Expressive DSP for Audio Applications in C++20

Tobias Pisani, 201809111

Bachelor Report (15 ECTS) in Computer Science
Department of Computer Science
May 18, 2021
Advisor: Aslan Askarov

AARHUS
UNIVERSITY
DEPARTMENT OF COMPUTER SCIENCE

Abstract

Acknowledgements

Contents

Chapter 1

Introduction

When using DSP for sound design, whether it be software instruments, audio effects or any other kind of creative or practical sound processing, the program can usually be expressed as a pipeline, or composition of common, reusable blocks. These blocks could be anything from filters and delay lines to envelopes and oscillators, with a rich tappestry of algorithms in between, but most audio processing programs use the from the same set of basic, well known blocks, and then work with composition ►**continue**◄

1.1 Audio Processing

Audio is usually represented as a stream of samples, each either a floating or fixed point number, and for processing, the stream is split up into buffers of a fixed size. In most normal applications, the samples are 16, 24 or 32 bits, at sample rates of anywhere from 44.1kHz to 192kHz. Buffer sizes vary a lot depending on the real time requirements, from as low as 16 samples up to 4096. Larger buffers usually allow for better performance, while increasing the processing latency.

When dealing with multiple channels in one stream, such as would be the case for stereo or surround sound, the samples from each channel are usually interleaved, forming buffers of frames of samples (see Figure 1.1). These buffers are then passed to a processing callback one at a time by the host audio system, whether that is the OS directly, or in the case of audio plugins, the plugin host application. This asynchronous callback-based model is what is used by most real time audio frameworks, such as ALSA, Jack, VST

```
1 int process(float* input, float* output, unsigned nframes) {
2   for (int i = 0; i < nframes; i++) {
3     output[i * 2] = input[i];
4     output[0 * 2 + 1] = input[i];
5   }
6 }
```

Listing 1.1: An example process function that sends one input channel to two output channels. Numbers of channels are determined before registering the process function.

1.2 The Problem

►**At a High level, which issues are the existing solutions trying to solve, and what do i want to solve?**◄

In many cases audio processing is being done live, and has hard real-time requirements. As an example, when processing a stereo signal with a sample rate of 44.1 kHz, one has to be able to process 88 200 samples per

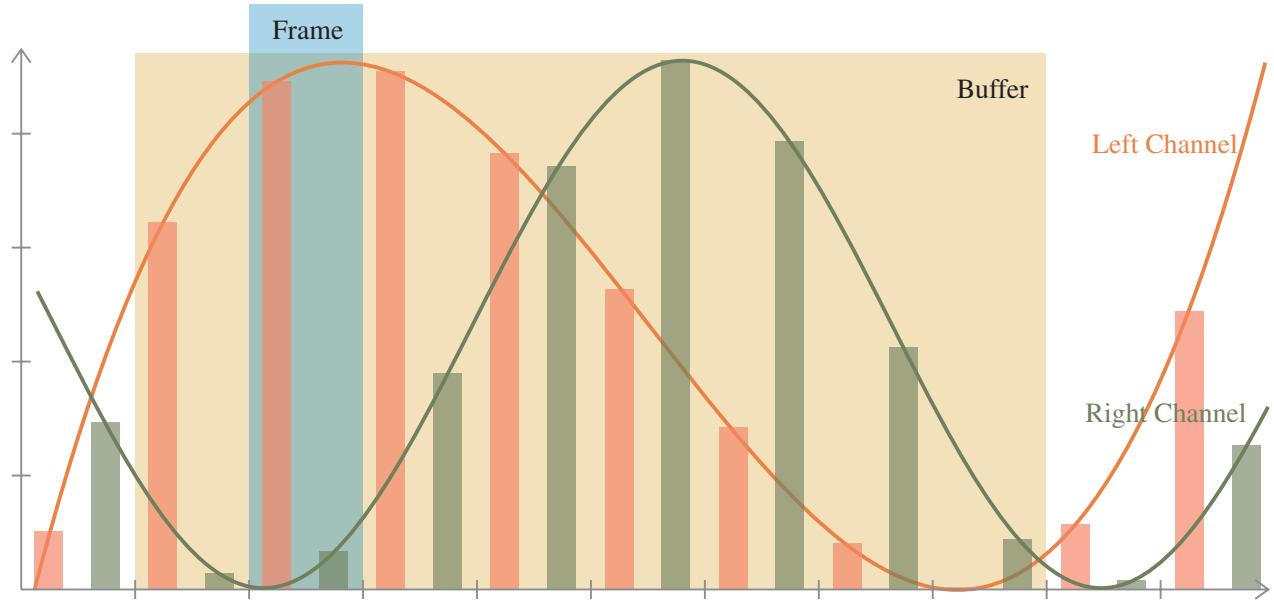


Figure 1.1: A stereo audio signal broken into buffers of 8 frames of two samples each.

second, which leaves around 11.3 µs per sample. For this reason, DSP code is usually written in a low-level high-performing language, like C or C++. However, as mentioned before designing audio processing is inherintly a largely compositional process, in which one rarely needs to worry about individual samples, buffer sizes, representation as fixed or floating point etc, except when implementing the low level components that make up the pipeline.

►What exact code do i want to be able to write?◄
►Set goals/constraints for evaluation◄
►Evaluation: Performance and comparisons of examples◄

1.3 The Example

As a case study, i will be using a very simple echo effect. It takes a single channel of input, which is delayed by some amount of samples, fed through a single-pole IIR ladder filter, decreased in gain, and fed back into the input to be delayed again. This can be seen in Figure 1.2

```
time_samples = 11025;
filter_a = 0.9;
feedback = 1.0;
dry_wet_mix = 0.5;

filter = (((_ * filter_a, _ * (1 - filter_a)) : + ) ~ _);
echo = (+ : @(time_samples)) ~ (filter * feedback);
process = _ <: (echo * dry_wet_mix) + (_ * (1 - dry_wet_mix));
```

Listing 1.2: Simple echo effect in faust, with time control, a 1-pole IIR filter, feedback gain and a dry/wet mix control. Paste into `https://faustide.grame.fr` to run the example.

This chapter contains 2915 characters and approximately 619 spaces = 0.84 standard pages

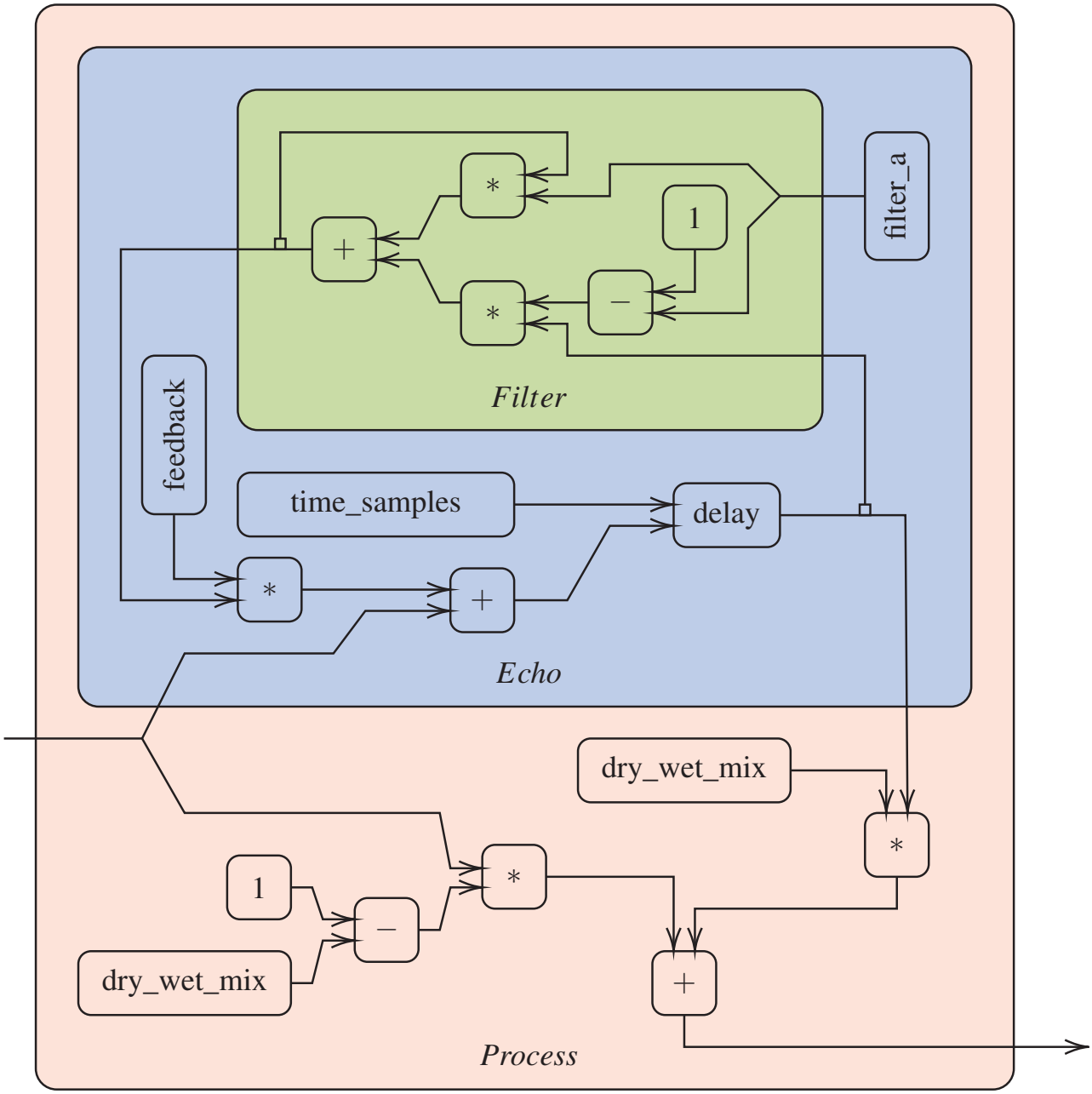


Figure 1.2: Block diagram of the echo effect.

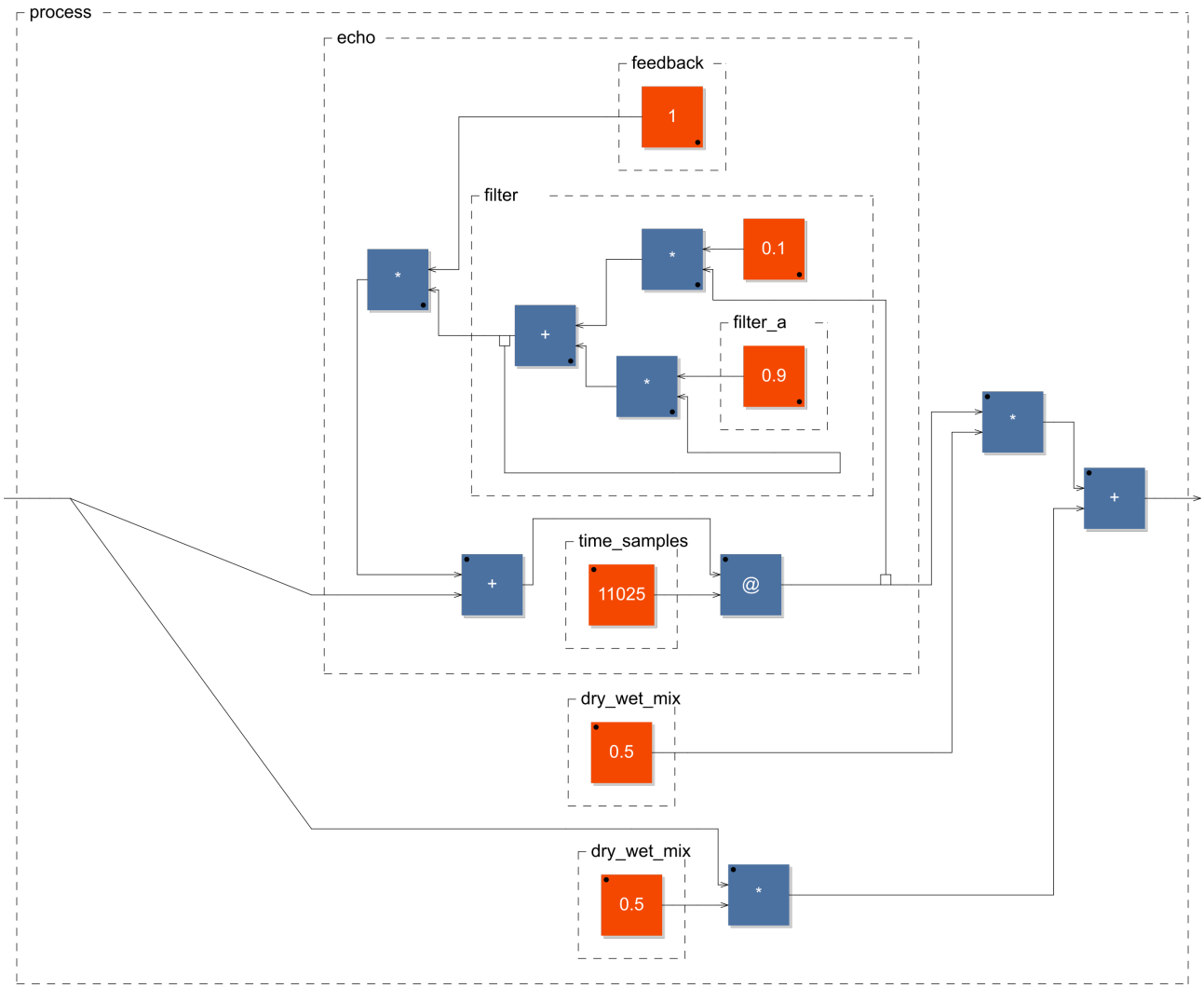


Figure 1.3: Block diagram generated by faust for code in Listing 1.2

Chapter 2

Review of Existing Material

►more examples◄

This chapter will cover some examples of existing solutions for writing DSP for audio applications in C++, along with examples of C++ libraries from other domains that attempt to solve similar issues.

2.1 Faust

Faust is described as a *"functional programming language for sound synthesis and audio processing with a strong focus on the design of synthesizers, musical instruments, audio effects, etc."[1]* Here, especially the notion of *functional* is interesting. Faust is written as a purely function processing pipeline, which allows for lazy evaluation and common subexpression elimination[1]. An example of basic faust code can be seen in Listing 2.1. This functional style also abstracts away nearly all implementation details with regards to buffers, code vectorization, loops etc, and represents purely the intent of the programmer in terms of the high level DSP algorithms. Thus, faust is a DSL for *specification* of DSP algorithms, which can then be compiled/transpiled to various targets, including C++, JAVA, WebAssembly etc. Other than just targeting multiple languages, Faust includes a system called *architectures*, which defines wrapper classes and files, allowing embedding in any system, such as smartphone applications, plugins for audio software, web apps etc.

```
1 filter = low_pass(5000, 0.2)
2 process = _ + std.noise * 0.5 <: filter, high_pass(100, 0.1)
```

Listing 2.1: Example faust code. Pass a mono signal in, add white noise scaled to 0.5, split the signal into two channels, and pass one channel through a low pass filter, and one through a high pass filter

Even though Faust might seem like the perfect solution at first sight, it has two major shortcomings.

Firstly, even with the *architecture* system, interoperability is between faust and the surrounding host code is still hard, especially when embedding in a larger system. Faust is fairly simple in terms of interop, and the architectures are defined in terms of functions describing user interfaces, which are awkward to use when the DSP is separated from the UI. For example, to add a volume input parameter to a faust program, one would use either the `vslider` or `hslider` functions, which represent the UI elements vertical and horizontal slider respectively. These functions also take a default value, minimum value, maximum value and step size, which are all options better suited for a separate UI implementation, especially when these options are not controlled directly by UI elements, but instead by some surrounding application code. These parameters are then (in C++) exposed to

[1] CSE: If the same subexpression appears in several places, the code is rearranged to only compute the value once

the host architecture as a string name for the slider and a reference to the float, leading to unchecked matching against strings, which is an area prone to errors.

Secondly, faust does not support resampling and multi-rate algorithms. This means that very important DSP algorithms like Fast Fourier Transform cannot be efficiently implemented. To make matters worse, there is no practical way to *inject* natively implemented algorithms into faust, or to step out of faust for an efficient implementation of some sub-algorithm. This is of course a fairly common issue with DSLs, where even if this is possible, it is often not easy and practical. For something like DSP it is very important to be able to hand-roll optimizations, especially of the often reused inner algorithms, where there exist implementations that are optimized many fold beyond what's possible in a high level of abstraction like faust.

Faust thus displays both the strengths and weaknesses of a high-level functional DSL: Composition of algorithms and designing signal chains is easy, and the code closely represents the block diagram and the mental model of the programmer, without being distracted by implementation details. However, this abstraction comes at the price of efficiency and ability to tweak the individual algorithms for platform-specific optimizations, along with being out of options when features are missing, like sample rate conversions, which are not only important for efficiency, but also sometimes for quality.

2.2 KFR

KFR Introduces itself as being a framework *"packed with ready-to-use C++ classes and functions for various DSP tasks from high-quality filtering to small helpers to improve development speed"*[2]. It is especially noteworthy for having a portable FFT implementation that often performs better than FFTW, *the Fastest Fourier Transform in the West*[3], but also offers high-performance implementations of many common DSP algorithms, like FIR filtering, IIR filtering, fast incremental sine/cosine generation, stereo conversions, delay lines, biquad filters etc[2]. All of these algorithms are optimized for various SIMD instruction sets, including SSE, AVX and NEON.

The algorithms in KFR can be applied to data in their custom `univector<T, N>` container, which essentially models a `std::span<T>`, `std::array<T, N>`, or `std::vector<T>` depending on the value of the parameter `N`. Using this class, the algorithms can be applied to data from many different sources, resulting in a system that can be easily integrated with any form of audio API, UI parameters etc. The user implements a `process` function, which takes a `univector<float, N>` or similar, and applies filters and algorithms to it as they please. Having access to the raw array of floats (in the form of a `univector`) also means it is very easy to do manual processing or combine it with functions from other libraries, even where only raw C APIs are available.

What is gained over Faust in performance and interoperability however, is lost in compositional expressivity. While KFR includes basic support for lazy evaluation of expressions involving `univectors`, it lacks the ability to describe the process function as a proper pipline of composed operations.

2.3 Eigen

Eigen is *"a C++ template library for linear algebra: matrices, vectors, numerical solvers, and related algorithms"*[4]. It is often used as the canonical example of Expression Templates in C++, and is especially interesting since it is one of the oldest most well-tested cases of this technology. It is currently used by projects like Google's Tensorflow[5] and MIRA, a middleware for robotics[6].

The Library provides, amongst other things, a simple and efficient interface for matrix operations, which are implemented using operator overloading and expression templates. This means syntax that closely represents the mathematical operators, and lazy evaluation where applicable. A very interesting part of how lazy evaluation is implemented in Eigen, is that it is deployed selectively. In some operations, the library decides at compile time

to internally evaluate some subexpressions into temporary variables instead of computing the whole operation at once. This shows the power of appropriately deployed expression templates. It can be a way to implement optimizations that could otherwise only be done at the compiler level, while staying within the ecosystem of the language, and providing an expressive API to the users.

Since Eigen was first released in 2006, a lot has changed on the C++ front, but being the industry standard that it is, and given the culture of not updating C++, the library still targets C++98. This means, that while the achievements are impressive and the library definitely holds up to todays standards, C++, and especially compile-time programming in C++, has changed a lot since. Given this, Eigen is a great example of what can be achieved using expression templates, but its internals may not the best place to look for inspiration on how to build an EDSL[2] with expression templates in C++.

2.4 C++20 Ranges

C++20[7] merged the long anticipated ranges proposal[8], which was significant in a number of ways. First of all, it added a lot of fairly simple utility functions and quality of life improvements to working with containers and algorithms, most notibly range-based versions of all standard algorithms. This means that functions like `std::sort` can now be called with a *range* as its first and only argument, instead of only being available to be called with separate begin and end iterators. The library defines the concept `std::range`, which models a type that has `begin()` and `end()` functions that return iterators[3]. This abstraction means that a `std::range` is simply an object that can be iterated over, like any standard container. These objects have always existed, but mostly the functions that use them have had to be passed begin and end iterators directly. This general shift from an iterator-based API to a range-based one, may at first glance appear trivial, but not only does it change `std::sort(vec.begin(), vec.end())` to `std::ranges::sort(vec)`, it also enables some interesting and very convenient syntax for more complex operations.

The second notable thing about the C++ ranges library, is that it was the first major part of the standard library to be designed with *concepts*, another C++20 feature[9], in mind. Concepts is the umbrella term often used to describe the whole system of static type constraints which was introduced in C++20[10], while concepts themselves is just a way to name these constraints. With Concepts and constraints came dedicated language features for selecting function overloads based on statically evaluated requirements on generic type parameters, something which had previously been done with library hacks and use of very esoteric aspects of the C++ template system called SFINAE[4]. There is a lot more to it than that, but for the sake of this section, this simplified view is enough. With a dedicated language feature came code *and* error messages that are both a lot easier to reason about. Using SFINAE would often result in hundreds, if not thousands of lines of error messages, where the source of the initial error could be extremely difficult to trace. This has even resulted in programmers having to write parsers for the error messages produced by C++ compilers[11]. All of this means, that while most of the code in the ranges library could be (and has been[12]) implemented before, with C++20 it, and code like it, has become a lot more feasible to write and maintain.

Thirdly, and most relevant to this report, The C++ Ranges library includes a new way of applying algorithms to ranges, and especially a new way to compose these algorithms. This is the system of *views* and their accompanying *range adaptors*. While the basic standard library algorithms and their range-based variants are applied eagerly, *views* apply algorithms lazily. As an example, lets take `std::views::transform(ints, to_string)`, which, given a range of `int`s and a function from `int` to `std::string`, returns a `std::views::transform_view<T, F>`, where `T` will be the concrete type of the `ints` range, and `F` will be the type of the function `to_string`. This view

[2] Embedded Domain Specific Language

[3] Technically, the end function returns a sentinel, which may be of a different type, but for the sake of this report i will refer to them both as iterators.

[4] Substition Failure Is Not An Error. The details of this are largely irrelevant to this report

is itself a range, that has captured the begin and end iterators of the range, and the function `to_string`. When iterating over the resulting view, upon each dereference of an iterator, the underlying iterator into the `ints` range will be dereferenced, and passed through `to_string`. This means, while `ints` is a range of integers, `std::views::transform(ints, to_string)` becomes a lazilly computed range of strings, which could in turn be passed to other views, which would also be lazilly evaluated. As an added bonus, the ranges library provides an overloaded `|` (pipe) operator to allow this composition, and with that, code like Listing 2.2 can be written. It is worth noting, that this code, specifically line 4 of Listing 2.2 comes very close to some of the syntax of faust, in that a high level, simple syntax, is used to describe a pipeline that is evaluated vertically instead of horizontally.

```
1 std::string to_string(int);
2 bool is_even(int);
3
4 std::vector<int> ints = {0, 1, 2, 3, 4, 5, 6};
5 for (std::string s : ints | std::views::filter(even) | std::views::transform(to_string)) {
6     std::cout << s << ' ';
7 }
```

Listing 2.2: Example of composition of views. Prints "0 2 4 6". Implementations of supporting functions omitted.

2.5 Conclusion

In this chapter, I described two existing solutions for DSP in audio applications. Faust provides a DSL for composing DSP algorithms, and while the syntax is highly expressive, a separately compiled DSL brings with it issues of integration, versatility and performance. KFR includes highly performant and versatile implementations of the algorithms, but ends up lacking in expressive syntax for composition, making the process of building complex applications from basic algorithms cumbersome. There are many other relevant DSP frameworks and libraries[5] that I will not go into here, but they tend to share the shortcomings of at least one of these systems.

I also covered Eigen and C++ Ranges, which aim to solve some of these issues of expressivity in other domains, i.e. linear algebra and composition of algorithms on containers respectively. In the rest of this report I will try to apply the technologies of these two solutions on the domain of DSP in audio applications, with the goal of proposing a solution to the issues posed by Faust and KFR respectively.

This chapter contains 10961 characters and approximately 2190 spaces = 3.13 standard pages

[5] **►List other relevant DSP frameworks◄**

Chapter 3

Expression Templates

►A Basic introduction to expression templates as an AST encoded in an object, where the structure of the expression is represented by recursive composition of types◄
►Introduce Bachelet/Yon (https://hal.archives-ouvertes.fr/hal-01351060/document) as a reference point for this section◄

3.1 Expression Templates with Concepts

►Introduce my design of expression templates◄
►Consider: Compare to basic design that doesnt have the AST / Visitor separation◄

3.2 Fixing operands

►Consider glossing over this section. While it is important to the implementation, it is not very important to the report, and I can probably refer directly to Bachelet/Yon, since the implementation will be very similar◄

Nope, the implementation is not similar. It is done a lot simpler, by just using a `fixed_t` type trait

3.3 Operator Overloading

►Basics of how operator overloading is used◄
►Mention of boxing literals, as well as marking types as valid operands◄

3.4 Evaluating with the Visitor Pattern

►How is this different from Bachelet/Yon?◄
►Plan: Based more on function overloading since C++20 concepts allows those to be more powerful, and this is an easier way to add visitor specializations◄

3.5 Type erasure

►Motivation? In practice this is important, but could it be left out of the report completely or partially?◄
►How is this different from Bachelet/Yon?◄

►Plan: `DynExpr<VISITORS..., TYPE>`, where `TYPE` might represent the full signature of the visitor, just the return type of the expression or something◄

3.6 (Potential) Usage

►Maybe not as a section like this, either move it out into the next chapter, or splice it into the previous sections where applicable◄
►The visitor based approach allows for other types of evaluations, like debug printing, or even a faust-style box diagram renderer. It will (probably) also be the basis of how we do optimizations, by adding platform-specific optimization passes, though this could also be done by specializing the evalution function templates.◄

This chapter contains 1594 characters and approximately 306 spaces = 0.45 standard pages

Chapter 4

Block Diagrams

►Introduce block diagrams◄

4.1 Algebra of Blocks

In the 2002 paper *An Algebraic approach to Block Diagram Constructions*[13] Orlarey et al introduce a series of five basic block diagram operations. In [14] they expand with two extra compositional operations, split and merge. The following section will give a short introduction to this algebra of blocks, and then go into details on how I implemented them in C++.

A block is a computational unit, which takes a certain number of input signals, and returns a number of output signals. They model impure functions of the form $\mathbb{S}^n \to \mathbb{S}^m$, that may have *memory*, i.e. depend on input from previous iterations. Signals are continuous streams of data at a fixed rate, which for the scope of this project is limited to a single type, `float`. 32-bit floating point numbers are a fairly common datatype for DSP operations, although work is also being done with fixed-point numbers.►cite◄

Given a set $\mathbb{B}$ of primitive domain-specific blocks, we describe a block diagram $d \in \mathbb{D}$ as terms of the language $\mathbb{D}$

$$
\begin{aligned}
d, d_1, d_2 \in \mathbb{D} ::= {} & b \in \mathbb{B} \\
& \mid \text{ident} \\
& \mid \text{cut} \\
& \mid \text{sequential}(d_1, d_2) \\
& \mid \text{parallel}(d_1, d_2) \\
& \mid \text{recursive}(d_1, d_2) \\
& \mid \text{split}(d_1, d_2) \\
& \mid \text{merge}(d_1, d_2)
\end{aligned}
$$

Faust (and [13]) uses single-character operator syntax for these operations, but since the same syntax cannot be achieved exactly in C++, I will be referring to them by their names as prefix functions.

Any element $d \in \mathbb{D}$ has a number of input ports $\textbf{ins}(d)$, and $\textbf{outs}(d)$. These values must be predefined for the primitive blocks $\mathbb{B} \cup \{\text{ident}, \text{cut}\}$, and can be computed for the block composition operations, based only on the

d	Faust Syntax	EDA Syntax	$\mathbf{ins}(d)$	$\mathbf{outs}(d)$
ident	`_`	`_`	1	1
cut	`!`	`cut`	1	0
sequential(d_1,d_2)	`d1 : d2`	`d1 | d2`	$\mathbf{ins}(d_1)$	$\mathbf{outs}(d_2)$
parallel(d_1,d_2)	`d1 , d2`	`d1 , d2`	$\mathbf{ins}(d_1)+\mathbf{ins}(d_2)$	$\mathbf{outs}(d_1)+\mathbf{outs}(d_2)$
recursive(d_1,d_2)	`d1 ~ d2`	`d1 % d2`	$\mathbf{ins}(d_1)-\mathbf{outs}(d_2)$	$\mathbf{outs}(d_1)$
split(d_1,d_2)	`d1 <: d2`	`d1 << d2`	$\mathbf{ins}(d_1)$	$\mathbf{outs}(d_2)$
merge(d_1,d_2)	`d1 :> d2`	`d1 >> d2`	$\mathbf{ins}(d_1)$	$\mathbf{outs}(d_2)$

Table 4.1: Basic block diagram compontents and their properties and syntax

ins and **outs** of the operands.

4.1.1 Basic block operations

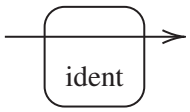
Each of these seven block operations is described in detail by the FAUST authors in [13] and [14], so here I will only give a brief introduction to each one, along with an example illustration, and

Identity

The ident block is the simplest block - it simply takes one input signal, and outputs that same signal untouched.

$$\frac{}{\text{ident} : \mathbb{S} \to \mathbb{S}}$$

$$\text{ident}(x) = x$$

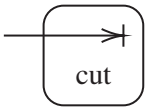


Cut

The cut block takes one input signal and outputs nothing. It can be very useful for discarding signals when composing blocks.

$$\frac{}{\text{cut} : \mathbb{S} \to \emptyset}$$

$$\text{cut}(x) = ()$$

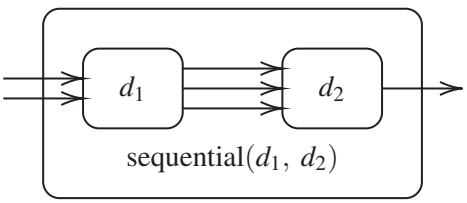


Sequential block composition

The simplest composition of two blocks is passing the outputs of one block to the inputs of another in sequence. It requires the number of outputs of the first block to equal the number of inputs on the second. Faust has defined semantics for when this is not the case as well, but since those cases can all be covered by combinations of sequential and parallel compositions, they have been left out here for simplicity.

$$\frac{d_1 : \mathbb{S}^n \to \mathbb{S}^p \qquad d_2 : \mathbb{S}^p \to \mathbb{S}^m}{\text{sequential}(d_1,d_2) : \mathbb{S}^n \to \mathbb{S}^m}$$

$$\text{sequential}(d_1,d_2)(\mathbf{x}) = d_2(d_1(\mathbf{x}))$$

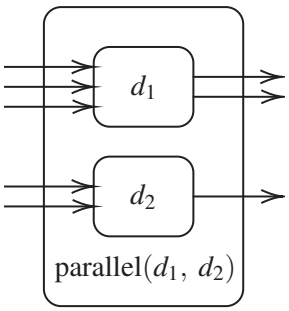


Parallel block composition

The parallel composition of two blocks can be intuitively seen as a concatenation of their input and output signals, resulting in a block where the two components are evaluated separately on their own segments of the input.

$$\frac{d_1 : \mathbb{S}^{n_1} \to \mathbb{S}^{m_1} \qquad d_2 : \mathbb{S}^{n_2} \to \mathbb{S}^{m_2}}{\text{parallel}(d_1, d_2) : \mathbb{S}^{n_1+n_2} \to \mathbb{S}^{m_1+m_2}}$$

$$\begin{aligned}\text{parallel}(d_1, d_2)(\mathbf{x}) = (&d_1(x_1, \ldots, x_{n_1}),\\ &d_2(x_{n_1+1}, \ldots, x_{n_1+n_2}))\end{aligned}$$



Recursive block composition

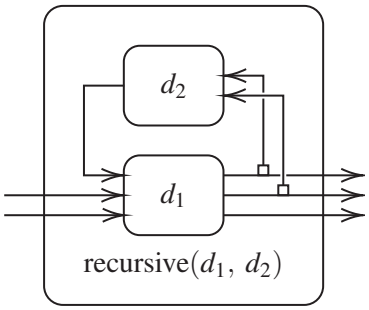
The recursive block composition is the most complex. Its purpose is to create cycles in the block diagram, by allowing a block to access the output it generated in the previous iteration. The outputs of d_1 are connected to the corresponding inputs of d_2, and the outputs of d_2 are connected to the corresponding inputs of d_1. The inputs to the composition are the remaining inputs to d_1, and the outputs are all outputs of d_1.

Since the recursion requires a cycle, the output from d_1 that is passed to d_2 is delayed by one sample, i.e. by one iteration. On the illustrations, this is denoted by a small square on the connection.

$$\frac{d_1 : \mathbb{S}^{n_1} \to \mathbb{S}^{m_1} \qquad d_2 : \mathbb{S}^{n_2} \to \mathbb{S}^{m_2} \qquad m_2 \leq n_1 \qquad n_2 \leq m_1}{\text{recursive}(d_1, d_2) : \mathbb{S}^{n_1-m_2} \to \mathbb{S}^{m_1}}$$

$$\text{recursive}(d_1, d_2)(\mathbf{x}) = d_1(d_2(x'_1, \ldots, x'_{n_2})_1, \ldots d_2(x'_1, \ldots, x'_{n_2})_{m_2},)$$

►what would be the best notation for these?◄

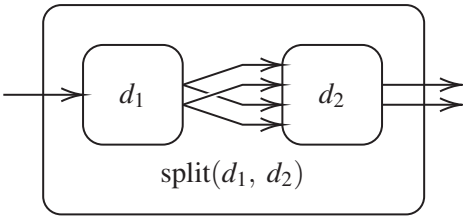


Split block composition

The split composition is used to sequentially compose blocks where the first one has fewer outputs than the second has inputs. The output signals are connected by repeating the entire output tuple the apropriate number of times, and this number is required to be an integer. This means $\textbf{ins}(d_2)$ must be an exact multiple of $\textbf{outs}(d_1)$.

$$\frac{d_1 : \mathbb{S}^{n} \to \mathbb{S}^{p} \qquad d_2 : \mathbb{S}^{p*k} \to \mathbb{S}^{m} \qquad k \in \mathbb{N}}{\text{split}(d_1, d_2) : \mathbb{S}^{n} \to \mathbb{S}^{m}}$$

$$\begin{aligned}\text{split}(d_1, d_2)(\mathbf{x}) = d_2(&d_1(\mathbf{x})_{1 \bmod m_1}, \ldots,\\ &d_1(\mathbf{x})_{n_2 \bmod m_1})\end{aligned}$$



Note that $\text{split}(d_1, d_2)$ is equal to $\text{sequential}(d_1, d_2)$ when $k = 1$

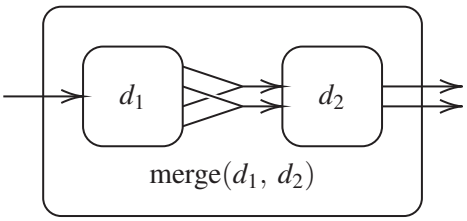
Merge block composition

Merge composition is the inverse operation of split composition, i.e. it is used to sequentially compose two blocks where the first one has more outputs than the second one. It places similar restrictions on d_1 and d_2 as split composition, i.e. it requires $\mathbf{outs}(d_1) = \mathbf{ins}(d_2) * k$, where k is an integer.

When multiple outputs from d_1 are connected to a single input on d_2, the signals are summed. Like split composition, $\text{merge}(d_1, d_2)$ is also equivalent $\text{sequential}(d_1, d_2)$ when $k = 1$.

$$\frac{d_1 : \mathbb{S}^n \to \mathbb{S}^{p*k} \qquad d_2 : \mathbb{S}^p \to \mathbb{S}^m \qquad k \in \mathbb{N}}{\text{split}(d_1, d_2) : \mathbb{S}^n \to \mathbb{S}^m}$$

$$\text{merge}(d_1, d_2)(\mathbf{x}) = d_2\left(\left(\sum_{i=0}^{m_1/n_2-1} d_1(\mathbf{x})_{i \cdot n_2+1}\right), \ldots, \left(\sum_{i=0}^{m_1/n_2-1} d_1(\mathbf{x})_{i \cdot n_2+n_2}\right)\right)$$



4.1.2 Domain Specific Blocks

Arithmetic

Memory

Delay

Ref

Ref

4.2 C++ Implementation

When implementing this algebra of blocks in C++, I used the basic concepts of expression templates to separate building an AST from evaluating it. This has the advantage that multiple visitors can be implemented for the AST, such as a basic evaluator, an optimizer, a buffer-based evaluator or even one that outputs a visual representation of the AST, like fausts block diagram generator.

The AST is represented by nodes that satisfy the concept ABLOCK, which just requires a block `T` to inherit from `BlockBase<T, T::in_channels, T::out_channels>`. `BlockBase` is a CRTP [1] base class, and provides `T::in_channels`, `T::out_channels`, and the function call operator overload required for currying.

4.2.1 `BlockBase` and `ABlock`

4.2.2 Literals

4.2.3 Currying

4.2.4 Repeat

4.2.5 Operator overloads and shorthand syntax

4.2.6 `evaluator`

For simplicity, I assume `float` for all signals.

[1] Curriously Recurring Template Pattern, see https://en.wikipedia.org/wiki/Curiously_recurring_template_pattern

4.2.7 `Frame`

This chapter contains 4287 characters and approximately 914 spaces = 1.24 standard pages

Chapter 5

Resampling

5.1 Aliasing

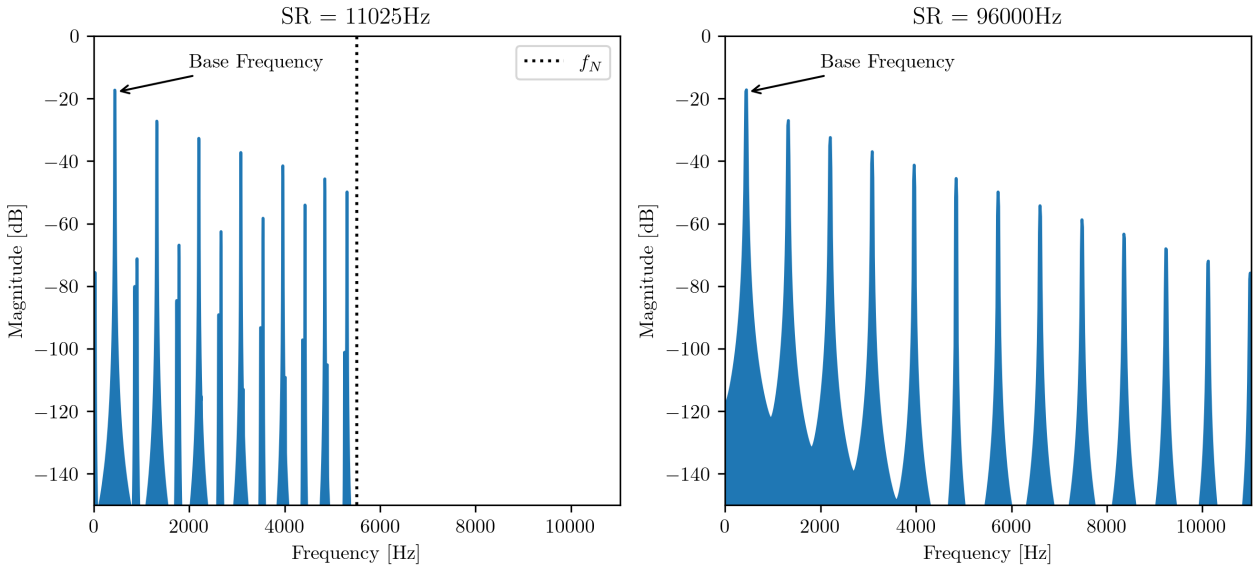


Figure 5.1: A 440 Hz sine wave passed through $\omega(x) = \tanh(8x) \cdot \frac{1}{8}$ at two sample rates. The reflection around the Nyquist frequency can very clearly be seen at 11 025 Hz.

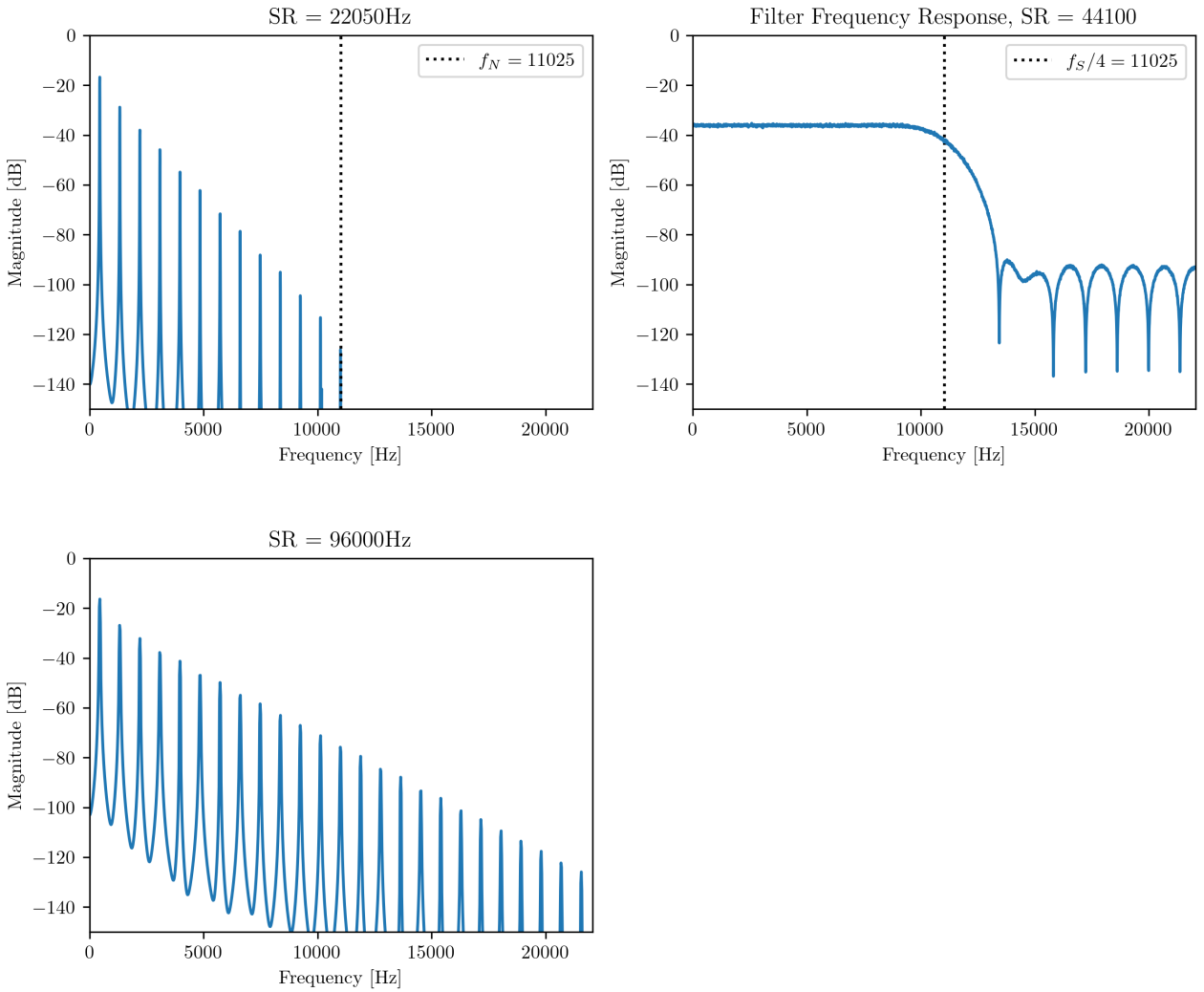


Figure 5.2: The same nonlinear transform, but applied through1

5.2 Oversampling for Alias reduction

5.3 Interpolation

5.4 Decimation

5.5 Resampling block

►argue this model of resampling instead of different in/out rates◄

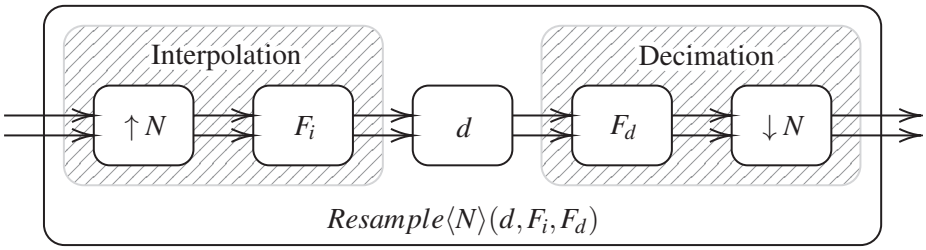


Figure 5.3: The resample block as implemented in the EDA library

5.5.1 C++ Implementation

This chapter contains 352 characters and approximately 65 spaces = 0.10 standard pages

Chapter 6

Evaluation

►Evaluation◄

This chapter contains 20 characters and approximately 2 spaces = 0.01 standard pages

Bibliography

[1] *Faust Programming Language*. URL: https://faust.grame.fr.

[2] *KFR | Fast, Modern C++ DSP Framework*. URL: https://www.kfrlib.com/.

[3] Matteo Frigo and Steven G. Johnson. "The Design and Implementation of FFTW3". In: *Proceedings of the IEEE* 93.2 (2005). Special issue on "Program Generation, Optimization, and Platform Adaptation", pp. 216–231.

[4] Gaël Guennebaud, Benoît Jacob, et al. *Eigen v3*. 2010. URL: http://eigen.tuxfamily.org.

[5] Martín Abadi et al. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. Software available from tensorflow.org. 2015. URL: https://www.tensorflow.org/.

[6] Erik Einhorn et al. "MIRA - middleware for robotic applications". In: *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*. IEEE, 2012.

[7] ISO. *ISO/IEC 14882:2017 Information technology — Programming languages — C++*. Sixth. Geneva, Switzerland: International Organization for Standardization, Dec. 2020, p. 1853. URL: https://www.iso.org/standard/79358.html.

[8] Eric Niebler, Casey Carter, and Christopher Di Bella. *P0896R4: The One Ranges Proposal*. https://wg21.link/p0896r4. Nov. 2018.

[9] Andrew Sutton. *N4674: Working Draft, C++ extensions for Concepts*. https://wg21.link/n4674. June 2017.

[10] Ville Voutilainen. *P0724R0: Merge the Concepts TS Working Draft into the C++20 working draft*. https://wg21.link/p0724r0. June 2017.

[11] Saar Raz. *Clang Concepts*. Talk given at CoreCPP. 2019. URL: https://corecppil.github.io/CoreCpp2019/Presentations/Saar_clang_concepts.pdf.

[12] Eric Niebler and Casey Carter et al. *Range library for C++14/17/20, basis for C++20's std::ranges*. URL: https://github.com/ericniebler/range-v3.

[13] Yann Orlarey, Dominique Fober, and Stéphane Letz. "An Algebraic approach to Block Diagram Constructions". In: (2002). Ed. by GMEM, pp. 151–158. URL: https://hal.archives-ouvertes.fr/hal-02158931.

[14] Yann Orlarey, Dominique Fober, and Stéphane Letz. "Syntactical and Semantical Aspects of Faust". In: *Soft Computing* (2004). URL: https://hal.archives-ouvertes.fr/hal-02159011.

Appendix A

Appendix section