

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

Romain Michon, Julius Smith, Chris Chafe, Ge Wang, Matthew Wright

▶ **To cite this version:**

Romain Michon, Julius Smith, Chris Chafe, Ge Wang, Matthew Wright. THE FAUST PHYSICAL MODELING LIBRARY: A MODULAR PLAYGROUND FOR THE DIGITAL LUTHIER. International Faust Conference, 2018, Mainz, Germany. hal-02158706

HAL Id: hal-02158706
https://hal.science/hal-02158706

Submitted on 19 Jun 2019

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

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

Romain Michon[1,2], Julius O. Smith[1], Chris Chafe[1], Ge Wang[1], and Matthew Wright[1]

[1]Center for Computer Research in Music and Acoustics (CCRMA), Stanford University, USA
[2]GRAME – Centre National de Création Musicale, Lyon, France
rmichon@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 GeoShred,[1] 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`,[2] 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 [4]. On the other hand, coupling is crucial for other types of instruments such as woodwinds (e.g., 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 (~) diagram composition operation:

```
process = (A : B) ~ (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.

[1]`http://www.moforte.com/` All the URLs presented in this paper were verified on March 12, 2018.

[2]The FAUST Physical Modeling Library is now part of the FAUST distribution.

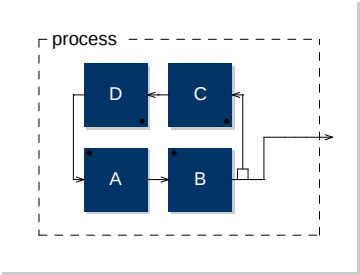


Figure 1: Bidirectional Construction in FAUST Using the Tilde Diagram Composition Operation.

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 `fdelay4` in `waveguide`, which is a fourth order fractional delay line [13].

The 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 `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:

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

The main limitation of this system is that it introduces a one sample delay in both directions for each block in the `chain` due to the internal use of ~ [14]. This has to be taken into account when implementing certain types of elements such as a string or a tube.

Terminations can be added on both sides of a chain using `lTermination(A,B)` for a left-side termination and `rTerminations(B,C)` for a right-side termination where `B` can be any bidirectional block, including a `chain`, and `A` and `C` are functions that can be put between left and right-going signals (see Figure 3).

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

Finally, a chain of blocks `A` can be "terminated" using `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 `in` and `out` are placed next to each other in the `chain`, the position of excitation and the position of the pickup are the same as well.

3. ASSEMBLING HIGH LEVEL PARTS: VIOLIN EXAMPLE

The 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 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: `violinModel` (see Code Listing 2) which implements a simple bowed string connected to a body through a bridge.

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

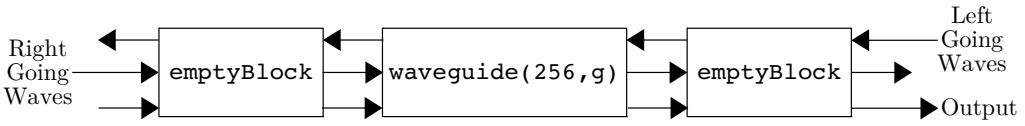


Figure 2: Bidirectional construction in FAUST using the `chain` primitive.

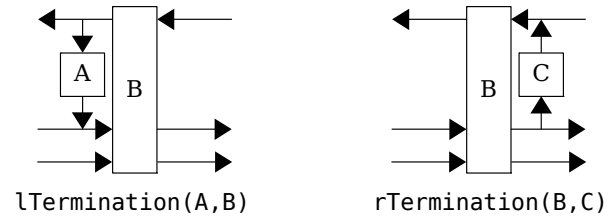


Figure 3: `lTermination(A,B)` and `rTermination(B,C)` in the FAUST Physical Modeling Library.

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

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

`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 [5]. `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.

FAUST-STK Model	Corresponding FPML Functions
`bowed.dsp`	`violin` / `violin_ui` / `violin_ui_MIDI`
`brass.dsp`	`brassModel` / `brass_ui` / `brass_ui_MIDI`
`clarinet.dsp`	`clarinetModel` / `clarinet_ui` / `clarinet_ui_MIDI`
`fluteStk.dsp`	`fluteModel` / `flute_ui` / `flute_ui_MIDI`

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.[3] 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[4] 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
```

[3] 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`

[4] `http://www.meshlab.net/`

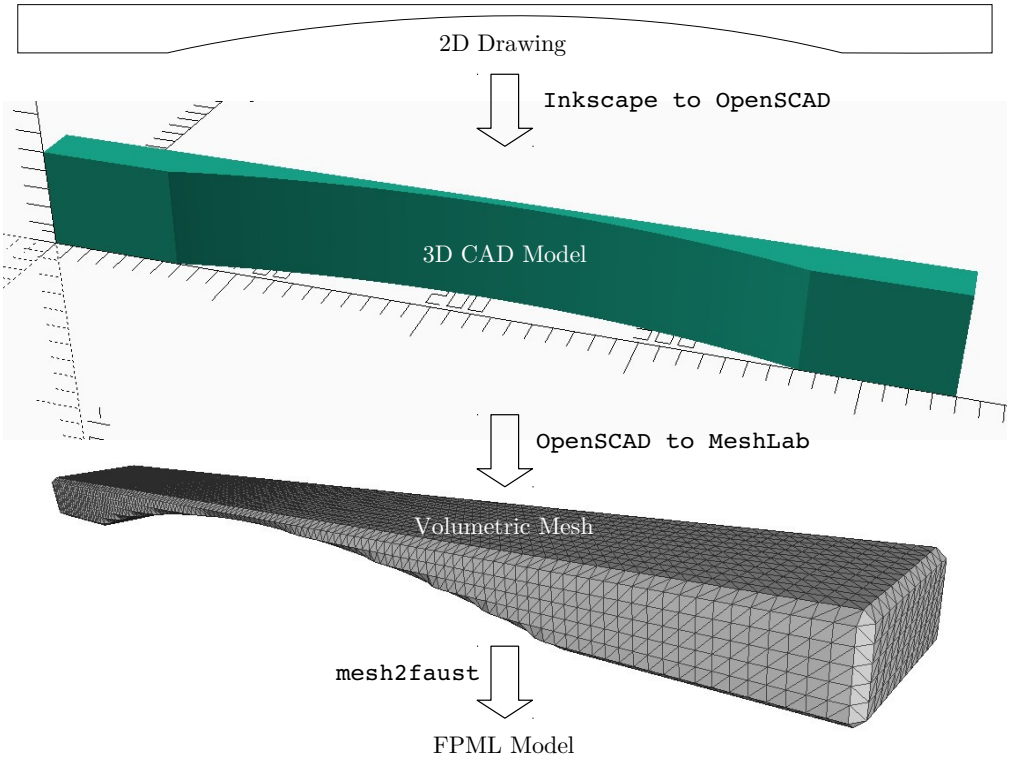


Figure 4: Marimba bar model – steps from a 2D drawing to a FAUST modal model.

The material 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.

```
marimbaModel(freq,exPos) =
  marimbaBarModel(freq,exPos,maxT60,
      T60Decay,T60Slope) :
  marimbaResTube(resTubeLength)
with{
  resTubeLength = freq : f2l;
  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]

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 §2 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.

[5] FAUST Physical Modeling Toolkit Webpage: `https://ccrma.stanford.edu/~rmichon/pmFaust/`.

6. REFERENCES

[1] Julius Orion Smith, *Physical Audio Signal Processing for Virtual Musical Instruments and Digital Audio Effects*, W3K Publishing, 2010.

[2] Jean-Marie Adrien, "The missing link: Modal synthesis," in *Representations of Musical Signals*, chapter The Missing Link: Modal Synthesis, pp. 269–298. MIT Press, Cambridge, USA, 1991.

[3] Romain Michon, Julius Smith, and Yann Orlarey, "New signal processing libraries for Faust," in *Proceedings of the Linux Audio Conference (LAC-17)*, Saint-Etienne, France, May 2017, Paper accepted to the conference but not published yet.

[4] Julius Orion Smith, "Virtual electric guitars and effects using Faust and Octave," in *Proceedings of the Linux Audio Conference (LAC-08)*, KHM, Cologne, Germany, 2008, pp. 123–127.

[5] Julius Orion Smith and Romain Michon, "Nonlinear allpass ladder filters in Faust," in *Proceedings of the 14th International Conference on Digital Audio Effects*, Paris, France, September 2011, IRCAM.

[6] Romain Michon and Julius O. Smith, "Faust-STK: a set of linear and nonlinear physical models for the Faust programming language," in *Proceedings of the 14th International Conference on Digital Audio Effects (DAFx-11)*, Paris, France, September 2011.

[7] Perry Cook and Gary Scavone, "The Synthesis Toolkit (stk)," in *Proceedings of the International Computer Music Conference (ICMC-99)*, Beijing, China, 1999.

[8] René Caussé, Joel Bensoam, and Nicholas Ellis, "Modalys, a physical modeling synthesizer: More than twenty years of researches, developments, and musical uses," *Journal of the Acoustical Society of America*, vol. 130, no. 4, 2011.

[9] Claude Cadoz, Annie Luciani, and Jean Loup Florens, "Cordis-anima: A modeling and simulation system for sound and image synthesis: The general formalism," *Computer Music Journal*, vol. 17, no. 1, pp. 19–29, Spring 1993.

[10] Edgar Berdahl, "An introduction to the Synth-A-Modeler compiler: Modular and open-source sound synthesis using physical models," in *Proceedings of the Linux Audio Conference (LAC-12)*, Stanford, USA, May 2012.

[11] Romain Michon, Sara R. Martin, and Julius O. Smith, "Mesh2Faust: a modal physical model generator for the Faust programming language – application to bell modeling," in *Proceedings of the International Computer Music Conference (ICMC-17)*, Shanghai, China, October 2017.

[12] Neville H. Fletcher and Thomas D. Rossing, *The Physics of Musical Instruments, 2nd Edition*, Springer Verlag, 1998.

[13] "Faust libraries documentation," On-line, `http://faust.grame.fr/library.html`.

[14] GRAME – Centre National de Création Musicale, Lyon, France, *FAUST Quick Reference*, June 2017.