THE FAUST PHYSICAL MODELING LIBRARY: A MODULAR PLAYGROUND FOR THE DIGITAL LUTHIER

Romain Michon¹,², Julius O. Smith³, Chris Chafe³, Ge Wang⁴, and Matthew Wright⁴

¹Center for Computer Research in Music and Acoustics (CCRMA), Stanford University, USA
²GRAME – Centre National de Création Musicale, Lyon, France
³rmichon@ccrma.stanford.edu

ABSTRACT

This paper introduces the FAUST Physical Modeling Library, an environment to create physical models of musical instruments in a modular way in the FAUST programming language. Low and high level elements can be combined to implement existing or completely novel instruments. Various examples of physical models are provided. The combined use of mesh2faust, a tool to generate FAUST physical models from 3D drawings, and of the FAUST Physical Modeling Library is also demonstrated through the implementation of a marimba physical model.

1. INTRODUCTION

The FAUST programming language has been used extensively in the framework of waveguide [1] and modal [2] physical modeling of musical instruments. Implementing such algorithms in FAUST is eased by the wide range of functions available in the FAUST DSP libraries [3] and by the block-diagram/signal-oriented syntax of the language.

Julius Smith was the first to exploit FAUST’s potential in this context by implementing a virtual electric guitar and its related effects [4]. This model was used later as the basis for GeoSheed [5], a mobile music app where a touchscreen interface controls a FAUST electric guitar physical model. Smith et al. also implemented a waveguide mesh [5] that can be used to model nonlinear percussion instruments.

Similarly, the FAUST-STK [6] which is a collection of physical models implemented in FAUST and based on some of the algorithms of the “original” STK [7] was implemented at the same period. Even though FAUST-STK models share many functions through the instrument.lib FAUST library, they are still implemented as standalone objects, and virtual instrument parts can’t be reused.

The idea of creating a physical modeling environment where musical instrument “parts” (e.g., embouchures, tubes, strings, bodies, etc.) can be assembled modularly has been explored in various projects such as Modalys [8] and CORDIS-ANIMA [9]. More recently, Synth-A-Modeler [10] was introduced. It offers a graphical environment for high-level physical modeling/“digital lutherie” and uses FAUST internally to implement its models.

FAUST itself makes it rather difficult to combine functions implementing instrument parts/elements, mostly because the constituting elements of most musical instruments are usually coupled to each other, which requires the use of bi-directional connections, a feature that FAUST doesn’t provide “out-of-the-box.”

In this paper, we introduce the FAUST Physical Modeling Library (FPML): physmodels.lib[11] an environment to create physical models of musical instruments in a modular way. Low and high level elements can be combined to implement existing or completely novel instruments.

First, we introduce a new system implementing a bidirectional block-diagram algebra in FAUST. Next, we demonstrate how it can be used to assemble virtual instrument parts: the example of the violin (bowed string) is studied in this context. We then show how mesh2faust [11] can be used to generate custom instrument parts to be used with the FAUST Physical Modeling Library. Finally, future directions for this work are presented in the conclusion.

2. BIDIRECTIONAL BLOCK-DIAGRAM ALGEBRA

In the physical/acoustical world, waves propagate in multiple dimensions and directions across the different parts of musical instruments. Thus, coupling among the constituting elements of an instrument sometimes plays an important role in its general acoustical behavior [12]. Coupling might be limited and even neglected when designing models of some instruments such as the electric guitar where energy is transmitted from the string to the pickup in a predominantly unidirectional way. Similarly, the clarinet, where the frequency of vibration of the reed(s) is determined by the length of the connected tube.

Coupling can be implemented by creating bidirectional connections between the different elements of a model. The block-diagram algebra of FAUST allows us to connect blocks in a unidirectional way (from left to right) and feedback signals (from right to left) can be implemented using the tilde (\(\sim\)) diagram composition operation:

\[
\text{process} = (A : B) \sim (C : D);
\]

where A, B, C, and D are hypothetical functions with a single argument and a single output. The resulting FAUST-generated block diagram can be seen in Figure 1.

In this case, the D/A and the C/B couples can be seen as bidirectional blocks/functions that could implement some musical instrument part. However, the FAUST semantics doesn’t allow them to be specified as such from the code (A, B, C, and D must be declared all together as part of the same expression), preventing the implementation of “bidirectional functions.” Since this feature is required to create a library of physical modeling elements, we had to implement it.

The FAUST Physical Modeling Library is now part of the FAUST distribution.
Bidirectional blocks in the FAUST Physical Modeling Library all have three inputs and outputs. Thus, an empty block can be expressed as:

```
emptyBlock = _, _, _;
```

The first input and output correspond to left-going waves (e.g., \( C \) and \( D \) in Figure 1), the second input and output to right-going waves (e.g., \( A \) and \( B \) in Figure 1), and the third input and output can be used to carry any signal to the end of the algorithm. As we’ll see in §3 this can be useful when picking up the sound at the middle of a virtual string, for example.

The chain primitive (part of physmodels.lib) connects bidirectional blocks to each other. For example, an open waveguide (no terminations) can be expressed as:

```
waveguide(nMax,n) = par(i,2,de.fdelay4(nMax,n)), _;
```

where \( nMax \) is the maximum length of the waveguide in samples and \( n \) its current length, could be connected to our emptyBlock:

```
foo = chain(emptyBlock : waveguide(256,n) : emptyBlock);
```

Note the use of \texttt{fdelay4} in \texttt{waveguide}, which is a fourth order fractional delay line \[13\].

The \textsc{faust} compiler is not able yet to lay out the block diagram corresponding to the previous expression in an organized bidirectional way. However, a “hand-made” diagram can be seen in Figure 2.

The placement of elements in a \texttt{chain} matters and corresponds to their order in the physical/acoustical world. For example, for a set of hypothetical functions implementing the different parts of a violin, we could write:

```
vigil = chain(nuts : string : bridge : body);
```

A signal \( x \) can be fed anywhere in a \texttt{chain} by using the \texttt{in}(\( x \)) primitive. Similarly, left and right-going waves can be summed and extracted from a chain (via the third output) using the \texttt{out} primitive (see Code Listing 1).

Finally, a chain of blocks \( A \) can be “terminated” using \texttt{endChain}(\( A \)) which essentially removes the three inputs and the first two outputs of \( A \).

Assembling a simple waveguide string model with “ideal” rigid terminations is simple using this framework:

```
string(length,pluckPosition,excitation) = endChain(wg)
with{
  maxStringLength = 3; // in meters
  lengthTuning = 0.08; // adjusted by hand
  tunedLength = length-lengthTuning;
  // upper string segment length
  nUp = tunedLength*pluckPosition;
  // lower string segment length
  nDown = tunedLength*(1-pluckPosition);
  // left phase inversion
  lTerm = lTermination(*(-1),emptyBlock);
  // right phase inversion
  rTerm = rTermination(emptyBlock,*(-1));
  stringSegment(maxLength,length) = waveguide(nMax,n)
  with{
    nMax = maxLength : l2s;
    // length in samples
    n = length : l2s/2;
  };
  wg = chain( // waveguide chain
    lTerm : stringSegment(maxStringLength,nUp) : in(excitation) : out :
    stringSegment(maxStringLength,nDown) : rTerm
  );
};
```

Listing 1: “Ideal” string model with rigid terminations.

In this case, since \texttt{in} and \texttt{out} are placed next to each other in the \texttt{chain}, the position of excitation and the position of the pickup are the same as well.

### 3. Assembling High Level Parts: Violin Example

The \textsc{faust} Physical Modeling Library contains a wide range of ready-to-use instrument parts and pre-assembled models. An overview of the content of the library is provided in the \textsc{faust} libraries documentation \[13\]. Detailing the implementation of each function of the library would be interesting, however this section focuses on one of its models: \texttt{violinModel} (see Code Listing 2) which implements a simple bowed string connected to a body through a bridge.

```
vigilinModel(stringLength,bowPressure, bowVelocity,bowPosition) = endChain(modelChain)
with{
  stringTuning = 0.08;
```

Figure 1: Bidirectional Construction in \textsc{faust} Using the Tilde Diagram Composition Operation.

```
Proceedings of the 1st International Faust Conference (IFC-18), Mainz, Germany, July 17–18, 2018

Figure 2: Bidirectional construction in FAUST using the chain primitive.

Figure 3: lTermination(A,B) and rTermination(B,C) in the FAUST Physical Modeling Library.

Listing 2: violinModel: a simple violin physical model from the FAUST Physical Modeling Library.

```
stringL = stringLength-stringTuning;
modelChain = chain(
  violinNuts :
    violinBowedString(stringL,bowPressure,
      bowVelocity,bowPosition) :
    violinBridge : violinBody : out
)
);
```

violinModel assembles various high-level functions implementing violin parts. violinNuts is a termination applying a light low-pass filter on the reflected signal. violinBowedString is made out of two open string segments allowing us to choose the bowing position. The bow nonlinearity is implemented using a table. violinBridge implements the “right termination” as well as the reflectance and the transmittance filters [1]. Finally, violinBody is a simple violin body modal model.

In addition to its various models and parts, the FAUST Physical Modeling Library also implements a series of ready-to-use models hosting their own user interfaces [13]. The corresponding functions end with the _ui suffix. For example:

```
process = pm.violin_ui;
```

is a complete FAUST program adding a simple user interface to control the violin model presented in Code Listing 2.

While [...]_ui functions associate continuous UI elements (e.g., knobs, sliders, etc.) to the parameters of a model, functions ending with the _ui_midi prefix automatically link the FAUST parameters (i.e., frequency, gain, and note-on/off) to MIDI using envelope generators. Thus, such functions are ready to be controlled by a MIDI keyboard.

Nonlinear behaviors play an important role in some instruments (e.g., gongs, cymbals, etc.). While waveguide models and modal synthesis are naturally linear, nonlinearities can be introduced using nonlinear allpass ladder filters [9]. allpassNL implements such a filter in the FAUST Physical Modeling Library.

Some of the physical models of the FAUST-STK [6] were ported to the FAUST Physical Modeling Library and are available through various functions summarized in Table 1.

<table>
<thead>
<tr>
<th>FAUST-STK Model</th>
<th>Corresponding FPML Functions</th>
</tr>
</thead>
<tbody>
<tr>
<td>bowed.dsp</td>
<td>violin/violin_ui/</td>
</tr>
<tr>
<td></td>
<td>violin_ui_midi</td>
</tr>
<tr>
<td>brass.dsp</td>
<td>brassModel/brass_ui/</td>
</tr>
<tr>
<td></td>
<td>brass_ui_midi</td>
</tr>
<tr>
<td>clarinet.dsp</td>
<td>clarinetModel/</td>
</tr>
<tr>
<td></td>
<td>clarinet_ui/</td>
</tr>
<tr>
<td></td>
<td>clarinet_ui_midi</td>
</tr>
<tr>
<td>fluteStk.dsp</td>
<td>fluteModel/flute_ui/</td>
</tr>
<tr>
<td></td>
<td>flute_ui_midi</td>
</tr>
</tbody>
</table>

Table 1: FAUST-STK models and their corresponding function re-implementations in the FAUST Physical Modeling Library.

4. EXAMPLE: MARIMBA PHYSICAL MODEL USING FPML AND MESH2FAUST

This section briefly demonstrates how a simple marimba physical model can be made using mesh2faust [11] and FPML. The idea is to use a 3D CAD model of a marimba bar, generate the corresponding modal model, and then connect it to a tube model implemented in FPML.

A simple marimba bar 3D model was made by extruding a marimba bar cross section (see Figure 4) using the Inkscape to OpenSCAD tool provided with mesh2faust [11]. The resulting CAD model was then turned into a volumetric mesh by importing it to MeshLab [12] and by uniformly re-sampling it to have approximately 4500 vertices. The mesh produced during this step (marimbaBar.obj in the following code listing) was processed by mesh2faust using the following command:

```
mesh2faust --infile marimbaBar.obj
  --nsynthmodes 50 --nfemmodes 200
  --maxmode 15000
  --expos 2831 3208 3624 3975 4403
  --freqcontrol --material 1.3E9 0.33 720
  --name marimbaBarModel
```

An extended version of this example with more technical details is also available here: https://ccrma.stanford.edu/~rmichon/faustTutorials/#making-custom-elements-using-mesh2faust
http://www.meshlab.net/
Figure 4: Marimba bar model – steps from a 2D drawing to a FAUST modal model.
The materials parameters are that of rosewood which is traditionally used to make marimba bars. The number of modes is limited to 50 and various excitation positions were selected to be uniformly spaced across the horizontal axis of the bar.

The `--freqcontrol` switch activates “frequency control mode,” to be able to transpose the modes of the generated model relative to the fundamental frequency, making the model more generic.

A simple marimba resonator was assembled using FPML and is presented in Code Listing 3. It is made out of an open tube where two simple lowpass filters placed at its extremities are used to model the wave reflections. The model is excited on one side of the tube and sound is picked-up on the other side.

```
marimbaResTube(tubeLength,excitation) =
  endChain(tubeChain)
with{
  lengthTuning = 0.04;
  tunedLength = tubeLength-lengthTuning;
  absorption = 0.99;
  lowpassPole = 0.95;
  endTubeReflection = si.smooth(lowpassPole)*absorption;
  tubeChain = chain(
    in(excitation) : terminations(endTubeReflection,
      openTube(maxLength,tunedLength),
      endTubeReflection) : out
  );
}
```

Listing 3: Simple marimba resonator tube implemented with FPML.

Code Listing 4 demonstrates how the marimba bar model generated with mesh2faust (marimbaBarModel) can be simply connected to the marimba resonator. A unidirectional connection can be used in this case since waves are only transmitted from the bar to the resonator.

```
marimbaBarModel(freq,exPos) =
  marimbaResTube(resTubeLength)
with{
  resTubeLength = freq : f21;
  maxT60 = 0.1; T60Decay = 1; T60Slope = 5;
}
```

Listing 4: Simple marimba physical model.

This model is now part of the FAUST Physical Modeling Library. More examples of models created using this technique can be found on-line.

5. CONCLUSIONS

The FAUST Physical Modeling Library is far from being exhaustive and many models and instruments could be added to it. We believe that mesh2faust will help enlarge the set of functions available in this system.

The framework presented in §4 allows us to assemble the different parts of instrument models in a simple way by introducing a bidirectional block diagram algebra to FAUST. While it provides a high level approach to physical modeling, FAUST is not able to generate the corresponding block diagram in a structured way. This would be a nice feature to add.

Similarly, we would like to extend the idea of being able to make multi-dimensional block diagrams in FAUST by adding new primitives to the language.

6. REFERENCES


