Expressive DSP for Audio Applications in C++20

Tobias Pisani, 201809111

Bachelor Report (15 ECTS) in Computer Science
Department of Computer Science
June 2, 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

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

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

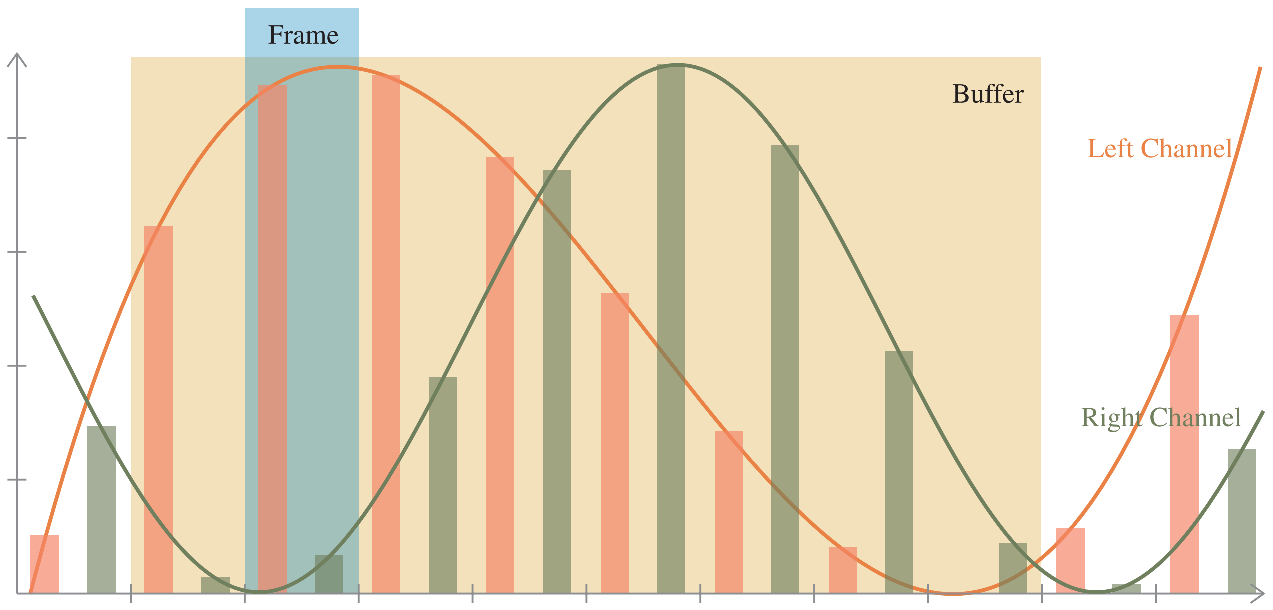


Figure 1.1: A stereo audio signal broken into buffers of 8 frames of two samples each.

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

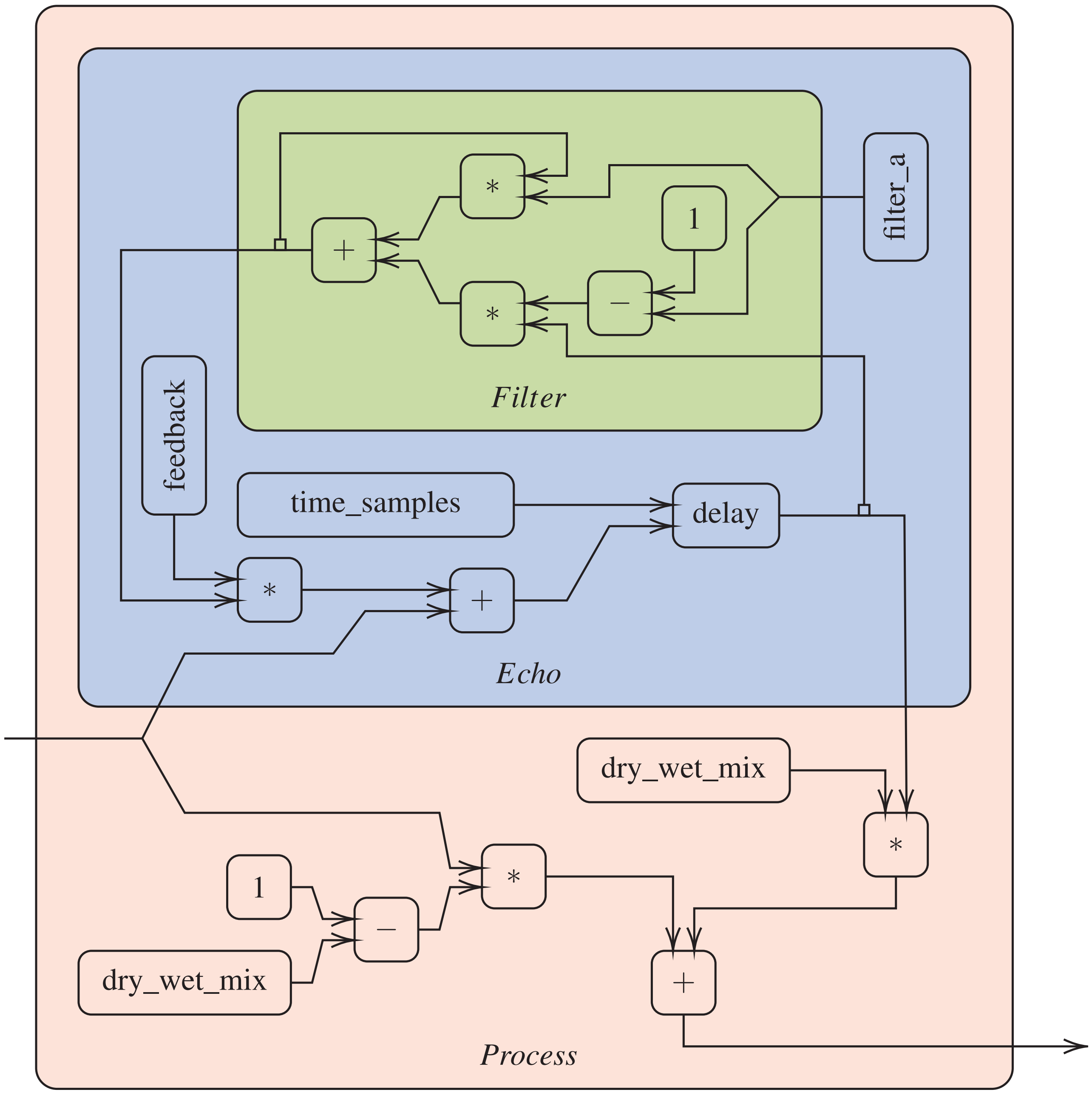


Figure 1.2: Block diagram of the echo effect.

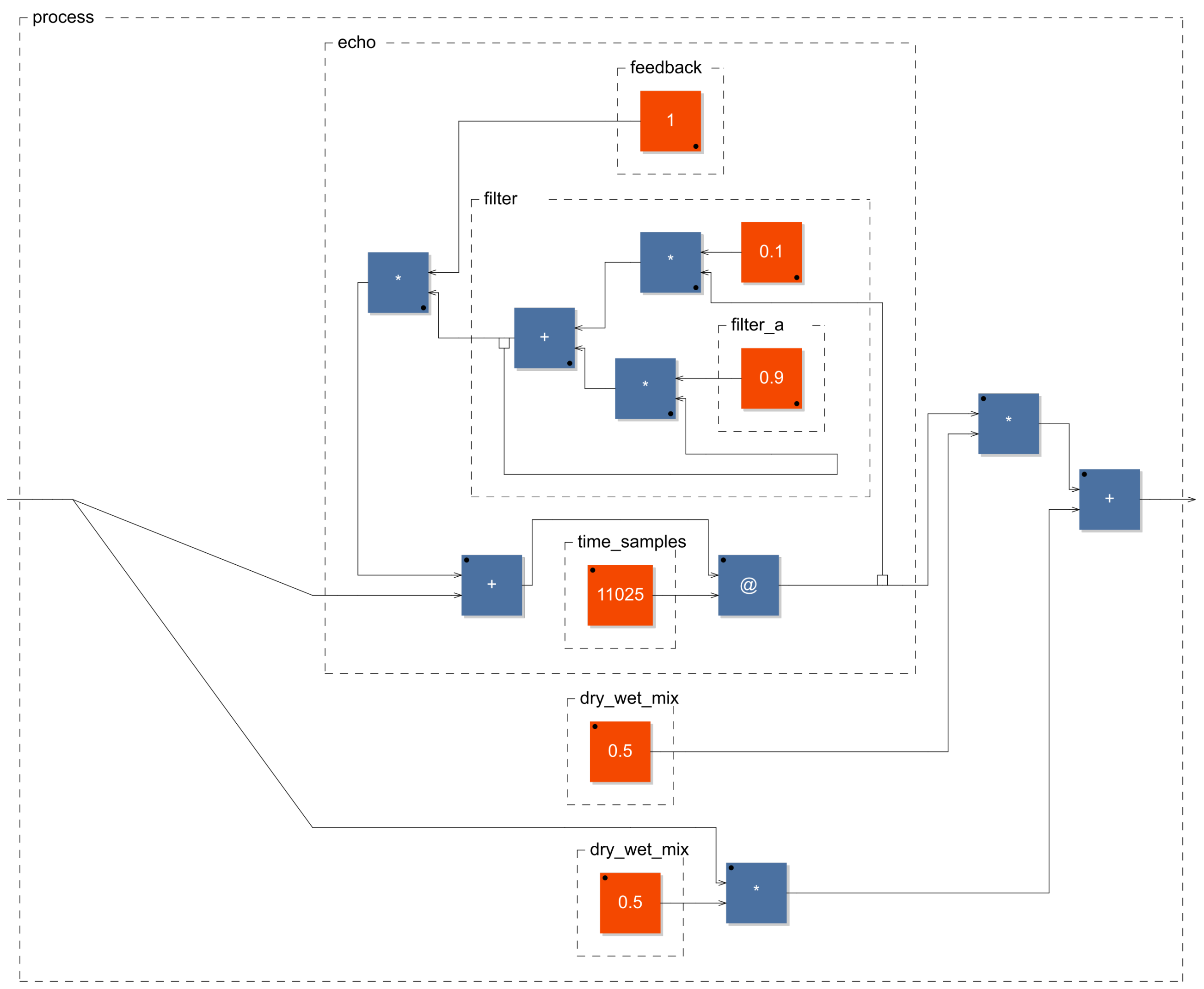


Figure 1.3: Block diagram generated by faust for code in Listing 1.2

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.

```
filter = low_pass(5000, 0.2)
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 `univector`s, 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

[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

view 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

Block Diagrams

►Introduce the concept of block diagrams for describing signal processors◄

3.1 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, which are expanded in [14] with two extra compositional operations, split and merge. The rest of this chapter introduces this algebra of blocks in a language very similar to [14], with minor differences in syntax and semantics noted along the way.

A Signal is a discrete function of time, such that the value of a signal $s \in \mathbb{S}$ at time t is denoted $s(t)$. The full set of all signals is written as $\mathbb{S} = \mathbb{N} \to \mathbb{R}$. Signals are mostly used in signal tuples, denoted as $(s_1, \ldots, s_n) \in \mathbb{S}^n$. To simplify the semantic specifications in the following section, tuples of signal tuples are always flattened, i.e. $\forall s \in \mathbb{S} : s = (s)$, and $\forall a \in \mathbb{S}^n, b \in \mathbb{S}^m : (a,b) = (a_1, \ldots, a_n, b_1, \ldots, b_m)$

In use with AD/DA converters and other audio software, it is convention to let the full range of signals be $[-1;1]$, and mostly this is represented as a 32-bit floating point value. Notice however, that signals may exceed this range, although inputs and outputs of the top-level signal processor should not.

A Signal Processor is a function $S^n \to S^m$, and the object of the model. Signal processors are a transformation from a number of *input* signals to a number of *output* signals, which are evaluated for each time value t in order. The result $p(s)(t)$ of signal processor p may depend on $s(t')$ for all $t' < t$, in other words, signal processors may have *memory*.

The full set of signal processors is notated as $\mathbb{P} = \bigcup_{n,m} \mathbb{S}^n \to \mathbb{S}^m$

A Block is the computational unit used to model signal processors. It is described in terms of the recursive

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

Table 3.1: Basic block diagram compontents and their properties and syntax

language $\mathbb{D}$:

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

Here, $\mathbb{B}$ denotes a domain-specific set of primitive blocks. Some of these will be addressed in a later section.

Faust and the related papers[13, 14] uses single-character operator syntax for the basic operators of $\mathbb{D}$, but since the same syntax cannot be achieved exactly in C++, I will be referring to them by their names as prefix functions to avoid confusion. The later chapter on the C++ implementation will cover the chosen syntax.

To separate the syntax of blocks from the semantics, the function $[\![\ .\ ]\!] : \mathbb{D} \to \mathbb{P}$ is used to map a block diagram d to the corresponding signal processor $[\![d]\!]$.

We also introduce the type-like syntax $d : i \to o$ to mean $[\![d]\!] : \mathbb{S}^i \to \mathbb{S}^o$. This is useful for declaring the type rules, which are covered in the following section.

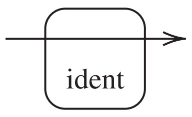
3.1.1 Basic block operations

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 the type rules. Some are slightly simplified here when possible to still get the same expressivity, in those cases it will be noted.

Identity

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

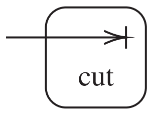
$$
\begin{aligned}
\text{ident} &: 1 \to 1 \\
[\![\text{ident}]\!](s) &= s
\end{aligned}
$$



Cut

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

$$\text{cut} : 1 \to 0$$
$$[\![\text{cut}]\!](s) = ()$$

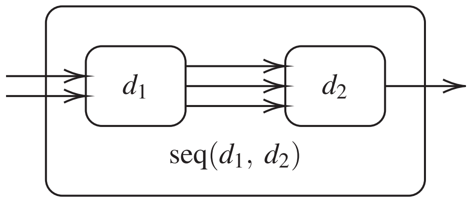


Sequential block composition

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.

$$\frac{d_1 : n \to p \qquad d_2 : p \to m}{\text{seq}(d_1, d_2) : n \to m}$$

$$[\![\text{seq}(d_1, d_2)]\!](s_1, \ldots, s_n) = [\![d_2]\!]\left([\![d_1]\!](s_1, \ldots, s_n)\right)$$

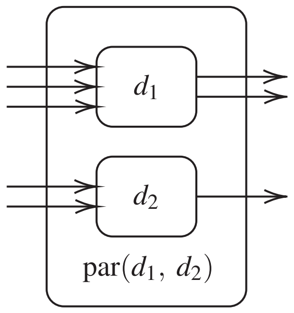


Parallel block composition

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.

$$\frac{d_1 : i_1 \to o_1 \qquad d_2 : i_2 \to o_2}{\text{par}(d_1, d_2) : i_1 + i_2 \to o_1 + o_2}$$

$$[\![\text{par}(d_1, d_2)]\!](s_1, \ldots, s_{i_1}, x_1, \ldots, x_{i_2}) = ([\![d_1]\!](s_1, \ldots, s_{i_1}), [\![d_2]\!](x_1, \ldots, x_{i_2}))$$



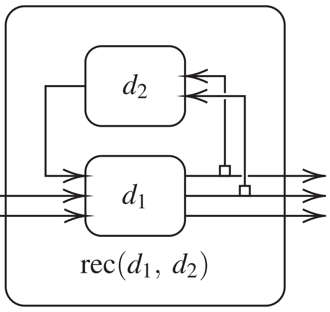
Recursive block composition

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.

$$\frac{d_1 : i_1 \to o_1 \qquad d_2 : i_2 \to o_2 \qquad o_2 \le i_1 \qquad i_2 \le o_1}{\text{rec}(d_1, d_2) : i_1 - o_2 \to o_1}$$

$$\frac{\begin{array}{c}[\![d_1]\!](r_1, \ldots, r_{o_2}, s_1, \ldots, s_n) = (y_1, \ldots, y_{o_1}) \\ [\![d_2]\!](y'_1, \ldots, y'_{i_2}) = (r_1, \ldots, r_{o_2})\end{array}}{[\![\text{rec}(d_1, d_2)]\!](s_1, \ldots, s_n) = (y_1, \ldots, y_{o_2})}$$



Where y' is the signal y delayed by one sample, i.e

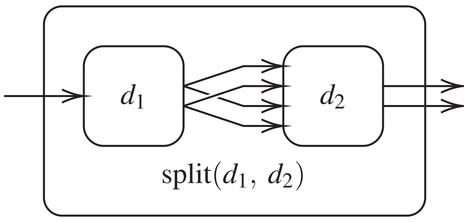
$$\forall y \in \mathbb{S}, t \in \mathbb{N}^+ : y'(0) = 0, y'(t) = y(t-1)$$

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 **ins**(d_2) must be an exact multiple of **outs**(d_1).

$$\frac{d_1 : i_1 \to o_1 \qquad d_2 : o_1 * k \to o_2 \qquad k \in \mathbb{N}}{\text{split}(d_1, d_2) : i_1 \to o_2}$$

$$\frac{\begin{array}{c}[\![d_1]\!](s_1, \ldots, s_{i_1}) = (x_1, \ldots, x_{o_1}) \\ \forall j \in \{1, \ldots, i_2\} y_j = x_{j \bmod o_1}\end{array}}{[\![\text{split}(d_1, d_2)]\!](s_1, \ldots, s_{i_1}) = [\![d_2]\!](y_1, \ldots, y_{i_2})}$$



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

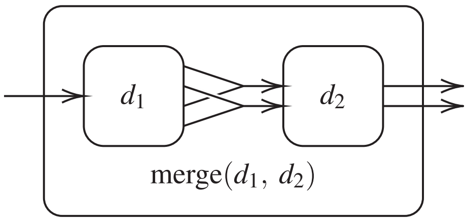
Merge block composition

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 **outs**$(d_1) =$ **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 $\text{seq}(d_1, d_2)$ when $k = 1$.

$$\frac{d_1 : i_1 \to i_2 * k \qquad d_2 : i_2 \to o_2 \qquad k \in \mathbb{N}}{\text{merge}(d_1, d_2) : i_1 \to o_2}$$

$$\frac{\begin{array}{c}[\![d_1]\!](s_1, \ldots, s_{i_1}) = (x_1, \ldots, x_{o_1}) \\ \forall j \in \{1, \ldots, i_2\} y_j = \sum_{l=0}^{k-1} x_{j+k*i_2}\end{array}}{[\![\text{merge}(d_1, d_2)]\!](s_1, \ldots, s_{i_1}) = [\![d_2]\!](y_1, \ldots, y_{i_2})}$$



3.1.2 Domain Specific Blocks

Arithmetic

Memory

Delay

Ref

Ref

This chapter contains 4353 characters and approximately 956 spaces = 1.26 standard pages

Chapter 4

The C++ Library

One important design decision in this library, is to split the declaration of a block diagram from the evaluation of the corresponding signal processor. While this makes implementing new block types slightly more verbose, it has a couple of advantages. Most importantly, a block diagram is a static structure that can be declared once, even constructed at compile time in many cases, and then multiple instances of the signal processor can be constructed at runtime as needed. Secondly, having the block diagram available as a declarative structure makes other evaluators than the signal processor possible, such as one that builds a visualization of the block diagram.

4.1 Blocks

BlockBase

As described in the previous chapter, a block in $\mathbb{D}$ has a number of inputs and a number of outputs. In EDA, these are modelled by extending the `BlockBase` CRTP[1] base class, meaning a base class template that is always passed the derived class as its first template parameter:

```
template<typename Derived, std::size_t InChannels, std::size_t OutChannels>
struct BlockBase {
  static constexpr std::size_t in_channels = InChannels;
  static constexpr std::size_t out_channels = OutChannels;

  constexpr auto operator()(auto&&... inputs) const noexcept
    requires(sizeof...(inputs) <= InChannels);
};
```

This base class provides the `in_channels` and `out_channels` constants, along with the call operator used for partial application (see section 4.5).

AnyBlock, AnyBlockRef and ABlock

Three basic concepts are introduced as well to check whether a type `T` is a block, a block with or without reference/const/volatile qualifiers, or a block with a specific signature. **`AnyBlock`** also requires a type to model `std::copyable`[2], to make sure that blocks can be copied around.

```
template<typename T>
concept AnyBlock = std::is_base_of_v<BlockBase<T, T::in_channels, T::out_channels>, T> && std::copyable<T>;
```

[1] Curiously Recurring Template Pattern, see https://en.wikipedia.org/wiki/Curiously_recurring_template_pattern
[2] See https://en.cppreference.com/w/cpp/concepts/copyable

```
template<typename T>
concept AnyBlockRef = AnyBlock<std::remove_cvref_t<T>>;

template<typename T, std::size_t I, std::size_t O>
concept ABlock = AnyBlock<T> &&(T::in_channels == I) && (T::out_channels == O);
```

The `ABlock` concept is especially useful when used as a placeholder type for declaring block diagrams:

```
ABlock<1, 2> auto d = (_ << (_ + _, _ * _));
```

Here the `ABlock<1, 2>` prefix becomes a checked annotation that `d` is a block with signature $1 \to 2$.

`ins<T>` and `outs<T>`

To access the number of input/output channels, the following shorthand variable templates are introduced:

```
template<AnyBlockRef T>
constexpr auto ins = std::remove_cvref_t<T>::in_channels;

template<AnyBlockRef T>
constexpr auto outs = std::remove_cvref_t<T>::out_channels;
```

They allow `ins<T> == 2` when `T` is a cv/ref-qualified block, i.e. it models `AnyBlockRef`.

4.1.1 Identity block

As the simplest example of a block type declaration, I take a look at the identity block. For convenience, it has here been extended with a template parameter `N` to allow for identity blocks of different numbers of channels. `ident<N>` is equal to the parallel composition of `N` identity blocks. As a side note, the cut block has been extended in a similar manner.

```
template<std::size_t N = 1>
struct Ident : BlockBase<Ident<N>, N, N> {};

template<std::size_t N = 1>
constexpr Ident<N> ident;
```

Here, the declaration consists of two parts, the `Ident` type itself, which inherits from `BlockBase`, and the constant `ident` variable template, which serves the function of the constructor.

4.2 Block Compositions

CompositionBase

Block compositions are implemented as class templates that derive from `CompositionBase`, which iself derives from `BlockBase`. `CompositionBase` keeps a tuple of the operand blocks, which can then be accessed by the deriving class. Likr `BlockBase`, it is a CRTP-style base class template, so its first template parameter is the class that is deriving from it.

```
template<typename D, std::size_t In, std::size_t Out, AnyBlock... Operands>
struct CompositionBase : BlockBase<D, In, Out> {
  using operands_t = std::tuple<Operands...>;
  constexpr CompositionBase(Operands... ops) noexcept : operands(std::move(ops)...) {}
  operands_t operands;
};
```

AComposition and ACompositionRef

Once again, a couple of acompanying concepts are added to check that a type `T` is a composition or reference to one:

```
template<typename T>
concept AComposition = AnyBlock<T> && requires (T& t) {
  typename T::operands_t;
  { t.operands } -> util::decays_to<typename T::operands_t>;
};

template<typename T>
concept ACompositionRef = AComposition<std::remove_cvref_t<T>>;
```

`operands_t`

As a shorthand for accessing the type of the operands of a cv-ref qualified composition, the `operands_t<T>` alias template is added:

```
template<ACompositionRef T>
using operands_t = typename std::remove_cvref_t<T>::operands_t;
```

Sequential

Recall the type rule for sequential composition from section 3.1.1:

$$\frac{d_1 : n \rightarrow p \qquad d_2 : p \rightarrow m}{\text{seq}(d_1, d_2) : n \rightarrow m}$$

Using type constraints, this can be encoded as the following block declaration:

```
template<AnyBlock Lhs, AnyBlock Rhs>
requires(outs<Lhs> == ins<Rhs>)
struct Sequential : CompositionBase<Sequential<Lhs, Rhs>, ins<Lhs>, outs<Rhs>, Lhs, Rhs> {};
```

First of all, `Lhs` and `Rhs` must both model the concept `AnyBlock`, which simply ensures that the types given are in fact blocks. Secondly, a *requires-clause* is added to the struct declaration to assert that the outputs of `Lhs` is equal to the inputs of `Rhs`. If these requirements are unsatisfied, the compiler emits useful error messages that are fairly easy to trace (see section 4.10). The second and third template parameters to `CompositionBase` specify the number of input and output channels, so by passing `ins<Lhs>` and `outs<Rhs>` respectively, `Sequential<Lhs, Rhs>` has the signature $n \rightarrow m$ as specified in the type rule. Finally, `Lhs` and `Rhs` are passed as the `Operands...` argument to `CompositionBase`, meaning those blocks will be stored in the `std::tuple<Lhs, Rhs> operands` member variable

For ease of construction, the free function `seq(a, b)` is written as follows:

```
template<AnyBlockRef Lhs, AnyBlockRef Rhs>
constexpr auto seq(Lhs&& lhs, Rhs&& rhs) noexcept
{
  return Sequential<std::remove_cvref_t<Lhs>, std::remove_cvref_t<Rhs>>{
    .lhs = std::forward<Lhs>(lhs),
    .rhs = std::forward<Rhs>(rhs)
  };
}
```

Remaining Binary compositions

When declaring a block diagram (as opposed to when evaluating its signal processor), the only differences between the various compositional operators are the requirements for the operands and the calculation of the signature. For example, parallel composition is declared as follows:

```cpp
template<AnyBlock Lhs, AnyBlock Rhs>
struct Parallel : CompositionBase<Parallel<Lhs, Rhs>, ins<Lhs> + ins<Rhs>, outs<Lhs> + outs<Rhs>, Lhs, Rhs> {};
```

Here the signature is calculated differently from sequential composition, and there are no requirements on `Lhs` and `Rhs`. Recursive, split, and merge composition are all implemented similarly by simply translating the requirements and signature to C++ type requirements. **►Reference the code in the appendix◄**

4.3 Literals and References

By now we know how to declare primitive and compositional blocks, so the following code should seem natural.

```cpp
struct Literal : BlockBase<Literal, 0, 1> {
  float value;
};

constexpr Literal literal(float f) noexcept {
  return Literal{.value = f};
}

struct Ref : BlockBase<Ref, 0, 1> {
  float* ptr = nullptr;
};

constexpr Ref ref(float& f) noexcept
{
  return Ref{.ptr = &f};
}
```

The `Literal` and `Ref` blocks each have a signature of $0 \to 1$, and can be used to introduce scalar values into the block diagram. `Literal` is used for constants, and `Ref` is used for values that change over time, i.e. non-signal values used to control parameters of the signal processor. This is a problem that faust solves using functions that model UI elements, such as `vslider(name, ...)`[14]

4.4 Operator overloads and shorthand syntax

With the block types and construction functions, block diagrams can be declared by composing the constructors:

```cpp
ABlock<2, 1> auto d = seq(seq(ident<2>, par(ident<1>, ident<1>)), par(cut<1>, ident<1>));
```

Ignoring for now that $[\![d]\!]$ does nothing other than discarding the first of two signals (i.e. $[\![d]\!] = [\![\mathrm{par}(\mathrm{cut}, \mathrm{ident})]\!]$), the issue at hand is the verbosity of the declaration of `d`. Since the goal of the project is to

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

```cpp
constexpr decltype(auto) as_block(AnyBlockRef auto&& input) noexcept
{
  return std::forward<decltype(input)>(input);
}

constexpr Literal as_block(float f) noexcept
{
  return literal(f);
}

template<typename T>
using as_block_t = std::remove_cvref_t<decltype(as_block(std::declval<T>()))>;
```

```cpp
// LITERAL ////////////////////////////////////////////

constexpr Literal operator"" _eda(long double f) noexcept
{
  return as_block(static_cast<float>(f));
}
constexpr Literal operator"" _eda(unsigned long long f) noexcept
{
  return as_block(static_cast<float>(f));
}

// IDENT //////////////////////////////////////////////

constexpr Ident<1> _ = ident<1>;
constexpr Cut<1> cut = cut<1>;

// PARALLEL ///////////////////////////////////////////

template<typename Lhs, typename Rhs>
constexpr auto operator,(Lhs&& lhs, Rhs&& rhs) noexcept //
  requires(AnyBlockRef<Lhs> || AnyBlockRef<Rhs>)
{
  return parallel(as_block(FWD(lhs)), as_block(FWD(rhs)));
}

// SEQUENTIAL /////////////////////////////////////////

template<typename Lhs, typename Rhs>
constexpr auto operator|(Lhs&& lhs, Rhs&& rhs) noexcept //
  requires(AnyBlockRef<Lhs> || AnyBlockRef<Rhs>)
{
  return sequential(as_block(FWD(lhs)), as_block(FWD(rhs)));
}

// Split //////////////////////////////////////////////

template<typename Lhs, typename Rhs>
constexpr auto operator<<(Lhs&& lhs, Rhs&& rhs) noexcept //
  requires(AnyBlockRef<Lhs> || AnyBlockRef<Rhs>)
{
  return split(as_block(FWD(lhs)), as_block(FWD(rhs)));
}

// MERGE //////////////////////////////////////////////

template<typename Lhs, typename Rhs>
constexpr auto operator>>(Lhs&& lhs, Rhs&& rhs) noexcept //
  requires(AnyBlockRef<Lhs> || AnyBlockRef<Rhs>)
{
  return merge(as_block(FWD(lhs)), as_block(FWD(rhs)));
}

// ARITHMETIC /////////////////////////////////////////

template<typename Lhs, typename Rhs>
constexpr auto operator+(Lhs&& lhs, Rhs&& rhs) noexcept requires(AnyBlockRef<Lhs> || AnyBlockRef<Rhs>)
{
  return plus(lhs, rhs);
}

template<typename Lhs, typename Rhs>
```

```
constexpr auto operator-(Lhs&& lhs, Rhs&& rhs) noexcept requires(AnyBlockRef<Lhs> || AnyBlockRef<Rhs>)
{
  return minus(lhs, rhs);
}

template<typename Lhs, typename Rhs>
constexpr auto operator*(Lhs&& lhs, Rhs&& rhs) noexcept requires(AnyBlockRef<Lhs> || AnyBlockRef<Rhs>)
{
  return times(lhs, rhs);
}

template<typename Lhs, typename Rhs>
constexpr auto operator/(Lhs&& lhs, Rhs&& rhs) noexcept requires(AnyBlockRef<Lhs> || AnyBlockRef<Rhs>)
{
  return divide(lhs, rhs);
}

// RECURSIVE /////////////////////////////////////////

template<typename Lhs, typename Rhs>
constexpr auto operator%(Lhs&& lhs, Rhs&& rhs) noexcept requires(AnyBlockRef<Lhs> || AnyBlockRef<Rhs>)
{
  return recursive(as_block(FWD(lhs)), as_block(FWD(rhs)));
}
```

4.5 Partial function application

```
template<AnyBlock Block, AnyBlock... Inputs>
requires((outs<Inputs> <= ins<Block>) &&...) //
  struct Partial
  : BlockBase<Partial<Block, Inputs...>, ins<Block> + (ins<Inputs> + ...) - (outs<Inputs> + ...), outs<Block>> {
  constexpr Partial(Block b, Inputs... input) noexcept : block(b), inputs(input...) {}

  Block block;
  std::tuple<Inputs...> inputs;
};

template<typename D, std::size_t I, std::size_t O>
constexpr auto BlockBase<D, I, O>::operator()(auto&&... inputs) const noexcept //
  requires(sizeof...(inputs) <= I)
{
  return Partial<D, as_block_t<decltype(inputs)>...>(static_cast<const D&>(*this), as_block(FWD(inputs))...);
}
```

4.6 Repeat

```
template<std::size_t N>
constexpr auto repeat(AnyBlock auto const& block, auto&& composition) requires requires
{
  composition(block, block);
}
{
  if constexpr (N == 0) {
    return ident<0>;
  } else if constexpr (N == 1) {
    return block;
  } else {
    return composition(block, repeat<N - 1>(block, composition));
  }
}

template<std::size_t N>
constexpr auto repeat_seq(AnyBlock auto const& block)
{
  return repeat<N>(block, [](auto&& a, auto&& b) { return seq(a, b); });
}

template<std::size_t N>
constexpr auto repeat_par(AnyBlock auto const& block)
{
  return repeat<N>(block, [](auto&& a, auto&& b) { return par(a, b); });
}
```

4.7 Evaluating the AST

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

4.7.1 `Frame`

4.8 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◄

4.9 Block implementations

```
template<std::size_t In, std::size_t Out, util::Callable<Frame<Out>(Frame<In>)> F>
requires std::copyable<F>
struct FunBlock : BlockBase<FunBlock<In, Out, F>, In, Out> {
  F func_;
};

template<std::size_t In, std::size_t Out>
constexpr auto fun(util::Callable<Frame<Out>(Frame<In>)> auto&& f)
{
  return FunBlock<In, Out, std::decay_t<decltype(f)>>{.func_ = f};
}

constexpr auto sin = fun<1, 1>(&::sinf);
constexpr auto cos = fun<1, 1>(&::cosf);
constexpr auto tan = fun<1, 1>(&::tanf);
constexpr auto tanh = fun<1, 1>(&::tanhf);
constexpr auto mod = fun<2, 1>([](auto in) { return std::fmod(in[0], in[1]); });
```

```
template<std::size_t In, std::size_t Out, typename Func, typename... States>
requires(util::Callable<Func, Frame<Out>(Frame<In>, States&...)>&& std::copyable<Func>) &&
  (std::copyable<States> && ...) //
  struct StatefulFunc : BlockBase<StatefulFunc<In, Out, Func, States...>, In, Out> {
  Func func;
  std::tuple<States...> states;
};

template<std::size_t In, std::size_t Out, typename... States>
```

```
constexpr auto fun(util::Callable<Frame<Out>(Frame<In>, std::remove_cvref_t<States>&...)> auto&& f,
                   States&&... states)
{
  return StatefulFunc<In, Out, std::decay_t<decltype(f)>, std::remove_cvref_t<States>...>{.func = f,
                                                                                    .states = {FWD(states)...}};
}
```

```
struct Plus : BlockBase<Plus, 2, 1> {};
constexpr Plus plus;
struct Minus : BlockBase<Minus, 2, 1> {};
constexpr Minus minus;
struct Times : BlockBase<Times, 2, 1> {};
constexpr Times times;
struct Divide : BlockBase<Divide, 2, 1> {};
constexpr Divide divide;
```

4.10 Compilation Errors

This chapter contains 10375 characters and approximately 2113 spaces = 2.97 standard pages

Chapter 5

Multirate DSP Algorithms

As an example of the advantages of working directly in C++, this chapter explores resampling, and working with a signal path that uses multiple sample rates. I Will cover why this can be very important in certain classes of DSP algorithms, how the transformations between sample rates is done, and how I have implemented it in the EDA library.

5.1 Aliasing

According to the Nyquist-Shannon theorem[15], a discrete-time sampled signal can only represent frequencies below half of the sample rate, called the Nyquist frequency f_N or the Nyquist limit. Intuitively, this is because no change in the waveform can be faster than the time between two samples.

The frequencies above the Nyquist limit don't just disappear from the sampled signal though, but will instead be mirrored back and fourth between the Nyquist frequency and $f = 0$. This behaviour is called aliasing, and is undesirable in most usecases, since it distorts the signal with non-harmonic frequencies.

When initially sampling an analogue signal, the main way to avoid aliasing is simply to use an analogue low-pass filter to remove any frequencies above the Nyquist limit before the signal is sampled to discrete time[16], which means there are no frequencies to be aliased. However, aliasing can also be an issue in some DSP operations that introduce new frequencies above the original signal, such as nonlinear waveshaping functions, i.e. an operation that applies a function $\omega(x)$ to the original signal x, where ω is non-linear.

5.1.1 Waveshaping Distortion

A simple and common example of nonlinear waveshaping is distortion using the hyperbolic-tangent trigonometric function $\omega(x) = \tanh(x)$ (or, in practice, a polynomial approximation thereof). By adjusting the gain of the input, i.e $\omega(x) = \tanh(g \cdot x)$, the effect can span from a soft saturation to hard clipping. When looking at the frequency spectrum, this introduces frequencies above the original signal, and in the case where the input is a simple sine wave (i.e. a single peak frequency), this waveshaping operation introduces odd harmonics, that is it introduces frequencies at odd multiples of the frequencies in the original signal. Adjusting the gain will change the magnitude (volume) of these frequencies.

In the following introduction to aliasing and oversampling, I will refer heavilly to the graphs in Figure 5.1. These are visualizations of various runs of this waveshaping distortion, all with an input signal of a sine wave at 1800 Hz, and $g = 5$. These parameters were chosen simply because of the visualizations they result in, but do reflect real-world usecases. (a) is generated at a very high sample rate relative to the visible frequency range, and

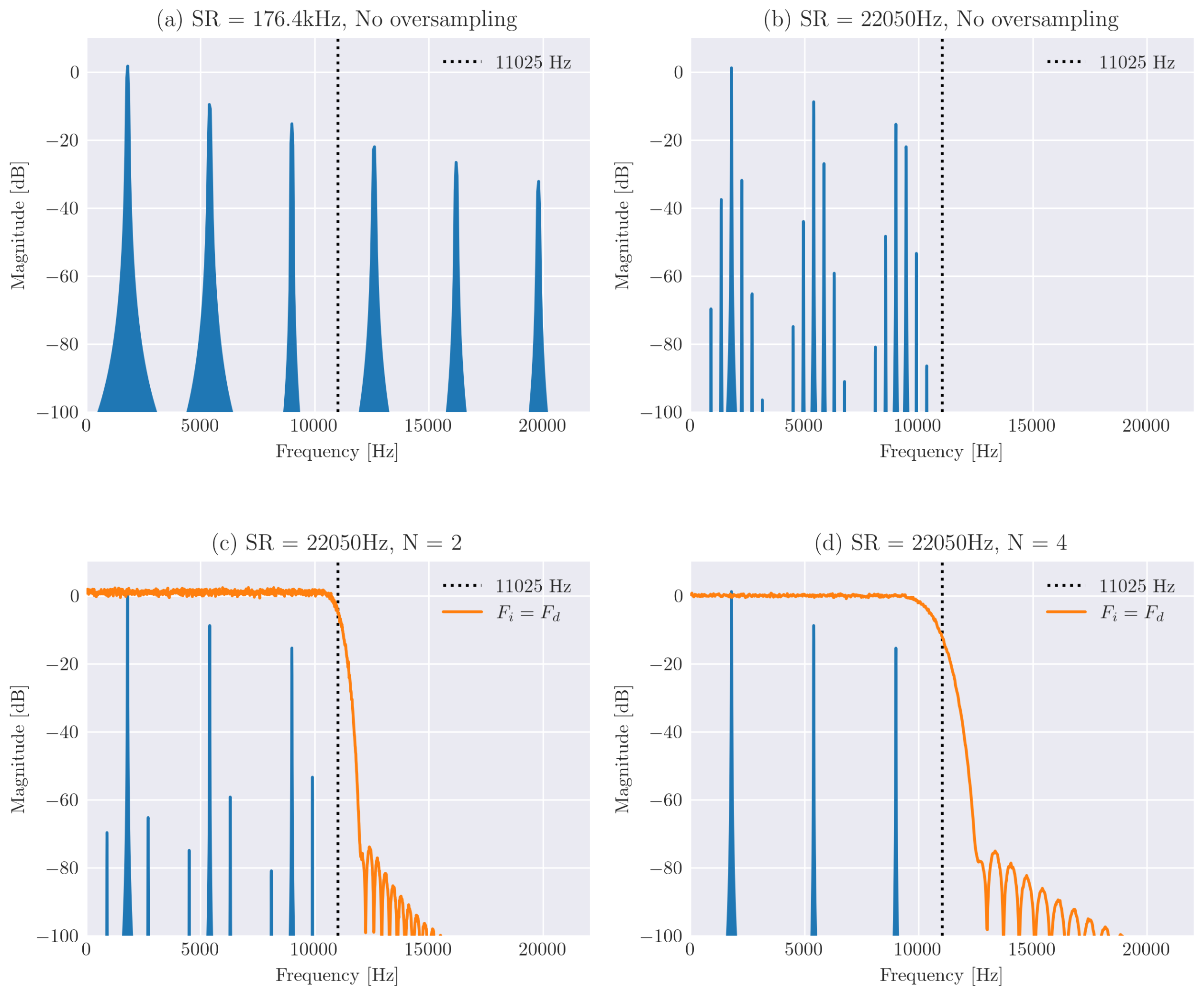


Figure 5.1: A 1800 Hz sine wave passed through the nonlinear waveshaping function $\omega(x) = \tanh(5x)$ at different sample rates and resampling factors N. Given the high samplerate, (a) can be seen as representing the ideal signal over this frequency range. (c) and (d) additionally show the frequency response of the interpolation/decemation filters used with oversampling. For better visualization, the logarithmic decibel scale is used on the vertical axis: $z\,\mathrm{dB} = 20 \cdot \log_{10}(z)$.

can be regarded as representing the ideal signal. Here, the fundamental frequency at 1800 Hz is visible as the first peak, and the harmonics can be seen at even intervals above the fundamental.

Figure 5.1 (b) shows the same signal operation applied at 22050Hz, where the aliasing becomes evident. Notice that the added frequencies are a clear mirroring back and forth between the Nyquist frequency and 0 Hz. It should be very clear that these frequencies are unwanted, and it is especially worth noting that the aliasing has added frequencies below the fundamental, which will be particularly noticable. The signal we actually want when applying ω at 22050Hz, is the left half of (a), i.e. all the frequencies of the ideal signal that are representable at that sample rate, i.e. the ones that are below the Nyquist limit of 11025Hz.

5.2 Oversampling for Alias reduction

The most common approach to reduce the effects of aliasing in DSP algorithms, is oversampling, which is the operation of upsampling by a factor of N, performing the required operations on the signal, and then downsampling by N again, to return to the original sample rate. The basic idea is, that by raising the sample rate, you raise the Nyquist frequency, which means the point at which frequencies will be mirrored is raised. This results in a smaller part of the signal being mirrored, and the first area that the loudest frequencies will be mirrored onto, is above the original Nyquist limit, and can be removed when downsampling to the original sample rate.

The effects of this process can be seen in Figure 5.1, where (c) and (d) show the operation applied with upsampling of $N = 2$ and $N = 4$. Ignoring the frequency response of the filter (shown in orange), these plots clearly show less aliasing, with no visible aliasing at all for $N = 4$. With $N = 2$ it can be seen that the mirroring has happened at $f = 22\,050\,\text{Hz}$ instead, but then the frequencies above $11\,025\,\text{Hz}$ have been filtered out. This clearly shows how oversampling reduces the effects of aliasing, and for this operation with this data, it looks like $N = 4$ is enough. However, $N = 8$ is often chosen in the more general case [17]

There are other and more efficient ways to avoid aliasing[18], but they mostly depend on specific knowledge of the nonlinear operation that is being used, and oversampling is widely recognized as the standard method for alias reduction[17, 19].

5.2.1 Interpolation

I will briefly introduce the most common method of increasing the sample rate of a sampled signal, in which the signal is first zero-stuffed and then interpolated using a filter. Like with alias reduction, this is an area where many variants and other methods have been developed[20, 21], and this section vastly simplifies the subject. However, it gives a general understanding of the problems involved, and how they are most commonly solved.

It is also worth noting that the terms *upsampling* and *interpolation* are often conflated. This can lead to some confusion, however when not talking about the implementation details of either, both terms usually refer to the joint operation of upsampling and interpolation.

The first step is to simply increase the sample rate of the signal. This is done by *zero-stuffing* the signal, i.e. inserting $N - 1$ zeros between each sample. In Figure 5.2 the result of this operation can be seen in the time and frequency domains. In the frequency domain, two things have happened: The gain of the signal has been scaled by $\frac{1}{N}$, and the signal has been mirrored around the old Nyquist frequency f_{N1}. The gain is simply restored by multiplying each sample by N, and for the new signal above f_{N1}, we can use a low pass filter to remove them.

5.2.2 FIR Filters

There are a lot of methods to designing and implementing interpolation filters for the best and most efficient results, but most of them are based on Finite Impulse Response (FIR) filters. A FIR filter of order N is a simple

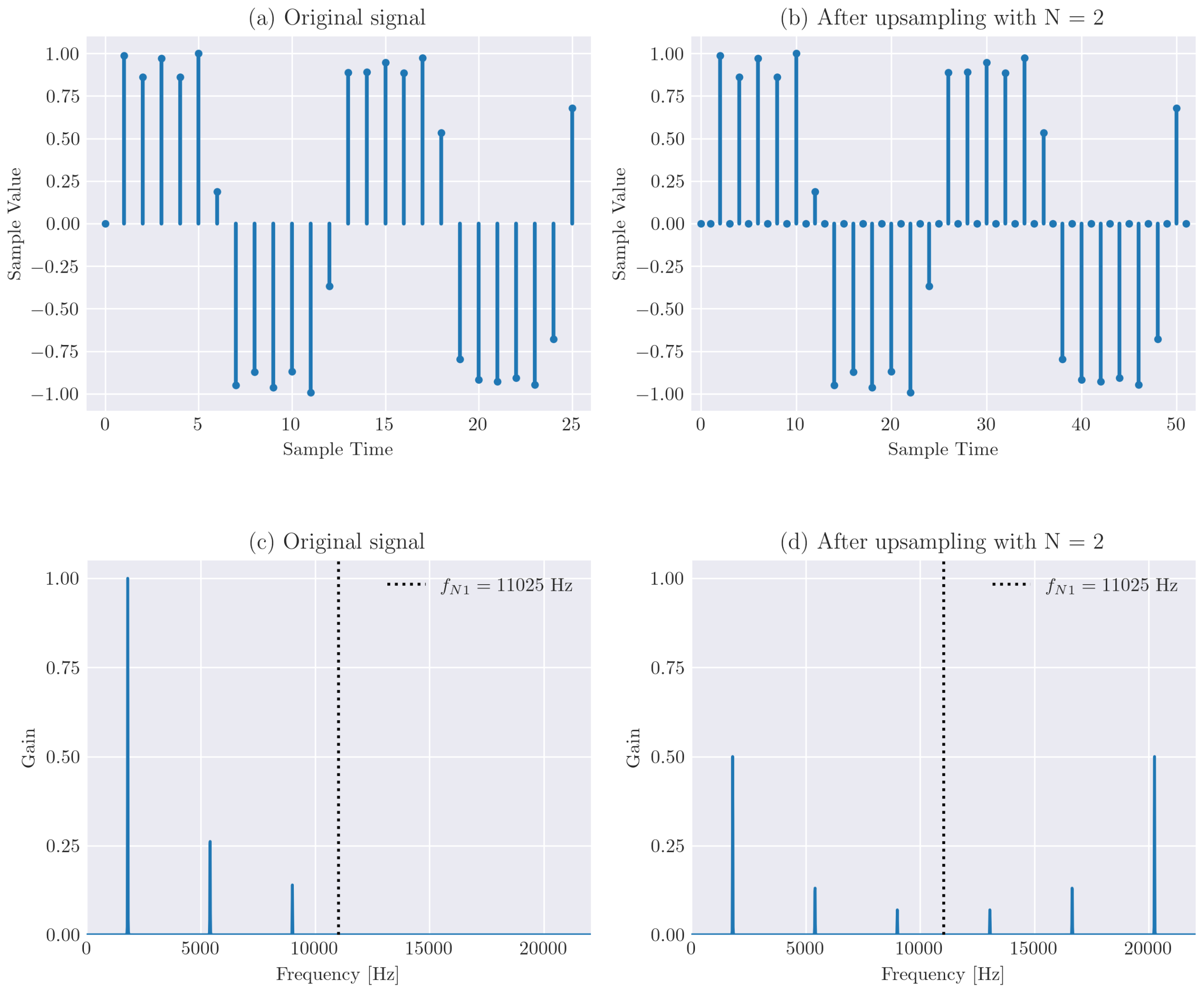


Figure 5.2: The effects of zero-stuffing on the waveform and frequency spectrum. The input signal is the same saturated 1800Hz sine wave as used in Figure 5.1, and the sample rates are 22050Hz and 44100Hz before and after upsampling respectively. To clearly show the reduction in gain on upsampling, linear gain is used on the vertical axis of the frequency graphs.

convolution operation where each output is a weighted sum of the $N+1$ most recent inputs. Designing FIR filters, i.e. picking the right coefficients, is a heavilly researched topic[17, 20, 22], and not something I will go into in this project, however, the filters used for testing and for Figure 5.1, are FIR filters of order $N = 128$, designed using the method described in [23]. These filters have a fairly high order to ensure a very steep cutoff, but in real use there is a tradeoff between the efficiency of a lower filter order, and the better results of a harder cutoff point.

5.2.3 Decimation

When downsampling, we first need to remove any frequencies above the Nyquist limit of the resulting sample rate, since those will otherwise be aliased, which was the original reason for oversampling. This process is called *decimation*, and consists of running a lowpass filter and then selecting every N^{th} sample for output. Thus, it closely resembles interpolation, and in fact the same filter can be used for both, even though different ones are often used for the best results, partly because decimation allows for some special optimizations, as only every N^{th} sample is actually required, which means some computations can be skipped in the FIR filter. These topics are covered in many of the referenced sources of this chapter, such as [17, 20, 22].

5.3 Multirate in the Algebra of Blocks

Now that we know why resampling is important, and the basics of how it is implemented, I will look at how to integrate it in the block algebra introduced in chapter 3.

5.3.1 Approach I

In the most general model, multirate blocks are introduced by adding an extra parameter R to blocks, which is the ratio of the sample rates f_{out}/f_{in}.

►Describe this approach◄

5.3.2 Approach II

Instead, a simpler approach is chosen, where all blocks have the same input/output rate, but a separate resample $\langle N \rangle\,(d)$ block is introduced, which can be used to perform over/undersampling for the block d.

$$\frac{d : i \to o \qquad F_i : i \to i \qquad F_d : o \to o \qquad N \in \mathbb{N}^+}{\text{resample}\langle N\rangle(d, F_i, F_d) : i \to o}$$

$$\frac{\begin{array}{l} u_j(t) = \begin{cases} s_j\left(\frac{t}{N}\right) & \exists k \in \mathbb{N} : t = N \cdot k \\ 0 & \text{otherwise} \end{cases} \\ [\![\text{seq}(\text{seq}(F_i, d), F_d)]\!](u_1, \ldots, u_i) = (v_1, \ldots, v_o) \\ y_j(t) = v_j(t \cdot N) \end{array}}{[\![\text{resample}\langle N\rangle(d, F_i, F_d)]\!](s_1, \ldots, s_i) = (y_1, \ldots, y_o)}$$

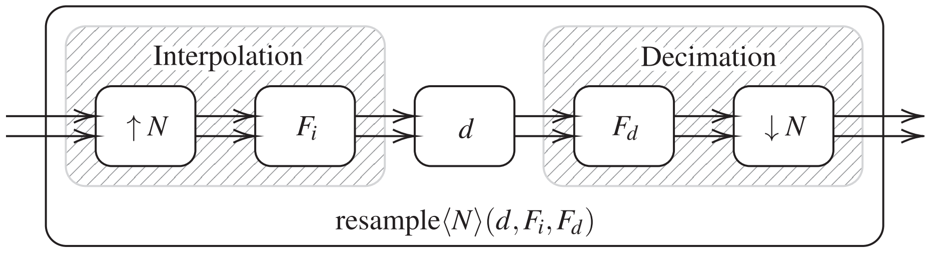


Figure 5.3: The resample block as implemented in the EDA library

5.4 Multirate in the EDA Library

This chapter contains 7749 characters and approximately 1573 spaces = 2.22 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.

[15] C.E. Shannon. "Communication in the Presence of Noise". In: *Proceedings of the IRE* 37.1 (1949), pp. 10–21. DOI: 10.1109/JRPROC.1949.232969.

[16] Bonnie C. Baker. "Anti-Aliasing, Analog Filters for Data Acquisition Systems". In: (1999).

[17] julen kahles julen, fabián esqueda fabián, and vesa välimäki vesa. "oversampling for nonlinear waveshaping: choosing the right filters". In: *journal of the audio engineering society* 67.6 (June 2019), pp. 440–449. DOI: https://doi.org/10.17743/jaes.2019.0012.

[18] Stefan Bilbao et al. "Antiderivative Antialiasing for Memoryless Nonlinearities". In: *IEEE Signal Processing Letters* 24.7 (2017), pp. 1049–1053. DOI: 10.1109/LSP.2017.2675541.

[19] Brecht De Man and Joshua D. Reiss. "Adaptive Control of Amplitude Distortion Effects". In: (). URL: https://www.eecs.qmul.ac.uk/~josh/documents/2014/DeMan%20Reiss%20-%20AES53.pdf.

[20] Francisco Rubén Castillo Soria et al. “A simplification to the fast FIR-FFT filtering technique in the DSP interpolation process for band-limited signals”. en. In: *Revista Facultad de Ingeniería Universidad de Antioquia* (Sept. 2013), pp. 09–19. ISSN: 0120-6230. URL: http://www.scielo.org.co/scielo.php?script=sci_arttext&pid=S0120-62302013000300002&nrm=iso.

[21] F. R. Castillo-Soria et al. “Comparative Analysis of DSP Interpolation Process for Diverse Insertion Techniques and FIR Filtering”. In: (). URL: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.1054.8052&rep=rep1&type=pdf.

[22] Kurbiel Thomas, Heinz Göckler, and Daniel Alfsmann. “A Novel Approach to the Design of Oversampling Low-Delay Complex-Modulated Filter Bank Pairs”. In: *EURASIP Journal on Advances in Signal Processing* 2009 (Dec. 2009). DOI: 10.1155/2009/692861.

[23] Tom Roelandts. *How to Create a Simple Low-Pass Filter*. 2014. URL: https://web.archive.org/web/20210522153834/https://tomroelandts.com/articles/how-to-create-a-simple-low-pass-filter (visited on 05/22/2021).

Appendix A

Appendix section