

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*

jos at ccrma.stanford.edu



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, Android, Windows, and embedded computing environments. Audio plugin formats supported include VST, lv2, AU, 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	4
1.2	FAUST Examples	5
2	Primer on the Faust Language	5
2.1	Basic Signal Processing Blocks (Elementary Operators on Signals)	7
2.2	Block Diagram Operators	8
2.3	Examples	8
2.4	Infix Notation Rewriting	8
2.5	Encoding Block Diagrams in the FAUST Language	9
2.6	Statements	9
2.7	Function Definition	10
2.8	Partial Function Application	11
2.9	Functional Notation for Operators	11
2.10	Examples	11

2.11	Summary of FAUST Notation Styles	12
2.12	Unary Minus	12
2.13	Fixing the Number of Input and Output Signals	12
2.14	Naming Input Signals	12
2.15	Naming Output Signals	13
2.16	Signal Types	13
2.17	Signal Comparison Operators	13
2.18	Bitwise Operations for Integer Signals	14
2.19	Foreign Constants and Variables	14
2.20	Foreign Functions	14
2.20.1	Example 1	15
2.20.2	Example 2	15
2.20.3	Functions from <code>math.h</code>	16
2.21	Parallel and Sequence Macros	16
2.22	Sum and Product Macros	16
2.23	Pattern Matching in FAUST	16
2.23.1	Formal Parameter Exception	17
2.23.2	Recursive Block Diagram Specification	18
2.23.3	Understanding <code>count</code> and <code>take</code> from <code>basics.lib</code>	18
2.23.4	Pattern Matching Implementation	19
2.23.5	Using Pattern Matching in Rewriting Rules	19
2.23.6	Using Lisp Syntax to Express Trees	20
2.23.7	Pattern-Matching Example	20
2.23.8	Pattern-Matching Algorithm Description	20
2.23.9	Miscellaneous Pattern-Matching Examples	21
2.24	Scope Rules	21
2.25	White Noise Generator	22
2.26	Further Readings on the FAUST Language	22
2.27	Acknowledgment	22
3	A Simple Example Faust Program	22
4	Verifying and Testing Faust Programs	23
4.1	Generating FAUST Block Diagrams	23
4.2	A Look at the Generated C++ code	26
4.3	Printing/Plotting the Output Signal(s)	26
4.4	Inspecting the Output Signal(s) in Matlab or Octave	29
4.5	Summary of FAUST Program Testing Strategies	29
5	Adding a GUI	30
6	Generating Stand-Alone Qt or GTK Applications	31
7	Generating Other Applications and Plugins	31
8	Generating a LADSPA Plugin via Faust	32

9	Feeding Soundfiles to Faust Standalone Apps	34
9.1	Offline Processing of Soundfiles in FAUST	34
9.2	Soundfile Input for Standalone FAUST Applications	37
9.3	Soundfile Input for FAUST Plugins	37
10	Generating a MIDI Synthesizer for PD	37
11	MIDI Synthesizer Test Patch	39
12	Using Faust with SuperCollider	40
12.1	Getting Started with SuperCollider	41
12.2	Linux and Faust-Generated SuperCollider Plugins	41
12.3	Mac OS X and Faust-Generated SuperCollider Plugins	42
13	Using Faust with Open Sound Control (OSC)	43
14	Conclusions	47
A	Appendix A: State-Space Models to Faust	48
A.1	State-Space BiQuad in FAUST	50
B	Appendix B: Inspecting Assembly to Fine-Tune Performance	52

1 Introduction

The FAUST *programming language*¹ by Yann Orlarey et al. at Grame [3, 7, 8, 4] generates C++ for real-time signal-processing applications and plugins from a high-level specification. In addition to generating efficient inner loops in C++, FAUST supports Graphical User Interface (GUI) specification in the source code. Moreover, FAUST can generate easy-to-read block diagrams directly from the source, illustrating signal flow and processing graphically.

This tutorial provides some basic getting-started info for FAUST, a brief overview of main aspects of the language, and example applications and plugins generated from a simple FAUST program specifying a resonator driven by white noise. It was written before most of the FAUST online documentation,² so there is naturally overlap in content, but the styles are quite different. In part because FAUST is always evolving forward, this tutorial is incomplete, but it is updated when anything addressed needs revision (please report suggestions and errata to the author). The most up-to-date specification of the language has been the FAUST Quick Reference³ manual.

The FAUST compiler translates the FAUST language into C++ files (and other back-end languages that will not be considered here). These C++ files must then be compiled and installed somewhere for use. The reader is therefore assumed to have elementary proficiency in UNIX operating environments such as all Linux distributions and Mac OS.⁴ Such proficiency includes familiarity

¹The FAUST home page is <https://faust.grame.fr/>. The examples in this tutorial were developed using FAUST version 0.9.9.2a2 and updated as needed over time, with the latest version considered being 2.32.1. The name “FAUST” comes from “Functional Audio Stream.”

²<https://faust.grame.fr/>

³<https://github.com/grame-cncm/faust/blob/master-dev/documentation/faust-quick-reference.pdf>

⁴For Windows, options include Cygwin, Windows System for Linux (WSL), and Linux virtual machines.

with a Command-Line Interface (CLI) such as a UNIX *shell* (principally `tcsh` or `bash` in this tutorial), source-code version control using `git`, and elementary compiling, linking, and installation using `make`, some C++ compiler, the standard linker `ld`, and so on. For text editing, Emacs is most highly recommended, but any will do.

1.1 Installing Faust

Packaged FAUST distributions for Linux, Mac, and Windows, etc., tend to be significantly behind the latest version under development. While you can probably get by fine with them, there may be different locations for various pieces of the installation, and some items might not be installed at all. Below we will assume you have installed the latest FAUST from GitHub.

To download the latest version of the FAUST distribution from GitHub (anonymously, read-only), say

```
> git clone https://github.com/grame-cncm/faust
> cd faust
> git checkout master-dev
```

where ‘>’ denotes your shell prompt, such as in `Terminal.app` on a Mac. Next, follow `README.md` to compile and install FAUST on your system. In particular, you will need CMake installed (`port install cmake` on a Mac using MacPorts, etc. Normally `port` can be replaced by `brew` for `brew` users). I always use the latest FAUST `master-dev` version, and find it to be quite stable. From time to time, update to the latest and reinstall as follows (in the `faust` directory):

```
> git pull --recurse-submodules
> make
> sudo make install
```

Recurring submodules takes care of updating the `faustlibraries` submodule as well as FAUST.

On the Mac, the FAUST distribution depends on the `pkgconfig` package, and also on `qt5` for making standalone apps using the `faust2caqt` script that installs with FAUST. A general trick for easily installing all necessary dependencies is to *first install* FAUST *using your local package manager*, and then uninstall FAUST, leaving its dependencies installed. For example, using MacPorts on a Mac:

```
> sudo port install faust
... (dependencies installed, then faust) ...
> sudo port uninstall faust
> cd ~
> git clone https://github.com/grame-cncm/faust.git faust
> cd faust
> make
> sudo make install
> rehash
```

To include OSC and HTTP support (which require `liblo` and `libmicrohttp` respectively), as well as all Faust language “back ends” (target languages beyond C++) say “`make world`” in place of “`make`” above.

On a Mac, you need the *Command Line Tools for Xcode* to be able to compile C++ code in the standard ways for the Mac. These tools are downloadable via <https://developer.apple.com/downloads/index.action>. See also the Xcode menu item Xcode / Open Developer Tool / More Developer Tools

1.2 Faust Examples

The FAUST distribution contains a set of programming examples in the `examples` subdirectory. For example, to see the graphical equalizer demo on a Mac, assuming your Terminal/shell working directory is where you typed `make` above, say

```
> cd examples/filtering/  
> faust2caqt graphicEqLab.dsp  
> open graphicEqLab.app
```

and experiment with the example real-time filterbank driven by a sawtooth oscillator or white/pink noise. The use of `faust2caqt` assumes you are on a Mac (`'ca'` stands for “Core Audio”) and have Qt installed (e.g., MacPorts’ `qt5-mac` or brew’s `qt5` on the Mac). If not, there is also `faust2jack` which uses the GNOME ToolKit (GTK), etc. (`sudo dnf install gtk2-devel` on Fedora Linux as of May 2020).

The source code for the functions used in `graphicEqLab.dsp` (prefix `'dm.'`) may be found in `libraries/demos.lib`. They, in turn, call functions in the FAUST libraries such as `libraries/filters.lib`.

Note that the FAUST libraries are developed in a separate git project⁵ from FAUST, but they are copied in the `faustlibraries` submodule of the FAUST project.

More FAUST demos/examples can be found online.⁶

2 Primer on the Faust Language

FAUST is a *functional programming language* [6] in which the “main” function, called `process` in FAUST, specifies a signal-processing *block diagram*.⁷ A very simple example is

```
process(x) = x;
```

which defines a simple “wire” block diagram that connects its single input to its single output. We can also define a pair of wires for stereo operation:

```
process(x,y) = x,y;
```

Here we have introduced the comma block-diagram operator (`,`) on the righthand side which combines block diagrams in *parallel*. (On the lefthand side, a comma separates function arguments, which is a different usage of comma.)

Common block-diagram functions are *predefined*, such as `“+”`:

⁵<https://github.com/grame-cncm/faustlibraries.git>

⁶<https://faust.grame.fr/>

⁷On July 26, 2021, Yann Orlarey wrote the following to Faust users <faudiostream-users@lists.sourceforge.net>: “Faust is inspired by Moses Schönfinkel’s combinatory logic (1924) and John Backus’ FP (1977). The idea of Schönfinkel was to eliminate the need for variables in mathematical logic. In functional programming, this style is known as ‘point-free’ or ‘tacit’ programming.”

```
process(x,y) = x,y:+;
```

Now we have introduced the colon (:) operator for combining block diagrams in *series*. The block diagram consisting of two wires `x` and `y` in parallel is connected to the two inputs of the `+` block diagram.

Since the `+` operator already has two inputs, we do not need to refer to them explicitly. Thus,

```
process = +;
```

is equivalent to “`process(x,y) = x,y:+`”, and it therefore also specifies a block diagram consisting of two input signals, one output signal, and a summer, as shown in Fig. 1.

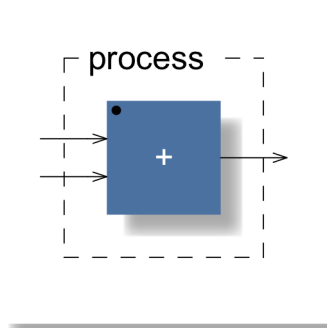


Figure 1: Main *process* block diagram for a two-input adder: `process = +;`

The underbar symbol `_`, also called a “wire”, denotes the trivial block diagram that feeds its single input to its single output. Thus, using the predefined block diagram “`_`” we can define the “mono wire” function “`process(x)=x;`” as

```
process = _;
```

and the “stereo bus” as

```
process = _,_;
```

which is equivalent to “`process(x,y)=x,y;`”.

The block diagram generated by `faust -svg wire.dsp` (as used by the script `faust2firefox`), where `wire.dsp` contains `process=_;`, is shown in Fig. 2.

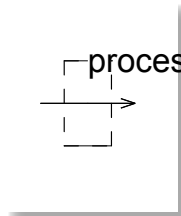


Figure 2: `process = _;`

Arguably even simpler is

```
process = 0;
```

where 0 can be thought of as a predefined block diagram having no input signals, and one output signal that is a stream of zeros. The block diagram for this is shown in Fig. 3.

It is convenient to refer to a block diagram as a *signal* when it has no input signals and one output signal.

Similarly,

```
process = 1;
```

specifies a block diagram having no input signals and an output signal that is a stream of ones *starting at time zero*. Block-diagrams such as ‘0’ and ‘1’ may be thought of as *constant signals*. However, since a constant signal, as defined, is semantically a block diagram, all operations valid for block diagrams (listed below) can be applied. For example, using the one-sample-delay postfix operator ‘*’*’, we can specify the *unit impulse* signal by

```
impulse = 1 - 1’;
```

because 1 is the unit-step signal (a unit constant turning on at time 0) and 1’ is the unit-step delayed by one sample.

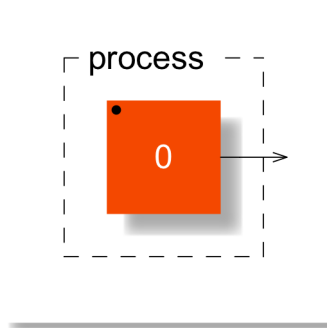


Figure 3: `process = 0;`

2.1 Basic Signal Processing Blocks (Elementary Operators on Signals)

In addition to numbers which specify constant signals, primitive signal-processing blocks include the following:

-	output signal = first input signal minus the second input signal (referring to Fig. 1, the “first” signal is the one nearest the black reference dot)
*	output signal = pointwise product of the two input signals
/	output signal = first input divided by second input (pointwise)
^	output signal = first input raised to the power of the second input (pointwise)
!	(“cut”) input signal is terminated (no output signal)
%	(“modulo”) output = remainder after dividing first input by second input
mem	output signal = input signal delayed by one sample (same as @(1))
@	output = first input delayed by (integer) value of second input

2.2 Block Diagram Operators

There are several fundamental block-diagram operators:

:	combine block diagrams in <i>series</i>
,	combine block diagrams in <i>parallel</i>
<:	<i>split</i> : signal fan-out
:>	<i>merge</i> : signal fan-in (with summation)
~	<i>recursive</i> : specify <i>feedback</i>

These will be illustrated in the examples below.

2.3 Examples

If any of the following examples are not obvious, paste them into the Faust Editor⁸, or a file named `test.dsp` followed by “`faust2firefox test.dsp`” in a shell:

```

process = _ : _;          // series combination (1 in, 1 out)
process = _ , _;          // parallel combination (2 ins, 2 outs)
process = +;              // summer (2 ins, 1 out)
process = _,_ : +;        // same summer
process = _,_ : + : _;    // same summer
process = -;              // signal subtractor
process = *;              // pointwise signal multiplier (nonlinear)
process = /;              // pointwise signal divider (nonlinear)
process = mem;            // unit-sample delay
process = _, 1 : @;       // unit-sample delay
process = _,10 : @;       // ten-sample delay
process = a ~ b;          // feedback thru b around a
process = _ ~ _;          // feedback thru _ (generates 0)
process = mem ~ _;        // two-sample closed loop (generates 0)
process = + ~ _;          // digital integrator
process = _ <: _ , _;     // mono to stereo
process = _ <: _ , _ , _; // mono to quad
process = _ , _ <: _ , _ , _; // stereo to quad (see diagram)
process = _ , _ :> _;      // stereo to mono [equiv to +]
process = _ , _ , _ , _ :> _; // quad to mono [equiv to +,+,+]

```

2.4 Infix Notation Rewriting

For readability and convenience, infix notation such as `x * y` is recognized and translated to *Block-Diagram Normal Form* (BDNF) as `x,y:*`, *i.e.*, the parallel signals `x` and `y` are fed to the two-input multiplier `*`. Similarly, the notation `f(x)` is rewritten to `x:f` if `f` is a primitive function. (More generally, `f(x)` is first rewritten as possible using *pattern-matching* (§2.23) and *definition-expansion*

⁸<https://faust.grame.fr/editor/>

(§2.7)). Postfix operators such as `'` are handled similarly. More formally, we can write

$x * y$	\rightarrow	$x, y : *$
x / y	\rightarrow	$x, y : /$
$x \wedge y$	\rightarrow	$x, y : \wedge$
$\text{pow}(x, y)$	\rightarrow	$x, y : \wedge$
$x @ y$	\rightarrow	$x, y : @$
x'	\rightarrow	$x : \text{mem}$
$f(x)$	\rightarrow	$x : f$
$f(x, y)$	\rightarrow	$x, y : f$
$f(x, y, z)$	\rightarrow	$x, y, z : f$

and so on.

2.5 Encoding Block Diagrams in the Faust Language

While many block diagrams are simple to write down in FAUST, such as elementary series and/or parallel combinations of basic primitives, there are others which are harder to immediately see, particularly when there are multiple overlapping feedback loops. A *slew limiter* is a simple example containing dual overlapping feedback.⁹

A general procedure for encoding block diagrams in FAUST, informally called “brute force Faustification,” is obtained by following the originally published theorem on the generality of FAUST as a language for encoding block diagrams [7]. An alternative approach, based on methods of *automatic control*, is to first write down a *state-space model* of the system, from which a FAUST description readily follows (see Appendix A). An early implementation of this method was the Synth-A-Modeler Compiler [1]. A disadvantage of this approach is that the resulting FAUST can be less readable. Instead of a natural left-to-right processing specification, it produces a general state-space model which takes the form of a vector first-order finite-difference recursion.

In this tutorial, apart from Appendix A, simple and immediately obvious translations of block diagrams into FAUST will be used, as these suffice most of the time and provide the most readable code. There is also usually a small performance advantage of the more intuitive encodings over the more general state-space formulation. The performance difference is small because the FAUST compiler does a good job of optimizing the computations implied by the sparse matrices of the state-space description.

2.6 Statements

As described in the FAUST¹⁰ Manual and FAUST Quick Reference,¹¹ there are four types of *statements* in FAUST:

- *definition* — define a function in the FAUST language
- *fileimport* — incorporate other files (like `#include` in C)
- *declaration* — declare “meta data” such as author, copyright, etc.

⁹Writing a Slew Limiter in the Faust Language (8m): <https://youtu.be/3WY0ikTFAe4>

¹⁰<https://faust.grame.fr>

¹¹<https://github.com/grame-cncm/faust/blob/master-dev/documentation/faust-quick-reference.pdf>

- *documentation* — provide XML-style “tags” for in-source documentation

The only required type of statement in a FAUST program is the *definition* statement, and the only required definition-statement is the one defining **process** (analogous to `main()` in C):

```
process = faust_expression;
```

In this tutorial, we will be concerned almost exclusively with definition statements (with some occasional file-imports).

2.7 Function Definition

FAUST is a functional programming language [7]. From this point of view, every block diagram may be seen as a function mapping its input signals to its output signals.

A fully general function definition in FAUST is of the form

```
f(a) = b;
```

where **f** is a name, and **a** and **b** are *block diagram specifications*. In principle, the compiler must recognize the block diagram **a** flowing into the symbol **f** and replace all that by **b**, with any occurrences of **a** within **b** appropriately wired up to the original input **a**. Function arguments are normally a simple parallel bank of named signals, such as `f(x,y,z) = b`, where **b** is a block-diagram expression that may contain symbols **x**, **y**, and **z** which will be bound to the input signals as expected. Naming input signals in this way is often the easiest and most readable way to *copy* input signals within a block-diagram expression:

```
sum_and_diff_unnamed = _,_<:_,_,_,_:+,-;
```

```
sum_and_diff_named(x,y) = x+y, x-y;
```

More general cases will be discussed in §2.23 below (“pattern matching”).

Function definitions can appear in any order; thus, FAUST statements can appear in any order; *however*, there is one exception: When defining a function differently for different *argument patterns* (§2.23), statement order matters because pattern-matches are tried in the order given.

Unused definitions (unused by **process**, either directly or indirectly) are discarded by the FAUST compiler and have no effect on the generated code:

```
x=1;
y=2; // no effect
process = x;
```

Sometimes we have to force inclusion of inaudible processing using the **attach** primitive. For example, the following cases appear equivalent, even in the generated block diagram, but only **levelmeter2** makes it into the generated C++ code (**levelmeter1** gets optimized away by the compiler):

```
smoother(s) = *(1.0 - s) : + ~ *(s); // unity-dc-gain lowpass filter
levelmeter1 = abs : smoother : vbargraph("Level 1 [unit:dB",-70,10);
levelmeter2 = abs : smoother : vbargraph("Level 2 [unit:dB",-70,10);
sol1 = _ <: _,levelmeter1 : _,!;
sol2 = _ <: attach(levelmeter2);
process = sol1,sol2;
```

In summary, every definition-statement defines a function mapping its name (together with any function-arguments) to a block diagram. Only functions encountered via the `process` function are used, unless retained in the compilation using `attach`.

2.8 Partial Function Application

FAUST supports *partial application* of functions. For example, if $f(x,y)$ specifies some stereo process, then $f(x)$ specifies the same stereo process, but the left channel has a formal-parameter name x and the right channel remains an unnamed input wire:

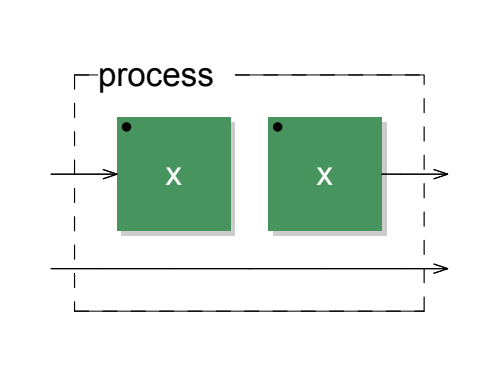


Figure 4: `process(x) = x, _;`

A common use of partial function application is to define named special cases:

```
general_case(case, arg1, arg2, ...) = ...;
special_case_1 = general_case(1);
special_case_2 = general_case(2);
...
```

2.9 Functional Notation for Operators

All of the basic signal processing blocks in §2.1 can be written also in functional notation:

```
+(x)  →  _, x :  +
/(x)  →  _, x :  /
-(x)  →  _, x :  -
^(x)  →  _, x :  pow
```

and so on.

2.10 Examples

```
process = @(1);      // _, 1 : @      [unit-sample delay]
process = @(10);     // _, 10 : @     [10-sample delay]
process = *(2);      // _, 2 : *      [scale by 2]
```

2.11 Summary of Faust Notation Styles

In summary, FAUST supports the following notational variations for the expression $f(x,y) = x+2*y$:

- Core: $x,y:f$
- Functional (“applicative”): $f(x,y)$
- Partial application: $y:f(x)$
- Infix: $x+2*y$

In a post to the FAUST mailing list, Yann Orlarey writes: “How do you choose between core, infix and partial application notations? It is largely a matter of taste. Personally, I like to combine core notation (for the overall structure) with partial application notation (for the slowly varying “parameters”) ... and infix notation for mathematical expressions.”

2.12 Unary Minus

There is a special *unary minus* in FAUST:

$$-x \rightarrow 0, x : -$$

Thus, assuming x denotes a signal, $-x$ is the same as $0-x$ (the negated signal) while $-(x)$ is a block diagram having one input from which x is subtracted (not a signal).

As another example, $-(1+\cos(w))$ is a single-input, single-output block diagram in which the output of $(1+\cos(w))$ (a signal) is subtracted from the input signal, while $-1-\cos(w)$ denotes a signal (no input).

2.13 Fixing the Number of Input and Output Signals

In the examples above, the input signals were usually defined *implicitly* by the defining FAUST expression:

```
foo = faust_expression; // inputs and outputs determined by expression
```

Sometimes, however, it is helpful to fix the number of input signals. For example, to define a stereo processor, one can begin and end its definition with two wires:

```
foo = _,_ : faust_expression : _,_;
```

Such practice also helps to more easily catch errors when the number of input or output signals comes out wrong in the defining expression.

2.14 Naming Input Signals

Input signals can be given a name by including them as *formal function parameters*. The previous example can be modified to do this as follows:

```
foo(x,y) = x,y : faust_expression : _,_;
```

This option is important to remember when the explicit names are easier to work with than unnamed “incoming wires”:

```
foo(x,y) = faust_expression_with_x_and_y_appearing_in_the_middle_somewhere;
```

2.15 Naming Output Signals

Suppose you have a block diagram `bd` that outputs two signals, and you would like to give them names. You cannot simply say “`x,y = bd`” as you would in matlab. Instead, use the signal blocking operator ‘!’:

```
x = bd : _,!;  
y = bd : !,_;
```

This does not result in two instances of `bd`. The optimization recognizes the common subexpression `bd` and computes it only once.

2.16 Signal Types

The two signal types in FAUST are `int` and `float`. When compiling, a `float` can be elevated to double or quad precision by means of the `-double` or `-quad` options to the FAUST compiler. An `int` is always 32-bit precision, for reasons mentioned in §2.25 below.

In the FAUST language, the type of a signal can be forced (with conversion, if necessary), as follows:

```
int(x)  → integer part of the signal x  
float(x) → treat the signal x as a float in expressions
```

The `int(x)` conversion implements what is normally called *magnitude truncation*, or “rounding toward zero”. Thus, `int(0.99) = int(-0.99) = 0`. Magnitude truncation is often preferred in digital signal processing applications because it normally best preserves the stability of all feedback loops. (When signals have a physical-amplitude interpretation, such as pressure or voltage, then magnitude truncation, unlike rounding, is always *passive*, *i.e.*, not energy-creating.)

Note that integer expressions are automatically converted to float when necessary to avoid a loss of precision. Thus, for example,

```
process = 1/2; // = 0.5, 0.5, 0.5, ...
```

outputs 0.5, while

```
process = int(1/2); // = 0, 0, 0, ...
```

puts out the constant zero signal.

2.17 Signal Comparison Operators

Signals can be *compared* sample by sample to produce an integer-valued signal that is 1 when the comparison is true and 0 otherwise. The comparison operator symbols coincide with those in the C language:

`< <= > >= == !=`

For example, the program

```
process(L,R) = L > R;
```

Produces a signal that is 1 when signal `L` is greater than signal `R` and 0 otherwise.

2.18 Bitwise Operations for Integer Signals

There are also C-style operators for manipulating the bits in each sample of an integer signal:

`<< >> | & xor`

For example, `&` can be used to apply a *bit mask*:

```
ramp = _~+(1); // integer ramp 1,2,3,...
process = ramp & ((1<<8)-1); // restart ramp every 256 samples
```

2.19 Foreign Constants and Variables

FAUST provides linkages to the hosting environment via “foreign” entities. For example, here are some selected declarations from the FAUST Libraries:¹²

```
SR = fconstant(int fSamplingFreq, <math.h>);
BS = fvariable(int count, <math.h>);
tanh = ffunction(float tanhf (float), <math.h>,"");
```

The *foreign constant* in the above example is the audio sampling rate `SR`, which is not known until run-time. It is typically used in FAUST expressions to calculate normalized frequencies `f/SR`. The FAUST compiler assumes that foreign constants are determined at initialization time and never change.

The *foreign variable* example is the audio block size `BS` (or “buffer size” or “inner-loop length”). The FAUST compiler assumes that foreign variables are constant within an inner loop (typically 64 samples), but may change between blocks (like values coming from user-interface widgets).

2.20 Foreign Functions

A *foreign function* is declared as

```
ffunction(<function-declaration>, <include-file>, <library>)
```

where the function-declaration must be of the form

```
<type> fn(<type>);
```

where `<type>` is either `int` or `float`. In addition, the input type can be omitted, indicating no input argument. Thus, `<function-declaration>` means one of the following:

```
int fn(int);
int fn(float);
float fn(int);
float fn(float);
int fn(); // not 'fn(void)'
float fn();
```

¹²These were originally defined in `math.lib`, while now they are in `maths.lib` and expanded in some cases.

2.20.1 Example 1

For example,

```
process = ffunction(float fn(float), "<math.h>", "-lm");
```

compiles to (as of 2020-04-29 using the latest FAUST master-dev branch)

```
...
/* link with : "-lm" */
#include "<math.h>"
...
virtual void compute(int count, FAUSTFLOAT** inputs, FAUSTFLOAT** outputs) {
    FAUSTFLOAT* input0 = inputs[0];
    FAUSTFLOAT* output0 = outputs[0];
    for (int i = 0; (i < count); i = (i + 1)) {
        output0[i] = FAUSTFLOAT(float(fn(float(input0[i]))));
    }
}
...
```

2.20.2 Example 2

The program

```
process = ffunction(int fn(int), "<math.h>", "-lm");
```

compiles to

```
...
/* link with : "-lm" */
#include "<math.h>"
...
virtual void compute(int count, FAUSTFLOAT** inputs, FAUSTFLOAT** outputs) {
    FAUSTFLOAT* input0 = inputs[0];
    FAUSTFLOAT* output0 = outputs[0];
    for (int i = 0; (i < count); i = (i + 1)) {
        output0[i] = FAUSTFLOAT(int(fn(int(float(input0[i])))));
    }
}
...
```

Note the explicit `float` to `int` conversions.

A foreign function can have zero or more input arguments, and it must return a single `FAUSTFLOAT` as output.

2.20.3 Functions from `math.h`

Typical special functions defined in `math.h`, such as `cos` and `sin`, are either native primitive functions in FAUST or may be defined as foreign functions. See `<faustlibraries_distribution>/maths.lib` for the complete list already incorporated or predefined. The list of incorporated primitives is given in §3.5.3 (p. 32) of the FAUST Quick Reference,¹³. All of these functions accept a signal and return a signal computed by applying the function to each sample.

2.21 Parallel and Sequence Macros

For compact specification of large parallel and series arrays of block diagrams, the `par` and `seq` macros are provided:

```
pf = par(i,N,f(i)); // pf = f(0) , f(1) , ... , f(N-1);
sf = seq(i,N,f(i)); // sf = f(0) : f(1) : ... : f(N-1);
```

A very useful example of `par` is in defining the `bus` macro¹⁴

```
bus(n) = par(i,n,_);
process = bus(4) <: bus(12); // quad to 12-channel (see diagram)
```

2.22 Sum and Product Macros

There are similarly `prod` and `sum` macros:

```
pf = prod(i,N,f(i)); // pf = f(0) * f(1) * ... * f(N-1);
sf = sum(i,N,f(i)); // sf = f(0) + f(1) + ... + f(N-1);
```

One often does not need `sum` because of the summing property of `:>_`, and the latter yields a more compact block-diagram drawing.

2.23 Pattern Matching in Faust

In FAUST, *pattern matching* is used in functional rewriting rules:

```
f(pattern) = expression;
```

where `f` is any valid function name, and both `pattern` and `expression` are arbitrary expressions in the FAUST language. Such a definition specifies a *rewriting rule*: When `f(pattern)` is recognized in any function definition, it is replaced by `expression`. Pattern matching is commonly supported in *functional programming languages* such as Haskell [6], a language that influenced the design of FAUST.

Pattern-matching allows different function-definitions for different argument-patterns. For example, the following use of pattern-matching defines different amplitudes and frequencies for a simple additive synthesizer [5]:

¹³<https://github.com/grame-cncm/faust/blob/master-dev/documentation/faust-quick-reference.pdf>

¹⁴The `bus` function was initially defined in the FAUST Libraries distribution's `maths.lib`, and was moved to `signals.lib` near the end of 2016.


```

import("stdfaust.lib");
amp(0) = 1.0; // amplitude of fundamental frequency
amp(1) = 0.5; // amplitude of first harmonic
amp(2) = 0.3;
freq(i) = (i+1)*440;
partial(i) = amp(i) * os.osc(freq(i)); // osc defined in oscillators.lib
process = sum(i, 3, partial(i));

```

The ‘0’ in `amp(0)` above is referred to as a *single-value pattern*, as are ‘1’ and ‘2’ in the next two lines. Each single-value pattern matches only one value exactly. The ‘i’ in `freq(i)` above may be called a *free variable* in the pattern (consisting of only that variable), and it matches anything at all (any block diagram in the FAUST language).

Pattern-matching can be used to define functions *recursively*. A simple example is the recursive definition of factorial:

```

fact(0) = 1;
fact(n) = n * fact(n-1);
process = fact(4); // output signal = 24,24,24,...

```

While FAUST function definitions can appear in any order, *lexical order matters for pattern definitions*. Thus, in the above factorial example, the rewrite rule using the single-value pattern 0 is tried first, while the one using the variable pattern `n` matches for *any block diagram* (which should, in our intended use, and to avoid an infinite loop, evaluate to an integer each sample). If the first two lines are interchanged in the above example, an infinite loop is obtained at compile time.

Another example [5] is the `fold` operator:

```

fold(1,f,x) = x(0);
fold(n,f,x) = f(fold(n-1,f,x),x(n-1));

```

Then in the additive synthesis example above, `sum(i, 3, partial(i))` can be replaced by `fsum(3,partial)` where `fsum(n) = fold(n,+)`.

More general expressions can appear in a pattern definition, as described in §2.23.4 below.

2.23.1 Formal Parameter Exception

The pattern-matching facility is not applied to ordinary formal function parameters [5]. In other words, `f(x,y,z)=expression` is treated as a function having three formal parameters that are expected to appear literally in `expression` (e.g., an expression such as `x*y+z`). This interpretation is in contrast to a function whose input is three parallel block diagrams of arbitrary generality. As a result of this exception, the mere *number* of formal parameters does not contribute to the uniqueness of a pattern. For example, the following program generates a compile-time error:

```

f(x,y) = f(x) + f(y); // (x,y) => f(x),f(y):+
f(x) = 2*x;           // (x) => 2,x:*
process = f(3,5);

```

The compiler-error triggered is “inconsistent number of parameters in pattern-matching rule: (x) => 2,x:*; previous rule was (x,y) => f(x),f(y):+”. On the other hand, the following program outputs the constant signal 16:

```
f((x,y)) = f(x) + f(y);
f(x) = 2*x;
process = f((3,5));
```

The extra parentheses distinguish the pattern (x,y) from formal parameters x,y in this case.

As another example, the following program also outputs the constant signal 16:

```
f(x*y) = f(x) + f(y);
f(x) = 2*x;
process = f(3*5);
```

Since the expression $x*y$ does not look like a list of formal parameters, it doesn't need additional parentheses.

2.23.2 Recursive Block Diagram Specification

Recursive pattern matching, introduced and illustrated for computing factorial in §2.23 above, also gives a powerful way to define a block diagram recursively in terms of its partitions. For example, the following FAUST program defines Hadamard matrices of order 2^n where n is a positive integer:

```
bus(n) = par(i,n,_); // There is si.bus(n) in the \FL s
//hmtx(2) = _,_ <: +,-; // scalar butterfly
hmtx(2) = _,_ <: (bus(2):>_),(_,*(-1):>_) ; // prettier drawing
hmtx(n) = bus(n) <: (bus(n):>bus(n/2)) , // vector butterfly
              ((bus(n/2),(bus(n/2):par(i,n/2,*(-1)))) :> bus(n/2))
              : (hmtx(n/2) , hmtx(n/2));
process = hmtx(16); // look at the diagram in the Faust Editor, e.g.
```

Other examples include Feedback Delay Networks (FDN) and the square waveguide mesh of order 2^n defined in the FAUST Libraries (see `fdnrev0` in `reverbs.lib` and `mesh_square()` in `misceffects.lib`). An especially interesting example is the Fast Fourier Transform (`fft` defined in `analyzers.lib`).

See also `basics.lib` for examples of using recursion for a kind of *list processing*, such as in `count` and `take`.

Note that it is also possible to implement counterparts to `par` and `seq` using pattern matching (see the `duplicate` function on page 14 of the FaustQR).

2.23.3 Understanding count and take from basics.lib

In `basics.lib`, we have the following definition of `count`:

```
count ((xs, xxs)) = 1 + count(xxs);
count (xx) = 1;
```

This definition uses *pattern matching* to count the number of block diagrams in parallel. For example `count((6,5,4))` evaluates to 3. The first pattern recognizes a parallel arrangement of two block diagrams, while the second pattern will match any block diagram. In the multi-element case, the list is parsed as its first element in parallel with a block diagram consisting of all remaining elements (analogous to `CAR` and `CDR` in the Lisp programming language). Note that `(a,b,c,d)` matches `(xs,xxs)` as `((a),(b,c,d))`.

Also in `basics.lib`, we have the following definition of `take`:

```

take (1, (xs, xxs)) = xs;
take (1, xs) = xs;
take (nn, (xs, xxs)) = take (nn-1, xxs);

```

This definition uses pattern matching to return the specified element. For example `take(2, (6,5,4))` yields 5. The extra parentheses around `(xs,xxs)` avoid the structure of mere formal arguments separated by commas.

Note that `take` is 1-based while `seq` and `par` et al. are 0-based.

2.23.4 Pattern Matching Implementation

The pattern matching facility in FAUST operates on block-diagram expressions in FAUST Block-Diagram Normal Form (BDNF), which is the low-level FAUST expression format appearing in compile-time errors and drawn in scalable vector graphics (.svg) files generated by the `faust -svg` option.

BDNF expressions can be viewed as trees. The *leaves* of these trees are numbers and primitive block diagrams such as `+`, `-`, `!`, `abs`, `sin`, etc. The *nodes* of the trees are the five operations of the Block-Diagram Algebra (BDA) (`,` `:` `<:` `>:` `~`).

A *pattern* is a FAUST expression optionally containing free variables. A FAUST expression is a pattern when it appears as a function argument on the left-hand side of a function definition:

```
f(pattern) = expression;
```

Such function definitions specify *rewriting rules* such that when `f(pattern)` is recognized elsewhere, it is replaced by `expression` with any free variables in the pattern replaced by what they matched in the expression. If there are no free variables in the pattern, then the pattern will only match block diagrams whose BDNF is identical.

For example, the pattern `(a)` consists of only the free variable `a`, which will match any expression. The pattern `(2*a) = (2,a:*)` can be represented as a tree consisting of a `:` node at the top, a `,` node as its left child, and the `*` operator leaf as its right child. The `,` node in turn has the left-leaf `2` and right-leaf `a` (a free variable).

2.23.5 Using Pattern Matching in Rewriting Rules

As mentioned above, rewriting rules are specified in FAUST source by function definitions of the form

```
f(pattern) = expression;
```

where `f` is any valid function name, and both `pattern` and `expression` are arbitrary expressions in the FAUST language. The FAUST compiler stores all rewriting rules in the lexical order they were specified (since lexical order determines pattern-matching precedence). When an instance of `f(arg)` is encountered in the FAUST source, the argument `arg` is compared, in Block Diagram Normal Form, to the first defined `pattern` for `f`, also in BDNF. The nodes of the BDNF are compared and traversed in the standard order (top-down, left-to-right), and the match is successful when (1) all nodes and non-variable leaves match literally, and (2) the free variables in the `pattern` (if any) “greedily” match subtrees in `arg`. After an unsuccessful match, additional patterns for `f` are tried, until a match is found. After a successful match, any free variables in `pattern` are bound to their matching subtrees in `arg`, and `expression` is evaluated and inserted in place of `f(arg)`. We will illustrate an example below using Lisp tree syntax.

2.23.6 Using Lisp Syntax to Express Trees

Lisp syntax is nice for expressing tree structure in linear text. A *Lisp expression* has the form

```
expr = (functionName expr1 expr2 ... exprN)
```

where `functionName` names a function (analogous to a procedure or subroutine in other languages), and `expr1...exprN` are the N function arguments, each of which is a Lisp expression itself. So, in Lisp, an example three-level tree consisting of one parent, two children, and five grandchildren, could look like

```
tree = (topNode
        (leftChild
          (leftLeftGrandChild leftMiddleGrandChild leftRightGrandChild))
        (rightChild
          (rightLeftGrandChild rightRightGrandChild)))
```

2.23.7 Pattern-Matching Example

Does the pattern `(a:b)` match `(2+3)`? Yes, as we will see below.¹⁵

Rewriting `2+3` in BDNF gives `((2,3):+)`. Expressing this in Lisp form gives

```
2+3 -> ((2,3):+) -> (: (, 2 3) (+))
```

The arguments to the function `'.'` are the expressions `'(, 2 3)'` and `'(+)'`. The arguments to the function `'.'` are `'2'` and `'3'`.

Rewriting the pattern `(a:b)` in Lisp form gives

```
(a:b) -> (: a b)
```

Since both patterns are of the form `(: a b)`, the patterns *match*.

2.23.8 Pattern-Matching Algorithm Description

The pattern matching algorithm can be roughly recursively defined as follows:¹⁶

```
// match (pattern, expression) -> bool
match (v , E) = true      a pattern-matching-variable matches any expression
match (E , E) = true      two identical expressions match
match ((op P1, P2), (op E1 E2)) = true
    if match(P1, E1) and match(P2, E2), false otherwise
match (P, E) = false
```

In other words, a pattern P matches an expression E if we can replace the free variables v_1, v_2, \dots in P with subexpressions E_1, E_2, \dots from E to make it identical to E . That is, P matches E if there exist E_1, E_2, \dots such that $P[v_1=E_1, v_2=E_2, \dots] == E$.

¹⁵Thanks to Yann Orlarey for this example and associated discussion on how pattern-matching works in FAUST.

¹⁶Thanks to Yann Orlarey for this description.

2.23.9 Miscellaneous Pattern-Matching Examples

Decide what you think `process` is defined as in the examples below, and then find the answers at the end:

Example 1:

```
a = 2; // this definition is not used
f(a,b) = a+b; // a is a formal argument that is used
process = f(3,4); // see below for the answer
```

Example 2:

```
a = 2; // this definition is used (not shadowed by formal arg)
f(c*b) = a+b;
process = f(3*4); // see below
```

Example 3:

```
a = 2; // not used - shadowed by pattern variable
f(a*b) = a+b;
process = f(3*4);
```

Answers: 7, 6, 7

2.24 Scope Rules

As illustrated in §3 below, a `with{...};` block may be used to define local symbols (block diagrams). Otherwise, everything is globally defined.

A library “foo.lib” may be loaded into its own namespace by writing

```
f = library("foo.lib");
```

and then symbols from that library may be accessed using the given prefix:

```
fPI = f.PI; // use definition of PI given in foo.lib
fTanZero = f.tan(0); // use definition of tan() given in foo.lib
```

An *environment* works similarly:

```
e = environment {
  Phi = 0.5*(1.0+sqrt(5));
}
golden_ratio = e.Phi;
```

For convenience, “`component("prog.dsp")`” is defined to mean the same thing as “`library("prog.dsp").process`”.

Definitions within an environment can be replaced (or appended) using the following bracket syntax:

```
f = library("filters.lib")[pole(p) = _];
process = f.dcblocker; // now it is one-zero, no pole
```

2.25 White Noise Generator

The FAUST noise generator defined in `noises.lib` is an instructive example. It generates uniform pseudo-random white noise in $[0, 1)$ by the linear congruential method.¹⁷

```
random = +(12345) ~ *(1103515245); // overflowing mpy & offset
RANDMAX = 2147483647.0;
noise = random / RANDMAX;
```

Note that for this noise-generator to give identical results on all platforms, FAUST must define integers as 32 bits everywhere, and overflow behavior must be normalized across platforms as well.

2.26 Further Readings on the Faust Language

It is important to note that this brief overview is *not complete*.

See the FAUST Manual¹⁸ and the FAUST Quick Reference,¹⁹ for more examples, features of the language, and discussion. Historically, the FAUST Quick Reference document is the only fully up-to-date specification of the latest features of the language.

New since this tutorial was written is an online description of FAUST syntax:
<https://faustdoc.grame.fr/manual/syntax/>.

2.27 Acknowledgment

Thanks to Yann Orlarey for helpful clarifications regarding the FAUST language.

3 A Simple Example Faust Program

Figure 5 lists a small FAUST program specifying the constant-peak-gain resonator discussed in [10].

```
process = firpart : + ~ feedback
with {
  bw = 100; fr = 1000; g = 1; // parameters - see caption
  SR = fconstant(int fSamplingFreq, <math.h>);
  pi = 4*atan(1.0); // circumference over diameter
  R = exp(-pi*bw/SR); // pole radius
  A = 2*pi*fr/SR; // pole angle (radians)
  RR = R*R;
  firpart(x) = (x - x'') * g * (1-RR)/2;
  // time-domain coefficients ASSUMING ONE-SAMPLE FEEDBACK DELAY:
  feedback(x) = 0 + 2*R*cos(A)*x - RR*x';
};
```

Figure 5: FAUST program specifying a constant-peak-gain resonator. Input parameters are resonance frequency `fr` (Hz), resonance bandwidth `bw` (Hz), and desired peak-gain `g`.

¹⁷The mathematical notation $[a, b)$ denotes a *half-open interval*, i.e., the interval includes endpoint a but not b .

¹⁸<https://faustdoc.grame.fr>

¹⁹<https://github.com/grame-cncm/faust/blob/master-dev/documentation/faust-quick-reference.pdf>

We will now study this example in a variety of ways. First we will illustrate the typical development cycle (look at the block diagram, etc.) Second, we will add a GUI and look at some of the various types of applications and plugins that can be generated from it.

4 Verifying and Testing Faust Programs

It takes a bit of experience to write a correct program on the first try. Therefore, we often have to *debug* our programs by some technique. Typically, inspecting the automatically generated block diagrams and listening to the results are tools enough for debugging FAUST source code.

4.1 Generating Faust Block Diagrams

A good first check on a FAUST program (after getting it to compile) is to generate its block diagram using the `-svg` option.²⁰ For example, the command

```
> faust -svg cpgr.dsp
```

creates a subdirectory of the current working directory named `cpgr-svg` which contains a “scalable vector graphics” (`.svg`) file for each block-diagram expression in `cpgr.dsp`. For this example, there is a block diagram generated for the `process` line, and for each of the last five lines in the `with` clause (not counting the comment).

Figure 6 shows the block diagram generated for the main `process` block from Fig. 5:

```
process = firpart : + ~ feedback
```

The dot on each block indicates its standard orientation (analogous to a “pin 1” indicator on an integrated circuit chip). The small open square at the beginning of the feedback loop indicates a *unit sample delay* introduced by creating a signal loop. Needless to say, it is important to keep track of such added delays in a feedback loop.

Figure 7 shows the block diagram generated for the `firpart` abstraction:

```
firpart(x) = (x - x'') * g * (1-RR)/2;
```

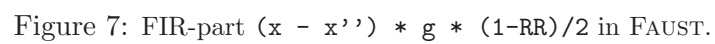
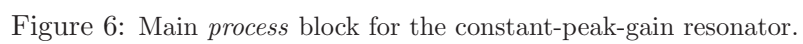
Similarly, Fig. 8 shows the block diagram generated for the feedback path:

```
feedback(x) = 0 + 2*R*cos(A)*x - RR*x';
```

If not for the added sample of delay in the feedback loop (indicated by the small open square in Fig. 6), the feedback-path processing would have been instead $0 + 2*R*cos(A)*v' - RR*v''$.

Note that the block diagrams are drawn as though all details of the expression are to be evaluated every sample. However, the FAUST compiler instead computes constant expressions at `init` time and allocates memory locations for them. More generally, the FAUST compiler separately optimizes full-rate signals at the sampling rate (calculated in the inner loop), slowly varying signals (updated at the “buffer rate” outside of the inner loop—currently every 64 samples), and constant signals (evaluated once at initialization time).

²⁰The `faust2firefox` script can be used to generate SVG block diagrams and open them in the Firefox web browser, among others.



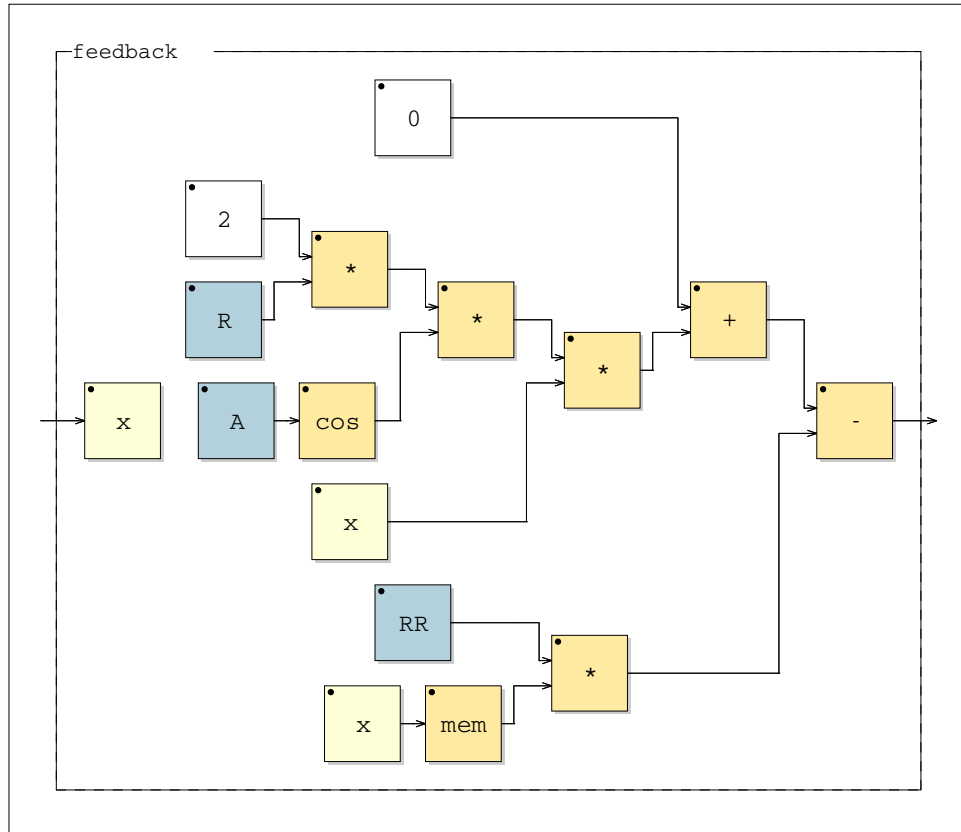


Figure 8: Feedback block $0 + 2 \cdot R \cdot \cos(A) \cdot x - RR \cdot x'$ in FAUST.

4.2 A Look at the Generated C++ code

One normally never needs to look at the C++ code generated by FAUST. However, we will do this now just to see how it looks, and note a few things.

Running FAUST with no architecture file, *e.g.*,

```
> faust cpgr.dsp
```

causes the C++ signal-processing code to be printed on the standard output, as shown for this example in Fig. 9.

We see that `init` calls `classInit`, which is where read-only wavetables are initialized (none being used in this example), followed by `instanceInit`, which resets all parameters to their default values. Thus, `instanceInit` provides a more efficient processor “reset” when readonly wavetables are in use.

Since all processor state is allocated as instance variables of the `mydsp` class (which can be changed to any name using the `-cn` FAUST-compiler option), there is no allocation in `init`.

Notice how constant subexpressions, such as for `fconst0`, are computed only once in `instanceInit`. The template `faustpower<2>(x)` (omitted in the above listing) expands to `x*x`, thereby avoiding calling the `pow` function. In general, FAUST does a lot of such optimization.

The `buildUserInterface` method calls the appropriate interface function for each control widget (slider, button, etc.), but there are none in this simple example. In §5 we will add GUI controls, and you compile that to see how `buildUserInterface` changes as a result. The GUI control variables are also included among the processor state variables, and the interface is given pointers to them. (The interface holds no signal-processing state, including both signal and controller values.) The interface may update the control variables asynchronously (*e.g.*, in another thread of execution), and they will get sampled in the signal processor once per execution of the `compute` inner loop. Thus, the *control rate* is the sampling rate divided by the audio buffer length `count`. As a result, elaborate FAUST expressions in the control variables are normally very inexpensive computationally. For optimization, we tend to look hard only at the `for` loop in the `compute` function; for example, we generally try to avoid calls to `libc` for things like `sin()` and `cos()`, which can be relatively slow. There are fast approximate alternatives such as the `fastapprox` library,²¹ and linearly interpolated lookup tables are often used.

4.3 Printing/Plotting the Output Signal(s)

Sometimes, beyond inspecting the block diagram, it may be necessary to verify the output signal(s) in more detail. For this purpose, FAUST has a useful “architecture file” named `plot.cpp` which results in generation of a main C++ program that simply prints the output signal(s) to the standard output. This printout can be used to plot the output (using, *e.g.*, `gnuplot`) or compare it to the output of some other program. A similar architecture file, `matlabplot.cpp`, results in a program that outputs an input file for Matlab or Octave that will define a matlab matrix containing each FAUST output signal in a column of the matrix. These techniques are discussed further in the following subsections.

This section gives an example of comparing the impulse response of the filter in Fig. 5 to the output of a matlab version. Specifically, we will compare the printed output from the Faust-generated program to the output of the matlab test program shown in Fig. 10.

²¹<https://fastapprox.googlecode.com/svn/tags/fastapprox>

```

class mydsp : public dsp {
private:
    float fConst0;    float fConst1;
    float fVec0[3];   float fConst2;
    float fRec0[3];
public:
    static void metadata(Meta* m) { }
    virtual int getNumInputs() { return 1; }
    virtual int getNumOutputs() { return 1; }
    static void classInit(int samplingFreq) { }
    virtual void instanceInit(int samplingFreq) {
        fSamplingFreq = samplingFreq;
        fConst0 = expf((0 - (314.1592653589793f / float(fSamplingFreq))));
        fConst1 = (2 * cosf((6283.185307179586f / float(fSamplingFreq))));
        fConst2 = (0.5f * (1 - faustpower<2>(fConst0)));
        for (int i=0; i<3; i++) fVec0[i] = 0;
        for (int i=0; i<3; i++) fRec0[i] = 0;
    }
    virtual void init(int samplingFreq) {
        classInit(samplingFreq);
        instanceInit(samplingFreq);
    }
    virtual void buildUserInterface(UI* interface) {
        interface->openVerticalBox("cpgr");
        interface->closeBox();
    }
    virtual void compute (int count, FAUSTFLOAT** input, FAUSTFLOAT** output) {
        FAUSTFLOAT* input0 = input[0];
        FAUSTFLOAT* output0 = output[0];
        for (int i=0; i<count; i++) {
            float fTemp0 = (float)input0[i];
            fVec0[0] = fTemp0;
            fRec0[0] = ((fConst2 * (fVec0[0] - fVec0[2]))
                + (fConst0 * ((fConst1 * fRec0[1]) - (fConst0 * fRec0[2]))));
            output0[i] = (FAUSTFLOAT)fRec0[0];
            // post processing
            fRec0[2] = fRec0[1]; fRec0[1] = fRec0[0];
            fVec0[2] = fVec0[1]; fVec0[1] = fVec0[0];
        }
    }
};

```

Figure 9: C++ code emitted by “faust cpgr.dsp”, slightly reformatted.

```

SR = 44100; % Sampling rate

fr = 1000; % Resonant frequency
bw = 100; % Bandwidth
g = 1; % Peak gain
N = 10; % Samples to generate in test

R = exp(-pi*bw/SR); % pole radius
A = 2*pi*fr/SR; % pole angle (radians)
firpart = g * [1 0 -1] * (1-R^2)/2;
feedback = [1 -2*R*cos(A) R^2]; % freq-domain coeffs
freqz(firpart,feedback); % freq-response display
format long;
h = impz(firpart,feedback,N) % print impulse response

```

Figure 10: Constant Peak-Gain Resonator—matlab version

In our FAUST program, we need a test impulse, *e.g.*,

```

process = 1-1' : firpart : + ~ feedback
with { ... <same as before> ... };

```

The signal $1 = [1, 1, 1, \dots]$ is the unit-step signal consisting of all ones, and $1' = [0, 1, 1, \dots]$ is the unit step delayed by one sample. Therefore, $1-1'$ is the impulse signal $\delta = [1, 0, 0, \dots]$.

Suppose the file `cpgrir.dsp` (“Constant-Peak-Gain Resonator Impulse-Response”) contains our test FAUST program. Then we can generate the impulse-response printout as follows at the command line:

```

> faust -a plot.cpp -o cpgrir-print.cpp cpgrir.dsp
> g++ -Wall -g -lm -lpthread cpgrir-print.cpp -o cpgrir-print
> cpgrir-print -n 10

```

(Commands similar to the first two lines above are carried out more conveniently using the `faust2plot` utility distributed with Faust.) The first line generates the C++ program `cpgrir.cpp` from the FAUST source file `cpgrir.dsp` using the architecture file `plot.cpp`. The second line compiles the C++ file to produce the executable program `cpgrir-print`. Finally, the third line generates and prints the first 10 samples of the output signal (anything more than the number of filter coefficients is usually enough), which is our desired impulse response:²²

```

h = [    0.00707331  0.0139039  0.013284
        0.012405   0.0112882  0.00995947
        0.00844865  0.00678877  0.00501544
        0.00316602    ... ]

```

The matlab version produces the following impulse response:

²²This specific output was obtained by editing `cpgrir-print.cpp` to replace `%8f` by `%g` in the print statements, in order to print more significant digits.

```

h =
[ 0.00707328459864603 0.01390382707778288 0.01328399389241600
  0.01240496991806334 0.01128815312793390 0.00995943544693653
  0.00844861689634155 0.00678874919376101 0.00501542304704597
  0.00316601431505539 ... ]

```

Since matlab uses double-precision floating-point while FAUST used single-precision `floats` in this example, we see differences after six or so decimal digits. The precision of the `float` type in FAUST can be extended to double or quad by changing the compile line as follows:

```

> faust -double ...
> faust -quad ...

```

4.4 Inspecting the Output Signal(s) in Matlab or Octave

The `faust2octave` script, distributed with FAUST, executes shell commands similar to the `faust2plot` script mentioned above, then executes the generated program to write a matlab input file, and finally loads the file in Octave. The result is typically as if the following commands were typed for the above example:

```

> faust -a matlabplot.cpp cpgrir.dsp -o cpgrir.cpp
> g++ -O3 cpgrir.cpp -o cpgrir
> cpgrir -n 600 > cpgrir.m
> octave --persist cpgrir.m

```

In Octave, the variable `faustout` is a matrix containing the program output. Each output signal is a column of this matrix. In the above example, we have one output signal that is 600 samples long, so the `faustout` matrix is a 600×1 column vector.

In Octave, an overlay of all output signals can be plotted by the command

```
octave:1> plot(faustout);
```

Very often in signal processing we need to see the spectrum of the signal:

```
octave:1> plot(20*log10(abs(fft(faustout,1024))(1:512,:))));
```

In this example, the signal is zero-padded out to 1024 samples, a Fast Fourier Transform (FFT) is performed, the first 512 samples are selected, the absolute value is taken, followed by conversion to dB, and finally this dB spectral magnitude is plotted. If there are multiple output signals, their dB-magnitude spectra are all plotted overlaid.

4.5 Summary of Faust Program Testing Strategies

The development of a FAUST program `p.dsp`, say, typically consists of the following steps:

```

> faust p.dsp          # does it compile?
> faust2firefox p.dsp  # check the block diagram
> faust2octave p.dsp   # (maybe) inspect the output signal(s) in Octave
> faust2jaqt p.dsp     # make a JACK-compatible application

```

These operations are so common that I have shell aliases `f`, `f2ff`, `f2o`, and `f2j` for these commands. More recently, the first two may be replaced by the Faust Editor. The third (`f2o`) is typically only used for serious testing, such as for a publication.

5 Adding a GUI

To illustrate automatic generation of user-interface controls, we will add two “numeric entry” fields (`nentry`) and one “horizontal slider” (`hslider`) to our example of Fig. 5. These controls will allow the application or plugin user to vary the center-frequency, bandwidth, and peak gain of the constant-peak-gain resonator in real time. A complete listing of `cpgrui.dsp` (“Constant-Peak-Gain Resonator with User Interface”) appears in Fig. 11.

```
declare name "Constant-Peak-Gain Resonator";
declare author "Julius Smith";
declare version "1.0";
declare license "GPL";

/* Controls */
fr = nentry("frequency (Hz)", 1000, 20, 20000, 1);
bw = nentry("bandwidth (Hz)", 100, 20, 20000, 10);
g  = hslider("peak gain", 1, 0, 10, 0.01);

/* Constants (FAUST provides these in math.lib) */
SR = fconstant(int fSamplingFreq, <math.h>);
PI = 3.1415926535897932385;

/* The resonator */
process = firpart : + ~ feedback
with {
  R = exp(-PI*bw/SR); // pole radius
  A = 2*PI*fr/SR;     // pole angle (radians)
  RR = R*R;
  firpart(x) = (x - x'') * g * (1-RR)/2;
  // time-domain coefficients ASSUMING ONE-SAMPLE FEEDBACK DELAY:
  feedback(v) = 0 + 2*R*cos(A)*v - RR*v';
};
```

Figure 11: Listing of `cpgrui.dsp`—a FAUST program specifying a constant-peak-gain resonator with three user controls. Also shown are typical header declarations.

Note that GUI element specifications such as “`nentry(<string>,<number>,...)`” and “`hslider(<string>,<number>,...)`” should be regarded as predefined block diagrams having one *slow* output (and also a signal input, in the case of `vbargraph()` and `hbargraph()`, and the string and number arguments are all compile-time constants. There is no partial application (§2.8 on page 11) or alternate notations such as “`widget(x) = <string>,<number>,x:widget`” for GUI elements, principally because strings are not elements of the FAUST language. While output-only GUI elements have no alternate syntax, bar graphs have two equivalent forms, *e.g.*, `x:vbargraph(<string>,<min>,<max>)` and `vbargraph(<string>,<min>,<max>)(x)`.

Because GUI widget outputs are “slow”, expressions involving them are moved out of the inner-loop by the compiler, which is very helpful for reducing CPU load. We can then choose audio block size to trade CPU load against GUI responsiveness. Thus, the “control rate” (often called “K rate”, at least since Csound) equals the audio sampling rate divided by audio block size.

6 Generating Stand-Alone Qt or GTK Applications

The next step after debugging a FAUST program is typically generating the desired application or plugin. For example,

```
> faust2jaqt p.dsp      # make a standalone JACK-compatible Qt application
> faust2jack p.dsp      # make a standalone JACK-compatible GTK application
```

where `p.dsp` is the FAUST program to be compiled. On the Mac, each of the above commands would create `p.app`. On a Linux system, the binary executable program `p` would be created.

`faust2jaqt` and `faust2jack` are convenience scripts distributed with FAUST.²³ A screen-shot of the Qt main window (obtained using `Grab.app` on the Mac) is shown in Fig. 12.



Figure 12: Main (and only) window of a Qt application generated by `faust2jaqt` from `cpgrui.dsp` on a Mac OS X system.

When the application is run, it automatically binds its outputs to the system output if JACK is running (and it will exit if JACK is not running!). In a Linux environment, it is necessary to manually connect the program output to the system audio outputs. JACK may be conveniently started on Mac OS X using JackPilot, and on Linux systems using `qjackctl`.

7 Generating Other Applications and Plugins

The `faust` compiler translates the FAUST language to C++. Using its architecture files (written in C++) and convenience scripts (such as `faust2jaqt`), working tests, applications, and plugins can be quickly generated from FAUST source. Above we looked at using `faust2plot`, `faust2matlabplot`, `faust2firefox`, `faust2octave`, `faust2jaqt`, and `faust2jack` on a simple example. There are many others. For the latest list, `cd` to the `faust/tools/faust2appls/` directory and list its contents. At the time of this writing (May 2020), the result is as follows:

²³The author has verified (July 2010) that working Qt applications are generated on both Mac OS X and Fedora 12 Linux systems, and working GTK applications are generated on Fedora 12 Linux.

Directory faust/tools/faust2appls/			
Info.plist	faust2firefox	faust2octave	faust2svgviewer
README.md	faust2gen	faust2osxiosunity	faust2teensy
check-au.sh	faust2graph	faust2owl	faust2unity
encoderunitypackage	faust2graphviewer	faust2paqt	faust2vst
faust2alqt	faust2ios	faust2pdf	faust2vsti
faust2alsa	faust2jack	faust2plot	faust2w32max6
faust2alsaconsole	faust2jackconsole	faust2png	faust2w32msp
faust2android	faust2jackrust	faust2portaudiorust	faust2w32puredata
faust2androidunity	faust2jackserver	faust2pure	faust2w32vst
faust2api	faust2jaqt	faust2puredata	faust2w64max6
faust2atomsnippets	faust2jaqtchain	faust2raqt	faust2w64vst
faust2au	faust2juce	faust2ros	faust2wasm
faust2bela	faust2ladspa	faust2rosgtk	faust2webaudiowasm
faust2caqt	faust2linuxunity	faust2rpialsaconsole	faust2webaudiowast
faust2caqtios	faust2lv2	faust2rpinetjackconsole	faust2winunity
faust2csound	faust2mathdoc	faust2sam	faustoptflags
faust2csvplot	faust2mathviewer	faust2sig	faustpath
faust2dssi	faust2max6	faust2sigviewer	filename2ident
faust2dummy	faust2md	faust2smartkeyb	readme-faust2au.txt
faust2dummymem	faust2msp	faust2sndfile	unsupported
faust2eps	faust2netjackconsole	faust2soul	usage.sh
faust2esp32	faust2netjackqt	faust2supercollider	
faust2faustvst	faust2nodejs	faust2svg	

These shell scripts are easily read to find out how each one works. Check out in particular the options supported (each script should accept a `-h` option that prints out a summary of options supported).

Additional information is found in the `faust/examples` directory. See also the FAUST Libraries documentation somewhere under <https://faust.grame.fr/>.²⁴

8 Generating a LADSPA Plugin via Faust

LADSPA stands for “Linux Audio Developer Simple Plugin API”, and it is a common audio plugin API for Linux applications. [Note: LADSPA has been superseded by LV2, and this section should be updated to show how to create lv2 plugins instead. See the `faust2lv2` script for details. It’s not significantly different at this level.] It can be considered the Linux counterpart of the widely used VST plugin standard for Windows applications. In the Planet CCRMA distribution, most of the LADSPA plugins are found in the directory `/usr/lib/ladspa/`. At the time of this writing, there are 161 audio plugins (`.so` files) in or under that directory.

To generate a LADSPA plugin from FAUST source, it is merely necessary to use the `ladspa.cpp` architecture file, as in the following example:

²⁴It is inadvisable to publish links into content within the FAUST website, even something as large as the manual, editor, or library documentation, as they frequently change incompatibly. Fortunately, the language rarely has an incompatible change.


```

> faust -a ladspa.cpp cpgrui.dsp -o cpgruilp.cpp
> g++ -fPIC -shared -O3 \
    -Dmydsp='Constant_Peak_Gain_Resonator' \
    cpgruilp.cpp -o cpgruilp.so
> cp cpgruilp.so /usr/local/lib/ladspa/

```

(Recall that `cpgrui.dsp` was listed in Fig. 11 on page 30.) We see that the C++ compilation step calls for “position-independent code” (option `-fPIC`) and a “shared object” format (option `-shared`) in order that the file be dynamically loadable by a running program. (Recall that `pd` similarly required its externals to be compiled `-shared`.) The FAUST distribution provides the make file `/usr/lib/faust/Makefile.ladspacompile` (among others) which documents such details.

Many Linux programs support LADSPA programs, such as the sound editor Audacity, the multitrack audio recorder/mixer Ardour, and the sequencer Rosegarden. However, for our example, we’ll use a simple application-independent LADSPA effects rack called JACK Rack (select “Applications / Planet CCRMA / Jack / JACK Rack”).

Figure 13 shows the appearance of the `jack-rack` main window after adding²⁵ the plugin named `Constant_Peak_Gain_Resonator`. Note that the two numeric entry fields have been converted to horizontal sliders. (Vertical sliders are also converted to horizontal.) Also, the controller names have been simplified. A bug is that the default values for the controls are not set correctly when the plugin loads. (They were set manually to obtain Fig. 13 as shown.)

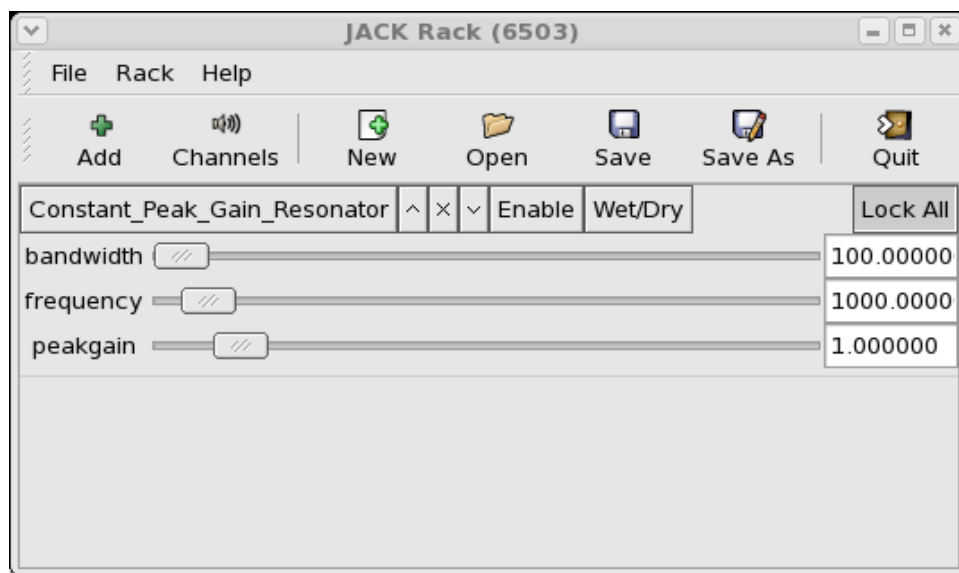


Figure 13: JACK Rack screenshot after adding the LADSPA plugin `Constant_Peak_Gain_Resonator`. Additional LADSPA plugins can be loaded in the space below (and connected in series).

To test the LADSPA plugin, any program’s audio output can be routed through `jack-rack` to the sound-out driver (typically “ALSA PCM” these days). For example, `pd`’s audio output can be

²⁵After running `jack-rack`, the LADSPA plugin was added by clicking on the menu items “Add / Uncategorized / C / Constant_Peak_Gain_Resonator”. If `jack-rack` does not find this or other plugins, make sure your `LADSPA_PATH` environment variable is set. A typical setting would be `/usr/local/lib/ladspa:/usr/lib/ladspa/`.

routed through jack-rack to alsa_pcm as shown in Fig. 14.²⁶

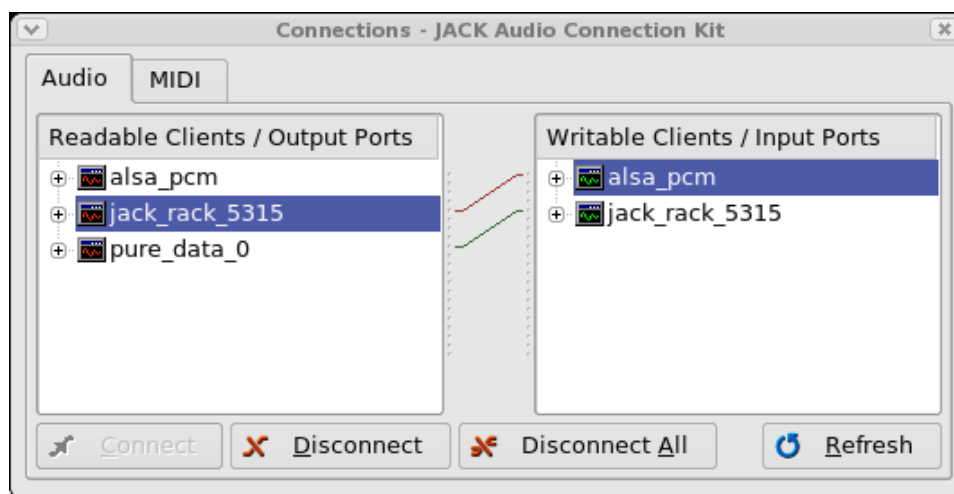


Figure 14: JACK audio connections routing pd through jack-rack to the ALSA sound-out driver alsa_pcm.

9 Feeding Soundfiles to Faust Standalone Apps

The FAUST standalone apps we have considered thus far expect sound input in real time, such as from your computer’s microphone input. Sometimes it is handy to be able to feed a prerecorded soundfile instead. Below we will discuss two approaches to soundfile processing in FAUST:

1. **Offline Processing** — an output soundfile is created from the input soundfile and no time-varying manipulation of processing parameters is possible, much like when running `sox` on a soundfile.
2. **Real Time Processing** — the input soundfile is processed in real time, allowing parameter manipulation and audio monitoring.

9.1 Offline Processing of Soundfiles in Faust

If you have the FAUST distribution and `libsndfile` installed on your computer, then you can simply say

```
> faust2sndfile myprocessor.dsp
> myprocessor input.wav output.wav
```

²⁶Sound routings such as this may be accomplished using the “Connect” window in `qjackctl`. In that window, there is an Audio tab and a MIDI tab, and the Audio tab is selected by default. Just click twice to select the desired source and destination and then click “Connect”. Such connections can be made automatic by clicking “Patchbay” in the `qjackctl` control panel, specifying your connections, saving, then clicking “Activate”. Connections can also be established at the command line using `aconect` from the `alsa-utils` package (included with Planet CCRMA).

to produce soundfile `output.wav` from `input.wav` using `myprocessor.dsp`. The first line uses the shell script `faust2sndfile`, normally installed in `/usr/local/bin/`, to create the binary program `myprocessor` from FAUST source, and the second line runs `myprocessor` on `input.wav`, writing the processed signal to `output.wav`.

You can also set FAUST parameters (such as sliders and buttons) on the command line, and generate a soundfile with no input file:

```
> mysynth -duration 1 -gate 1 -gain 0.5 -freq 440 output.wav
```

where of course `myprocessor.dsp` defines these UI elements. Note that the parameter names are made lower case and have spaces and such removed; you can find out what they became using the `-help` option:

```
> myprocessor -h
*** USAGE: myprocessor input_sfile output_sfile
...
-duration [0..1 ]
-gate [0..1 ]
-gain [0..1 ]
-freq [20..10000 ]
```

Thus, the name of each parameter is printed along with its range.

Under the hood, the `faust2sndfile` script uses the FAUST architecture file `sndfile.cpp`, which in turn uses Erik de Castro Lopo's `libsndfile` library to process the channels of an input soundfile, producing a single output sound file containing all of the output channel signals.

For example, suppose the file `gain-stereo.dsp` contains the one-line FAUST program

```
process = *(0.5),*(0.5);
```

Then the following command will compile it to C++:

```
> faust -a sndfile.cpp gain-stereo.dsp > gain-stereo.cpp
```

The file `gain-stereo.cpp` contains

```
#include <sndfile.h>
```

which is installed with the `libsndfile` distribution. The main function in `gain-stereo.cpp` is as follows:

```
int main(int argc, char *argv[] )
{
    SNDFILE* in_sf;
    SNDFILE* out_sf;
    SF_INFO in_info;
    SF_INFO out_info;

    CMDUI* interface = new CMDUI(argc, argv);
    DSP.buildUserInterface(interface);
    interface->process_command();
```

```

// open input file
in_info.format = 0;
in_sf = sf_open (interface->input_file(), SFM_READ, &in_info);
if (in_sf == NULL) { sf_perror(in_sf); exit(0); }

// open output file
out_info = in_info;
out_info.format = in_info.format;
out_info.channels = DSP.getNumOutputs();
out_sf = sf_open(interface->output_file(), SFM_WRITE, &out_info);
if (out_sf == NULL) { sf_perror(out_sf); exit(0); }

// create separator and interleaver
Separator sep (kFrames, in_info.channels, DSP.getNumInputs());
Interleaver ilv (kFrames, DSP.getNumOutputs(), DSP.getNumOutputs());

// init signal processor
DSP.init(in_info.samplerate);
//DSP.buildUserInterface(interface);
interface->process_init();

// process all samples
int nbf;
do {
    nbf = sf_readf_float(in_sf, sep.input(), kFrames);
    sep.separate();
    DSP.compute(nbf, sep.outputs(), ilv.inputs());
    ilv.interleave();
    sf_writef_float(out_sf, ilv.output(), nbf);
    //sf_write_raw(out_sf, ilv.output(), nbf);
} while (nbf == kFrames);

// close the input and output files
sf_close(in_sf);
sf_close(out_sf);
}

```

Thus, after opening the input and output soundfiles, there is a loop over time frames (sample times).²⁷ For each frame, the interleaved input channels are read from disk by `sf_readf_float()` and deinterleaved into a set of separate buffers by `sep.separate()`. The input buffers are processed by `DSP.compute` to produce output buffers, one for each output signal. The output buffers are then interleaved by `ilv.interleave()` and written to the output soundfile on disk by `sf_writef_float()`. After all time frames have been processed, the input and output soundfiles are closed.

²⁷At each time sampling instant, the set of audio samples from all channels is called a *frame*.

9.2 Soundfile Input for Standalone Faust Applications

JACK standalone apps read and write JACK ports which are easily connected to other JACK-compatible sound sources/sinks via `qjackctl` in Linux or Jack Pilot on a Mac (see §8 and §6 for other examples of using JACK to connect audio streams).

Thus, to feed a soundfile to a standalone FAUST app, run any program that can output a soundfile on a JACK port (such as `pd`), and connect the programs JACK output ports to the FAUST app input ports using the connect/routing GUI interface of `qjackctl` or Jack Pilot.

The following convenience scripts are distributed with FAUST:²⁸

- `faust2jack` - make a JACK-GTK standalone app
- `faust2jaqt` - make a JACK-Qt standalone app
- `faust2jackconsole` - make a JACK console standalone app (no local GUI)

9.3 Soundfile Input for Faust Plugins

Each plugin host has its own soundfile input facilities. For example, in SuperCollider, one often reads an entire soundfile into an instance of the `Buffer` class using the `Buffer.read` method. Similarly, `pd` has a `soundfiler` object for reading a soundfile into a table in memory (see, for example, the `pd` Help Browser at Pure Data / 2.audio.examples / B07.sampler.pd). Finally, typical VST plugin hosts have extensive facilities for reading, writing, and manipulating sound clips, plugin parameters, and so on.

10 Generating a MIDI Synthesizer for PD

The `faust2puredata` script has a `-poly` option for generating a *MIDI synthesizer plugin* for `pd`. The synth has eight voices and manages voice allocation when played from MIDI. For this to work, the FAUST program should be written to synthesize one voice using the following three standard synthesis parameters (which are driven from MIDI data in the `pd` plugin):

- `freq` - frequency of the played note (Hz)
- `gain` - amplitude of the played note (0 to 1)
- `gate` - 1 while “key is down”, 0 after “key up”

The parameters `freq` and `gain` are set according to MIDI note-number and velocity, respectively, while the `gate` parameter is set to 1 on a MIDI “note-on” and back to zero upon “note-off”. The abstraction `midi-in.pd` receives and decodes MIDI data in `pd`.

Let’s make a simple 8-voiced MIDI synthesizer based on the example Faust program `cpgrs.dsp` (“Constant-Peak-Gain Resonator Synth”) listed in Fig. 15 below.

- Copy the text in Fig. 15 and save it in your working directory in a file named `cpgrs.dsp`.
- Creating a `pd synth` is like creating a `pd effect` plugin, except that the `-poly` option is used:

```
> faust2puredata -poly cpgrs.dsp
```

This creates the file `cpgrs~.pd_darwin` in your working directory.

²⁸Additionally there are `faust2jackinternal` and `faust2jackserver` which are beyond the scope of this tutorial.

- If you see the error “m_pd.h file not found”, you may not have Pure Data installed.

In addition to converting the frequency and amplitude parameters to the standard names `freq` and `gain`, we have added a classic ADSR envelope generator²⁹ (defined in FAUST’s `envelopes.lib` file) which uses our new `gate` parameter, and which adds four new *envelope parameters* `attack`, `decay`, `sustain`, and `release`.

To see lower-level details of how the `pd` plugin is created, read the `faust2puredata` shell script, typically installed in `/usr/local/bin/` from `faust/tools/faust2appls/faust2puredata`.

```
import("stdfaust.lib"); // define en.adsr, ma.SR, ma.PI

declare name "Constant-Peak-Gain Resonator Synth";
declare author "Julius Smith";
declare version "1.0";
declare license "GPL";

/* Standard synth controls supported by faust2pd */
freq = nentry("freq", 440, 20, 20000, 1); // Hz
gain = nentry("gain", 0.1, 0, 1, 0.01); // frac
gate = button("gate"); // 0/1

/* User Controls */
bw = hslider("bandwidth (Hz)", 100, 20, 20000, 10);

/* ADSR envelope parameters */
attack = hslider("attack", 0.01, 0, 1, 0.001); // sec
decay = hslider("decay", 0.3, 0, 1, 0.001); // sec
sustain = hslider("sustain", 0.5, 0, 1, 0.01); // frac
release = hslider("release", 0.2, 0, 1, 0.001); // sec

/* Synth */
process = no.noise * env * gain : filter
with {
  env = gate :
    vgroup("1-adsr",
      en.adsr(attack, decay, sustain, release));
  filter = vgroup("2-filter", (firpart : + ~ feedback));
  R = exp(0 - ma.PI * bw / ma.SR); // pole radius
  A = 2 * ma.PI * freq / ma.SR; // pole angle (radians)
  RR = R*R;
  firpart(x) = (x - x'') * (1-RR)/2;
  // time-domain coefficients ASSUMING ONE-SAMPLE FEEDBACK DELAY:
  feedback(v) = 0 + 2*R*cos(A)*v - RR*v';
};
```

Figure 15: Listing of `cpgrs.dsp`—a FAUST program specifying a simple synth patch consisting of white noise through a constant-peak-gain resonator.

²⁹The Attack, Decay, Sustain, and Release (ADSR) envelope is said to have been invented by Robert Moog in collaboration with composer Herbert Deutsch in the 1960s.

11 MIDI Synthesizer Test Patch

The example synth is loaded into `pd` like any plugin-wrapper. A manually written test patch (`cpgrshelp.pd`) is shown in Fig. 16. Note that the standard MIDI-synth control parameters (`freq`, `gain`, `gate`) are handled behind the scenes and do not appear among the plugin GUI controls.

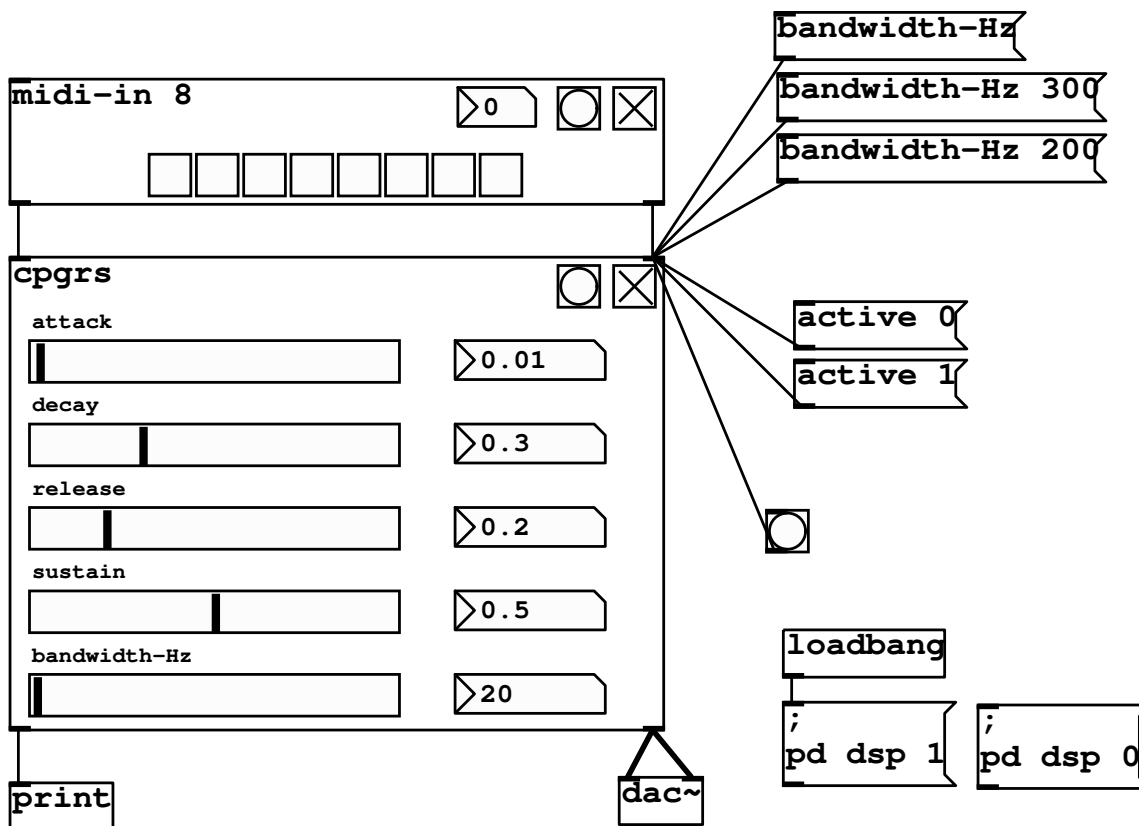


Figure 16: Test patch for the `pd` synth plugin `cpgrs.pd` generated by `faust2pd` based on `cpgrs.dsp` in Fig. 15.

To drive our MIDI synth, we need a source of MIDI data. Perhaps the simplest resource for this purpose is the Virtual Keyboard (`vkeybd`), which is standard in Red Hat Fedora, and in the `planetccrma-menus` at “Applications / Planet CCRMA / MIDI / Vkeybd”). Figure 17 shows a screen shot of the Virtual Keyboard with its key-range and velocity controllers displayed (menu item “View / Key/Velocity”). The velocity controller sets the `gain` parameter, mapping MIDI velocity (0-127) to the unit interval (0-1). The key-range controller transposes the keyboard by octaves. Pressing a key determines, together with the key-range, the `freq` parameter in our synth. Pressing a key also sets the `gate` parameter to 1, and releasing it sets `gate` to 0. The ADSR envelope is triggered when `gate` transitions to 1, and it begins its “release” phase when `gate` transitions to 0, as is standard for ADSR envelopes triggered by a keyboard. Note that the bottom two rows of ASCII keyboard keys are mapped to virtual-keyboard keys, enabling the playing of chords in real time on the regular computer keyboard.

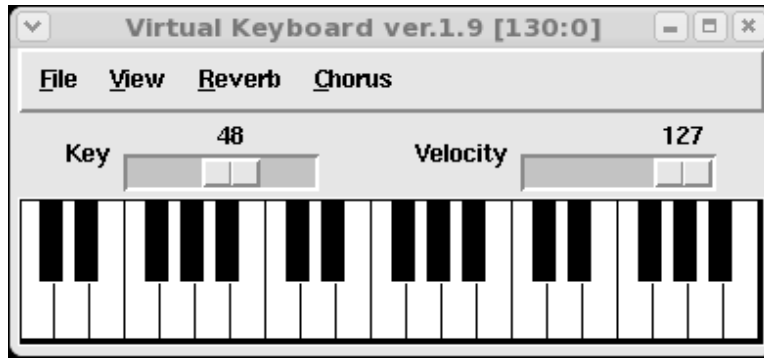


Figure 17: The Virtual Keyboard (MIDI source).

Figure 18 illustrates the MIDI tab of `qjackctl`'s Connect window after connecting the Virtual Keyboard MIDI output to `pd`'s MIDI input.³⁰

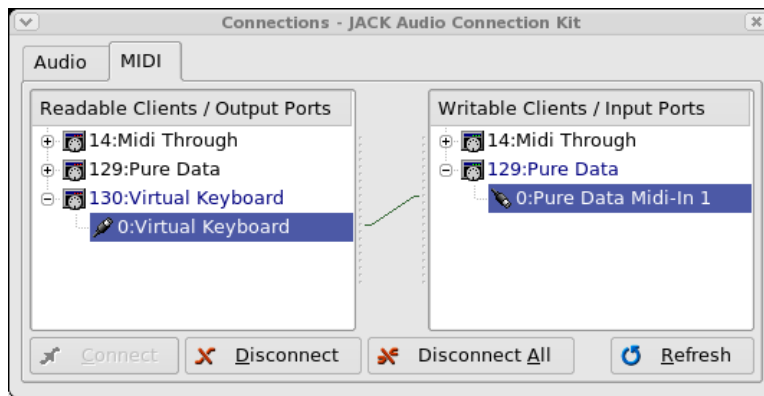


Figure 18: JACK MIDI connections routing MIDI from the Virtual Keyboard (`vkeybd`) to `pd`'s first MIDI input port.

To play back a MIDI file (extension `.mid`), a nice way is to open it in Rosegarden (“Applications / Planet CCRMA / Sequencers / Rosegarden”) and connect Rosegarden’s MIDI output to `pd`’s MIDI input as above. (You can still play along on the Virtual Keyboard.)

12 Using Faust with SuperCollider

This section describes and illustrates making SuperCollider (SC) plugins from FAUST source on Linux and Mac OS X. (Windows should also work.)

³⁰Pd must have at least one MIDI-input port defined at startup for this to work. For example, a typical `~/pdrc` file might contain the following startup options for `pd`:

```
-jack -r 48000 -alsamidi -midiindev 1 -midioutdev 1 -audiooutdev 1 -outchannels 2 -path
/usr/lib/pd/...
```


12.1 Getting Started with SuperCollider

The reader is assumed to have worked through a first tutorial on SuperCollider (SC). In particular, the “Getting Started with SC” tutorial in the SuperCollider Documentation³¹ is a good place to begin. The SC installation page is <https://supercollider.github.io/download>. After a basic orientation via tutorials, the online documentation is excellent. It can be effective to work through various tutorial examples, placing the cursor (in Emacs, *e.g.*) on a class name, and typing C-c C-h to jump to the class documentation (Cmd-D in the Mac app), which in turn refers to other classes and online tutorials, and so on. (Note that `sclang` must be running for this to work.) For more on using FAUST with SuperCollider, see [8]. See the “SuperCollider Book” [12] for introductions to and fuller presentation of many aspects of SC. When you are ready for it, read through relevant SC source code (*.sc). Note that “C-c :” in Emacs will go to the class definition file when the editing cursor is on the name of the class. Also, “C-c ;” will look up references to methods marked by the cursor. Reading .sc code will rapidly get you comfortable with SC as a general-purpose object-oriented programming language.

12.2 Linux and Faust-Generated SuperCollider Plugins

The examples below were last fully tested on a Fedora 15 64-bit Linux system running FAUST version 0.9.43 (on 8/4/2011), and later updated based on email feedback in May of 2020. If anything else needs updating, please let me know via email.

The shell script `faust2supercollider` can be used to create a SuperCollider (SC) plugin from a FAUST source file:

```
> faust2supercollider mysynth.dsp
```

The resulting class file `mysynth.sc` (the `sclang` “extension”) and shared-object file `mysynth.so` (the `scsynth` or `supernova` “plugin”) can be copied from the current working directory to your SC extensions directory, typically as follows on Linux systems:

```
> mkdir -p ~/share/SuperCollider/Extensions/Faust/
> cp mysynth.s[co] ~/share/SuperCollider/Extensions/Faust/.
```

Below we speak of using `scsynth` as the SuperCollider sound-synthesis server, but `supernova`, the more recent multithreaded generalization of `scsynth`, should also work in its place.

To try out the FAUST example `osc.dsp`, for example, first copy the generated files into your SC extensions directory:

```
cp <faust>/examples/supercolliderdir/osc.s[co] ~/share/SuperCollider/Extensions/Faust
```

Next, restart `sclang` and `scsynth`,³² and execute the following code in `sclang`:

```
y = {
  var out;
  out = {FaustOsc.ar(freq:100.0,volume:-40.0)}.dup;
}.play(s)
```

³¹<http://doc.sccode.org>

³²After restarting `sclang`, the class name `FaustOsc` will be defined. In emacs, this is indicated by typesetting it in the special color for known class names. When `scsynth` is started, you can obtain a printout of the line “Faust: osc numControls=2” in the post buffer when the module is loaded, if you remove `-DNDEBUG` from the compiler flags in either `faust2supercollider` or `Makefile.sccompile`.

You should hear a soft sinusoid at frequency 100 Hz. (Execute “y.free;” when you are tired of listening to it, which will likely be quite soon.)

The FaustOsc class in the file `osc.sc` created by the make is as follows:

```
FaustOsc : UGen
{
  *ar { | freq(1000.0), volume(0.0) |
    ^this.multiNew('audio', freq, volume)
  }
  *kr { | freq(1000.0), volume(0.0) |
    ^this.multiNew('control', freq, volume)
  }
  name { ^"osc" }
}
```

We see for example that the default frequency and volume are 1 kHz and 0 dB, respectively. Adapting a cool example from the SuperCollider Book, we can control the amplitude and frequency of the oscillator with the mouse as follows:

```
z = {
  var out;
  out = (FaustOsc.ar(
    mouseX.kr(400,3000,\exponential), // freq (Hz)
    mouseY.kr(-90,10,\linear) // amp (dB)
  ) * MouseButton.kr).dup; // gate = LEFT mouse button
}.play(s);

z.free;
```

12.3 Mac OS X and Faust-Generated SuperCollider Plugins

On the Mac, SuperCollider (SC) extensions go in the directory

```
~/Library/Application Support/SuperCollider/Extensions
```

instead of `~/share/SuperCollider/Extensions` as on Linux. Again it is nice to organize all Faust-generated plugins in a `Faust` subdirectory of `Extensions`.

For a nice collection of example SC plugins, see <https://github.com/supercollider/example-plugins>

Since pre-built versions of SC for the Mac do not seem to include the headers for compiling plugins, you probably need also to download the SC source and set the `SUPERCOLLIDER_HEADERS` environment variable to point into it. For example, at the time of this writing, the latest stable SC source is v3.4.4, so I have

```
setenv SUPERCOLLIDER_HEADERS $HOME/sc-3.4.4/common/Headers
```

in my `.tcshrc` file. For Bourne shell (`bash`) users, add the lines

```
SUPERCOLLIDER_HEADERS=$HOME/sc-3.4.4/common/Headers
export SUPERCOLLIDER_HEADERS
```

in `~/ .bashrc`, etc.

To create a 32-bit plugin instead of the default 64-bit case, copy the `faust2supercollider` script and add “-m32” at the end of the `SCFLAGS` variable to get

```
SCFLAGS="-DNO_LIBSNDFILE -DSC_DARWIN -bundle -m32"
```

One difference on the Mac relative to Linux is that the mouse-controlled example of the previous section has its vertical axis flipped. That is, on the Mac, the volume gets louder as the mouse goes down on the screen.

If either of the above `FaustOsc` examples does not work on either Linux or the Mac, try checking out the latest `git` FAUST distribution, as described in §1.1. This document is updated to stay in sync with that (latest) version, as opposed to any particular prior FAUST release.

For some SuperCollider doc on making FAUST UGens, see <https://doc.sccode.org/Guides/WritingUGens.h>

13 Using Faust with Open Sound Control (OSC)

FAUST contains some very nice Open Sound Control (OSC) support facilities [2, 9]. OSC, which can be viewed as a generalized replacement for MIDI, is often used for messaging between music applications, even across networks. It is used, for example, as the communication protocol between the SuperCollider client (`sclang`) and server (`scsynth`). It is also the protocol of choice for messages from external controllers, such as iOS/Android applications running on tablets.

OSC uses the User Datagram Protocol (UDP) to send and receive messages. UDP is like TCP/IP (the more typical internet message protocol) except that message delivery is not guaranteed. In other words, if some process gets behind and a message is dropped, nobody worries about it. FAUST OSC support makes use of three UDP ports for input, output, and errors. By default these are the UDP ports numbered 5510, 5511, and 5512, respectively; when necessary, FAUST will try higher numbers until a free UDP port is found.

To send an OSC message to some process on some host on the Internet, one needs to know the host’s IP address, the UDP port used by the process for receiving, and a “name path” (much like a UNIX file path) to the control being affected. The path starts with the application name and includes all group names down to the control name itself. For example, in the `zita_rev1.dsp` example below, the path to the “dry-wet mix” slider is `/Zita_Rev1/Output/Dry_Wet_Mix`, where `Zita_Rev1` is the name of the application, `Output` is a control group defined in the FAUST source, and `Dry_Wet_Mix` is the name of the control slider itself within the `Output` group.

We will use three shells for the examples below, each in its own Terminal window. In Window 1, we will run a FAUST standalone JACK application that will receive OSC messages on UDP port 5510 and respond on port 5511. In Window 2, we will run `oscdump` (distributed with the `liblo` package) to print out OSC message activity on port 5511. In Window 3, we will run `oscsend` (also from `liblo`) to send OSC messages to port 5510, where the FAUST JACK app is listening. Any responses from the app will appear in Window 2.

Since we are using a standalone JACK application for these examples, remember to start the JACK daemon `jackd` (via `qjackctl` on Linux or `JackPilot` on the Mac). If you forget, `jackd` will be autolaunched with default setup parameters from your `~/ .jackdrc` configuration file.

Below, comments and program printout are on lines beginning with ‘#’.

In Terminal Window 1 (application window):

```
> cd <faust>/examples/generator/
> faust2jack -osc noise.dsp
# Drill down on /usr/local/bin/faust2jack and its output to see all the code "under the hood".
> ./noise
# After perhaps some delay (especially if JACK is autolaunched), you should see
# ...
# Faust OSC version 0.91 application 'noise' is running on UDP ports 5510, 5511, 5512
# ...
```

The UDP port numbers are for input, output, and error messages, respectively.

In Terminal Window 2 (OSC dump window):

```
> oscdump 5511
# watch this space for OSC message replies from the Faust JACK app
```

In Terminal Window 3 (OSC send window):

```
> oscsend localhost 5510 /\* s "hello"
# Window 2 receives the following, after some seconds of delay:
# /noise siii "0.0.0.0" 5510 5511 5512
```

Notice the use of 's' to indicate that a string follows. Note that `*` is passed to `oscsend` as `*`. We are just quoting it to avoid shell filename expansion here. On Red Hat Fedora 17 Linux (liblo v0.26), say 0 in place of `localhost` above. On a remote machine, use the server-machine's IP address in place of `localhost`.

```
> oscsend localhost 5510 /\* s "get"
# Window 2 receives the following (immediately):
# /noise/Volume fff 0.000000 0.000000 1.000000
```

This printout says there is one parameter whose OSC address path is `/noise/Volume` (case matters) and its current value is 0, minimum value is 0, and maximum value is 1. Let's set it to 0.1:

```
> oscsend localhost 5510 /noise/Volume f 0.1
# Nothing is echoed, so let's ask for the current value:

> oscsend localhost 5510 /noise/Volume s "get"
# /noise/Volume fff 0.100000 0.000000 1.000000
^C # Stop the noise app so that the next example can use port 5510.
```

It worked! Notice the use of 'f' to indicate that a floating-point value follows. If we didn't type control-C (`^C`) to end the program, the next example would listen for OSC messages on UDP port 5513. This is fine, and both programs would work in parallel (both being connected to system output in JACK), but we will keep to one example at a time here.

Now let's make OSC JACK apps out of some more FAUST examples and exercise them:

In Window 1:

```
> cd <faust>/examples/dynamic/
> # Mac:
```

```

> faust2jaqt -osc distortion.dsp
> open ./distortion.app
> # Linux:
> faust2jack -osc distortion.dsp
> ./distortion

```

If Window 2 is still set up running `oscdump` as above, we see

```
Faust s " OSC version 1.22 - 'CUBIC_NONLINEARITY_cubicnl' is running on UDP ports 5513, 5511, 5512,
```

In Window 3:

```

> oscsend localhost 5510 /\* s "hello"
# /distortion siii "0.0.0.0" 5510 5511 5512

> oscsend localhost 5510 /\* s "get"
# /distortion//SINE_WAVE_OSCILLATOR_oscrs//Amplitude fff -38.299999 -120.000000 10.000000
# /distortion//SINE_WAVE_OSCILLATOR_oscrs//Frequency fff 37.599998 1.000000 88.000000
# /distortion//SINE_WAVE_OSCILLATOR_oscrs//Portamento fff 0.846000 0.000000 1.000000
# /distortion//CUBIC_NONLINEARITY_cubicnl//Bypass fff 0.000000 0.000000 1.000000
# /distortion//CUBIC_NONLINEARITY_cubicnl//Drive fff 0.630000 0.000000 1.000000
# /distortion//CUBIC_NONLINEARITY_cubicnl//Offset fff 0.000000 0.000000 1.000000
# /distortion//SPECTRUM_ANALYZER_CONTROLS/Level_Averaging_Time fff 0.510000 0.000000 1.000000
# /distortion//SPECTRUM_ANALYZER_CONTROLS/Level_dB_Offset fff 18.000000 0.000000 100.000000

> oscsend localhost 5510 /distortion//CUBIC_NONLINEARITY_cubicnl//Drive f 0.9

> oscsend localhost 5510 /distortion//CUBIC_NONLINEARITY_cubicnl//Drive s "get"
# /distortion//CUBIC_NONLINEARITY_cubicnl//Drive fff 0.900000 0.000000 1.000000

```

and so on. Notice how the GUI grouping gets into the control name path. The appearance of `//` in the path indicates an unnamed group, which causes no problem. An attempt to set a parameter out of range (less than 0 or greater than 1 in this case) results in the parameter being clipped to the limit.

Here's another example (you now know which window is being listed):

```

^C
> cd <faust>/examples/reverb/
> # Mac:
> faust2jaqt -osc zitaRev.dsp
> open ./zitaRev.app
> # Linux:
> faust2jack -osc zitaRev.dsp
> ./zitaRev
# Faust OSC version sssiii "0.91" "-" "'Zita_Rev1'" "is running on UDP ports " 5510 5511 5512

> oscsend localhost 5510 /\* s "get"
# /Zita_Rev1/Input/In_Delay fff 24.400000 20.000000 100.000000
# /Zita_Rev1/Decay_Times_in_Bands_(see_tooltips)/LF_X fff 1000.000000 50.000000 1000.000000
# /Zita_Rev1/Decay_Times_in_Bands_(see_tooltips)/Low_RT60 fff 1.875000 1.000000 8.000000
# /Zita_Rev1/Decay_Times_in_Bands_(see_tooltips)/Mid_RT60 fff 1.455000 1.000000 8.000000

```

```
# /Zita_Rev1/Decay_Times_in_Bands_(see_tooltips)/HF_Damping fff 10418.099609 1500.000000 23520.000000
# /Zita_Rev1/RM_Peaking_Equalizer_1/Eq1_Freq fff 1108.900024 40.000000 2500.000000
# /Zita_Rev1/RM_Peaking_Equalizer_1/Eq1_Level fff 11.000000 -15.000000 15.000000
# /Zita_Rev1/RM_Peaking_Equalizer_2/Eq2_Freq fff 2474.000000 40.000000 2500.000000
# /Zita_Rev1/RM_Peaking_Equalizer_2/Eq2_Level fff -0.200000 -15.000000 15.000000
# /Zita_Rev1/Output/Dry_Wet_Mix fff 1.000000 -1.000000 1.000000
# /Zita_Rev1/Output/Level fff 14.150000 -70.000000 40.000000
```

Other examples are similar.

Finally, let's make an OSC JACK app out of the Faust-STK piano and try it out:

```
> cd <faust>/examples/physicalModeling/faust-stk/
> # Mac:
> faust2jaqt -osc piano.dsp
> open ./piano.app
> # Linux:
> faust2jack -osc piano.dsp
> ./piano
# Faust OSC version ssssi "0.91" "-" "'piano'" "is running on UDP ports " 5510 5511 5512

> oscsend localhost 5510 /\* s "get"
# /piano/Basic_Parameters/freq fff 440.000000 20.000000 20000.000000
# /piano/Basic_Parameters/gain fff 1.000000 0.000000 1.000000
# /piano/Basic_Parameters/gate fff 0.000000 0.000000 1.000000
# /piano/Physical_Parameters/Brightness_Factor fff 0.000000 0.000000 1.000000
# /piano/Physical_Parameters/Detuning_Factor fff 0.100000 0.000000 1.000000
# /piano/Physical_Parameters/Hammer_Hardness fff 0.100000 0.000000 1.000000
# /piano/Physical_Parameters/Stiffness_Factor fff 0.280000 0.000000 1.000000
# /piano/Reverb/reverbGain fff 0.137000 0.000000 1.000000
# /piano/Reverb/roomSize fff 0.720000 0.010000 2.000000
# /piano/Spat/pan_angle fff 0.600000 0.000000 1.000000
# /piano/Spat/spatial_width fff 0.500000 0.000000 1.000000
```

As with all synth examples, a note is played on the piano patch by setting parameters as desired (such as freq) and then setting the **gate** parameter to 1 to start the note. Setting **gate** back to 0 starts the decay phase of the note. In patches with ADSR or ASR envelopes, **gate** transitioning to 1 starts the Attack phase, while a transition to 0 starts the Release phase, as is typical in synthesizers driven by a keyboard. The bowed instrument is an example using an ADSR envelope:

```
^C
> # (Make bowed like piano above)
> ./bowed
# Faust OSC version 0.91 application 'bowed' is running on UDP ports 5510, 5511, 5512

> oscsend localhost 5510 /\* s "get"
# /bowed/Basic_Parameters/freq fff 440.000000 20.000000 20000.000000
# /bowed/Basic_Parameters/gain fff 1.000000 0.000000 1.000000
# /bowed/Basic_Parameters/gate fff 0.000000 0.000000 1.000000
# /bowed/Envelopes_and_Vibrato/Envelope_Parameters/Envelope_Attack fff 0.010000 0.000000 2.000000
# /bowed/Envelopes_and_Vibrato/Envelope_Parameters/Envelope_Decay fff 0.050000 0.000000 2.000000
# /bowed/Envelopes_and_Vibrato/Envelope_Parameters/Envelope_Release fff 0.100000 0.000000 2.000000
# /bowed/Envelopes_and_Vibrato/Vibrato_Parameters/Vibrato_Attack fff 0.500000 0.000000 2.000000
# /bowed/Envelopes_and_Vibrato/Vibrato_Parameters/Vibrato_Begin fff 0.050000 0.000000 2.000000
```

```

# /bowed/Envelopes_and_Vibrato/Vibrato_Parameters/Vibrato_Freq fff 6.000000 1.000000 15.000000
# /bowed/Envelopes_and_Vibrato/Vibrato_Parameters/Vibrato_Gain fff 0.010000 0.000000 1.000000
# /bowed/Envelopes_and_Vibrato/Vibrato_Parameters/Vibrato_Release fff 0.010000 0.000000 2.000000
# /bowed/Physical_and_Nonlinearity/Nonlinear_Filter_Parameters/Modulation_Frequency fff 220.000000
# /bowed/Physical_and_Nonlinearity/Nonlinear_Filter_Parameters/Modulation_Type fff 0.000000 0.000000
# /bowed/Physical_and_Nonlinearity/Nonlinear_Filter_Parameters/Nonlinearity fff 0.000000 0.000000
# /bowed/Physical_and_Nonlinearity/Nonlinear_Filter_Parameters/Nonlinearity_Attack fff 0.100000 0
# /bowed/Physical_and_Nonlinearity/Physical_Parameters/Bow_Position fff 0.700000 0.010000 1.000000
# /bowed/Physical_and_Nonlinearity/Physical_Parameters/Bow_Pressure fff 0.750000 0.000000 1.000000
# /bowed/Reverb/reverbGain fff 0.137000 0.000000 1.000000
# /bowed/Reverb/roomSize fff 0.720000 0.010000 2.000000
# /bowed/Spat/pan_angle fff 0.600000 0.000000 1.000000
# /bowed/Spat/spatial_width fff 0.500000 0.000000 1.000000

```

There are many more examples and Faust-STK examples to look into. This is just a start.

OSC Aliases

See the FAUST OSC documentation (Section 6.8) in the FAUST Quick Reference,³³ for more advanced techniques such as the use of *OSC aliases* which allow arbitrary OSC messages³⁴ to be mapped to controller parameters (for use with remote controllers such as TouchOSC on Android that can only transmit predefined messages).

The FAUST OSC documentation provides additional information.

14 Conclusions

In summary, writing signal processing applications in the FAUST language is rewarding in several respects. First, the language is high-level, yet it compiles to efficient C++ code. The development cycle is very short, typically involving only (1) syntax debugging, (2) block-diagram inspection, and (3) trying out the app/plugin. When precise verification is required, the output signals can be printed or loaded into Matlab/Octave for detailed analysis. Finally, writing in FAUST makes it very easy to generate applications and plugins for a wide range of hosting environments.

³³<https://github.com/grame-cncm/faust/blob/master-dev/documentation/faust-quick-reference.pdf>

³⁴Example: `vslider("Volume[osc:/1/fader1]", 0, 0, 1, 0.1)` will map the OSC path `/1/fader1` to this Volume slider.

A Appendix A: State-Space Models to Faust

A digital state-space model is normally written mathematically as

$$\begin{aligned}\underline{y}(n) &= \mathbf{C}\underline{x}(n) + \mathbf{D}\underline{u}(n) \\ \underline{x}(n+1) &= \mathbf{A}\underline{x}(n) + \mathbf{B}\underline{u}(n)\end{aligned}$$

where

- $\underline{x}(n) \in \mathbb{R}^N$ = *state vector* at time n
- $\underline{u}(n) = p \times 1$ vector of inputs
- $\underline{y}(n) = q \times 1$ output vector
- $\mathbf{A} = N \times N$ *state transition matrix*
- $\mathbf{B} = N \times p$ *input coefficient matrix*
- $\mathbf{C} = q \times N$ *output coefficient matrix*
- $\mathbf{D} = q \times p$ *direct path coefficient matrix*

The matrices \mathbf{B} , \mathbf{C} and \mathbf{D} make up the *feedforward* part, while \mathbf{A} is the *feedback* part.

Note that all four matrices may change each sample instant n to implement a general *time-varying* linear system.

A FAUST example for the general linear state-space model is shown in Fig. 19. The FAUST-generated block diagram is shown in Fig. 20.

Since a state-space model can implement any N th-order linear system with p inputs and q outputs, all we have to do is come up with the $(\mathbf{A}, \mathbf{B}, \mathbf{C}, \mathbf{D})$ matrix entries and use the general state-space formulation in FAUST given in Fig. 19.

Here are the steps for finding the state-space matrices:

1. Label each delay element as a state variable x_i . Delay lines can be treated as one “generalized delay” (Feedback Delay Networks are often written this way). In terms of the mathematical description above, the i th delay input is $\underline{x}_i(n+1)$, and the output is $\underline{x}_i(n)$.
2. By tracing connections in the block diagram, write $\underline{x}_i(n+1)$ as a linear combination of either inputs $\underline{u}(n)$ or states $\underline{x}(n)$ for each $i = 1, 2, \dots, N$. Place these coefficients where they go in the i th row of \mathbf{B} (for inputs) and \mathbf{A} (for states), and enter zeros for inputs and states not needed to create $\underline{x}_i(n+1)$.
3. Now do the same for each output $\underline{y}_i(n)$, $i = 1, \dots, q$, which is similarly a linear combination of inputs and/or states. Use these coefficients to populate the i th row of \mathbf{D} (for inputs) and \mathbf{C} (for states).
4. Now you’ve filled your $(\mathbf{A}, \mathbf{B}, \mathbf{C}, \mathbf{D})$ matrices, so they can be plugged into FAUST code based on Fig. 19.

A simple example for trying out and checking this process can be found by starting at step 3 of “Converting to State-Space Form by Hand” [11].³⁵ The details appear below in §A.1. Also be aware of the matlab function `tf2ss` for converting a transfer function to state-space form.

³⁵https://ccrma.stanford.edu/~jos/StateSpace/Converting_State_Space_Form_Hand.html


```

// General Linear State-Space Model Example

import("stdfaust.lib");

p = 2; // number of inputs
q = 3; // number of outputs
N = 5; // number of states

A = matrix(N,N); // state transition matrix
B = matrix(N,p); // input-to-states matrix
C = matrix(q,N); // states-to-output matrix
D = matrix(q,p); // direct-term matrix, bypassing state

matrix(M,N) = tgroup("Matrix: %M x %N", par(in, N, _)
    <: par(out, M, mixer(N, out))) with {
    fader(in) = ba.db2linear(vslider("Input %in", -10, -96, 4, 0.1));
    mixer(N,out) = hgroup("Output %out", par(in, N, *(fader(in)) ) >: _ );
};

Bd = par(i,p,mem) : B; // input delay needed for conventional definition
vsum(N) = si.bus(2*N) >: si.bus(N); // vector sum of two N-vectors
impulse = 1-1'; // For zero initial state, set impulse = 0 or simplify code
x0 = par(i,N,i*impulse); // initial state = (0,1,2,3,...,N-1)

system = si.bus(p) <: D, (Bd : vsum(N)^(A), x0 : vsum(N) : C) >: si.bus(q);

process = system;

```

Figure 19: FAUST example illustrating a general state-space model having 2 inputs, 3 outputs, and 5 state variables.

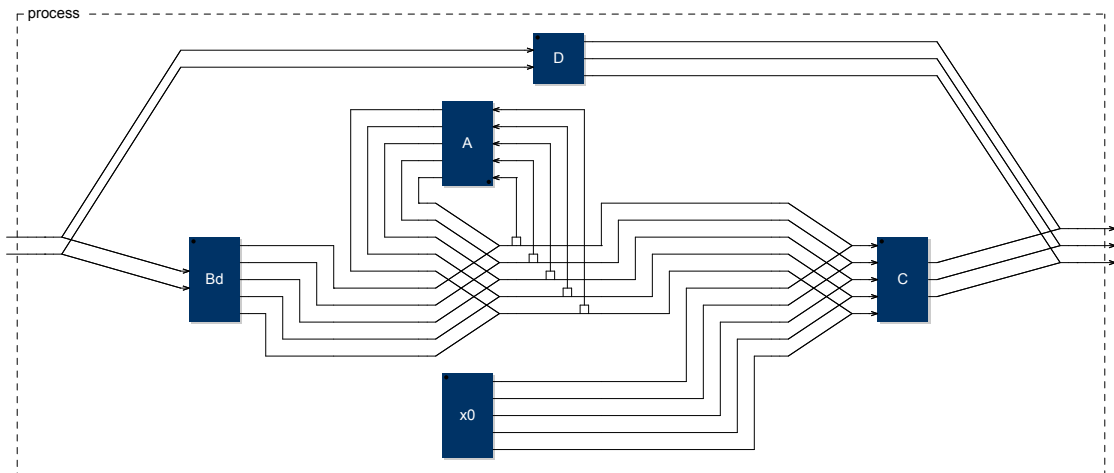


Figure 20: State-space model in FAUST having 2 inputs, 3 outputs, and 5 state variables.

A.1 State-Space BiQuad in Faust

The block diagram of a general BiQuad filter section realized in Direct Form II is shown in Fig. 21 [10].³⁶

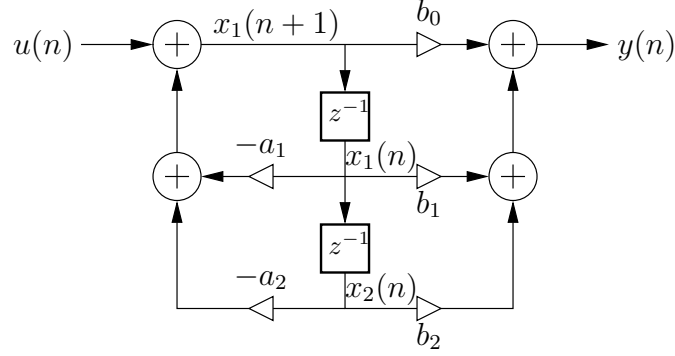


Figure 21: Direct-Form-II implementation of a 2nd-order filter section, with state variables labeled as the delay outputs $x_1(n)$ and $x_2(n)$.

From this diagram, we find the state-space matrices to be

$$A = \begin{bmatrix} -a_1 & -a_2 \\ 1 & 0 \end{bmatrix} \quad B = \begin{bmatrix} 1 \\ 0 \end{bmatrix} \quad C = [b_1 - b_0 a_1 \quad b_2 - b_0 a_2] \quad D = [b_0]$$

The FAUST code for the state-space realization of a specific resonator is shown in Fig. 22.

³⁶https://ccrma.stanford.edu/~jos/filters/Direct_Form_II.html

```

// State-Space BiQuad Example

import("stdfaust.lib");

process = tpss; // state-space form
//process = tpdirect; // direct form
//Test to compare outputs of the two:
//process = 1-1' <: tpdirect, tpss * (-1) :> _; // ~0

// Make direct-form coefficients for a simple resonator:
R = 0.9; // pole radius
fc = ma.SR/10.0; // pole angle frequency in Hz
wcT = 2.0 * ma.PI * fc / ma.SR;
a1 = -2*R*cos(wcT); a2 = R*R;
b0 = 1.0; b1 = 0.0; b2 = -1.0; // zeros at dc and SR/2

tpdirect = fi.tf2(b0,b1,b2,a1,a2); // filters.lib implementation

// State Space Model for Direct Form II:

p = 1; // number of inputs
q = 1; // number of outputs
N = 2; // number of states

a(1,1) = -a1; a(1,2) = -a2;
a(2,1) = 1; a(2,2) = 0;

b(1,1) = 1;
b(2,1) = 0;

c(1,1) = b1-b0*a1; c(1,2) = b2-b0*a2;

d(1,1)= b0;

// We presently also need these catch-all rules (which are not used):
a(m,n) = 10*m+n; b(m,n) = a(m,n);
c(m,n) = a(m,n); d(m,n) = a(m,n);

matrix(M,N,f) = si.bus(N) <: ro.interleave(N,M)
: par(n,N, par(m,M,(f(m+1,n+1)))) :> si.bus(M);

A = matrix(N,N,a); B = matrix(N,p,b);
C = matrix(q,N,c); D = matrix(q,p,d);

Bd = par(i,p,mem) : B; // input delay needed for conventional definition
vsum(N) = si.bus(2*N) :> si.bus(N); // vector sum of two N-vectors

tpss = si.bus(p) <: D, (Bd : vsum(N)~(A) : C) :> si.bus(q);

```

Figure 22: FAUST state-space BiQuad filter, comparing to direct form.

B Appendix B: Inspecting Assembly to Fine-Tune Performance

On the FAUST mailing list, Dario Sanfilippo suggested that we change the default one-pole smoother `fi.smoo` in `filters.lib` from the form $(1-b)*x(n)+b*y(n-1)$ to instead $x(n)+b*[y(n-1)-x(n)]$, recognizing that multiplications are more expensive in hardware than additions. This appendix was written from my email reply to the list:

This is a winner. I am strongly in favor of the change. One multiply and two additions is fundamentally less work than two multiplies and one addition. However, when two multiplies are available in parallel, then $(1-b) * x + b * y$ can be faster than $x + b * (y-x)$ because it takes two steps instead of three. Thus, a parallel-processing implementation might prefer the first form.

Ideally both forms would compile to the same assembly, but this is not yet the case. Neither the FAUST compiler nor the C++ compiler appear to work to minimize multiplies relative to additions when the target architecture warrants that.

Of course we should run benchmarks to measure the actual improvement on each architecture,³⁷ but looking at assembly can also give the answer. I recently learned about the Compiler Explorer at godbolt.org, for comparing assemblies on various processors, and this was my first use of it:

First, the FAUST source, adapted from mailing-list thread, is shown in Fig. 23.

```
import("stdfaust.lib");
smooth(coeff, x) = fb ~ _ with { fb(y) = y + (1.0 - coeff) * (x - y); };
c = 1.0 - 44.1 / ma.SR
smooth3(s, x) = fb ~ _ with { fb(y) = s * (y - x) + x; };
process = _ <: si.smooth(c), smooth(c), smooth3(c);
```

Figure 23: FAUST program specifying three different one-pole smoothers in parallel.

Next, I compiled the FAUST source at the command line with no options, and copy/pasted the `compute()` function to create the stand-alone code snippet shown in Fig. 24. (Note that it’s no longer a virtual function.)

This code can be pasted into the left panel of the Compiler Explorer at godbolt.org. Next choose your processor architecture and compiler on the right panel, and your C++ compiler options. Here I chose the first Intel case (more readable than ARM): `x86-64 clang (assertions trunk)`, with compiler options `-std=c++17 -O3`. The term “trunk” refers to the latest version of the compiler source, but you can also try earlier versions of the compiler listed next. WebAssembly, alternative compilers, and many embedded processors appear as choices. My choice is ideal for my iMac Pro where I do most of my work, but I need to check ARM also for my iOS work. The Compiler Explorer is a great tool for fine-tuning performance at the lowest level.

Figure 25 shows the assembly output with added comments indicating where I guessed things came from. You can see that the specified computation structure is preserved all the way down to the bottom, even with `-O3` optimization. The clear winner is `smooth3`, and benchmarks should verify that. It has only one multiply and two additions, and only six instructions (lines of assembly code) total.

Stéphane Letz commented in the discussion thread that in his tests using `faustbench-llvm` on an Apple M1, `process = par(i, 10, si.smoo);` was faster, while the following ran a bit slower:

³⁷FAUST benchmarking tools are documented at <https://github.com/grame-cncm/faust/tree/master-dev/tools/benchmark>

```

#define FAUSTFLOAT float

int fSampleRate = 44100;
float fConst0 = 0.1; // linear-interpolation constant
float fConst1 = 0.9; // 1-fConst0
float fRec0[2];
float fRec1[2];
float fRec2[2];

void compute(int count, FAUSTFLOAT** inputs, FAUSTFLOAT** outputs) {
    FAUSTFLOAT* input0 = inputs[0];
    FAUSTFLOAT* output0 = outputs[0];
    FAUSTFLOAT* output1 = outputs[1];
    FAUSTFLOAT* output2 = outputs[2];
    for (int i0 = 0; (i0 < count); i0 = (i0 + 1)) {
        float fTemp0 = float(input0[i0]);
        fRec0[0] = ((fConst1 * fRec0[1]) + (fConst0 * fTemp0));
        output0[i0] = FAUSTFLOAT(fRec0[0]);
        fRec1[0] = (fRec1[1] + (fConst0 * (fTemp0 - fRec1[1])));
        output1[i0] = FAUSTFLOAT(fRec1[0]);
        fRec2[0] = (fTemp0 + (fConst1 * (fRec2[1] - fTemp0)));
        output2[i0] = FAUSTFLOAT(fRec2[0]);
        fRec0[1] = fRec0[0];
        fRec1[1] = fRec1[0];
        fRec2[1] = fRec2[0];
    }
}

```

Figure 24: C++ program adapted from the output of simplest FAUST compilation at the command line.

```

compute(int, float**, float**): # @compute(int, float**, float**)
...
.LBB0_2: # %for.body

# output0[i0] = ((fConst1 * fRec0[1]) + (fConst0 * input0[i0])), 7 lines:
movss xmm1, dword ptr [r8 + 4*rax] # xmm1 = mem[0],zero,zero,zero
mulss xmm0, dword ptr [rip + fConst1]
movss xmm2, dword ptr [rip + fConst0] # xmm2 = mem[0],zero,zero,zero
mulss xmm2, xmm1
addss xmm2, xmm0
movss dword ptr [rip + fRec0], xmm2
movss dword ptr [rcx + 4*rax], xmm2

# output1[i0] = (fRec1[1] + (fConst0 * (input0[i0] - fRec1[1]))), 7 lines:
movss xmm0, dword ptr [rip + fRec1+4] # xmm0 = mem[0],zero,zero,zero
movaps xmm2, xmm1
subss xmm2, xmm0
mulss xmm2, dword ptr [rip + fConst0]
addss xmm2, xmm0
movss dword ptr [rip + fRec1], xmm2
movss dword ptr [rsi + 4*rax], xmm2

# output2[i0] = (input0[i0] + (fConst1 * (fRec2[1] - input0[i0]))), 6 lines:
movss xmm0, dword ptr [rip + fRec2+4] # xmm0 = mem[0],zero,zero,zero
subss xmm0, xmm1
mulss xmm0, dword ptr [rip + fConst1]
addss xmm0, xmm1
movss dword ptr [rip + fRec2], xmm0
movss dword ptr [rdx + 4*rax], xmm0

# fRec0[1] = fRec0[0];
movss xmm0, dword ptr [rip + fRec0] # xmm0 = mem[0],zero,zero,zero
movss dword ptr [rip + fRec0+4], xmm0

# fRec1[1] = fRec1[0];
movss xmm1, dword ptr [rip + fRec1] # xmm1 = mem[0],zero,zero,zero
movss dword ptr [rip + fRec1+4], xmm1

# fRec2[1] = fRec2[0];
movss xmm1, dword ptr [rip + fRec2] # xmm1 = mem[0],zero,zero,zero
movss dword ptr [rip + fRec2+4], xmm1

# i0 = i0 + 1
add rax, 1
cmp rdi, rax
jne .LBB0_2
#----- end of loop -----
...

```

Figure 25: Intel x86-64 assembly language, generated using the first (latest) Intel option of the Compiler Explorer at godbolt.org (compiler options `-std=c++17 -O3`).

```
voice(i) = os.osc(400+i*300) : si.smoo;  
process = par(i, 10, voice(i));
```

This is not yet understood.

References

- [1] E. Berdahl and J. O. Smith, “An introduction to the synth-a-modeler compiler: Modular and open-source sound synthesis using physical models,” in *Proceedings of the 10th International Linux Audio Conference (LAC-12)*, CCRMA, Stanford, <http://lac.linuxaudio.org/>, 2012, paper: <http://lac.linuxaudio.org/2012/papers/34.pdf>, software: <http://github.com/eberdahl/SaM.git>.
- [2] D. Fober, Y. Orlarey, and S. Letz, “FAUST architectures design and OSC support,” in *Proceedings of the 14th International Conference on Digital Audio Effects (DAFx-11)*, Paris, France, September 19–23, 2011.
- [3] E. Gaudrain and Y. Orlarey, “A FAUST tutorial,” Sept. 2007, See <https://www.grame.fr>.
- [4] A. Gräf, “Interfacing Pure Data with FAUST,” in *Proceedings of the 5th International Linux Audio Conference (LAC-07)*, TU Berlin, <http://www.kgw.tu-berlin.de/~lac2007/proceedings.shtml>, 2007.
- [5] A. Gräf, “Term rewriting extension for the FAUST programming language,” in *Proceedings of the 8th International Linux Audio Conference (LAC-10)*, Utrecht, <http://lac.linuxaudio.org/>, 2010, <http://lac.linuxaudio.org/2010/papers/30.pdf>.
- [6] P. Hudak, “Conception, evolution, and application of functional programming languages,” *ACM Computing Surveys*, vol. 21, pp. 359–411, Sept. 1989, Available without fee for noncommercial use: <https://ccrma.stanford.edu/~jos/pdf/FunctionalProgramming-p359-hudak.pdf>.
- [7] Y. Orlarey, D. Fober, and S. Letz, “Syntactical and semantical aspects of FAUST,” *Soft Computing*, vol. 8, no. 9, pp. 623–632, 2004.
- [8] Y. Orlarey, A. Gräf, and S. Kersten, “DSP programming with FAUST, Q and SuperCollider,” in *Proceedings of the 4th International Linux Audio Conference (LAC2006)*, ZKM Karlsruhe, <http://lac.zkm.de/2006/proceedings.shtml>, pp. 39–40, 2006, http://lac.zkm.de/2006/proceedings.shtml#orlarey_et_al.
- [9] A. Schmeder, A. Freed, and D. Wessel, “Best practices for open sound control,” in *Linux Audio Conference*, (Utrecht, NL), 01/05/2010 2010.
- [10] J. O. Smith, *Introduction to Digital Filters with Audio Applications*, <https://ccrma.stanford.edu/~jos/filters/>, Sept. 2007, online book.
- [11] J. O. Smith, *Physical Audio Signal Processing*, <https://ccrma.stanford.edu/~jos/pasp/>, Dec. 2010, online book.
- [12] S. Wilson, N. Collins, and D. Cottle, eds., *The SuperCollider Book*, Cambridge, MA: MIT Press, 2011.