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@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

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

1 Introduction

The FAUST *programming language*[1] by Yann Orlarey et al. at Grame [2, 6, 7, 3] 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, a brief overview of the language, and example applications and plugins generated from a simple FAUST program specifying a resonator driven by white noise.[2]

1.1 Installing Faust

On a Fedora Linux system (and presumably many other Linux distributions), say

```
yum search faust
```

[1] The FAUST home page is `http://faust.grame.fr/`. FAUST is included in the Planet CCRMA distribution (`http://ccrma.stanford.edu/planetccrma/software/`). The examples in this tutorial have been tested with FAUST version 0.9.9.2a2. "Faust" is derived from "Functional AUdio STream."

[2] See also the online course on learning FAUST, based on Romain Michon's FAUST Workshops:
`https://ccrma.stanford.edu/~rmichon/faustWorkshops/course2015/`
`https://ccrma.stanford.edu/~rmichon/faustDay2017/`

to find out about the FAUST-related packages. On a Mac with MacPorts installed, say this:

```
port search faust
```

At the time of this writing (January 2015), there are seven FAUST packages available on the Mac via MacPorts; the only one needed here is the main one, `faust`, but `faustlive-devel` is highly recommended (and would have been used for the examples here had it existed when this document was first written). To install the main FAUST package on a Mac, say `port install faust` (in Terminal.app on your Mac).

On Fedora Linux, there are even more available packages to consider, but again we only need the main ones `faust`, `faust-doc`, and `faust-tools`. If you are a Pd user, look at the output of `yum search faust | grep pd`. If you want to use Open Sound Control (OSC) with FAUST, look at the output of `yum search faust | grep osc`.

1.2 Keeping up with the Latest Faust Distribution

The FAUST versions in the Linux and MacPorts distributions tend to be significantly behind the latest version under development. To check out the latest version (anonymously, read-only), say

```
> git clone https://github.com/grame-cncm/faust
```

where '`>`' denotes your shell prompt, such as in Terminal.app on a Mac. I always use the latest version, and find it to be quite stable. From time to time, update and reinstall as follows:

```
> cd faust
> git pull
> make
> sudo make install
```

To include OSC and HTTP support (which require `liblo` and `libmicrohttp` respectively), say "`make world`" in place of "`make`" above. By default, you are on the `master` branch of the FAUST distribution (say "`git branch`" to see this). Note that there is also a `faust2` branch where the latest developments are taking place, such as compilation of FAUST to `llvm` or JavaScript.

To use Qt GUIs with Faust, you need Qt5 installed. On the Mac, say `sudo port install qt5`. On Linux, say `yum install qt-devel`.

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 ... .

1.3 Faust Examples

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

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

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` on the Mac). If not, there is also `faust2jack` which uses the GNOME ToolKit (GTK), etc.

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`.

More FAUST demos can be found online.[3]

2 Primer on the Faust Language

FAUST is a *functional programming language* [5] in which the "main" function, called `process`, specifies a signal-processing *block diagram*. 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 (,) which combines block diagrams in *parallel*.

Common block-diagram functions are *predefined*, such as "+":

```
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.

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.

Arguably even simpler is

[3] `http://faust.grame.fr/modules/`

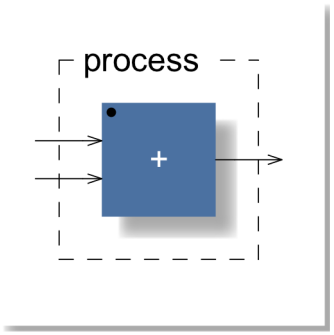

Figure 1: Main *process* block for a two-input adder: `process = +;`

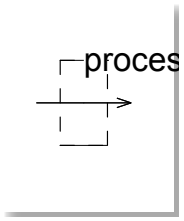

Figure 2: `process = _;`

```
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.

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:

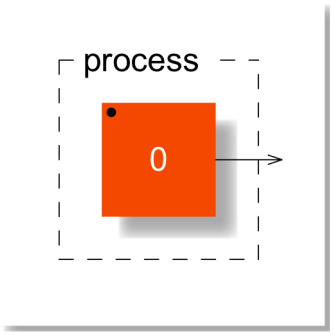

Figure 3: `process = 0;`

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

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 Editor, the Faust Online Compiler, 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* (§2.22) and *definition-expansion* (§2.6)). Postfix operators such as `'` are handled similarly. More formally, we can write

```
   x * y  →    x,y  :  *
   x / y  →    x,y  :  /
   x ^ y  →    x,y  :  ^
 pow(x,y) →    x,y  :  ^
   x @ y  →    x,y  :  @
      x'  →      x  :  mem
    f(x)  →      x  :  f
  f(x,y)  →    x,y  :  f
f(x,y,z)  →  x,y,z  :  f
```

and so on.

2.5 Statements

As described in the FAUST Quick Reference,[4] 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.
- *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):

[4] `http://faudiostream.cvs.sourceforge.net/viewvc/faudiostream/faust/documentation/faust-quick-reference.pdf`

```
process = faust_expression;
```

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

2.6 Function Definition

FAUST is a functional programming language [6]. 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 §2.22 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* (§2.22), 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`.

2.7 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:

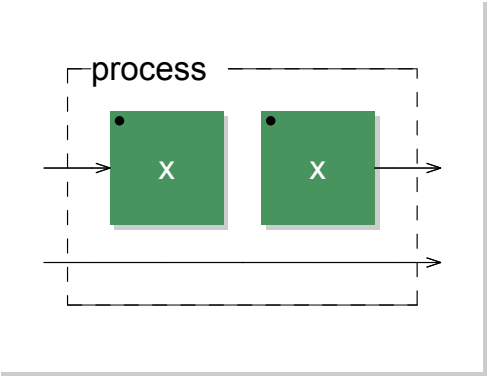

Figure 4: `process(x) = x,_;`

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);
...
```

2.8 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.

2.9 Examples

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

2.10 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."

2.11 Unary Minus

There is a special *unary minus* in FAUST:

```
-x  →  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).

2.12 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.

2.13 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;
```

2.14 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.

2.15 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 §2.24 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.

2.16 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.

2.17 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
```

2.18 Foreign Constants and Variables

FAUST provides linkages to the hosting environment via "foreign" entities. For example, here are some selected declarations from `math.lib`:

```
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).

2.19 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();
```

2.19.1 Example 1

For example,

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

compiles to

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

2.19.2 Example 2

The program

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

compiles to

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

Note that the function usage is identical. That is, FAUST relies on the C++ compiler to carry out any necessary conversions between `int` and `float`.

If the foreign function has no input argument, the inner-loop line becomes

```
output0[i] = (FAUSTFLOAT)fn();
```

2.19.3 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 `<faust_distribution>/architecture/math.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[5]. All of these functions accept a signal and return a signal computed by applying the function to each sample.

[5] `http://faudiostream.cvs.sourceforge.net/viewvc/faudiostream/faust/documentation/faust-quick-reference.pdf`

2.20 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` macro[6]

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

2.21 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.

2.22 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 [5], 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 [4]:

```
import("music.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) * osc(freq(i)); // osc defined in music.lib
process = sum(i, 3, partial(i));
```

[6]The `bus` function was initially defined in the FAUST distribution's `math.lib`, and was moved to `signal.lib` near the end of 2016.

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 [4] 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 §2.22.4 below.

2.22.1 Formal Parameter Exception

The pattern-matching facility is not applied to ordinary formal function parameters [4]. 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.

2.22.2 Recursive Block Diagram Specification

Pattern matching 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:

```
  import("math.lib");   // define bus(n) = par(i,n,_);
//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 `effect.lib` (see `fdnrev0` and `mesh_square()` in `effect.lib`).

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 FAUST *Quick Reference*[7]).

2.22.3 Understanding `count` and `take` from `math.lib`

In `math.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 `math.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.

[7] `http://faudiostream.cvs.sourceforge.net/viewvc/faudiostream/faust/documentation/faust-quick-reference.pdf`

2.22.4 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).

2.22.5 Using Pattern Matching in Rewriting Rules

As mentioned above, rewriting rules are specfied 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.

2.22.6 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)))
```

2.22.7 Pattern-Matching Example

Does the pattern `(a:b)` match `(2+3)`? Yes, as we will see below.[8]

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*.

2.22.8 Pattern-Matching Algorithm Description

The pattern matching algorithm can be roughly recursively defined as follows:[9]

```
// 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 v1, v2, ... in P with subexpressions E1, E2, ... from E to make it identical to E. That is, P matches E if there exist E1, E2, ... such that P[v1=E1, v2=E2, ...] == E.

2.22.9 Miscellaneous Pattern-Matching Examples

The following process evaluates to 7 not 6:

```
a = 2; // this is ignored
f(a,b) = a+b;  // a is a formal argument that is used
process = f(3,4); // 7
```

[8] Thanks to Yann Orlarey for this example and associated discussion on how pattern-matching works in FAUST.
[9] Thanks to Yann Orlarey for this description.

The following evaluates to 6:

```
a = 2; // this one is used (not shadowed by formal arg)
f(c*b) = a+b;
process = f(3*4); // 6
```

The following evaluates to 7:

```
a = 2; // not used - shadowed by pattern variable
f(a*b) = a+b;
process = f(3*4); // 7
```

2.23 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").proc`

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

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

2.24 White Noise Generator

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

```
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.

2.25 Further Readings on the Faust Language

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

See the FAUST Tutorial[10] or FAUST Quick Reference[11] for more examples, features of the language, and discussion.

2.26 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 [11].

```
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`.

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

[10] `http://faudiostream.cvs.sourceforge.net/viewvc/faudiostream/faust/documentation/faust_tutorial.pdf`

[11] `http://faudiostream.cvs.sourceforge.net/viewvc/faudiostream/faust/documentation/faust-quick-reference.pdf`

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. 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`.[12] 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.

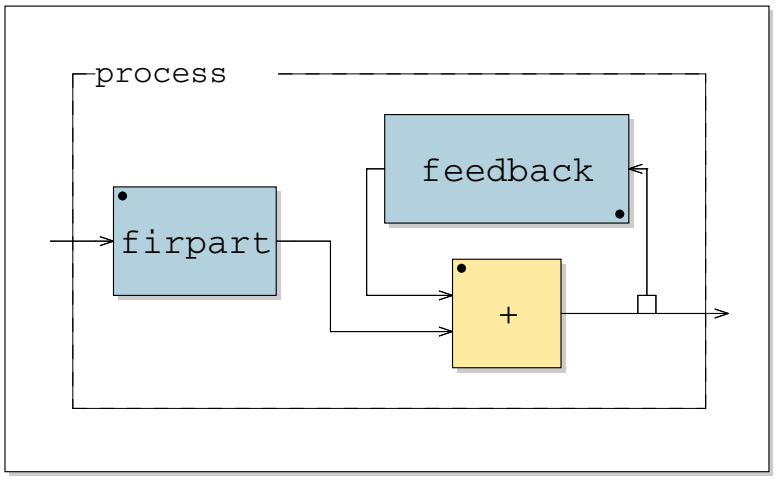

Figure 6: Main *process* block for the constant-peak-gain resonator.

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''`.

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

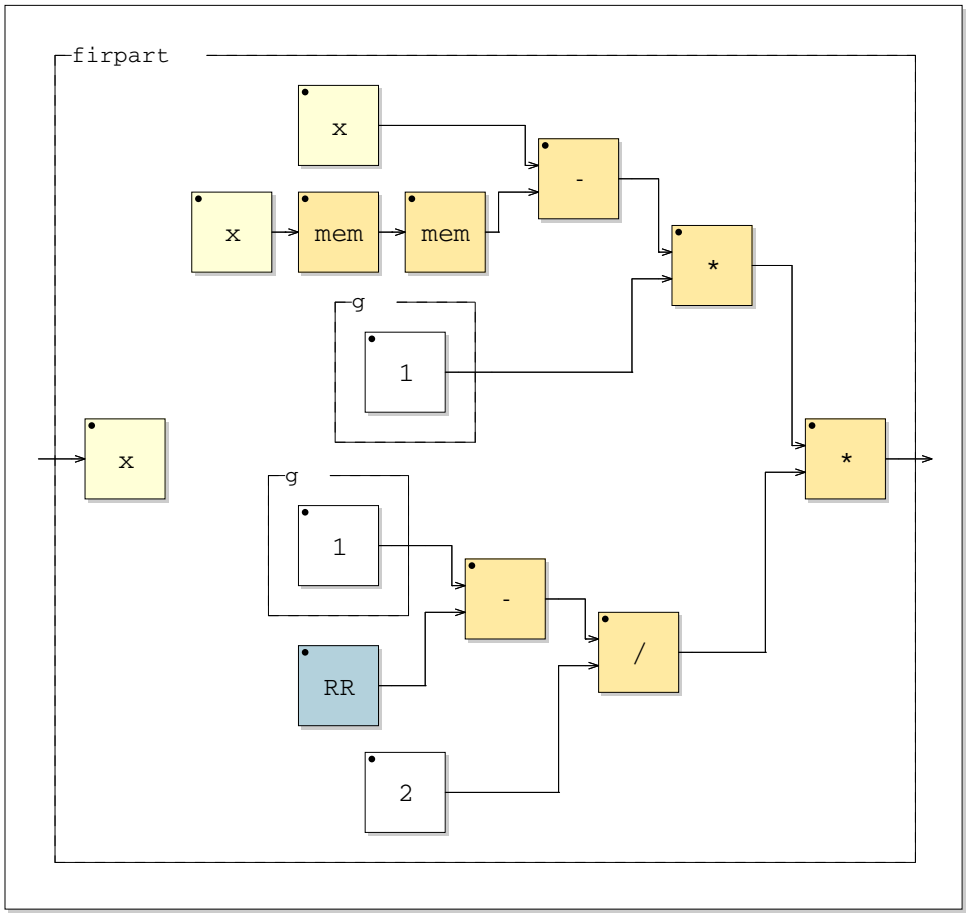

Figure 7: FIR-part `(x - x'') * g * (1-RR)/2` in FAUST.

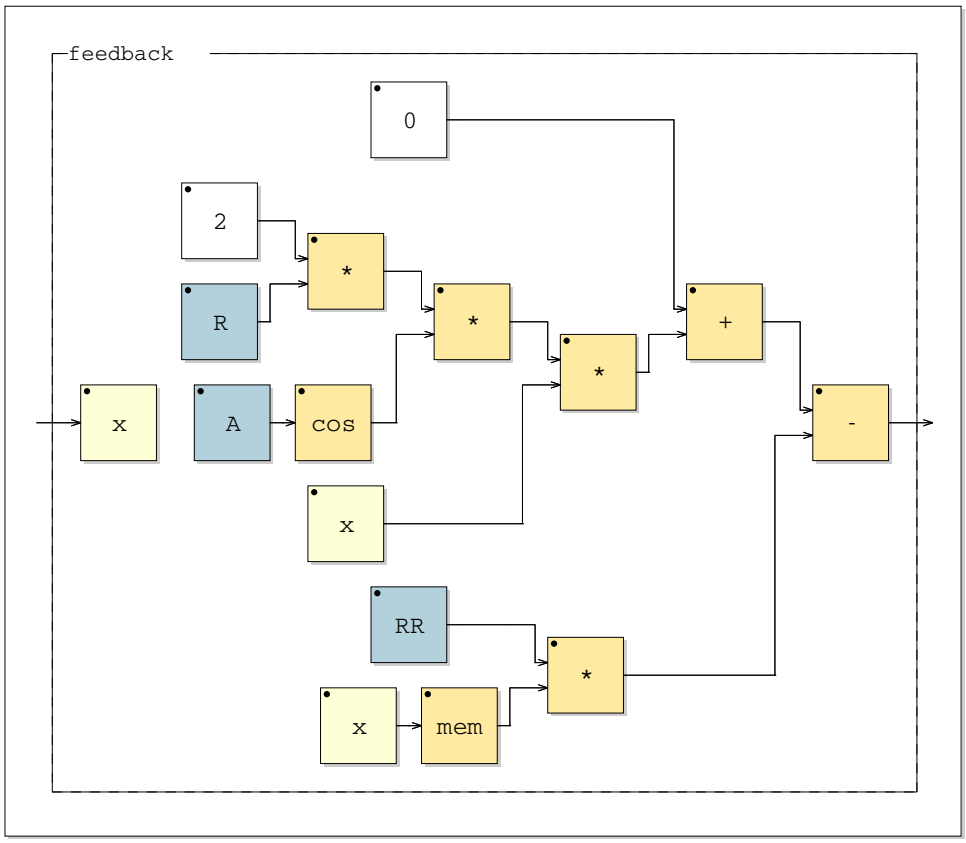

Figure 8: Feedback block `0 + 2*R*cos(A)*x - RR*x'` in FAUST.

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).

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,[13] 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

[13] `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.

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.

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

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:[14]

[14]This 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 = [    0.00707331  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:

```
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 `float`s 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 and one "horizontal slider" 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.

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.[15] 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 (March 2011), the result is as follows:

[15] The 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/
Makefile                faust2graphviewer       faust2octave
README                  faust2jack              faust2paqt
faust2alsa              faust2jackinternal      faust2pdf
faust2caqt              faust2jackserver        faust2plot
faust2csound            faust2jaqt              faust2png
faust2eps               faust2mathdoc           faust2puredata
faust2firefox           faust2mathviewer        faust2svg
faust2graph             faust2msp
```

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. The `Makefile` there has a help target. Just cd there and invoke it:

```
> make help
```

As of March 2011, the result is as follows (edited to fit the printing margins):

```
make alsagtk       : compile examples as ALSA applications with a GTK GUI
make alsaqt        : compile examples as ALSA applications with a QT4 GUI
make sndfile       : compile examples as command-line sound file processors
make jackconsole   : compile examples as command-line JACK applications
make jackgtk       : compile examples as JACK applications with a GTK GUI
make jackqt        : compile examples as JACK applications with a QT4 GUI
make jackwx        : compile examples as JACK applications with a wxWindows GUI
make ossgtk        : compile examples as OSS applications with a GTK GUI
make osswx         : compile examples as OSS applications with a wxWindows GUI
make pagtk         : compile examples as PortAudio applications with a GTK GUI
make paqt          : compile examples as PortAudio applications with a QT4 GUI
make pawx          : compile examples as PortAudio applications with a wxWindows GUI
make caqt          : compile examples as CoreAudio applications with a QT4 GUI
-------------------------------------------
make ladspa        : compile examples as LADSPA plugins
make csound        : compile examples as CSOUND opcodes
make csounddouble  : compile examples as double precision CSOUND opcodes
make maxmsp        : compile examples as Max/MSP externals
make vst           : compile examples as native VST plugins
make w32vst        : crosscompile examples as windows VST plugins
make iphone        : compile examples for Apple iPhone/iPod
make supercollider : compile examples as Supercollider plugins
make puredata      : compile examples as Puredata externals
make q             : compile examples as Q plugins
-------------------------------------------
make svg           : generate the examples block-diagrams in SVG format
make mathdoc       : generate the examples math documentation in TEX and PDF formats
make bench         : compile examples as command line benchmarks
```

```
make plot          : compile examples as command line programs that print samples
make matlabplot    : like plot but printing in matlab-loadable format
-------------------------------------------
make clean         : remove all object files
```

Studying the makefiles invoked by these make targets will inform you of the underlying details of compilation and (when applicable) linking. The `faust2*` scripts implement these details. Note that in many cases (such as in `Makefile.qtcompile`, there is automatic support for both Linux and Mac OS X. The makefiles (presently) cover more than what is summarized by `make help`. Paraphrasing Richard Feynman,[16] "much more is implemented than has been documented." (◡‿◡)

8 Generating a LADSPA Plugin via Faust

LADSPA stands for "Linux Audio Developer Simple Plugin API", and it is the most common audio plugin API for Linux applications. 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:

```
> 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 adding[17] 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.)

[16] who said "Much more is known than has been proved."

[17] After running `jack-rack`, the LADSPA plugin was added by clicking on the menu items "Add / Uncategorised / 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/`.

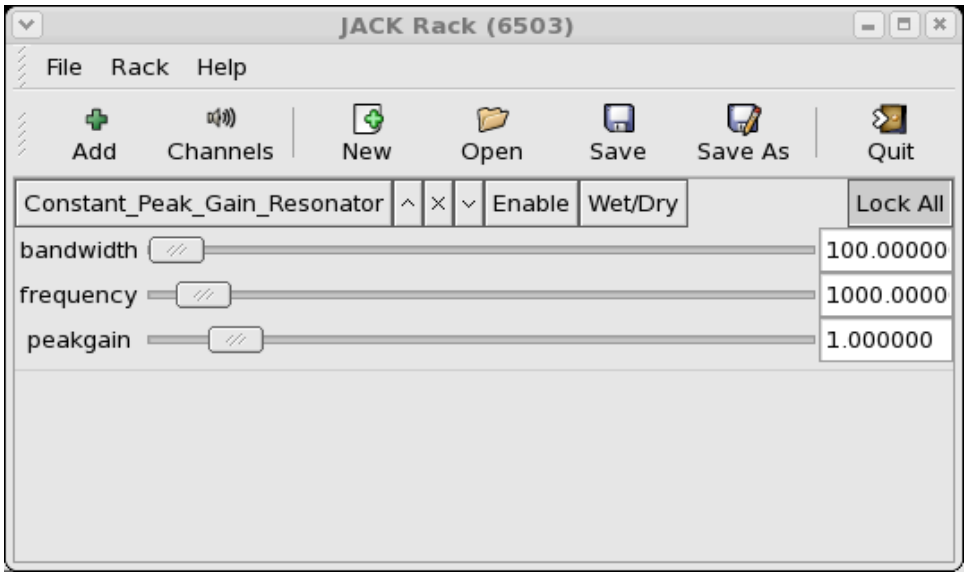

Figure 13: JACK Rack screenshot after adding the LADSPA plugin `Constant_Peak_Gain_Resonator`. Additional LADSPA plugins can be loaded in the space below (and connected in series).

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 routed through `jack-rack` to `alsa_pcm` as shown in Fig. 14.[18]

9 Generating a Pure Data (PD) Plugin

This section illustrates making a `pd` plugin using the FAUST architecture file `puredata.cpp`, and Albert Gräf's `faust2pd` script (version 2.3, tested on Fedora 14 Linux and Mac OS X v10.6.6).[19] Familiarity with Pure Data (the `pd` program by Miller Puckette [8, 9]) is assumed in this section. Also, the original `faust2pd` paper [3] contains the most complete description of `faust2pd` at the time of this writing.

9.1 Generating a Pd Plugin

A Pure Date (Pd) plugin may be compiled on Linux as follows:

[18] Sound 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 `aconnect` from the `alsa-utils` package (included with Planet CCRMA).

[19] After installing `faust2pd` and `pure-faust`, I found it convenient to make a symbolic link from `/usr/local/share/q/apps/faust2pd/faust2pd.q` to `/usr/local/bin/faust2pd`. Also, on my Fedora 14 Linux machine, I had to tell the linking loader `ld` how to find `libpure.so.6` for `pure` (`faust2pd` is written in Pure) by creating the file `/etc/ld.so.conf.d/pure.conf` containing the line `/usr/local/lib` and executing (as root) `ldconfig`. The same effect can be had (temporarily) by saying (as root) `ldconfig /usr/local/lib`. You know it is time to run `ldconfig` again when running `pure` or `faust2pd` results in the error message `pure: error while loading shared libraries: libpure.so.6: cannot open shared object file: No such file or directory`.

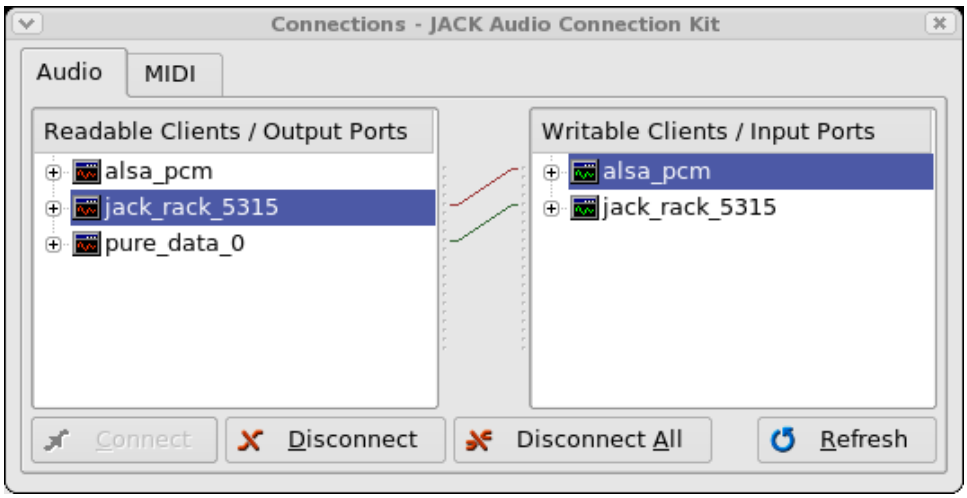

Figure 14: JACK audio connections routing `pd` through `jack-rack` to the ALSA sound-out driver `alsa_pcm`.

```
> faust -a puredata.cpp -o cpgrui-pd.cpp cpgrui.dsp
> g++ -DPD -Wall -g -shared -Dmydsp=cpgrui \
      -I/usr/include/pdextended \
      -o cpgrui~.pd_linux  cpgrui-pd.cpp
```

(If the `g++` compile fails, see the next paragraph.) The first line uses `faust` to generate a compilable `.cpp` file, this time using the architecture file `puredata.cpp` which encapsulates the `pd` plugin API. The second line (which wraps) compiles `cpgrui-pd.cpp` to produce the dynamically loadable (binary) object file `cpgrui~.pd_linux`, which is our signal-processing plugin for `pd`. Such `pd` plugins are also called *externals* (externally compiled loadable modules). The filename extension "`.pd_linux`" indicates that the plugin was compiled on a Linux system.

If the compile fails because `m_pd.h` cannot be found, say

```
> locate m_pd.h
```

to find out where this file is installed, and then modify the `-I` option. The "guess" above (`-I/usr/include/pdextended`) works on the author's Linux machine with Pd-extended installed (`yum install pd-extended`), while on the Mac, the needed include-file is in the application directory:

```
...
-I/Applications/Pd-extended.app/Contents/Resources/include/ \
...
```

Finally, it is normal to have a few `g++` compiler warnings (I see two on Linux).

Figure 15 shows an example test patch,[20] named `cpgrui~-help.pd`,[21] written (manually) for the generated plugin. By convention, the left inlet and outlet of a FAUST-generated plugin correspond to control info and general-purpose messages. Any remaining inlets and outlets are signals.

[20] All manually generated `.dsp` files and `pd` patches in this tutorial are available at

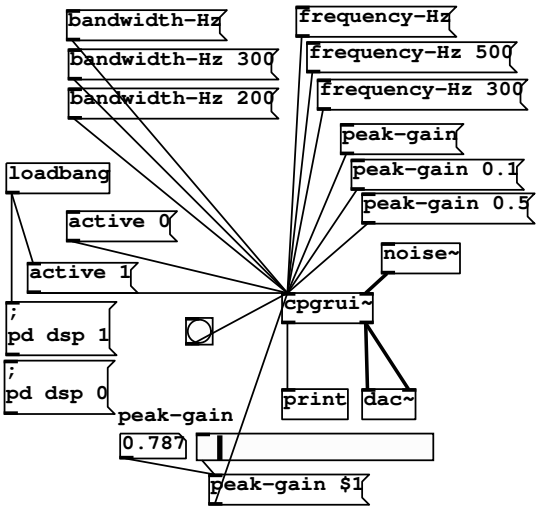

Figure 15: Pure Data test patch `cpgrui~-help.pd` exercising features of the external `pd` plugin `cpgrui~` generated by Faust using the `puredata.cpp` architecture file.

A simple "bang" message to the control-inlet of the plugin (sent by clicking on the "button" drawn as a circle-within-square in Fig. 15), results in a list being sent to the control (left) outlet describing all plugin controls and their current state. The `print` object in Fig. 15 prints the received list in the main `pd` console window. For our example, we obtain the following bang-response in the `pd` console:

```
print: nentry /faust/bandwidth-Hz 100 100 20 20000 10
print: nentry /faust/frequency-Hz 1000 1000 20 20000 1
print: hslider /faust/peak-gain 1 1 0 10 0.01
```

These are the three controls we expected corresponding to the frequency, bandwidth, and gain of the resonator. However, note that the message-names generated for the controls have changed. In particular, spaces have been replaced by hyphens, and parentheses have been removed, to observe `pd` naming rules for messages [3].

Controls may be queried or set to new values in the plugin by sending the following `pd` *messages*:

- `frequency-Hz [newval]`
- `bandwidth-Hz [newval]`
- `peak-gain [newval]`

The longer form of the control name printed in the `pd` console, *e.g.*,
`/faust/peak-gain`, is the complete "fully qualified path" that can be used to address controls within a hierarchy of nested controls and abstractions. For example, if we were to add the instance

`http://ccrma.stanford.edu/realsimple/faust/faustpd.tar.gz`.

[21] In `pd`, a dynamically loadable module (`pd` plugin) is called an *abstraction*. (This is distinct from the *one-off subpatch* which is encapsulated code within the parent patch, and which resides in the same file as the parent patch [9].) It is customary to document each abstraction with its own "help patch". The convention is to name the help patch "name-help.pd", where "name" is the name of the abstraction. Right-clicking on an object in `pd` and selecting "Help" loads the help patch in a new `pd` window.

argument "foo" to the plugin (by changing the contents of the plugin box to "`cpgrui~ foo`" in Fig. 15), then the path to the `peak-gain` control, for example, would become `/foo/faust/peak-gain` (see [3] and the FAUST documentation for more details and examples).

In the test-patch of Fig. 15, the various controls are exercised using `pd` *message boxes*. For example, the message "`peak-gain`" with no argument causes the plugin to print the current value of the peak-gain parameter on its control outlet. Messages with arguments, such as "`peak-gain 0.01`", set the parameter to the argument value without generating an output message. The slider and number-box output raw numbers, so they must be routed through a message-box in order to prepend the controller name ("peak-gain" in this case).

The plugin input signal (second inlet) comes from a `noise~` object in Fig. 15, and the output signal (second outlet) is routed to both channels of the D/A converter (for center panning).

In addition to the requested controls, all plugins generated using the `puredata.cpp` architecture file respond to the boolean "`active`" message, which, when given a "false" argument such as 0, tells the plugin to bypass itself. This too is illustrated in Fig. 15. Note that setting active to "true" at load time using a `loadbang`[22] message is not necessary; the plugin defaults to the active state when loaded and initialized—no `active` message is needed. The `loadbang` in this patch also turns on `pd` audio computation for convenience.

9.2 Generating a PD Plugin-Wrapper Abstraction

The test patch of Fig. 15 was constructed in `pd` by manually attaching user-interface elements to the left (control) inlet of the plugin. As is well described in [3], one can alternatively use the `faust2pd` script to generate a `pd` abstraction containing the plugin and its `pd` controllers. When this abstraction is loaded into `pd`, its controllers are brought out to the top level using the "graph on parent" mechanism in `pd`, as shown in Fig. 17 on page 39.

The `faust2pd` script works from the XML file generated by Faust using the `-xml` option:

```
> faust -xml -a puredata.cpp -o cpgrui-pd.cpp cpgrui.dsp
> faust2pd cpgrui.dsp.xml
```

Adding the `-xml` option results in generation of the file `cpgrui.dsp.xml` which is then used by `faust2pd` to generate `cpgrui.pd`. Type `faust2pd -h` (and read [3]) to learn more of the features and options of the `faust2pd` script.

The generated abstraction can be opened in `pd` as follows:

```
> pd cpgrui.pd
```

Figure 16 shows the result. As indicated by the `inlet~` and `outlet~` objects, the abstraction is designed to be used in place of the plugin. For this reason, we will refer to it henceforth as a *plugin wrapper*.

Notice in Fig. 16 that a plugin wrapper forwards its control messages (left-inlet messages) to the encapsulated plugin, as we would expect. However, it also forwards a copy of each control message to its control outlet. This convention facilitates making *cascade chains* of plugin-wrappers, as illustrated in `faust2pd` examples such as `synth.pd`.[23]

[22] The `loadbang` object sends a "bang" message when the patch finishes loading.

[23] On a Linux system with Planet CCRMA installed, the command "`locate synth.pd`" should find it, *e.g.*, at `/usr/share/doc/faust-pd-0.9.8.6/examples/synth/synth.pd` .

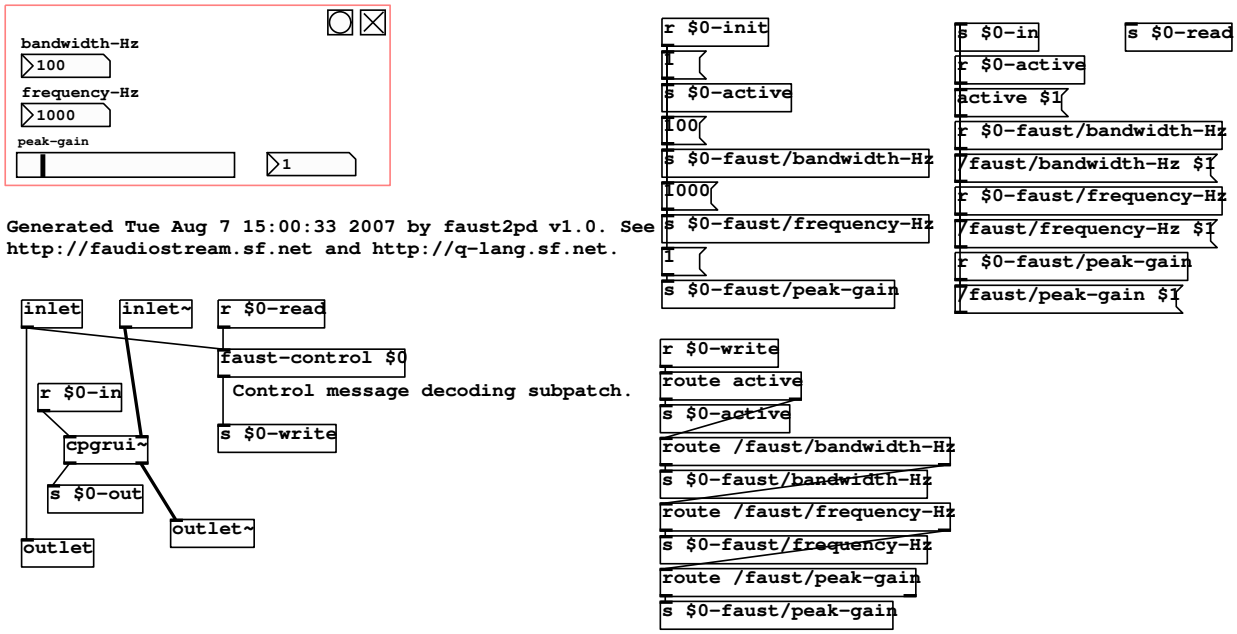

Figure 16: Pure Data abstraction generated by `faust2pd` from the XML file emitted by FAUST for the constant-peak-gain resonator (`cpgrui.dsp`).

9.3 A PD Test Patch for the Plugin Wrapper

Figure 17 shows `pd` patch developed (manually) to test the plugin wrapper generated by `faust2pd`. Compare this with Fig. 15 on page 36. Notice how the three controls are brought out to the plugin-wrapper object automatically using the "graph on parent" convention for `pd` abstractions with controllers. The bang button on the plugin resets all controls to their default values, and the toggle switch in the upper-right corner functions as a "bypass" switch (by sending the `active` message with appropriate argument). The previous mechanism of setting controls via message boxes to the control inlet still works, as illustrated. However, as shown in Fig. 16 (or by opening the plugin-wrapper in `pd`), the control outlet simply receives a copy of everything sent to the control inlet. In particular, "bang" no longer prints a list of all controls and their settings, and controls cannot be queried.

10 Generating a MIDI Synthesizer for PD

The `faust2pd` script (introduced in §9 above) also has a mode for generating *MIDI synthesizer plugins* for `pd`. This mode is triggered by use of the `-n` option ("number of voices"). For this mode, the FAUST program should be written to synthesize one voice using the following three 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"

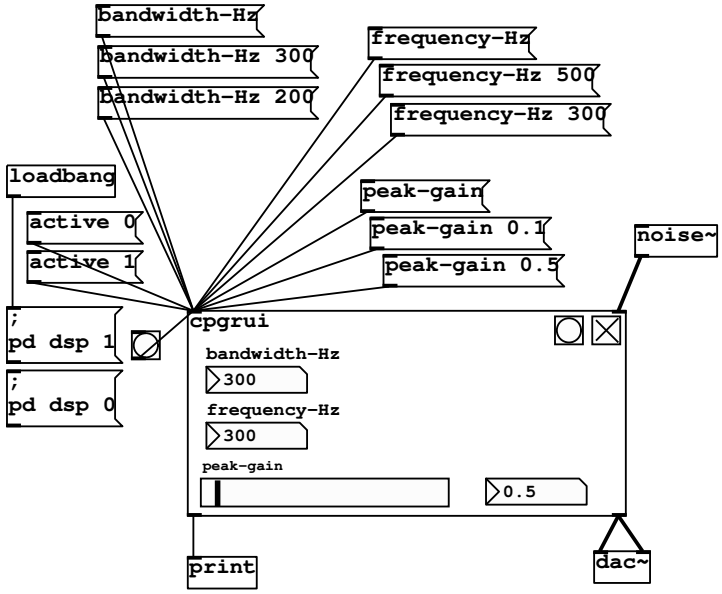

Figure 17: Pure Data test patch (`cpgrui-help.pd`) for exercising the plugin-wrapper (`cpgrui.pd`) generated by `faust2pd` to control the `faust`-generated `pd` plugin (`cpgrui~.pd_linux`).

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 `faust2pd` script handles instantiation of up to 8 instances of the synth patch, and provides the abstraction `midi-in.pd` for receiving and decoding 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. 18. In addition to converting the frequency and gain parameters to the standard names, we have added a classic ADSR envelope generator (defined in FAUST's `music.lib` file) which uses the new `gate` parameter, and which adds the four new envelope parameters `attack`, `decay`, `sustain`, and `release`.

Compiling the example is the same as for a `pd` plugin, except that the `-n` option is used (8 voices is the maximum):

```
> faust -xml -a puredata.cpp -o cpgrs-pd.cpp cpgrs.dsp
> g++ -DPD -Wall -g -shared -Dmydsp=cpgrs \
      -o cpgrs~.pd_linux cpgrs-pd.cpp
> faust2pd -n 8 -s -o cpgrs.pd cpgrs.dsp.xml
```

11 MIDI Synthesizer Test Patch

The example synth is loaded into `pd` like any plugin-wrapper. A manually written test patch (`cpgrs-help.pd`) is shown in Fig. 19. Note that the standard MIDI-synth control parameters

```
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);

import("music.lib"); // define noise, adsr, PI, SR, et al.

/* 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 = noise * env * gain : filter
with {
  env = gate :
        vgroup("1-adsr",
               adsr(attack, decay, sustain, release));
  filter = vgroup("2-filter", (firpart : + ~ feedback));
  R = exp(-PI*bw/SR); // pole radius
  A = 2*PI*freq/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 18: Listing of `cpgrs.dsp`—a FAUST program specifying a simple synth patch consisting of white noise through a constant-peak-gain resonator.

(`freq, gain, gate`) are handled behind the scenes and do not appear among the plugin GUI controls.

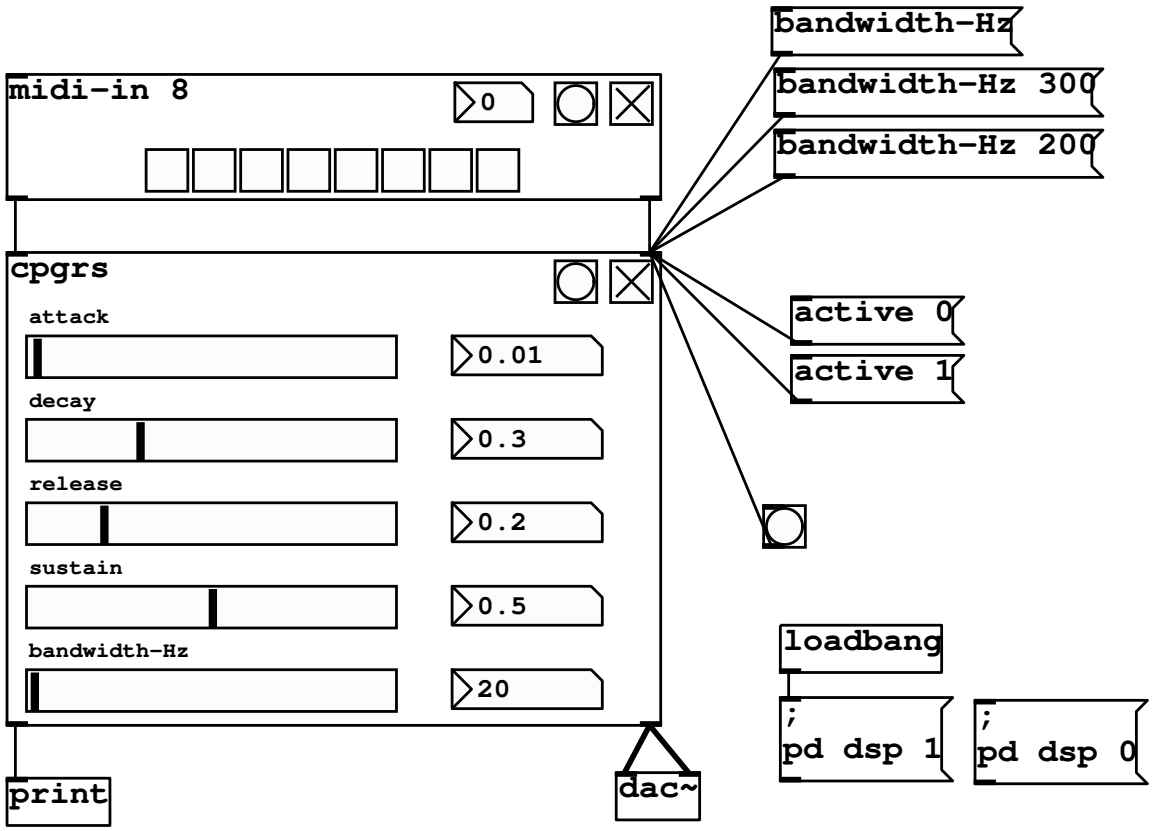

Figure 19: Test patch for the `pd` synth plugin `cpgrs.pd` generated by `faust2pd` based on `cpgrs.dsp` in Fig. 18.

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 6, and in the `planetccrma-menus` at "Applications / Planet CCRMA / MIDI / Vkeybd"). Figure 20 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 21 illustrates the MIDI tab of `qjackctl`'s Connect window after connecting the Virtual

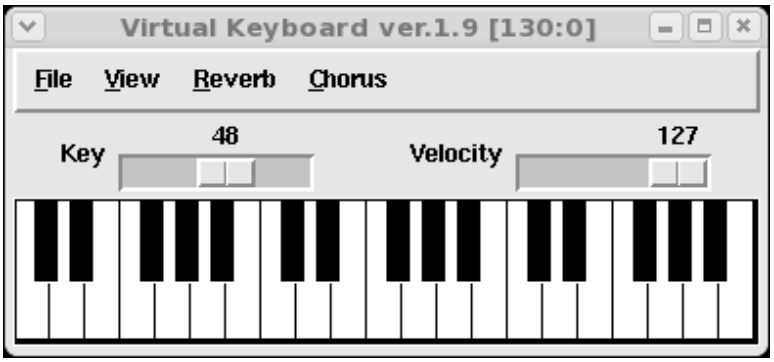

Figure 20: The Virtual Keyboard (MIDI source).

Keyboard MIDI output to `pd`'s MIDI input.[24]

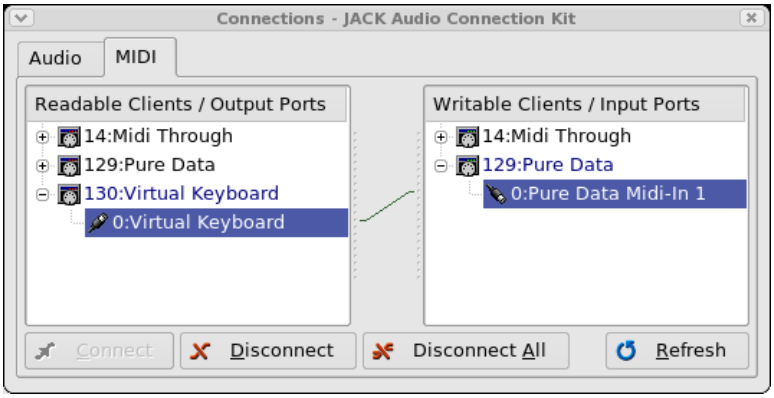

Figure 21: JACK MIDI connections routing MIDI from the Virtual Keyboard (`vkeybd`) to `pd`'s first MIDI input port.

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.)

[24] Pd 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 Using Faust with SuperCollider

This section describes and illustrate making SuperCollider (SC) plugins from FAUST source on Linux and Mac OS X.

13 Getting Started with SuperCollider

The reader is assumed to have worked through a first tutorial on SuperCollider. In particular, the "Getting-Started" tutorial in the SuperCollider online help[25] is especially thorough on the basics, and clearly written. Additional tutorials may be found via the CCRMA SuperCollider Wiki Page.[26] 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 [7]. 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.

13.1 Linux and Faust-Generated SuperCollider Plugins

The examples below were last tested successfully on a Fedora 15 64-bit Linux system running FAUST version 0.9.43 (on 8/4/2011).

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` "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/.
```

Similarly, in the FAUST `examples` directory, one can say

```
make sc
```

to make SC plugins from all the examples. The same goes for the `examples/faust-stk` directory. 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
```

[25] `http://supercollider.svn.sourceforge.net/viewvc/supercollider/trunk/common/build/Help/Help.html`
[26] `https://ccrma.stanford.edu/wiki/SuperCollider`

Next, restart `sclang` and `scsynth`,[27] and execute the following code in `sclang`:

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

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;
```

13.2 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`.

[27] After 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`.

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` master FAUST distribution, as described in §1.2. This document is updated to stay in sync with that (latest) version, as opposed to any particular prior FAUST release.

14 Using Faust with Open Sound Control (OSC)

FAUST contains some very nice Open Sound Control (OSC) support facilities [1, 10]. 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 '#'. (The printout examples below were obtained on a 2013 MacBook Pro.)

In Terminal Window 1 (application window):

```
> cd <faust>/examples
> faust2jack -osc noise.dsp
  # Drill down on the above line 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 below 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 all the FAUST examples and exercise a few:

In Window 1:

```
> cd <faust>/examples/
> make jackgtk OSC=1
> cd ./jackgtkdir/
> ./cubic_distortion
```

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

```
# Faust OSC version ssssiii "0.91" "-" "'cubic_distortion'" "is running on
#   UDP ports " 5510 5511 5512
```

which is essentially the same information printed out in Window 1 when the app is run.

In Window 3:

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

> oscsend localhost 5510 /\* s "get"
  # /cubic_distortion//SINE_WAVE_OSCILLATOR_oscrs//Amplitude fff -38.299999 -120.000000 10.000000
  # /cubic_distortion//SINE_WAVE_OSCILLATOR_oscrs//Frequency fff 37.599998 1.000000 88.000000
  # /cubic_distortion//SINE_WAVE_OSCILLATOR_oscrs//Portamento fff 0.846000 0.000000 1.000000
  # /cubic_distortion//CUBIC_NONLINEARITY_cubicnl//Bypass fff 0.000000 0.000000 1.000000
  # /cubic_distortion//CUBIC_NONLINEARITY_cubicnl//Drive fff 0.630000 0.000000 1.000000
  # /cubic_distortion//CUBIC_NONLINEARITY_cubicnl//Offset fff 0.000000 0.000000 1.000000
  # /cubic_distortion//SPECTRUM_ANALYZER_CONTROLS/Level_Averaging_Time fff 0.510000 0.000000 1.00000
  # /cubic_distortion//SPECTRUM_ANALYZER_CONTROLS/Level_dB_Offset fff 18.000000 0.000000 100.000000

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

> oscsend localhost 5510 /cubic_distortion//CUBIC_NONLINEARITY_cubicnl//Drive s "get"
  # /cubic_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
> zita_rev1
  # Faust OSC version ssssiii "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 23520.000
  # /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
```

and another:

```
  ^C
> parametric_eq
  # Faust OSC version ssssiii "0.91" "-" "'parametric_eq'" "is running on UDP ports " 5510 5511 5512

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

> oscsend localhost 5510 /\* s "get"
  # /parametric_eq//SAWTOOTH_OSCILLATOR//Amplitude fff -13.100000 -120.000000 10.000000
  # /parametric_eq//SAWTOOTH_OSCILLATOR//Frequency fff 23.379999 1.000000 88.000000
  # /parametric_eq//SAWTOOTH_OSCILLATOR//Detuning_1 fff -0.100000 -10.000000 10.000000
  # /parametric_eq//SAWTOOTH_OSCILLATOR//Detuning_2 fff 0.100000 -10.000000 10.000000
  # /parametric_eq//SAWTOOTH_OSCILLATOR//Portamento fff 0.100000 0.010000 1.000000
  # /parametric_eq//SAWTOOTH_OSCILLATOR//Alternate_Signals/Pink_Noise_Instead_(uses_only_Amplitude_
  # /parametric_eq//SAWTOOTH_OSCILLATOR//Alternate_Signals/External_Input_Instead_(overrides_Sawtoo
  # /parametric_eq//PARAMETRIC_EQ_SECTIONS/Low_Shelf/Low_Boost|Cut fff 0.000000 -40.000000 40.000000
  # /parametric_eq//PARAMETRIC_EQ_SECTIONS/Low_Shelf/Transition_Frequency fff 987.000000 1.000000 50
  # /parametric_eq//PARAMETRIC_EQ_SECTIONS/Peaking_Equalizer/Peak_Boost|Cut fff 0.000000 -40.000000
  # /parametric_eq//PARAMETRIC_EQ_SECTIONS/Peaking_Equalizer/Peak_Frequency fff 54.000000 1.000000
  # /parametric_eq//PARAMETRIC_EQ_SECTIONS/Peaking_Equalizer/Peak_Q fff 26.500000 1.000000 50.000000
  # /parametric_eq//PARAMETRIC_EQ_SECTIONS/High_Shelf/High_Boost|Cut fff -0.000000 -40.000000 40.000
  # /parametric_eq//PARAMETRIC_EQ_SECTIONS/High_Shelf/Transition_Frequency fff 2169.000000 20.000000
  # /parametric_eq//SPECTRUM_ANALYZER_CONTROLS/Level_Averaging_Time fff 0.100000 0.000000 1.000000
  # /parametric_eq//SPECTRUM_ANALYZER_CONTROLS/Level_dB_Offset fff 36.000000 0.000000 100.000000
    ^C
```

Finally, let's make OSC JACK apps out of all the Faust-STK examples and try a couple:

```
> cd <faust>/examples/faust-stk/
> make jackgtk OSC=1
> cd ./jackgtkdir/
> ./piano
  # Faust OSC version ssssiii "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
> ./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 0.000
  # /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[28] for more advanced techniques such as the use of *OSC aliases* which allow arbitrary OSC messages[29] to be mapped to controller parameters (for use with remote controllers such as TouchOSC on Android that can only transmit predefined messages).

OSC Audio Data

[28] Section 6.8 of `http://faust.grame.fr/docs/faust-quick-reference.pdf`

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

The FAUST OSC documentation also describes how to use OSC for audio signal streams using an OSC-audio (`oscio-*`) architecture file.

15 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.

15.1 Offline Processing of Soundfiles in Faust

The `sndfile.cpp` architecture file uses Erik de Castro Lopo's `libsndfile` library to process the channels of an input soundfile to produce 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. 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.

15.2 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:[30]

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

15.3 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.

16 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.

References

[1] 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.

[2] E. Gaudrain and Y. Orlarey, "A FAUST tutorial," Sept. 2007, `http://www.grame.fr/pub/faust_tutorial.pdf`.

[3] 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, `http://www.grame.fr/ressources/publications/lac07.pdf`.

[4] 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`.

[30] Additionally there are `faust2jackinternal` and `faust2jackserver` which are beyond the scope of this tutorial.

[5] 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`.

[6] Y. Orlarey, D. Fober, and S. Letz, "Syntactical and semantical aspects of FAUST," *Soft Computing*, vol. 8, no. 9, pp. 623–632, 2004.

[7] 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`.

[8] M. Puckette, *Pure Data (PD)*, `http://www.puredata.org`, July 2004.

[9] M. Puckette, *Theory and Techniques of Electronic Music*, http://www.worldscibooks.com/compsci/6277.html: World Scientific Press, May 2007, `http://www-crca.ucsd.edu/~msp/techniques.htm`.

[10] A. Schmeder, A. Freed, and D. Wessel, "Best practices for open sound control," in *Linux Audio Conference*, (Utrecht, NL), 01/05/2010 2010.

[11] J. O. Smith, *Introduction to Digital Filters with Audio Applications*, `https://ccrma.stanford.edu/~jos/filters/`, Sept. 2007, online book.

[12] S. Wilson, N. Collins, and D. Cottle, eds., *The SuperCollider Book*, Cambridge, MA: MIT Press, 2011.