Expressive DSP for Audio Applications in C++

Tobias Pisani, 201809111

Bachelor Report (15 ECTS) in Computer Science
Department of Computer Science
June 8, 2021
Advisor: Aslan Askarov

AARHUS UNIVERSITY
DEPARTMENT OF COMPUTER SCIENCE

Abstract

This thesis explores using an embedded domain specific language to write audio signal processing programs modelled as block diagrams. Building on existing research on an algebra to describe block diagrams, it provides the basis of a C++ library that implements such an EDSL. It goes on to explore real-world usage of said library by implementing an audio plugin, as well as expanding the library with support for multirate DSP algorithms. Using these techniques, the characteristics and advantages of an EDSL are shown.

Acknowledgements

A special thank you to Jonatan Midtgaard for his continuous guidance and support throughout the project, and to the Baadsmand family for feeding me for the last five days.

Contents

Chapter 1

Introduction

1.1 Audio Processing

Digital Signal Processing consists of working with continuous signals broken into discrete samples, usually at a fixed sample rate. In the case of audio, the signals represent audio, and are eventually converted either from or to an analog signal, being for example played through speakers. In this thesis the focus is on audio processing, but most of the subjects covered would be applicable in other DSP domains.

Digital audio processing is used in many areas, and on very different hardware. In music production on desktop systems, audio editors called Digital Audio Workstations (DAWs) typically have plugin systems, of which there exist a few standards, with VST[1] being the most popular. These audio plugins, be it effects or synthesizers are a large industry of very specialized and artistic tools that are a big part of most modern music. The same goes for audio effects in embedded contexts, where digital guitar effects especially has been a large industry for many decades, along with processing embedded in live concert equipment, or even apps on smartphones and other portable devices.

In most of these contexts, predictable performance is a requirement, as these are hard realtime systems, where failure to process a signal in time can at best result in unpleasant noise, and at worst damage equipment or the hearing of users or audience. Thus, audio processing usually follows a fixed signal path, i.e. algorithms that work in constant time, independently of the characteristics of the input. In practice, this means little to no branching, no loops of a non-fixed size, and often system-level constraints such as avoiding memory allocation and arbitrary system calls.

1.1.1 Characteristics and Terminology

Digital Audio is 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 32 bit floats, at sample rates of anywhere from 44.1kHz to 192kHz. Buffer sizes vary a lot depending on the realtime requirements, from as low as 16 samples up to 4096. Larger buffers usually allow for better performance, while increasing the processing latency. Low latency is often an important requirement, especially when the processing in question is being used by live musicians, or in other real-time processing situations.

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

[1] https://www.steinberg.net/en/company/technologies/vst3.html

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

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

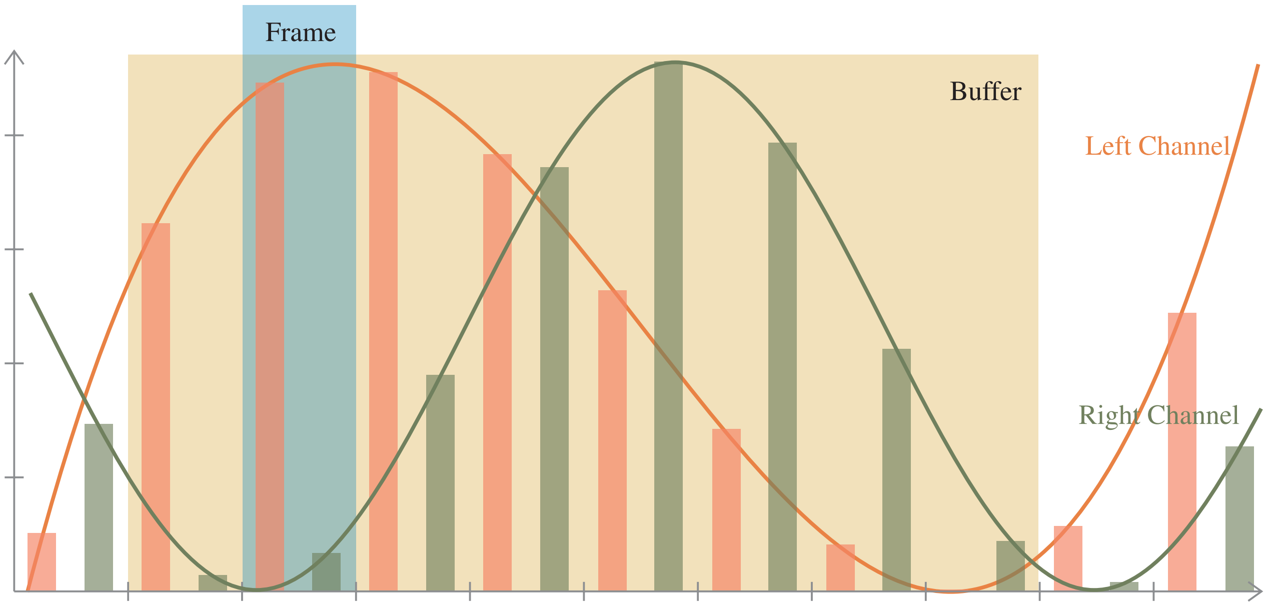


Figure 1.1: A stereo audio signal broken into buffers of 8 frames of two samples each.

what is used by most real time audio frameworks, such as ALSA[2], JACK[3] and VST. An example of using such an interface can be seen in subsection ??.

1.1.2 Constraints & Problems

The area of digital audio signal processing poses some interesting problems from a programming language perspective, especially because of two primary constraints:

Firstly, as mentioned earlier, high and predictable performance and integration with embedded systems is very important, which is why DSP is usually implemented in a low-level systems programming language such as C or C++. As an example, when processing a stereo signal at the standard CD sample rate of 44.1 kHz, one has to be able to process 88 200 samples per second, which leaves around 11.3 µs per sample.

Secondly, audio effects and synthesizers are often designed by people who are not primarily programmers, but creative sound designers who compose low-level DSP algorithms into new tools to be used by musicians. And even when being developed by a programmer with experience in writing low-latency high-performance code, DSP programs are inherently compositional, often resembling an electronic circuit in structure more than an imperative program.

For these reasons, various Domain Specific Languages have been developed that try to solve these issues. One

[2] https://alsa-project.org
[3] https://jackaudio.org

such DSL is FAUST[1], which will be further introduced in **??**, but is especially notable for being the product of a lot of research for the past 20 years. FAUST addresses these issues as a functional language based on an algebra developed to describe signal processors as block diagrams, and compiles to C++ to be easily integrable in existing systems.

Domain specific languages however, while providing syntax and semantics that are tailored to the domain, bring the inherent trade-off of ease of integration and expansion, compared to using a library in a general purpose language.

1.1.3 The Following Chapters

In this project, I will use the basic research on which FAUST is built to develop a C++ library that instead constitutes an *Embedded Domain Specific Language*, or EDSL for audio DSP. I will show the advantages of staying within a general purpose language, and specifically how using operator overloading and expression templates in C++ allows for syntax that closely resembles that of FAUST, while staying easy to extend and integrate with existing systems and technologies.

In chapter 2 I look at existing solutions in this domain, including more detail about FAUST, as well as a few similar solutions in other domains.

In chapter 3 I introduce the block algebra originally developed for FAUST, along with a few extensions and changes applied for this project.

In chapter 4 I use this block algebra to develop a C++ library which implements an EDSL with syntax and semantics based on the previously introduced block algebra.

In chapter 5 I use this library to implement an audio effect plugin, and demonstrate how it can be used in a real-world scenario.

Finally, in chapter 6, I delve into multirate DSP as an example of how the library can easily be extended with even a fairly complex feature that remains unimplemented by FAUST, and in this process cover some interesting intricacies of DSP. *This chapter contains 5359 characters and approximately 1053 spaces = 2.23 standard pages*

Chapter 2

Review of Relevant Material

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

As already mentioned in the introduction, this project builds largely on the research performed by the FAUST authors. It was first developed Orlarey, Fober and Letz at Grame, Centre National de Creation Musciale, Lyon, France in 2002[2], and has for the past 20 years been the basis of a lot of research in the area of expressive DSP programming languages[3, 4, 5, 6].

The authors describe FAUST 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 section **??**. 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.

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

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

```
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 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 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"*[7]. It is especially noteworthy for having a portable FFT implementation that often performs better than FFTW, *the Fastest Fourier Transform in the West*[8], 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[7]. 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"*[9]. 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[10] and MIRA, a middleware for robotics[11].

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[12] merged the long anticipated ranges proposal[13], 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[14], in mind. Concepts is the umbrella term often used to describe the whole system of static type constraints which was introduced in C++20[15], 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[16]. All of this means, that while most of the code in the ranges library could be (and has been[17]) 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

[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

```
std::string to_string(int);
bool is_even(int);

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

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

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 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.

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 11363 characters and approximately 2267 spaces = 4.73 standard pages*

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

Chapter 3

Block Diagrams

Block diagrams are used to describe many kinds of systems in various engineering fields, and are also the primary specification language for DSP algorithms[18, 19, 20, 21, 22]. Besides just being a convention that probably stems from electronic circuit design leading to DSP, block diagrams are a good model for the fixed program structure of a DSP algorithm. Block diagrams are also inherintly recursively defined, where some blocks can be described as block diagrams themselves. The idea of programming DSP in terms of block diagrams has also been used by many DSP DSLs, where visual programming languages that let the user program by drawing connections between blocks such as Pure Data[1] and Max[2] have become very popular, especially with users who are not primarily programmers. It is thus clear that block diagrams are a very useful tool for modelling DSP programs, which is why the researchers that designed FAUST selected it as the model for their DSL[23]. In this chapter I will summarize the algebra designed by them to describe block diagrams as expressions that can be sequentially evaluated.

It is also worth noting, that while DSP is the focus of this project, much of this applies to other domains where block diagrams are a useful model. For example, reactive programming[24] can be modeled using flow diagrams[25], which are very similar to block diagrams.

3.1 Algebra of Blocks

In the 2002 paper *An Algebraic approach to Block Diagram Constructions*[2] Orlarey et al introduce a series of five basic block diagram operations, which are expanded in [23] with two extra compositional operations, split and merge. The rest of this chapter introduces this algebra of blocks in a language very similar to these papers, 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 $(s_1, \dots, s_n) \in \mathbb{S}^n$. 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 $\forall a \in \mathbb{S}^n, b \in \mathbb{S}^m : (a,b) = (a_1, \dots, a_n, b_1, \dots, b_m)$

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

[1]https://puredata.info

[2]https://cycling74.com

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 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[2, 23] 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.

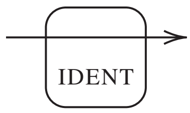
3.1.1 Basic block operations

Each of these seven block operations is described in detail by the FAUST authors in [2] and [23], 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.

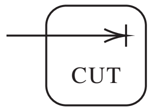
$$
\begin{aligned}
\text{IDENT} : 1 \to 1 \\
[\![\text{IDENT}]\!](s) = s
\end{aligned}
$$



Cut

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

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

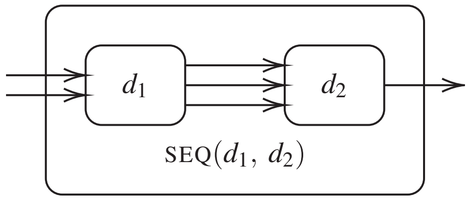


Sequential block composition

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.

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

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

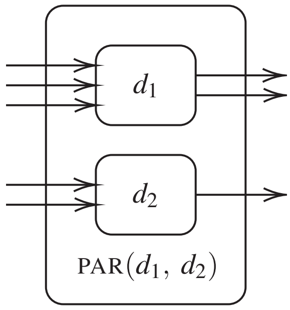


Parallel block composition

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.

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

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



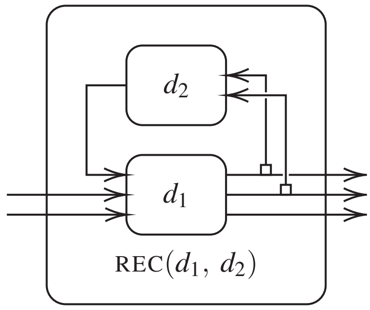
Recursive block composition

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.

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

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



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

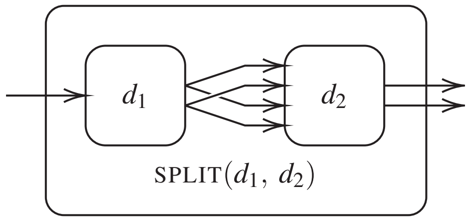
$$\forall y \in \mathbb{S}, t \in \mathbb{N}^+ : y'(0) = 0, y'(t) = y(t-1)$$

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 $\textbf{ins}(d_2)$ must be an exact multiple of $\textbf{outs}(d_1)$.

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

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



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

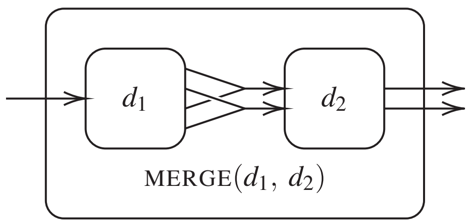
Merge block composition

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 $\textbf{outs}(d_1) = \textbf{ins}(d_2) * k$, 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 $k = 1$.

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

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



3.1.2 Domain Specific Blocks

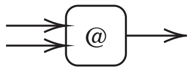
As mentioned in the beginning of this chapter, the full set of blocks includes some domain-specific primitives, denoted by the set $\mathbb{B}$. These are the blocks that perform actual useful operations on the signals, and the blocks defined until now, are used to compose the primitives in $\mathbb{B}$ into block diagrams.

This set can be extended with many more operations, but some of the most important ones are covered here. Faust itself includes quite a few more[23], such as comparisons, branching, and UI elements. During the following chapters more will be defined as well, and adding additional blocks should be easy with the information given here.

Arithmetic

The most basic blocks perform arithmetic operations on pairs of signals. For all operators @ in $+, -, *, /$, a block with the following properties is defined:

$$@ : 2 \to 1$$
$$[\![@]\!](s_1, s_2)(t) = s_1(t)@s_2(t)$$

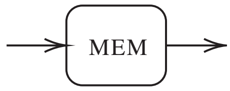


Keep in mind that these blocks are primitive, and while FAUST as well as EDA use infix operator syntax for them, they are not compositional operators, meaning the block itself is not parameterized.

Memory

DSP operations commonly depend on previous signal values, and while the recursive composition often covers those usecases, some are better suited with a simple memory block. It takes one input signal, and outputs that same signal delayed by a single sample, and the very first sample has the value zero:

$$\text{MEM} : 1 \to 1$$
$$[\![@]\!](s)(0) = 0, [\![@]\!](s)(t+1) = s(t)$$



Memory blocks are often chained by sequential composition for longer delays, and this is denoted MEM^n. The corresponding signal processor becomes

$$[\![\text{MEM}^n]\!](s)(t) = \begin{cases} s(t-n) & \text{if } t \geq n \\ 0 & \text{otherwise} \end{cases}$$

Delay

By composing memory blocks, one can get a delay of arbitrary length, however, sometimes this length needs to vary at runtime. For this, we have the DELAY block. Given two signals (d, s), this block outputs s delayed by $d(t)$ samples.

$$\text{DELAY} : 2 \to 1$$
$$[\![\text{DELAY}]\!](d, s)(t) = \begin{cases} s(t - d(t)) & \text{if } t \geq d \\ 0 & \text{otherwise} \end{cases}$$

In implementations of this block, there may need to be some constraint on the value of $d(t)$, and/or some buffer-growth policy that results in slightly different semantics. *This chapter contains 6967 characters and approximately 1475 spaces = 2.90 standard pages*

Chapter 4

The C++ Library

Using the block algebra introduced in the previous chapter, I have developed an embedded domain specific language as a C++ Library called EDA (Expressive DSP for Audio). The goal of this library is to implement a syntax based on the algebra of blocks, and thus resembling FAUST, while staying within the C++ ecosystem, and allowing easy integration in both directions, i.e. using EDA within other C++ frameworks *and* using other C++ libraries and algorithms within EDA.

The library is based on expression templates[26], which is the idea of representing an AST as a static tree of templated types, and then building this AST by overloading operators. For example, the expression `a + (b * 2)` would evaluate to an object of the type `Plus<Var, Mult<Var, Literal>>` or similar. This idea can be used to use native expression syntax to build the AST of the expression instead of actually evaluating it. A further introduction to expression templates in general, and how to implement them in C++ can be found in Bachelet/Yon (2017)[26], but the following sections go through this specific implementation.

It is worth noting that the library makes heavy usage of certain C++20[12] features, especially the features commonly refered to under the umbrella term *concepts*[27], which help constrain the template parameters of templated entities.

4.1 Blocks

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.

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;
```

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

```cpp
  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 subsection 4.6.2).

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.

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

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);
```

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

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

```cpp
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.

```cpp
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.

[2]See https://en.cppreference.com/w/cpp/concepts/copyable

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:

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

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 4.7). 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 $n \to m$ 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

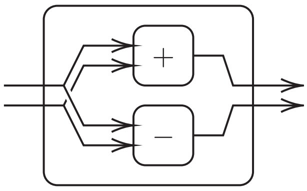


Figure 4.1: A block diagram that computes the sum and difference of its inputs and outputs both

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:

```
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◄**

4.2.1 Arithmetic

The arithmetic blocks defined in section 3.1.2 are declared as basic block types:

```
struct Plus : BlockBase<Plus, 2, 1> {};
struct Minus : BlockBase<Minus, 2, 1> {};
struct Times : BlockBase<Times, 2, 1> {};
struct Divide : BlockBase<Divide, 2, 1> {};
```

And as with the IDENT and CUT blocks, we declare constants to use the blocks:

```
constexpr Plus plus;
constexpr Minus minus;
constexpr Times times;
constexpr Divide divide;
```

With these, we can start to build simple block diagrams, such as the following, which is drawn in Figure 4.1:

```
ABlock<2, 2> auto d = split(par(plus, minus))
```

4.3 Literals and References

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

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

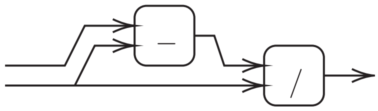


Figure 4.2: A block diagram that computes the relative difference between two signals

```
};

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, ...)` [23]

As a shorthand for `lit(0.5)`, the user-defined literal[3] `0.5_eda` is also provided:

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

4.4 Operator overloads and shorthand syntax

With the block types and construction functions, block diagrams can be declared by composing the constructors. For example, the following declares a block diagram that calculates the relative difference between two signals, i.e. given the values a, b, it outputs $\frac{a-b}{b}$:

```
auto d = seq(par(ident<1>, split(ident<1>, ident<2>)), seq(par(minus, ident<1>), divide);
```

This syntax quickly becomes very verbose, and the prefix functions make it hard to read. So, using operator overloading and single-character constants, this section introduces syntax so the above can be rewritten as

```
auto d = (_, _ << (_, _)) | ((_ - _) / _);
```

4.4.1 Selecting operators

The first step is to decide which C++ operators will be mapped to which block diagram operations, and for this the EDA library has to make some other choices than Faust, as the availability of overloadable operators in C++ obviously places some constraints. The syntax here has been selected to stay as close as possible to faust, but in practice one might consider some changes, as especially overriding the comma operator is usually discouraged in idiomatic C++ ►**citation needed**◄.

Firstly, for the IDENT and CUT blocks, the variables `_` and `$` are used:

[3] see https://en.cppreference.com/w/cpp/language/user_literal

d	Faust Syntax	EDA Syntax
IDENT	`_`	`_`
CUT	`!`	`$`
$\text{SEQ}(d_1, d_2)$	`d1 : d2`	`d1 | d2`
$\text{PAR}(d_1, d_2)$	`d1 , d2`	`d1 , d2`
$\text{REC}(d_1, d_2)$	`d1 ~ d2`	`d1 % d2`
$\text{SPLIT}(d_1, d_2)$	`d1 <: d2`	`d1 << d2`
$\text{MERGE}(d_1, d_2)$	`d1 :> d2`	`d1 >> d2`

Table 4.1: Faust and EDA syntax for basic block diagram components.

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

For the compositional operators, most of them follow a similar structure. Sequential composition is done with `:` in faust, but since `:` is not an operator in C++, the bitwise OR operator (`|`) is selected instead. This choice follows naturally from the C++20 ranges library[12], which uses the operator for a similar purpose, and in turn is inspired by the UNIX shell pipe[13].

Parallel composition uses the `,` operator, recursion has been changed to the modulo operator `%`, and split/merge use the left/right shift operators << and >>.

The final selection of operators and syntax compared to faust, can be seen in Table 4.1. All in all these operators follow a similar structure, so EDA and Faust programs should read fairly similarly.

4.4.2 Operator overloads

All of the selected operators can in C++ be overloaded as free functions, which makes the implementation very simple. They all follow the exact same pattern, so as an example, here is the implementation of the `|` operator:

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

Notably, this operator takes two references to arbitrary objects, and delegates to the constructor function `sequential`. However, two interesting things are going on.

Firstly, when implementing an operator overload template in namespace scope, it is very important to make sure that the template arguments are propperly constrained, since the operator would otherwise be valid in ambiguous situations, such as when `Lhs` and `Rhs` are both `int`. The approach used here is to use a requires-clause to make the operator only participate in overload resolution if either `Lhs` *or* `Rhs` to model **`AnyBlockRef`**. Then, both arguments are passed through the function `as_block`, before being passed to `sequential`.

Autoboxing of literals

The purpose of this, is to allow literal floating-point values to be involved in EDA block expressions, i.e. allow the expression `_ * 0.5`, which takes a signal and multiplies it by the floating point value 0.5. Without auto-boxing literals, the equivalient expression would be `_ * literal(0.5)` (or `_ * 0.5_eda` with the user-defined literal introduced in section 4.3).

The implementation of `as_block` is very simple. It consists of two overloads, one that takes an object that models

`AnyBlockRef`, and returns it unchanged, and one that takes a float, and wraps it in a `Literal` block. By adding more `as_block` overloads, one can add implicit convertions from other types to blocks as well, for example some may want a `as_block(float*)` overload to wrap a pointer to a float as a `Ref` block. This would allow expressions of the form `(_ * &x)` to equal `(_ * ref(x))`.

```
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);
}
```

As an extra utility, the `as_block_t<T>` type trait is defined to get the result type of calling `as_block` on an instance of `T`:

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

One problem with auto-boxing is that it still requires at least one operand to be a block type. This can result in issues when for example `(_ , 2)` is a block that takes one signal, and outputs a tuple of the input unchanged and the constant signal 2, but `(1, 2)` is just the value `int(2)`, and not a block with no inputs that outputs the constant signals 1 and 2. This is because `(1, 2)` has selected the standard C++ comma-operator, which returns its last argument, and not the overloaded comma-operator from EDA, which builds a parallel block. This means the second expression would have to be written with at least one argument explicitly converted to a block type, like `(1_eda, 2)`. Whether this slight decrease in verbosity is worth the added complexity is left up to the reader, as simply removing the `as_block` calls and requiring both operands to model `AnyBlockRef` should be an easy change to make to the relevant code. For completeness sake, the library as described here includes auto-boxing.

4.5 Evaluating Signal Processors

Up until now, this chapter has only described the declarations of block diagrams, and nothing about how to evaluate the corresponding signal processors. As mentioned in the beginning of this chapter, the declarations of block types are completely decoupled from the evaluators of the signal processors. This visitor-based design makes sense for a couple of reasons, but most importantly, it allows for other visitors than just the signal processor evaluator. For example, one could build a visitor that generates a visual representation of the block diagram, or one that generates a user interface. Alternatively, other evaluators could be built, for example one that vectorizes the operations (see Scaringella et al. *Automatic vectorization in Faust* (2003) [28] for an aproach that could be taken in EDA as well)

As presented here however, EDA contains only one visitor, the `evaluator<Block>`, which evaluates the signal processor of a block for a single frame of data at a time, i.e. the values $(s_0(t), \ldots, s_n(t))$ of a signal tuple at a single time t. In code, this frame is represented as an instance of the `Frame<N>` class, where `N` is the number of channels. This class can mostly just be regarded as a wrapper around `std::array<float, N>`, with some minor tweaks in construction. It also provides the free functions `slice<I, J>(Frame<N>) -> Frame<J - I>` and `concat(Frame<N>, Frame<M>) -> Frame<N + M>`, which are mostly self explanatory, and thus also omitted here. The full class and function definitions can be seen in **►appendix ???◄**.

To illustrate the relationship between the block declarations and evaluator instances, consider the following program, which evaluates the difference between its current input and its previous input:

```
constexpr auto d = _ << (_ - mem<1>);
auto a = make_evaluator(d);
```

```
auto b = make_evaluator(d);

a.eval(1) // ⇒ 1 − 0 = 1
a.eval(2) // ⇒ 2 − 1 = 1
b.eval(10) // ⇒ 10 − 0 = 10
a.eval(5) // ⇒ 5 − 2 = 3
b.eval(1) // ⇒ 1 − 10 = −9
```

Notice the `constexpr` on line 1 - the block diagram is declared as a compile-time constant, however, the evaluators are runtime mutable, as they store the state of the signal processor. Continuing the example, note that the two evaluators have separate memory, and can be executed independently from each other. This is the basic usage of evaluators, and should give an understanding of the difference between blocks as declarations of programs, and evaluators as instances of them.

4.5.1 evaluator

The basic evaluator class template is declared as such:

```
template<AnyBlock T>
struct evaluator;
```

It is acompanied by the following type trait to extract the block type:

```
template<typename T>
struct block_for;

template<typename T>
struct block_for<evaluator<T>> {
  using type = T;
};

template<typename T>
using block_for_t = typename block_for<T>::type;
```

For each block type `T`, the class template `evaluator<T>` shall be specialized to define the evaluator. This specialization shall model the **`AnEvaluator`** concept, which is given in code below, and has the following requirements:

1. `T` shall publicly derive from `EvaluatorBase<block_for_t>` (see the following section).
2. `T` shall be constructible from a const reference to an object of type `block_for_t<T>`.
3. `T` shall have a member function `eval`, that takes a `Frame` of the appropriate number of channels, and returns a `Frame` of the appropriate number of channels, according to the signature of the block `block_for_t<T>`.

```
template<typename T>
concept AnEvaluator =
  std::derived_from<T, EvaluatorBase<block_for_t<T>>>
  && std::is_constructible_v<T, block_for_t<T> const&>
  && requires (T t, Frame<ins<block_for_t<T>>> in) {
    { t.eval(in) } -> std::convertible_to<Frame<outs<block_for_t<T>>>>;
  };
```

To construct the evaluator of a block diagram, the factory function `make_evaluator` is supplied:

```
template<AnyBlockRef T>
constexpr auto make_evaluator(T&& b)
requires AnEvaluator<evaluator<std::remove_cvref_t<T>>>
{
  return evaluator<std::remove_cvref_t<T>>(b);
}
```

EvaluatorBase

For primitive blocks, `EvaluatorBase` is an empty base class.

```
template<AnyBlock T>
struct EvaluatorBase {};
```

It could however be used to add common functions to all evaluators, like a wrapper to `eval` that works on whole buffers of frames. Other than that, its main purpose is for composition evaluators.

Primitive block evaluator

For primitive blocks, the evaluator implementations are fairly simple, one just needs to make sure to follow the three requirements of **`AnEvaluator`**, i.e. deriving from `EvaluatorBase<Block>`, having a `evaluator(Block)` constructor, and implementing the propper `eval` function.

As an example, here is the `evaluator<Plus>` specialization:

```
template<>
struct evaluator<Plus> : EvaluatorBase<Plus> {
  constexpr evaluator(Plus) {};
  Frame<1> eval(Frame<2> in)
  {
    return {in[0] + in[1]};
  }
};
```

4.5.2 Composition Evaluators

Evaluators of block compositions follow the exact same model, however, they need to defer to evaluators of the operands, stored as member variables. This is done through the `EvaluatorBase` specialization for blocks that model **`AComposition`**, and using some light metaprogramming, the evaluators are stored in a `std::tuple<evaluator<Operand>...>`, constructed from the operand blocks accessed through the block composition passed to the constructor. The implementation looks like this:

```
namespace detail {
  template<typename T>
  struct add_evaluator {};

  template<typename... Ts>
  struct add_evaluator<std::tuple<Ts...>> {
    using type = std::tuple<evaluator<Ts>...>;
  };

  template<typename T>
  using add_evaluator_t = typename add_evaluator<T>::type;
} // namespace detail

template<AComposition T>
struct EvaluatorBase<T> {
  constexpr EvaluatorBase(const T& t) : operands(t.operands) {}
  detail::add_evaluator_t<operands_t<T>> operands;
};
```

The important part is that classes that derive from `EvaluatorBase<T>` can access the operand evaluators through `std::get<I>(this->operands)`, where `I` is the index of the operands.

As in the other sections on compositions, the SEQ composition is used as an example of an evaluator:

```
template<AnyBlock Lhs, AnyBlock Rhs>
struct evaluator<Sequential<Lhs, Rhs>> : EvaluatorBase<Sequential<Lhs, Rhs>> {
  constexpr evaluator(const Sequential<Lhs, Rhs>& block) : EvaluatorBase<Sequential<Lhs, Rhs>>(block) {}

  constexpr Frame<outs<Sequential<Lhs, Rhs>>> eval(Frame<ins<Sequential<Lhs, Rhs>>> in)
  {
    auto l = std::get<0>(this->operands).eval(in);
    return std::get<1>(this->operands).eval(l);
  }
};
```

Notice that this is all very similar to the primitive block evaluator shown earlier, and the only difference is forwarding the block instance to the `EvaluatorBase` constructor. The `eval` function implements the signal processor for $\mathrm{SEQ}(l,r)$, by first directly calling the evaluator of the first operand, and passing the result of that operation to the evaluator of the second operand.

The other compositional blocks are slightly more complicated to implement, but at this point it is just "normal" C++ in the `eval` functions. The full implementations can be seen in ►**appendix ??**◄

4.6 Extra features

One of the advantages of working inside C++ instead of in a DSL, is the ability to easily add features and integrations with other parts of the C++ ecosystem. At the time of writing, the EDA library contains a few such examples, and a couple of the more interesting ones are covered in this section. I will not go deep into the implementation details here, but the code can be seen in ►**appendix ??**◄.

4.6.1 Functions

A simple, but important enhancement to the library, is the `fun<I, O>(Callable)` block. It can be used to easily wrap normal C++ functions to stateless blocks. As an example, the following defines blocks for the hyperbolic tangent function (as used in chapter 6), and floating point modulo.

```
auto tanh = fun<1, 1>(&std::tanhf);
auto mod = fun<2, 1>([](auto in) { return std::fmod(in[0], in[1]); });
```

The `fun` block simply captures the function, and the evaluator delegates to it directly. The implementation can be seen in ►**appendix ??**◄.

A stateful version of `fun` also exists, which can store arbitrary state in the evaluator. `fun<I, O>(f, state_inits...)` copies `state_inits...` into new objects on each evaluator construction, and the evaluator passes references to these objects as extra arguments to `f(data, states...)`.

As an example, the following $\mathrm{COUNTER} : 0 \to 1$ block outputs the signal $s(t) = t$, by keeping a state of type `int`, initialized to 0, and incrementing it each time the evaluator is called:

```
auto counter = fun<0, 1>([](Frame<0> in, int& state) { state++; return {state}}, 0);
```

By using these constructs, most simple blocks, stateful or stateless, can be implemented without having to write out the type and evaluator declarations.

4.6.2 Function call syntax for blocks

FAUST has the ability to call blocks as functions, meaning $d(x_1,\ldots,x_{\mathbf{ins}(d)})$ is equal to $\textsc{seq}(\textsc{par}(x_1,\ldots,x_{\mathbf{ins}(d)}),d)$. To make this even more useful, partial application is supported, i.e.

$$d(x_1,\ldots,x_n) = \textsc{seq}(\textsc{par}(x_1,\ldots,x_n,\textsc{ident}^{\mathbf{ins}(d)-\sum_{i=1}^{n}\mathbf{outs}(x_i)}),d)$$
$$\sum_{i=1}^{n}\mathbf{outs}(x_i) \leq \mathbf{ins}(d)$$

This description can be translated directly to an implementation of the function call operator on `BlockBase`, making it available to all blocks:

```
template<typename D, std::size_t I, std::size_t O>
constexpr auto BlockBase<D, I, O>::operator()(auto&&... inputs) const noexcept
requires((outs<Inputs> + ...) <= I)
{
  return seq(par(inputs..., ident<I - (outs<Inputs> + ...)>), *this);
}
```

4.7 Compilation Errors

A big advantage of using C++20 requirements and constraints, is the vastly improved error context. As an example, the following EDA expression violates the constraints of the SEQ block.

```
auto block = (_, _) | (_);
```

When compiled with clang v12.0.0, this code gives the following error message:

```
In file included from tests/block.cpp:1:
include/eda/block.hpp:185:12: error: constraints not satisfied for class template 'Sequential' [with Lhs =
↪  eda::Parallel<eda::Ident<1>, eda::Ident<1>>, Rhs = eda::Ident<1>]
    return Sequential<std::remove_cvref_t<Lhs>, std::remove_cvref_t<Rhs>>{{FWD(lhs), FWD(rhs)}};
           ^~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
include/eda/syntax.hpp:38:12: note: in instantiation of function template specialization
↪  'eda::seq<eda::Parallel<eda::Ident<1>, eda::Ident<1>>, const eda::Ident<1> &>' requested here
    return seq(as_block(FWD(lhs)), as_block(FWD(rhs)));
           ^
tests/block.cpp:19:21: note: in instantiation of function template specialization
↪  'eda::syntax::operator|<eda::Parallel<eda::Ident<1>, eda::Ident<1>>, const eda::Ident<1> &>' requested here
auto block = (_, _) | (_);
                    ^
include/eda/block.hpp:178:12: note: because 'outs<eda::Parallel<eda::Ident<1>, eda::Ident<1> > > ==
↪  ins<eda::Ident<1> >' (2 == 1) evaluated to false
  requires(outs<Lhs> == ins<Rhs>) //
           ^
```

Error messages emitted from programs that rely heavily on templates, but are implemented before or without explicit constraints have notoriously long and cryptic error messages[4], so comparatively this goes straight to the point.

To give the compiler as much context as possible for errors, it can be useful to use the **`ABlock`** concept as a placeholder type when declaring blocks:

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

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

[4] In his CppCon 2019 talk *How to Implement Your First Compiler Feature:The Story of Concepts in Clang*[16], Saar Raz famously talks about how he had to first implement a parser for the gigabyte of compiler error messages produced by one such program.

4.8 Conclusion & Further Work

In this chapter I have shown how an EDSL based on FAUSTs algebra of blocks can be implemented in C++, using the basic techniques of expression templates to build an AST that can then be evaluated by a separate class.

The evaluator shown here is the simplest one possible, as it merely walks the AST, evaluating one sample at a time. However, other evaluators could be explored that work more efficiently, such as a vectorizing evaluator as mentioned in section 4.5. Additionally, the AST represented by block diagram declarations could be transformed and optimized, detecting patterns and redundancies that can be reduced to other blocks.

Additionally, visitors for other purposes could be built, for example one that generated a visual representation of the block diagram, or more specialized ones that could be used to generate user interfaces from block diagrams that contain special control blocks.

However, as presented here, the library can already be used to implement real applications, as will be explored in the next chapter.

This chapter contains 21562 characters and approximately 4416 spaces = 8.98 standard pages

Chapter 5

Using the Library

This chapter explores using the C++ library designed and developed in the previous chapters to implement an audio plugin that can be used by audio host applications. This is an ideal usecase for the library, as audio plugins are often implemented in C++, and follow a similar structure to the signal processors introduced in chapter 3, i.e. are defined by a number of input/output channels, and evaluated on buffers of data passed in one at a time.

5.1 The Effect

The example developed in this chapter is a simple echo audio effect, i.e. a plugin that continuously repeats its input with some delay, where each echo gets quieter and looses "brightness", i.e. the high frequency components of the signal. Echos were introduced in the 1950s, and have become a staple of music production, being used especially for vocals and guitars. The first analog echos were based on a tape loop, where the input audio was continuously written and then read back. The continuous repetitions provided by an echo are achieved using a feedback loop, meaning the output of the echo is sent back into the input, but with reduced gain and/or other processing applied.

The echo effect designed here is a simple, but fully featured one. It has a variable delay time, feedback gain control, and a simple low-pass filter in the feedback loop to gradually remove more and more high frequencies from the repeated signal. Lastly it has a dry/wet mix control, to mix the original (dry) signal with the output (wet) signal of the effect. The block diagram for the effect can be seen in Figure 5.1, and the following sections will go through this block diagram one part at a time.

5.1.1 Controls

The effect has 4 control variables, which are exposed through LV2, and controlled by the UI generated by the plugin host. In code, these are represented as `float` variables, which can be used in the EDA expression by wrapping them in a REF block (see section 4.3).

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

The meaning of the individual controls will be explained where they are used in the following sections.

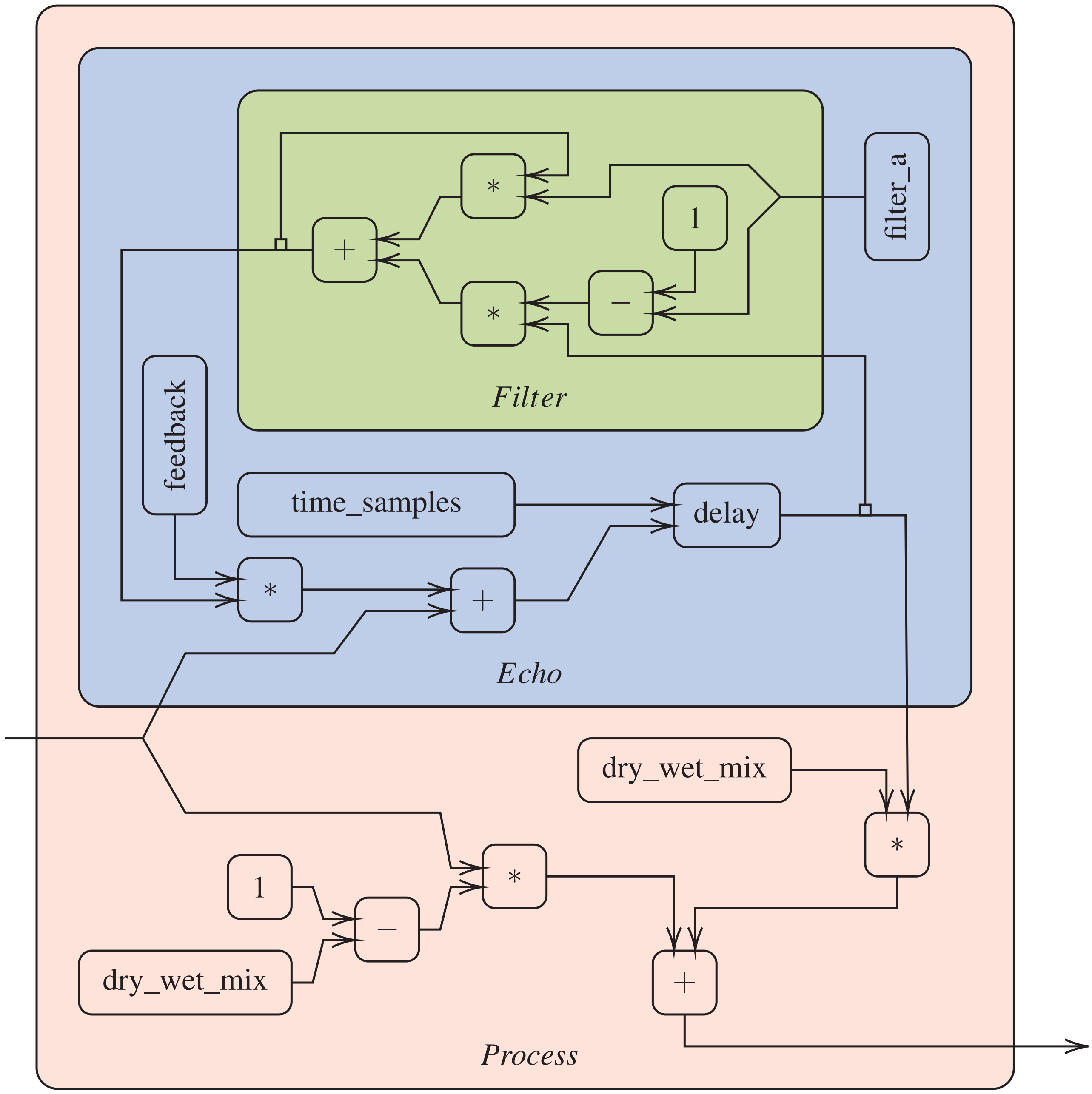


Figure 5.1: Block diagram of the echo effect.

5.1.2 Filter

Echoes often include a low-pass filter in their feedback loop, meaning each repetition of the echo will not only be quieter, but also proportionally lose more high frequencies than low. This seems most natural to the human ear, since low frequencies travel further, and will thus be louder when a sound returns as an echo.

The low-pass filter used in this echo is the simplest DSP filter possible, a single-pole Infinite Impulse Response filter. It can be understood as a weighted average between the input, and the previous output, i.e its signal processor can be written as follows:

$$\begin{aligned}
[\![Filter]\!](a,s) &= y \\
y(-1) &= 0 \\
y(t) &= (1-a(t))\cdot s(t) + a(t)\cdot y(t-1) \\
a(t) &\in [0;1]
\end{aligned}$$

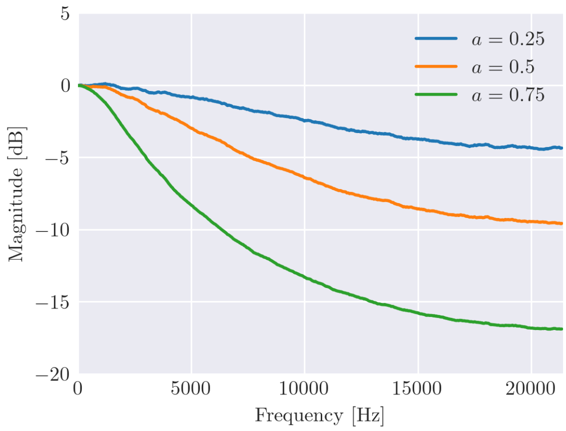


Figure 5.2: Frequency response of the single-pole IIR filter used in the echo effect at different values of the a parameter.

Here, s is the audio input signal, and a is a signal used to control the weighting between the input and previous output, with higher values meaning high frequencies are more suppressed. For an illustration of this parameter, see Figure 5.2, which plots the magnitude of frequencies at a selection of values for a.

This particular filter is chosen mainly for the simplicity of its implementation, but it also fits the purpose well, and results in a quite nice effect.

The EDA implementation of the filter follows from the relevant (green) part of the block diagram in Figure 5.1, and can be written as such:

```
ABlock<2, 1> auto const filter = (_ << (_, _), _) | (((_ * _) + ((1 - _) * _)) % _);
```

Notably, this implementation uses the REC block (`%`) to pass the result of the previous iteration back into the filter. It also uses the SPLIT block (`<<`) to send a to both parts of the expression. By passing a as a signal, this filter becomes easily reusable, and could be included in a library of standard components.

5.1.3 Echo

This (blue) part of the block diagram contains the feedback loop and the delay, which perform the main work of the effect. It is once again a recursive algorithm, meaning part of its input is its previous output. Specifically, the output is passed through the filter discussed previously, multiplied by the feedback gain control, and summed with the new input, to then be delayed by the specified echo time.

This block uses three control variables: `feedback`, which controls the feedback gain, `time_samples` which sets the delay time in number of samples, and `filter_a` which is the a control passed to the filter. The signal processor for this part is

$$\begin{aligned}
[\![Echo]\!](s)(t) &= x(t) \\
\forall t < 0 : x(t) &= 0 \\
\forall t \geq 0 : x(t) &= [\![Filter]\!](x')(t - time_samples)\cdot feedback + s(t) \\
x'(t) &= x(t-1) \\
feedback &\in [0;1]
\end{aligned}$$

Implemented in EDA, the code looks as follows:

```
ABlock<1, 1> auto const echo = (plus | delay(ref(time_samples))) % (filter(ref(filter_a)) * ref(feedback));
```

This uses the DELAY block introduced in section 3.1.2 to get the variable time delay, and partial function application (see subsection 4.6.2) to pass control signals with a nicer syntax.

5.1.4 Process

Lastly, the outermost (red) part of the block diagram in Figure 5.1 contains the dry/wet mix, to mix between the unprocessed (dry) and processed(wet) signal. This is a very common control to have on all kinds of audio effects, as it (together with the feedback gain control) is an easy way to get *more* or *less* of the effect. The signal processor here is quite simple:

$$\llbracket Process \rrbracket(s)(t) = mix \cdot \llbracket Echo \rrbracket(s)(t) + (1 - mix) \cdot s(t)$$
$$mix \in [0;1]$$

The EDA implementation splits the input signal to both sides of a `+` operation, with the one side passing through the `echo` block defined earlier, and one through the identity block `_`, with both sides multiplied by the appropriate mix factor:

```
ABlock<1, 1> auto const process = _ << (echo * ref(dry_wet_mix)) + (_ * (1 - ref(dry_wet_mix)));
```

5.2 Comparison with FAUST

This example is a good opportunity to compare the EDA code to equivalent FAUST code. Here is the full EDA program declaration from the previous section:

```
auto filter = (_ << (_, _), _) | (((_ * _, (1 - _) * _) | plus) % _);
auto echo = (plus | delay(ref(time_samples))) % (filter(ref(filter_a)) * ref(feedback));
auto process = _ << (echo * ref(dry_wet_mix)) + (_ * (1 - ref(dry_wet_mix)));
```

And to compare, the equivalent code in FAUST:

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

A few syntactical differences can be seen. Setting aside the obvious `auto` prefix and difference in operators, the main difference is FAUSTs support for function declaration syntax, which makes the `filter` declaration a bit simpler to read. Mostly however, the differences are small, and more work could be done to improve the syntax of EDA even further, where even something similar to fausts function declarations should be possible.

5.3 The Plugin

The framework chosen to implement the echo plugin is LV2[1], an open standard for audio plugins, supported by applications such as Audacity[2], Ardour[3] and REAPER[4]. LV2 is similar to the more widely used alternatives such as VST[5] and AU[6], and is chosen here only for practical reasons, such as better linux support and availability in

[1] https://lv2plug.in
[2] https://www.audacityteam.org/
[3] https://ardour.org/
[4] https://www.reaper.fm/
[5] https://www.steinberg.net/en/company/technologies/vst3.html
[6] https://developer.apple.com/documentation/audiounit

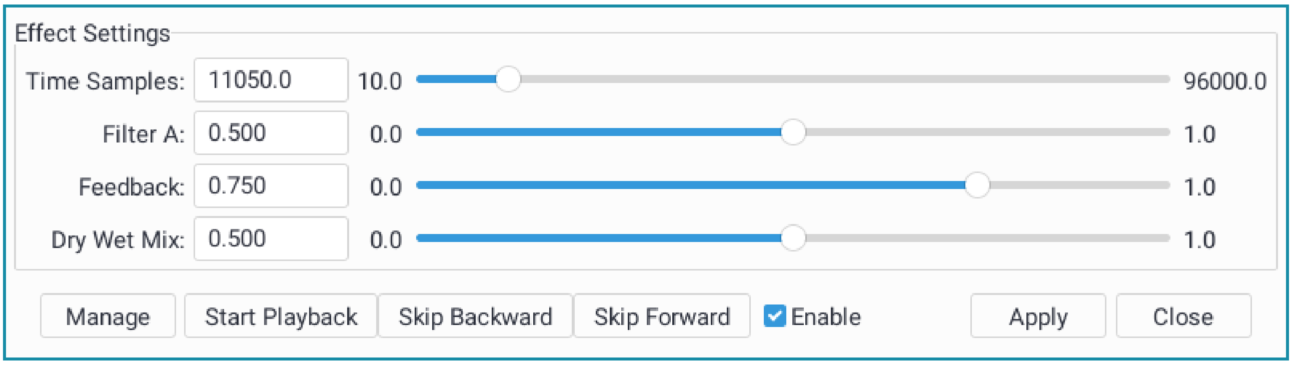


Figure 5.3: The UI of the Echo plugin as displayed in Audacity.

Audacity, which provides useful analysis tools for scientific purposes, such as the frequency spectrum analyzer used to generate Figure 5.2 and similar plots in chapter 6.

An LV2 plugin consists of data definition files in the *turtle* specification language, and implementation in C/C++. For the EDA plugins, I wrote a small wrapper to write plugins as a C++ class instead (see ►**appendix ??**◄), but it follows the structure of the C API closely.

The C++ code that implements the echo plugin can be seen in Listing 5.1. It consists of three functions, that each are provided to the LV2 framework.

`connect_port` is called by the framework to provide pointers to the data ports. The four control are single values, and the two audio port pointers are arrays. All will be updated with values before each call to `run`.

`activate` is called after all ports have been connected, before the first call to `run`, and in it the EDA blocks and evaluator are constructed. The evaluator is stored in a `DynEvaluator` a type-erasing wrapper for evaluators (see ►**appendix ??**◄). Since the control ports are pointers to float values, they can be used directly with the REF block.

`run` is called to process incoming audio data. Its single parameter `n_samples` is the number of samples available in the audio port arrays given earlier, so the implementation is a simple loop through and call to the evaluator constructed in `activate`.

This implementation, along with a metadata file (see ►**appendix ??**◄), defines the plugin which is distributed as a folder containing the shared object binary and metadata. When installed on the system, these can be used from a host program that supports LV2, such as Audacity, which will then provide a UI for the controls. The UI generated for this effect by audacity can be seen in Figure 5.3.

5.4 Performance Testing

Only very minimal performance testing of the library has been done, and only of this single example program. The performance was compared to an equivalent handwritten C++ program, by measuring 1000 iterations of processing buffers of 1024 samples each. The code for this benchmark can be seen in ►**appendix ??**◄.

Program	Average time pr. iteration
Handwritten C++	11.198 µs
EDA	16.770 µs
Relative Difference	+49.7%

```cpp
struct Echo final : LV2Plugin {
  Echo() = default;

  void connect_port(uint32_t port, float* data) override
  {
    switch (port) {
      case 0: time_samples = data; break;
      case 1: filter_a = data; break;
      case 2: feedback = data; break;
      case 3: dry_wet_mix = data; break;
      case 4: input_port = data; break;
      case 5: output_port = data; break;
    }
  };

  void activate() override
  {
    ABlock<2, 1> auto const filter = (_ << (_, _), _) | (((_ * _, (1 - _) * _) | plus) % _);
    ABlock<1, 1> auto const echo = (plus | delay(ref(*time_samples))) % (filter(ref(*filter_a)) *
    ↪ ref(*feedback));
    ABlock<1, 1> auto const process = _ << (echo * ref(*dry_wet_mix)) + (_ * (1 - ref(*dry_wet_mix)));
    eval = make_evaluator(process);
  }

  void run(uint32_t n_samples) override
  {
    for (int i = 0; i < n_samples; i++) {
      output_port[i] = eval(input_port[i]);
    }
  }

private:
  DynEvaluator<1, 1> eval;

  float* time_samples = nullptr;
  float* filter_a = nullptr;
  float* feedback = nullptr;
  float* dry_wet_mix = nullptr;

  float* input_port = nullptr;
  float* output_port = nullptr;
};
```

Listing 5.1: Implementation of Echo LV2 plugin

As a simple benchmark, this shows a performance penalty of roughly 50%, which is very promising for the completely unoptimized version of the library. It seems highly likely from this that some of the optimizations mentioned in section 4.8 could reduce this number a lot, but of course more measurements would need to be done in tandem with optimizations.

5.5 Conclusion

By showing a small EDA program and real-life usecase for the library, this chapter has illustrated the merits of the library. I also compared the EDA implementation to an equivalent FAUST program, showing the minor differences in syntax. To further explore the library in practical usage, more examples and testing should be done, leading to further development and optimization of the library. However, even as presented in this project, the library can clearly be used to implement at least semi-complex DSP algorithms and integrate them in existing systems, with acceptable performance. *This chapter contains 8725 characters and approximately 1857 spaces*

= 3.64 standard pages

Chapter 6

Multirate DSP Algorithms

This chapter explores resampling, and working with a signal path that uses multiple sample rates. Multirate DSP is not supported by FAUST[1], and this chapter 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. This is all intended as an example of some advantages of working with a C++ library compared to a DSL.

6.1 Aliasing

According to the Nyquist-Shannon theorem[29], 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 analog 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[30], 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.

6.1.1 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 to heavilly to the graphs in Figure 6.1. 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

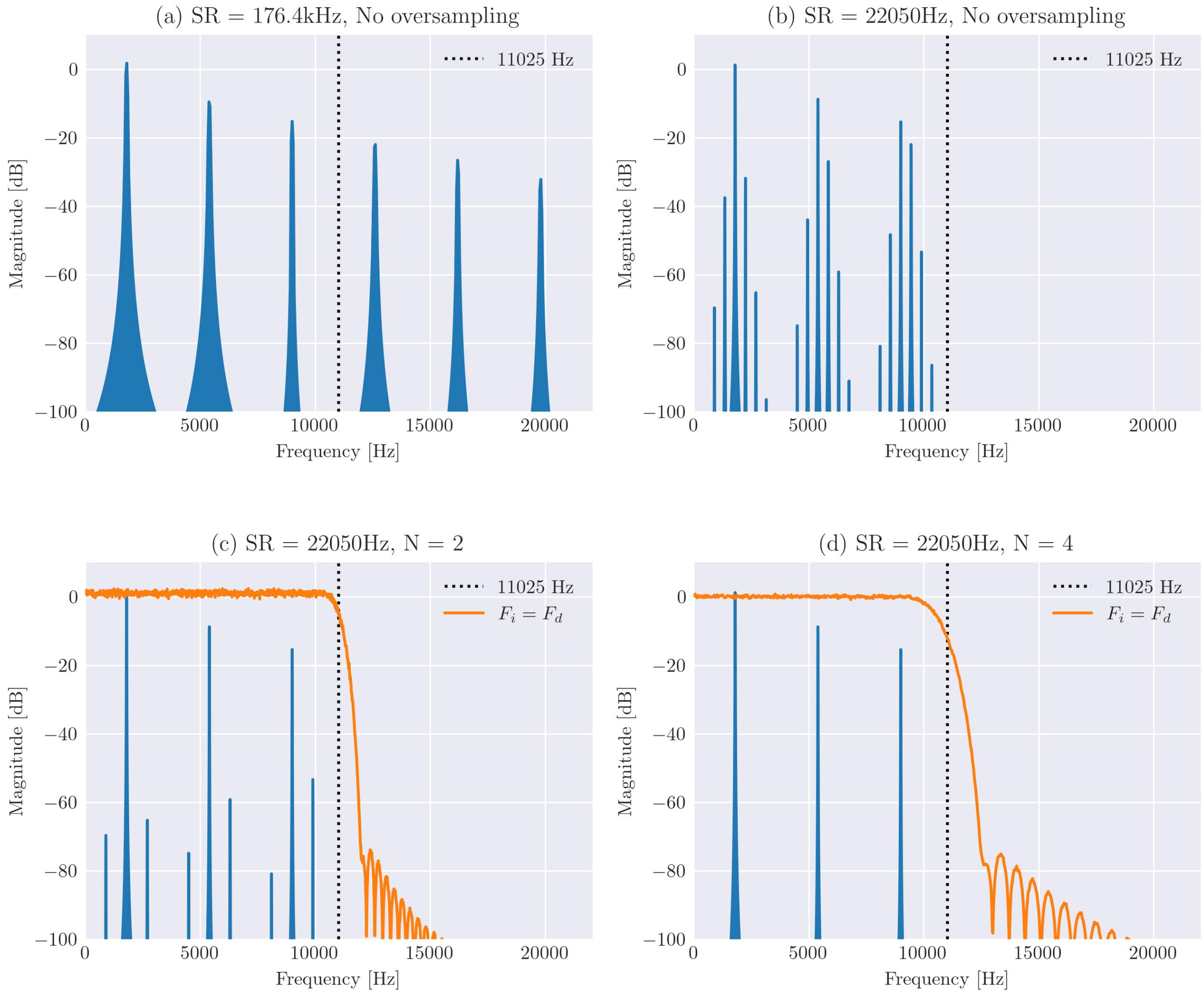


Figure 6.1: A 1800 Hz sine wave passed through the nonlinear waveshaping function $\omega(x) = \tanh(5x)$ at different sample rates and resampling factors N. Given the high samplerate, (a) can be seen as representing the ideal signal over this frequency range. (c) and (d) additionally show the frequency response of the interpolation/decemation filters used with oversampling. For better visualization, the logarithmic decibel scale is used on the vertical axis: $z\,\text{dB} = 20 \cdot \log_{10}(z)$.

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 6.1 (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.

6.2 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 6.1, 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 [18]

There are other and more efficient ways to avoid aliasing[31], 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[18, 32].

6.2.1 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[33, 19], 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 6.2 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.

6.2.2 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

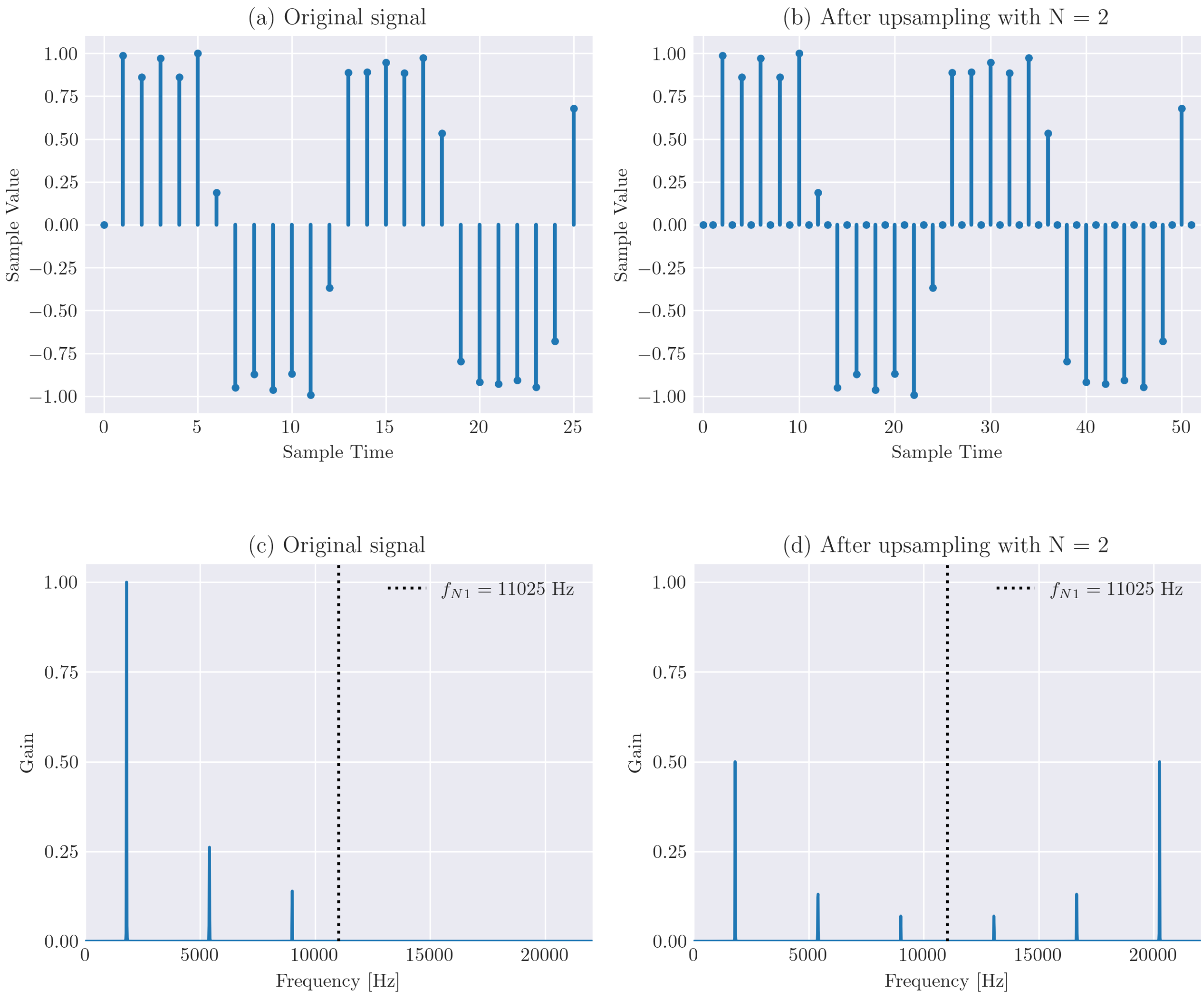


Figure 6.2: The effects of zero-stuffing on the waveform and frequency spectrum. The input signal is the same saturated 1800Hz sine wave as used in Figure 6.1, and the sample rates are 22050Hz and 44100Hz before and after upsampling respectively. To clearly show the reduction in gain on upsampling, linear 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[18, 33, 20], and not something I will go into in this project, however, the filters used for testing and for Figure 6.1, are FIR filters of order $N = 128$, designed using the method described in [22]. 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.

6.2.3 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 [18, 33, 20].

6.3 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.

6.3.1 Approach I

In the most general model, multirate blocks are introduced by adding an extra parameter R to signals, as the ratio of the sample rates f_{out}/f_{in}. This means blocks will be defined with the types $i \to_R o$, and block compositions need type rules to compute the rate ratio as well. As an example, here is the type rule as it would be stated for SEQ:

$$\frac{d_1 : n \to_r p \qquad d_2 : p \to_q m}{\text{SEQ}(d_1, d_2) : i_1 \to_{r \cdot q} o_2}$$

This means the rate ratio of $\text{SEQ}(d_1, d_2)$ is the product of the rate ratios of d_1 and d_2. Similarly, $\text{PAR}(d_1, d_2)$ requires both operands to have the same rate ratio, which becomes the rate ratio of the compositio n. One would then introduce $\text{UPSAMPLE}(n) : 1 \to_n 1$ and $\text{DOWNSAMPLE} : 1 \to_{\frac{1}{n}} 1$ blocks, which could be used to change the sample rate.

This model has the advantage of allowing other uses for resampling, such as interfacing between two systems that use different sample rates, as a top-level signal processor can have different input/output sample rates. However, it introduces a large amount of complexity to evaluating the signal processors, since blocks that downsample can only output after receiving $\frac{1}{R}$ samples, and blocks that upsample output R samples at a time.

6.3.2 Approach II

Instead, a simpler approach is chosen, where all blocks have the same input/output rate, but a separate $\text{RESAMPLE}\langle N\rangle(d, F_i, F_d)$ block is introduced, which can be used to perform oversampling for the block d, with ratio N, interpolation filter F_i, and decimation filter F_d. This means, that while d is evaluated at a higher sample rate, the full RESAMPLE block outputs at the same sample rate as its input. This block can be seen in Figure 6.3, and the type rules and signal processesor of it are described here:

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

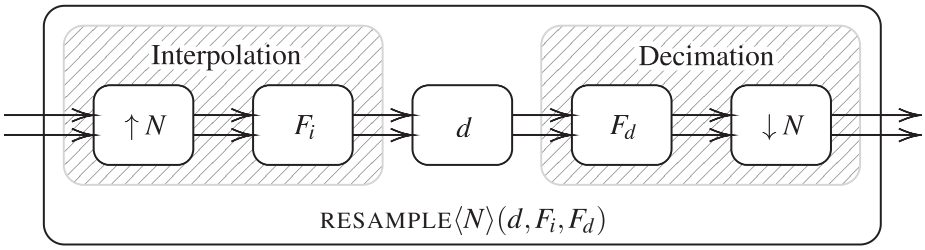


Figure 6.3: The resample block as implemented in the EDA library

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

While this approach does not allow a signal processor to output at a different rate than its input, it still covers most relevant usecases. For instance, all plugin frameworks mentioned in the previous chapter have this constraint. In general, actual sample-rate conversion is usually done at the edge of the system, and in the cases where it is needed, it can easily be done outside of the component described as a block diagram.

6.4 Multirate in the EDA Library

As with most of the constructs described in chapter 4, translating the resampling block to C++ for the EDA library is a fairly straight forward process. RESAMPLE is a compositional block, so it inherits from `CompositionBase`. In the implementation shown here, the filters are just sequentially composed around the inner block in the construction function, but in an optimized implementation custom filters would be used, with optimizations such as not invoking the decimation filter on the samples that are discarded during downsampling (see subsection 6.2.3).

```
template<int N, AnyBlock Block>
requires(N > 1)
struct Resample : CompositionBase<Resample<N, Block>, ins<Block>, outs<Block>, Block> {};

template<int N>
constexpr auto resample(AnyBlock auto block, AnyBlock auto f1, AnyBlock auto f2)
{
  auto filter_block = seq(f1, block, f2);
  return Resample<N, decltype(filter_block)>{{filter_block}};
}
```

The evaluator is where the interesting parts recide:

```
template<int N, AnyBlock Block>
struct evaluator<Resample<N, Block>> : EvaluatorBase<Resample<N, Block>> {
  constexpr evaluator(const Resample<N, Block>& resample) noexcept
  : EvaluatorBase<Resample<N, Block>>(resample)
  {}

  constexpr Frame<outs<Block>> eval(Frame<ins<Block>> in)
  {
    auto& e = std::get<0>(this->operands);
    for (float& f : in) {
      f *= N;
    }
    Frame<outs<Block>> res = e.eval(in);
```

```cpp
    for (int i = 1; i < N; i++) {
      e.eval({});
    }
    return res;
  }
};
```

- *Line 10-12*: The gain is increased to compensate for the loss during interpolation (see subsection 6.2.1)
- *Line 13*: The block is evaluated on the input data.
- *Line 14-16*: The block is evaluated $N-1$ times with all-zero inputs.
- *Line 17*: The result of the first evaluation is returned.

Since the block here already contains the interpolation/decimation filters, these operations constitute oversampling.

6.4.1 FIR filters in EDA

While Finite Impulse Response filters (as introduced in subsection 6.2.2) are used for many other purposes than rate conversions, the implementation is included in this chapter for context. Given filter kernel (b_0, b_n), the block $\text{FIR}(b_0, \dots, b_n) : 1 \to 1$ is a primitive block implemented as such:

```cpp
template<std::size_t N>
struct FIRFilter : BlockBase<FIRFilter<N>, 1, 1> {
  std::array<float, N> kernel;
};

template<std::size_t N>
constexpr auto fir(std::array<float, N> kernel) noexcept
{
  return FIRFilter<N>{.kernel = kernel};
}
```

I will not go into detail on the implementation, but it is shown here mainly to illustrate that except for the EDA-speciffic structure of the classes, the filter is implemented in normal C++, similar to how a point-wise evaluated FIR filter would usually be implemented, with a `std::array` used as a ringbuffer, the kernel stored in two consecutive copies, and the meat of the evaluation being done with `std::inner_product`[1]:

```cpp
template<std::size_t N>
struct evaluator<FIRFilter<N>> : EvaluatorBase<FIRFilter<N>> {
  constexpr evaluator(const FIRFilter<N>& fir) noexcept
  {
    std::ranges::copy(fir.kernel, kernel.begin());
    std::ranges::copy(fir.kernel, kernel.begin() + N);
  }
  constexpr Frame<1> eval(Frame<1> in)
  {
    if (t == N) t = 0;
    t++;
    z[N - t] = in;
    auto start = kernel.begin() + t;
    return std::inner_product(start, start + N, z.begin(), 0.f);
  }
private:
  std::size_t t = 0;
  std::array<float, N> z = {0};
```

[1] see https://en.cppreference.com/w/cpp/algorithm/inner_product

```
  std::array<float, 2 * N> kernel;
};
```

This implementation shows that it is possible to implement DSP algorithms in "normal" C++, which allows for hand-written optimizations and complex state, and still have them completely transparently available as a block in an EDA context.

6.4.2 Example

To demonstrate oversampling in EDA, the $\tanh(5x)$ example used throughout this chapter is implemented in EDA as such:

```
auto quarterpass = fir(qp_coefficients);
auto tanh = fun<1, 1>(&std::ftanh);
auto saturation = resample<4>(_ * 5 | tanh, quarterpass, quarterpass);
```

In fact, LV2 plugins constructed in a manner similar to the echo plugin described in section 5.3 containing variations on this implementation is what was used to generate Figure 6.2 and Figure 6.1.

The coefficients for the filter can be seen in ►**appendix ??**◄.

6.5 Conclusion

This chapter has covered the motivation and methods behind multirate DSP, with a focus on oversampling. By implementing this feature in the algebra of blocks, and then in the EDA library in roughly 30 lines of code, I have given an example of the merits of a domain specific library in a general-purpose language compared to a domain specific language, and specifically the versatility allowed by the design choices made in this project. To help support this argument, an implementation of a FIR filter was shown, illustrating how complicated primitive blocks can be implemented in vanilla C++ by the user of the library, instead of having to rewrite the problem in terms of recursion and other basic compositional blocks.

In section 6.3, two possible solutions for multirate DSP were introduced, and while the second was selected here, the first also has potential, though the implementation would be more involved. It does however enable operations that are not supported by the chosen design, by allowing blocks to output at a different sample rate than their input. Potential further work in this area includes exploring possible implementations of that design, and especially how it would integrate with work on vectorization and buffer-wise evaluation. *This chapter*

contains 13280 characters and approximately 2711 spaces = 5.53 standard pages

Chapter 7

Conclusion & Further Work

Building on existing research, this thesis has explored an alternative way to write DSP programs modelled as block diagrams. I have built an embedded DSL in a C++ library by applying existing techniques from other domains to a this domain, and used that library to write an audio plugin that can be used in audio editors. Furthermore I have shown the advantages of working inside a general purpose language, and specifically how constrained C++ templates can provide a framework where parts of the algorithms can be written in conventional C++, using the proven constructs and structures of the language in low-latency situations. I have also explored multirate DSP as an example of an easy extension to the library to solve a specific signal processing problem that requires more specialized design to implement in a DSL compiler.

While the testing of the library performed so far is minimal, it gives an indication of what is possible using the ideas presented. Even in its current unoptimized state, the library only has a performance overhead of around 1.5x compared to handwritten code. While this number is based on only a single example, it is an example that covers most areas of the library, and should give an indication of the magnitude. I have also presented some ideas on how to improve the performance, such as an evaluator that uses vector instructions and optimizing the AST before evaluation. In both of these areas a lot of prior research exists, that could be applied to this project.

Further enhancements could also be made to the syntactical aspects provided by the library, such as naming signals as variables, and declaring blocks with a function syntax. However, the current syntax has been shown to be useful already, and provide a good mapping from blocks diagrams of signal processors to code.

Lastly, as described in chapter 6, multirate DSP is a complex problem that can be highly optimized. The current implementation shows the basics of how it could be implemented, but for real use it should be improved with better filters, as well as expanded to support undersampling in addition to oversampling.

All in all, I believe that this project has shown the idea of using an EDSL built on block algebra to be viable, as well as providing the basis of how such a library could be implemented.

This chapter contains 1912 characters and approximately 394 spaces = 0.80 standard pages

Bibliography

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

[2] 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.

[3] Stéphane Letz, Yann Orlarey, and Dominique Fober. “FAUST Domain Specific Audio DSP Language Compiled to WebAssembly”. In: *The Web Conference*. Lyon, France: International World Wide Web Conferences Steering Committee, 2018, pp. 701–709. DOI: 10.1145/3184558.3185970. URL: https://hal.archives-ouvertes.fr/hal-02158925.

[4] Michel Buffa et al. “Towards an open Web Audio plug-in standard”. In: *WWW2018 - TheWebConf 2018 : The Web Conference, 27th International World Wide Web Conference*. Lyon, France, Mar. 2018. DOI: 10.1145/3184558.3188737. URL: https://hal.univ-cotedazur.fr/hal-01721483.

[5] Smilen Dimitrov, Romain Michon, and Stefania Serafin. “Blender2faust: from drawn 3d objects to physically based sound models”. In: *Sound and Music Computing Conference*. Limassol, Cyprus, 2018. URL: https://hal.archives-ouvertes.fr/hal-02158954.

[6] Edgar Berdahl, Peter Vasil, and Andrew Pfalz. “Automatic Visualization and Graphical Editing of Virtual Modeling Networks for the Open-Source Synth-A-Modeler Compiler”. In: *International Conference on Human Haptic Sensing and Touch Enabled Computer Applications*. London, United Kingdom, 2016, pp. 490–500. DOI: 10.1007/978-3-319-42324-1_48. URL: https://hal.archives-ouvertes.fr/hal-03162897.

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

[8] 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.

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

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

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

[12] 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.

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

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

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

[16] Saar Raz. *Clang Concepts*. Talk given at CoreCPP. 2019. URL: https://corecppil.github.io/CoreCpp2019/Presentations/Saar_clang_concepts.pdf.
[17] 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.
[18] 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.
[19] 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.
[20] 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.
[21] Will C Pirkle. *Designing Audio Effect Plugins in C++: For AAX, AU, and VST3 with DSP Theory*. Routledge, 2019.
[22] 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).
[23] 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.
[24] Zhanyong Wan and Paul Hudak. "Functional reactive programming from first principles". In: *Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation*. 2000, pp. 242–252.
[25] Romain Picard. *Hands-On Reactive Programming with Python*. Packt, 2018. ISBN: 9781789138726.
[26] Bruno Bachelet and Loïc Yon. "Designing Expression Templates with Concepts". In: *Software: Practice and Experience* 47.11 (Mar. 2017), pp. 1521–1537. DOI: 10.1002/spe.2483. URL: https://hal.archives-ouvertes.fr/hal-01351060.
[27] URL: https://en.cppreference.com/w/cpp/language/constraints.
[28] Nicolas Scaringella et al. "Automatic vectorization in Faust". In: *Journées d'Informatique Musicale*. Ed. by JIM. Montbeliard, France, 2003. URL: https://hal.archives-ouvertes.fr/hal-02158949.
[29] 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.
[30] Bonnie C. Baker. "Anti-Aliasing, Analog Filters for Data Acquisition Systems". In: (1999).
[31] 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.
[32] 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.
[33] 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.