FAUST: Functional AUdio STream

Y. Orlarey*, S. Letz*, P. Jouvelot†, D. Fober*, K. Barkati*

(*)GRAME – Centre National de Création Musicale
(+)CRI – Mines ParisTech

Synchron 2011, November 30, 2011
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.
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.
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
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
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
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
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
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
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
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
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
Introduction

What is a FAUST program?

A FAUST program describes a signal processor:

- A signal processor is a signals to signals function
- A signal is a time to samples function
- Everything in FAUST is a signal processor
- Programming in FAUST is essentially combining signal processors
Introduction

What is a FAUST program?

A FAUST program describes a *signal processor*:

- A *signal processor* is a *signals to signals* function
- A *signal* is a *time to samples* function
- Everything in FAUST is a *signal processor*
- Programming in FAUST is essentially combining signal processors
Introduction

What is a FAUST program?

A FAUST program describes a *signal processor*:

- A *signal processor* is a *signals to signals* function
- A *signal* is a *time to samples* function
- Everything in FAUST is a *signal processor*
- Programming in FAUST is essentially combining signal processors
Introduction

What is a FAUST program?

A FAUST program describes a *signal processor*:

- A *signal processor* is a *signals to signals* function
- A *signal* is a *time to samples* function
- Everything in FAUST is a *signal processor*
- Programming in FAUST is essentially combining signal processors
Introduction

What is a FAUST program?

A FAUST program describes a \textit{signal processor}:

- A \textit{signal processor} is a \textit{signals} to \textit{signals} function.
- A \textit{signal} is a \textit{time} to \textit{samples} function.
- Everything in FAUST is a \textit{signal processor}.
- Programming in FAUST is essentially combining signal processors.
Introduction
What is a FAUST program?

A FAUST program describes a *signal processor*:

- A *signal processor* is a *signals to signals* function
- A *signal* is a *time to samples* function
- Everything in FAUST is a *signal processor*
- Programming in FAUST is essentially combining signal processors
Introduction
A simple FAUST program

Figure: Source code of a simple 1-voice mixer

```plaintext
// 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: Resulting application
Introduction

Main characteristics

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
- 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 characteristics

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
- 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 characteristics

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
- 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
- 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 characteristics

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
  - 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 characteristics

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
- 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 characteristics

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
- 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 characteristics

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
- 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
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 \(\((*,/):+\)\)
The *split composition* \((A <: B)\) operator is used to distribute \(A\) outputs to \(B\) inputs.

**Figure:** example of split composition \(((10, 20) <: (+, *, /))\)
The *merge composition* \((A :\rightarrow B)\) is used to connect several outputs of \(A\) to the same inputs of \(B\).

**Figure:** example of merge composition \(((10,20,30,40) :\rightarrow *)\)
Block-Diagram Algebra

Recursive Composition

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

Figure: example of recursive composition \(+ \langle 12345 \rangle \sim * \langle 1103515245 \rangle\)
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);
Stereo Pan

\[ p = \text{hslider}(\text{"pan"}, \ 0.5, \ 0, \ 1, \ 0.01); \]
\[ \text{process} = _ <: \ast(\sqrt{1 - p}), \ast(\sqrt{p}); \]
4-Demo
5-Compiler/Code Generation
FAUST Compiler
Main Phases of the compiler

1. Faust Program
2. Evaluation
3. Block-Diagram in Normal Form
4. Symbolic Propagation
5. Signal Equations
6. Normalization
7. Signal Equations in Normal Form
8. Type Inference
9. Typed Signals
10. Code Generation
11. Implementation Code (C++)
FAUST Compiler

Four Code generation modes

- **parallel code generator (OpenMP directives)**
- **parallel code generator (Work Stealing Scheduler)**
- **vector code generator (loop separation)**
- **scalar code generator**
Performance of the generated code
How the C++ code generated by FAUST compares with hand written C++ code?

STK vs FAUST (CPU load)

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

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)

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

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:

![Graph showing performance comparison of parallelization strategies](image-url)
6-Automatic documentation
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 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
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 independently of any programming language.
- The solution is to generate automatically the mathematical description of any \texttt{Faust} program.
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`

```bash
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:

```
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

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
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.
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.
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

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

\[
serialize : [n] T^r/n \rightarrow T^r
\]
access : $[n] T^r \times \mathbb{N}[0..n]^r \rightarrow T^r$
Extensions

Concat

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

Simple examples

Some very simple examples involving the multirate extension.

- **upsampling**: \( \text{up}_2 = \text{vectorize}(1) <: \# : \text{serialize} \);
- **downsampling**: \( \text{down}_2 = \text{vectorize}(2) : [0] \);
- **sliding window**: 
  \[ \text{slide}(n) = \text{vectorize}(n) <: @(1),_ : \#; \]
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**: \( \text{up2} = \text{vectorize}(1) <: \# : \text{serialize}; \)
- **downsampling**: \( \text{down2} = \text{vectorize}(2) : [0]; \)
- **sliding window**:
  \( \text{slide}(n) = \text{vectorize}(n) <: @1, _ : \#; \)
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/](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/](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](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](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](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](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

- No installation required
- Compile to C++ as well as binary (for Linux and Windows)
Resources

Using FAUST Online Compiler

- No installation required
- Compile to C++ as well as binary (for Linux and Windows)
Resources
Using FAUST Online Compiler

- [http://faust.grame.fr](http://faust.grame.fr)
- No installation required
- Compile to C++ as well as binary (for Linux and Windows)
Resources
Using FAUST Online Compiler

- [http://faust.grame.fr](http://faust.grame.fr)
- No installation required
- Compile to C++ as well as binary (for Linux and Windows)
Resources
FAUST Quick Reference

Figure: Faust Quick Reference, Grame
Resources
Some research papers


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