A FAUST Tutorial

Etienne Gaudrain, Yann Orlarey

A Faust Tutorial

Grame, 9 rue du Garet, 69001 Lyon

by Etienne Gaudrain, Yann Orlarey
September 10, 2003

Special thanks to Samuel Bousard
for the support, the reading and the corrections.

Introduction

Faust is a programming language designed for the creation of stream processors. The name *Faust* stands for **F**unctional **au**dio **st**ream, and suggests both a musical and a devilish (!) side. It has been developed by Grame since 2000 [OFL02], to help programmers, but also composers and various sound hackers to build audio stream processors. The Faust language is based on functional programming paradigms [Hug89], but the Faust compiler produces C++ code that is optimized by a λ-abstraction-based mechanism.

The portability of C++ entails the portability of Faust if one can bring the good wrapper files that will make the C++ code compilable as a standalone application on your system.

This tutorial aim is to present the Faust syntax and grammar, and commented examples.

The Part I begins with a presentation of some theoretical aspects of the language (Chapter 1) that may help to understand the tutorial, and the way to program with Faust. Then, in Chapter 2 are listed the keywords, and in Chapter 3 the composition operators. In Chapter 4 we'll introduce to the structure of a Faust program. This part ends with how to compile a Faust program (Chapter 5).

The Part II gives some commented examples. In Chapter 6, are presented oscillators that would be used as primitive generators in synthesisers applications. And from Chapter 7 to the end of the part, a few examples of filters and complex generators. All examples are presented the same way : first the purpose of the application, then the Faust source code, and eventually the explanation — line by line — of the program.

Conventions

Here are some font conventions used in this tutorial. The Faust source code is written with a Typewriter font :

```
process = hslider("Frequency",1,1,2000,1):sinus;
```

The same font is used for C++ code, except for the keywords that are in bold Roman :

```
for (int i=0; i<count; i++) {
   output0[i] = ((input0[i] * 0.5) + (input1[i] * 0.5));
}
```

The file names and paths are written with a Sans-Serif font :

/src/faust/architecture/alsa-gtk.cpp

Contents

Part I

Language description

Chapter 1

Preview

An algebra for block-diagram languages was proposed [OFL02] to provide descriptions and tools to handle data streams in a functional way. The Faust language is an implementation of this algebra. Therefore, Faust is a real functional programming language, especially designed to process numeric data streams.

This chapter gives a theoretical approach of the Faust language. The introduced notions will be used to link the Faust materials to mathematical descriptions or to block diagrams. Therefore we'll first present the matter of Faust programming : streams and stream processors.

1.1 An Algebra for Block Diagram Languages

Here is a brief summary of the work of the GRAME team [OFL02]. This algebra is the root of the Faust language. It presentation should help to understand the Faust programming philosophy, an in particular, it will highlight the difference between a number and a stream, i.e. a signal. The comprehension of this difference is essential while Faust programming.

Therefore, we define a stream or a signal, as a real-valued function of time. Because the streams we're speaking about are sampled streams, the time is an integer (in fact the number of samples). So, we can define $\mathbb{S}$, the ensemble of the streams, this way :

$$\mathbb{S} = \{ s : \mathbb{N} \to \mathbb{R} \} = \mathbb{R}^{\mathbb{N}}$$

The value of a stream s at the moment t is noted $s(t)$. A special notation can be used for the elements of $\mathbb{S}$:

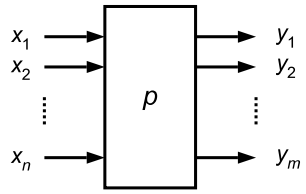
$$\forall\, s \in \mathbb{S},\ s^{-1} \text{ is defined by } : \forall\, t \in \mathbb{N},\ s^{-1}(t+1) = s(t)$$

A stream processor is a function from streams tuples to streams tuples. More precisely, if n and m are integers, one obtains a m-tuple of streams from a n-tuple of streams. The n-tuples are noted with parenthesis. The $\mathbb{S}^0$ ensemble only contains the empty-tuple ( ). $(s_1, s_2, \ldots, s_n) \in \mathbb{S}^n$ is a n-tuple of streams. Therefore, if $\mathbb{P}$ is the ensemble of the stream processors, then $p \in \mathbb{P}$ is equivalent to :

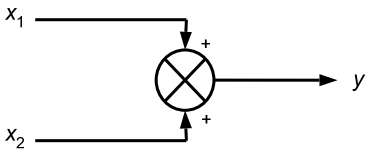
$$\exists (n, m) \in \mathbb{N}^2, p : \mathbb{S}^n \to \mathbb{S}^m$$

Then, p is a function so we should write $p\Big((x_1, x_2, \ldots, x_n)\Big) = (y_1, y_2, \ldots, y_m)$, where $x_1, x_2, \ldots, x_n$ and $y_1, y_2, \ldots, y_m$ are streams. But, to simplify the notation from now, only some single parenthesis will be used :

$$p(x_1, x_2, \ldots, x_n) = (y_1, y_2, \ldots, y_m)$$



For example if p is a stream processor that 'adds' two signals to make a single signal.



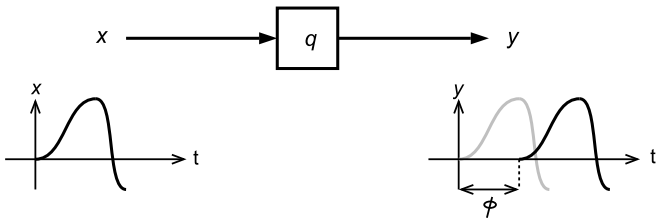
Then $p : \mathbb{S}^2 \to \mathbb{S}^1$ and :

$$\forall (x_1, x_2) \in \mathbb{S}^2,\ p(x_1, x_2) = (y)$$
$$\text{with} \quad y \in \mathbb{S} \setminus \forall\, t \in \mathbb{N},\ y(t) = x_1(t) + x_2(t)$$

With the λ-notation :

$$\forall (x_1, x_2) \in \mathbb{S}^2,\ p(x_1, x_2) = (\lambda t\ .\ x_1(t) + x_2(t))$$

Then one has to deal with a different stream processor, q that adds a phase difference ϕ to a signal.



Then $q : \mathbb{S}^1 \to \mathbb{S}^1$ and :

$$\begin{aligned} &\forall\, x \in \mathbb{S}, \exists\, y \in \mathbb{S},\ q(x) = (y) \\ \text{where} \quad &\forall\, t \in \mathbb{N},\ y(t) = x(t - \phi) \end{aligned}$$

This last example shows the importance of the difference between numbers and streams, and so between real-valued functions and stream processors.

1.2 The semantic function

This function is used to distinguish the programming language notations from their mathematical meaning [Gun92]. It is noted with double brackets $[\![\ ]\!]$ with no suffix for the Faust semantic function, and $[\![\ ]\!]_{\mathrm{C}}$ for the C++ one.

For example, if A and B are some C++ expressions, we can write :

$$[\![\ \texttt{+(}A\texttt{,}B\texttt{)}\ ]\!]_{\mathrm{C}} = [\![\ A\texttt{+}B\ ]\!]_{\mathrm{C}} = [\![\ A\ ]\!]_{\mathrm{C}} + [\![\ B\ ]\!]_{\mathrm{C}}$$

which just means that the addition in C++ can be noted as a function `+(,)`, or with the normal math notation, and that the math signification is exactly the normal addition.

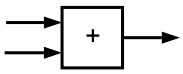
Thus, as a Faust expression, i.e. written in the Faust language, is a signal processor :

$$[\![\ \textit{Faust expression}\ ]\!] \in \mathbb{P}$$

Let's take the two examples shown in the previous section. The first one, which sums two signals, has a direct Faust implementation noted `+`. Thus we have the following relation :

$$[\![\ \texttt{+}\ ]\!] = p$$

where p is the same signal processor as before. It can be represented by this block diagram :



The second example is slightly more complex unless the delay duration is only of one sample. Then, the signal processor q will be written `mem` in a Faust program :

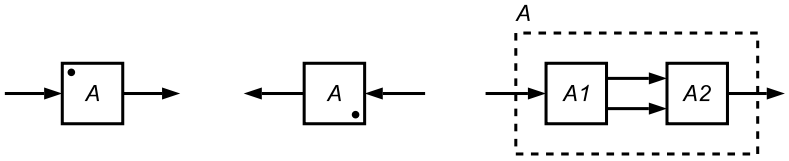
$$[\![\texttt{mem}]\!] = q$$

Thus a Faust expression can always be associated to a signal processor that can be described by some mathematical expressions. In this tutorial, the Faust material will be described both by block diagrams and mathematical expressions. You may use the block diagrams to have a global idea of what the Faust expressions represent, and the mathematical expressions to have a more accurate idea of how it works.

1.3 Block-diagrams

On diagrams, the box-name is drawn in the center of it, or above the upper left corner of it when it contains something (a symbol or some other boxes). When a box is made with a combination of boxes, it is drawn with dashed lines.

The box sometime contains a dot normally in the upper left corner, as on chipsets, that may show if the box is turned upside down. In that case, the text isn't rotated to keep it readable.



The name is written in italic if the box-name isn't a keyword of the language. For example, the built-in box `mem` will be drawn with the name in upright types. A name chosen by the programmer will be drawn with an italic font.

Chapter 2

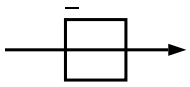
Primitives

A block-diagram — i.e. a Faust expression — can be a single primitive box or the composition of two block-diagrams. These primitive boxes, can also be called built-in boxes.

2.1 Plug boxes

2.1.1 Identity box : _

In certain cases this box can be considered as a wire box. This is the mono dimensional identity function : it does nothing else but transmit the data stream. Because the Faust compiler operates a formal simplification of the block-diagram before the C++ conversion, this kind of box never appears — by any way — in the C++ code. This box is mostly used to clarify the Faust source code.

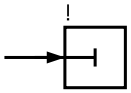


The mathematical representation is :

$$\forall\,(x) \in \mathbb{S}^1,\ [\![\ \_\ ]\!](x) = (x)$$

2.1.2 Cut box : !

This box terminates the stream. It's a dead end. It has one input and zero output. It is mostly used to make the number of input and output match when connecting boxes.



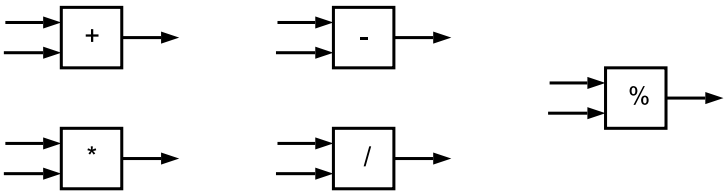
The mathematical representation is :

$$\forall\,(x) \in \mathbb{S}^1,\ [\![\ \texttt{!}\ ]\!](x) = (\,)$$

2.2 Math operators

2.2.1 Arithmetic operators

The four math operators $+ - \times /$ are implemented as boxes in Faust. The operation *modulo*, noted `%` in C++, is also available. Since they are binary operators, the equivalent boxes have 2 inputs and 1 output.



The Faust notations for these boxes are respectively : `+ - * / %`. Beware these operators are boxes, i.e. functions on streams. Consequently, they're not — strictly speaking — Faust operators (those are presented in the next chapter). But we'll see later how the syntax of these functions can be simplified to look like real operators with a little syntactic sugar.

Here's the mathematical representations of `+`, `-`, `*` and `/` :

$$\forall\,(x_1, x_2) \in \mathbb{S}^2,\ [\![\ \texttt{+}\ ]\!](x_1, x_2) = (y)$$
$$\text{with}\quad \forall\, t \in \mathbb{N},\ y(t) = x_1(t) + x_2(t)$$

$$\forall\,(x_1, x_2) \in \mathbb{S}^2,\ [\![\ \texttt{-}\ ]\!](x_1, x_2) = (y)$$
$$\text{with}\quad \forall\, t \in \mathbb{N},\ y(t) = x_1(t) - x_2(t)$$

$$\forall\,(x_1, x_2) \in \mathbb{S}^2,\ [\![\ \texttt{*}\ ]\!](x_1, x_2) = (y)$$
$$\text{with}\quad \forall\, t \in \mathbb{N},\ y(t) = x_1(t) \times x_2(t)$$

$$\forall\,(x_1, x_2) \in \mathbb{S}^2,\ [\![\ \texttt{/}\ ]\!](x_1, x_2) = (y)$$
$$\text{with}\quad \forall\, t \in \mathbb{N},\ y(t) = \frac{x_1(t)}{x_2(t)}$$

The `%` symbol is the C++-like operation *modulo*. It gives the rest of the division of the first argument by the second one. Beware there's no rest when dividing a real number, or by a real number, hence you cannot combine this box with some floating-point-valued streams. See casting boxes in section 2.5, page 14.

2.2.2 Comparison operators

The six comparison operators are also available with the C++ syntax[1] : `<`, `>`, `>=`, `<=`, `!=` and `==`. These other binary operators are provided as boxes of which the output is either the integer 0 or 1. These boxes compare the two inputs and set the output to 0 if the result of the comparison is *false*, and 1 if it's *true*.

Here's the mathematical representation of `<=` :

$$\forall\,(x_1,x_2)\in\mathbb{S}^2,\ [\![\,\texttt{<=}\,]\!](x_1,x_2)=(y)$$
$$\text{with}\quad \forall\,t\in\mathbb{N},\ y(t)=\begin{cases}1 & \text{if } x_1(t)\le x_2(t)\\ 0 & \text{else}\end{cases}$$

Beware to add a space between the serial composition operator ‘ `:` ’ and the greater-than operator ‘ `>` ’ to differentiate it from the merge operator ‘ `:>` ’ (next chapter).

2.3 Bitwise operators

The C++ bitwise operators `<<` and `>>` are provided as boxes. And also the bitwise comparison operators `&`, `|` and `^`, that are respectively the logical *and*, *or* and *xor*. They are all boxes with 2 inputs and 1 output.

$$\forall\,(x_1,x_2)\in\mathbb{S}^2,\ [\![\,\texttt{<<}\,]\!](x_1,x_2)=(y)$$
$$\text{with}\quad \forall\,t\in\mathbb{N},\ y(t)=x_1(t)\ [\![\,\texttt{<<}\,]\!]_{\mathrm{C}}\ x_2(t)$$

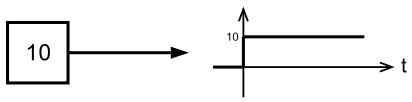
$$\forall\,(x_1,x_2)\in\mathbb{S}^2,\ [\![\,\texttt{\&}\,]\!](x_1,x_2)=(y)$$
$$\text{with}\quad \forall\,t\in\mathbb{N},\ y(t)=x_1(t)\ [\![\,\texttt{\&}\,]\!]_{\mathrm{C}}\ x_2(t)$$

2.4 Constants

A constant box has no input and generates a constant signal. It is defined from the literals C++ syntax. For example : $[\![\,\texttt{10}\,]\!(\ )=(\lambda t\ .\ 10)$.

In the following paragraphs, the symbol # represents any digit or string of digits.

[1] See [Str00] if you are unfamiliar with these notations.



Integers '#' represents an integer. '`012`' and '`12`' both represent a constant box of which the output signal is the integer 12. '`-12`' represents a constant box of which the output signal is the integer −12.

Floats '`#.#`' represents a floating-point number, a rational number. '1.2' represents a constant box of which the output signal is the rational number 1.2. If you want to force an integer (no decimal part) to be considered as a float, then you can write just the point next to the integer : '`12.`' represents a constant box of which the output signal is the rational number 12. You can also write '`12.0`'. If the whole number portion is zero, then you don't have to write the zero : '`.35`' represents a constant box of which the output signal is the rational number 0.35. Negative numbers can also be used.

Beware that Faust only accepts the C++ types `float` and `int`, and no other (no pointer). As for now, there's not yet any predefined constants (like π or the sampling rate).

2.5 Casting

There are two boxes to make the cast operations : `float` and `int`. It will make a C++ implicit cast on a single input in order to generate the output. Most of the time Faust will do the cast, but you may however have to check the process for Faust isn't always accurate and may even in certain circumstances fail to do it. In certain cases, an absence of cast would seriously prevent a good functioning of the program. A correct use of the C++ literals will reduce the need in cast boxes.

2.6 Foreign functions

You can define a box from a C++ function defined in an extern C++ file. To do it you must use the reserved word `ffunction` :

`ffunction(` *prototype* `,` *include* `,` *library* `)`

prototype is the C++ prototype of the function you want to use, e.g. `float sin(float)`. The C++ function can have many arguments. All the arguments of this foreign function will be used as box inputs, respecting the same order. Because the C++ function can only have a maximum of one output, then the box will have one or no output. Remember that Faust only handles `int` and `float` (no pointer nor `double`).

include is the argument of the C++ command `#include`, e.g. `<math.h>`. You must use the exact same argument as you would do to use the function in a C++ program. So you can use both `< >` and `" "`, as in a C++ file.

library is a filename added as a comment at the beginning of the C++ file generated by Faust, useful to remind to the programmer to add this file in the linking mechanism (see section 5.4, page 34).

```
/* Link with library */
```

Because the standard libraries don't have to be explicitly linked to, then you will often use a blank `""` for this argument.

Here's an example that provides the `sin()` function as a box :

```
ffunction( float sin(float), <math.h>, "")
```

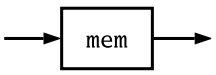
2.7 Memories

The memories are useful to make delays, to store a few values, parameters or data.

2.7.1 Simple delay `mem`

The `mem` box is a delay of which duration equals the sampling period. Here's the mathematical definition :

$$\forall\,(x) \in \mathbb{S}^1,\ [\![\texttt{mem}]\!](x) = (x^{-1})$$

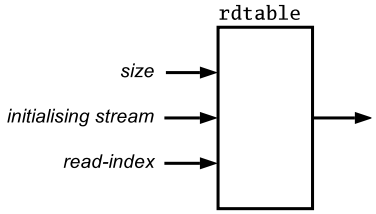


2.7.2 Read-only table

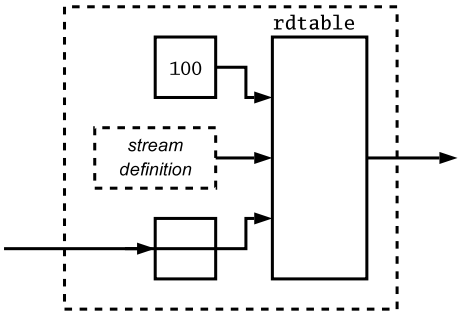
The read-only table `rdtable` allows you to store values from a stream at initialisation time. The data will then be accessed to during execution.

The `rdtable` box has 3 inputs and 1 output. The 3 input streams are :

- size of table : open flow only during initialisation time. Must be a constant integer defined before initialisation.
- initialising stream : open flow during initialisation time which fills the data table.
- read-index : will only be used during execution. Must be a positive integer bounded by 0 and the table size minus one.



During the execution, this box uses only one stream. So, at this time, the box can be considered as having only one input which is the read index. For example :



During execution, the values contained in the table can't be changed. If you need a table which content can be changed during execution, then you need the `rwtable`.

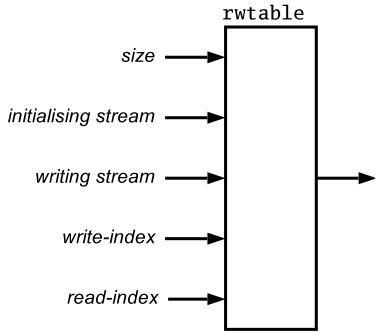
2.7.3 Read-write table

The read-write table `rwtable` is almost the same as the `rdtable` box, except that the data stored at initialisation time can be modified. So it has 2 more input streams for the writing stream and the write-index.

Then, the 5 input streams are :

- size of table,
- initialising stream,

- write-index that must be a positive integer between zero and the table size minus one,
- writing stream, of which instant value will be written a write-index position in the table,
- read-index.

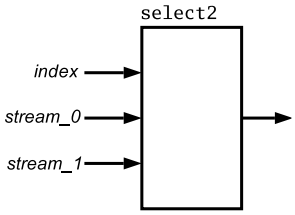


Beware the writing stream can't be stopped, and is always writing at write-index position. You will have to provide an additional cell in the table that will be used as a dead-end, if the writing stream isn't always useful.

2.8 Selection switch

There's two selection switches in Faust : `select2` and `select3`. They let you select 1 stream between 2 or 3, from an index stream.

The `select2` box receives 3 input streams, the selection index, and the two alternative streams.



The math signification of this box is :

$$\forall\,(i,s_1,s_2)\in\mathbb{S}^3,$$
$$[\![\,\texttt{select2}\,]\!](i,s_1,s_2)=(\lambda t\;.\;(1-i(t))\cdot s_1(t)+i(t)\cdot s_2(t)\;)$$

If *index* is 0, then the output is *stream_0*, and if *index* is 1, then the output is *stream_1*. If *index* is greater than 1, then the output is 0, and an error can occur during execution.

The `select3` box is exactly the same except that you can choose between 3 streams.

2.9 Graphic user interface

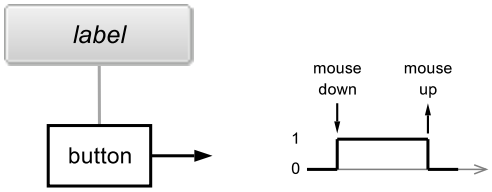
One interesting aspect of Faust is that it can generate graphic user interfaces automatically, which represents an important time-saving. Those primitives have a special behaviour that is described in section 4.1, page 27.

2.9.1 Button

The `button` primitive has the following syntax :

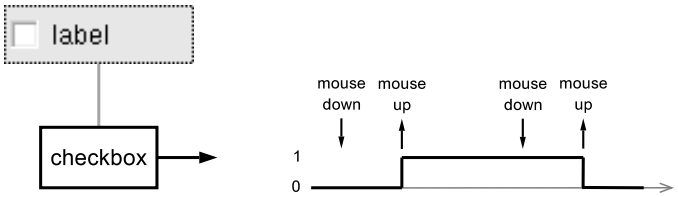
`button("`*label*`")`

This box is a monostable trigger. It has no input, and one output that is set to 1 if the user click the button, and else to 0.



2.9.2 Checkbox

The `checkbox` is a bistable trigger. After a mouse click, the output is set to 1. If you click again the output is set to 0. The state changes on mouse up.



Here's the syntax :

`checkbox("`*label*`")`

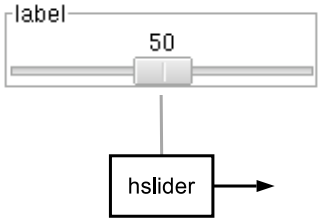
2.9.3 Sliders

The slider boxes `hslider` (horizontal) and `vslider` (vertical) provide some powerful controls for the parameters. Here's the syntax :

`hslider("`*label*`",` *start*`,` *min*`,` *max*`,` *step*`)`

This produce a slider, horizontal or vertical, that let the user pick a value between *min* and *max*−*step*. The initial value is *start*. When the user moves the slider, the value changes by steps of the value of *step*. All the parameters can be `float`s or `int`s.

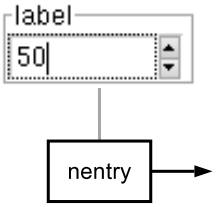
The associated box has no input, and one output which is the value that the slider displays.



2.9.4 Numeric entry

This primitive displays a numeric entry field on the GUI. The output of this box is the value displayed in the field.

`nentry("`*label*`",` *start*`,` *min*`,` *max*`,` *step*`)`



2.9.5 Groups

The primitives `hgroup` and `vgroup` gather GUI objects in a group :

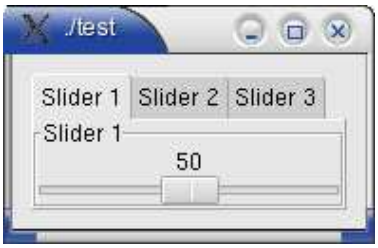
`hgroup("`*label*`",` *Faust expression*`)`

The group gather the GUI objects contained in the *Faust expression* : they are **not** boxes.

On the contrary, the `tgroup` primitive spreads the GUI objects found in a faust expression into tabs, eg :

```
A = hslider("Slider 1",50,0,100,1);
B = hslider("Slider 2",50,0,100,1);
C = hslider("Slider 3",50,0,100,1);
process = tgroup("Sliders",(A,B,C));
```

Here is what one can get after Faust and C++ compilation (with GTK) :



Chapter 3

Composition operators

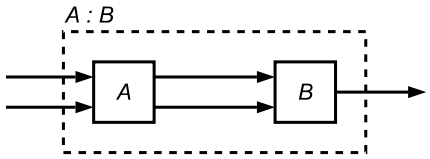
The composition operators are used to combine block-diagrams. From a mathematical point of view, these operators are like the composition of functions. Five composition operators are available in Faust : serial, parallel, split, merge and recursive.

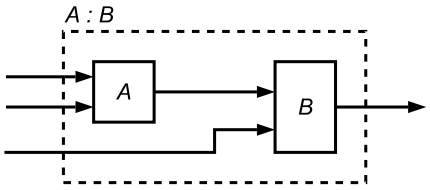
The precedence and the associativity of the operators are given in the last section of this chapter.

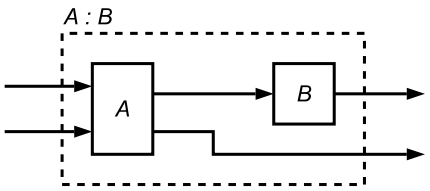
3.1 Serial

Notation : $A : B$

This operator makes a serial connection of the two operand boxes. It means that the outputs of A are connected to the inputs of B, respecting their order. The first output of A is connected to the first input of B, etc... The remaining inputs or outputs are directly used as inputs or outputs of the resulting block-diagram.



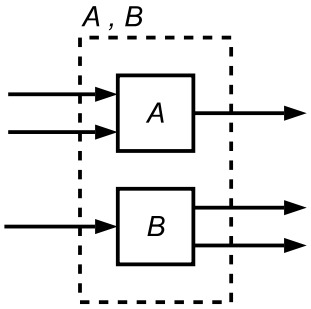




3.2 Parallel

Notation : A **,** B

This operator makes a block-diagram by positioning the two operands in parallel. There's no connection between the boxes. All the inputs of both the boxes are used as inputs of the resulting block-diagram, as for the outputs.



3.3 Split

Notation : A <: B

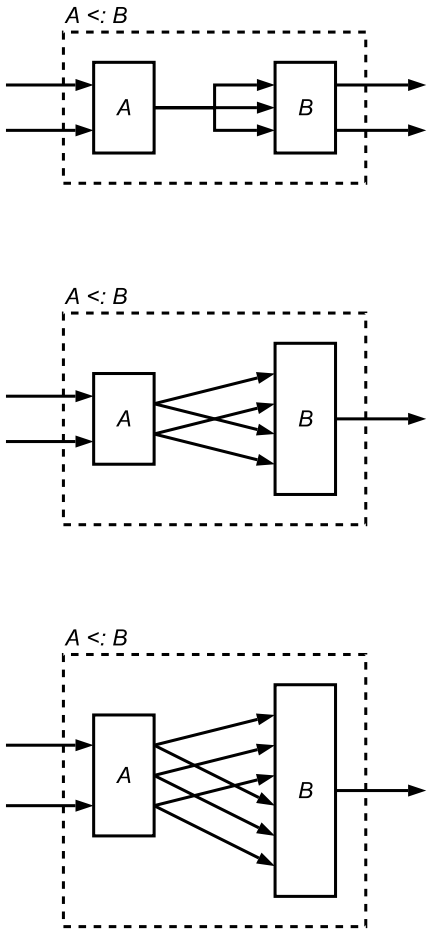
This operator is slightly more complex. The outputs of A are spread on the inputs of B. This means that each output of A is connected to n inputs of B, where n is the number of inputs of B, i_B, divided by the number of outputs of A, o_A :

$$n = \frac{i_B}{o_A}$$

So, because n have to be an integer, i_B have to be a multiple of o_A. If it is not, the Faust compiler ends with this error message :

```
Connection error (<:)  between :  ... error position ...
The number of outputs (2) of the first expression should
be a divisor of the number of inputs (1) of the second
expression
```

The o_A first inputs of B receive the outputs of A, as for the next o_A inputs of B... etc...

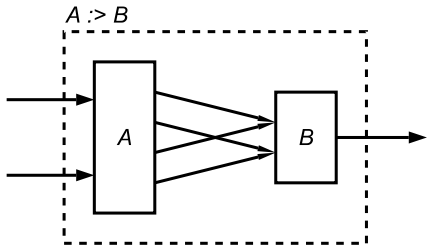


Note that if $o_A = i_B$, the split operator is just equivalent to the serial operator.

3.4 Merge

Notation : A :> B

As the notation suggested it, the merge operator does the inverse composition than the split operator. This time, that's the number of inputs of B, i_B, which is lower than the number of outputs of A, o_A. So, the signals from A are added on the inputs of B with the following rule. The first i_B outputs of A go to the B inputs. Then, the next i_B outputs of A do the same, and so on... Hence, o_A has to be a multiple of i_B.



Since there's the same kind of condition on inputs and outputs number of the argument boxes as with the split operator, there will the same kind of error message if those numbers aren't compatible.

3.5 Recursive

Notation : $A \sim B$

The recursive composition is the most complex one. It is useful to create loops in a program.

This composition has restriction on number of inputs and outputs of each of the boxes A and B. If i_A and o_A are the numbers of inputs and outputs of A, and i_B and o_B, are the numbers of inputs and outputs of B, you must have the following relation :

$$\begin{aligned} & i_B \leq o_A \\ \text{and,} \quad & o_B \leq i_A \end{aligned}$$

The number of inputs and outputs of the resulting box is given by :

$$\begin{aligned} i_{A\sim B} &= i_A - o_B \\ o_{A\sim B} &= o_A \end{aligned}$$

To understand what it does, a diagram will be more explicit than a long explanation. A first example is given in Figure 3.1.

Note that a delay of one sample is set between the A outputs and the B inputs to avoid infinite loops. This is represented by a circle on the diagrams. Here's a mathematical representation of this first example :

$$\begin{aligned} \forall\,(x_1,x_2) \in \mathbb{S}^2,\ [\![\, A \,]\!](x_1,x_2) &= (y_1,y_2) \\ \text{and} \quad [\![\, B \,]\!](y_1^{-1}) &= (z) \\ \text{then} \quad [\![\, A \sim B \,]\!](x_2) &= (y_1,y_2) \end{aligned}$$

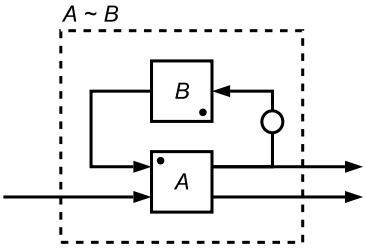


Figure 3.1: Recursive composition, first example

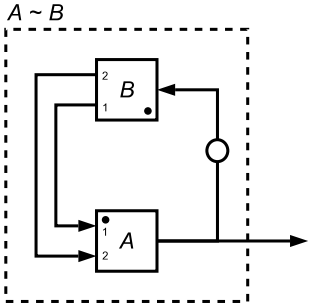


Figure 3.2: Recursive composition, second example

The Figure 3.2 gives another example. In this second example we now have :

$$\begin{aligned} \forall\,(x_1,x_2)\in\mathbb{S}^2,\ [\![\,A\,]\!](x_1,x_2) &= (y) \\ \text{and}\quad [\![\,B\,]\!](y^{-1}) &= (z_1,z_2) \\ \text{then}\quad [\![\,A\sim B\,]\!]() &= (y) \end{aligned}$$

3.6 Precedence

Because of the priority of some operators against some others, you can sometime trim the expressions of its parenthesis. The precedence for each operator is given in the next table, in the decreasing order, so the first is the most priority :

Priority	Symbol	Name
3	~	recursive
2	,	parallel
1	:, <:, :>	serial, split, merge
0	+, -, *, /, ...	arithmetic operators

So instead of

```
A = (B,C):>(D~E) ;
```

you can just write

```
A = B,C :> D~E ;
```

But note that the parenthesis clarify the expression so help in maintaining the code.

To understand very well how the expression you wrote will be interpreted by the compiler, you also need to know the associativity of the operators :

Symbol	Name	Associativity
~	recursive	Left
,	parallel	Right
:, <:, :>	serial, split, merge	Right

Chapter 4

Structure of a Faust program

A Faust program is a signal processor you can represent by a block-diagram. Thus, you should first draw the block-diagram and then transcript it in Faust language. But in Faust, you only handle boxes, no wire. The connections are determined by the composition operators, that is the way the boxes are combined to make a block-diagram.

In practice, a Faust program is a unordered sequence of definitions. Only one definition is required : the definition of `process`. All others are optional.

4.1 Expression and definition

A Faust definition will just provide a shortcut for a Faust expression. A Faust expression can be :

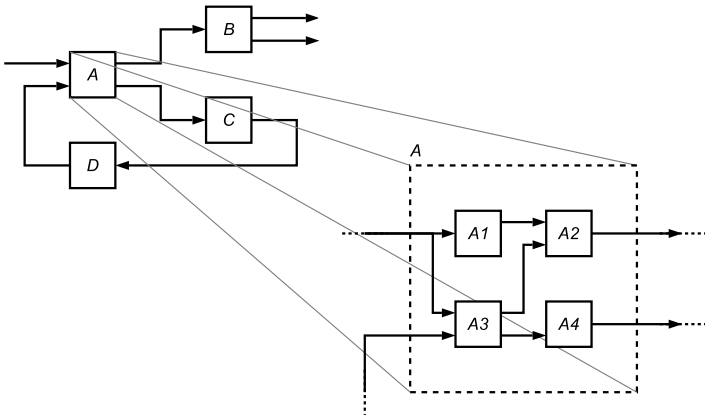
- a primitive box,
- a combination of Faust expressions *via* the composition operators.

A Faust expression can be considered as a block-diagram. Then a Faust definition begins with the name you want to give to this block-diagram, that we'll call *box_name*. Then comes the symbol ' `=` ', the Faust expression, and finally a semicolon ' `;` ' :

box_name `=` *expression* `;`

A *box_name* has to begin with a letter, upper or lower-case, and can contain some figures and underscores ' `_` '. The expression can be written on several lines. And the order of the definitions doesn't matter. You can use a box-name that is defined anywhere, before or after its use. Just beware not to make any recursive definition.

In fact, making a definition is just giving a name to a piece of a bigger expression. It is also to consider a block-diagram as a single box. The block-diagram the Faust program describes is a combination of such boxes. This global block-diagram is defined by the definition of `process`. Therefore, a Faust program must always have this definition, just as a C or C++ program must have a `main` function.



The definitions are just shortcuts means that each occurrence of a box-name in expressions are replaced by their associated expression at compilation. A Faust program is really described by the definition of `process` you get after this substitution mechanism. The `process` box can be split up in boxes, from one to many. Thus, there's always a lot of ways to write a Faust program. The unused definitions are ignored.

Remember that the definition mechanism is just a simple naming mechanism, very different of the naming mechanism of C++ that is in fact an addressing mechanism that links a name to a memory address. There's here nothing comparable. This also explains why you can't use recursive definitions, eg :

```
A = B;
B = A;
process = A:B;
```

This will produce the following error message at compilation :

```
example.dsp:1:  recursive definition of BoxIdent[B]
```

You need the recursive composition operator `~`to do something recursive. In the same way, you can't have multiple definitions or re-definitions of a name. You cannot write :

```
x = 1;
x = 2;
```

In that case, the Faust compiler will use the first definition, but will show no error nor warning.

4.2 Comments

To make your source code readable, you may want to add some comments to it. You can use the common C++-like comments `//` and `/* ... */`. The first syntax makes uninterpreted all the characters up to the end of the line. The second one allows you to make multiple lines comments. All the characters between `/*` and `*/` will not be ridden by the compiler. Note that such comments are not written in the C++ output file.

4.3 Abstraction

An abstraction is a way to describe a generic combination of boxes.

abstraction_name`(`*box1*`,` *box2*`, ...)` `=` *expression* `;`

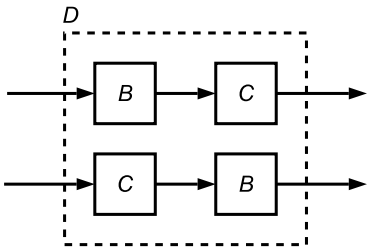
Where the *expression* is a composition of the *box1*, *box2*, and maybe other boxes. Here's an example of what can be done with abstraction :

```
A(x,y) = (x,y):(y,x) ;
```

Then it can be used with some boxes, for example B and C :

```
A(x,y) = (x,y):(y,x) ;
B = ...  ; // definition of B
C = ...  ; // definition of C
D = A(B,C) ;
```

That will define the D box this way :



Note that you can't define a default value as in C++ functions. If you use already defined boxes as formal parameters (*box1*...), their definition

will – locally – not be used. The formal parameters names locally mask any definition with the same name.

You can use some already defined boxes in the definition of the abstraction.

4.4 Syntactic sugar

Some Faust expressions can be written in a more simple way. The math operators provided as boxes can be used with a more natural syntax. Furthermore, the Faust boxes can be used as functions so a function syntax, with parenthesis, is also available.

The math primitives `+ - * / %` can be used in two other ways. Here is the first alternative notation with the `/` primitive :

$$A \texttt{ / } B$$
$$\Updownarrow$$
$$\texttt{(} A \texttt{ , } B \texttt{) : /}$$

The second one would often be used with a constant :

$$\texttt{/(} A \texttt{)}$$
$$\Updownarrow$$
$$\texttt{(\_ , } A \texttt{) : /}$$

This notation also works for the boxes you define. For example, let's have two definitions :

```
sinus = ffunction(float sinus(float), <math.h>, "");
i = +(0.1) ~ _ ;
```

then the following syntax can be used :

```
process = sinus(i);
```

which is equivalent to :

```
process = i:sinus;
```

When the box used as function has several inputs, they are considered as arguments, separated with comma.

Chapter 5

Compilation

The faust compiler will build a C++ file from the program you wrote and from a wrapper file you specify, that will encapsulate your Faust program in a bigger C++ environment.

5.1 Generated C++ code

The Faust program is described in a single C++ class called `mydsp` that inherits of the virtual class `dsp`. The `mydsp` class always contains a method called `compute()`, and if needed two methods `buildUserInterface()` and `init()`.

The `compute()` method begins with some declarations and initialisations, and then there's a loop `for`. The treatment is mainly done in this loop. The aim of this method is to fill the sound-buffer. Thus the number of iterations for this loop is the size of the buffer.

The values that come from the user-interface are constant during this loop for stability reasons. Then, if you can change the size of this buffer, remember that the bigger the buffer is the less your application is reactive.

Faust tries to simplify the numeric expressions but it sometimes fail and some numeric expressions are not simplified. It depends on the way you wrote the Faust expression. So you should try different orders and positions for parenthesis in order to gather all numeric parts of an expression.

5.2 Command-line options

The Faust compiler v1.0a admits some command-line options that allow you to set the way the Faust compiler will process the Faust source file. You can try `faust -h` at prompt to get the list of available options. The most basic command line would be `faust` *filename*`.dsp`.

-v This option prints the compiler version information.

-d This option will verbose the compilation process with details, including the given Faust source file.

-a *filename* This option can be used to specify a C++ wrapper file. An architecture filename must be given as parameter. If no wrapper file is given, the Faust compiler only produces the C++ class.

-o *filename* This option is used to set the C++ output filename. If no filename is given, the Faust compiler just print the resulting C++ source code on the current output.

-ps It will produce a PS file that is the block-diagram representation of the Faust program.

-svg It will produce a SVG file that is the block-diagram representation of the Faust program.

Here's an example :

```
# faust -a oss-gtk.cpp -o rev.cpp rev.dsp
```

That line will compile the reverb.dsp Faust file into the reverb.cpp C++ file, using the architecture/oss-gtk.cpp wrapper file.

5.3 Wrapper files

There's 10 main wrapper files :

alsa-gtk The sound driver is ALSA (Advanced Linux Sound Architecture : `http://www.alsa-project.org`). The graphic user interface is GTK (Gimp Tool-Kit : `http://www.gtk.org`).

jack-gtk The sound driver is Jack (Jack Audio Connection Kit : `http://jackit.sourceforge.net`). The graphic user interface is GTK.

jack-wx The sound driver is Jack. The graphic user interface is cross-platform wxWindows (`http://www.wxwindows.org`).

minimal Just the compilable C++ file. The difference with no wrapper file is that the virtual class `dsp` of which `mydsp` inherits is included in the C++ file to make it ready-to-compile. This wrapper is handy to use the stream processor as a module in a bigger C++ program.

oss-gtk The sound driver is OSS (Open Sound System : `http://www.opensound.com`). The graphic user interface is GTK.

oss-wx The sound driver is OSS. The graphic user interface is wxWindows.

pa-gtk The sound driver is cross-platform PortAudio (`http://www.portaudio.com`). The graphic user interface is GTK.

pa-wx The sound driver is cross-platform PortAudio. The graphic user interface is wxWindows.

plot There's no sound driver nor graphic user interface. It displays the output values in a terminal. The graphical elements (sliders) can be set with the command line options, by preceding the slider names with '`--`'. If the name contains spaces, you must use single quotes '' to bound the slider name. The special option `--n` let you choose the number of samples you want to display.

sndfile There's no sound driver nor graphic user interface. This wrapper handle sound files. You must specify the input and output filenames.

The wrapper files are C++ files. You must add '`.cpp`' to their names to use them.

5.4 How to compile the C++ file

It's a good idea to write a Makefile if you have a Make program on your system. You will have to add some specific commands in the compilation command-line, which depends on the wrapper file you choose. You can consult the Makefile of the examples directory of your Faust distribution to see how the examples are compiled. Here is the standard command line for GNU compiler gcc :

`g++ -Wall -O3 -lm` *libs* *source-file*`.cpp -o` *binary-file*

where *source-file* is the name of the C++ file generated by Faust, and *binary-file* is the name of the binary file (executable) you will obtain. *libs* depends on the wrapper file you used. You must concatenate the corresponding options to build the *libs* statement :

Wrapper types	**Parameters**
Alsa	`-lpthread -lasound`
Oss	`-lpthread`
Jack	`-ljack`
PortAudio	`-lpthread -lportaudio`
SoundFile	`-lsndfile`
Gtk	`` `gtk-config --cflags --libs` ``
wxWindows	`` `wx-config --cflags --libs` ``

The wrapper files minimal and plot don't need any specific libraries.

Beware to link the files required by some `ffunction()`. The file to link should be notified in a comment `/* Link with ... */` at the beginning of the C++ file (see section 2.6, page 15).

Part II

Examples

Chapter 6

Oscillators and signal generators

This chapter presents some oscillators that will be used, then, to built some synthesisers.

6.1 Square wave oscillator

Presentation

A symmetric square wave oscillator is the most basic generator that can be done with Faust. An asymmetric one, with a customisable cyclic ratio is more interesting.

Faust source code

```
1  //----------------------------------
2  //     A square wave oscillator
3  //----------------------------------
4
5  T = hslider("Period",1,0.1,100.,0.1); // Period (ms)
6  N = 44100./1000.*T:int; // The period in samples
7  a = hslider("Cyclic ratio",0.5,0,1,0.1); // Cyclic ratio
8  i = +(1)~%(N):-(1); // 0,1,2...,n
9
10 process = i,N*a : < : *(2) : -(1) ;
```

Block-diagram

See figure 6.1, page page 38.

Output

See figure 6.2, page 39. The command-line was :

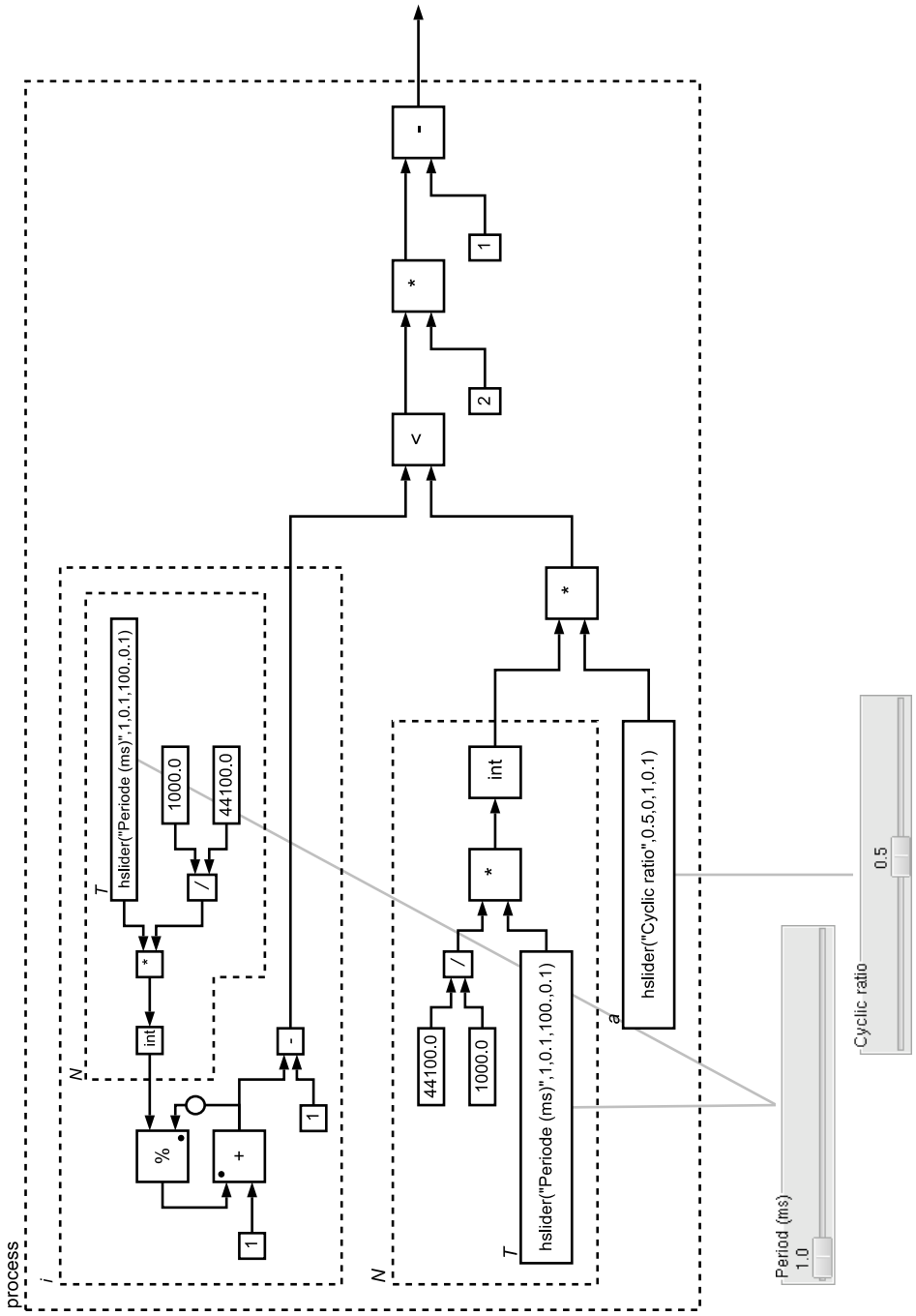


Figure 6.1: Block diagram representation of the Faust program square.dsp

```
# ./square --'Cyclic ratio' 0.2 --'Period' 1 --n 100
```

The period is about 44 samples, ie $44 \times 1000/44100 \approx 1$ ms. During a period, the output signal is 1 for 8 samples, ie for 20% of the samples. This is the cyclic ratio that was expected.

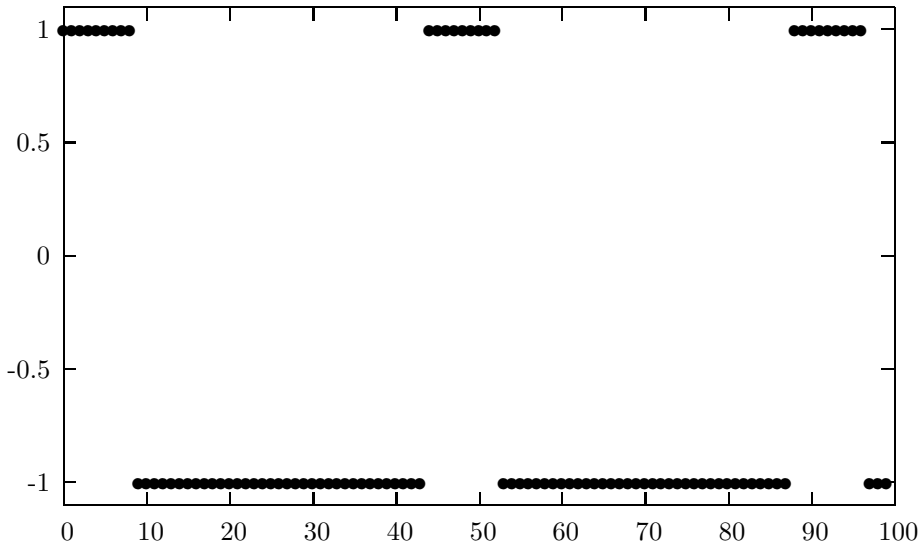


Figure 6.2: 100 first output samples of the oscillator, at 442 Hz

Graphic user interface

The figure 6.3 shows the user interface it produces with the oss-gtk.cpp wrapper file.

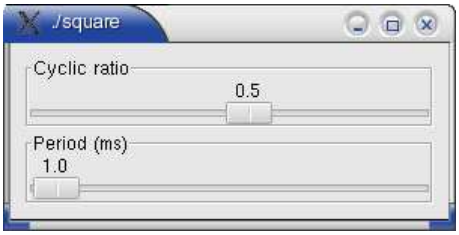


Figure 6.3: GUI of square.dsp

Comments and explanations

For this oscillator we simply count the samples to determine both the period and the number of samples that are set to 1 from the cyclic ratio.

Therefore, we first provide the period in milliseconds with a slider `T` (line 5), and then in number of samples with `N` (line 6). See figure 6.4.

```
T = hslider("Period",1,0.1,100.,0.1); // Period (ms)
N = 44100./1000.*T:int; // The period in samples
```

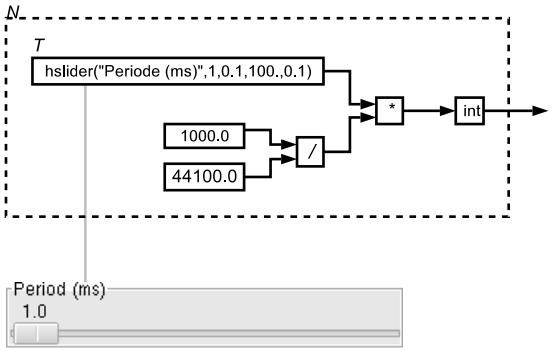


Figure 6.4: `N` sub-block-diagram

Line 7, the cyclic ratio can be chosen by the user. This is the percentage of samples set to 1 during a period. The default value 0.5 will generate a symmetric square wave. See figure 6.5.

```
a = hslider("Cyclic ratio",0.5,0,1,0.1); // Cyclic ratio
```

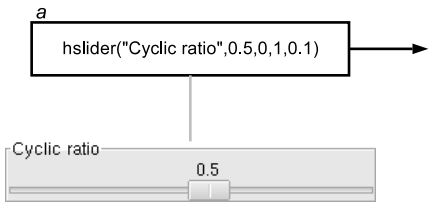


Figure 6.5: `a` sub-block-diagram

Line 8, there's an incremental counter that runs from 0 to the number of samples in a period minus 1. See figure 6.6.

```
i = +(1)~%(N):-(1); // 0,1,2...,n
```

Finally, line 10, the `process` is defined as the comparison between the output streams from the counter `i` and the number of samples that must be set to 1, ie the output stream of `N*a`. Before this number of samples is raised, the output is 1. After, but during the period, the output is 0. To adjust this output to the whole possible range $[-1; 1]$, i.e. to normalize, it is multiplied by 2, and then diminished of 1. See figure 6.7

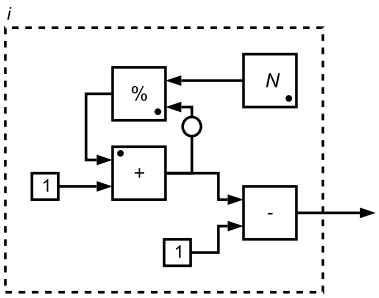


Figure 6.6: `i` sub-block-diagram

```
process = i,N*a : < : *(2) : -(1) ;
```

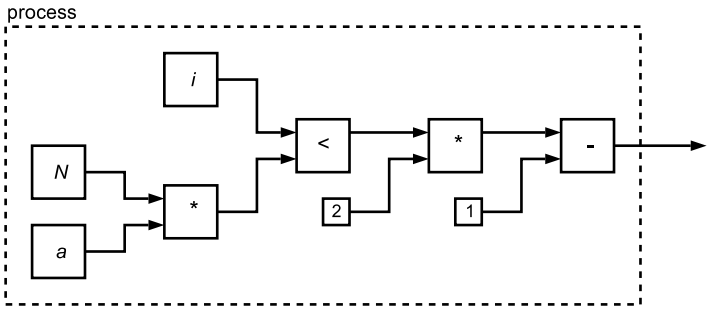


Figure 6.7: `process` block-diagram

The signal we generate has one main flaw. The period converted in whole number of samples. So the signal may not have the exact expected period. We could have implement a compensation mechanism on several periods. But such a mechanism would have produce some periods of variable duration (i.e. a frequency modulation) that induces some uncontrolled additional harmonics in the signal.

6.2 Harmonic oscillator

Presentation

This oscillator is the basic sinusoidal oscillator. It uses a `rdtable` to store some values of the sin function in order to avoid their calculation during the process. These values are then read at the speed that is computed from the frequency parameter. It is supposed here that the sampling rate is 44 100 Hz.

Faust source code

Here's the Faust source code e13_osc.dsp :

```
1  //-----------------------------------------------
2  //       Sinusoidal Oscillator
3  //-----------------------------------------------
4
5  // Mathematical functions & constants
6  //-------------------------------------
7  sin   = ffunction(float sin (float), <math.h>, "");
8  floor = ffunction(float floor (float), <math.h>, "");
9  PI    = 3.1415926535897932385;
10
11 // Oscillator definition
12 //-------------------------
13 tablesize    = 40000;
14 samplingfreq = 44100.;
15
16 time        = (+(1)~_) - 1;  // 0,1,2,3,...
17 sinwaveform = time*(2*PI)/tablesize : sin;
18
19 decimal     = _ <: -(floor);
20 phase(freq) = freq/samplingfreq :
21                   (+ : decimal)~_ : *(tablesize) : int;
22 osc(freq)   = phase(freq) : rdtable(tablesize,sinwaveform);
23
24 // User interface
25 //----------------
26 vol  = hslider("volume", 0, 0, 1, 0.001);
27 freq = hslider("freq", 400, 0, 15000, 0.1);
28
29 //----------------
30 process = osc(freq) * vol;
```

Block-diagram

See figure 6.8, page 43.

Output

See figure 6.9, page 44. We used the plot wrapper file to get the values. This is an easy way to debug when the signal processor has no input. Here's the command lines we typed :

```
# faust -a plot.cpp -o e13_osc.cpp e13_osc.dsp
```

to convert the Faust program in a C++ program. Then :

```
# g++ -Wall -O3 -lm -lpthread e13_osc.cpp -o e13_osc
```

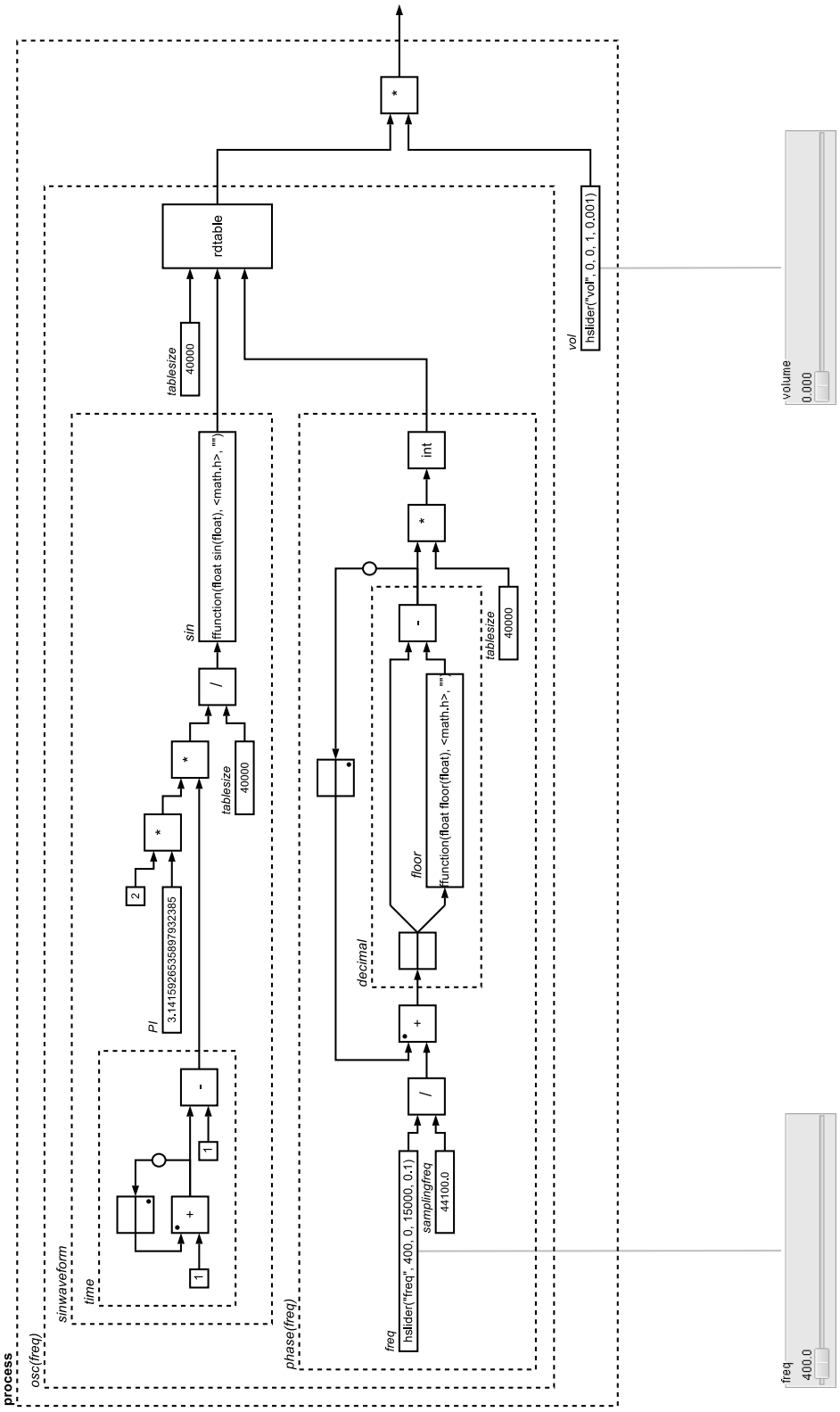


Figure 6.8: Block-diagram representation of the Faust program e13_osc.dsp

to compile the C++ program. The list of flags depends on the wrapper file you used. For example, if you choose a GTK interface, you'll need the GTK libraries.

Finally, we typed :

```
# ./e13_osc --Frequency 442 --n 250 > sinus.dat
```

to launch the program and keep the output values in the file sinus.dat. Then, GnuPlot made this graphical representation of the array of values.

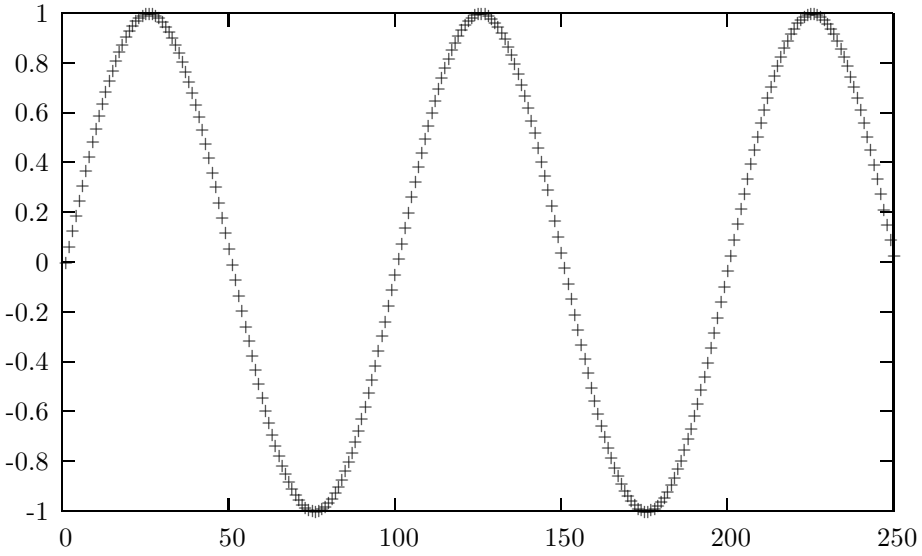


Figure 6.9: 250 first output samples of the oscillator, at 442 Hz

Generated C++ source code

Only the code of the class `mydsp` is given here. This is the wrapper independent part of the resulting file. The processing function is `compute()`. You might understand it even if you're unfamiliar with C++.

```
1  class mydsp :  public dsp {
2   private:
3    class SIG0 {
4     private:
5      int R1_0;
6     public:
7      virtual int getNumInputs() { return 0; }
8      virtual int getNumOutputs() { return 1; }
9      void init(int samplingRate) {
10       R1_0 = 0;
11     }
```

```
12      void fill (int count, float output[ ]) {
13        for (int i=0; i<count; i++) {
14          R1_0 = (R1_0 + 1);
15          output[i] = sin((((R1_0 - 1) * 6.283185) / 40000));
16        }
17      }
18    };
19
20    float fslider0;
21    float R0_0;
22    float fslider1;
23    float ftbl0[40000];
24  public:
25    virtual int getNumInputs() { return 0; }
26    virtual int getNumOutputs() { return 1; }
27    virtual void init(int samplingRate) {
28      fslider0 = 0.0;
29      R0_0 = 0.0;
30      fslider1 = 400.0;
31      SIG0 sig0;
32      sig0.init(samplingRate);
33      sig0.fill(40000,ftbl0);
34    }
35    virtual void buildUserInterface(UI* interface) {
36      interface->openVerticalBox("");
37      interface->addHorizontalSlider("volume", &fslider0,
38                                     0.0, 0.0, 1.0, 1.000000e-03);
39      interface->addHorizontalSlider("freq", &fslider1,
40                                     400.0, 0.0, 15000.0, 0.1);
41      interface->closeBox();
42    }
43    virtual void compute (int count, float** input, float** output)
44  {
45      float* output0 = output[0];
46      float ftemp0 = fslider0;
47      float ftemp1 = (fslider1 / 44100.0);
48      for (int i=0; i<count; i++) {
49        float ftemp2 = (R0_0 + ftemp1);
50        R0_0 = (ftemp2 - floor(ftemp2));
51        output0[i] = (ftbl0[int((R0_0 * 40000))] * ftemp0);
52      }
53    }
54  };
```

Graphic user interface

The figure 6.10 shows the user interface it produces with the oss-gtk.cpp wrapper file.

Comments and explanations

We can already appreciate the compactness of the Faust code compared to the block diagram, or the C++ code. Then, we can observe that the

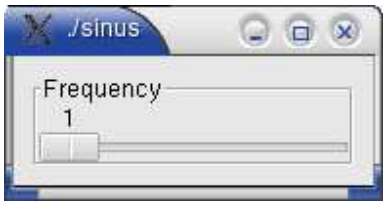


Figure 6.10: GUI of e13_osc.dsp

functional specifications are satisfied : this is an harmonic oscillator, we can choose the frequency, and it uses a table to store the sinus values. Note that the graph of figure 6.9 is graduated with samples. It means that the period, and then the frequency, have to be calculated. Actually, we observed the expected frequency (442 Hz in this example).

First we needed the sinus function, so we took it from the standard C++ library math.h. This is the first line of the Faust program :

```
sin = ffunction(float sin(float), <math.h>, "");
```

The name that will be replaced by the `ffunction` expression is `sin`. The box it represents acts exactly as sin function. See figure 6.11.

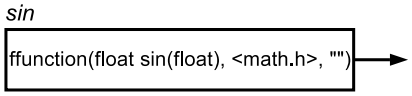


Figure 6.11: `sin` sub-block-diagram of e13_osc.dsp

The C++ `floor` function will also be used. So a similar definition is written line 8. See figure 6.12.

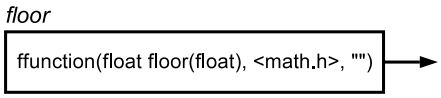


Figure 6.12: `floor` sub-block-diagram of e13_osc.dsp

Line 9, we defined the constant π (see figure 6.13) :

```
PI = 3.1415926535897932385;
```

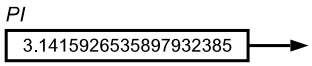


Figure 6.13: `PI` sub-block-diagram of `e13_osc.dsp`

Then we need two more constants that will be the size of the table in which we'll save the values of sinus :

```
tablesize = 40000;
```

The table-size doesn't really matter, and you can have good results even with a very smaller table. Then the sampling-rate constant is defined :

```
samplingfreq = 44100.;
```

See figure 6.14. Note that the table-size is an integer, whereas the sampling frequency is a float.

Figure 6.14: `tablesize` and `samplingfreq` sub-block-diagrams

Line 16, we defined an 'infinite' incremental counter that will give the number of samples from 0 to n :

```
time = (+(1)~_) - 1;  // 0,1,2,3,...
```

The output of `time` is an integer. It just add 1 to the previous value of the output. At the first iteration, the delay of one sample due to the recursive operator (see section 3.5, page 24) will set to 0 the input of the `+(1)` box because signals are causal ones. Then the first output of the block `+(1)~_` is $0 + 1 = 1$. So this block is followed by minus 1 to reset it to 0 at beginning. The incrementation cannot be infinite. When the maximal integer value is reached, the output jump to the maximal negative value, and increment again. Note that the `+(1)` syntax provides a box that adds 1 to the input signal. This is a sweetened syntax for `_,1:+`. See figure 6.15.

Line 17, we defined the stream that will be used to feed the table. We want to have the values of sinus for a whole period stored in the table. Then we divide 2π by the table-size and we use `time` as variable. Thus the table

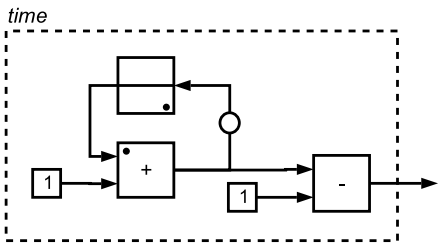


Figure 6.15: `time` sub-block-diagram of e13_osc.dsp

we'll be filled we these values :

$$\begin{bmatrix} \sin(0) \\ \sin(2\pi \cdot \frac{1}{n}) \\ \sin(2\pi \cdot \frac{2}{n}) \\ \vdots \\ \sin(2\pi \cdot \frac{n-2}{n}) \\ \sin(2\pi \cdot \frac{n-1}{n}) \end{bmatrix}$$

where n is the size of the table. The `sin` function is used as a box (see figure 6.16) :

```
sinwaveform = time*(2*PI)/tablesize : sin;
```

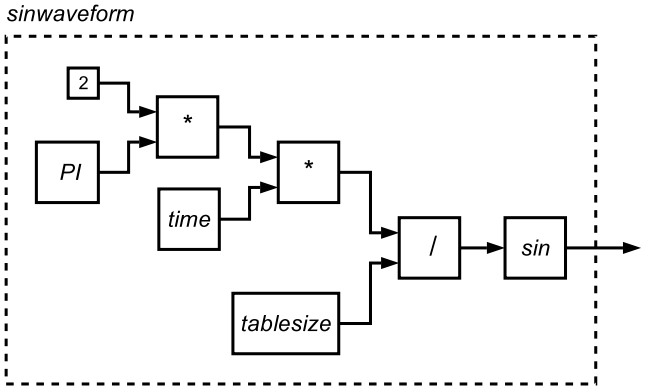


Figure 6.16: `sinwaveform` sub-block-diagram of e13_osc.dsp

The table we'll be read with a read-index that we must compute. The matter is that we always read the table values at the sampling frequency. Therefore, to generate any frequency, we must skip some values while reading the table. Thus the read-index is an incremental counter which step depends

on the frequency the user decided to generate. Moreover, the read-index mustn't exceed the table-size. So we will first figure a ratio between 0 and 1.0 that we'll next multiply by the size of the table. This ratio we'll be incremented of a step we must figure too, and that depends on frequency.

After incrementation, the ratio may go over 1.0. So we first defined a box, line 19, that only keeps the decimal value of a national number, which is obviously between 0 and 1.0 :

```
decimal = _ <: -(floor);
```

The `floor` box gives the largest integer not greater the value of the input stream. In C++ `floor(6.04)` returns `6.0`. Note that the `-(floor)` block has two inputs. Then the split operator ' `<:` ' spread the output of the wire box on those two inputs. See figure 6.17.

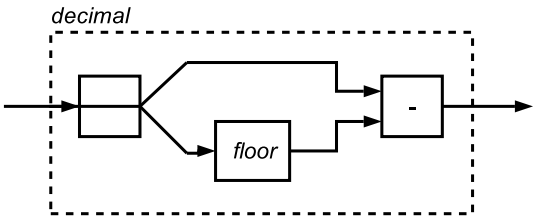


Figure 6.17: `decimal` sub-block-diagram of `e13_osc.dsp`

Line 20, we defined an abstraction with one argument `freq` :

```
phase(freq) = freq/samplingfreq :
                    (+ : decimal) ~ _ : *(tablesize) : int;
```

When this abstraction will be used, `freq` has to be replaced by the definition-name of the block that defines the frequency. As explained before, the incrementing step is `freq/samplingfreq`. This step increments a value for each sample. This value is then translated between 0 and 1.0 with the `decimal` block. The obtained ratio is multiplied by the table-size and casted to `int` to make a valid read-index. Then after i iterations, i.e. samples, the value the table gives with this read-index is :

$$\sin(2\pi \cdot \frac{i}{n} \cdot \frac{f}{F_s} \cdot n) = \sin(2\pi \cdot f \cdot \frac{i}{F_s})$$

where f is the frequency given by `freq`, F_s is the sampling-frequency and n the table-size. The time in seconds is given by the number of samples multiplied by the duration of one sample, i.e. the sampling period $1/F_s$. Thus, the output signal will be a pure sinusoidal signal which frequency is

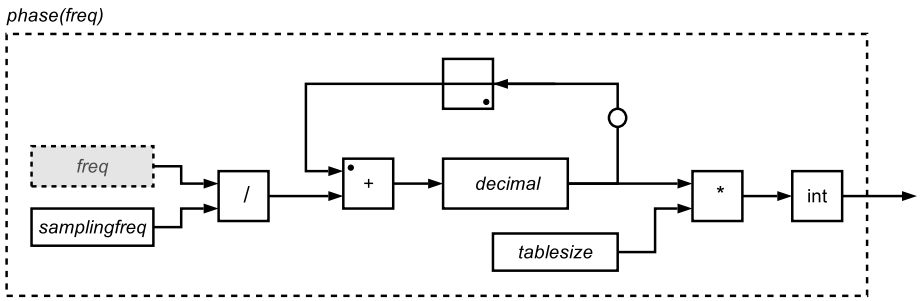


Figure 6.18: `phase(freq)` sub-block-diagram of e13_osc.dsp

f. See figure 6.18. The `freq` is grayed because this is the argument of the `phase` abstraction.

Line 22, the oscillator is defined as an abstraction too. It takes the same argument `freq`. As described before, we make table of sin values with `rdtable(tablesize,sinwaveform)`, and we then read it with the read-index `phase(freq)`.

```
osc(freq) = phase(freq) : rdtable(tablesize,sinwaveform);
```

Note that the syntax used here for the `rdtable` is equivalent to :

```
(tablesize,sinwaveform) : rdtable
```

The third input of the `rdtable` is connected to `phase(freq)`. See figure 6.19.

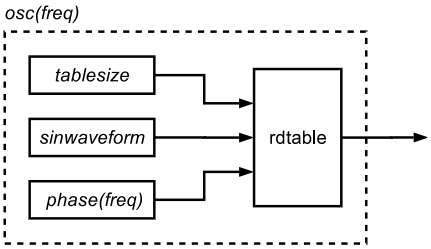


Figure 6.19: `osc(freq)` sub-block-diagram of e13_osc.dsp

Finally, the user interface is defined with two `hslider`. The first one, line 26, is the volume of the output signal.

```
vol = hslider("volume", 0, 0, 1, 0.001);
```

This is just a factor between 0 and 1.0. The default value, that is set when the program is launched, is 0. Thus the user will have to move the slider to hear something.

Line 27, the second slider is for the frequency :

```
freq = hslider("freq", 400, 0, 15000, 0.1);
```

Here the lowest frequency allowed by the interface is 0 Hz. This has no interest for two reasons. First, we can't hear frequencies lower than about 15 Hz. Then, because the table only contains 40,000 values. So even if we read one value at each sample, we'll get an output signal frequency higher than 1 Hz.

Finally, the global `process` is defined by :

```
process = osc(freq) * vol;
```

We used the slider `freq` for the argument of the abstraction `osc()`, and the output signal is multiplied by the value the slider `vol` emits. See figure 6.20.

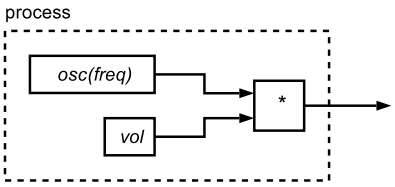


Figure 6.20: `process` block-diagram of e13_osc.dsp

6.3 Noise generator

Presentation

This example presents two way to make some noise. The first one uses the C++ `random()` function. The second one uses a simple function that based on the `int` property. Because integers can't be infinite, if you increment one, at a moment it will wrap to a negative number, and restart to increase. If you increment this integer of a non-constant value and big enough, you'll have a pseudo-randomised numbers sequence.

The Faust program we'll do compares this to noises. The user interface has only one button that let the user switch between the two noises.

Faust source code

```
1 //-----------------------------------------------
2 //          Two noises compared
3 //-----------------------------------------------
4
```

```
5  RANDMAX = 2147483647;
6
7  random1 = ffunction(int random (), <stdlib.h>, "");
8  noise1 = (random1 << 1) * (1.0/RANDMAX);
9
10 random2 = (*(1103515245)+12345) ~ _ ;
11 noise2 = random2 * (1.0/RANDMAX);
12
13 compare(a,b) = (a*(1-button("Switch")) +
14                                   b*button("Switch"));
15
16 process = compare(noise1, noise2) *
17           hslider("volume", 0, 0, 1, 0.01) <: _,_ ;
```

Block-diagram

See figure 6.21, page 53.

Output

See figure 6.22, page 54. There's in fact two output streams, but because they are identical, only one has been represented. The first graph represents the output when the button is unclicked (with volume set to 0.5), ie the output of the `noise1` block-diagram. The second graph, figure 6.23, page 54 shows the output when the button is clicked.

Graphic user interface

The figure 6.24 shows the user interface it produces with the oss-gtk.cpp wrapper file.

Comments and explanations

The first algorithm uses the `random()` C++ function. In the Faust program, it is used in the `noise1` block-diagram. The second one is more rustic. It consists in incrementing a number of a big variable value that will make wrap the integer to negative values, and so on. This algorithm should use less resources because there's no call to any external function.

The `compare` abstraction is exactly the equivalent of the `select2` box that could have been used instead. The output of the `button("Switch")` box is 0 when the button is unclicked, and 1 when it is clicked. Therefore, the output of `compare(a,b)` is `a` when the button is unclicked, and `b` else.

The output of the resulting block-diagram of `compare` applied to the two noises is then multiplied by a volume factor that is interfaced with a slider. The output is split in two streams, with the split composition operator `<:` to have a stereo output. The two channels (left and right) are identical.

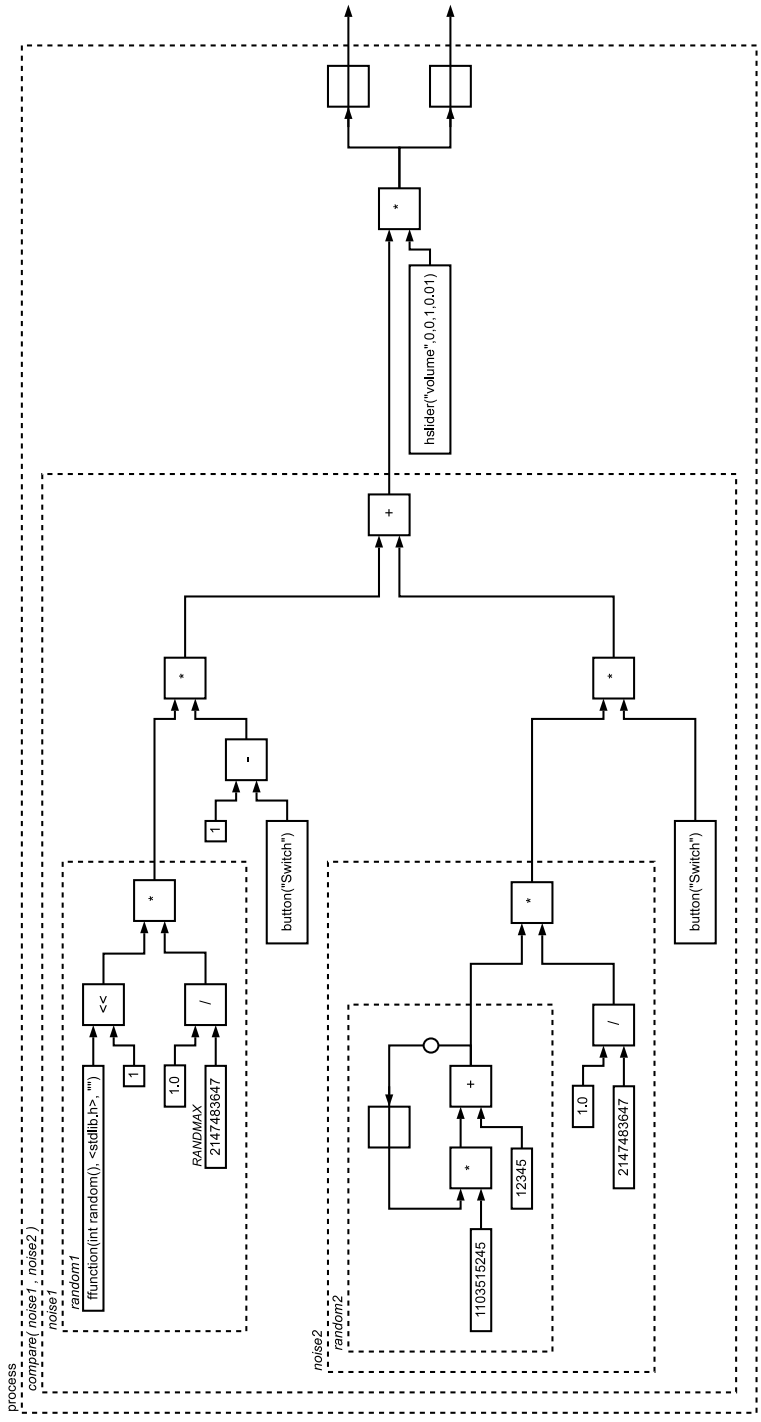


Figure 6.21: Block diagram representation of the Faust program 2noises.dsp

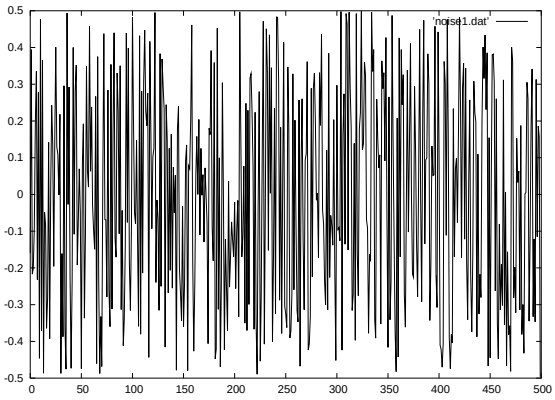


Figure 6.22: 500 first values from 2noises.dsp, with 'Switch' button unclicked

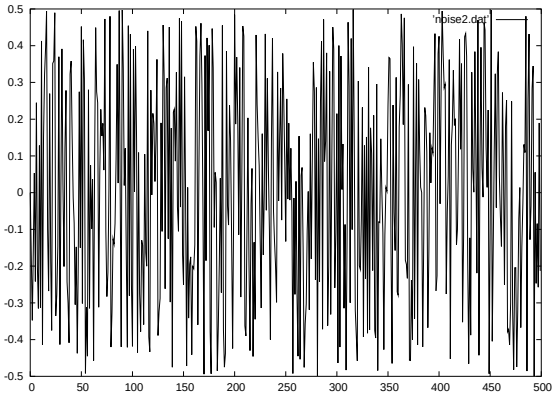


Figure 6.23: 500 first values from 2noises.dsp, with 'Switch' button clicked

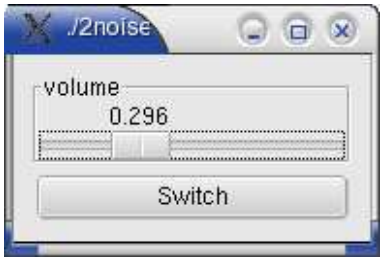


Figure 6.24: GUI of 2noises.dsp

Chapter 7

Karplus-Strong

7.1 Presentation

The KS algorithm was presented by Karplus and Strong in 1983 [Roa96]. This algorithm can generate metallic picked-string sounds, and, by extension, some metallic sounds (drums, strings...). It has been introduced in a chipset called Digitar.

7.2 Faust source code

Here's the Faust source code e05_karplus_strong.dsp :

```
1  //-----------------------------------------------
2  //          Karplus - Strong
3  //-----------------------------------------------
4
5  // noise generator
6  //---------------------
7  random  = (*(1103515245)+12345) ~ _;
8  RANDMAX = 2147483647;
9  noise   = random * (1.0/RANDMAX);
10
11 // delay lines
12 //---------------------
13 index(n)   = _ & (n-1) ~ +(1);      // n = 2**i
14 delay(n,d) = n, 0.0, index(n), _, (index(n)-int(d))
15                                        & (n-1) : rwtable;
16
17 // trigger
18 //--------
```

```
19 impulse    = _ <: _, mem : - : (_>0.0);
20 release(n) = + ~ (_ <: _, (_>0)/n : -);
21 trigger(n) = impulse : release(n) : _>0;
22
23 // user interface
24 //-----------------
25 play(n) = button("play"):trigger(n);
26 dur     = hslider("duration", 128, 2, 512, 1);
27 att     = hslider("attenuation", 0, -10, 10, 0.1);
28
29 karplus1(bt, dl, att) = noise*bt:
30            (+ <: delay(4096, dl-1), delay(4096, dl))
31            ~ (+:*((1-bt)/(2+att/100))) : !,_ ;
32
33 karplusStrong = karplus1(play(dur), dur, att);
34
35 process = karplusStrong;
```

7.3 Block-diagram

The whole block-diagram is to big to be displayed on a single page. Some definitions and a portion of code are first presented. They are used then in the diagram of the whole process.

The figure 7.1 page 56 represents the sub-block-diagram `noise`.

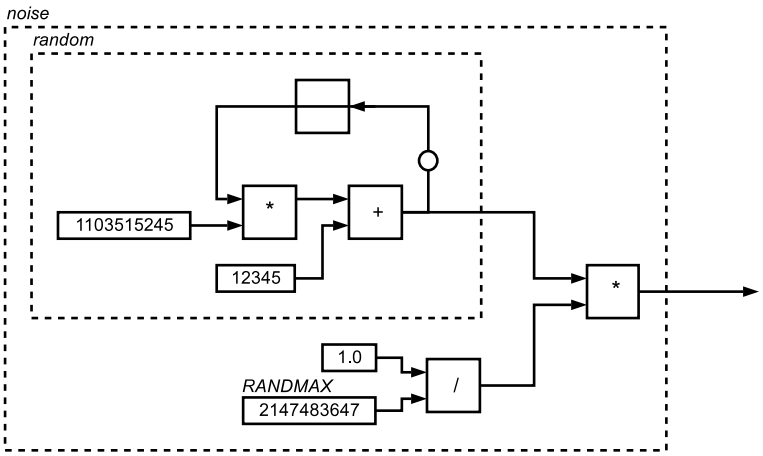


Figure 7.1: Block-diagram of `noise`

The figure 7.2 page 57 represents the block-diagram of the abstraction `delay(n,d)`. The formal parameters `n` and `d` are grayed.

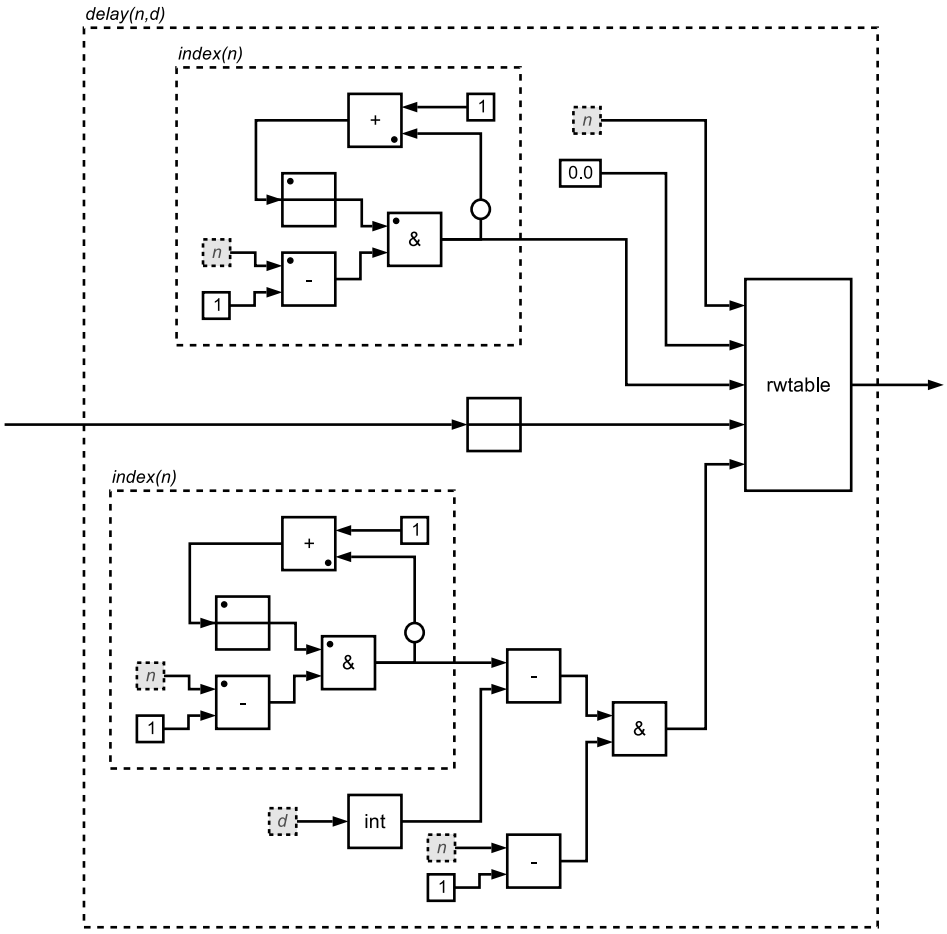


Figure 7.2: Block-diagram of the abstraction `delay(n,d)`

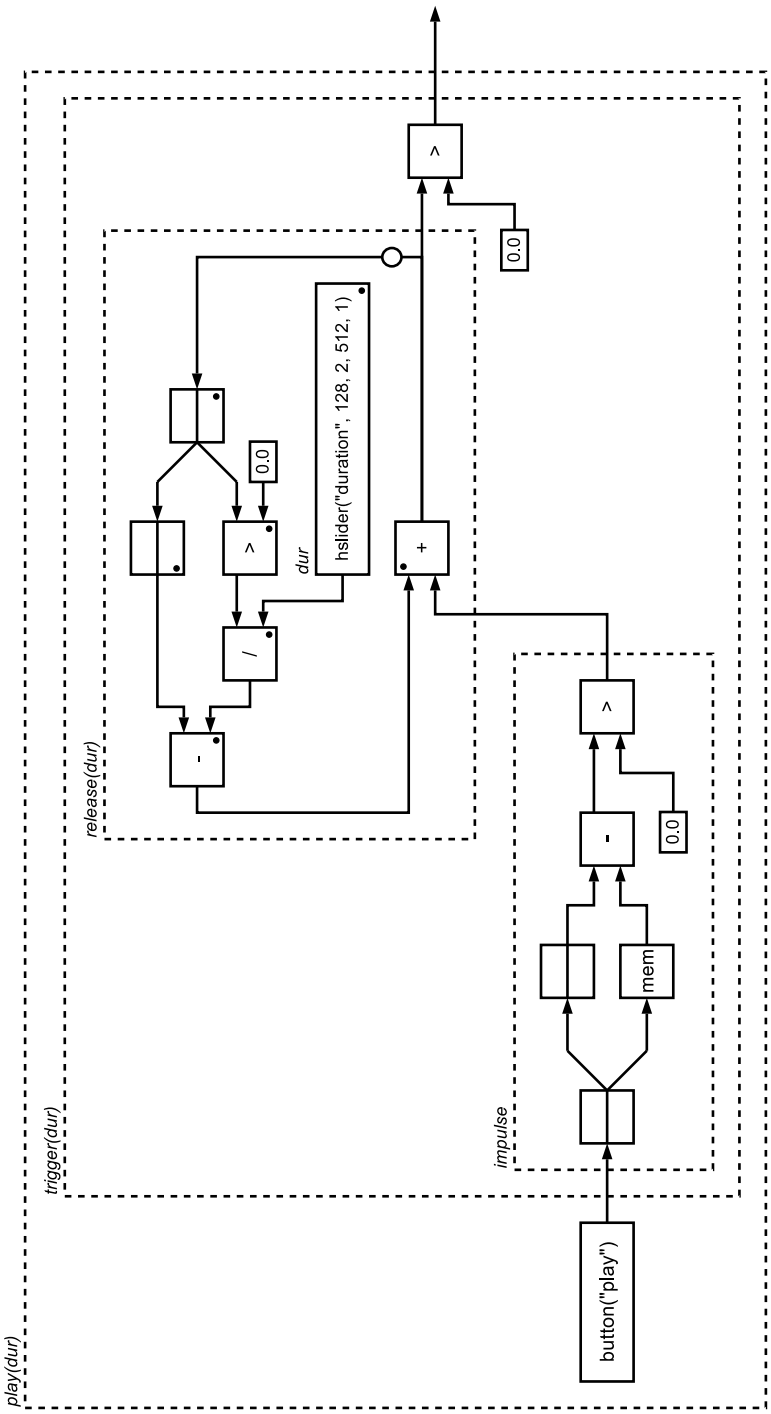


Figure 7.3: Block-diagram of the abstraction `play(n)`

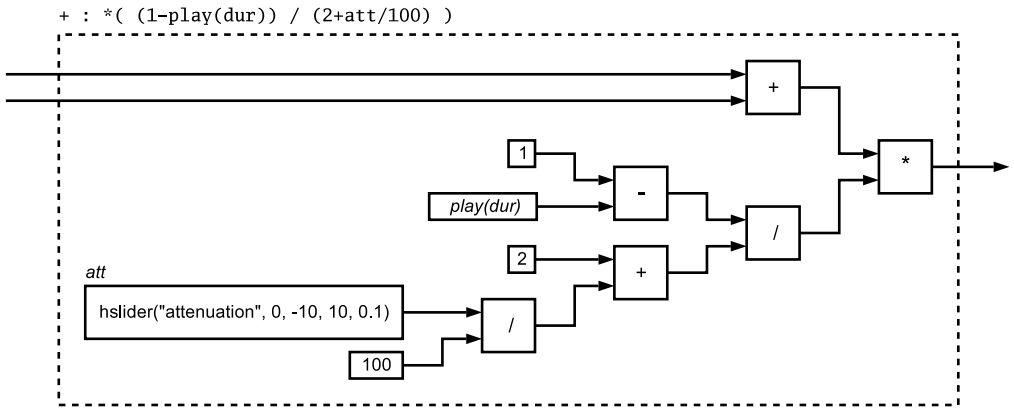


Figure 7.4: Block-diagram for part of line 31 '`+ : *( (1-bt)/(2+att/100))`' with '`bt`' replaced by '`play(dur)`'

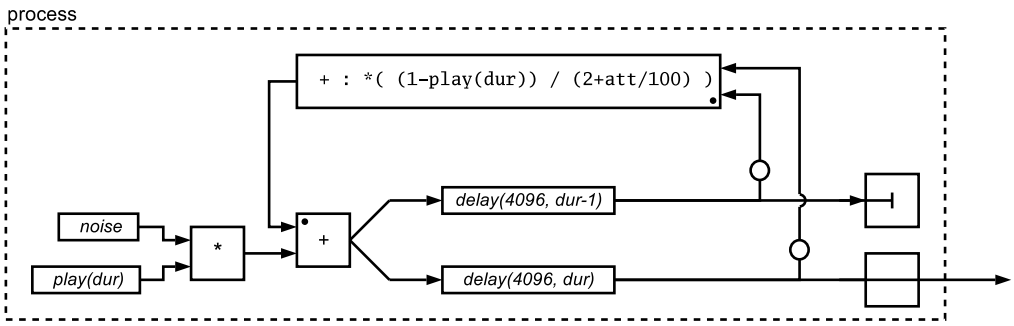


Figure 7.5: Block-diagram of `process`

The figure 7.3 page 58 represents the block-diagram of the abstraction `play(n)`.
The figure 7.4 page 59 represents the sub-block-diagram for the portion of code of line 31 :

```
+:*((1-bt)/(2+att/100))
```

The figure 7.5 page 59 represents the whole block-diagram for `process`.

7.4 Graphic user interface

You can see, figure 7.6 page 60, the graphic user interface of the program compiled with a GTK wrapper file.

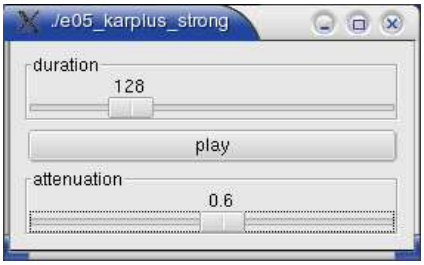


Figure 7.6: Karplus-Strong synthesiser graphic user interface

7.5 Comments and explanations

The algorithm begins with a wave-table fed by random values. These values are then read, modified and re-used to feed the table again. See figure 7.7 page 60.

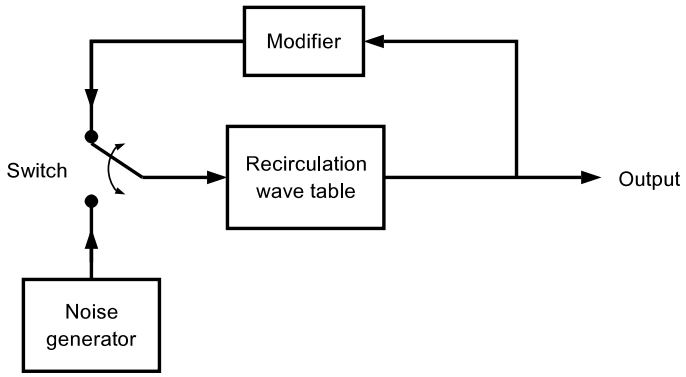


Figure 7.7: General principle diagram of the KS algorithm

The most basic modifier is a mean function. Here the modifier is more complex. There's some specific delays in it, and it fades out the sound.

Noise generator

The table must be fed with some new noise values. Here's the noise generator used in the e05_karplus_strong.dsp, line 7 to 9.

First, we generate an integer random value. The integers are bounded by a maximal value and a negative minimal value. When the maximal value is reached while incrementing a variable, it jumps to the minimal value. The current pseudo-random integers generator uses both a multiplication and an addition to make a non-constant increment value. See figure 7.8.

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

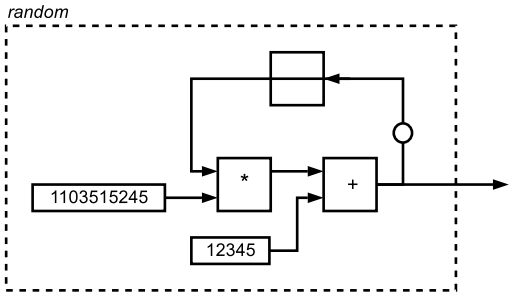


Figure 7.8: Sub-block-diagram for `random`

The parameters are chosen to avoid any periodic result. In fact, the generated sequence may be periodic after a long moment.

The sound streams in Faust are floats between -1.0 and $+1.0$. The generated random stream is between the max-integer and the min-integer. So we defined a constant `RANDMAX` which is used to normalize the stream (between -1.0 and $+1.0$). This constant depends on the number of bits on which the integers are coded. So it depends on the C++ compiler that will be used.

```
RANDMAX = 2147483647;
noise = random * (1.0/RANDMAX);
```

See figure 7.1, page 56. Note that the division with `RANDMAX` has to involve a float somewhere to make a float division, otherwise, we would always have get 0. So we could have defined `RANDMAX` as a float and divide `random` directly by `RANDMAX` to gain one operation. But `RANDMAX` is a constant. So when we write `(1.0/RANDMAX)`, it will be simplified by the Faust compiler. The C++ translation of this portion is :

```
R3_0 = ((R3_0 * 1103515245) + 12345);
```

for `random`, and `noise` is then used this way :

```
...(R3_0 * 4.656613e-10)...
```

Delay lines

The recirculation-wave-table in fact uses some delay lines [Roa96]. Those delay lines are based on `rwtable`s.

We first implemented an incremental index that will be used to read and write in the `rwtable`. It will loop-count from 0 to $n-1$. This index uses the bitwise operator `&` instead of modulo `%`, so n must be a power of 2 : $n = 2^i$.

```
index(n) = _ & (n-1) ~ +(1);   // n = 2**i
```

This is an abstraction.

The delay is then defined on a `rwtable` :

```
delay(n,d) = n, 0.0, index(n), _, (index(n)-int(d))
                                        & (n-1) : rwtable;
```

The table is initialised with silence, i.e. the constant stream 0.0. The write index is `index(n)`. The read index is `index(n)-int(d)`. `n` is the size of the table and `d` is the delay. The read index is reset between 0 and $n-1$ with another `&`. See figure 7.2, page 57.

Trigger

The user will click the "Play" button for any duration. So we make a trigger that generates a signal of controlled duration on mouse-down event.

First, we just keep a single pulse at the pulse edge. This is `impulse`, line 19 :

```
impulse = _ <: _, mem : - : (_>0.0);
```

The figure 7.9 shows chronograms. The `mem` box add a delay of one sample to the input signal. The difference between the original and the dephased signal produces a signal with two pulses large of one sample at the pulse edges. The greater-than box just let the first positive pulse.

The abstraction `release(n)` is used to produce a decreasing slope from 1.0 to 0 and then negative values, and of which duration is given by the output stream of `n`. The slope begins when a single pulse comes to the input.

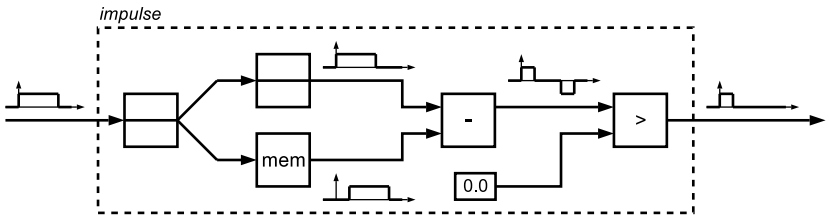


Figure 7.9: Sub-block-diagram for `impulse`

```
release(n) = + ~ ( _ <: _, (_>0)/n : -);
```

See figure 7.3, page 58.

This abstraction is used in `trigger(n)` with `impulse` connected to the input, and `_>0` to the output. The output signal is a Heaviside unit step which duration is controlled by the formal parameter `n`.

```
trigger(n) = impulse : release(n) : _>0;
```

User interface

We first have a "play" button that will key on the sound. This button is implemented by the abstraction `play(n)` :

```
play(n) = button("play"):trigger(n);
```

The output will be the one of `trigger(n)`. The aim is just to control the duration of the launching stream that is user event.

Two sliders are then described :

```
dur = hslider("duration", 128, 2, 512, 1);
att = hslider("attenuation", 0, -10, 10, 0.1);
```

The first one `dur` is the duration of the Heaviside unit step generated by `trigger`. It will replace the formal parameter `n` in the previous abstractions. The second one, `att` is an attenuation factor. It will be used in order to alter mean used as modifier.

Whole stream processor

The KS algorithm uses the noise generated during `dur` as initial waveform. We then use two delay-lines, one with one sample time-lag more. The mean of those two streams is done with a specific factor that depends on the duration and the attenuation.

```
karplus1(bt, dl, att) = noise*bt :
              (+ <: delay(4096, dl-1), delay(4096, dl))
                    ~ (+:*((1-bt)/(2+att/100))) : !,_ ;
```

This abstraction will be used with these parameters :

```
karplusStrong = karplus1(play(dur), dur, att);
```

The abstraction isn't necessary here. In another version, a lot of KS algorithm with different parameters are launched in parallel. The abstraction lighten the code. And remember that contrary to the C++ function call, the Faust abstraction mechanism doesn't increase the resources cost.

The calculation of the mean factor is represented in figure 7.4, page 59.

Finally, we just keep one of the two output signals of the delay-lines. The other is cut withe a `!`.

The `process`, is just the `karplusStrong` stream processor :

```
process = karplusStrong;
```

Part III

Appendices

Appendix A

Known bugs and limitations

The following bugs and limitations have been reported for the Faust compiler v1.0a. This list isn't exhaustive.

A.1 Bugs

- The multiple definitions are not detected. It should be tested if they are equivalent or at least identical.
- Some automatic casts are not performed by the Faust compiler. For example, it lets the user make modulo (%) operations with floats, which will produce an error at the compilation of the C++ code.
- That's a small bug but, in the error message ERROR infering write table type, it should be written inferring.

A.2 Limitations

- The symbolic simplification isn't accurate enough.
- There's no simple mechanism to generate simple sequences of numbers.
- There's not any predefined constant. Some useful constants like π and the sampling-rate will be predefined in the next versions.
- The error messages are not really explicit.
- The errors detected during the compilation process (not during the parse process) cannot be localised due to the simplification mechanism.

- When using a table (`rdtable` or `rwtable`) the index (read or write) isn't controlled and can be override the table size. It produces unexpected results or an abnormal termination of the program.
- The compiler only generates a C++ file. The user has to compile it. The next versions should produce a wrapper-dependent Makefile that would help compiling.

Appendix B

Error messages

Here are some common error messages that the Faust compiler v1.0a is susceptible to generate at compilation. There's two types of errors.

First, there are some syntax errors that are detected during the parsing operation. Those errors can be localised in the source file. So the error message begins with the filename and the line number at which the error has been detected. Some other errors, specific to the language can be detected at this level.

The second type is some errors that are detected during the compilation. The Faust compiler gathers all the definition in a single expression that will be simplified. Therefore, it is impossible to find the origin of the error. The generated error message will not show the localisation of the error, which sometime makes the debugging laborious. They are qualified of wriggled errors because of there elusiveness.

B.1 Localised errors

- **test.dsp:2:parse error, unexpected IDENT**

  Means that an unexpected identifier as been found line 12 of the file test.dsp. Usually, this error is due to the absence of the semi-column ' ; ' at the end of the previous line. Here is the test.dsp source code :

```
1 box1 = 1
2 box2 = 2;
3 process = box1,box2;
```

test.dsp:1:parse error, unexpected ENDDEF

The parser find the end of the definition (' ; ') while searching a closing right parenthesis.

```
1 t = _~(+(1);
2 process = t / 2147483647.;
```

- **test.dsp:1:parse error, unexpected ENDDEF, expecting LPAR**

  The parser was expecting a left parenthesis. It identified a keyword of the language that requires arguments.

```
1 process = ffunction;
```

- **test.dsp:1:parse error, unexpected RPAR**

  The wrong parenthesis may has been used :

```
1 process = +)1);
```

  The following messages may be equivalent :

  test.dsp:1:parse error, unexpected RPAR, expecting LPAR
  test.dsp:1:parse error, unexpected RPAR, expecting PAR

- **test.dsp:1:parse error, unexpected SPLIT**

  The ‘ < ’ box is followed by the serial composition operator ‘ ; ’, with no space :

```
1 process = <:+;
```

  The same confusion is possible with the merge operator and greater-than.

- **test.dsp:1: undefined symbol BoxIdent[turlututu]**

  Happens when you use a undefined name.

```
1 process = turlututu;
```

- **test.dsp:1:parse error, unexpected IDENT, expecting ADD or SUB or INT or FLOAT**

  Happens when you use a undefined name instead of something expected.

```
1 process = hslider("Slider 1",box,0,100,1);
```

- **test.dsp:18:parse error, unexpected STRING**

  The signals can only be numeric streams. To generate symbols, you must translate them in numeric streams. Moreover, you can’t, for the moment, generate a looped sequence.

```
1 process = "01001";
```

- **test.dsp:1: recursive definition of BoxIdent[B]**

  Means that the box named `B`, i.e. the definition of `B`, is recursive. You can't make a recursive definition like in C++ because the definition names are replaced by there associated expression during parsing process. To make a recursion you must use `~`, the recursive composition operator.

```
1 A = B;
2 B = A;
3 process = A:B;
```

  The message is repeated for the second line and cause another one. So here is the complete error message :

  test.dsp:1: recursive definition of BoxIdent[B]
  test.dsp:1: recursive definition of BoxIdent[B]
  Internal Error, box expression not recognized : BoxError

B.2 Wriggled errors

These errors are mainly type errors. Faust tries to generates some error-free C++ code. So it checks the types needed by the different primitives and sometimes makes itself the appropriate cast. When it fails, the Faust compiler displays a wriggled error message.

- **Error : checkInit failed for type RSEVN**

  On a `rdtable`, the read-index must be an integer. So the type is checked. Here, nothing is specified, so the check failed.

```
1 process = rdtable;
```

- **Error : checkInt failed for type RKCVN**

  The check failed for the size of the table.

```
1 process = 10.0, 1.0, int : rdtable;
```

- **ERROR infering read table type, wrong write index type : RSEVN**

  Actually, this error message should be "inferring" and "wrong read index type".

```
1 process = 10, 1.0, float : rdtable;
```

- **ERROR infering write table type, wrong write index type : RSEVN**

  The same for `rwtable`.

Bibliography

[Gun92] C. A. Gunter, *Semantics of Programming Languages: Structures and Techniques*. MIT Press, 1992.

[Hug89] J. Hughes, "Why Functional Programming Matters," *Computer Journal*, vol. 32, no. 2, pp. 98–107, 1989.

[OFL02] Y. Orlarey, D. Fober, and S. Letz, "An algebra for block diagram languages," in *Proceedings of International Computer Music Conference* (ICMA, ed.), pp. 542–547, 2002.

[Roa96] C. Roads, *The Computer Music Tutorial*. MIT Press, 1996.

[Str00] B. Stroustrup, *The C++ Programming Language*. Addison-Wesley, Reading, Massachusetts, USA, 2000.

Index