FAUST
Functional Synchronous Programming for Signal Processing

Y. Orlarey

GRAME – Centre National de Création Musicale

Séminaire MaMux/Langages Synchrones

1-Introduction

Introduction

What is FAUST ?

FAUST stands for *Functional AUdio STream*:

- It is a *Domain-Specific Language* for real-time audio signal processing and synthesis.
- It can be used to develop:
  - *audio effects,*
  - *sound synthesizers*
  - real-time applications processing signals.
- Who uses FAUST ?
  - Developers of audio applications and plugins
  - Sound engineers and musical assistants
  - Researchers in Computer Music

FAUST stands for *Functional AUdio STream*:

- It is a *Domain-Specific Language* for real-time audio signal processing and synthesis.
- It can be used to develop:
  - *audio effects*
  - *sound synthesizers*
  - real-time applications processing signals
- Who uses FAUST ?
  - Developers of audio applications and plugins
  - Sound engineers and musical assistants
  - Researchers in Computer Music

FAUST stands for *Functional AUdio STream*:

- It is a *Domain-Specific Language* for real-time audio signal processing and synthesis.
- It can be used to develop:
  - *audio effects*,
  - *sound synthesizers*
  - real-time applications processing *signals*.
- Who uses FAUST ?
  - Developers of audio applications and plugins,
  - Sound engineers and musical assistants
  - Researchers in Computer Music

FAUST stands for *Functional AUdio STream*:

- It is a *Domain-Specific Language* for real-time audio signal processing and synthesis.
- It can be used to develop:
  - *audio effects*,
  - *sound synthesizers*
  - real-time applications processing *signals*.
- Who uses FAUST ?
  - Developers of audio applications and plugins,
  - Sound engineers and musical assistants
  - Researchers in Computer Music

FAUST stands for *Functional AUdio STream*:

- It is a *Domain-Specific Language* for real-time audio signal processing and synthesis.
- It can be used to develop:
  - *audio effects*,
  - *sound synthesizers*
  - real-time applications processing *signals*.
- Who uses FAUST ?
  - Developers of audio applications and plugins,
  - Sound engineers and musical assistants
  - Researchers in Computer Music

FAUST stands for *Functional AUdio STream*:

- It is a *Domain-Specific Language* for real-time audio signal processing and synthesis.
- It can be used to develop:
  - *audio effects*,
  - *sound synthesizers*
  - real-time applications processing *signals*.
- Who uses FAUST ?
  - Developers of audio applications and plugins,
  - Sound engineers and musical assistants
  - Researchers in Computer Music

FAUST stands for *Functional AUdio STream*:

- It is a *Domain-Specific Language* for real-time audio signal processing and synthesis.
- It can be used to develop:
  - *audio effects*,
  - *sound synthesizers*
  - real-time applications processing *signals*.
- Who uses FAUST ?
  - Developers of audio applications and plugins,
  - Sound engineers and musical assistants
  - Researchers in Computer Music

FAUST stands for *Functional AUdio STream*:

- It is a *Domain-Specific Language* for real-time audio signal processing and synthesis.
- It can be used to develop:
  - *audio effects*,
  - *sound synthesizers*
  - real-time applications processing *signals*.
- Who uses FAUST ?
  - Developers of audio applications and plugins,
  - Sound engineers and musical assistants
  - Researchers in Computer Music

FAUST stands for *Functional AUdio STream*:

- It is a *Domain-Specific Language* for real-time audio signal processing and synthesis.
- It can be used to develop:
  - *audio effects*,
  - *sound synthesizers*
  - real-time applications processing *signals*.
- Who uses FAUST ?
  - Developers of audio applications and plugins,
  - Sound engineers and musical assistants
  - Researchers in Computer Music

FAUST stands for *Functional AUdio STream*:

- It is a *Domain-Specific Language* for real-time audio signal processing and synthesis.
- It can be used to develop:
  - *audio effects*,
  - *sound synthesizers*
  - real-time applications processing *signals*.
- Who uses FAUST ?
  - Developers of audio applications and plugins,
  - Sound engineers and musical assistants
  - Researchers in Computer Music

FAUST stands for *Functional AUdio STream*:

- It is a *Domain-Specific Language* for real-time audio signal processing and synthesis.
- It can be used to develop:
  - *audio effects*,
  - *sound synthesizers*
  - real-time applications processing *signals*.
- Who uses FAUST ?
  - Developers of audio applications and plugins,
  - Sound engineers and musical assistants
  - Researchers in Computer Music

Introduction

What is a FAUST program ?

A FAUST program describes a *signal processor* :

- A (periodically sampled) *signal* is a *time* to *samples* function:
  - $\mathbb{S} = \mathbb{N} \rightarrow \mathbb{R}$
- A *signal processor* is a *signals* to *signals* function:
  - $\mathbb{P} = \mathbb{S}^n \rightarrow \mathbb{S}^m$
- Everything in FAUST is a *signal processor* :
  - $+ : \mathbb{S}^2 \rightarrow \mathbb{S}^1 \in \mathbb{P}$,
  - $3.14 : \mathbb{S}^0 \rightarrow \mathbb{S}^1 \in \mathbb{P}, \ldots,$
- Programming in FAUST is essentially combining signal processors :
  - $\{ : \;\; , \;\; <: \;\; :> \;\; \sim \} \subset \mathbb{P} \times \mathbb{P} \rightarrow \mathbb{P}$

Introduction

What is a FAUST program ?

A FAUST program describes a *signal processor* :

- A (periodically sampled) *signal* is a *time* to *samples* function:
  - $\mathbb{S} = \mathbb{N} \rightarrow \mathbb{R}$
- A *signal processor* is a *signals* to *signals* function:
  - $\mathbb{P} = \mathbb{S}^n \rightarrow \mathbb{S}^m$
- Everything in FAUST is a *signal processor* :
  - $+ : \mathbb{S}^2 \rightarrow \mathbb{S}^1 \in \mathbb{P}$,
  - $3.14 : \mathbb{S}^0 \rightarrow \mathbb{S}^1 \in \mathbb{P}, \ldots,$
- Programming in FAUST is essentially combining signal processors :
  - $\{ : \; , \; <: \; :> \; \sim \} \subset \mathbb{P} \times \mathbb{P} \rightarrow \mathbb{P}$

A FAUST program describes a *signal processor* :

- A (periodically sampled) *signal* is a *time* to *samples* function:
  - $\mathbb{S} = \mathbb{N} \rightarrow \mathbb{R}$
- A *signal processor* is a *signals* to *signals* function:
  - $\mathbb{P} = \mathbb{S}^n \rightarrow \mathbb{S}^m$
- Everything in FAUST is a *signal processor* :
  - $+ : \mathbb{S}^2 \rightarrow \mathbb{S}^1 \in \mathbb{P}$,
  - $3.14 : \mathbb{S}^0 \rightarrow \mathbb{S}^1 \in \mathbb{P}, \ldots,$
- Programming in FAUST is essentially combining signal processors :
  - $\{$`:` `,` `<:` `:>` `~`$\} \subset \mathbb{P} \times \mathbb{P} \rightarrow \mathbb{P}$

A FAUST program describes a *signal processor* :

- A (periodically sampled) *signal* is a *time* to *samples* function:
  - $\mathbb{S} = \mathbb{N} \rightarrow \mathbb{R}$
- A *signal processor* is a *signals* to *signals* function:
  - $\mathbb{P} = \mathbb{S}^n \rightarrow \mathbb{S}^m$
- Everything in FAUST is a *signal processor* :
  - $+ : \mathbb{S}^2 \rightarrow \mathbb{S}^1 \in \mathbb{P}$,
  - $3.14 : \mathbb{S}^0 \rightarrow \mathbb{S}^1 \in \mathbb{P}, \ldots,$
- Programming in FAUST is essentially combining signal processors :
  - $\{$`:` `,` `<:` `:>` `~`$\} \subset \mathbb{P} \times \mathbb{P} \rightarrow \mathbb{P}$

A FAUST program describes a *signal processor* :

- A (periodically sampled) *signal* is a *time* to *samples* function:
  - $\mathbb{S} = \mathbb{N} \rightarrow \mathbb{R}$
- A *signal processor* is a *signals* to *signals* function:
  - $\mathbb{P} = \mathbb{S}^n \rightarrow \mathbb{S}^m$
- Everything in FAUST is a *signal processor* :
  - `+` : $\mathbb{S}^2 \rightarrow \mathbb{S}^1 \in \mathbb{P}$,
  - `3.14` : $\mathbb{S}^0 \rightarrow \mathbb{S}^1 \in \mathbb{P}, \ldots,$
- Programming in FAUST is essentially combining signal processors :
  - $\{\texttt{:}\ \ \texttt{,}\ \ \texttt{<:}\ \ \texttt{:>}\ \ \texttt{\~{}}\} \subset \mathbb{P} \times \mathbb{P} \rightarrow \mathbb{P}$

Introduction

What is a FAUST program ?

A FAUST program describes a *signal processor* :

- A (periodically sampled) *signal* is a *time* to *samples* function:
  - $\mathbb{S} = \mathbb{N} \rightarrow \mathbb{R}$
- A *signal processor* is a *signals* to *signals* function:
  - $\mathbb{P} = \mathbb{S}^n \rightarrow \mathbb{S}^m$
- Everything in FAUST is a *signal processor* :
  - `+` : $\mathbb{S}^2 \rightarrow \mathbb{S}^1 \in \mathbb{P}$,
  - `3.14` : $\mathbb{S}^0 \rightarrow \mathbb{S}^1 \in \mathbb{P}, \ldots,$
- Programming in FAUST is essentially combining signal processors :
  - $\{$`:  ,  <:  :>  ~`$\} \subset \mathbb{P} \times \mathbb{P} \rightarrow \mathbb{P}$

Introduction

Example of signal processor

- A digital signal processor, here a Lexicon 300, can be modeled as a *mathematical function* transforming *input signals* into *output signals*.
- FAUST allows to describe both the *mathematical computation* and the *user interface*.

Introduction

Example of signal processor

- A digital signal processor, here a Lexicon 300, can be modeled as a *mathematical function* transforming *input signals* into *output signals*.
- FAUST allows to describe both the *mathematical computation* and the *user interface*.

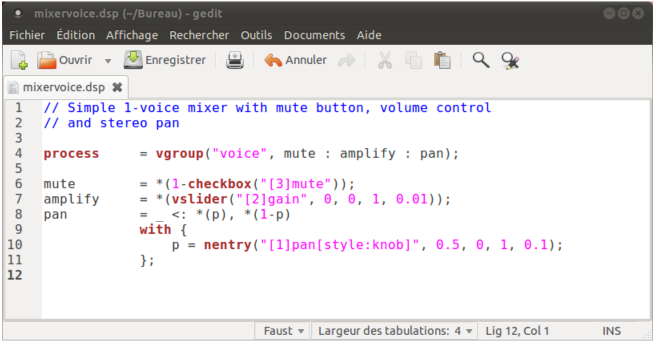
Introduction

Example of signal processor

- A digital signal processor, here a Lexicon 300, can be modeled as a *mathematical function* transforming *input signals* into *output signals*.
- FAUST allows to describe both the *mathematical computation* and the *user interface*.

Introduction

A simple FAUST program




```
// Simple 1-voice mixer with mute button, volume control
// and stereo pan

process     = vgroup("voice", mute : amplify : pan);

mute        = *(1-checkbox("[3]mute"));
amplify     = *(vslider("[2]gain", 0, 0, 1, 0.01));
pan         = _ <: *(p), *(1-p)
            with {
                p = nentry("[1]pan[style:knob]", 0.5, 0, 1, 0.1);
            };
```

Figure: Source code of a simple 1-voice mixer

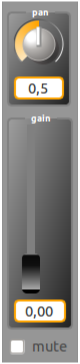


Figure: Resulting application

FAUST is based on several design principles:

- High-level Specification language
- Purely functional approach
- Textual, block-diagram oriented, syntax
- Efficient sample level processing
- Fully compiled code (sequential or parallel)
- Embeddable code (no runtime dependences, constant memory and CPU footprint)
- Easy deployment : single code multiple targets (from VST plugins to iPhone or standalone applications)

FAUST is based on several design principles:

- High-level Specification language
- Purely functional approach
- Textual, block-diagram oriented, syntax
- Efficient sample level processing
- Fully compiled code (sequential or parallel)
- Embeddable code (no runtime dependences, constant memory and CPU footprint)
- Easy deployment : single code multiple targets (from VST plugins to iPhone or standalone applications)

FAUST is based on several design principles:

- High-level Specification language
- Purely functional approach
- Textual, block-diagram oriented, syntax
- Efficient sample level processing
- Fully compiled code (sequential or parallel)
- Embeddable code (no runtime dependences, constant memory and CPU footprint)
- Easy deployment : single code multiple targets (from VST plugins to iPhone or standalone applications)

FAUST is based on several design principles:

- High-level Specification language
- Purely functional approach
- Textual, block-diagram oriented, syntax
- Efficient sample level processing
- Fully compiled code (sequential or parallel)
- Embeddable code (no runtime dependences, constant memory and CPU footprint)
- Easy deployment : single code multiple targets (from VST plugins to iPhone or standalone applications)

FAUST is based on several design principles:

- High-level Specification language
- Purely functional approach
- Textual, block-diagram oriented, syntax
- Efficient sample level processing
- Fully compiled code (sequential or parallel)
- Embeddable code (no runtime dependences, constant memory and CPU footprint)
- Easy deployment : single code multiple targets (from VST plugins to iPhone or standalone applications)

FAUST is based on several design principles:

- High-level Specification language
- Purely functional approach
- Textual, block-diagram oriented, syntax
- Efficient sample level processing
- Fully compiled code (sequential or parallel)
- Embeddable code (no runtime dependences, constant memory and CPU footprint)
- Easy deployment : single code multiple targets (from VST plugins to iPhone or standalone applications)

FAUST is based on several design principles:

- High-level Specification language
- Purely functional approach
- Textual, block-diagram oriented, syntax
- Efficient sample level processing
- Fully compiled code (sequential or parallel)
- Embeddable code (no runtime dependences, constant memory and CPU footprint)
- Easy deployment : single code multiple targets (from VST plugins to iPhone or standalone applications)

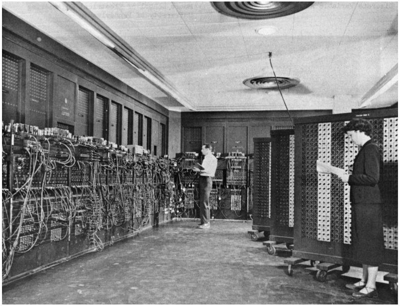
FAUST is based on several design principles:

- High-level Specification language
- Purely functional approach
- Textual, block-diagram oriented, syntax
- Efficient sample level processing
- Fully compiled code (sequential or parallel)
- Embeddable code (no runtime dependences, constant memory and CPU footprint)
- Easy deployment : single code multiple targets (from VST plugins to iPhone or standalone applications)

2-Block Diagram Algebra

Block-Diagram Algebra

Programming by patching is familiar to musicians :



Block-Diagram Algebra

Today programming by patching is widely used in Visual Programming Languages like Max/MSP:

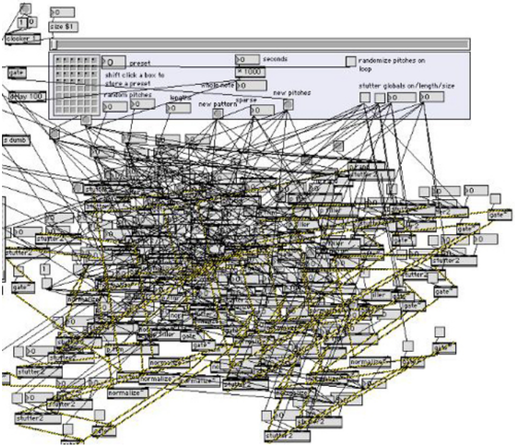


Figure: Block-diagrams can be a mess

Block-Diagram Algebra

Faust allows structured block-diagrams

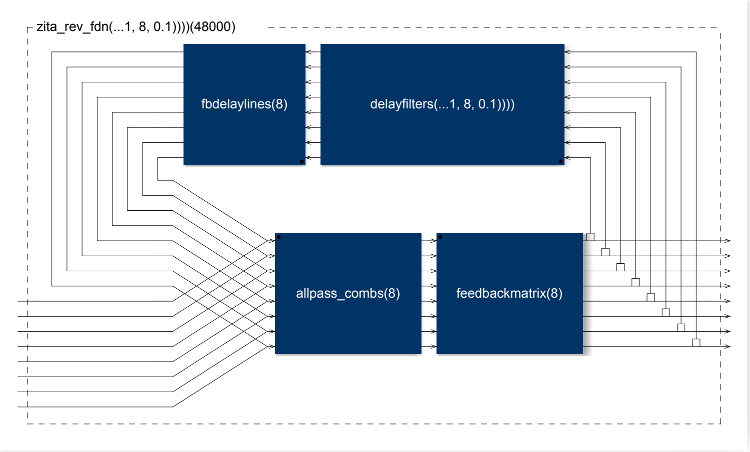


Figure: A complex but structured block-diagram

Block-Diagram Algebra

Faust syntax is based on a *block diagram algebra*

5 Composition Operators

- `(A,B)` parallel composition
- `(A:B)` sequential composition
- `(A<:B)` split composition
- `(A:>B)` merge composition
- `(A~B)` recursive composition

2 Constants

- `!` cut
- `_` wire

Block-Diagram Algebra

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.

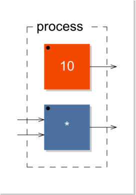


Figure: Example of parallel composition `(10,*)`

Block-Diagram Algebra
Sequential Composition

The *sequential composition* $(A : B)$ connects the outputs of A to the inputs of B. $A[0]$ is connected to $[0]B$, $A[1]$ is connected to $[1]B$, and so on.

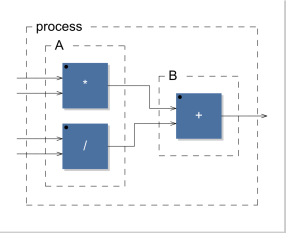


Figure: Example of sequential composition `((*,/):+)`

Block-Diagram Algebra

Split Composition

The *split composition* $(A <: B)$ operator is used to distribute A outputs to B inputs.

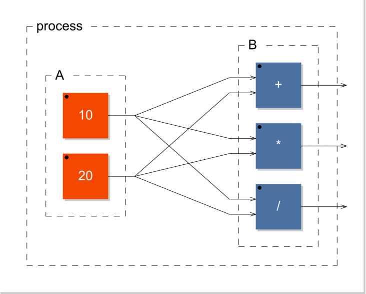


Figure: example of split composition `((10,20) <: (+,*,/))`

Block-Diagram Algebra

Merge Composition

The *merge composition* ($A :> B$) is used to connect several outputs of A to the same inputs of B.

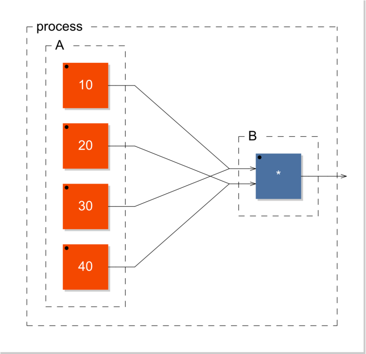


Figure: example of merge composition `((10,20,30,40) :> *)`

The *recursive composition* (A~B) is used to create cycles in the block-diagram in order to express recursive computations.

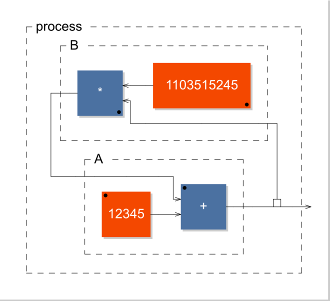


Figure: example of recursive composition +(12345) ~ *(1103515245)

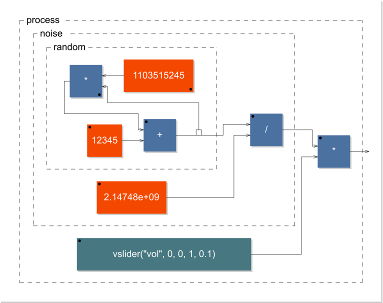
3-Some examples

Block-Diagram Algebra

Example 1

Noise Generator

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

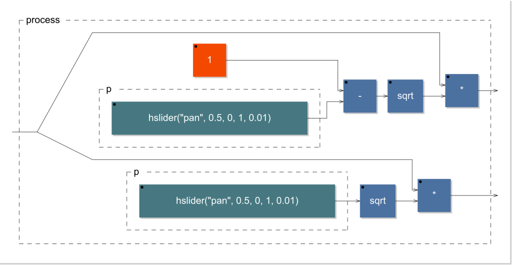


Block-Diagram Algebra

Example 2

Stereo Pan

```
p = hslider("pan", 0.5, 0, 1, 0.01);
process = _ <: *(sqrt(1 - p)), *(sqrt(p));
```



3-Primitive operations

Faust Primitives

Arithmetic operations

Syntax	**Type**	**Description**
`+`	$\mathbb{S}^2 \rightarrow \mathbb{S}^1$	addition: $y(t) = x_1(t) + x_2(t)$
`-`	$\mathbb{S}^2 \rightarrow \mathbb{S}^1$	subtraction: $y(t) = x_1(t) - x_2(t)$
`*`	$\mathbb{S}^2 \rightarrow \mathbb{S}^1$	multiplication: $y(t) = x_1(t) * x_2(t)$
$\wedge$	$\mathbb{S}^2 \rightarrow \mathbb{S}^1$	power: $y(t) = x_1(t)^{x_2(t)}$
`/`	$\mathbb{S}^2 \rightarrow \mathbb{S}^1$	division: $y(t) = x_1(t)/x_2(t)$
`%`	$\mathbb{S}^2 \rightarrow \mathbb{S}^1$	modulo: $y(t) = x_1(t)\%x_2(t)$
`int`	$\mathbb{S}^1 \rightarrow \mathbb{S}^1$	cast into an int signal: $y(t) = (int)x(t)$
`float`	$\mathbb{S}^1 \rightarrow \mathbb{S}^1$	cast into an float signal: $y(t) = (float)x(t)$

Faust Primitives

Bitwise operations

<table>
<tr><th>Syntax</th><th>Type</th><th>Description</th></tr>
<tr><td><code>&</code></td><td>$\mathbb{S}^2 \rightarrow \mathbb{S}^1$</td><td>logical AND: $y(t) = x_1(t) \& x_2(t)$</td></tr>
<tr><td><code>|</code></td><td>$\mathbb{S}^2 \rightarrow \mathbb{S}^1$</td><td>logical OR: $y(t) = x_1(t) | x_2(t)$</td></tr>
<tr><td><code>xor</code></td><td>$\mathbb{S}^2 \rightarrow \mathbb{S}^1$</td><td>logical XOR: $y(t) = x_1(t) \wedge x_2(t)$</td></tr>
<tr><td><code><<</code></td><td>$\mathbb{S}^2 \rightarrow \mathbb{S}^1$</td><td>arith. shift left: $y(t) = x_1(t) << x_2(t)$</td></tr>
<tr><td><code>>></code></td><td>$\mathbb{S}^2 \rightarrow \mathbb{S}^1$</td><td>arith. shift right: $y(t) = x_1(t) >> x_2(t)$</td></tr>
</table>

Faust Primitives

Comparison operations

Syntax	Type	Description
<	$\mathbb{S}^2 \rightarrow \mathbb{S}^1$	less than: $y(t) = x_1(t) < x_2(t)$
<=	$\mathbb{S}^2 \rightarrow \mathbb{S}^1$	less or equal: $y(t) = x_1(t) \Leftarrow x_2(t)$
>	$\mathbb{S}^2 \rightarrow \mathbb{S}^1$	greater than: $y(t) = x_1(t) > x_2(t)$
>=	$\mathbb{S}^2 \rightarrow \mathbb{S}^1$	greater or equal: $y(t) = x_1(t) >= x_2(t)$
==	$\mathbb{S}^2 \rightarrow \mathbb{S}^1$	equal: $y(t) = x_1(t) == x_2(t)$
!=	$\mathbb{S}^2 \rightarrow \mathbb{S}^1$	different: $y(t) = x_1(t)! = x_2(t)$

Faust Primitives

Trigonometric functions

Syntax	Type	Description
`acos`	$\mathbb{S}^1 \rightarrow \mathbb{S}^1$	arc cosine: $y(t) = \mathrm{acosf}(x(t))$
`asin`	$\mathbb{S}^1 \rightarrow \mathbb{S}^1$	arc sine: $y(t) = \mathrm{asinf}(x(t))$
`atan`	$\mathbb{S}^1 \rightarrow \mathbb{S}^1$	arc tangent: $y(t) = \mathrm{atanf}(x(t))$
`atan2`	$\mathbb{S}^2 \rightarrow \mathbb{S}^1$	arc tangent of 2 signals: $y(t) = \mathrm{atan2f}(x_1(t), x_2(t))$
`cos`	$\mathbb{S}^1 \rightarrow \mathbb{S}^1$	cosine: $y(t) = \mathrm{cosf}(x(t))$
`sin`	$\mathbb{S}^1 \rightarrow \mathbb{S}^1$	sine: $y(t) = \mathrm{sinf}(x(t))$
`tan`	$\mathbb{S}^1 \rightarrow \mathbb{S}^1$	tangent: $y(t) = \mathrm{tanf}(x(t))$

Faust Primitives

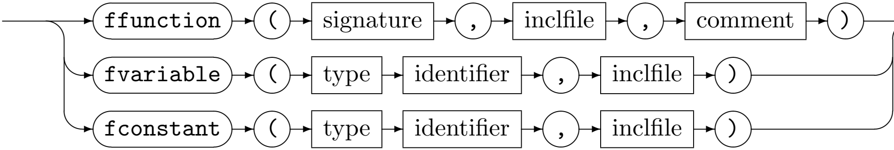
Other Math operations

Syntax	Type	Description
`exp`	$\mathbb{S}^1 \rightarrow \mathbb{S}^1$	base-e exponential: $y(t) = \mathrm{expf}(x(t))$
`log`	$\mathbb{S}^1 \rightarrow \mathbb{S}^1$	base-e logarithm: $y(t) = \mathrm{logf}(x(t))$
`log10`	$\mathbb{S}^1 \rightarrow \mathbb{S}^1$	base-10 logarithm: $y(t) = \mathrm{log10f}(x(t))$
`pow`	$\mathbb{S}^2 \rightarrow \mathbb{S}^1$	power: $y(t) = \mathrm{powf}(x_1(t), x_2(t))$
`sqrt`	$\mathbb{S}^1 \rightarrow \mathbb{S}^1$	square root: $y(t) = \mathrm{sqrtf}(x(t))$
`abs`	$\mathbb{S}^1 \rightarrow \mathbb{S}^1$	absolute value (int): $y(t) = \mathrm{abs}(x(t))$
		absolute value (float): $y(t) = \mathrm{fabsf}(x(t))$
`min`	$\mathbb{S}^2 \rightarrow \mathbb{S}^1$	minimum: $y(t) = \min(x_1(t), x_2(t))$
`max`	$\mathbb{S}^2 \rightarrow \mathbb{S}^1$	maximum: $y(t) = \max(x_1(t), x_2(t))$
`fmod`	$\mathbb{S}^2 \rightarrow \mathbb{S}^1$	float modulo: $y(t) = \mathrm{fmodf}(x_1(t), x_2(t))$
`remainder`	$\mathbb{S}^2 \rightarrow \mathbb{S}^1$	float remainder: $y(t) = \mathrm{remainderf}(x_1(t), x_2(t))$
`floor`	$\mathbb{S}^1 \rightarrow \mathbb{S}^1$	largest int $\leq$: $y(t) = \mathrm{floorf}(x(t))$
`ceil`	$\mathbb{S}^1 \rightarrow \mathbb{S}^1$	smallest int $\geq$: $y(t) = \mathrm{ceilf}(x(t))$
`rint`	$\mathbb{S}^1 \rightarrow \mathbb{S}^1$	closest int: $y(t) = \mathrm{rintf}(x(t))$

Faust Primitives

Add new ones using Foreign Functions

foreignexp



- Reference to external C *functions*, *variables* and *constants* can be introduced using the *foreign function* mechanism.
- example :

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

Faust Primitives

Delays and Tables

Syntax	Type	Description
`mem`	$\mathbb{S}^1 \rightarrow \mathbb{S}^1$	1-sample delay: $y(t+1) = x(t), y(0) = 0$
`prefix`	$\mathbb{S}^2 \rightarrow \mathbb{S}^1$	1-sample delay: $y(t+1) = x_2(t), y(0) = x_1(0)$
`@`	$\mathbb{S}^2 \rightarrow \mathbb{S}^1$	fixed delay: $y(t + x_2(t)) = x_1(t), y(t < x_2(t)) = 0$
`rdtable`	$\mathbb{S}^3 \rightarrow \mathbb{S}^1$	read-only table: $y(t) = T[r(t)]$
`rwtable`	$\mathbb{S}^5 \rightarrow \mathbb{S}^1$	read-write table: $T[w(t)] = c(t); y(t) = T[r(t)]$
`select2`	$\mathbb{S}^3 \rightarrow \mathbb{S}^1$	select between 2 signals: $T[] = \{x_0(t), x_1(t)\}; y(t) = T[s(t)]$
`select3`	$\mathbb{S}^4 \rightarrow \mathbb{S}^1$	select between 3 signals: $T[] = \{x_0(t), x_1(t), x_2(t)\}; y(t) = T$

Faust Primitives

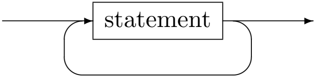
User Interface Primitives

Syntax	Example
button(*str*)	button("play")
checkbox(*str*)	checkbox("mute")
vslider(*str*, *cur*, *min*, *max*, *inc*)	vslider("vol",50,0,100,1)
hslider(*str*, *cur*, *min*, *max*, *inc*)	hslider("vol",0.5,0,1,0.01)
nentry(*str*, *cur*, *min*, *max*, *inc*)	nentry("freq",440,0,8000,1)
vgroup(*str*, *block-diagram*)	vgroup("reverb", ...)
hgroup(*str*, *block-diagram*)	hgroup("mixer", ...)
tgroup(*str*, *block-diagram*)	vgroup("parametric", ...)
vbargraph(*str*, *min*, *max*)	vbargraph("input",0,100)
hbargraph(*str*, *min*, *max*)	hbargraph("signal",0,1.0)

4-Expressions

Faust Program

program



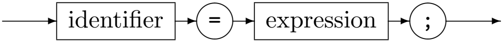
- A Faust program is essentially a list of *statements*. These statements can be :
  - metadata *declarations*,
  - file *imports*
  - *definitions*
- Example :

```
declare name        "noise";
declare copyright   "(c)GRAME␣2006";
import("music.lib");
process = noise * vslider("volume", 0, 0, 1, 0.1);
```

Definitions

Simple Definitions

definition



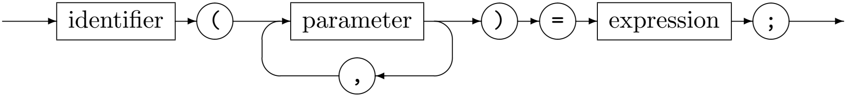
- A *definition* associates an identifier with an expression it stands for. Example :

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

Definitions

Functions' definitions

definition



- Definitions with formal parameters correspond to functions' definitions. Example :

```
linear2db(x) = 20*log10(x);
```

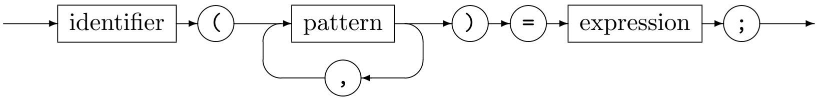
- Alternative notation using a *lambda-abstraction*:

```
linear2db = \(x).(20*log10(x));
```

Definitions

Pattern Matching Definitions

definition



- Formal parameters can also be full expressions representing patterns. Example :

```
duplicate(1,exp) = exp;
duplicate(n,exp) = exp, duplicate(n-1,exp);
```

- Alternative notation :

```
duplicate = case {
              (1,exp) => exp;
              (n,exp) => duplicate(n-1,exp);
            };
```

Statement

Import file

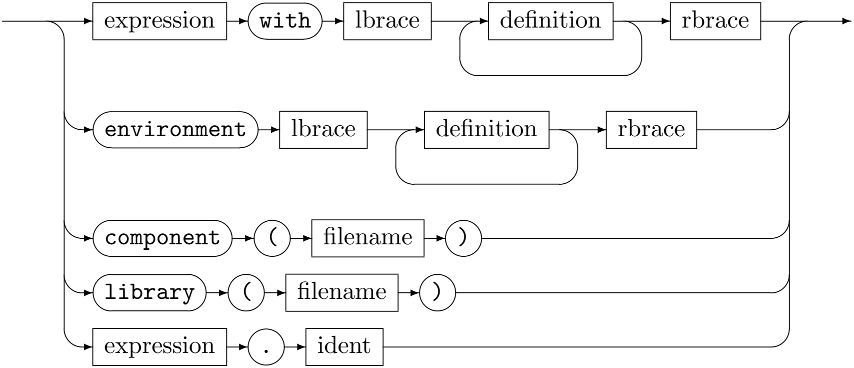
fileimport

- allows to import definitions from other source files.
- for example `import("math.lib");` imports the definitions from `"math.lib"` file, a set of additional mathematical functions provided as foreign functions.

Expressions

Environments

envexp

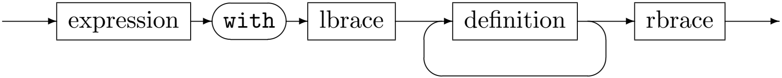


- Each Faust expression has an associated *lexical environment*

Environments

With Expression

withexpression



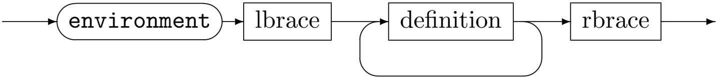
- With expression allows to specify a *local environment*, a private list of definitions that will be used to evaluate the left hand expression
- example pink noise filter :

```
pink = f : + ~ g with {
     f(x) = 0.04957526213389*x
          - 0.06305581334498*x@1
          + 0.01483220320740*x@2;
     g(x) = 1.80116083982126*x
          - 0.80257737639225*x@1;
    };
```

Environments

Environment

environment



- an `environment` is used to group together related definitions :

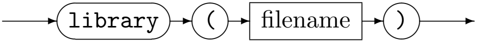
```
constant = environment {
    pi = 3.14159;
     e = 2,718 ;
     ....
   };
```

- definitions of an environment can be easily accessed : `constant.pi`

Environments

Library

library

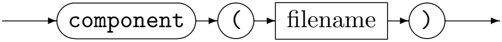


- allows to create an environment by reading the definitions from a file.
- example : `library("filter.lib")`
- definitions are accesed like this : `library("filter.lib").smooth`

Environments

Component

component



- allows to reuse a full Faust program as a simple expression.
- example :

```
component("osc.dsp")<:component("freeverb.dsp")
```

- equivalence between :

```
component("freeverb.dsp")
```

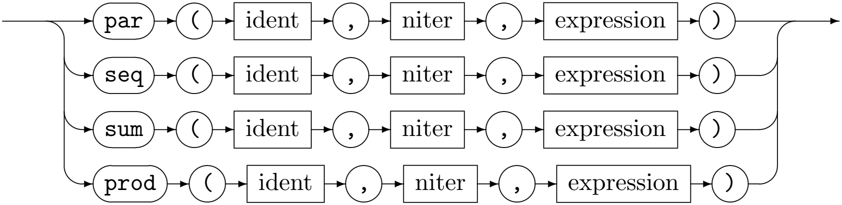
and

```
library("freeverb.dsp").process
```

Expressions

Iterations

diagiteration



- Iterations are analog to `for(...)` loops
- provide a convenient way to automate some complex block-diagram constructions.

Expressions

Iterations

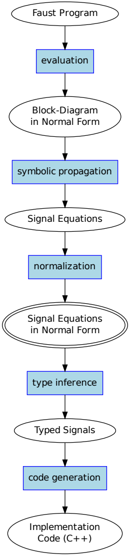
The following example shows the use of seq to create a 10-bands filter:

```
process   = seq(i, 10,
                 vgroup("band␣%i",
                     bandfilter( 1000*(1+i) )
                 )
             );
```

5-Compiler/Code Generation

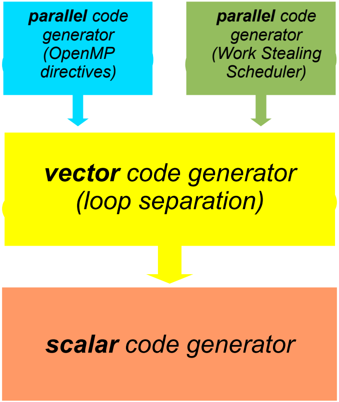
FAUST Compiler

Main Phases of the compiler



FAUST Compiler

Four Code generation modes



5-Performances

Performance of the generated code

How the C++ code generated by FAUST compares with hand written C++ code ?

STK vs FAUST (CPU load)

File name	STK	FAUST	Difference
blowBottle.dsp	3,23	2,49	-22%
blowHole.dsp	2,70	1,75	-35%
bowed.dsp	2,78	2,28	-17%
brass.dsp	10,15	2,01	-80%
clarinet.dsp	2,26	1,19	-47%
flutestk.dsp	2,16	1,13	-47%
saxophony.dsp	2,38	1,47	-38%
sitar.dsp	1,59	1,11	-30%
tibetanBowl.dsp	5,74	2,87	-50%

Overall improvement of about 41 % in favor of FAUST.

Performance of the generated code

How the C++ code generated by FAUST compares with hand written C++ code ?

STK vs FAUST (CPU load)

File name	STK	FAUST	Difference
blowBottle.dsp	3,23	2,49	-22%
blowHole.dsp	2,70	1,75	-35%
bowed.dsp	2,78	2,28	-17%
brass.dsp	10,15	2,01	-80%
clarinet.dsp	2,26	1,19	-47%
flutestk.dsp	2,16	1,13	-47%
saxophony.dsp	2,38	1,47	-38%
sitar.dsp	1,59	1,11	-30%
tibetanBowl.dsp	5,74	2,87	-50%

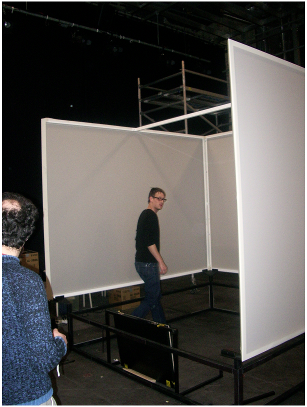
Overall improvement of about 41 % in favor of FAUST.

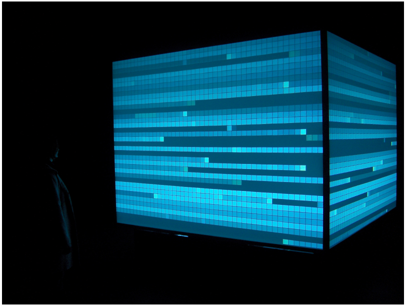
Performance of the generated code

What improvements to expect from parallelized code ?

Sonik Cube

Audio-visual installation involving a cube of light, reacting to sounds, immersed in an audio feedback room (Trafik/Orlarey 2006).



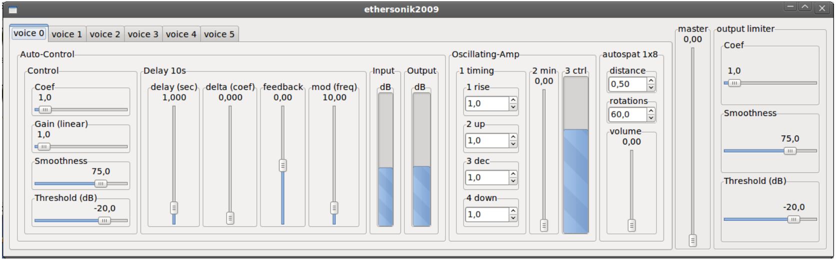


Performance of the generated code

What improvements to expect from parallelized code ?

Sonik Cube

- 8 loudspeakers
- 6 microphones
- audio software, written in FAUST, controlling the audio feedbacks and the sound spatialization.

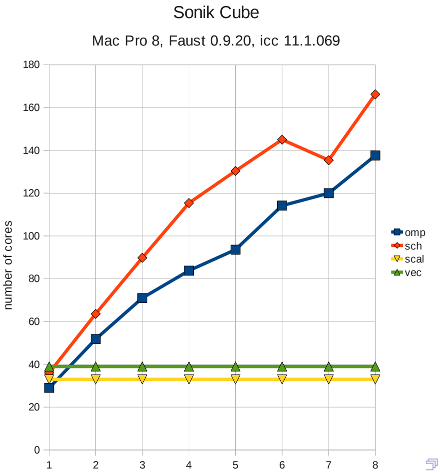


Performance of the generated code

What improvements to expect from parallelized code ?

Sonik Cube

Compared performances of the various C++ code generation strategies according to the number of cores :



6-Automatic documentation

- Binary and source code preservation of programs is not enough : quick obsolescence of languages, systems and hardware.
- We need to preserve the mathematical meaning of these programs independetly of any programming language.
- The solution is to generate automatically the mathematical description of any FAUST program

- Binary and source code preservation of programs is not enough : quick obsolescence of languages, systems and hardware.
- We need to preserve the mathematical meaning of these programs independetly of any programming language.
- The solution is to generate automatically the mathematical description of any FAUST program

- Binary and source code preservation of programs is not enough : quick obsolescence of languages, systems and hardware.
- We need to preserve the mathematical meaning of these programs independetly of any programming language.
- The solution is to generate automatically the mathematical description of any FAUST program

Automatic Mathematical Documentation

Motivations et Principles

- Binary and source code preservation of programs is not enough : quick obsolescence of languages, systems and hardware.
- We need to preserve the mathematical meaning of these programs independetly of any programming language.
- The solution is to generate automatically the mathematical description of any FAUST program

Automatic Mathematical Documentation

Tools provided

- The easiest way to generate the complete mathematical documentation is to call the `faust2mathdoc` script on a FAUST file.
- This script relies on a new option of the FAUST compile : `-mdoc`
- `faust2mathdoc noise.dsp`

Automatic Mathematical Documentation

Tools provided

- The easiest way to generate the complete mathematical documentation is to call the `faust2mathdoc` script on a FAUST file.
- This script relies on a new option of the FAUST compile : `-mdoc`
- `faust2mathdoc noise.dsp`

- The easiest way to generate the complete mathematical documentation is to call the `faust2mathdoc` script on a FAUST file.
- This script relies on a new option of the FAUST compile : `-mdoc`
- `faust2mathdoc noise.dsp`

- The easiest way to generate the complete mathematical documentation is to call the `faust2mathdoc` script on a FAUST file.
- This script relies on a new option of the FAUST compile : `-mdoc`
- `faust2mathdoc noise.dsp`

Automatic Mathematical Documentation

Files generated by `Faust2mathdoc noise.dsp`

- ▼ `noise-mdoc/`
  - ▼ `cpp/`
    - ◇ `noise.cpp`
  - ▼ `pdf/`
    - ◇ `noise.pdf`
  - ▼ `src/`
    - ◇ `math.lib`
    - ◇ `music.lib`
    - ◇ `noise.dsp`
  - ▼ `svg/`
    - ◇ `process.pdf`
    - ◇ `process.svg`
  - ▼ `tex/`
    - ◇ `noise.pdf`
    - ◇ `noise.tex`

7-Architectures

Faust Architecture System

Motivations

- Easy deployment (one Faust code, multiple audio targets) is an essential feature of the Faust project
- This is why Faust programs say nothing about audio drivers or GUI toolkits to be used.
- There is a *separation of concerns* between the audio computation itself, and its usage.

Faust Architecture System

Motivations

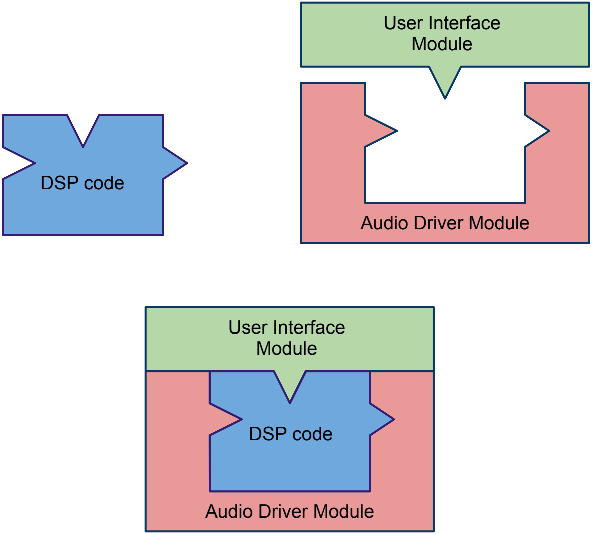
- Easy deployment (one Faust code, multiple audio targets) is an essential feature of the Faust project
- This is why Faust programs say nothing about audio drivers or GUI toolkits to be used.
- There is a *separation of concerns* between the audio computation itself, and its usage.

- Easy deployment (one Faust code, multiple audio targets) is an essential feature of the Faust project
- This is why Faust programs say nothing about audio drivers or GUI toolkits to be used.
- There is a *separation of concerns* between the audio computation itself, and its usage.

- Easy deployment (one Faust code, multiple audio targets) is an essential feature of the Faust project
- This is why Faust programs say nothing about audio drivers or GUI toolkits to be used.
- There is a *separation of concerns* between the audio computation itself, and its usage.

Faust Architecture System

The *architecture file* describes how to connect the audio computation to the external world.



Faust Architecture System

Examples of supported architectures

- Audio plugins :
  - LADSPA
  - DSSI
  - Max/MSP
  - VST
  - PD
  - CSound
  - Supercollider
  - Pure
  - Chuck
  - Octave
  - Flash
- Standalone audio applications :
  - Jack
  - Alsa
  - CoreAudio
  - iPhone

8-Multirate extension

Extensions

What is currently missing in FAUST

- Applications that we can't address :
  - oversampling, upsampling, downsampling
  - spectral processing
  - video processing
- What we need :
  - multirate signals
  - multidimension signals

Extensions

What is currently missing in FAUST

- Applications that we can't address :
  - oversampling, upsampling, downsampling
  - spectral processing
  - video processing
- What we need :
  - multirate signals
  - multidimension signals

Extensions

What is currently missing in FAUST

- Applications that we can't address :
  - oversampling, upsampling, downsampling
  - spectral processing
  - video processing
- What we need :
  - multirate signals
  - multidimension signals

Extensions

What is currently missing in FAUST

- Applications that we can't address :
  - oversampling, upsampling, downsampling
  - spectral processing
  - video processing
- What we need :
  - multirate signals
  - multidimension signals

Extensions

What is currently missing in FAUST

- Applications that we can't address :
  - oversampling, upsampling, downsampling
  - spectral processing
  - video processing
- What we need :
  - multirate signals
  - multidimension signals

Extensions

What is currently missing in Faust

- Applications that we can't address :
  - oversampling, upsampling, downsampling
  - spectral processing
  - video processing
- What we need :
  - multirate signals
  - multidimension signals

Extensions

What is currently missing in Faust

- Applications that we can't address :
  - oversampling, upsampling, downsampling
  - spectral processing
  - video processing
- What we need :
  - multirate signals
  - multidimension signals

Extensions

What is currently missing in FAUST

- Applications that we can't address :
  - oversampling, upsampling, downsampling
  - spectral processing
  - video processing
- What we need :
  - multirate signals
  - multidimension signals

Extensions

What we propose

- Minimal extension with 4 new primitives
  - Vectorize
  - Serialize
  - Concat
  - Access
- Only Vectorize and Serialize change rates (but keep the flow constant).
- All other operations assume arguments at the same rate
- All numerical operations extended to vectors, vectors of vectors, etc.

Extensions

What we propose

- Minimal extension with 4 new primitives
  - Vectorize
  - Serialize
  - Concat
  - Access
- Only Vectorize and Serialize change rates (but keep the flow constant).
- All other operations assume arguments at the same rate
- All numerical operations extended to vectors, vectors of vectors, etc.

Extensions

What we propose

- Minimal extension with 4 new primitives
  - Vectorize
  - Serialize
  - Concat
  - Access
- Only Vectorize and Serialize change rates (but keep the flow constant).
- All other operations assume arguments at the same rate
- All numerical operations extended to vectors, vectors of vectors, etc.

Extensions

What we propose

- Minimal extension with 4 new primitives
  - Vectorize
  - Serialize
  - Concat
  - Access
- Only Vectorize and Serialize change rates (but keep the flow constant).
- All other operations assume arguments at the same rate
- All numerical operations extended to vectors, vectors of vectors, etc.

Extensions

What we propose

- Minimal extension with 4 new primitives
  - Vectorize
  - Serialize
  - Concat
  - Access
- Only Vectorize and Serialize change rates (but keep the flow constant).
- All other operations assume arguments at the same rate
- All numerical operations extended to vectors, vectors of vectors, etc.

Extensions

What we propose

- Minimal extension with 4 new primitives
  - Vectorize
  - Serialize
  - Concat
  - Access
- Only Vectorize and Serialize change rates (but keep the flow constant).
- All other operations assume arguments at the same rate
- All numerical operations extended to vectors, vectors of vectors, etc.

Extensions

What we propose

- Minimal extension with 4 new primitives
  - Vectorize
  - Serialize
  - Concat
  - Access
- Only Vectorize and Serialize change rates (but keep the flow constant).
- All other operations assume arguments at the same rate
- All numerical operations extended to vectors, vectors of vectors, etc.

Extensions

What we propose

- Minimal extension with 4 new primitives
  - Vectorize
  - Serialize
  - Concat
  - Access
- Only Vectorize and Serialize change rates (but keep the flow constant).
- All other operations assume arguments at the same rate
- All numerical operations extended to vectors, vectors of vectors, etc.

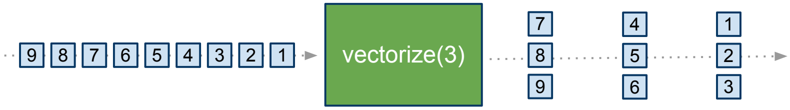
Extensions

What we propose

- Minimal extension with 4 new primitives
  - Vectorize
  - Serialize
  - Concat
  - Access
- Only Vectorize and Serialize change rates (but keep the flow constant).
- All other operations assume arguments at the same rate
- All numerical operations extended to vectors, vectors of vectors, etc.

Extensions

Vectorize



$$\text{vectorize} : T^r \times n \rightarrow [n]T^{r/n}$$

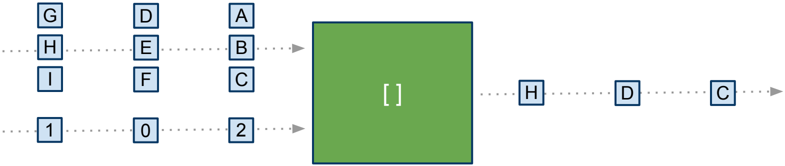
Extensions

Serialize

$$\text{serialize} : [n]\,T^{r/n} \rightarrow T^{r}$$

Extensions

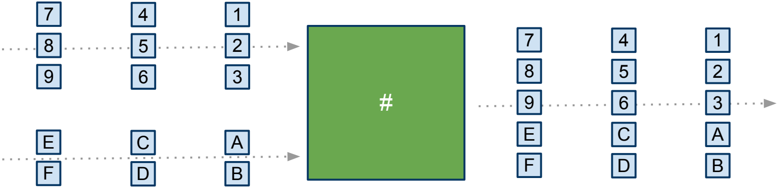
Access



$$\text{access} : [n]\,T^r \times \mathbb{N}[0..n]^r \rightarrow T^r$$

Extensions

Concat



$$\# : [n]T^r \times [m]T^r \rightarrow [n+m]T^r$$

Some very simple examples involving the multirate extension.

- upsampling : `up2 = vectorize(1) <: # : serialize;`
- downsampling : `down2 = vectorize(2) : [0];`
- sliding window :
  `slide(n) = vectorize(n) <: @(1),_ : #;`

Some very simple examples involving the multirate extension.

- upsampling : `up2 = vectorize(1) <: # : serialize;`
- downsampling : `down2 = vectorize(2) : [0];`
- sliding window :
  `slide(n) = vectorize(n) <: @(1),_ : #;`

Some very simple examples involving the multirate extension.

- upsampling : `up2 = vectorize(1) <: # : serialize;`
- downsampling : `down2 = vectorize(2) : [0];`
- sliding window :
  `slide(n) = vectorize(n) <: @(1),_ : #;`

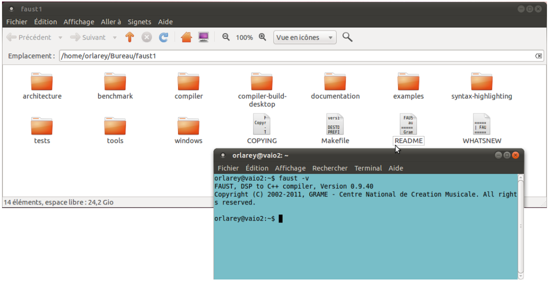
Some very simple examples involving the multirate extension.

- upsampling : `up2 = vectorize(1) <: # : serialize;`
- downsampling : `down2 = vectorize(2) : [0];`
- sliding window :
  `slide(n) = vectorize(n) <: @(1),_ : #;`

9-Resources

Resources

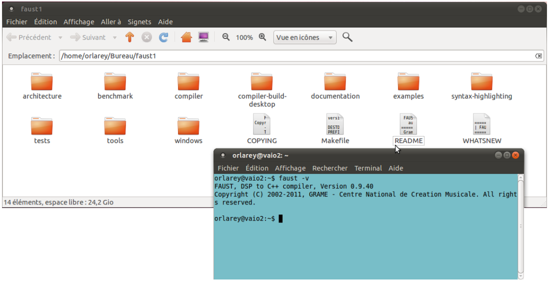
FAUST Distribution on Sourceforge



- http://sourceforge.net/projects/faudiostream/
- git clone git://faudiostream.git.sourceforge.net/gitroot/faudiostream/faudiostream faust
- cd faust; make; sudo make install

Resources

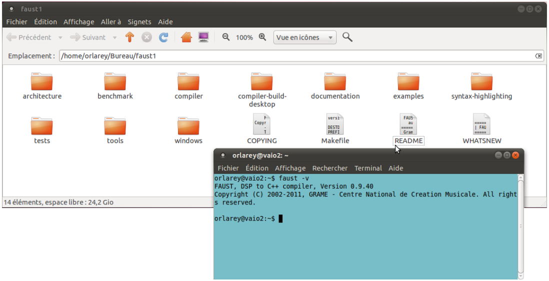
FAUST Distribution on Sourceforge



- http://sourceforge.net/projects/faudiostream/
- git clone git://faudiostream.git.sourceforge.net/gitroot/faudiostream/faudiostream faust
- cd faust; make; sudo make install

Resources

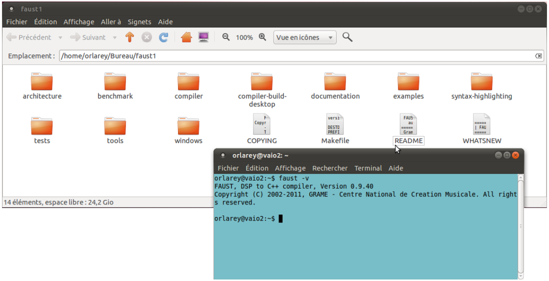
FAUST Distribution on Sourceforge



- http://sourceforge.net/projects/faudiostream/
- git clone git://faudiostream.git.sourceforge.net/gitroot/faudiostream/faudiostream faust
- cd faust; make; sudo make install

Resources

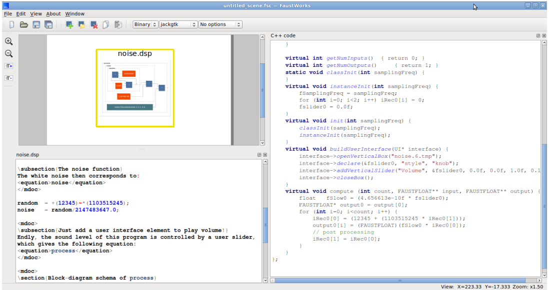
FAUST Distribution on Sourceforge



- http://sourceforge.net/projects/faudiostream/
- git clone git://faudiostream.git.sourceforge.net/gitroot/faudiostream/faudiostream faust
- cd faust; make; sudo make install

Resources

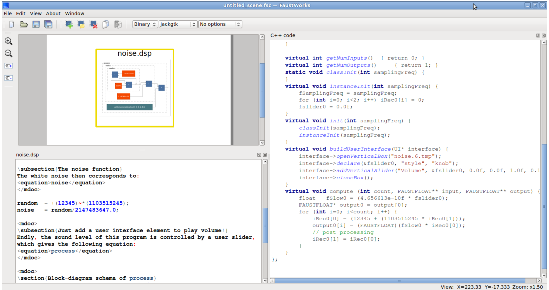
FaustWorks IDE on Sourceforge



- http://sourceforge.net/projects/faudiostream/files/FaustWorks-0.3.2.tgz/download
- git clone git://faudiostream.git.sourceforge.net/gitroot/faudiostream/FaustWorks
- cd FaustWorks; qmake; make

Resources

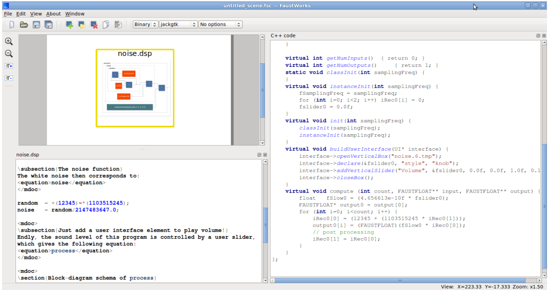
FaustWorks IDE on Sourceforge



- http://sourceforge.net/projects/faudiostream/files/FaustWorks-0.3.2.tgz/download
- git clone git://faudiostream.git.sourceforge.net/gitroot/faudiostream/FaustWorks
- cd FaustWorks; qmake; make

Resources

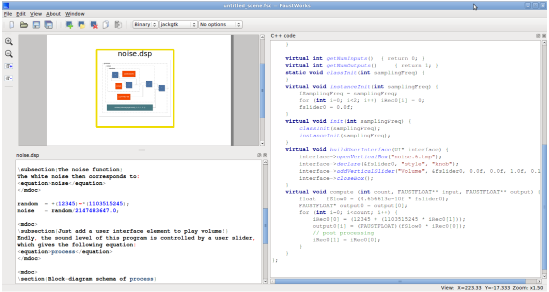
FaustWorks IDE on Sourceforge



- http://sourceforge.net/projects/faudiostream/files/FaustWorks-0.3.2.tgz/download
- git clone git://faudiostream.git.sourceforge.net/gitroot/faudiostream/FaustWorks
- cd FaustWorks; qmake; make

Resources

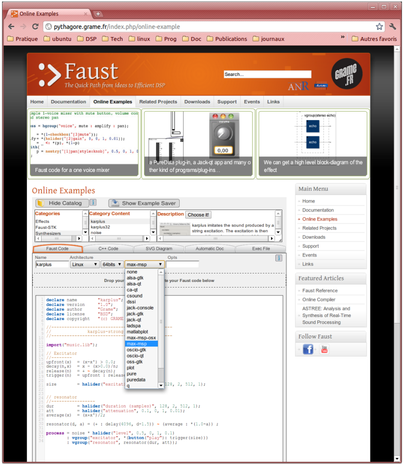
FaustWorks IDE on Sourceforge



- `http://sourceforge.net/projects/faudiostream/files/FaustWorks-0.3.2.tgz/download`
- git clone git://faudiostream.git.sourceforge.net/gitroot/faudiostream/FaustWorks
- cd FaustWorks; qmake; make

Resources

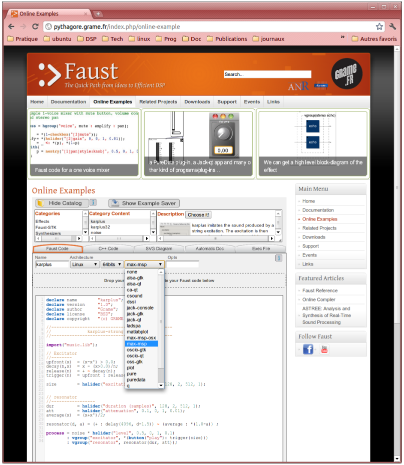
Using FAUST Online Compiler



- http://faust.grame.fr
- No installation required
- Compile to C++ as well as binary (Linux, MacOSX and Windows)

Resources

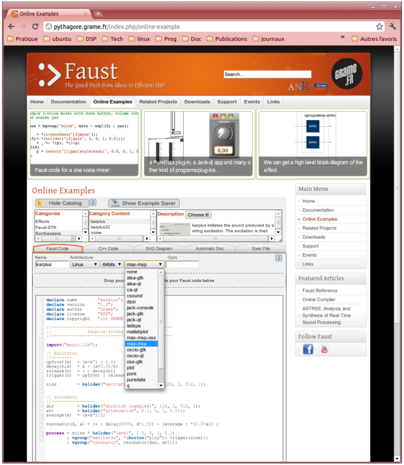
Using FAUST Online Compiler



- http://faust.grame.fr
- No installation required
- Compile to C++ as well as binary (Linux, MacOSX and Windows)

Resources

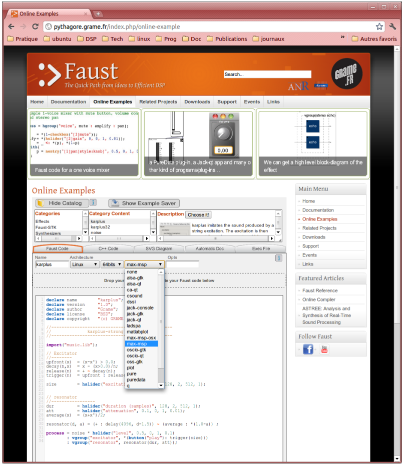
Using FAUST Online Compiler



- http://faust.grame.fr
- No installation required
- Compile to C++ as well as binary (Linux, MacOSX and Windows)

Resources

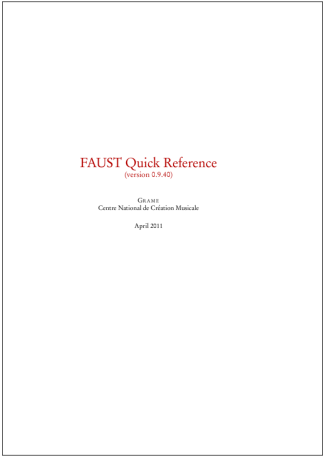
Using FAUST Online Compiler



- http://faust.grame.fr
- No installation required
- Compile to C++ as well as binary (Linux, MacOSX and Windows)

Resources

FAUST Quick Reference


FAUST Quick Reference
(version 0.9.40)

GRAME
Centre National de Création Musicale

April 2011

Figure: *Faust Quick Reference*, Grame

Resources

Some research papers

- 2004 : **Syntactical and semantical aspects of Faust**, Orlarey, Y. and Fober, D. and Letz, S., in *Soft Computing*, vol 8(9), p623-632, Springer.
- 2009 : **Parallelization of Audio Applications with Faust**, Orlarey, Y. and Fober, D. and Letz, S., in *Proceedings of the SMC 2009-6th Sound and Music Computing Conference*,
- 2011 : **Dependent vector types for data structuring in multirate Faust**, Jouvelot, P. and Orlarey, Y., in *Computer Languages, Systems & Structures*, Elsevier

10-Acknowledgments

Acknowledgments

OS Community

Fons Adriaensen, Thomas Charbonnel, Albert Gräf, Stefan Kersten, Victor Lazzarini, Kjetil Matheussen, Rémy Muller, Romain Michon, Stephen Sinclair, Travis Skare, Julius Smith

Sponsors

French Ministry of Culture, Rhône-Alpes Region, City of Lyon, National Research Agency

Partners from the ASTREE project (ANR 2008 CORD 003 02)

Jérôme Barthélemy (IRCAM), Karim Barkati (IRCAM), Alain Bonardi (IRCAM), Raffaele Ciavarella (IRCAM), Pierre Jouvelot (Mines/ParisTech), Laurent Pottier (U. Saint-Etienne)

Former Students

Tiziano Bole, Damien Cramet, Étienne Gaudrin, Matthieu Leberre, Mathieu Leroi, Nicolas Scaringella