FAUST Tutorial for Functional Programmers

Yann Orlarey
GRAME, France
orlarey@grame.fr

Stéphane Letz
GRAME, France
letz@grame.fr

Dominique Fober
GRAME, France
fober@grame.fr

Romain Michon
CCRMA, USA
rmichon@ccrma.stanford.edu

Abstract

This paper is an introduction to FAUST, a functional programming language for sound synthesis and audio processing. We assume that the reader has some familiarity with functional programming, but no previous knowledge in signal processing. The text describes several examples that the reader will be able to try online using a web browser. These examples are preceded by two more technical sections presenting some aspects of the language.

CCS Concepts • **Applied computing → Sound and music computing**; • **Software and its engineering → Functional languages**; **Data flow languages**; **Compilers**; **Domain specific languages**;

Keywords Signal processing, Faust

ACM Reference Format:
Yann Orlarey, Stéphane Letz, Dominique Fober, and Romain Michon. 2017. FAUST Tutorial for Functional Programmers. In *Proceedings of 5th ACM SIGPLAN International Workshop on Functional Art, Music, Modeling and Design, Oxford, UK, September 9, 2017 (FARM'17)*, 8 pages.
https://doi.org/10.1145/3122938.3122941

1 Introduction

FAUST is a functional programming language for sound synthesis and audio processing with a strong focus on the design of synthesizers, musical instruments, audio effects, etc. Its main sources of inspiration are *lambda-calculus*, *combinatory logic*, John Backus' *FP*, and Stefanescu's *Algebra of Flownomials*.

FAUST is used on stage for concerts and artistic productions, in education and research, in open-source projects as well as in commercial applications. Typical users are musicians, sound engineers, researchers, musical assistants, etc. They often have a background in signal processing or at least a clear idea of how audio effects and sound synthesis systems should work or sound. But users are not necessarily computer scientists or professional developers. The development of real-time audio software in C is usually out of reach for most of them. The ambition of FAUST is to offer them a viable and efficient high-level alternative. The FAUST compiler can generate optimized code for a variety of languages: C++, C, Java, Javascript, asm.js, LLVM, and WebAssembly.

FARM'17, September 9, 2017, Oxford, UK

ACM ISBN 978-1-4503-5180-5/17/09…$15.00
https://doi.org/10.1145/3122938.3122941

The generated code is then combined with an *architecture file* describing how to relate the audio computation with the external world. Thanks to this approach, the exact same source code can be used to generate native applications and plugins for more than 20 different targets, from VST and Unity plugins to Android applications, from embedded systems to Web Audio applications.

As we will see through various examples, programming in FAUST is essentially combining *signal processors* using a set of binary operations that form the Block-Diagram Algebra. The functional programming approach is particularly suited for this purpose. *Signals* are discrete-time functions. *Signal processors* are second-order functions operating on *signals*. The *block-diagram algebra* used to combine signal processors together is a set of third-order composition operations on *signal processors*. Finally, user-defined functions are higher-order functions on block-diagram expressions. Powerful means, like pattern matching, are available to algorithmically generate complex audio circuits.

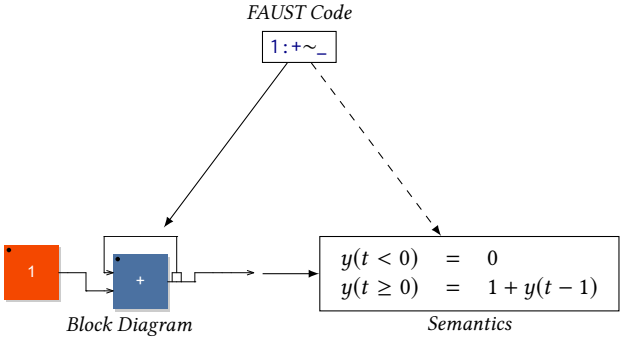


Figure 1. FAUST programs have a straightforward visual representation as block-diagrams, as well as a simple and well defined formal semantics. The block-diagram is a useful intermediate step to compute the semantics of a program.

FAUST is a textual language, but programs have straightforward translations into visual block-diagrams as well as mathematical descriptions. The relation between the FAUST code, and its translations is represented Figure 1. The FAUST compiler is able to produce automatically these diagrams and the mathematical semantics of a program. This feature is used in particular for preservation purposes, an important concern for music pieces relying on real time programs. But these features are also very useful when learning FAUST to better understand the meaning of expressions and programs.

2 Definitions

Before we dive into the examples, let's start with the definition of basic concepts like *signals* and *signal processors* that we will use extensively:

Definition 2.1. A *signal* $s \in \mathbb{S}$ is a discrete-time function. FAUST considers signals of two types, floating point signals and integer signals: $\mathbb{S} \subset (\mathbb{Z} \to \mathbb{R}) \cup (\mathbb{Z} \to \mathbb{Z})$. The values of a signal at specific time-points are called *samples*. The full scale audio range for samples is typically $[-1.0, +1.0]$. The actual computation time starts at $t = 0$, but to take into account *delay* operations signals are extended toward $-\infty$ with 0 values. In other words $\forall s \in \mathbb{S}, s(t < 0) = 0$.

Definition 2.2. A FAUST program denotes a *signal processor* $p \in \mathbb{P} \subset \mathbb{S}^n \to \mathbb{S}^m$, a function on tuple of signals. Every primitive of the language is a *signal processor*, including numbers, and user interface elements. For instance `3.14` or `button("play")` are both signal processors of type $\mathbb{S}^0 \to \mathbb{S}^1$.

All numerical operations in C have an equivalent on signals in FAUST. For example, the FAUST primitive `+` of type $\mathbb{S}^2 \to \mathbb{S}^1$ is defined as $\lambda(s_1, s_2).(\lambda t.s_1(t) + s_2(t))$. The int to float promotion rules are those of C. If one of two input signals of `+` is a floating point signal, the result will be an floating point signal.

Definition 2.3. Beside numerical operations there are some *specific primitives*, in particular:

`_` : $\mathbb{S}^1 \to \mathbb{S}^1$ the identity function $\lambda(s).(s)$,
`!` : $\mathbb{S}^1 \to \mathbb{S}^0$ the *cut* function $\lambda(s).()$ that returns the empty tuple,
`@` : $\mathbb{S}^2 \to \mathbb{S}^1$ the *delay* function $\lambda(s_1, s_2).(\lambda t.s_1(t - s_2(t)))$. The delay operation supposes that s_2 is positive and bounded: $\exists l \in \mathbb{N}, \forall t \in \mathbb{Z}, 0 \le s_2(t) \le l$

as well as user interface elements like buttons and sliders.

For example the expression `hslider("level", 0.5, 0, 1, 0.01)` : $\mathbb{S}^0 \to \mathbb{S}^1$ represents a horizontal slider named `"level"` that produces a signal according to the actions of the user. The default value (when the program starts) of the produced signal is `0.5`, the minimal value is `0`, the maximal value `1`, and the step value `0.001`.

Definition 2.4. Programming in FAUST is essentially composing signal processors together using an algebra of five *composition operations* $c \in \mathbb{C} = \{$ `<: :> : , ~` $\}$. Each operator takes two signal processors and connects them to form a new one: $\mathbb{C} \subset \mathbb{P} \times \mathbb{P} \to \mathbb{P}$.

Definition 2.5. Beside the core syntax based on the above composition operations, *syntactic sugar* exists for infix expressions: `A+B` is equivalent to `A,B:+`, and for partial applications of primitives: `/(B)` is equivalent to `_,B:/`.

3 The Block-Diagram Algebra

The Block-Diagram Algebra [6], a set of five *composition operations* used to combine signal processors to form more complex ones (see Table 1). For example, the expression `2,3 <: +,*` (see Figure 2) uses two parallel compositions (`,`) and a split composition (`<:`) that distributes the two outputs of `2,3` to the four inputs of `+,*`. An equivalent expression would be notated `[+,*]:<2,3>` in FP and `\x.\y.(x+y,x*y)2 3` in λ-calculus.

To be valid, most of these composition operations have restrictions on the number of inputs and outputs of the signal processors they combine. For example `A:B` is only valid if the number of outputs of `A` is equal to the number of inputs of `B`. If this is not the case, the compiler will trigger a type error.

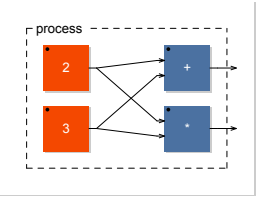


Figure 2. block-diagram of `2,3 <: +,*`.

Table 1. The five composition operations.

`A <: B`	split composition	priority 1
`A :> B`	merge composition	priority 1
`A : B`	sequential composition	priority 2
`A , B`	parallel composition	priority 3
`A ~ B`	recursive composition	priority 4

3.1 Split Composition

The *split composition* `A<:B` operator is used to distribute the outputs of `A` to the inputs of `B` (see Figure 3). To be valid, the number of inputs of `B` must be a multiple of the number of outputs of `A`.

$$\frac{\texttt{A} : \mathbb{S}^n \to \mathbb{S}^m \qquad \texttt{B} : \mathbb{S}^{k.m} \to \mathbb{S}^p}{\texttt{A<:B} : \mathbb{S}^n \to \mathbb{S}^p}$$

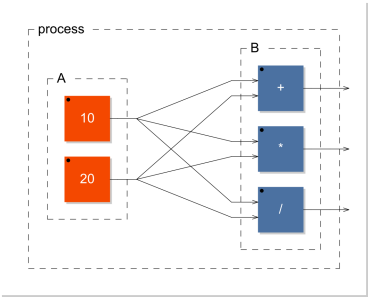


Figure 3. split composition: `(2,3) <: (+,*,/)`.

3.2 Merge Composition

The *merge composition* `A:>B` is used to connect several outputs of `A` to the same inputs of `B` (see Figure 4). To be valid, the number of outputs of `A` must be a multiple of the number of inputs of `B`. Signals connected to the same input are added together at sample level.

$$\frac{\texttt{A} : \mathbb{S}^n \to \mathbb{S}^{k.m} \qquad \texttt{B} : \mathbb{S}^m \to \mathbb{S}^p}{\texttt{A:>B} : \mathbb{S}^n \to \mathbb{S}^p}$$

3.3 Sequential Composition

The *sequential composition* `(A:B)` connects the outputs of `A` to the corresponding inputs of `B` (see Figure 5). The number of outputs of `A` must be equal to the number of inputs of `B`.

$$\frac{\texttt{A} : \mathbb{S}^n \to \mathbb{S}^m \qquad \texttt{B} : \mathbb{S}^m \to \mathbb{S}^p}{\texttt{A,B} : \mathbb{S}^n \to \mathbb{S}^p}$$

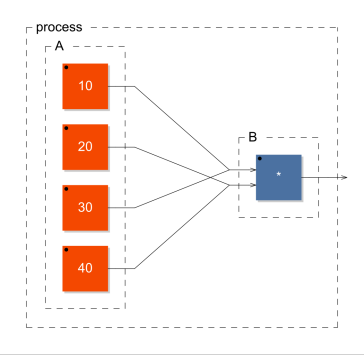


Figure 4. merge composition: `((10,20,30,40):> *)`.

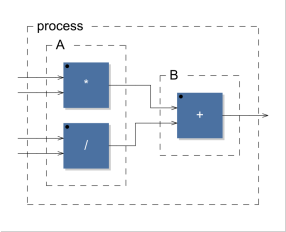


Figure 5. sequential composition: `((*,/):+)`.

3.4 Parallel Composition

The *parallel composition* `(A,B)` is probably the simplest one. It places the two block-diagrams one on top of the other, without connections (see Figure 6). The inputs (resp. outputs) of `(A,B)` are the inputs (resp. outputs) of `A` then the inputs (resp. outputs) of `B`.

$$\frac{\texttt{A} : \mathbb{S}^n \to \mathbb{S}^m \qquad \texttt{B} : \mathbb{S}^p \to \mathbb{S}^q}{\texttt{A,B} : \mathbb{S}^{n+p} \to \mathbb{S}^{m+q}}$$

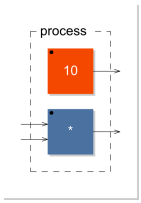


Figure 6. parallel composition: `(*,/)`.

3.5 Recursive Composition

The *recursive composition* `(A~B)` is used to create cycles in the block-diagram in order to express recursive computations (see Figure 7). It is the most complex operation. The number of inputs of `B` must be less or equal to the number of outputs of `A` and the number of outputs of `B` must be less or equal to the number of inputs of `A`. Note that all the inputs of `B` are connected to the corresponding output of `A` via an implicit one-sample delay represented by a small white square in the picture. Please also notice the following asymmetry: the outputs of `(A~B)` are the outputs of `A`, but the inputs of `(A~B)` are the remaining inputs of `A`, those that are not connected to an output of `B`.

$$\frac{\texttt{A} : \mathbb{S}^{q+n} \to \mathbb{S}^{p+m} \qquad \texttt{B} : \mathbb{S}^p \to \mathbb{S}^q}{\texttt{A} \sim \texttt{B} : \mathbb{S}^n \to \mathbb{S}^{p+m}}$$

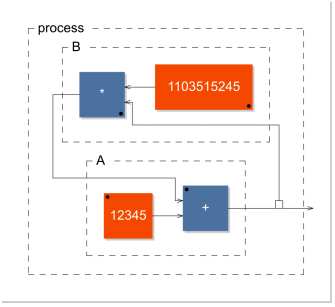


Figure 7. recursive composition: `+(12345)~ *(1103515245)`.

4 White Noise Generator

Let's start with a simple example: a *white noise generator*. A white noise is a signal with a flat spectrum that contains all audible frequencies. It has several interesting applications in sound synthesis, for example substractive synthesis as we will see in the wind simulator of section 6.

The principle is fairly simple. It consists in randomly choosing each sample in the range $[-1.0, 1.0]$. The corresponding FAUST code is the following:

```
process = +(12345)~*(1103515245)
    :/(2147483647.0)
    :*(vslider("vol",0,0,1,0.1));
```

Listing 1. White Noise Generator.

The first part `+(12345)~*(1103515245)` is a *linear congruential* pseudo random number generator. It generates values in the range $[-2^{31}, 2^{31}-1]$. The `~` operator creates a feedback loop in the circuit by connecting the output of `+(12345)` to the input of `*(1103515245)` (via a one sample delay), and the output of `*(1103515245)` to the input of `+(12345)`.

The second part `/(2147483647.0)` scales down the pseudo random signal into the audio range $[-1.0, 1.0]$. And finally a user interface element is used to control the output volume `*(vslider("vol",0,0,1,0.1))`.

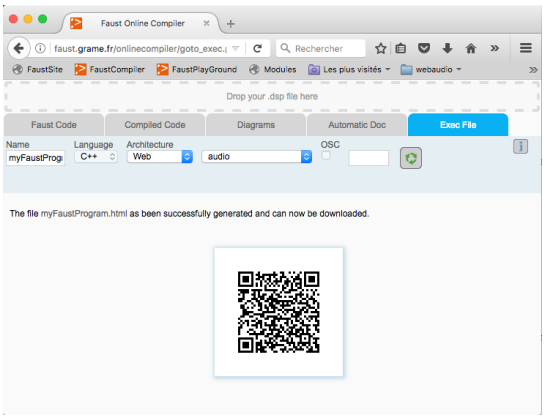
The `:` operator is used to sequentially compose the three parts of the expression.

4.1 Running the Code

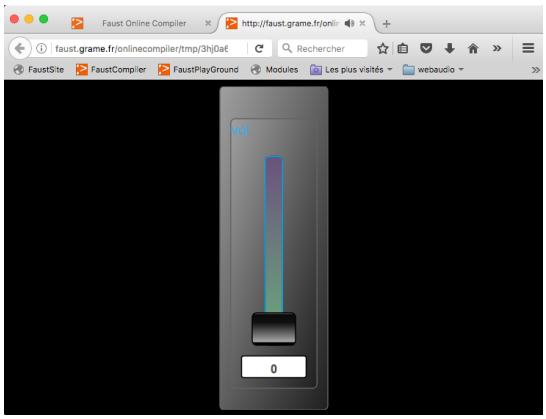
Let's first see how to run this example using the *Online Compiler*:

- open http://faust.grame.fr/onlinecompiler the *FAUST Online Compiler*,
- enter the above definition in the editor,[1]
- click on the "Compiled Code" tab to check that the code has no typo and compiles,
- click on the "Exec File" tab, then choose architecture "web/audio" to create a WebAudio application (Figure 8a),
- click on the resulting link to run it (Figure 8b).

[1]Please note that if you copy and paste the example from the PDF file, the tilde (~) character may not be copied correctly



(a) Select the Web/audio target and click on the generated link



(b) The user interface of the noise generator

Figure 8. Compiling and running the noise example as a WebAudio application.

4.2 Visualizing the Block-Diagram

FAUST programs have a straightforward *block-diagram* representation. At the beginning it is often useful to refer to this graphical representation to understand a program. It can be obtained using the "Diagram" tab (Figure 9).

4.3 Visualizing the Semantics

The semantics of a FAUST program can also be computed using "Documentation" tab of the Online Compiler. This will produce a PDF file containing a full mathematical description of the program. Here is an extract of the documentation generated for our noise example:

> **1 Mathematical definition of process**
>
> The *myFaustProgram* program evaluates the signal transformer denoted by process, which is mathematically defined as follows:
>
> 1. Output signal y such that
>
> $$y(t) = p_1(t) \cdot r_1(t)$$
>
> 2. Input signal (none)
> 3. User-interface input signal u_{s1} such that
>
> $$\text{"vol"} \quad u_{s1}(t) \in [0,1] \quad \text{(default value = 0)}$$
>
> 4. Intermediate signals p_1 and r_1 such that
>
> $$p_1(t) = 4.6566128752458 \cdot 10^{-10} \cdot u_{s1}(t)$$
>
> $$r_1(t) = 1103515245 \odot r_1(t-1) \oplus 12345$$

This automatic documentation system [1] is built into the FAUST compiler. It was designed to partially address the preservation problem faced by music pieces heavily relying on computer programs to be performed. From experience we know that preserving the source code of the involved programs is not enough. The idea is to preserve also the mathematical semantics of the program, independently of the programming language itself. It turned out that this system is also useful to understand FAUST programs when learning the language.

4.4 Reorganizing the Code

For more readability our code can be split into several definitions. Note that the order of definitions is not important, but redefinitions are not allowed:

```
process = noise : *(vslider("vol",0,0,1,0.1));
noise   = random : /(2147483647.0);
random = +(12345)~*(1103515245);
```

Listing 2. White noise generator with separate definitions for noise and random.

Definitions local to an expression can be declared using the `with {}` construction:

```
process = noise : *(vslider("vol",0,0,1,0.1))
  with {
    noise = random : /(2147483647.0);
    random = +(12345)~*(1103515245);
  };
```

Listing 3. White noise generator with local definitions.

If you compile these two versions, you will see that the generated code doesn't change compared to the one-line example. Two different FAUST programs, but with the same mathematical meaning, should ideally result in the same implementation[2].

5 Oscillators

In this section we will create some typical waveform oscillators. We will start with a phase generator that delivers a periodic sawtooth signal between 0 and 1. Then we will implement a sine wave and a square wave oscillator.

5.1 Phase Generator

A phase generator produces a periodic sawtooth signal that goes from 0 to 1 at each period. For example the expression: `0.1 : (+,1.0:fmod)~_` , corresponding to Figure 10, will produce the periodic signal :

```
0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 0.0...
```

[2] Although this is equivalent to the Halting problem and undecidable in general.

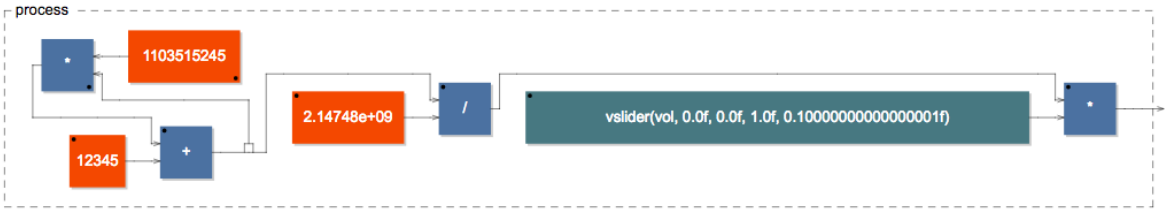


Figure 9. The "Diagram" tab allows to visualize the graphical representation of a program.

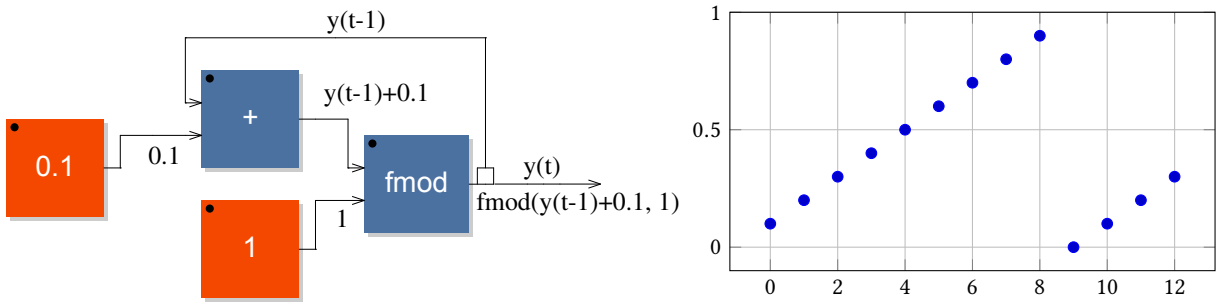


Figure 10. A phase generator producing a signal $y(t) = \text{fmod}(y(t-1) + 0.1, 1)$.

In this expression `0.1` is the increment between successive samples, and `fmod` is the floating point remainder operation used to wrap the signal in the interval [0, 1[.

By controlling the increment we can control the frequency of the generated signal. Let's say that the sampling rate is `44100` samples per second. By using an increment of `1/44100` we will produce a signal at $1Hz$, and by using an increment of `f/44100` we will produce it at frequency fHz. We can therefore define our phase generator as:

```
phasor(f) = f/44100 : (+,1.0:fmod) ~ _ ;
```

Listing 4. Phase generator.

5.2 Sine Wave Oscillator

Once we have a phase generator, it is easy to define a sine wave oscillator by multiplying the phase signal by 2π and taking the *sine* of the result:

```
osc(f) = phasor(f) * 6.28318530718 : sin;
```

Listing 5. Sine wave oscillator.

We can complete our program with sliders to control the frequency and the level of the oscillator:

```
phasor(f) = f/44100 : (+,1.0:fmod) ~ _ ;
osc(f) = phasor(f) * 6.28318530718 : sin;

process = osc(hslider("freq", 440, 20, 20000,1))
        : *(hslider("level", 0, 0, 1, 0.01));
```

Listing 6. Sine wave oscillator, full implementation with frequency and volume control.

You can try it by pasting the above code into the online compiler and selecting the web/audio target in the exec code tab will result in the web page Figure 11.

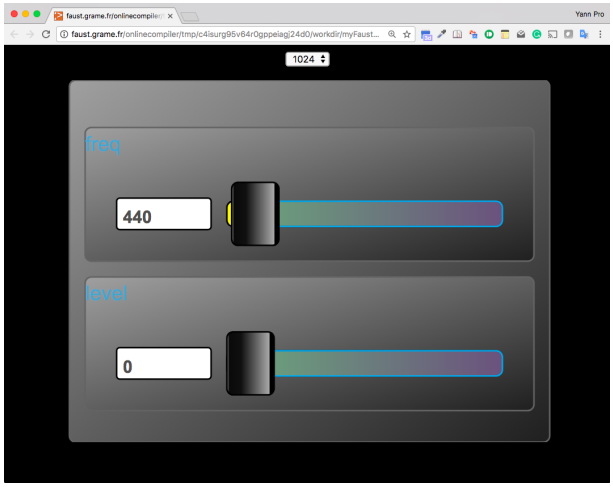


Figure 11. The sine wave oscillator running on the web.

5.3 Square Wave Oscillator

A square wave signal between 0 and 1 can be easily obtained by comparing the output signal of the phase generator with 0.5. This will produce a signal that is 1 when the condition is true and 0 when it is false. If we then multiply the resulting signal by 2 and subtract 1 we have a square wave between -1 and 1 (see Listing 5).

Please note that this is not a very good square wave generator to listen to because of aliasing [5] problems. A better, band limited

implementation, called `square` is available in the "oscillators.lib" library.

```
phasor(f) = f/44100 : (+,1.0:fmod) ~ _ ;
squarewave(f) = phasor(f) > 0.5 : *(2) : -(1);
process = squarewave(hslider("freq", 440, 20,
    20000, 1))
: *(hslider("level", 0, 0, 1, 0.01));
```

Listing 7. Square wave oscillator, full implementation with frequency and volume control.

6 Wind Simulator

In this example we will create a wind simulator based on a filtered white noise. The principle is to correlate the strength of the wind to the resonance and the frequency of a filter applied to a noise generator.

6.1 Monophonic Wind Simulator

Let's start with a very simple monophonic wind simulator.

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

wind(strength) = noise : moog_vcf_2bn(strength,
    freq) : *(strength)
with {
  freq = pianokey2hz(strength*87+1);
};

process = wind( hslider("v:wind/strength"
    ,0.66,0,1,0.01) : smooth(0.99997) );
```

Listing 8. Monophonic wind simulator.

The first line imports the standard libraries that define `noise`, `moog_vcf_2bn`, `piano2hz` and `smooth`. Then we have the definition of the wind generator with a strength parameter between 0 and 1. This wind generator is essentially a noise generator connected to a resonant moog filter. The filter has two parameters: the amount of corner-resonance between 0 and 1, and the corner-resonance frequency in Hz. By correlating the strength of the wind with the resonance and frequency of the filter, we reproduce the typical whistling of the wind.

We use an horizontal slider to control the strength of the wind. The values delivered by the slider are filtered for a more realistic wind control.

6.2 Stereo Wind Simulator

Now, we would like to create a stereophonic wind generator, with two slightly different wind sounds on each channel. It is not enough to put in parallel the wind expression twice because the stream of random numbers (and therefore the sound) will be exactly the same. We have to decorrelate the two channels. A simple way to do that is to delay one of the two channels, but that is not completely satisfactory. Instead we are going to create independent white noise generators and for that purpose we need a multichannel random number generator.

6.2.1 Rearranging the Random Number Generator

We will start by rearranging our previous random number generator. First observe that for any two FAUST expressions `A`: $\mathbb{S}^1 \rightarrow \mathbb{S}^m$ and `B`: $\mathbb{S}^m \rightarrow \mathbb{S}^n$, then `(A:B)~_` and `B~A` are equivalent expressions (of type $\mathbb{S}^0 \rightarrow \mathbb{S}^n$), the difference is just "topological" so to speak. In both cases the output signals generated are identical: $\vec{y}(t) = B(A(\vec{y}(t-1)))$ and $\vec{y}(t) = B(A(\mathrm{Id}(\vec{y}(t-1))))$, where Id is the identity function.

Therefore `+(12345)~*(1103515245)` can be replaced by the equivalent `(*(1103515245):+(12345))~_`:

```
random = scramble ~ _;
scramble = *(1103515245):+(12345);
```

Listing 9. Random number generator rearranged.

It is easy now to create a 2-channels or a 3-channels random number generator:

```
random2 = (scramble <: scramble, _) ~ _;
random3 = (scramble <: (scramble <: scramble, _)
    , _) ~ _;
```

Listing 10. 2-channels and 3-channels random number generators.

6.2.2 Multichannel Random Number Generator

The above recursive structure can be generalized to an arbitrary number of channels using pattern matching and term rewriting [2].

```
polyrandom(N) = scramble(N) ~ _
  with {
    scramble (1) = *(1103515245):+(12345);
    scramble (n) = scramble(1) <: scramble(n-1),
    _;
};
```

Listing 11. Multiple random signals in parallel.

Please note that here N must be constant and known at compile time to be used for pattern matching. The result of `process = polyrandom(8);` is illustrated Figure 12.

6.2.3 Multiple Noises

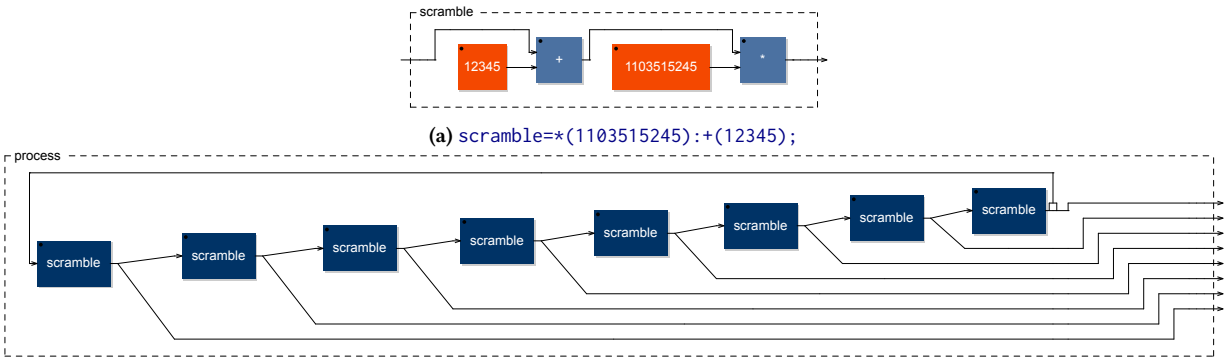
We can now define our multi noise generator by scaling down the random signals to the audio range.

```
polynoise(N) = polyrandom(N) : par(i,N
    ,/(2147483647.0));
```

Listing 12. Multiple white noises in parallel.

Note the `par(...)` construction: the expression `par(i,N,E(i))` is equivalent to `E(0), E(1),...,E(N-1)`. Three other analogue constructions exist: `seq(i,N,E(i))` equivalent to `E(0): E(1):...:E(N-1)`, `sum(i,N,E(i))` equivalent to `E(0)+E(1)+...+E(N-1)`, and `prod(i,N,E(i))` equivalent to `E(0)* E(1)*...*E(N-1)`.

Combined with pattern matching these constructions allow to algorithmically describe complex circuits. A nice example of such algorithmic description is the FFT implemented by Julius Smith in the *analyzers.lib* library of FAUST (see Figure 13).



(a) `scramble=*(1103515245):+(12345);`

(b) `process = polyrandom(8);`

Figure 12. Multirandom generator.

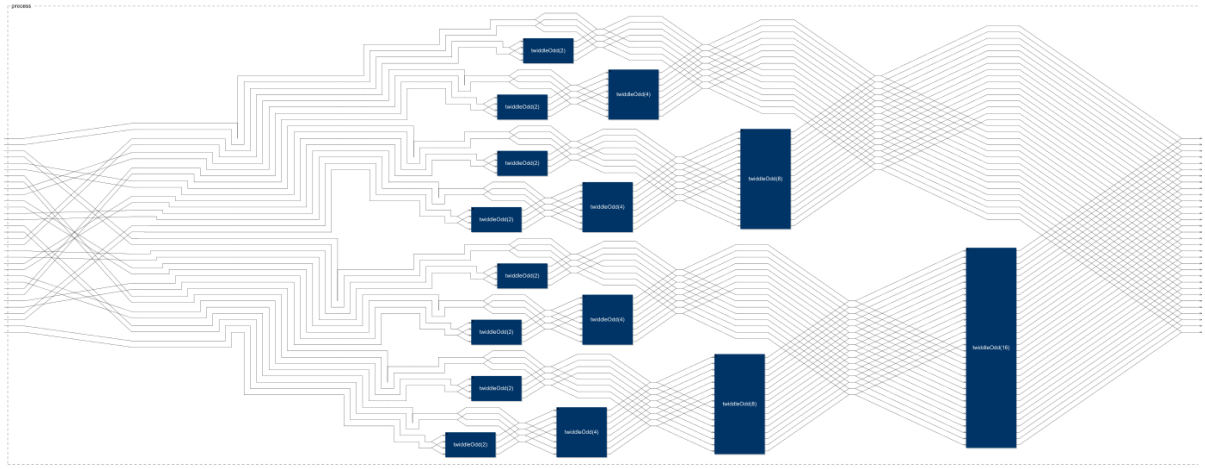


Figure 13. The FFT circuit, an example of complex algorithmic description.

6.2.4 Multiple Winds

We now have all the pieces needed to implement our multichannels wind generator:

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

polyrandom(N) = randomize(N) ~ _
  with {
    scramble = *(1103515245):+(12345);
    randomize(1) = scramble;
    randomize(n) = scramble <: randomize(n-1),_;
  };

polynoise(N) = polyrandom(N)
             : par(i,N,/(2147483647.0));

polywind(N, strength) = polynoise(N):par(i,N,
    moog_vcf_2b(strength,freq):*(strength))
  with {
    freq = pianokey2hz(strength*87+1);
  };
process = polywind(2, hslider("v:wind/strength"
    ,0.66,0,1,0.01) : smooth(0.99997) );
```

Listing 13. Full implementation of a stereo wind generator.

You can test this example using the online compiler following the same procedure as for the noise generator.

6.2.5 A Wind Generator for Android

The Online Compiler can generate binaries for many platforms, including Android smartphones. We could use the above code unmodified to create a native Android .apk. But instead of relying on a slider to control the application we would like to use the x-axis accelerometer. This can be indicated by adding some specific metadata : `"...[acc:0 3 -10 0 10]..."` to the label of the slider.

```
process = polywind(2, hslider("v:wind/strength[
    acc:0 3 -10 0 10]",0.66,0,1,0.01) : smooth
    (0.99997) );
```

Listing 14. Accelerometer metadata.

These metadata are interpreted by the Android and iOS architectures as indicating how to convert the accelerometers values into slider values. The first one: 0 indicates the x-axis. The second one: 3 is the V-shape mapping curve and the last three the mappings values in m/s^2.

Then follow the same procedure with the Online Compiler but in the Exec Code tab, choose the target Android/android. The generation of the Android application is quite long and takes nearly 2 mn. Once the compilation is finished you can flash the QR code from your smartphone to install the application.

7 Further Readings

This tutorial is far from being self-contained and many details were skipped. Nevertheless, we hope that the reader has been able to grasp the spirit of the language. For those who want to go further, an overview of the language is available here [7]. The *Faust Quick Reference* is also an useful reading [3] as well as the *Faust Standard Libraries* documentation [4].

References

[1] Karim Barkati, Dominique Fober, Stéphane Letz, and Yann Orlarey. Two recent extensions to the Faust compiler. In *Proceedings of the Linux Audio Conference*, 2011.
[2] Albert Gräf. Term rewriting extension for the Faust programming language. *signal*, 3:6, 2010.
[3] Grame. Faust Quick Reference.
[4] Grame. Faust Standard Libraries.
[5] Juhan Nam, Vesa Välimäki, Jonathan S Abel, and Julius O Smith. Alias-free virtual analog oscillators using a feedback delay loop. In *Proc. 12th Int. Conf. Digital Audio Effects (DAFx-09)*, pages 1–4, 2009.
[6] Yann Orlarey, Dominique Fober, and Stephane Letz. An algebra for block diagram languages. In ICMA, editor, *Proceedings of International Computer Music Conference*, pages 542–547, 2002.
[7] Yann Orlarey, Dominique Fober, and Stephane Letz. Syntactical and semantical aspects of Faust. *Soft Computing*, 8(9):623–632, 2004.