What’s new in Faust

Y. Orlarey, K. Barkati, D. Fober, S. Letz

GRAME
Centre National de Création Musicale

LAC, May 6, 2011
ASTREE, ANR-08-CORD-003 02
## Brief introduction to FAUST

### Main characteristics

- **Faust Stands for** *Functional AUdio STream*
- It’s a Synchronous, Functional, Domain Specific Language for signal processing
- A Faust program describes a *signal processor*
- It’s a compiled language (to C++)
- Various code back-ends (including parallelization)
- It works at sample level
- Constant memory and CPU footprint
Brief introduction to FAUST

Main characteristics

- Faust Stands for Functional Audio Stream
- It’s a Synchronous, Functional, Domain Specific Language for signal processing
- A Faust program describes a signal processor
- It’s a compiled language (to C++)
- Various code back-ends (including parallelization)
- It works at sample level
- Constant memory and CPU footprint
Brief introduction to FAUST

Main characteristics

- Faust Stands for *Functional Audio Stream*
- It’s a Synchronous, Functional, Domain Specific Language for signal processing
- A Faust program describes a *signal processor*
- It’s a compiled language (to C++)
- Various code back-ends (including parallelization)
- It works at sample level
- Constant memory and CPU footprint
Main characteristics

- Faust Stands for *Functional Audio Stream*
- It’s a Synchronous, Functional, Domain Specific Language for signal processing
  - A Faust program describes a *signal processor*
  - It’s a compiled language (to C++)
  - Various code back-ends (including parallelization)
  - It works at sample level
  - Constant memory and CPU footprint
Brief introduction to FAUST

Main characteristics

- Faust Stands for Functional Audio Stream
- It’s a Synchronous, Functional, Domain Specific Language for signal processing
- A Faust program describes a signal processor
- It’s a compiled language (to C++)
- Various code back-ends (including parallelization)
- It works at sample level
- Constant memory and CPU footprint
Brief introduction to FAUST

Main characteristics

- Faust Stands for *Functional Audio Stream*
- It’s a Synchronous, Functional, Domain Specific Language for signal processing
- A Faust program describes a *signal processor*
- It’s a compiled language (to C++)
- Various code back-ends (including parallelization)
- It works at sample level
- Constant memory and CPU footprint
Brief introduction to FAUST

Main characteristics

- Faust Stands for *Functional Audio Stream*
- It’s a Synchronous, Functional, Domain Specific Language for signal processing
- A Faust program describes a *signal processor*
- It’s a compiled language (to C++)
- Various code back-ends (including parallelization)
- It works at sample level
- Constant memory and CPU footprint
Brief introduction to FAUST

Main characteristics

- Faust Stands for *Functional AUdio STream*
- It’s a Synchronous, Functional, Domain Specific Language for signal processing
- A Faust program describes a *signal processor*
- It’s a compiled language (to C++)
- Various code back-ends (including parallelization)
- It works at sample level
- Constant memory and CPU footprint
Main characteristics

- Faust Stands for Functional Audio Stream
- It's a Synchronous, Functional, Domain Specific Language for signal processing
- A Faust program describes a signal processor
- It's a compiled language (to C++)
- Various code back-ends (including parallelization)
- It works at sample level
- Constant memory and CPU footprint
Brief introduction to FAUST

Design Principles

- **Specification Language**: The user gives the DSP specification, the compiler generates the most appropriate implementation.
- **Block-Diagram syntax**: A textual block-diagram language based on function composition.
- **Strong formal basis**: A language with a simple and well-defined formal semantic.
- **Functional approach**: A purely functional programming language for real-time signal processing.
- **Efficient compiled code**: It should be an alternative to C. The generated C++ code should compete with hand-written code.
- **Easy deployment**: Multiple native implementations from a single Faust program.
Brief introduction to FAUST

Design Principles

- Specification Language: The user gives the DSP specification, the compiler generates the most appropriate implementation.
- Block-Diagram syntax: A textual block-diagram language based on function composition.
- Strong formal basis: A language with a simple and well-defined formal semantic.
- Functional approach: A purely functional programming language for real-time signal processing.
- Efficient compiled code: It should be an alternative to C. The generated C++ code should compete with hand-written code.
- Easy deployment: Multiple native implementations from a single Faust program.
### Design Principles

- **Specification Language**: The user gives the DSP specification, the compiler generates the most appropriate implementation.
- **Block-Diagram syntax**: A textual block-diagram language based on function composition.
- **Strong formal basis**: A language with a simple and well-defined formal semantic.
- **Functional approach**: A purely functional programming language for real-time signal processing.
- **Efficient compiled code**: It should be an alternative to C. The generated C++ code should compete with hand-written code.
- **Easy deployment**: Multiple native implementations from a single Faust program.
**Design Principles**

- *Specification Language*: The user gives the DSP specification, the compiler generates the most appropriate implementation.

- *Block-Diagram syntax*: A textual block-diagram language based on function composition.

- *Strong formal basis*: A language with a simple and well-defined formal semantic.


- *Efficient compiled code*: It should be an alternative to C. The generated C++ code should compete with hand-written code.

- *Easy deployment*: Multiple native implementations from a single Faust program.
Brief introduction to FAUST

Design Principles

- **Specification Language**: The user gives the DSP specification, the compiler generates the most appropriate implementation.
- **Block-Diagram syntax**: A textual block-diagram language based on function composition.
- **Strong formal basis**: A language with a simple and well-defined formal semantic.
- **Functional approach**: A purely functional programming language for real-time signal processing.
- **Efficient compiled code**: It should be an alternative to C. The generated C++ code should compete with hand-written code.
- **Easy deployment**: Multiple native implementations from a single Faust program.
Design Principles

- **Specification Language**: The user gives the DSP specification, the compiler generates the most appropriate implementation.
- **Block-Diagram syntax**: A textual block-diagram language based on function composition.
- **Strong formal basis**: A language with a simple and well-defined formal semantic.
- **Functional approach**: A purely functional programming language for real-time signal processing.
- **Efficient compiled code**: It should be an alternative to C. The generated C++ code should compete with hand-written code.
- **Easy deployment**: Multiple native implementations from a single Faust program.
**Design Principles**

- **Specification Language**: The user gives the DSP specification, the compiler generates the most appropriate implementation.
- **Block-Diagram syntax**: A textual block-diagram language based on function composition.
- **Strong formal basis**: A language with a simple and well-defined formal semantic.
- **Functional approach**: A purely functional programming language for real-time signal processing.
- **Efficient compiled code**: It should be an alternative to C. The generated C++ code should compete with hand-written code.
- **Easy deployment**: Multiple native implementations from a single Faust program.
Design Principles

- **Specification Language**: The user gives the DSP specification, the compiler generates the most appropriate implementation.
- **Block-Diagram syntax**: A textual block-diagram language based on function composition.
- **Strong formal basis**: A language with a simple and well-defined formal semantic.
- **Functional approach**: A purely functional programming language for real-time signal processing.
- **Efficient compiled code**: It should be an alternative to C. The generated C++ code should compete with hand-written code.
- **Easy deployment**: Multiple native implementations from a single Faust program.
Demo 1: Simple Sound Generator
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
- \((A \sim B)\)  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.
Block-Diagram Algebra

Sequential Composition

The \textit{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.

\textbf{Figure:} Example of sequential composition \(((\ast,/) : +)\)
Block-Diagram Algebra

Split Composition

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

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

**Figure:** example of recursive composition \(+ (12345) \sim *(1103515245)\)
Block-Diagram Algebra

Same expression in Lambda-Calculus, FP and Faust

Lambda-Calculus
\( \lambda x. \lambda y. (x+y,x*y) \text{ 2 3} \)

FP/FL (John Backus)
[+,\*]:<2,3>

Faust
2,3 <: +,\*
Block-Diagram Algebra

Same expression in Lambda-Calculus, FP and Faust

Lambda-Calculus
\( \lambda x. \lambda y. (x+y, x*y) \) 2 3

FP/FL (John Backus)
\([+, *]:<2,3>\)

Faust
2,3 <=: +,*

Figure: block-diagram of 2,3 <=: +,*
Block-Diagram Algebra

Same expression in Lambda-Calculus, FP and Faust

Lambda-Calculus
\( \lambda x. \lambda y. (x+y, x*y) \) 2 3

FP/FL (John Backus)
[+,*]:<2,3>

Faust
2,3 <: +,*

Figure: block-diagram of 2,3 <: +,*
Block-Diagram Algebra

Same expression in Lambda-Calculus, FP and Faust

**Lambda-Calculus**
\[ \lambda x. \lambda y. (x+y, x*y) \]

**FP/FL (John Backus)**
\[ [+,*] : <2,3> \]

**Faust**
\[ 2,3 <: +,* \]

**Figure:** block-diagram of \( 2,3 <: +,* \)
Demo 2: Parallelization
So, What’s new?

- Modular architecture files
- Automatic Documentation
So, What’s new?

- Modular architecture files
- Automatic Documentation
So, What’s new?

- Modular architecture files
- Automatic Documentation
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.
- It is the role of the architecture file to describe how to connect the audio computation to the external world.
- Monolithic architectures, as used so far, work very well but have reached their limits in terms of flexibility and maintainability.
- A new approach was needed: modular architecture files easier to develop, to combine and to maintain.
Faust Modular Architecture Files

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.
- It is the role of the architecture file to describe how to connect the audio computation to the external world.
- Monolithic architectures, as used so far, work very well but have reached their limits in terms of flexibility and maintainability.
- A new approach was needed: modular architecture files easier to develop, to combine and to maintain.
Faust Modular Architecture Files
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.
- It is the role of the architecture file to describe how to connect the audio computation to the external world.
- Monolithic architectures, as used so far, works very well but have reached their limits in terms of flexibility and maintainability
- A new approach was needed: modular architecture files easier to develop, to combine and to maintain
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.

It is the role of the architecture file to describe how to connect the audio computation to the external world.

Monolithic architectures, as used so far, works very well but have reached their limits in terms of flexibility and maintainability.

A new approach was needed: modular architecture files easier to develop, to combine and to maintain.
Faust Modular Architecture Files

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.
- It is the role of the *architecture file* to describe how to connect the audio computation to the external world.
- Monolithic architectures, as used so far, works very well but have reached their limits in terms of flexibility and maintainability.
- A new approach was needed: *modular architecture files* easier to develop, to combine and to maintain.
Faust Modular Architecture Files

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.
- It is the role of the architecture file to describe how to connect the audio computation to the external world.
- Monolithic architectures, as used so far, works very well but have reached their limits in terms of flexibility and maintainability
- A new approach was needed: modular architecture files easier to develop, to combine and to maintain
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.
- It is the role of the architecture file to describe how to connect the audio computation to the external world.
- Monolithic architectures, as used so far, work very well but have reached their limits in terms of flexibility and maintainability.
- A new approach was needed: modular architecture files easier to develop, to combine and to maintain.
Faust Modular Architecture Files

Easy Deployment

Faust can be used to write:

- Audio plugins:
  - LADSPA
  - DSSI
  - Max/MSP
  - VST
  - PD
  - Csound
  - Supercollider
  - Pure
  - Chuck
  - Octave
  - Flash

As well as:

- Standalone audio applications:
  - Jack
  - Alsa
  - CoreAudio
  - iPhone

- Mathematical documentation
- Graphic block-diagram
- Internal tasks DAG
- XML User Interface description

Y. Orlarey, K. Barkati, D. Fober, S. Letz (GRAME Centre National de Création Musicale)

What's new in Faust

May 2011 15 / 42
Faust can be used to write:

- Audio plugins:
  - LADSPA
  - DSSI
  - Max/MSP
  - VST
  - PD
  - SuperCollider
  - Pure
  - ChucK
  - Octave
  - FLTK

As well as:

- Standalone audio applications:
  - Jack
  - Alsa
  - CoreAudio
  - iPhone

- Mathematical documentation
- Graphic block-diagram
- Internal tasks DAG
- XML User Interface description
Faust Modular Architecture Files

Easy Deployment

Faust can be used to write:

- Audio plugins:
  - LADSPA
  - DSSI
  - Max/MSP
  - VST
  - PD
  - Csound
  - Supercollider
  - Pure
  - Chuck
  - Octave
  - Flash

As well as:

- Standalone audio applications:
  - Jack
  - Alsa
  - CoreAudio
  - iPhone

- Mathematical documentation
- Graphic block-diagram
- Internal tasks DAG
- XML User Interface description
Faust can be used to write:

- **Audio plugins:**
  - LADSPA
  - DSSI
  - Max/MSP
  - VST
  - PD
  - CSound
  - Supercollider
  - Pure
  - Chuck
  - Octave
  - Flash

As well as:

- **Standalone audio applications:**
  - Jack
  - Alsa
  - CoreAudio
  - iPhone

- **Mathematical documentation**
- **Graphic block-diagram**
- **Internal tasks DAG**
- **XML User Interface description**
Faust can be used to write:

- **Audio plugins**:
  - LADSPA
  - DSSI
  - Max/MSP
  - VST
  - PD
  - Csound
  - Supercollider
  - Pure
  - Chuck
  - Octave
  - Flash

As well as:

- **Standalone audio applications**:
  - Jack
  - Alsa
  - CoreAudio
  - iPhone

- **Mathematical documentation**
- **Graphic block-diagram**
- **Internal tasks DAG**
- **XML User Interface description**
Faust Modular Architecture Files

Easy Deployment

Faust can be used to write:

- Audio plugins:
  - LADSPA
  - DSSI
  - Max/MSP
  - VST
  - PD
  - Csound
  - Supercollider
  - Pure
  - Chuck
  - Octave
  - Flash

As well as:

- Standalone audio applications:
  - Jack
  - Alsa
  - CoreAudio
  - iPhone

- Mathematical documentation
- Graphic block-diagram
- Internal tasks DAG
- XML User Interface description
Faust can be used to write:

- **Audio plugins**:
  - LADSPA
  - DSSI
  - Max/MSP
  - VST
  - PD
  - Csound
  - Supercollider
  - Pure
  - Chuck
  - Octave
  - Flash

As well as:

- **Standalone audio applications**:
  - Jack
  - Alsa
  - CoreAudio
  - iPhone

- **Mathematical documentation**

- **Graphic block-diagram**

- **Internal tasks DAG**

- **XML User Interface description**
Faust can be used to write:

- **Audio plugins**:
  - LADSPA
  - DSSI
  - Max/MSP
  - VST
  - PD
  - Csound
  - Supercollider
  - Pure
  - Chuck
  - Octave
  - Flash

As well as:

- **Standalone audio applications**:
  - Jack
  - Alsa
  - CoreAudio
  - iPhone

- **Mathematical documentation**
- **Graphic block-diagram**
- **Internal tasks DAG**
- **XML User Interface description**
Faust Modular Architecture Files

Easy Deployement

Faust can be used to write:

- Audio plugins:
  - LADSPA
  - DSSI
  - Max/MSP
  - VST
  - PD
  - Csound
  - Supercollider
  - Pure
  - Chuck
  - Octave
  - Flash

As well as:

- Standalone audio applications:
  - Jack
  - Alsa
  - CoreAudio
  - iPhone

- Mathematical documentation
- Graphic block-diagram
- Internal tasks DAG
- XML User Interface description
Faust can be used to write:

- Audio plugins:
  - LADSPA
  - DSSI
  - Max/MSP
  - VST
  - PD
  - Csound
  - Supercollider
  - Pure
  - Chuck
  - Octave
  - Flash

As well as:

- Standalone audio applications:
  - Jack
  - Alsa
  - CoreAudio
  - iPhone

- Mathematical documentation
- Graphic block-diagram
- Internal tasks DAG
- XML User Interface description
Faust can be used to write:

- Audio plugins:
  - LADSPA
  - DSSI
  - Max/MSP
  - VST
  - PD
  - Csound
  - Supercollider
  - Pure
  - Chuck
  - Octave
  - Flash

As well as:

- Standalone audio applications:
  - Jack
  - Alsa
  - CoreAudio
  - iPhone

- Mathematical documentation
- Graphic block-diagram
- Internal tasks DAG
- XML User Interface description
Faust Modular Architecture Files

Easy Deployment

---

Faust can be used to write:

- **Audio plugins:**
  - LADSPA
  - DSSI
  - Max/MSP
  - VST
  - PD
  - Csound
  - Supercollider
  - Pure
  - Chuck
  - Octave
  - Flash

As well as:

- **Standalone audio applications:**
  - Jack
  - Alsa
  - CoreAudio
  - iPhone

- **Mathematical documentation**
- **Graphic block-diagram**
- **Internal tasks DAG**
- **XML User Interface description**

---

Y. Orlarey, K. Barkati, D. Fober, S. Letz (GRAME Centre National de Création Musicale)

What's new in Faust

May 2011
Faust can be used to write:

- **Audio plugins**: 
  - LADSPA
  - DSSI
  - Max/MSP
  - VST
  - PD
  - Csound
  - Supercollider
  - Pure
  - Chuck
  - Octave
  - Flash

As well as:

- **Standalone audio applications**: 
  - Jack
  - Alsa
  - CoreAudio
  - iPhone

- **Mathematical documentation**
- **Graphic block-diagram**
- **Internal tasks DAG**
- **XML User Interface description**
Faust can be used to write:

- Audio plugins:
  - LADSPA
  - DSSI
  - Max/MSP
  - VST
  - PD
  - Csound
  - Supercollider
  - Pure
  - Chuck
  - Octave
  - Flash

As well as:

- Standalone audio applications:
  - Jack
  - Alsa
  - CoreAudio
  - iPhone

- Mathematical documentation
- Graphic block-diagram
- Internal tasks DAG
- XML User Interface description
Faust Modular Architecture Files

Easy Deployment

Faust can be used to write:

- Audio plugins:
  - LADSPA
  - DSSI
  - Max/MSP
  - VST
  - PD
  - Csound
  - Supercollider
  - Pure
  - Chuck
  - Octave
  - Flash

As well as:

- Standalone audio applications:
  - Jack
  - Alsa
  - CoreAudio
  - iPhone
- Mathematical documentation
- Graphic block-diagram
- Internal tasks DAG
- XML User Interface description
Faust Modular Architecture Files

Easy Deployement

Faust can be used to write:

- Audio plugins:
  - LADSPA
  - DSSI
  - Max/MSP
  - VST
  - PD
  - Csound
  - Supercollider
  - Pure
  - Chuck
  - Octave
  - Flash

As well as:

- Standalone audio applications:
  - Jack
  - Alsa
  - CoreAudio
  - iPhone

- Mathematical documentation
- Graphic block-diagram
- Internal tasks DAG
- XML User Interface description

Y. Orlarey, K. Barkati, D. Fober, S. Letz (GRAME Centre National de Création Musicale)

What's new in Faust

May 2011
Faust Modular Architecture Files

Easy Deployment

Faust can be used to write:

- Audio plugins:
  - LADSPA
  - DSSI
  - Max/MSP
  - VST
  - PD
  - Csound
  - Supercollider
  - Pure
  - Chuck
  - Octave
  - Flash

As well as:

- Standalone audio applications:
  - Jack
  - Alsa
  - CoreAudio
  - iPhone
- Mathematical documentation
- Graphic block-diagram
- Internal tasks DAG
- XML User Interface description
Faust can be used to write:

- Audio plugins:
  - LADSPA
  - DSSI
  - Max/MSP
  - VST
  - PD
  - Csound
  - Supercollider
  - Pure
  - Chuck
  - Octave
  - Flash

As well as:

- Standalone audio applications:
  - Jack
  - Alsa
  - CoreAudio
  - iPhone

- Mathematical documentation
- Graphic block-diagram
- Internal tasks DAG
- XML User Interface description
Faust Modular Architecture Files

Easy Deployment

Faust can be used to write:

- Audio plugins:
  - LADSPA
  - DSSI
  - Max/MSP
  - VST
  - PD
  - Csound
  - Supercollider
  - Pure
  - Chuck
  - Octave
  - Flash

As well as:

- Standalone audio applications:
  - Jack
  - Alsa
  - CoreAudio
  - iPhone

- Mathematical documentation
- Graphic block-diagram
- Internal tasks DAG
- XML User Interface description

Y. Orlarey, K. Barkati, D. Fober, S. Letz (GRAME Centre National de Création Musicale)

What's new in Faust
Faust Modular Architecture Files

Easy Deployment

Faust can be used to write:

- Audio plugins:
  - LADSPA
  - DSSI
  - Max/MSP
  - VST
  - PD
  - Csound
  - Supercollider
  - Pure
  - Chuck
  - Octave
  - Flash

As well as:

- Standalone audio applications:
  - Jack
  - Alsa
  - CoreAudio
  - iPhone

- Mathematical documentation
- Graphic block-diagram
- Internal tasks DAG
- XML User Interface description
Faust Modular Architecture Files

Easy Deployment

Faust can be used to write:

- Audio plugins:
  - LADSPA
  - DSSI
  - Max/MSP
  - VST
  - PD
  - CSound
  - Supercollider
  - Pure
  - Chuck
  - Octave
  - Flash

As well as:

- Standalone audio applications:
  - Jack
  - Alsa
  - CoreAudio
  - iPhone

- Mathematical documentation
- Graphic block-diagram
- Internal tasks DAG
- XML User Interface description
Faust can be used to write:

- Audio plugins:
  - LADSPA
  - DSSI
  - Max/MSP
  - VST
  - PD
  - Csound
  - Supercollider
  - Pure
  - Chuck
  - Octave
  - Flash

As well as:

- Standalone audio applications:
  - Jack
  - Alsa
  - CoreAudio
  - iPhone

- Mathematical documentation
  - Graphic block-diagram
  - Internal tasks DAG
  - XML User Interface description
Faust can be used to write:

- Audio plugins:
  - LADSPA
  - DSSI
  - Max/MSP
  - VST
  - PD
  - Csound
  - Supercollider
  - Pure
  - Chuck
  - Octave
  - Flash

As well as:

- Standalone audio applications:
  - Jack
  - Alsa
  - CoreAudio
  - iPhone

- Mathematical documentation

- Graphic block-diagram

- Internal tasks DAG

- XML User Interface description
Faust can be used to write:

- Audio plugins:
  - LADSPA
  - DSSI
  - Max/MSP
  - VST
  - PD
  - Csound
  - Supercollider
  - Pure
  - Chuck
  - Octave
  - Flash

As well as:

- Standalone audio applications:
  - Jack
  - Alsa
  - CoreAudio
  - iPhone
- Mathematical documentation
- Graphic block-diagram
- Internal tasks DAG
- XML User Interface description
Faust can be used to write:

- Audio plugins:
  - LADSPA
  - DSSI
  - Max/MSP
  - VST
  - PD
  - Csound
  - Supercollider
  - Pure
  - Chuck
  - Octave
  - Flash

As well as:

- Standalone audio applications:
  - Jack
  - Alsa
  - CoreAudio
  - iPhone

- Mathematical documentation
- Graphic block-diagram
- Internal tasks DAG
- XML User Interface description
Demo 3: Easy Deployment
To provide easy deployment, the DSP code generated by compiling a Faust program should be pure audio computation, abstracted from any audio drivers or GUI toolkit.
class dsp {
    protected:
        int fSamplingFreq;
    public:
        dsp() {}  // Constructor
        virtual ~dsp() {}  // Destructor

        virtual int getNumInputs() = 0;
        virtual int getNumOutputs() = 0;
        virtual void buildUserInterface(UI* interface) = 0;
        virtual void init(int samplingRate) = 0;
        virtual void compute(int len, float** inputs, float** outputs) = 0;
};
The role of the architecture file is to provide the missing information: the audio drivers and the user interface. The new modular architecture file combines an Audio driver module and one or more User Interface modules.
The Faust compiler wraps the DSP code into the selected architecture file. For examples, `faust -a jack-gtk.cpp noise.dsp` will wrap the dsp code of a noise generator into the architecture of jack-gtk standalone application.
Faust Modular Architecture Files

The Audio Class

class audio {
  public:
    audio() {}
    virtual ~audio() {}
    virtual bool init(const char*, dsp*) = 0;
    virtual bool start() = 0;
    virtual void stop() = 0;
};
class UI
{
    public:

    UI() { }  
    virtual ~UI() { }

    virtual void openFrameBox(const char* label) = 0;
    virtual void openTabBox(const char* label) = 0;
    virtual void openHorizontalBox(const char* label) = 0;
    virtual void openVerticalBox(const char* label) = 0;
    virtual void closeBox() = 0;

    virtual void addButton(const char* label, float* zone) = 0;
    virtual void addToggleButton(const char* label, float* zone) = 0;
    virtual void addCheckButton(const char* label, float* zone) = 0;
    virtual void addVerticalSlider(const char* label, float* zone, ...) = 0;
    virtual void addHorizontalSlider(const char* label, float* zone, ...) = 0;
    virtual void addNumEntry(const char* label, float* zone, ...) = 0;

    virtual void addNumDisplay(const char* label, float* zone, int precision) = 0;
    virtual void addTextDisplay(const char* label, float* zone, const char* names[], ...) = 0;
    virtual void addHorizontalBargraph(const char* label, float* zone, ...) = 0;
    virtual void addVerticalBargraph(const char* label, float* zone, ...) = 0;

    virtual void declare(float* , const char* , const char* ) {}}

Faust Modular Architecture Files

Audio modules Hierarchy

- alsaaudio
- coreaudio
- netjackaudio
- portaudio

Samples via osc mesages

Access netjack2 as a "master" or a "slave"
Faust Modular Architecture Files

The jack-gtk architecture

```c
int main(int argc, char *argv[]) {
    char appname[256];
    char rcfilename[256];
    char* home = getenv("HOME");

    snprintf(appname, 255, "%s", basename(argv[0]));
    snprintf(rcfilename, 255, "%s/%src", home, appname);

    DSP = new mydsp();
    if (DSP==0) {
        cerr << "Unable to allocate Faust DSP object" << endl; exit(1);
    }
    GUI* interface = new GTKUI (appname, &argc, &argv);
    FUI* finterface = new FUI();
    GUI* oscinterface = new OSCUI(appname, argc, argv);

    DSP->buildUserInterface(interface);
    DSP->buildUserInterface(finterface);
    DSP->buildUserInterface(oscinterface);

    jackaudio audio;
    audio.init(appname, DSP);
    finterface->recallState(rcfilename);
    audio.start();
    oscinterface->run();
    interface->run();
    audio.stop();
    finterface->saveState(rcfilename);
    delete DSP;
    return 0;
}
```
Demo 4: OSC support
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.
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.
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`.

```bash
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 \texttt{faust2mathdoc} script on a Faust file.

This script relies on a new option of the Faust compile: \texttt{-mdoc}

\texttt{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
Automatic Mathematical Documentation

Tags

- The documentation can be generated fully automatically from a regular Faust program.
- It can also be controlled by embedding documentation statements in Faust programs.
- A documentation statements starts with an opening <mdoc> tag and ends with a closing </mdoc> tag.
- It contains arbitrary text, typically in \LaTeX\ format, as well as five sub-tags:
  - <diagram>
  - <equation>
  - <metadata>
  - <notice/>
  - <listing/>
Automatic Mathematical Documentation

Tags

- The documentation can be generated fully automatically from a regular Faust program.
- It can also be controlled by embedding documentation statements in Faust programs.
- A documentation statements starts with an opening <mdoc> tag and ends with a closing </mdoc> tag.
- It contains arbitrary text, typically in \LaTeX format, as well as five sub-tags:
  - <diagram>
  - <equation>
  - <metadata>
  - <notice/>
  - <listing/>
The documentation can be generated fully automatically from a regular Faust program.

It can also be controlled by embedding *documentation statements* in Faust programs.

A *documentation statements* starts with an opening `<mdoc>` tag and ends with a closing `</mdoc>` tag.

It contains arbitrary text, typically in LaTeX format, as well as five sub-tags:

- `<diagram>`
- `<equation>`
- `<metadata>`
- `<notice/>`
- `<listing/>`
The documentation can be generated fully automatically from a regular Faust program.

It can also be controlled by embedding documentation statements in Faust programs.

A documentation statement starts with an opening <mdoc> tag and ends with a closing </mdoc> tag.

It contains arbitrary text, typically in \LaTeX format, as well as five sub-tags:

- <diagram>
- <equation>
- <metadata>
- <notice/>
- <listing/>
The documentation can be generated fully automatically from a regular Faust program.

It can also be controlled by embedding documentation statements in Faust programs.

A documentation statement starts with an opening `<mdoc>` tag and ends with a closing `</mdoc>` tag.

It contains arbitrary text, typically in LaTeX format, as well as five sub-tags:

- `<diagram>`
- `<equation>`
- `<metadata>`
- `<notice/>`
- `<listing/>`
Automatic Mathematical Documentation

Tags

- The documentation can be generated fully automatically from a regular Faust program.
- It can also be controlled by embedding documentation statements in Faust programs.
- A documentation statements starts with an opening `<mdoc>` tag and ends with a closing `</mdoc>` tag.
- It contains arbitrary text, typically in \LaTeX\ format, as well as five sub-tags:
  - `<diagram>`
  - `<equation>`
  - `<metadata>`
  - `<notice/>`
  - `<listing/>`
The documentation can be generated fully automatically from a regular Faust program.

It can also be controlled by embedding documentation statements in Faust programs.

A documentation statements starts with an opening `<mdoc>` tag and ends with a closing `</mdoc>` tag.

It contains arbitrary text, typically in LaTeX format, as well as five sub-tags:

- `<diagram>`
- `<equation>`
- `<metadata>`
- `<notice/>`
- `<listing/>`
Automatic Mathematical Documentation

Tags

- The documentation can be generated fully automatically from a regular Faust program.
- It can also be controlled by embedding documentation statements in Faust programs.
- A documentation statements starts with an opening <mdoc> tag and ends with a closing </mdoc> tag.
- It contains arbitrary text, typically in \LaTeX\ format, as well as five sub-tags:
  - <diagram>
  - <equation>
  - <metadata>
  - <notice/>
  - <listing/>
Automatic Mathematical Documentation

Tags

- The documentation can be generated fully automatically from a regular Faust program.
- It can also be controlled by embedding *documentation statements* in Faust programs.
- A *documentation statements* starts with an opening `<mdoc>` tag and ends with a closing `</mdoc>` tag.
- It contains arbitrary text, typically in LaTeX format, as well as five sub-tags:
  - `<diagram>`
  - `<equation>`
  - `<metadata>`
  - `<notice/>`
  - `<listing/>`
Automatic Mathematical Documentation

Tags

- The documentation can be generated fully automatically from a regular Faust program.
- It can also be controlled by embedding documentation statements in Faust programs.
- A documentation statements starts with an opening `<mdoc>` tag and ends with a closing `</mdoc>` tag.
- It contains arbitrary text, typically in \LaTeX\ format, as well as five sub-tags:
  - `<diagram>`
  - `<equation>`
  - `<metadata>`
  - `<notice/>`
  - `<listing/>`
Automatic Mathematical Documentation

Diagram tags

- The generation of the graphical block-diagram of a Faust expression can be requested using `<diagram>...</diagram>` tags.
- For example: `<diagram> +~_ </diagram>` will produce the embedded diagram:
Diagram tags

- The generation of the graphical block-diagram of a Faust expression can be requested using `<diagram>...</diagram>` tags.
- For example: `<diagram> +~_ </diagram>` will produce the embedded diagram:
The generation of the graphical block-diagram of a Faust expression can be requested using `<diagram>...</diagram>` tags.

For example: `<diagram> +~_ </diagram>` will produce the embedded diagram:
The generation of the graphical block-diagram of a Faust expression can be requested using `<diagram>...</diagram>` tags.

For example: `<diagram> +~_ </diagram>` will produce the embedded diagram:
Automatic Mathematical Documentation

Equation tag

- The generation of the mathematical equations of a Faust expression can be requested using `<equation>...</equation>` tags.
- For example: `<equation> +~_ </equation>` will produce the following description:

**Expression equations**

- Output signal $y$ such that

\[ y(t) = r_1(t) \]

- Input signal $x$

- Intermediate signal $r_1$ such that

\[ r_1(t) = x(t) + r_1(t-1) \]
The generation of the mathematical equations of a Faust expression can be requested using `<equation>...</equation>` tags.

For example: `<equation> +~< /equation>` will produce the following description:

**Expression equations**

- Output signal $y$ such that
  
  $$y(t) = r_1(t)$$

- Input signal $x$

- Intermediate signal $r_1$ such that
  
  $$r_1(t) = x(t) + r_1(t-1)$$
Automatic Mathematical Documentation

Equation tag

- The generation of the mathematical equations of a Faust expression can be requested using `<equation>...</equation>` tags.
- For example: `<equation> _ + ~ </equation>` will produce the following description:

Expression equations

- Output signal $y$ such that

  $$y(t) = r_1(t)$$

- Input signal $x$

- Intermediate signal $r_1$ such that

  $$r_1(t) = x(t) + r_1(t-1)$$
The generation of the mathematical equations of a Faust expression can be requested using `<equation>...<equation>` tags.

For example: `<equation> +~_ </equation>` will produce the following description:

Expression equations

- Output signal $y$ such that
  
  $$y(t) = r_1(t)$$

- Input signal $x$

- Intermediate signal $r_1$ such that
  
  $$r_1(t) = x(t) + r_1(t-1)$$
Automatic Mathematical Documentation

Metadata tags

- The `<metadata>...</metadata>` tags allow to refer Faust metadatas declarations.
- For example if the Faust code declares:
  
  ```
  declare author "Albert";
  ```

- then occurrences in the documentation of:
  
  ```
  <metadata>author</metadata>
  ```

- will be replaced by: "Albert".
The `<metadata>...</metadata>` tags allow to refer Faust metadata declarations.

For example if the Faust code declares:

```
declare author "Albert";
```

then occurrences in the documentation of:

```
<metadata>author</metadata>
```

will be replaced by: "Albert".
The `<metadata>...</metadata>` tags allow to refer Faust metadata declarations.

For example if the Faust code declares:

```plaintext
declare author "Albert";
```

then occurrences in the documentation of:

```plaintext
<metadata>author</metadata>
```

will be replaced by: "Albert".
The `<metadata>...</metadata>` tags allow to refer Faust metadata declarations.

For example if the Faust code declares:

```plaintext
declare author "Albert";
```

then occurrences in the documentation of:

```plaintext
<metadata>author</metadata>
```

will be replaced by: "Albert".
Automatic Mathematical Documentation

Metadata tags

- The `<metadata>`...`</metadata>` tags allow to refer Faust metadata declarations.
- For example if the Faust code declares:

```plaintext
declare author "Albert";
```

- then occurrences in the documentation of:

```plaintext
<metadata>author</metadata>
```

- will be replaced by: "Albert".
The `<notice/>` empty-element tag is used to generate the conventions used in the mathematical equations.
The `<notice/>` empty-element tag is used to generate the conventions used in the mathematical equations.
Listing tag

- The `<listing/>` tag is used to generate the listing of the Faust program.
- It can be customized with three boolean attributes that can take the value `true` or `false`:
  - `mdoctags` when true (the default) indicates to include the mdoc comments into the listing.
  - `dependencies` when true (the default) indicates to include the code of the imported libraries into the listing.
  - `distributed` when true (the default) indicates to respect the distribution of faust code and mdoc tags.

Example

```xml
<listing mdoctags="true"
   dependencies="false"
   distributed="false" />
```
Listing tag

- The `<listing/>` tag is used to generate the listing of the Faust program.
- It can be customized with three boolean attributes that can take the value `true` or `false`:
  - `mdoctags` when `true` (the default) indicates to include the mdoc comments into the listing.
  - `dependencies` when `true` (the default) indicates to include the code of the imported libraries into the listing.
  - `distributed` when `true` (the default) indicates to respect the distribution of faust code and mdoc tags.

Example

```xml
<listing mdoctags="true"
  dependencies="false"
  distributed="false"/>
```
Automatic Mathematical Documentation

Listing tag

- The `<listing/>` tag is used to generate the listing of the Faust program.
- It can be customized with three boolean attributes that can take the value `true` or `false`:
  - `mdoctags` when true (the default) indicates to include the mdoc comments into the listing.
  - `dependencies` when true (the default) indicates to include the code of the imported libraries into the listing.
  - `distributed` when true (the default) indicates to respect the distribution of faust code and mdoc tags.

Example

```xml
<listing mdoctags="true"
    dependencies="false"
    distributed="false" />
```
Automatic Mathematical Documentation

Listing tag

- The `<listing/>` tag is used to generate the listing of the Faust program.
- It can be customized with three boolean attributes that can take the value `true` or `false`:
  - `mdoctags` when true (the default) indicates to include the mdoc comments into the listing.
  - `dependencies` when true (the default) indicates to include the code of the imported libraries into the listing.
  - `distributed` when true (the default) indicates to respect the distribution of faust code and mdoc tags.

Example

```xml
<listing  mdoctags="true"
  dependencies="false"
  distributed="false" />
```
Automatic Mathematical Documentation

Listing tag

- The `<listing/>` tag is used to generate the listing of the Faust program.
- It can be customized with three boolean attributes that can take the value `true` or `false`:
  - `mdoctags` when true (the default) indicates to include the mdoc comments into the listing.
  - `dependencies` when true (the default) indicates to include the code of the imported libraries into the listing.
  - `distributed` when true (the default) indicates to respect the distribution of faust code and mdoc tags.

Example

```xml
<listing mdoctags="true"
  dependencies="false"
  distributed="false" />
```
Automatic Mathematical Documentation

Listing tag

- The `<listing/>` tag is used to generate the listing of the Faust program.
- It can be customized with three boolean attributes that can take the value `true` or `false`:
  - `mdoctags` when true (the default) indicates to include the mdoc comments into the listing.
  - `dependencies` when true (the default) indicates to include the code of the imported libraries into the listing.
  - `distributed` when true (the default) indicates to respect the distribution of faust code and mdoc tags.

Example

```xml
<listing mdoctags="true"
    dependencies="false"
    distributed="false" />
```
Automatic Mathematical Documentation

Listing tag

- The `<listing/>` tag is used to generate the listing of the Faust program.
- It can be customized with three boolean attributes that can take the value `true` or `false`:
  - `mdoctags` when true (the default) indicates to include the mdoc comments into the listing.
  - `dependencies` when true (the default) indicates to include the code of the imported libraries into the listing.
  - `distributed` when true (the default) indicates to respect the distribution of faust code and mdoc tags.

Example

```xml
<listing mdoctags="true"
       dependencies="false"
       distributed="false" />
```
Demo 5: Documentation
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

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 Distribution on Source Forge

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

What's new in Faust

May 2011
Resources

Using FAUST Distribution on Source Forge

- git clone
git://faudiostream.git.sourceforge.net/gitroot/faudiostream/faudiostream/faust

- cd faust; make; sudo make install
Resources

Using FAUST Distribution on Source Forge

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

Using FaustWorks IDE

- `git clone git://faudiostream.git.sourceforge.net/gitroot/faudiostream/FaustWorks`
- `cd FaustWorks; qmake; make`
Resources

Using FaustWorks IDE

- git clone
  git://faudiostream.git.sourceforge.net/gitroot/faudiostream/FaustWorks

- cd FaustWorks; qmake; make
Resources

Using FaustWorks IDE

- `git clone`
  `git://faudiostream.git.sourceforge.net/gitroot/faudiostream/FaustWorks`

- `cd FaustWorks; qmake; make`
Resources

Books with a FAUST related content

Figure: «*Introduction To Digital Filters with Audio Applications*», prof. Julius O. Smith III, W3K publishing
Audio Signal Processing in FAUST
Julius O. Smith III
Center for Computer Research in Music and Acoustics (CCRMA)
Department of Music, Stanford University, Stanford, California 94305 USA

Abstract

FAUST is a high-level programming language for digital signal processing, with special support for real-time audio applications and plugins on various software platforms including Linux, Mac OS X, iOS, Windows, and embedded computing environments. Audio plugin formats supported include VST, LADSPA, Pd, Max/MSP, SuperCollider, and more. This tutorial provides an introduction focusing on a simple example of white noise filtered by a variable resonator.

Contents

1 Introduction 3
1.1 Installing FAUST 3
1.1.1 A Note About Using FAUST on CCRMA Machines 4
1.2 Installing FaustWorks 4
1.3 FAUST Examples 4
2 Primer on the FAUST Language 5
2.1 Basic Signal Processing Blocks 6
2.2 Block Diagram Operators 6
2.3 Examples 7
2.4 Infix Notation Rewriting 8
2.5 Statements 8
2.6 Function Definition 9
2.7 Term Rewriting 9
2.8 Fixing the Number of Input and Output Signals 9
2.9 Naming Input Signals 9
2.10 Naming Output Signals 9

*This document is an extended subset of http://ccrma.stanford.edu/realsimple/faust/.
Resources

FAUST Quick Reference

Figure: «Faust Quick Reference», Grame