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 all-pass 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 is set of virtual musical instruments programmed in the FAUST programming 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 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 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.

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 all-pass 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. It can be viewed as descendent of either the Kelley-Lochbaum vocal-tract model or Karplus-Strong “digitar” algorithm. 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.
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 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.

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

---

5 Attack - Decay - Sustain - Release.
of filter in the \textit{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:

$$DL = (SR/F) - FO \times NF$$

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.

Finally, it is interesting to mention that we were able to implement a frequency modulation synthesizer in the \textit{FAUST-STK} by using this kind of filter on a sine wave signal. A related result is reported in [13].
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.

E. Maestre worked during his PhD on modeling the instrumental gesture for the violin [20] at the MTG. 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;

Music Technology Group, University Pompeu Fabra, Barcelona (Spain).
• 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. Therefore, 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.

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 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 \([\text{21}]\).
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.

<table>
<thead>
<tr>
<th>FAUST file name</th>
<th>C++ code nb of declarations</th>
<th>FAUST code nb of declarations</th>
<th>Size gain for C++ code nb of lines</th>
<th>FAUST code nb of lines</th>
<th>Size gain for FAUST code nb of lines</th>
</tr>
</thead>
<tbody>
<tr>
<td>blowBottle.dsp</td>
<td>74</td>
<td>30</td>
<td>59.5%</td>
<td>237</td>
<td>54</td>
</tr>
<tr>
<td>blowHole.dsp</td>
<td>131</td>
<td>66</td>
<td>49.6%</td>
<td>373</td>
<td>104</td>
</tr>
<tr>
<td>bowed.dsp</td>
<td>92</td>
<td>45</td>
<td>51.1%</td>
<td>274</td>
<td>69</td>
</tr>
<tr>
<td>brass.dsp</td>
<td>90</td>
<td>36</td>
<td>60%</td>
<td>272</td>
<td>63</td>
</tr>
<tr>
<td>clarinet.dsp</td>
<td>78</td>
<td>35</td>
<td>55.1%</td>
<td>255</td>
<td>60</td>
</tr>
<tr>
<td>flutesk.dsp</td>
<td>109</td>
<td>43</td>
<td>60.6%</td>
<td>309</td>
<td>70</td>
</tr>
<tr>
<td>modalBar.dsp</td>
<td>63</td>
<td>37</td>
<td>42.3%</td>
<td>217</td>
<td>78</td>
</tr>
<tr>
<td>saxophony.dsp</td>
<td>98</td>
<td>42</td>
<td>57.1%</td>
<td>308</td>
<td>69</td>
</tr>
<tr>
<td>sitar.dsp</td>
<td>57</td>
<td>25</td>
<td>56.1%</td>
<td>193</td>
<td>42</td>
</tr>
<tr>
<td>bars *</td>
<td>164</td>
<td>35</td>
<td>78.7%</td>
<td>396</td>
<td>70</td>
</tr>
<tr>
<td>voiceForm.dsp</td>
<td>121</td>
<td>65</td>
<td>46.3%</td>
<td>325</td>
<td>109</td>
</tr>
<tr>
<td>piano.dsp *</td>
<td>292</td>
<td>158</td>
<td>45.9%</td>
<td>750</td>
<td>246</td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>FAUST file name</th>
<th>STK</th>
<th>FAUST</th>
<th>Difference</th>
</tr>
</thead>
<tbody>
<tr>
<td>blowBottle.dsp</td>
<td>3.23</td>
<td>2.49</td>
<td>22.91</td>
</tr>
<tr>
<td>blowHole.dsp</td>
<td>2.70</td>
<td>1.75</td>
<td>35.19</td>
</tr>
<tr>
<td>bowed.dsp</td>
<td>2.78</td>
<td>2.28</td>
<td>17.99</td>
</tr>
<tr>
<td>brass.dsp</td>
<td>10.15</td>
<td>2.01</td>
<td>80.20</td>
</tr>
<tr>
<td>clarinet.dsp</td>
<td>2.26</td>
<td>1.19</td>
<td>47.35</td>
</tr>
<tr>
<td>flutesk.dsp</td>
<td>2.16</td>
<td>1.13</td>
<td>47.69</td>
</tr>
<tr>
<td>saxophony.dsp</td>
<td>2.38</td>
<td>1.47</td>
<td>38.24</td>
</tr>
<tr>
<td>sitar.dsp</td>
<td>1.59</td>
<td>1.11</td>
<td>30.19</td>
</tr>
<tr>
<td>tibetanBowl.dsp</td>
<td>5.74</td>
<td>2.87</td>
<td>50</td>
</tr>
</tbody>
</table>

10. REFERENCES


