Romain Michon - Music Ph.D. Applicant – Writing Sample: Proc. of the 14th Int. Conference on Digital Audio Effects (DAFx-11), Paris, France, September 19-23, 2011

FAUST-STK: A SET OF LINEAR AND NONLINEAR PHYSICAL MODELS FOR THE FAUST PROGRAMMING LANGUAGE

Romain Michon,[*]
CIEREC, EA 3068
Université Jean Monnet
F-42023, Saint-Etienne, France
rmichon@ccrma.stanford.edu

Julius O. Smith,
Center for Computer Research in Music and Acoustics.
(CCRMA) Stanford University
Palo Alto, CA 94305, USA
jos@ccrma.stanford.edu

ABSTRACT

The *FAUST Synthesis ToolKit* is a set of virtual musical instruments written in the FAUST programming language and based on waveguide algorithms and on modal synthesis. Most of them were inspired by instruments implemented in the *Synthesis ToolKit* and the program *SynthBuilder*.

Our attention has partly been focused on the pedagogical aspect of the implemented objects. Indeed, we tried to make the FAUST code of each object as optimized and as expressive as possible.

Some of the instruments in the *FAUST-STK* use nonlinear allpass filters to create interesting and new behaviors. Also, a few of them were modified in order to use gesture data to control the performance. A demonstration of this kind of use is done in the *Pure Data* program.

Finally, the results of some performances tests of the generated C++ code are presented.

1. INTRODUCTION

The *FAUST Synthesis ToolKit*[1] is set of virtual musical instruments programmed in the FAUST[2] programming language. Most of them are based on physical models inspired from the algorithms implemented in the *Synthesis ToolKit (STK)*[3] [1] and the program *SynthBuilder* [2].

The *STK* has been developed since 1996 by P. R. Cook and G. P. Scavone. It is a set of open source audio signal processing and algorithmic synthesis classes written in the C++ programming language that can be used in the development of music synthesis and audio processing software.

SynthBuilder was a program developed at Stanford's CCRMA[4] in the nineties to implement sound synthesis based on physical models of musical instruments. Most of its algorithms use the waveguide synthesis technique but some of them are also based on modal synthesis [3].

An important part of our work consisted of improving and simplifying the models from these two sources in order to make them more efficient thanks to the FAUST semantic. All FAUST code in the *FAUST-STK* is commented, including frequent references to external bibliographical elements. Finally, many of the algorithms from the *STK* and *SynthBuilder* were upgraded with nonlinear allpass filters.

First, we will present the different models of musical instruments implemented in the *FAUST-STK*. We will discuss the problems we encountered during their development and then describe the selected solutions. A brief overview on the use of allpass nonlinear filters with waveguide models will be given. Finally, we'll study the performance of the generated C++ code.

2. WAVEGUIDE MODELS

Waveguide synthesis of string and wind instruments was introduced during the 1980s [4, 5, 3]. It can be viewed as descendent of either the Kelley-Lochbaum vocal-tract model [6, 7] or Karplus-Strong "digitar" algorithm [8, 3]. Waveguide synthesis makes it possible to model any kind of string, bore, or vibrating structures with a network of delay lines and filters. Such waveguide instruments are very suitable for implementation in the FAUST language because of their 1D "stream like" architecture.

We now give a brief overview of the *FAUST-STK* waveguide instruments.

2.1. Wind Instruments

The algorithms used in the *FAUST-STK* are almost all based on instruments implemented in the *Synthesis ToolKit* and the program *SynthBuilder*. Although, it is important to observe that some of them were slightly modified in order to adapt them to the FAUST semantic.

Despite the fact that we used as often as possible functions already defined in the default FAUST libraries to build our models, we often needed to write new filters in order to be able to use the parameters from the *STK* classes and the *SynthBuilder* patches

[*] CCRMA visiting researcher from Saint Étienne University, France. Work carried out in the frame of the ASTREE Project (ANR-08-CORD-003).

[1] http://stk-faust.googlecode.com/

[2] Functional AUdio STream is programming language that proposes an abstract, purely functional approach to signal processing. It has been developed at Lyon's GRAME (Groupe de recherche en Acoustique et en Musique Electronique) since 2002: http://faust.grame.fr/.

[3] https://ccrma.stanford.edu/software/stk/

[4] Center for Computer Research in Music and Acoustics

without transformation. All of these functions were placed in a file called `instrument.lib`.

All the wind instruments implemented in the *FAUST-STK* are based on a similar architecture. Indeed, in most cases, the breath pressure that corresponds to the amplitude of the excitation is controlled by an envelope. The excitation is used to feed one or several waveguides that implement the body of the instrument. For example, in the case of a clarinet, the excitation corresponds to the reed that vibrates in the mouthpiece, and the body of the instrument is the bore and the bell. In Figure 1, it is possible to see the block diagram of one of the two clarinet models that are implemented in the *FAUST-STK*. In that case, an ADSR[5] envelope that is embedded in the *breathPressure* box controls the breath pressure.

The other clarinet implemented in the *FAUST-STK* is a bit more complex as it has a tone hole model that makes it possible to change the pitch of the note being played in a more natural way. Indeed, in the algorithm showed in Figure 1 and as in most of the basic waveguide models, the pitch is modulated by changing the length of the loop delay line which would correspond in "the real world" to changing dynamically the size of the clarinet's bore during the performance, as if it were a trombone.

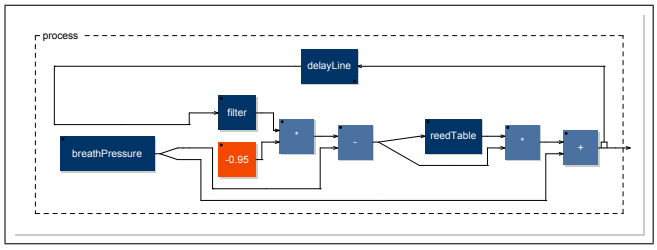

Figure 1: `clarinet.dsp` *algorithm drawn by FAUST using faust2svg.*

The reed table employed with the two clarinets to excite the model was also used to create a very simple saxophone model that is even more comparable to a violin whose strings are excited by a reed.

Two models of flute are implemented in the *FAUST-STK*. The first one is based on the algorithm used in the *Synthesis ToolKit* that is a simplified version of [9]. The other model is showed in Figure 2. It uses two loops and an improved jet filter. (A Butterworth filter is used instead of a simple one-pole.)

A simple model of a brass instrument inspired from a class of the *Synthesis ToolKit* and with a mouthpiece based on the model described in [10] is implemented in the *FAUST-STK*. It can be used to emulate a wide range of instrument such as a french horn, a trumpet or even a trombone. Its algorithm can be seen in Figure 3.

Finally, a tuned bottle in which it is possible to blow through the neck to make sound is also implemented in the *FAUST-STK*.

[5] Attack - Decay - Sustain - Release.

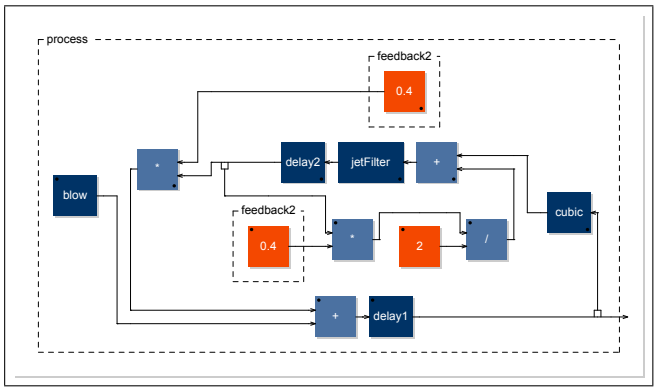

Figure 2: `flute.dsp` *algorithm drawn by FAUST using faust2svg.*

2.2. String Instruments

Some of the waveguide algorithms for plucked strings have already been implemented in FAUST by J. O. Smith that worked in [11] on the extended Karplus-Strong. Although, it is possible to find in the *FAUST-STK* a few models of string instruments such as a Sitar, a bowed-string instrument, a nonlinear extended Karplus-Strong (Cf. §3 about nonlinear waveguide models), an acoustic bass, a piano and an harpsichord (Cf. §6 about keyboards instruments in the *FAUST-STK*).

Except the nonlinear extended Karplus-Strong, all this algorithms are inspired from the *Synthesis ToolKit* and the program *SynthBuilder*.

2.3. Percussion Instruments

Four objects in the *FAUST-STK* use the banded waveguide synthesis technique (described in [12]) to model the following percussion instruments:

- an iron plaque;
- a wooden plaque;
- a glass harmonica;
- a tibetan bowl.

Each of them can be excited with a bow or a hammer.

3. USING NONLINEAR PASSIVE ALLPASS FILTER WITH WAVEGUIDE MODELS

Some of the instruments implemented in the *FAUST-STK* are using nonlinear passive allpass filters in order to generate nice natural and unnatural sound effects [13]. Nonlinear allpass filters can add interesting timbral evolution when inserted in waveguide synthesis/effects algorithms. The nonlinearities are generated by dynamically modulating the filter coefficients at every sample by some function of the input signal. For the instruments that use this kind

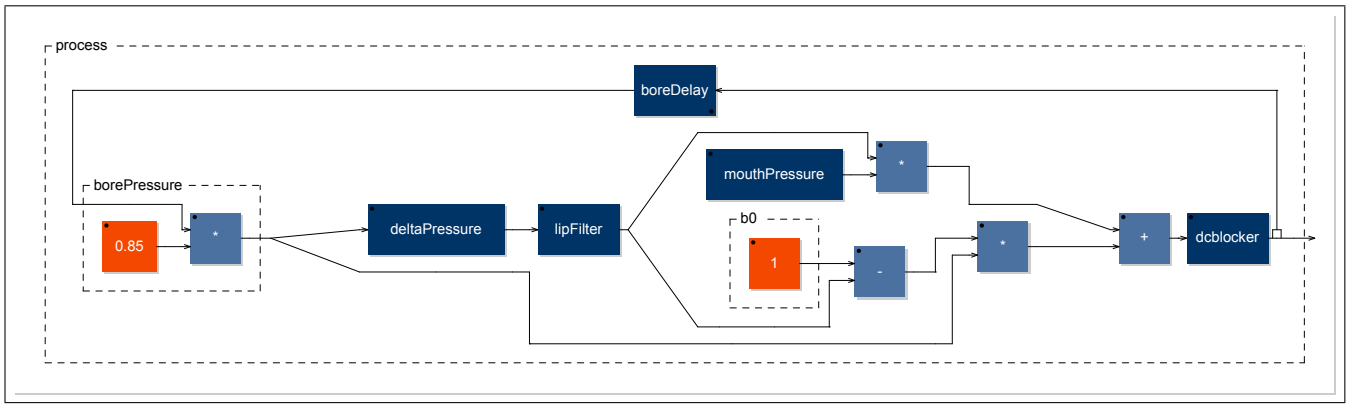

Figure 3: `brass.dsp` *algorithm drawn by FAUST using faust2svg.*

of filter in the *FAUST-STK*, the user can decide whether the coefficients are modulated by the input signal or by a sine wave. In both cases, a "nonlinearity factor" parameter scales the range of the modulation of the filter coefficients. This parameter can be controlled by an envelope in order to make the modulated behavior more natural.

We adjust the length of the delay line of the instruments that use nonlinear allpass filters in function of the nonlinearity factor and of the order of the filter as follows:

$$DL = (SR/F) - FO \times NF \tag{1}$$

where DL is the delay length in number of samples, SR is the sampling rate, F is the pitch frequency, FO is the filter order and NF the nonlinearity factor (value between 0 and 1).

The nonlinear allpass filter can be placed anywhere in the waveguide loop, for example just before the feedback as showed in Figure 4.

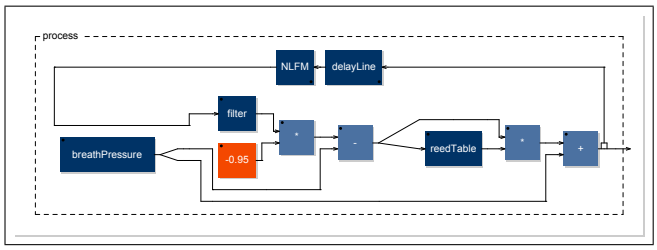

Figure 4: *Modified version of* `clarinet.dsp` *(Cf. figure 1) that uses a nonlinear allpass filter in its feedback loop.*

Finally, it is interesting to mention that we were able to implement a frequency modulation synthesizer in the *FAUST-STK* by using this kind of filter on a sine wave signal. A related result is reported in [14].

4. MODAL MODELS

A set of instruments using modal synthesis can be found in the *FAUST-STK*. They are all implemented in the same code as they are based on the same algorithm.

Modal synthesis was developed primarily by J-M. Adrien in the 1980s [15] and is very similar to the technique used in §2.3 as it consists in exciting a filter-bank with an impulse (Figure 5).

Implementing modal synthesis with FAUST was a bit challenging, as it requires handling a large number of parameters and an excitation signal stored in a wave file. The first problem was solved by using the foreign-function primitive in FAUST which allows using a C++ function within FAUST code. The different values were stored in an array of floats used in a function that takes an index as an argument and that returns the corresponding number.

To solve the other problem of importing a wavetable from a sound file in a FAUST object, we first tried to use the `libsndfile` library developed by E. de Castro Lopo [16] that makes it possible to easily handle wave files in C++. Unfortunately, it appears that this solution was not compatible with all the FAUST architectures. Based on this observation and the fact that the wave tables used in the STK had a maximal size of 1024 samples, we decided to use the same technique as the one previously explained. Indeed, the raw data were extracted from the wave file to be put in an array of floats that can be used in a C++ function to return the values with an index. This C++ function can then be called in FAUST using the foreign-function mechanism to fill a buffer with the `rdtable` primitive.

5. VOICE SYNTHESIS

A very simple voice synthesizer based on the algorithm from the *Synthesis ToolKit* is implemented in the *FAUST-STK*. It uses a lowpass-filtered impulse-train to excite a bank of 4 bandpass filters that shape the voice formants. The formant parameters are stored in a C++ function in the same way described in §4 as a set of cen-

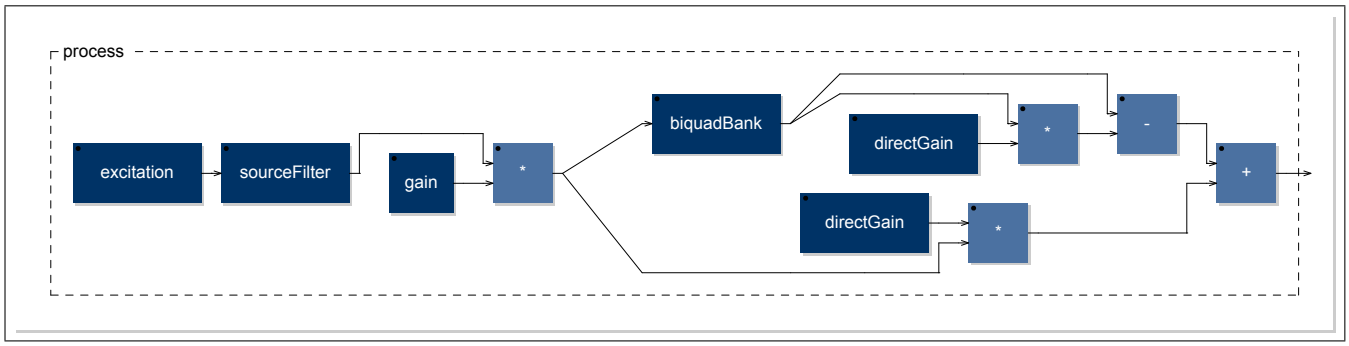

Figure 5: `modalBar.dsp` *algorithm drawn by FAUST using faust2svg.*

ter frequencies, amplitudes, and bandwidths. This function is then called in the FAUST code using the foreign function primitive. The thirty-two phonemes stored in this function are the same as in the *Synthesis ToolKit*.

6. KEYBOARDS

A *SynthBuilder* patch implementing a commuted piano [17] was written in the late 1990s at Stanford's CCRMA. This patch was partly ported in 2006 by Stephen Sinclair at McGill University in the *Synthesis ToolKit* [18]. A big part of his work consisted of extracting parameter-values from the *SynthBuilder* patch and storing them in a set of C++ functions. We reused them to build our FAUST commuted piano version by using the *foreign function* mechanism as described in §4.

In this piano model, the keyboard is split in two parts, each using a different algorithm: The tones below *E6* use the commuted waveguide synthesis technique [3] while tones above or equal to *E6* use modal synthesis (a series of biquad filters) to generate the sound (Figure 6).

A commuted harpsichord has also been implemented in the *FAUST-STK*. It was inspired by another *SynthBuilder* patch that uses a very similar algorithm to the one described above.

The current FAUST versions of the commuted piano and harpsichord are not polyphonic. However, the *faust2pd* program developed by Albert Graef [19] makes it possible to automatically produce *Pure Data* patches that implement polyphonic synthesizers that use FAUST generated Pd plug-ins. They can then be controlled via MIDI or OSC directly in Pure Data.

7. USING A FAUST-STK PHYSICAL MODEL WITH GESTURE-FOLLOWING DATA

Parameter values are very important when dealing with physical modeling. Indeed, even if in most cases it is possible to produce nice sounds with static values for each parameter, the sound quality can be improved a lot by using dynamic values that can describe better the state of the model as a function of the note and the amplitude being played.

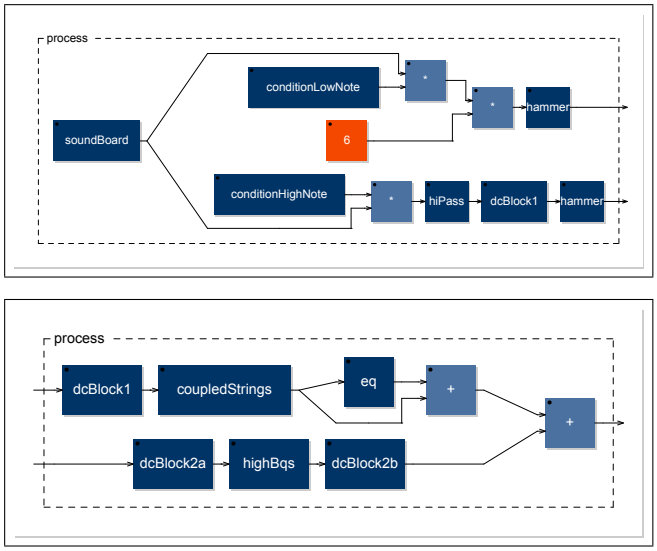

Figure 6: *Commuted piano algorithm drawn by FAUST using faust2svg. The upper figure is the beginning of the model and the lower figure the end.*

E. Maestre worked during his PhD on modeling the instrumental gesture for the violin [20] at the MTG.[6] With his help, it was possible to modify the algorithm of the bowed instrument from the *STK* in order to make it compatible with gesture data. The following changes were performed on the model:

- the *ADSR* used to control the bow velocity was removed;
- a "force" parameter that controls the slope of the bow table was added;
- a switch was added at the output of the bow table;

[6]Music Technology Group, University Pompeu Fabra, Barcelona (Spain).

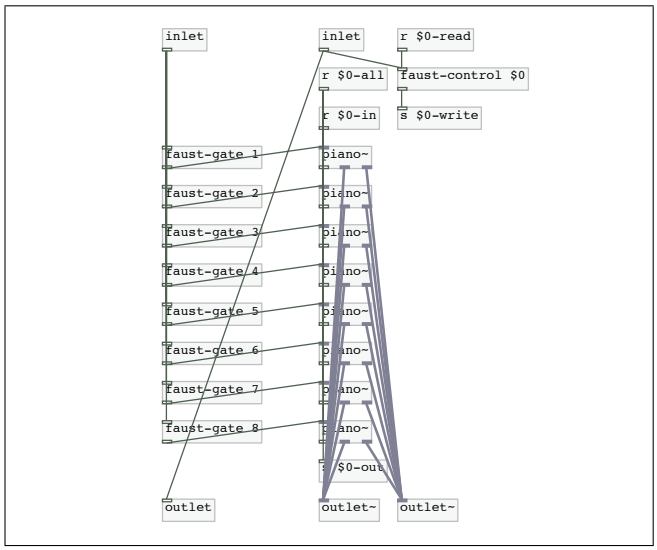

Figure 7: *Synthesis part of the Pure Data polyphonic sub-patch generated with faust2pd from "piano.dsp". In the current case, a height voices polyphony synthesizer is implemented so piano∼.dsp is called height times.*

- we created a four-string violin where it is possible to modify the value of the parameters of each string independently;
- the simple body filter was replaced by a bank of biquad filters that impart a violin body response on the generated sound;
- an improved reflection filter also based on a bank of biquads is used.

The FAUST code was used to create a *Pure Data* plug-in. The gesture data for each physical parameter (note frequencies, bow position, bow velocity, bow force, and number of the string to be used) of the violin model were placed in separated text files that can be used in a *Pd* patch. In the example shown in Figure 8, the values are changed every 4.167 milliseconds. The gesture dataset used plays a traditional Spanish song called Muiñeira.

8. OPTIMIZATION AND PERFORMANCE

8.1. File size

Digital signal processing algorithms can be expressed very compactly in FAUST. The reduction in code size over C++ or even matlab implementations is most of the time very significant. Thereby, we tried to make the *FAUST-STK* algorithms as concise and readable as possible.

It is difficult to compare the STK C++ and FAUST source, because most of the physical models in the *Synthesis ToolKit* were implemented using several functions spread-out among different files. Moreover, these functions may contain information not related to the algorithm itself.

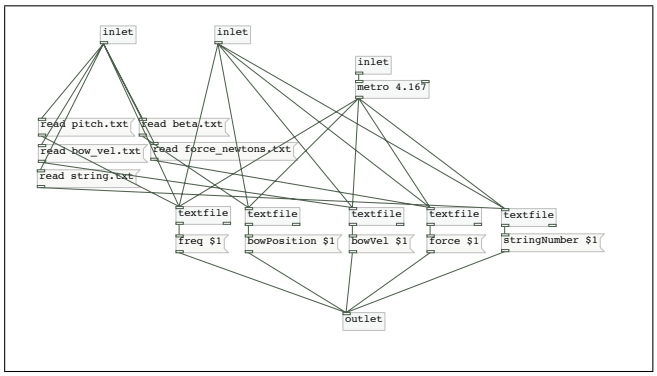

Figure 8: *Pure Data sub-patch used to send the gesture data for Muiñeira in the FAUST generated plug-in.*

We nevertheless carried out a source-size comparison as a rough guide, and the results are given in Table 1. We took into account, in both FAUST and C++, the implementation of the algorithm itself, and the code concerning parameter-handling. While the precision of the comparison is open to debate, we see clearly that the FAUST code is generally more compact than the C++.

8.2. CPU load

The FAUST compiler optimizes the efficiency of its generated C++ code. Thus, we tried to compare for some models the CPU load between *Pure Data* plug-ins created using the *stk2pd*[7] program with Pd plug-ins generated by FAUST using the *Pure Data* architecture file.

In both cases, Pd plug-ins were compiled in 32 bits and the signal processing is scalar. Tests were carried out on a MacBook Pro with the following configuration:

- processor: 2.2 GHz Intel Core 2 Duo;
- RAM: 2GBytes DDR2.

Results of this comparison can be seen in Table 2.

As the original STK C++ code is already very well written and optimized, this comparison shows how efficient the FAUST compiler is at generating highly optimized C++ codes.

9. CONCLUSIONS

Even if the primary goal of the *FAUST-STK* is the use of its physical models in a musical manner, it was also built to be a pedagogical tool. Indeed, because of its transparency and efficiency, the FAUST programming language is particularly suitable for teaching digital audio signal processing. Therefore, a clean and well commented FAUST program is arguably the best way to document the implemented instruments, especially in view of the automatic

[7] *stk2pd* is a program that was developed at Stanford's CCRMA by M. Gurevich and C. Chafe. It converts any C++ code from the *STK* into a plug-in for *Pure Data* [21].

FAUST file name	C++ code nb of declarations	FAUST code nb of declarations	Size gain for nb of lines	C++ code nb of lines	FAUST code nb of lines	Size gain for nb of lines
blowBottle.dsp	74	30	**59.5%**	237	54	**77.2%**
blowHole.dsp	131	66	**49.6%**	373	104	**72.1 %**
bowed.dsp	92	45	**51.1%**	274	69	**74.8%**
brass.dsp	90	36	**60%**	272	63	**76.8%**
clarinet.dsp	78	35	**55.1%**	255	60	**76.5%**
flutestk.dsp	109	43	**60.6%**	309	70	**77.3%**
modalBar.dsp *	63	37	**42.3%**	217	78	**64%**
saxophony.dsp	98	42	**57.1%**	308	69	**77.6%**
sitar.dsp	57	25	**56.1%**	193	42	**78.2%**
bars *	164	35	**78.7%**	396	70	**82.3%**
voiceForm.dsp *	121	65	**46.3%**	325	109	**66.5%**
piano.dsp *	292	158	**45.9%**	750	246	**67.2%**

Table 1: *Comparison of the code size of the STK object's C++ code with the FAUST code from FAUST-STK. The number of declarations was calculated in both cases by counting the number of semicolons in the code. In the case of the instruments where the file name is followed by the * sign, parameters data-bases were not taken into account.*

FAUST file name	STK	FAUST	Difference
blowBottle.dsp	3,23	2,49	22,91
blowHole.dsp	2,70	1,75	35,19
bowed.dsp	2,78	2,28	17,99
brass.dsp	10,15	2,01	80,20
clarinet.dsp	2,26	1,19	47,35
flutestk.dsp	2,16	1,13	47,69
saxophony.dsp	2,38	1,47	38,24
sitar.dsp	1,59	1,11	30,19
tibetanBowl.dsp	5,74	2,87	50

Table 2: *Comparison of the performance of Pure Data plug-ins using the STK C++ code with their FAUST generated equivalent. Values in the "STK" and "FAUST" columns are CPU loads in percents. The "difference" column give the gain of efficiency in percents.*

block-diagram facility. FAUST also has the advantage of being committed to a stable computational specification, unlike C++ in which the meanings of "long" and "short" may change over time, for example, or even across computing platforms.

With its continually growing user community, FAUST is becoming a high quality tool for the implementation of audio digital signal processing algorithms. The number of filters, effects and sound synthesizers available in FAUST is constantly increasing. The combined forces of JACK[8] and FAUST recently upgraded by the possibility to control the generated programs with the OSC[9] communication standard constitute a high efficiency work platform whose limits are only constrained by one's imagination.

[8]JACK Audio Connection Kit: `http://jackaudio.org`.

[9]Open Source Control is a content format for messaging among computers, sound synthesizers and other multimedia devices.

10. REFERENCES

[1] P. Cook, "The synthesis toolkit (stk)," in *Proceedings of the International Computer Music Conference (ICMC)*, Beijing, China, Oct., 1999, pp. 299–304.

[2] D. Jaffe, N. Porcaro, G. Scandalis, J. Smith, and T. Stilson, "Synthbuilder: a graphical real-time synthesis, processing and performance system," in *Proceedings of the International Computer Music Conference (ICMC)*, Banff, Canada, 1995, pp. 61–64.

[3] Julius O. Smith III, *Physical Audio Signal Processing*, `https://ccrma.stanford.edu/~jos/pasp/`, Dec. 2010, online book.

[4] J. Smith, "Efficient simulation of the reed-bore and bow-string mechanisms," in *Proceedings of the International Computer Music Conference (ICMC)*, The Hague, Holland, 1986, pp. 275–280.

[5] J. Smith, "Physical modeling using digital waveguides," *Computer Music Journal*, vol. 16, no. 4, pp. 74–91, 1992.

[6] J. L. Kelly and C. C. Lochbaum, "Speech synthesis," *Proc. Fourth Int. Congress on Acoustics, Copenhagen*, pp. 1–4, September 1962, Paper G42.

[7] J. D. Markel and A. H. Gray, *Linear Prediction of Speech*, Springer Verlag, New York, USA, 1976.

[8] K. Karplus and A. Strong, "Digital synthesis of plucked string and drum timbres," *Computer Music Journal*, vol. 7, no. 2, pp. 43–55, 1983.

[9] M-P. Verge, *Aeroacoustics of Confined Jets with Applications to the Physical Modeling of Recorder-Like Instruments*, Ph.D. thesis, Eindhoven University, 1995.

[10] X. Rodet, "One and two mass model oscillations for voice and instruments," in *Proceedings of the International Computer Music Conference (ICMC)*, Banff, Canada, 1995, pp. 207–214.

[11] J. Smith, "Making virtual electric guitars and associated effects using faust," Tech. Rep., CCRMA, Stanford University, 2009.

[12] G. ESSL and P. COOK, "Banded waveguides: Towards physical modeling of bowed bar percussion instruments," in *Proceedings of the International Computer Music Conference (ICMC)*, Beijing, China, Oct., 1999, pp. 321–324.

[13] Julius Smith and Romain Michon, "Nonlinear allpass ladder filters in faust," in *DAFX*, Paris, France, 2011.

[14] Julius O. Smith III and Perry R. Cook, "The second-order digital waveguide oscillator," in *Proc. 1992 Int. Computer Music Conf., San Jose*. 1992, pp. 150–153, Computer Music Association, `http://ccrma.stanford.edu/~jos/wgo/`.

[15] J-M. Adrien, *Representations of Musical Signals*, chapter The missing link: Modal Synthesis, pp. 269–297, G. DePoli, A. Picialli, and C. Roads Eds. MIT Press, Cambridge, MA, 1991.

[16] E. Castro Lopo (de), "libsndile," Available at http://www.mega-nerd.com/libsndfile/, accessed March 01, 2011.

[17] J. Smith and S. Van Duyne, "Commuted piano synthesis," in *Proceedings of the International Computer Music Conference (ICMC)*, Banff, Canada, 1995, pp. 319–326.

[18] S. Sinclair, "Implementing the synthbuilder piano in stk," Tech. Rep., McGill University, Canada, 2006.

[19] A. Graef, "faust2pd: Pd Patch Generator for Faust," Available at http://docs.pure-lang.googlecode.com/hg/faust2pd.html, accessed March 01, 2011.

[20] E. Maestre-Gomez, *Modeling Instrumental Gesture: An Analysis/Synthesis Framework for Violin Bowing*, Ph.D. thesis, University Pompeu Fabra, 2009.

[21] M. Gurevich and C. Chafe, "Stk2pd," Available at https://ccrma.stanford.edu/wiki/Stk2pd, accessed March 01, 2011.