v1);
    };

    mulp(f1,f2,w,m) = max(lowest(w), min(
        highest(w), w * f))
    with {
        f = f1+m*(f2-f1);
    };

    mapp(v1,v2,w,p) = v1 + p*(v2-v1);

    add(v1,v2,w,m) = addp(v1,v2,w,(m+1)/2);
    mul(f1,f2,w,m) = mulp(f1,f2,w,(m+1)/2);
    map(v1,v2,w,m) = mapp(v1,v2,w,(m+1)/2);

};
```

The revised frequency modulation example is as follows:

```
process = osc(0)
        : ["freq":md.add(-500,500) -> osc
           (1)];
```

It is interesting to note that the same target widget can be modulated several times by different modulation circuits. In the following *Widget Modulation* expression, two modulation circuits are applied to the same "freq" widget. It is first modulated by an `add(-600,600)`, and the result by a `mul(0.1,10)`:

```
["freq":md.add(-600,600), "freq":md.mul
    (0.1,10) -> osc(1)]
```

The circuit figure 9, shows how this double modulation is implemented.

To complete this section, here is a more elaborate example. It combines double frequency modulation—by an oscillator and by its own output signal via feedback—with the modulation of the "gain" widgets of these two oscillators by a third oscillator.

```
process = osc(0) : ["gain":md.add(0,0.5) ->
     (_ ,osc(1): ["freq":md.add(-600,600), "
    freq":md.mul(0.1,10) -> osc(2)])~@(200)
    ]<: _,@(5000);
```

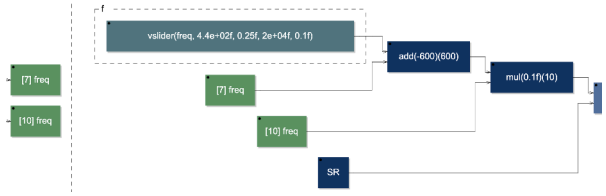


Figure 9: *Double modulation of the* `"freq"` *widget.*

5. CONCLUSION

This article introduces *Widget Modulation*, a novel extension to the FAUST programming language. This high-order primitive directly manipulates audio circuits, marking the first instance of such functionality within FAUST.

Inspired by the principles of modular synthesizers, *Widget Modulation* enables developers to seamlessly implement voltage control-type modulation into existing FAUST circuits. This allows for the redesign of user interfaces without necessitating direct access to the underlying source code.

While mastering the use of *Widget Modulation* may require time, its potential to significantly influence the development of FAUST libraries is substantial [5]. Users will be empowered to create libraries of modules, akin to those found in modular synthesizers, featuring rich and detailed user interfaces, with the assurance that a posteriori customization remains feasible. Furthermore, the extension opens avenues for the development of new libraries dedicated to modulation circuits.

6. REFERENCES

[1] Laurent de Wilde, *Les fous du son, d'Edison à nos jours*, Grasset, 2016.

[2] Suzanne Ciani, *Foreword*, Bjooks, May 2018.

[3] Yann Orlarey, Stéphane Letz, and Dominique Fober, *New Computational Paradigms for Computer Music*, chapter "Faust: an Efficient Functional Approach to DSP Programming", Delatour, Paris, France, 2009.

[4] John M. Chowning, "The synthesis of complex audio spectra by means of frequency modulation," *Journal of the audio engineering society*, vol. 21, no. 7, pp. 526–534, 1973.

[5] Julius O. Smith, "Signal processing libraries for Faust," in *Proceedings of Linux Audio Conference (LAC-12)*, Stanford, USA, May 2012.

ASSESSMENT OF SIMULATIONS IN FAUST AND TASCAR FOR THE DEVELOPMENT OF AUDIO ALGORITHMS IN ACOUSTIC ENVIRONMENTS

Felix Holzmüller
Institute of Electronic Music and Acoustics
University of Music and Performing Arts
Graz, Austria
holzmueller@iem.at

Christian Blöcher
Institute of Electronic Music and Acoustics
University of Music and Performing Arts
Graz, Austria
bloecher@iem.at

Alois Sontacchi
Institute of Electronic Music and Acoustics
University of Music and Performing Arts
Graz, Austria
sontacchi@iem.at

ABSTRACT

Testing real-time capable audio algorithms in a time-variant acoustic environment can be a tedious process, often requiring specialised hardware to fulfil latency constraints. In this publication, an alternative approach based on an acoustic scene simulation in TASCAR and signal processing using FAUST is assessed and tested against actual measurements. In dry acoustic environments, a good match of level distribution and spatial coherence at various evaluated points can be observed when modelling direct sound and early reflections. A simulation of an ANC algorithm shows also comparable performance to a measurement-based reference. Larger deviations are observed for reverberant and anisotropic scenarios. The proposed toolchain can be used for low-cost development and assessment of real-time capable algorithms in time-variant acoustic scenarios, as all parameters and virtual positions can be updated during operation.

1. INTRODUCTION

The development and testing of real-time capable audio algorithms can be a complex undertaking. Some algorithms like active noise control (ANC) are especially time critical and cannot be tested in the real world with common audio hardware and processing on an ordinary computer. Therefore, specialised hardware such as Field Programmable Gate Arrays (FPGAs) or digital signal processors (DSPs) are usually needed, which are often associated with considerable programming effort. Although dedicated rapid prototyping platforms are lowering the entry barrier by supporting higher-level code written in FAUST [1, 2, 3] or MATLAB/Simulink, they are generally costly and not always easily accessible. Another issue of testing systems in reality is a limited repeatability. Even small variations, for example inaccuracies in the geometrical arrangement of acoustic sources and receivers, can change the exact outcome of an experiment. A viable alternative to testing algorithms in the real world can be the combination of a time-variant, real-time capable acoustic simulation environment with an implementation of the algorithm under test in FAUST for a realistic simulation of its behaviour.

TASCAR[1] [4] is an open-source program for rendering acoustic scenes. Originally developed for hearing aid research, it features time-domain methods like a geometric room-acoustic simulation using an image source model (ISM), an implementation of a simple feedback-delay network (FDN) for diffuse reverberation, and various source and receiver types. A unique feature compared to other simulation tools is the ability to run simulations in real-time with inputs and outputs provided as JACK Audio[2] connections. All parameters of objects like position and orientation can be changed via Open Sound Control (OSC)[3] during operation. This enables researchers to conduct experiments in virtual scenes with variable positions in real-time.

As TASCAR only handles acoustic scene creation, the algorithm under test has to be implemented separately. Here FAUST's ability to compile applications for various architectures comes into play. FAUST has already been used successfully for implementing active noise control algorithms [2, 3]. After an algorithm is implemented in FAUST, it can be compiled as standalone JACK application with the `faust2ja[...]` tools. This application can receive signals from TASCAR, process them, and send them back to the acoustic scene simulation. Due to JACK's real-time capability, only a small delay equal to the audio device's block size is introduced.

The aim of this publication is to assess the capability of TASCAR and FAUST for testing and evaluation of time-variant audio algorithms compared to a co-simulation, based on recorded signals and measurements. In section 2, the acoustic scene simulation in TASCAR and the performed measurements are described. Section 3 compares acoustic properties of the recorded sound field and the acoustic scene simulation for different rooms and varying level of detail in the simulation. Section 4 showcases the use of TASCAR and FAUST for an ANC algorithm before summarising the findings in section 5. Measurements and code are openly accessible [5].

2. MEASUREMENTS AND SIMULATIONS

To assess the validity and accuracy of the simulation, acoustic properties and the ANC algorithm are evaluated for two different rooms:

- An acoustically treated measurement chamber with non-parallel walls, dimensions of approximately $5\,\mathrm{m} \times 4\,\mathrm{m} \times 3\,\mathrm{m}$, and a reverberation time of less than $50\,\mathrm{ms}$ above $300\,\mathrm{Hz}$ and less than $30\,\mathrm{ms}$ above $1\,\mathrm{kHz}$.
- A meeting room with an overall reverberation time of about $0.9\,\mathrm{s}$ and anisotropic acoustic properties (surfaces with absorbers, glass, plasterwall, and wooden cabinets).

Figure 1 shows the floor plan of both rooms.

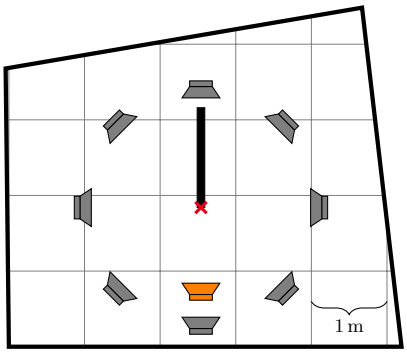
2.1. Measurements

In the measurement chamber and meeting room, respectively, an arrangement of eight and four Genelec 8020B loudspeakers was em-

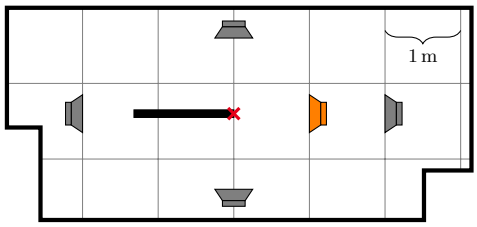
[1] https://www.tascar.org

[2] https://jackaudio.org/

[3] https://opensoundcontrol.stanford.edu



(a) Measurement chamber



(b) Meeting room

Figure 1: Floor plans of the evaluated rooms with positions of primary loudspeakers (gray), secondary loudspeakers (orange), and the microphone array (black bar) with indicated central position (red cross).

ployed as acoustic source. It was ensured that the level differences between loudspeakers were below $1\,\mathrm{dB}$.

The sound field in the rooms was recorded via an arrangement of 25 measurement microphones (NTi MA2230), uniformly spaced with a distance of $5.5\,\mathrm{cm}$ on a straight line, extending from the centre of the loudspeaker arrangement. The microphones are class 1 certified according to IEC 61672 and equipped with a $^1/_2$" capsule. After measurements, all microphones were calibrated using a Brüel & Kjær 4231 calibrator.[4] The microphone arrangement is shown in figure 2.

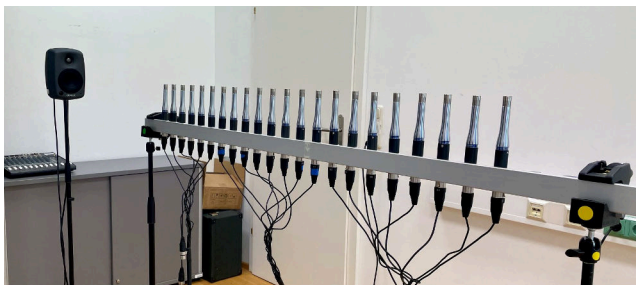


Figure 2: Microphone arrangement in the meeting room.

A DirectOut ANDIAMO AD/DA converter and microphone preamplifier provided in- and outputs for microphones and loudspeakers. The device, running at $44.1\,\mathrm{kHz}$ sample rate, was connected via MADI with an RME sound card (HDSPe MADI in the measurement chamber, Madiface Pro in the meeting room) to the measurement PC. Any occurring latency caused by the AD/DA conversion was measured and compensated on the recordings.

The sound field was captured for several primary source configurations with loudspeakers playing uncorrelated uniform white noise. In the first configuration all loudspeakers were playing, while in the second configuration only the loudspeaker pointed at by the microphone arrangement was active. With this source position defined as $0°$, an additional configuration with four active loudspeakers at $\{0°, 90°, 180°, 270°\}$ was recorded in the measurement chamber. Additionally, impulse responses between each loudspeaker and microphone were obtained using multiple exponential sine-sweeps [6]. For the ANC co-simulation, an additional loudspeaker acting as a secondary source positioned closer to the central position was set up ans measured as indicated in figure 1. This speaker was not present during measurements of the primary sound sources.

2.2. Simulation

The geometry and transducer locations of both assessed rooms were recreated in TASCAR for acoustic scene simulations. As receivers in simulations do not influence the sound field nor have any placement constraints, a finer grid of measurement positions with a spacing of $0.5\,\mathrm{cm}$, that also extended closer to the source at $0°$, was chosen.[5]

To assess the capability of TASCAR for acoustic sound field simulations, different degrees of accuracy were implemented. In the most basic case, only a free field scenario without any reflections is considered. In the next case, early reflections are simulated based on a 2^{nd} order ISM. Estimations for the frequency dependent absorption coefficients are used based on the surface materials. Values were used from the manufacturer's data sheet on the absorber material in the measurement chamber [7] and from a comprehensive, openly accessible database of common materials [8]. To increase the grade of detail further, in the next case the sources' directivity is adjusted. TASCAR has a `cardioidmod` source type, where a cardioid directivity pattern is achieved at a defined frequency. Towards lower frequencies an omnidirectional, towards higher frequencies a narrower directivity is modelled. To get an estimate for the frequency at which the cardioid pattern should be reproduced, the magnitude response of the physical loudspeaker was measured at $0°$ and $90°$ horizontal angle. The frequency at which both responses started to diverge by $6\,\mathrm{dB}$, $940\,\mathrm{Hz}$, was used as the value for the simulation. In the last case, a diffuse FDN reverb [9] with the reverberation time taken from the measurements is added. The different degrees of accuracy for all cases are summarised in table 1.

Case	Description
0	Free-field conditions
1	2^{nd} order image source model (ISM)
2	2^{nd} order ISM + source directivity
3	2^{nd} order ISM + source directivity + FDN reverb

Table 1: Description of the simulations' degree of accuracy.

[4] Due to a too high preamplifier gain in the measurement chamber, the calibrator's signal at $94\,\mathrm{dB}$ caused clipping. Therefore, the digitally settable amplifier gain has been reduced by $10\,\mathrm{dB}$ for calibration in this case. Subsequently, the lower gain was used for the meeting room recordings.

[5] The simulations are split into several segments with smaller numbers of receivers to reduce computational load and memory requirements.

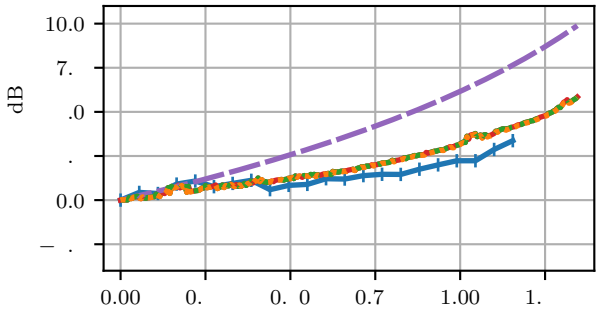
For the room acoustic evaluation, the acoustic scene was rendered offline on a high performance computing node for faster calculation as the input signal is known beforehand. Real-time simulations are performed on an ordinary notebook.

3. ROOM ACOUSTIC PARAMETERS

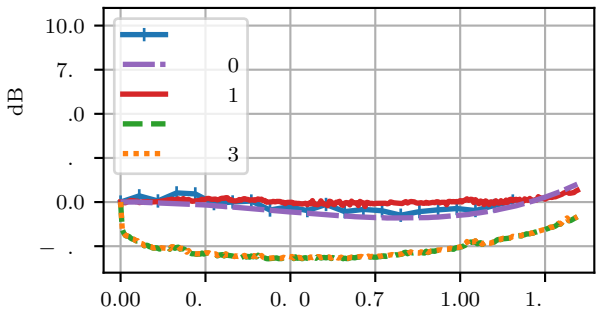
To assess the quality and accuracy of the recreated, simulated sound field, two properties are evaluated and compared to measurements for every source setup - the level distribution and the spatial coherence along the microphone arrangement.

3.1. Level Distribution

As quite simple yet effective measure, the level distribution is based on 30 s long recordings of uncorrelated uniform white noise for each source configuration (one, four or eight loudspeakers). After applying the individual calibration for each microphone, the respective root mean square (RMS) levels are calculated. The same noise realisation has also been used as input signal for the simulations in TASCAR for the respective source configurations.



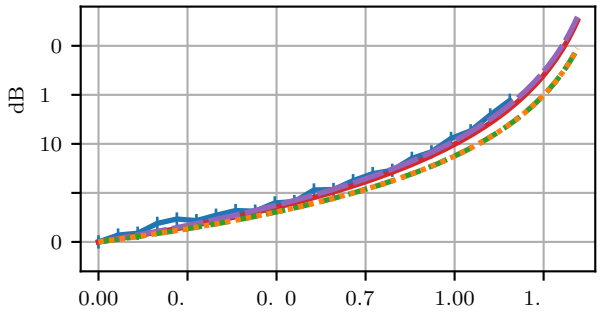
(a) 1 loudspeaker



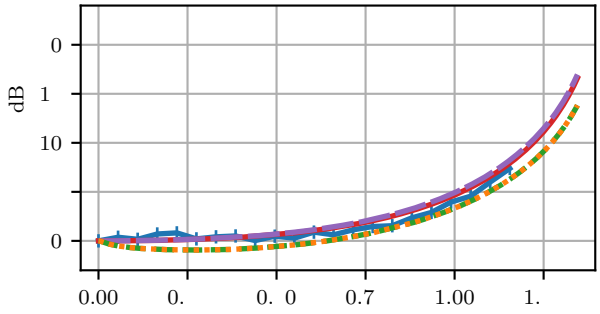
(b) 4 loudspeakers

Figure 3: Comparison of measured and simulated sound level distributions relative to the levels at the central position in the meeting room along the microphone array.

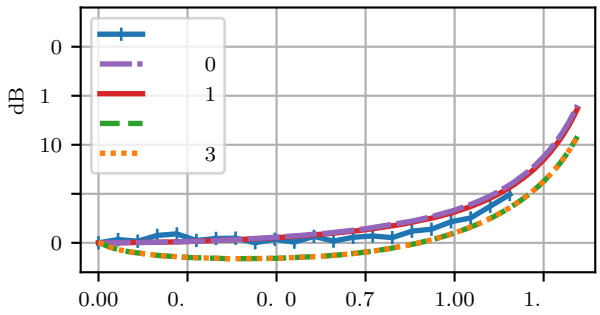
Figures 3 and 4 show the level distributions with different source configurations and rooms. All curves are normalised to the level at the central position. In the measurement chamber with



(a) 1 loudspeaker



(b) 4 loudspeakers



(c) 8 loudspeakers

Figure 4: Comparison of measured and simulated sound level distributions relative to the levels at the central position in the measurement chamber along the microphone array.

almost free field conditions towards higher frequencies, only minor differences between simulations and measurements can be observed. However, in the more reverberant meeting room, differences between the simulation cases are more obvious. With only one source active, simulation case 0 with only free field conditions shows a too steep rise in level, as the influence of the room reflections are neglected here. With four active sources in figure 3(b), cases 2 and 3 result in a decrease in level when leaving the central position.

Most likely `cardioidmod` sources exhibit too much directivity towards high frequencies, generating a too narrow sweet spot at the central position. Over all scenarios, the simulations with early reflections and omnidirectional source directivity (case 1) deliver the most accurate results. Small ripples in the measured levels around the central positions can be observed, most likely due to variances between the measurement microphones.

3.2. Spatial Coherence

A second parameter often used to describe sound fields is the spatial correlation between two points. It was initially proposed as additional parameter to describe the acoustics of a closed room [10]. Various authors derived analytical results for the correlation in various scenarios [11, 12, 13]; In the most notable finding by Cook et al. [10], it takes the form

$$R = \frac{\sin(kr)}{kr} \tag{1}$$

in a pure-tone diffuse sound field, where $k = \frac{2\pi}{\lambda}$ is the wave number with wavelength λ and r the distance between the two observed points.

Measuring the spatial correlation can be a tedious and time consuming process, as usually only a single frequency can be assessed at once. A faster, alternative approach is to play and record a broadband signal and calculate the coherence. It has been shown by Jacobsen and Roisin [14], that the coherence at the evaluated frequency corresponds to the squared correlation, if a sufficiently fine analysis bandwidth is chosen. With this technique, all excited frequencies can be evaluated using only a single measurement.

The coherence is analysed and compared for three different frequencies of $\{300, 1000, 1700\}$Hz, all of which lie above the Schroeder frequency [15] for both assessed rooms (approximately 60 Hz in the measurement chamber and 260 Hz in the meeting room). Figure 5 shows the results for the measurement chamber. Unsurprisingly, the coherence in these close to free field conditions is estimated perfectly as unity over all positions for a single source. Also for four and eight sources a good resemblance of the measurements can be observed over the whole assessed range. Interestingly, simulation case 0 without any early reflections performs best at 300 Hz. This could be explained by a sub-optimal choice of absorption coefficients for the ISM.

The coherence in the meeting room in figure 6 shows larger differences between measurements and simulations. While simulations perform relatively well at lower frequencies at positions close to the reference location, less resemblance is seen with increased frequency and distance. This behaviour can be explained by the lack of proper diffuse reverberation. Discrepancies between measurement and simulations with a single source at low frequencies are most likely the result of room modes which cannot be reproduced by a 2nd order ISM.

4. APPLICATION FOR ACTIVE NOISE CONTROL

To show the capabilities of the proposed toolchain for real-time processing, a co-simulation of an active noise control algorithm based on actual measurements is compared to a full simulation in TASCAR and FAUST. Before describing the details of the simulation setup, the basics of signal processing for active control are presented.

4.1. Feedforward Filtered-X Least Mean Squares Filter

ANC can be implemented using different fundamental designs. As this section only highlights the relevant portions of the algorithm for the investigated scenario, the interested reader may be referred to more detailed literature [16, 17].

A feedforward system can be used to control both deterministic and random disturbances. A time-advanced reference signal $x[n]$ is captured, filtered by a so-called control filter $W(z)$, and played back via transducers, also referred to as secondary sources. The control signal $u[n]$, the signal for the secondary sources, can therefore be described in z domain as

$$U(z) = W(z)X(z)\,, \tag{2}$$

where e.g. $X(z)$ corresponds to the z-transform of $x[n]$. The residual error signal at the listening position is now found as the superposition of the primary disturbances $d[n]$ and the control signal $u[n]$, whereby the alterations to the signal by the acoustic plant $G(z)$ between the secondary source and the listening position have to be considered as well. This results in

$$E(z) = D(z) + G(z)U(z) = D(z) + G(z)W(z)X(z) \tag{3}$$

for the z-transform of the residual error.

ANC often relies on adaptive filters to compensate for slight variations in the acoustic scenario. An ordinary least mean squares (LMS) filter cannot be used in this case, as the output of the control filter is altered by the acoustic transfer path $G(z)$ before summation with the disturbances [18]. However, a trick can be used to find a suitable structure. Assuming both the acoustic plant $G(z)$ and the control filter $W(Z)$ are in a modelling phase linear and time-invariant systems, the position of the two blocks can be switched due to their commutative property. Now the reference signal is first filtered by the plant, producing the so-called filtered reference signal $r[n]$, which is then used as input for the control filter. In time domain, this results in the error signal

$$e[n] = d[n] + \mathbf{w}^T\mathbf{r}[n]\,, \tag{4}$$

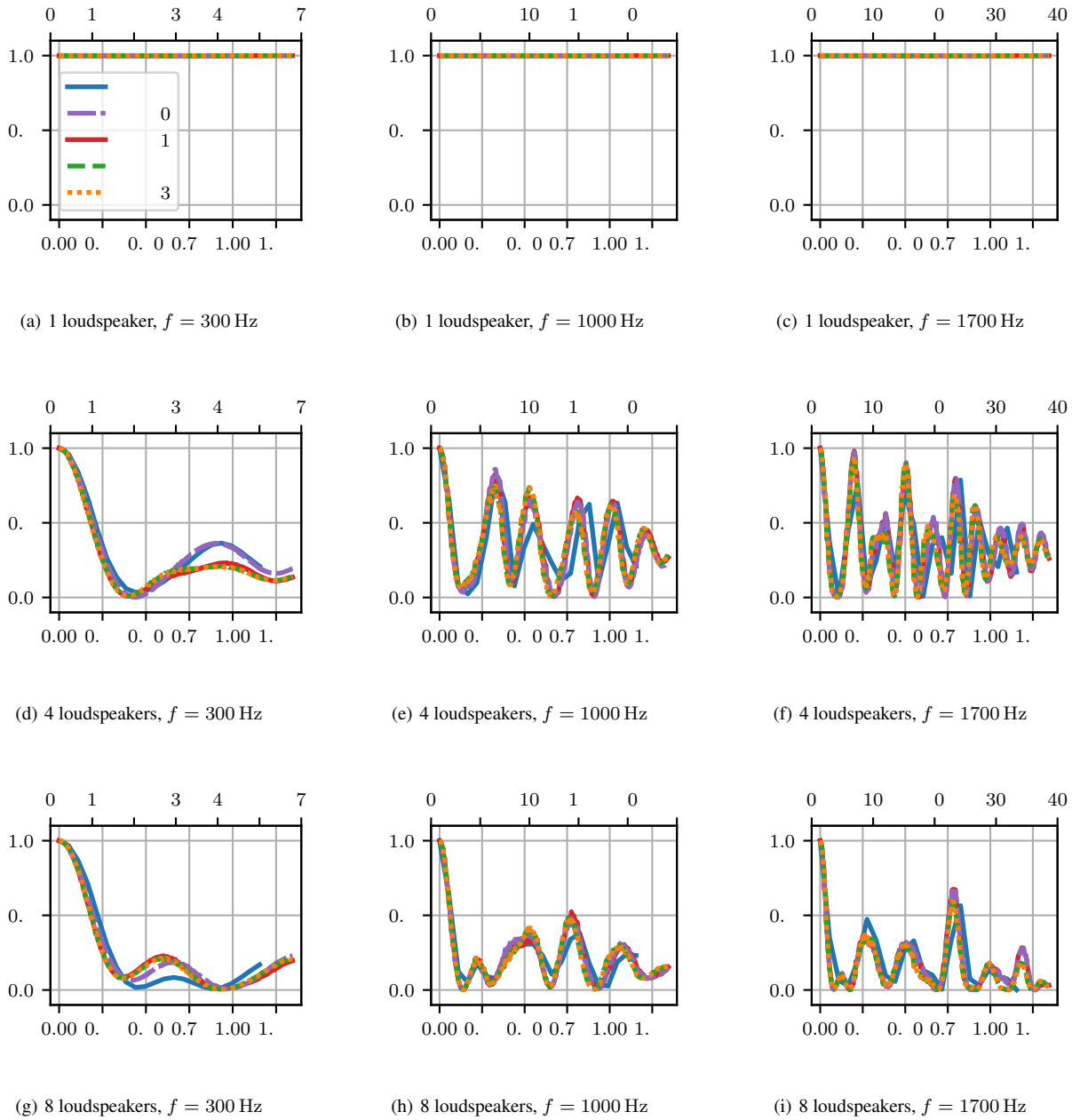
where $\mathbf{w} = [w_0, w_1, \ldots, w_{N-1}]^T$ holds the coefficients of the control filter, $\mathbf{r}[n] = [r[n], r[n-1], \ldots, r[n-N+1]]^T$ is a buffer of filtered reference signals, and N corresponds to the control filter order.

As the control filter output now directly contributes to the residual error signal, its coefficients can be adapted with an LMS approach as

$$\mathbf{w}[n+1] = \mathbf{w}[n] - \mu\mathbf{r}[n]e[n]\,, \tag{5}$$

where μ is the adaptation stepsize.

As in a real physical system plant and control filter cannot be switched, the reference signals are filtered for adaptation with an estimate or measurement of the plant response $\hat{G}(z)$. This algorithm is known as *filtered-x* or *filtered-reference LMS* (FxLMS) filter.



(a) 1 loudspeaker, $f = 300$ Hz

(b) 1 loudspeaker, $f = 1000$ Hz

(c) 1 loudspeaker, $f = 1700$ Hz

(d) 4 loudspeakers, $f = 300$ Hz

(e) 4 loudspeakers, $f = 1000$ Hz

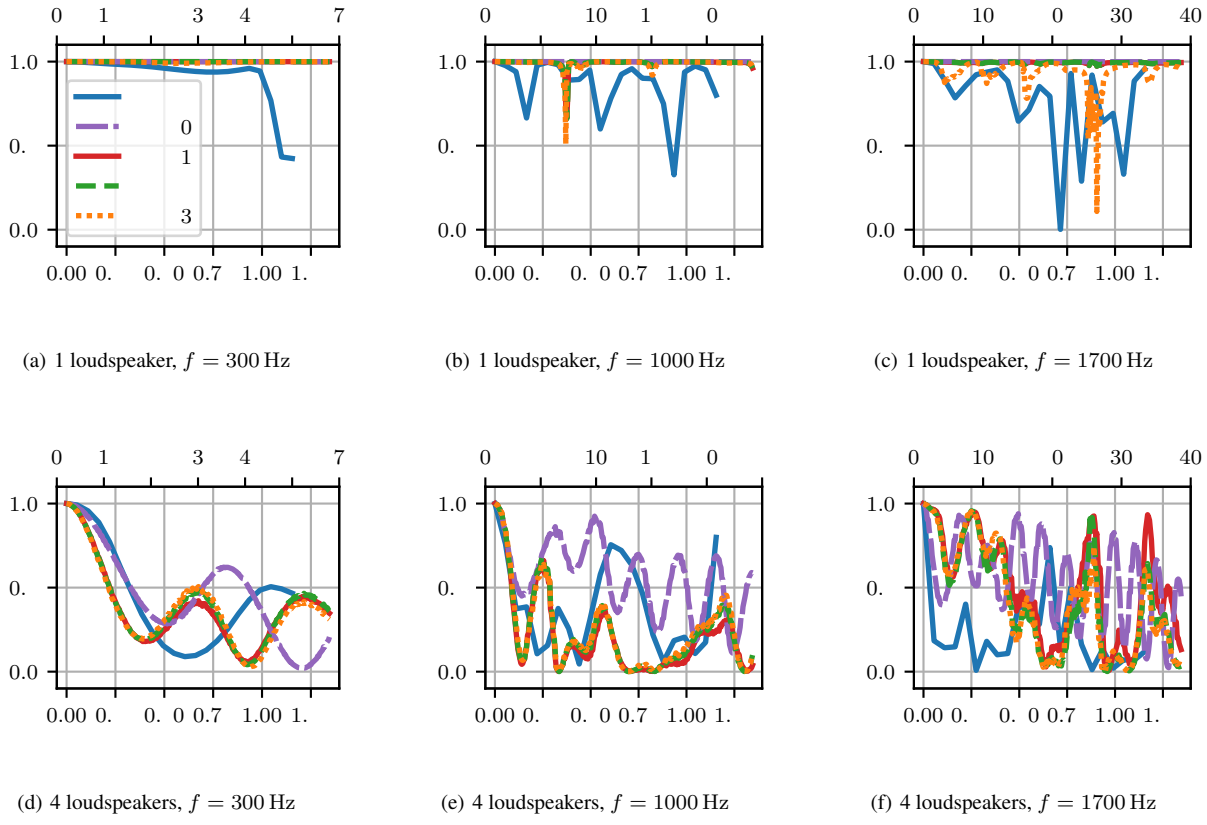
(f) 4 loudspeakers, $f = 1700$ Hz

(g) 8 loudspeakers, $f = 300$ Hz

(h) 8 loudspeakers, $f = 1000$ Hz

(i) 8 loudspeakers, $f = 1700$ Hz

Figure 5: Comparison of the spatial coherence of measured and simulated sound fields in the measurement chamber between the central position and points along the microphone array at distance r for analysis frequencies f with corresponding wavenumber k.



(a) 1 loudspeaker, $f = 300$ Hz

(b) 1 loudspeaker, $f = 1000$ Hz

(c) 1 loudspeaker, $f = 1700$ Hz

(d) 4 loudspeakers, $f = 300$ Hz

(e) 4 loudspeakers, $f = 1000$ Hz

(f) 4 loudspeakers, $f = 1700$ Hz

Figure 6: Comparison of the spatial coherence of measured and simulated sound fields in the meeting room between the central position and points along the microphone array at distance r for analysis frequencies f with corresponding wavenumber k.

4.2. Simulation Setup

The goal of the ANC experiment is to determine FAUST's and TASCAR's ability to perform simulations for audio algorithms. In this context, the real-time capability of standalone applications written in FAUST constitutes a vital prerequisite for the success of the combined simulation approach. As the focus lies on both feasibility and comparability to measurements, only a setup with a single primary source at 0° and a single receiver at the central position is assessed for both rooms and all degrees of detail for the simulations at 22 050 Hz sample rate. All experiments are performed on an ordinary notebook (Intel Core i7-13700H with 32 GB RAM, running Ubuntu Studio 24.04).

The FxLMS algorithm described in section 4.1 is implemented in FAUST. The secondary transfer path $\hat{G}(z)$ is obtained by rendering an impulse response in TASCAR between the secondary source and the receiver position. Both the adaptive filter and the plant $\hat{G}(z)$ are implemented as FIR filters with 128 taps[6] ($\widehat{=} 5.8\,\mathrm{ms}$). For more efficiency, the predelay, i.e. the acoustic time of flight between source and receiver, is cut from the secondary path and modelled as a simple delay. A white noise innovation signal is played back by pure data (pd). For optimal performance, it is used both as signal for the primary source in TASCAR and as reference signal $x[n]$ for the FxLMS algorithm. The secondary path filter $\hat{G}(z)$ and the adaptive filter itself are compiled as two separate programs. The FxLMS filter generates the control signal $u[n]$ for the secondary sources, which is routed to TASCAR as well. TASCAR renders the signal at the virtual microphone position sample by sample in real-time, based on the primary and secondary sources' signals with the selected degree of detail. The resulting residual error signal $e[n]$ is then recorded in pd and transmitted to the FxLMS filter for adaptation as well. Figure 7 shows the signal flow of the ANC simulation. For bulk processing and better repeatability, central control of the ANC experiment (e.g. initialisation of all applications and scenes, setting parameters via OSC) is implemented in Python, while the JACK connections are built automatically with the `jcmess`[7] utility.

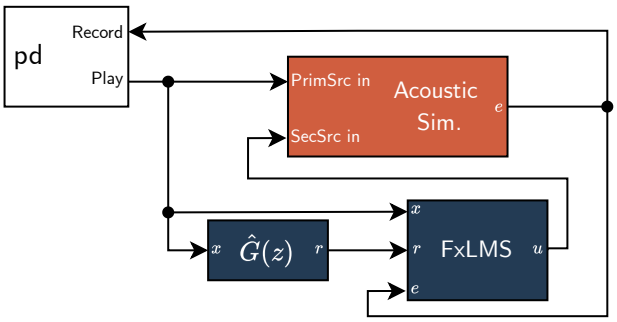


Figure 7: Signal flow of the ANC simulation using pure data (white), FAUST (blue) and TASCAR (orange).

All FAUST scripts are compiled as standalone JACK application by the `faust2jaqt` tool. The signals are routed using JACK Audio with the virtual dummy driver at 22 050 Hz sample rate and a block size of 16 samples. Due to the recursion of the error signal back into to the FxLMS block for adaptation and to pd for recording, a latency of one block (16 samples $\widehat{=} 0.726\,\mathrm{ms}$) is introduced on the error signal $e[n]$. This latency has to be taken into account for adaptation, meaning $r[n]$ has to be delayed by the same amount. For better comparability, the RMS output signal levels in TASCAR have been calibrated for each case to match those of the actual measurements.

For the measurement-based reference, the innovation signal is convolved with the impulse responses measured in the physical rooms to obtain the disturbance signal $d[n]$ at the central position. The disturbance signal as well as the innovation signal, used as reference signal $x[n]$, are processed in a Python co-simulation of the FxLMS ANC algorithm. The same set of parameters (step size μ, FxLMS filter order N, secondary path order) is chosen for the ANC algorithm in Python, the loopback delay caused by JACK in the TASCAR and FAUST simulation is modelled as well. For a realistic scenario, a full length version of the secondary path response $G(z)$ is convolved sample by sample with the calculated control signal $u[n]$ before summation with the primary disturbances $d[n]$ to get the error signal $e[n]$.

4.3. Results

Table 2 and figure 8 show a comparison of simulations and measurement based co-simulations in terms of error signal levels before and after adaptation. In a simulation with free field conditions (case 0), the residual error decays towards zero and approaches numerical limits. However, in a real room, even a measurement chamber, this perfect cancellation is not reached due to unavoidable reflections and reverberation. In the measurement chamber, simulation scenarios with at least early reflections (cases 1 to 3) exhibit more realistic results. A relatively similar level with engaged ANC is achieved this way compared to the co-simulation, although with about 5 dB to 6 dB more attenuation. It is also noticeable, that the simulation in FAUST and TASCAR adapts slower compared to the measurements. A look at the spectrum of the error signal before and after adaptation in figure 9 shows that the ISM in TASCAR causes an increased level towards low frequencies, as the extrapolated absorption coefficients in this frequency range exhibit lower values. This behaviour is intensified in the signal level after adaptation as spectral tilt, most likely as early reflections in the secondary path are not considered by the ANC algorithm with the chosen filter order.

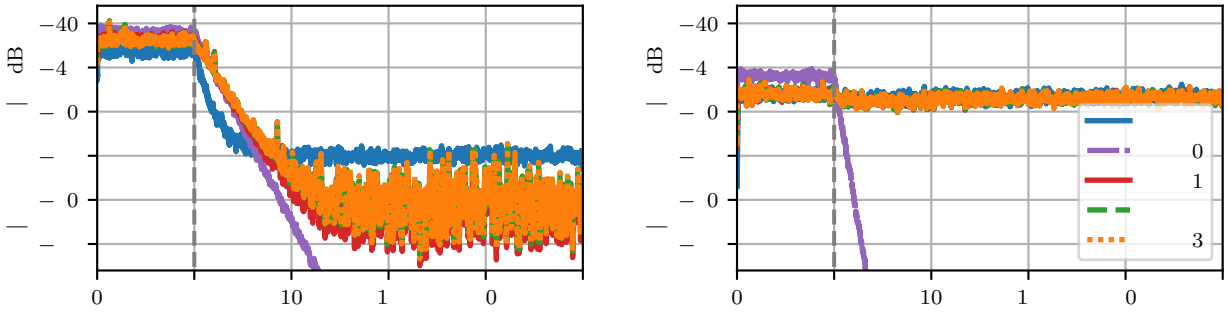
In the meeting room, results of both measurement and simulations with early reflections show no meaningful noise reduction. This is due to the fact, that early reflections with strong contributions cannot be controlled as they are exceeding the 128 samples length of both adaptive filter and modelled secondary path.

	Measurement Chamber		Meeting Room	
	ANC off	ANC on	ANC off	ANC on
Meas	−39.8	−51.6	−44.7	−44.8
Sim 0	−39.8	−103.3	−44.7	−122.1
Sim 1	−39.8	−57.9	−44.7	−45.1
Sim 2	−39.8	−57.2	−44.7	−45.1
Sim 3	−39.8	−57.2	−44.7	−45.1

Table 2: Comparison of ANC performance in terms of error signal root mean square (RMS) values in dB.

[6] Higher filter orders would lead to an unreasonably long build time of the adaptive filter in its current implementation.

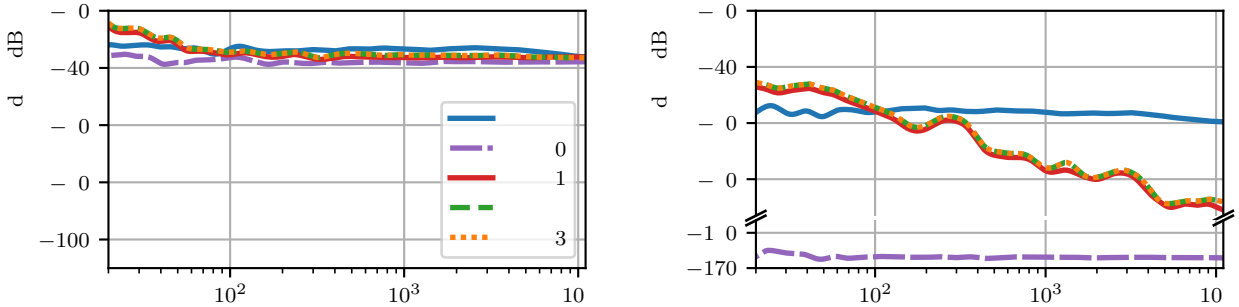
[7] `https://github.com/synthnassizer/jcmess`



(a) Measurement chamber

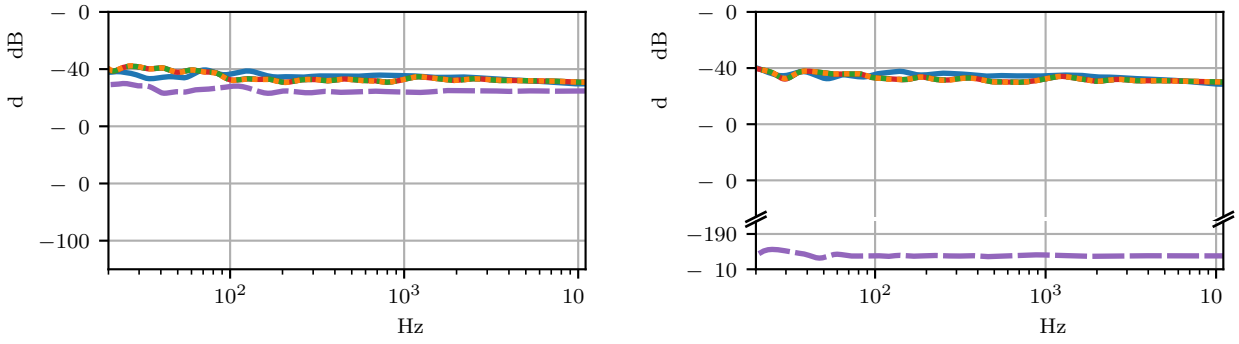
(b) Meeting room

Figure 8: Comparison of error signals for one noise realisation with ANC adaptation enabled from 5 s onwards.



(a) Measurement chamber, ANC off

(b) Measurement chamber, ANC on



(c) Meeting room, ANC off

(d) Meeting room, ANC on

Figure 9: Comparison of $^1/_3$ octave smoothed error magnitude spectra for one noise realisation with and without ANC.

5. CONCLUSION AND OUTLOOK

This study assesses the capability of combining FAUST and TASCAR for signal processing and acoustic scene simulations in order to develop and test real-time capable audio algorithms. Simulation results from these tools are compared to measurements in two real rooms: a measurement chamber with an extremely short reverberation time, and a meeting room with anisotropic acoustic properties. Room acoustic evaluations based on the level distribution along a microphone arrangement show a good match between measurements and simulations just considering early reflections and omitting source directivity. The spatial coherence is highly similar between simulations and measurements in the measurement chamber as well. Greater differences are observed in the more diffuse meeting room, most likely caused by the ISM order and lack of proper diffuse reverberation.

An active noise control algorithm is tested in TASCAR for acoustic scene simulation and FAUST for signal processing on an ordinary PC, using JACK Audio for signal routing. Compared to a measurement based co-simulation, a full scale simulation in TASCAR and FAUST exhibits comparable results when considering at least early reflections. However, the low filter order of the implemented FxLMS algorithm only permits the evaluation of relatively dry acoustic scenarios.

The full potential of a combined simulation approach with FAUST and TASCAR lies in time-variant simulations, as all parameters and virtual positions can be updated during operation via OSC. Future works may include the integration of head-tracking devices, for example to develop and improve position-dependent local ANC algorithms [19, 20].

6. REFERENCES

[1] Maxime Popoff, Romain Michon, Tanguy Risset, Yann Orlarey, and Stéphane Letz, "Towards an FPGA-Based Compilation Flow for Ultra-Low Latency Audio Signal Processing," in *Sound and Music Computing Conference*, Saint-Étienne, France, June 2022, https://inria.hal.science/hal-03805199.

[2] Loïc Alexandre, Pierre Lecomte, Marie-Annick Galland, and Maxime Popoff, "Feedback Acoustic Noise Control with Faust on FPGA: Application to Noise Reduction in Headphones," in *International Faust Conference*, Saint-Étienne, France, June 2022, https://hal.science/hal-03781085.

[3] Loïc Alexandre, Pierre Lecomte, and Marie-Annick Galland, "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, European Acoustics Association, https://hal.science/hal-04221590.

[4] Giso Grimm, Joanna Luberadzka, and Volker Hohmann, "A Toolbox for Rendering Virtual Acoustic Environments in the Context of Audiology," *Acta Acustica united with Acustica*, vol. 105, no. 3, pp. 566–578, May 2019.

[5] Felix Holzmüller, Christian Blöcher, and Alois Sontacchi, "Assessment of Simulations in Faust and Tascar for the Development of Audio Algorithms in Acoustic Environments - Code and Data," Sept. 2024, https://zenodo.org/records/13859827.

[6] Angelo Farina, "Simultaneous Measurement of Impulse Response and Distortion with a Swept-Sine Technique," in *AES 108th Convention*, Paris, Feb. 2000, https://www.aes.org/e-lib/inst/browse.cfm?elib=10211.

[7] BASF SE, "Basotect® G+ - Technical Information," https://soundproofwarehouse.com.au/website-images/TI-Basotect-G-eng-11-2011-1.pdf, Oct. 2011.

[8] Physikalisch-Technsiche Bundesanstalt, "Absorption Coefficient Database," https://www.ptb.de/cms/ptb/fachabteilungen/abt1/fb-16/ag-163/absorption-coefficient-database.html, May 2012.

[9] Michael Gerzon, "Synthetic Stereo Reverberation: Part One," *Studio Sound*, vol. 13, no. 12, pp. 632–635, Dec. 1971, https://www.worldradiohistory.com/Archive-All-Audio/Archive-Studio-Sound/70s/Studio-Sound-1971-12.pdf.

[10] Richard K. Cook, Richard V. Waterhouse, Raymond D. Berendt, Seymour Edelman, and Moody C. Thompson, Jr., "Measurement of Correlation Coefficients in Reverberant Sound Fields," *The Journal of the Acoustical Society of America*, vol. 27, no. 6, pp. 1072–1077, Nov. 1955.

[11] Chetlur G. Balachandran, "Random Sound Field in Reverberation Chambers," *The Journal of the Acoustical Society of America*, vol. 31, no. 10, pp. 1319–1321, Oct. 1959.

[12] Gerard C. J. Bart, "Spatial Crosscorrelation in Anisotropic Sound Fields," *Acoustica*, vol. 28, no. 1, pp. 45–49, Jan. 1973.

[13] Wing T. Chu, "Comments on the coherent and incoherent nature of a reverberant sound field," *The Journal of the Acoustical Society of America*, vol. 69, no. 6, pp. 1710–1715, June 1981.

[14] Finn Jacobsen and Thibaut Roisin, "The coherence of reverberant sound fields," *The Journal of the Acoustical Society of America*, vol. 108, no. 1, pp. 204–210, July 2000.

[15] Manfred R. Schroeder, "The "Schroeder frequency" revisited," *The Journal of the Acoustical Society of America*, vol. 99, no. 5, pp. 3240–3241, May 1996.

[16] Stephen J. Elliott, *Signal Processing for Active Control*, Signal Processing and Its Applications. Academic Press, 1 edition, 2001.

[17] Bernard Widrow and Samuel D. Stearns, *Adaptive Signal Processing*, Prentice-Hall, 1985.

[18] Dennis R. Morgan, "An analysis of multiple correlation cancellation loops with a filter in the auxiliary path," *IEEE Transactions on Acoustics, Speech, and Signal Processing*, vol. 28, no. 4, pp. 454–467, Aug. 1980.

[19] Stephen J. Elliott, Marcos Simón-Gálvez, Jordan Cheer, and Woomin Jung, "Head tracking for local active noise control," in *12th Western Pacific Acoustics Conference*, Singapore, Dec. 2015, https://eprints.soton.ac.uk/388063/.

[20] Woomin Jung, Stephen J. Elliott, and Jordan Cheer, "Combining the remote microphone technique with head-tracking for local active sound control," *The Journal of the Acoustical Society of America*, vol. 142, no. 1, pp. 298–307, July 2017.

Presentation of Workshops

abc.lib - News From Multichannel Audio Processing in Mixed Music

Alain Bonardi and Paul Goutmann

abc.lib is an open source library written in Faust language and distributed in the form of a Max package and a set of Pd objects.The Max and Pd distributions bring together objects for synthesis, multichannel and ambisonic processing, as well as numerous utilities for mixed music. In this workshop, we'll introduce the library's ecosystem. Step by step, we'll build musical situations involving synthesis, multichannel processing and spatial sound processing in ambisonics. We'll take advantage of special cases to explain the approach we've chosen to build a Max package for multichannel processing from a set of Faust code.

Introducing the Sound Corpus Survey: What's that I hear? Understanding Human Sound Description to Make Better Generative Audio Engines

Domenico Cipriani

This session consists of a 40-minute workshop and a 20-minute live performance. The workshop introduces *Phausto*, a lightweight Pharo library and API for sound generation and DSP programming using Faust. Participants will learn to install tools, understand Pharo syntax, and develop DSPs, making it accessible even to sound artists with little programming experience. The live coding performance features *MOOFLOD*, a Pharo tool developed through research by Evref and MINT, for creating music on-the-fly. It pairs sound generation (via Phausto) with visualizations of rhythmic patterns and synthesis manipulation. Afterward, audience members will assess their emotional engagement and understanding via a questionnaire.

Some useful links:

- `https://linktr.ee/lucretiomsp`
- `https://www.youtube.com/watch?v=NpOX_UoFV_g`
- `https://www.youtube.com/watch?v=7nLFGuZ-I5w&t=192s`

Presentation of Demos

Experiences with Rust + Faust

Leon Gnaedinger

I want to talk about my general experiences with integrating Rust in Faust and writing audio applications with it. This means I want to talk about how the Rust language feels around Faust, how I integrate Faust into my Rust projects and what difficulties and positives I think there are. Next I want to talk about the ecosystem of rust-faust integration in more detail. The options there are (nih-plug, faust-build, faust-types, faust-llvm). Maybe share some little tricks I learned. Besides the ecosystem around the Rust + Faust, I also want to share some insights about the broader audio dsp Rust ecosystem and share some libraries that I tend to use in my projects.

Demo of Stratus

Martin Bartlett and Landon McCoy

Demo of Stratus in general showcasing how it works. This is a guitar demo as well.

SHCdyna, a Dynamic and Interactive Application for Musical Performance

Ruolun Allen Weng and Christophe Lebreton

Developed by LiSiLoG, SHCdyna is a continuation and extension of the faust2smartphone project, first introduced at International Faust Conference (IFC) 2018. SHCdyna enhances the flexibility of creating musical applications through dynamic compilation, allowing musicians and developers to create, upload, and execute Faust projects in real-time on iOS devices. Building on the key concepts of customizable SHC (Smart Hand Computer) interfaces, motion control, and Faust's powerful digital signal processing (DSP) capabilities, SHCdyna opens up new possibilities for interactive music creation. This presentation will highlight the evolution introduced in SHCdyna and explore its potential applications in education and live performance.

Some useful links:

- SHCdyna Doc: `https://github.com/RuolunWeng/SHCdyna`
- faust2smartphone: `https://github.com/RuolunWeng/faust2smartphone`
- Smart Hand Computer: `https://www.lisilog.com/en/shc-2`

9 782959 791109

4th International Faust Conference

Proceedings