From Music III to Faust (1/2)
A journey into audio and music DSLs

Y. Orlarey, S. Letz

GRAME – Centre National de Création Musicale

Keynotes in HPC languages, Lyon, July 2th 2013

Overview

Some Music DSLs

- 4CED
- Adagio
- Algorithmic Music Language
- AMPLE
- Arctic
- Autoklang
- Canon
- CHANT
- **Chuck**
- CLCE
- CMIX
- Cmusic
- CMUSIC
- Common Lisp Music
- Common Music
- Common Music Notation
- **Csound**
- CyberBand
- DARMS
- DCMP
- DMIX
- **Elody**
- EsAC
- EUTERPE
- **Faust**
- Flavors Band
- FOIL
- FORMES
- FORMULA
- Fugue
- GROOVE
- GUIDO
- HARP
- Haskore
- HMSL
- INV
- invokator
- KERN
- Keynote
- Kyma
- LOCO
- LPC
- Mars
- MASC
- **Max**
- MidiLisp
- MidiLogo
- MODE
- MOM
- Moxc
- MSX
- MUS10
- MUS8
- MUSCMP
- MuseData
- MusES
- MUSIC 10
- MUSIC 11
- MUSIC 360
- MUSIC 4B
- MUSIC 4BF
- MUSIC 4F ORPHEUS
- MUSIC 6
- Music Composition Language
- **MUSIC III/IV/V**
- MusicLogo
- Music1000
- MUSIC7
- Musictex
- MUSIGOL
- MusicXML
- Musixtex
- NIFF
- NOTELIST
- Nyquist
- OPAL
- OpenMusic
- Organum1
- Outperform
- PE
- Patchwork
- PILE
- Pla
- PLACOMP
- PLAY1
- PLAY2
- PMX
- POCO
- POD6
- POD7
- PROD
- **Puredata**
- Ravel
- SALIERI
- SCORE
- ScoreFile
- SCRIPT
- SIREN
- SMDL
- SMOKE
- SSP
- SSSP
- ST
- **Supercollider**
- Symbolic Composer

Digital Sound Synthesis

1957, first experiments

- First experiments in digital sound synthesis in 1957 by Max Mathews and colleagues at Bell Labs
- Sounds computed on an IBM 704 at IBM headquarters in New York and stored on a digital tape
- Digital tape played back at Bell Labs in Murray Hill using a unique 12-bits "digital-to-sound" converter

Digital Sound Synthesis

First Tools, Music I and Music II

- 1957 : Music I (single triangle waveform generator, written by M. Mathews for the IBM 704)
- "In a Silver Scale" a piece by Newman Guttman written with Music I in 1957
- 1958 : Music II (4 voices, 16 waveforms, written by M. Mathews for the IBM 7094)
- "Pitch Variations" a piece by Newman Guttman written with Music II in 1958 (play)

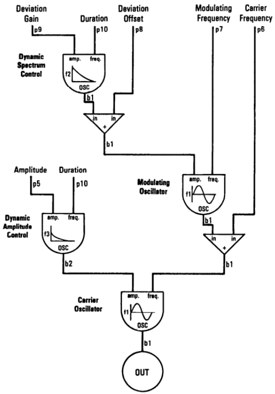
Digital Sound Synthesis

First Languages, Music III/IV/V

- 1960 : Music III introduces the concept of Unit Generators
- 1963 : Music IV, a port of Music III using a macro assembler
- 1968 : Music V written in Fortran (inner loops of UG in assembler)

```
ins 0 FM;
osc bl p9 p10 f2 d;
adn bl bl p8;
osc bl bl p7 fl d;
adn bl bl p6;
osc b2 p5 p10 f3 d;
osc bl b2 bl fl d;
out bl;
```

FM synthesis coded in CMusic



Csound

Originally developed by Barry Vercoe in 1985, Csound is today ”a sound design, music synthesis and signal processing system, providing facilities for composition and performance over a wide range of platforms.” (see http://www.csounds.com)

```
      instr 2
a1    oscil     p4, p5, 1   ; p4=amp
      out       a1          ; p5=freq
      endin
```

Example of Csound instrument

```
f1   0    4096 10 1        ; sine wave

;ins strt dur  amp(p4)    freq(p5)
i2   0    1    2000       880
i2   1.5  1    4000       440
i2   3    1    8000       220
i2   4.5  1    16000      110
i2   6    1    32000      55

e
```

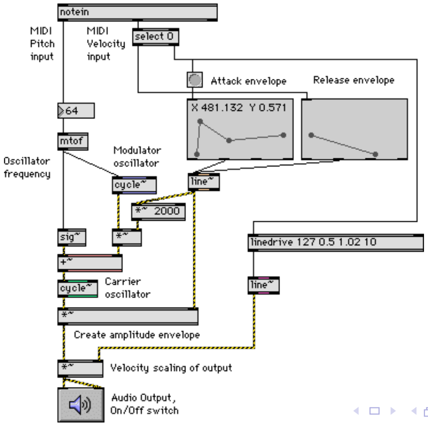
Example of Csound score

Supercollider

SuperCollider (John McCarthy, 1986) is an open source environment and programming language for real time audio synthesis and algorithmic composition. It provides an interpreted object-oriented language which functions as a network client to a state of the art, realtime sound synthesis server. (see http://supercollider.sourceforge.net/)

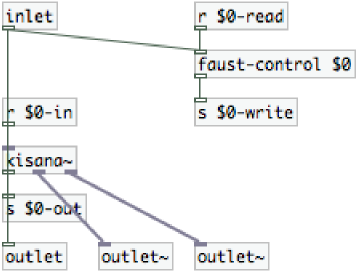
Max

Max (Miller Puckette, 1987), is visual programming language for real time audio synthesis and algorithmic composition with multimedia capabilities. It is named Max in honor of Max Mathews. It was initially developed at IRCAM. Since 1999 Max has been developed and commercialized by Cycling74. (see http://cycling74.com/)



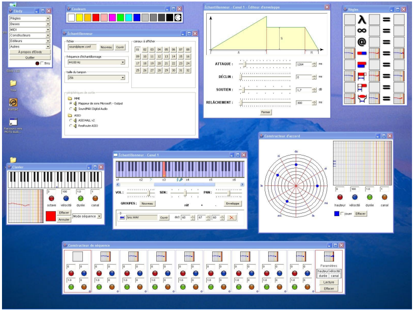
Puredata

Pure Data (Miller Puckette 1996) is an open source visual programming language of the Max family. "Pd enables musicians, visual artists, performers, researchers, and developers to create software graphically, without writing lines of code". (see http://puredata.info/)



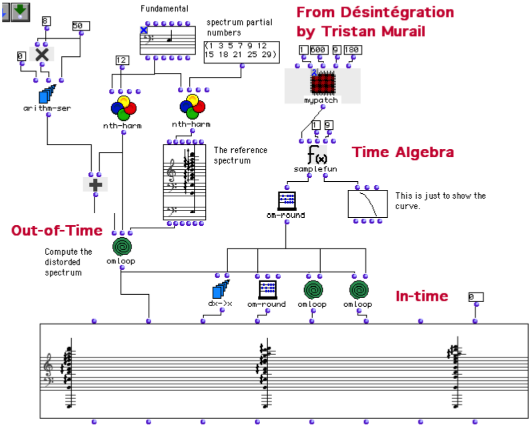
Elody

Elody (Fober, Letz, Orlarey, 1997) is a music composition environment developed in Java. The heart of Elody is a visual functional language derived from lambda-calculus. The languages expressions are handled through visual constructors and Drag and Drop actions allowing the user to play in realtime with the language.



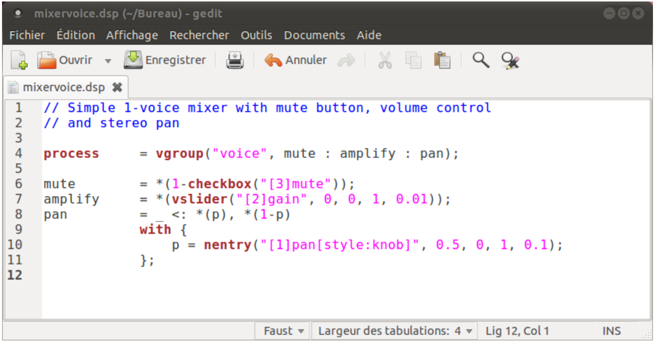
OpenMusic

OpenMusic (Agon et al. 1998) is a music composition environment based on Common Lisp. It introduces a powerful visual syntax to Lisp and provides composers with a large number of composition tools and libraries.



Faust

Faust (Orlarey et al. 2002) is a programming language that provides a purely functional approach to signal processing while offering a high level of performance. FAUST offers a viable and efficient alternative to C/C++ to develop audio processing libraries, audio plug-ins or standalone applications.



ChucK

ChucK (Ge Wang, Perry Cook 2003) is a concurrent, on-the-fly, audio programming language. It offers a powerful and flexible programming tool for building and experimenting with complex audio synthesis programs, and real-time interactive control. (see http://chuck.cs.princeton.edu)

```
// make our patch
SinOsc s => dac;

// time-loop, in which the osc's frequency
// is changed every 100 ms
while( true ) {
   100::ms => now;
   Std.rand2f(30.0, 1000.0) => s.freq;
}
```

Reactable

The Reactable is a tangible programmable synthesizer. It was conceived in 2003 by Sergi Jordà, Martin Kaltenbrunner, Günter Geiger and Marcos Alonso at the Pompeu Fabra University in Barcelona.

From Music III to Faust (2/2)
A journey into audio and music DSLs

Y. Orlarey, S. Letz

GRAME – Centre National de Création Musicale

Keynotes in HPC languages, Lyon, July 2th 2013

1-Introduction

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 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

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 :
  - $\{ :\ ,\ <:\ :>\ \tilde{}\ \} \subset \mathbb{P} \times \mathbb{P} \rightarrow \mathbb{P}$

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 :
  - $\{ \texttt{:} \;\; \texttt{,} \;\; \texttt{<:} \;\; \texttt{:>} \;\; \texttt{\~{}} \} \subset \mathbb{P} \times \mathbb{P} \rightarrow \mathbb{P}$

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 :
  - $\{$`:` `,` `<:` `:>` `~`$\} \subset \mathbb{P} \times \mathbb{P} \rightarrow \mathbb{P}$

Introduction

Example of signal processor

GRAME CENTRE NATIONAL DE CRÉATION MUSICALE

- 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

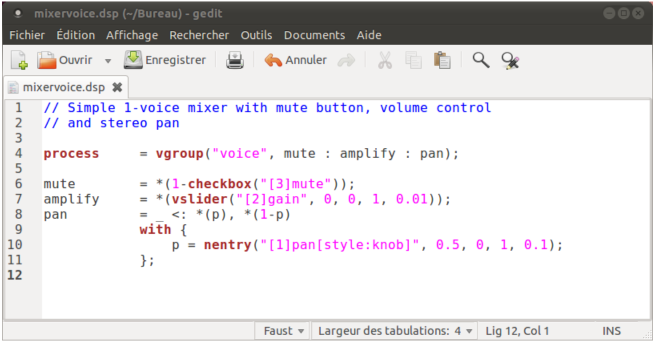


Figure: Source code of a simple 1-voice mixer

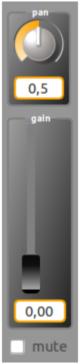


Figure: Resulting application

Introduction

Main caracteristics

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)

Introduction

Main caracteristics

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)

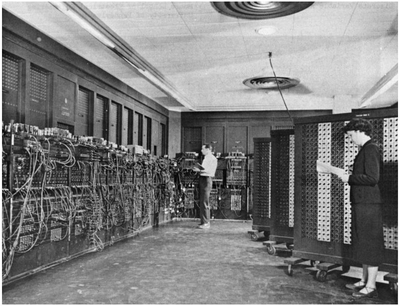
FAUST is based on several design principles:

- 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:

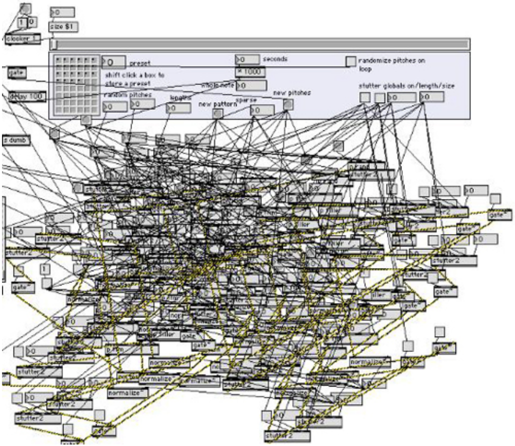


Figure: Block-diagrams can be a mess

Block-Diagram Algebra

Faust allows structured block-diagrams

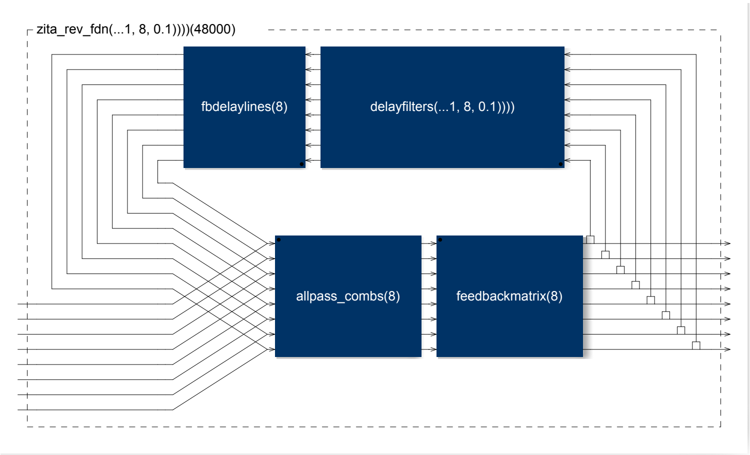


Figure: A complex but structured block-diagram

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

The *parallel composition* (A, B) is probably the simplest one. It places the two block-diagrams one on top of the other, without connections.

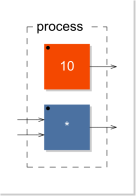


Figure: Example of parallel composition `(10,*)`

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.

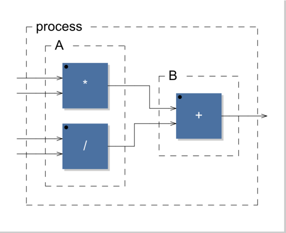


Figure: Example of sequential composition `((*,/):+)`

Block-Diagram Algebra

Split Composition

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

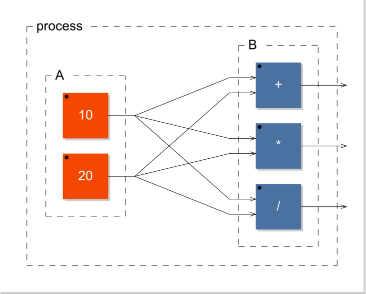


Figure: example of split composition ((10,20) <: (+,*,/))

Block-Diagram Algebra

Merge Composition

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

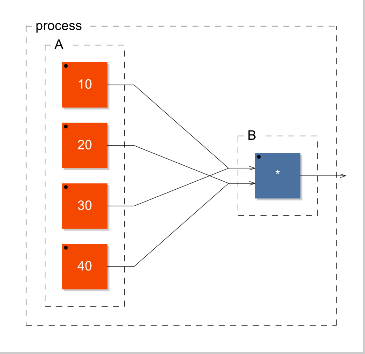


Figure: example of merge composition `((10,20,30,40) :> *)`

Block-Diagram Algebra

Recursive Composition

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

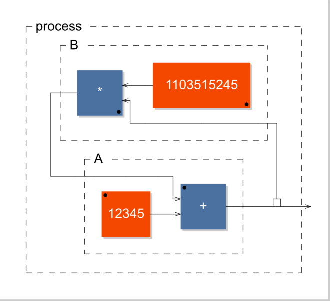


Figure: example of recursive composition `+(12345) ~ *(1103515245)`

3-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: $y(t) = x_1(t) * x_2(t)$
`∧`	$\mathbb{S}^2 \rightarrow \mathbb{S}^1$	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

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

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

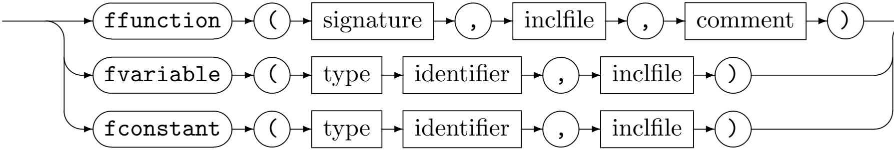
Other Math operations

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-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.

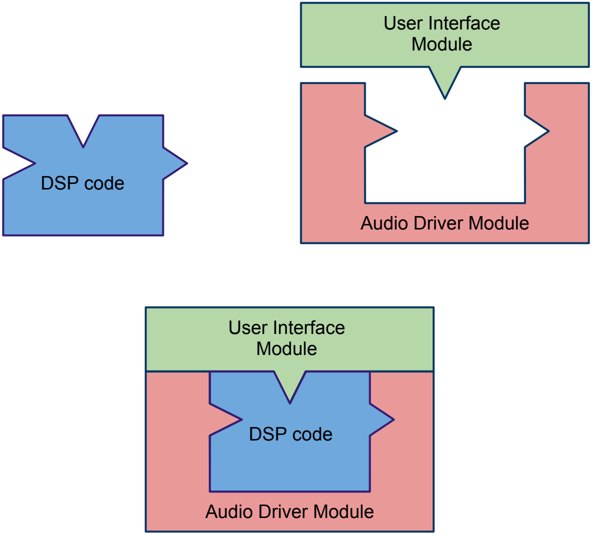
- Easy deployment (one Faust code, multiple audio targets) is an essential feature of the Faust project
- This is why Faust programs say nothing about audio drivers or GUI toolkits to be used.
- There is a *separation of concerns* between the audio computation itself, and its usage.

- 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

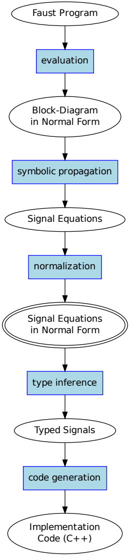
Examples of supported architectures

- Audio plugins :
  - LADSPA
  - DSSI
  - LV2
  - Max/MSP
  - VST
  - PD
  - CSound
  - Supercollider
  - Pure
  - Chuck
  - Octave
  - Flash
- Audio drivers :
  - Jack
  - Alsa
  - CoreAudio
- Graphic User Interfaces :
  - QT
  - GTK
  - iOS5
- Other User Interfaces :
  - OSC
  - HTTPD

5-Compiler/Code Generation

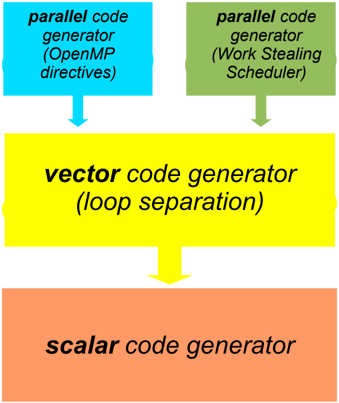
FAUST Compiler

Main Phases of the compiler



FAUST Compiler

Four Code generation modes



6-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%

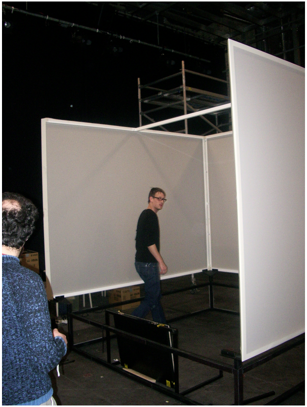
Overall improvement of about 41 % in favor of FAUST.

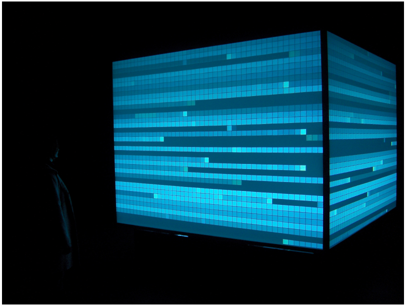
Performance of the generated code

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).



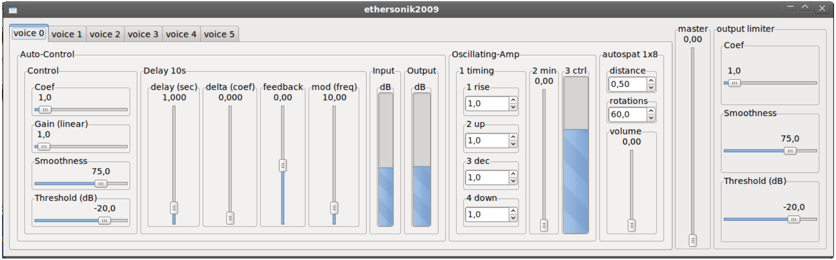


Performance of the generated code

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.

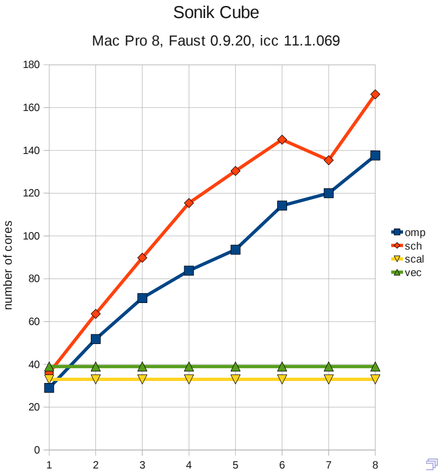


Performance of the generated code

What improvements to expect from parallelized code ?

Sonik Cube

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



7-DocumentationPreservation

- 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 independently 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 independently 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 independently 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 independently 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`

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
  ```
  ```

Automatic Mathematical Documentation

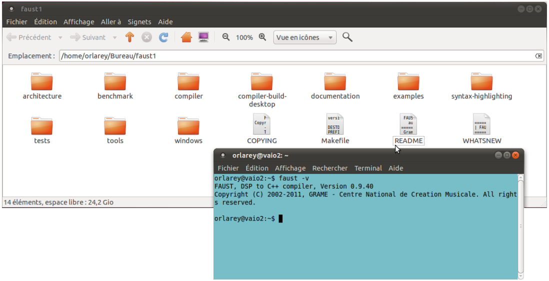
Files generated by `Faust2mathdoc noise.dsp`

- ▼ `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`

8-Resources

Resources

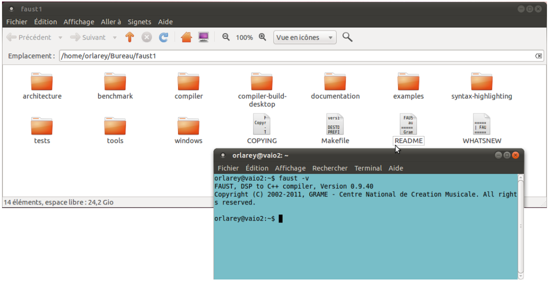
FAUST Distribution on Sourceforge



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

Resources

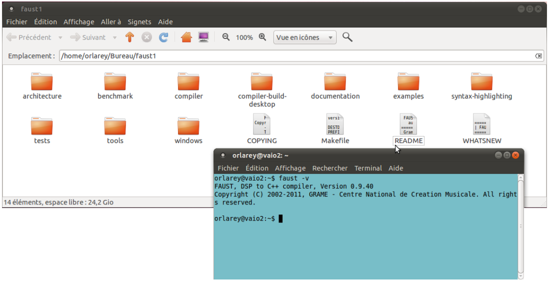
FAUST Distribution on Sourceforge



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

Resources

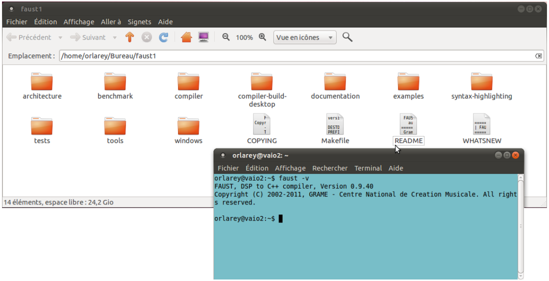
FAUST Distribution on Sourceforge



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

Resources

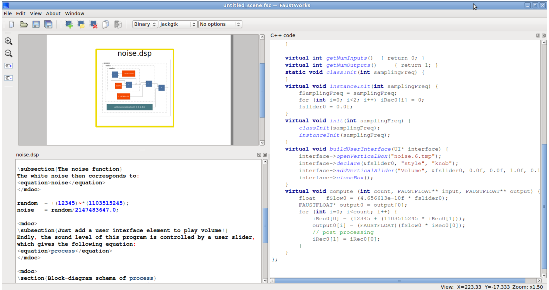
FAUST Distribution on Sourceforge



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

Resources

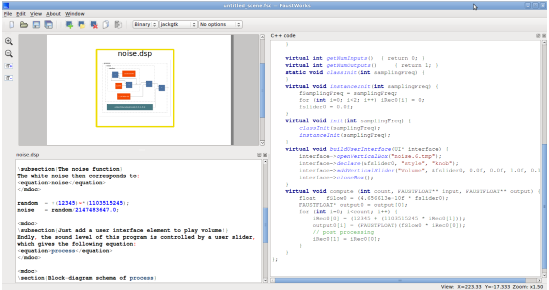
FaustWorks IDE on Sourceforge



- 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

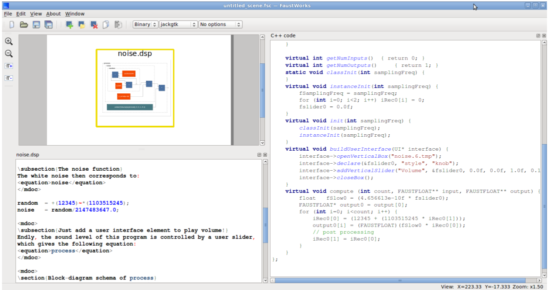
FaustWorks IDE on Sourceforge



- 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

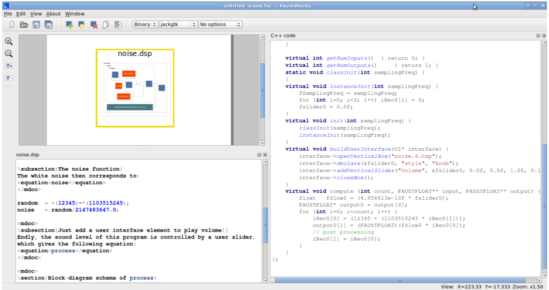
FaustWorks IDE on Sourceforge



- 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

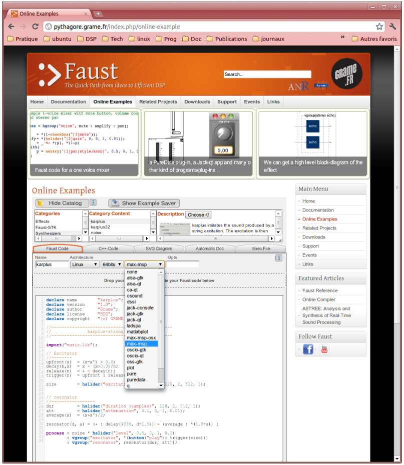
FaustWorks IDE on Sourceforge



- 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

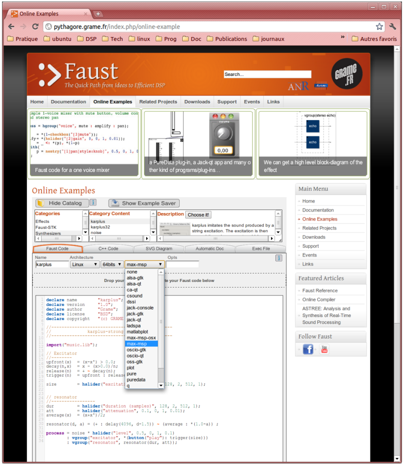
Using FAUST Online Compiler



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

Resources

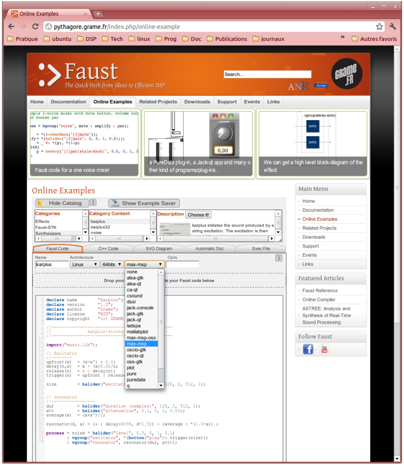
Using FAUST Online Compiler



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

Resources

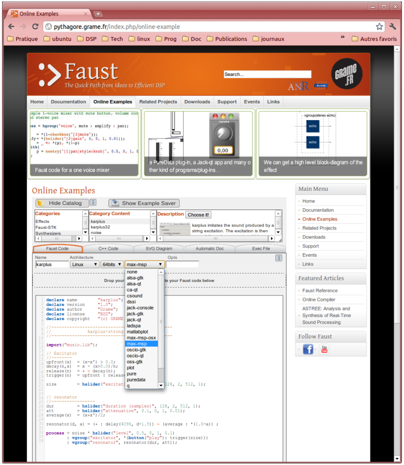
Using FAUST Online Compiler



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

Resources

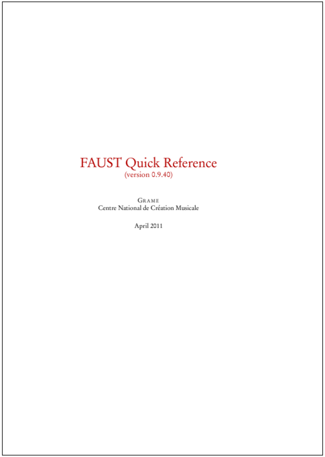
Using FAUST Online Compiler



- 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

9-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