Ubiquitous Music Ecosystems: Faust Programs in Csound

Victor Lazzarini[1], Damián Keller[2], Marcelo Pimenta[3], Joseph Timoney[1]

[1]Sound and Music Research Group
National University of Ireland, Maynooth
Co. Kildare Ireland

[2]Amazon Center for Music Research - NAP
Universidade Federal do Acre - Federal University of Acre

[3]Computer Science Department
Universidade Federal do Rio Grande do Sul

{victor.lazzarini,joseph.timoney}@nuim.ie,

dkeller@ccrma.Stanford.EDU, mpimenta@inf.ufrgs.br

Abstract. *This paper describes the combination of two high-level audio and music programming systems, Faust and Csound. The latter is a MUSIC N-derived language, with a large set of unit generators, and a long history of development. The former is a purely functional language designed to describe audio processing algorithms that can be compiled into a variety of formats. The two systems are combined in the Faust Csound opcodes, which allow the on-the-fly programming, compilation and instantiation of Faust DSP programs in a running Csound environment. Examples are presented and the concept of ubiquitous music ecosystem is discussed.*

Resumo. *Este artigo descreve a combinaao de dois sistemas de programação de áudio e música, Faust e Csound. Esta 'e uma linguagem derivada de MUSIC N, com um conjunto grande de unidades geradoras, e uma longa história de desenvolvimento. Aquela é uma linguagem funcional pura desenhada para descrever algoritmos de processamento de audio, que pode ser compilada em vários formatos. Os dois sistemas são combinados nos opcodes Faust Csound, que permitem a programação, compilação, e instanciamento imediato de programas em um ambiente Csound. Exemplos são apresentados e o conceito de um eco-sistema para música ubíqua é discutido.*

1. Introduction

Audio signal processing algorithms can be expressed and implemented in a variety of environments. These range from the lower-level of microcode and assembler programming, to high-level matrix-manipulation programs such as MatLab and Octave, and patching systems such as PD or MaxMSP[Puckette 2002]. In general, the advantage of higher-level specifications is that the algorithm is presented compactly in encapsulating blocks, which afford good readability and are easily manipulated. On the other hand, such gains are normally accompanied by a loss of computational efficiency, especially in the case

of general-purpose systems. In high-level realtime audio programming systems, where processes can be run efficiently, there is a limit of what can be expressed, if compared to lower-level environments.

In this scenario, we find that languages that can sit at a middle-level in terms of complexity are optimally placed to provide efficiency and generality to allow the design and implementation of audio processes. In this paper, we will describe the combination of two such systems, Csound and Faust, in the development of support tools for ubiquitous music making[Keller et al. 2011]. The article is organised as follows. First, we will introduce the two systems and discuss their characteristics. The embedding of Faust in Csound will then be detailed, with some use examples. Finally, the paper will conclude with a discussion of a proposal for a new concept for audio programming in a multi-language environment: the ubiquitous music ecosystem.

2. Csound

Csound[Vercoe 1986] is a heir to the MUSIC N systems derived from Mathews' MUSIC IV[Mathews and Miller 1964]. Although it still allows a traditional score + orchestra programming approach, it is not limited to it. The system is built around a library[ffitch 2005] that is accessed through its API, and is manipulated via a variety of frontends, the most basic of them being the command-line interface (CLI) program `csound`. The API can be used directly from a number of languages (C, C++, Java, Clojure, Python, Lua, among others).

Most of Csound programming is done through its orchestra language. In this, the majority of the code is structured around blocks called *instruments*, where it is based on the simple forms of

```
[out,...] opcode [in, ...]
```

or

```
[out =] opcode([in,...])
```

where `opcode` is a given unit generator that will either generate an output, process, or just consume an input. In the second form, inputs can be taken directly from other opcodes, by function composition. In general, inputs can be expressions of any complexity.

Instruments are defined between the keywords `instr` and `endin`. They will contain code that is executed sequentially in two separate stages: at initialisation time (only once), and at performance time (continuously in an implicit loop). For instance,

```
instr 1
i1 = 1000
a1 rand i1
     out a1
endin
```

The first line, `i1 = 1000`, is executed at init-time only, whereas the second will be executed at init-time (where the opcode `rand` is initialised), and at perf-time, producing audio at its output. The third line, `out a1`, is executed at perf-time only. The

Csound compiler automatically assigns code to init- or perf-time depending on the types of variables used, which are defined by the first letter of their name. For instance, init-time numeric variables start with `i`, whereas `a` types are audio variables and thus imply code that is run at perf-time.

There is an implicit perf-time loop, which makes the instrument compute audio in blocks of `ksmps` samples, so that a-variables are constantly updated with new blocks of samples. There are other variable types, which follow similar rules. Particularly relevant to our discussion is the `k`-type, which holds a single sample at perf-time and is normally used to carry control signals.

Csound has a full complement of arithmetic operators, mathematical functions and control-flow structures, in addition to over 1800 unit generators (opcodes). It can be used to describe any time-domain audio signal processing algorithm. It has also special frequency-domain types and opcodes, which can be used to design spectral processing instruments, although not quite from first principles as in the time-domain case. All aspects of the Csound processing engine can be configured with system parameters passed to it at the start of a session.

3. Faust

Faust[Orlarey et al. 2009] is a purely functional language designed to describe audio streams, with which we can implement any time-domain audio processing algorithm. Its compiler can produce C, C++, Javascript or LLVM code. The compiled code is an efficient audio digital signal processing (DSP) program, that can be then used in a variety of environments (as plugins to various systems, including Csound, or as standalone programs). Signal processing programs created in Faust will generally be more efficient than their equivalent code written in other high-level music programming languages (Csound included).

The Faust program describes a process. For instance, the following minimal program implements a mixer of two input signals:

```
process = + ;
```

Each Faust statement is terminated by a semicolon (`;`). So here we have an arithmetic operator (+), which by definition takes two inputs and produces one output. Summing two signals is the same as mixing them. Similarly, if we want to scale a signal by 2, we can have

```
process = *(2);
```

which is a program of one input and one output, because the multiplication by 2 is a function of one input to one output. Faust programs can also use the sequential (`:`) and parallel (`,`) operators:

```
process = _ ,2 : *;
```

This takes some audio input (_), in parallel with the constant 2, and sends them in sequence (`:`) to the function '*'. The other important primitives are split (`<:`) and merge (`:>`). Here's how to square a signal

```
process =  _ <: *;
```

and another version of the mix-two program above:

```
process = _,_:> _;
```

Faust includes a number of generic User Interface (UI) functions that can be used to create controls. These get compiled to various forms, depending on the target of the compilation. For instance, a horizontal slider is defined by the following line

```
freq = hslider(``frequency'', 440, 100, 1000, 1);
```

where the parameters are, respectively: name label, default value, minimum, maximum, and minimum step. Faust also allows access to C library functions for trigonometric operations, etc. With these and time-counting function `time`, we can for instance, write a sine wave oscillator as

```
process = time : *(2*PI/SR) : *(440) : sin;
```

where PI and SR are constants set to π and the sampling rate.

4. The Faust Csound unit generators

Faust programs can be compiled into C code for Csound opcodes, and built with standard C compiler tools as dynamic library plugins, for loading into the system. This is the standard way to employ Faust to implement signal processing algorithms.

With Faust version 2, however, the compiler system has been redesigned into a library, libfaust. This allows the embedding of the compiler into other programs and environments. In addition to C code, Faust can produce LLVM[Lattner and Adve 2004] bitcode via a just-in-time compiler. This allows the complete compilation and running of a Faust program to happen on-the-fly, under a host.

The Faust Csound unit generators have been built to take advantage of these new capabilities. They can take an arbitrary Faust program, compile it and then create running instances of it within a Csound instrument. So it is possible to implement a completely new process, from first principles and run it as efficiently as compiled C code.

The design of these opcodes mirrors the facilities offered by the libfaust LLVM support, where the following steps are present:

1. **Compilation**: we take in a text string containing Faust code, compile it and create an LLVM factory. This contains the binary representation of a Faust program.
2. **Instantiation**: an LLVM factory is instantiated in memory and initialised as a DSP instance. Any controls provided by the Faust program UI are made available for Csound access.
3. **Performance**: the Faust DSP instances are run, and their inputs and outputs are made available to Csound.

Four opcodes have been implemented to allow this functionality:

```
ifac  faustcompile  Scode, Sargs
```

This opcode implements step 1 above, taking in strings holding the Faust program (Scode), and compiler options (Sargs). It produces a handle to the compiled LLVM factory. It works completely on init-time.

```
idsp,asig[,...]  faustaudio  ifac[,ain, ...]
```

This opcode has a double function of instantiation (at init-time), and performance (at perf-time). It takes in any inputs defined in the Faust program and produces as many outputs as necessary. It also produces a handle to the DSP instance, which is used to access controls defined in the program.

```
faustctl idsp, Sname, kval
```

The faustctl opcode is used to access a control named Sname in the DSP instance idsp, setting it to kval.

```
idsp,asig[,...]  faustgen Scode[,ain, ...]
```

This code implements the three steps in one single operation. At i-time, it compiles and instantiates the program in the string Scode. It then performs the DSP process, working similarly to faustaudio. It is designed for 'one-off' processes, which will not have more than one DSP instance.

4.1. Examples

Our first example shows the complete sine wave oscillator example discussed in section 3. The program is placed in an one-off faustgen opcode (the {{ and }} enclose a multi-line string in Csound) and faustctl is used to set the oscillator frequency:

```
instr 1

 idsp, asig faustgen {{
 PI  = 3.14159265358979323 85;
 SR = 44100;
 freq = hslider("freq", 440,100,1000,1);
 time = (+(1) ~ _ )  - 1;
 process =    time : *(2*PI/SR) : *(freq) : sin;
 }}

 faustctl idsp,"freq", 440

   outs asig
endin
```

The second example is a bit more involved, showing the classic Karplus-Strong program in Faust, embedded in a Csound orchestra that can be controlled via MIDI (in a complete CSD source code). The faustcompile opcode is placed at global level in Csound, so it is run only once, but its factory can be instantiated by any instruments in the orchestra:

```
<CsoundSynthesizer>
<CsOptions>
--midi-velocity-amp=4 --midi-key-cps=5
</CsOptions>
<CsInstruments>
ksmps=100
nchnls=2
0dbfs = 1

giPluck faustcompile {{

import("music.lib");
upfront(x)      = (x-x') > 0.0;
decay(n,x)      = x - (x>0.0)/n;
release(n)      = + ~ decay(n);
trigger(n)      = upfront : release(n) : >(0.0);

size = hslider("excitation", 128, 2, 1024, 1);
dur = hslider("duration", 128, 2, 1024, 1);
att  = hslider("attenuation", 0.1, 0, 1, 0.01);

average(x) = (x+x')/2;
resonator(d, a) = (+ : delay(4096, d-1.5))
                            ~ (average : *(1.0-a)) ;
process = noise * hslider("level", 0.5, 0, 1, 0.01)
: vgroup("excitator", *(button("play"): trigger(size)))
: vgroup("resonator", resonator(dur, att));

}}, "-vec -lv 1"

instr 1
 i3, a1 faustaudio giPluck
 faustctl i3,"level", p4*0.5
 faustctl i3,"duration", sr/(p5)
 faustctl i3,"excitation", sr/(p5)
 faustctl i3,"attenuation", 0.01
 faustctl i3,"play", 1
 kenv linsegr 1,1,1, 0.01, 0
   outs a1*kenv,a1*kenv
endin

</CsInstruments>
<CsScore>
</CsScore>
</CsoundSynthesizer>
```

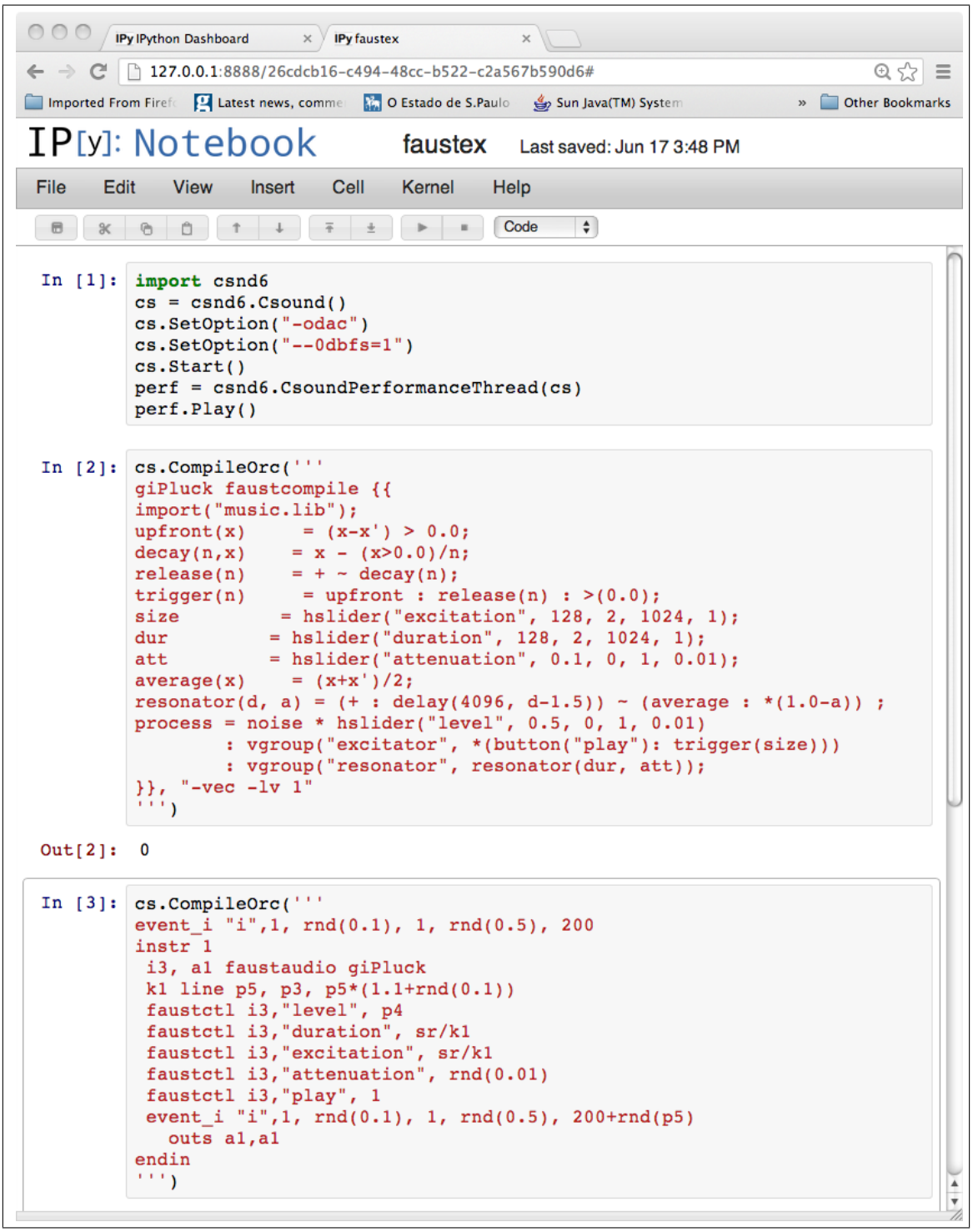

Figure 1. Faust embedded in Csound running under Python in an IPython Notebook.

5. Music Programming in a Multi-language environment

The embedding of a language such as Faust in Csound, and indeed of others such as Python and Lua, as well as the embedding of Csound within other systems, places the question of a multi-language environment for Music Programming at the centre of the ideas expressed in this paper. We believe that such mix of environments not only fits in the separation of concerns paradigm that is commonplace in systems development[Ousterhout 1998][Damasevicius and Stuikys 2002], but also provides a creative hothouse for ubiquitous music.

An example of this is the use of Csound in the IPython Notebook environment, through the Csound API in Python (fig.1). In such environment, it is possible to prototype and perform new instruments interactively, with optional user interfaces, and through protocols such as MIDI and Open Sound Control. Adding Faust to this combination allows us to do write programs to do sample-level computation at the C-language level of performance. Such multi-language, interactive, mix of graphical, controller and text interfaces is an example of a generic, flexible and powerful environment for Computer Music. This functionality is not achieved by any single language system alone. We propose the concept of ubiquitous music ecosystems for this class of creativity-support environments.

Other examples of such multi-language approaches involving Csound include the use of external data-processing systems for algorithmic composition (via its score processor/generator facility); application development for mobile environments such as iOS and Android (aka MCP[Lazzarini et al. 2012]), where Csound is used as the sound engine, while system languages (Objective-C, Java) take care of the UI.

6. Conclusions

In this paper we have explored the embedding of Faust programs in the Csound language. We provided an introduction to the two systems, highlighting the most salient features of each as an example of a creativity support ecosystem for ubiquitous music tool development. The embedding of Faust into host system was briefly explained, and the design of the Csound unit generators was detailed. Four opcodes were developed to provide support for the integration of the two environments. Finally, the paper discussed the merits of multi-language environments for computer music, exploring some scenarios where they demonstrate advantages over single-language systems.

References

Damasevicius, R. and Stuikys, V. (2002). Separation of concerns in multi-language specifications. *Informatica*, 13(3):255–274.

ffitch, J. (2005). The Design of Csound5. In *LAC2005*, pages 37–41, Karlsruhe, Germany. Zentrum für Kunst und Medientechnologie.

Keller, D., Flores, L. V., Pimenta, M. S., Capasso, A., and Tinajero, P. (2011). Convergent trends toward ubiquitous music. *Journal of New Music Research*, 40(3):265–276.

Lattner, C. and Adve, V. (2004). LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, California.

Lazzarini, V., Yi, S., Timoney, J., Keller, D., and Pimenta, M. (2012). The Mobile Csound Platform. In *Proc. Int. Computer Music Conf. 2012, Ljubliuana*. Computer Music Association.

Mathews, M. and Miller, J. E. (1964). *MUSIC IV Programmer's Manual*. Bell Telephone Labs.

Orlarey, Y., Letz, S., and Fober, D. (2009). Automatic Parallelization of FAUST code. In *LAC2009*, Parma, Italy. Casa della Musica.

Ousterhout, J. (1998). Scripting: higher-level programming for the 21st century. *IEEE Computer*, 31(3):23–30.

Puckette, M. (2002). Max at seventeen. *Computer Music Journal*, 26(4):31–43.

Vercoe, B. (1986). *The Csound Reference Manual*. MIT.