FAUST-STK: A SET OF LINEAR AND NONLINEAR PHYSICAL MODELS FOR THE FAUST PROGRAMING 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 *PureData* program.

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

1. INTRODUCTION

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

The *STK* is 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[2].

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

An important part of our work consisted at improving and simplifying the models from these two sources in order to make them more efficient thanks to the FAUST semantic. All the FAUST codes from the *FAUST-STK* are commented and refer as often as possible to external bibliographical elements. Finally, lots 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 about 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 done. Finally, we'll study the performances of the generated C++ code.

2. WAVEGUIDE MODELS

Waveguide synthesis was created by J. O. Smith during the eighties [**?**]. It is based on a modified version of the Karplus-Strong algorithm [**?**] that makes it possible to model any kind of string, bore or vibrating structures with a network of delay lines and filters. Waveguide instruments appears to be very suitable to be implemented with the FAUST language mainly because of their "stream like" architecture.

A brief overview of the *FAUST-STK* waveguide instruments is done in this section.

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 needed most of the time to write our own filters in order to be able to use the parameters from the *STK* classes and the *SynthBuilder* patches. All these functions were put 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 correspond to the amplitude of the excitation is controlled by an envelope. The excitation is used to feed one or several waveguides that implements 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 **??**, it is possible to see the block diagram of one of the two clarinet models that are implemented in

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

[1] 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/`.

[2] `https://ccrma.stanford.edu/software/stk/`.

[3] Center for Computer Research in Music and Acoustics

the *FAUST-STK*. In that case, an ADSR[4] 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 **??** 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 which is of course impossible.

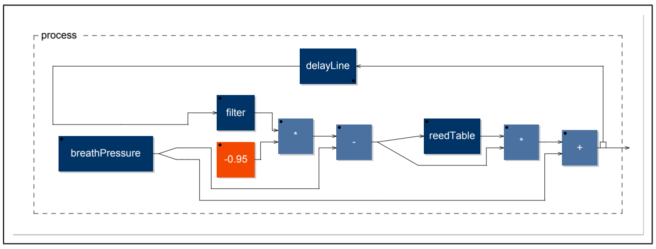

Figure 1: `clarinet.dsp` *algorithm drawn by FAUST using faust2pd.*

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 [**?**]. The other model is showed in figure **??**. It uses two loops and a more sophisticated jet filter (a butterworth filter is used instead of a simple one pole).

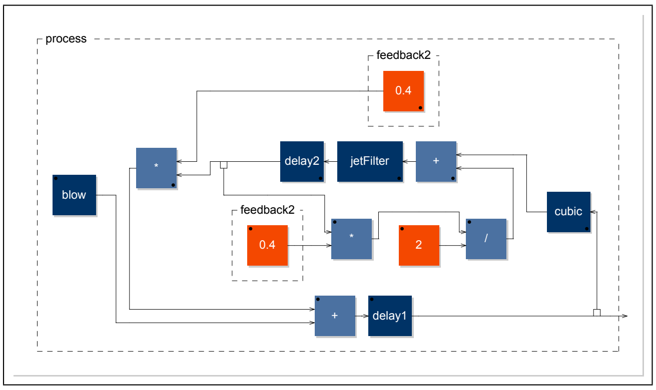

Figure 2: `flute.dsp` *algorithm drawn by FAUST using faust2pd.*

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 [**?**] 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 trombon. Its algorithm can be seen in figure **??**.

[4]Attack - Decay - Sustain - Release.

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

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 [**?**] 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. **??** about nonlinear waveguide models), an acoustic bass, a piano and an harpsichord (Cf. **??** 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 [**?**]) to model the following percussion instruments:

- an iron plaque;
- a wooden plaque;
- a glass plaque;
- 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. The allpass Ladder and Latice filters described in [**?**] makes it possible to create nonlinear behaviors when they are used in waveguide algorithms. The nonlinearities are generated by dynamicaly modulating the filter coefficients at every sample. For the instruments that use this kind of filter in the *FAUST-STK*, the user can decide whether the coefficients are modulated by the incoming signal or by a sine wave. In both cases, a "nonlinearity factor" parameter scale the range of the modulation of the filter coefficients. This parameter can be controlled by an envelope in order to make the nonlinear behavior more natural.

It is necessary to 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 follow:

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

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).

Typically, the nonlinear allpass filter has to be placed just before the feedback of the waveguide loop as showed in figure **??**.

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.

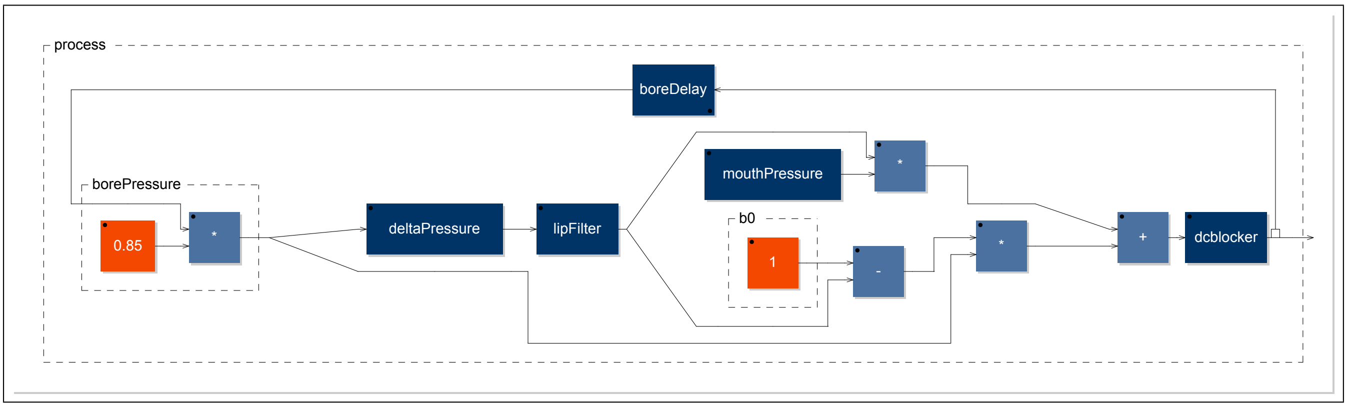

Figure 3: `brass.dsp` *algorithm drawn by FAUST using faust2pd.*

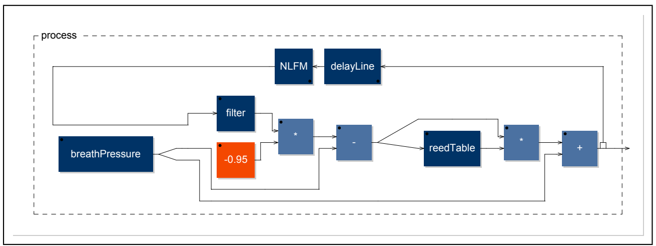

Figure 4: *Modified version of* `clarinet.dsp` *(Cf. figure* **??***) that uses a nonlinear allpass filter in its feedback loop.*

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 discovered by J-M. Adrien in the eighties [**?**] and is very similar to the technique used in **??** as it consists in exciting a filter bank with an impulsion (figure **??**).

Implementing modal synthesis with FAUST was a bit chaleng-ing. Indeed, it requires to handle an important amount of values and to use an excitation signal stored in a wave file. The first problem was solved by using the foreign function primitive in FAUST that allows to use C++ function within a 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.

In order to solve the other problem about importing a wave table in a FAUST object from a wave file, we first tried to use the `libsndfile` library developed by E. de Castro Lopo [**?**] 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 than 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 wave table to excite a bank of 4 bandpass filters that shape the voice formants. The formant values are stored in a C++ function in the same way than described in **??** as a set of center 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 than the one from the *Synthesis ToolKit*.

6. KEYBOARDS

A *SynthBuilder* patch implementing a commuted piano [**?**] has been written in the late nineties at Stanford's CCRMA. This patch was partly ported in 2006 by Stephen Sinclair at McGill University in the *Synthesis ToolKit* [**?**]. A big part of his work consisted in extracting the values for each parameters from the *SynthBuilder* patch to store 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 **??**.

In this piano model, the keyboard is splited in two part which use a different algorithm. Indeed, the tones below *E6* use the commuted waveguide technique while tones above or equal to *E6* use a serie of biquad filters to generate the sound (figure **??**).

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 is not a polyphonic instrument. However, the *faust2pd* program developed by Albert Graef [**?**] makes it possible to automatically produce *PureData* patches that implement polyphonic synthesizers that use FAUST generated PD plug-ins. They can then be controlled via MIDI or OSC directly in PureData.

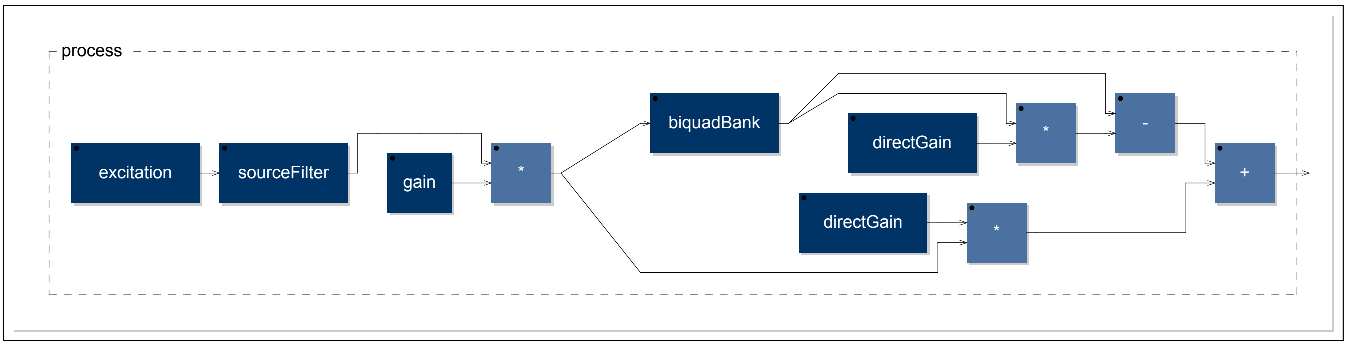

Figure 5: `modalBar.dsp` *algorithm drawn by FAUST using faust2pd.*

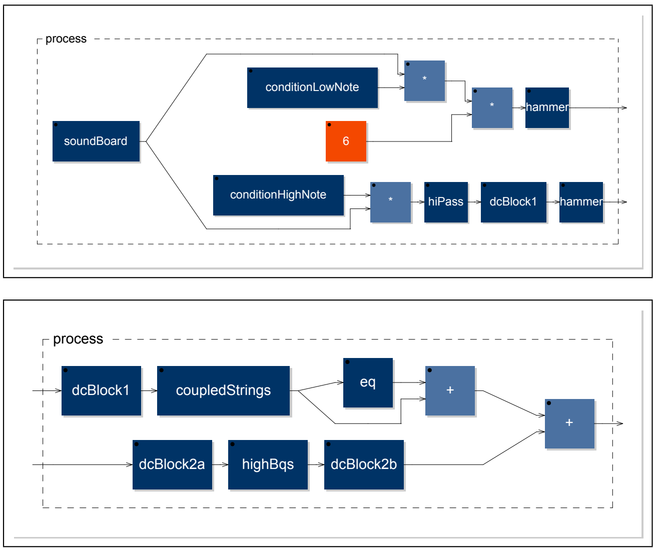

Figure 6: *Commuted piano algorithm drawn by FAUST using faust2svg. The upper figure is the beginning of the model and the lower figure the end.*

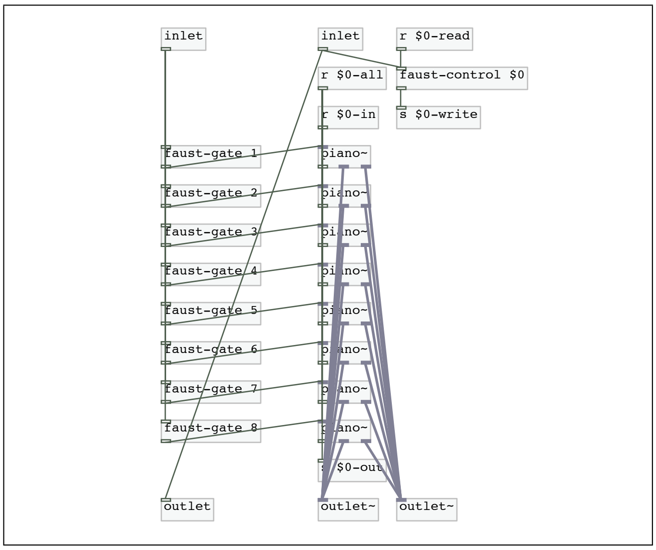

Figure 7: *Synthesis part of the PureData polyphonic sub-patch generated with faust2pd from "piano.dsp". In the current case, a height voices polyphony synthesizer is implemented so piano∼.dsp is called height times.*

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 in function of the note and the amplitude being played.

E. Maestre worked during his phd on modeling the instrumental gesture for the violin [?] at the MTG[5]. With his help, it was possible to modify the algorithm of the bowed instrument from the *STK* in order to make it compatible to gesture data. The following changes were performed on the model:

- the *ADSR* used to control the bow velocity was removed;
- a "force" parameter that control the slope of the bow table was added;
- a switch was added at the output of the bow table;
- we created a four strings 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 apply a violin impulse response on the generated sound
- an improved reflexion filter also based on a bank of biquad is used.

The FAUST code was used to create a *PureData* 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 **??**,

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

the values are changed every 4.167 milliseconds. The gesture data used plays a traditional spanish song called Muiñeira.

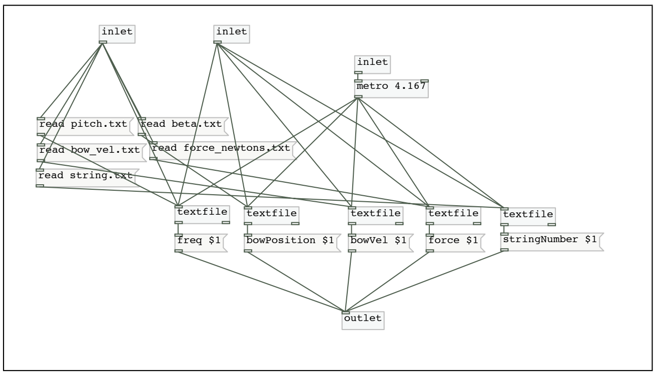

Figure 8: *PureData sub-patch used to send the gesture data of Muiñeira in the FAUST generated plug-in.*

8. OPTIMIZATION AND PERFORMANCES

8.1. File size

Digital signal processing algorithms can be expressed very shortly in FAUST. The gain in term of code size with 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 would be very hard and inaccurate to compare the raw C++ and FAUST codes together as most of the physical models were implemented in the *Synthesis ToolKit* with several spread out functions that are most of the time strewn in different files. Moreover, these functions also contains informations that are not related to the algorithm itself.

Nevertheless, we were able to carry out such a comparison as a rough guide. We took into account in FAUST as in C++ the implementation of the algorithm itself and the line concerning parameters handling.

The result of this comparison can be seen in table **??**. Once again, even if its results are certainly arguable, it shows very well how FAUST is efficient in reducing the code size.

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 *PureData* plug-ins created using the *stk2pd*[6] program with PD plug-ins generated by FAUST using the *PureData* architecture file.

In both cases, PD plug-ins were compiled in 32bits 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 **??**.

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 performances of PureData 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.*

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 transparence and its efficiency, the FAUST programing language is particularly suitable for teaching digital audio signal processing. Therefore, a clean and well commented FAUST code is probably the best way to document the implemented instruments.

With its continually growing users 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[7] and FAUST recently upgraded by the possibility to control the generated programs with the OSC[8] communication standard constitue a high efficiency work platform whose limits are only constrained by the user imagination.

10. ACKNOWLEDGMENTS

This work was carried out in the frame of the ASTREE[9] project supported by the Agence Nationale de Recherche (ANR-08-CORD-003).

[6] *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 *PureData* [**?**].

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

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

[9] Analyse et Synthèse de Traitement en Temps Réel.

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.*