Proceedings of the 17th Linux Audio Conference (LAC-19), CCRMA, Stanford University, USA, March 23–26, 2019

FORMALIZING MASS-INTERACTION PHYSICAL MODELING IN FAUST

James Leonard and Jérôme Villeneuve

Univ. Grenoble Alpes, CNRS, Grenoble INP*, GIPSA-lab,
38000 Grenoble, France
* Institute of Engineering Univ. Grenoble Alpes
james.leonard@gipsa-lab.grenoble-inp.fr

Romain Michon,[1,2] Yann Orlarey[1] and Stéphane Letz[1]

[1]GRAME-CNCM, Lyon (France)
[2]CCRMA, Stanford University (USA)
michon@grame.fr

ABSTRACT

This paper presents recent work conducted on the integration of mass-interaction physical models in the FAUST programming language. After a brief introduction to mass-interaction networks, FAUST, and previous works on this topic, we present a simple modeling framework, a FAUST code generator and its associated library, allowing to implement 1D mass-interaction models. In addition to the open-source tool itself, this research offers a perspective on formalizing arbitrarily large networks of bidirectional feedback couplings and state-space models in FAUST, through routing patterns. We finish with a set of examples, and discuss future perspectives and challenges.

1. INTRODUCTION

For several decades, physical modeling has been used to synthesize audio by means of simulating the behaviour of vibrating objects. A panoply of methods have been proposed over the years, from lumped discrete models [1], to Waveguides [2], to large scale Finite Difference schemes [3], that have gained in popularity with the increase of computing power. Creating a model of a mechanical instrumental system can be simpler than explicitly formulating the signal that it produces (as sound properties emerge from the physical conditions of the matter) and offers direct means for control and interaction, either by simulating musical gestures or by coupling the user and the virtual object, for instance using haptic technologies [4].

FAUST [5] is a functional programming language for real-time Digital Signal Processing (DSP) with a strong focus on the design of synthesizers, musical instruments, audio effects, etc. The FAUST compiler can be used to "translate" a FAUST program to various non-domain-specific-languages such as C++, C, JAVA, JavaScript, LLVM bit code, WebAssembly, etc. Thanks to a wrapping system, code generated by FAUST can be easily compiled into a variety of objects ranging from audio plug-ins to standalone applications, smartphone apps, web apps, etc.[1] This mechanism also makes it possible to add MIDI, OSC, polyphony, etc. support to any FAUST-generated program.

1.1. Mass-Interaction Physical Models

Pioneered in artistic applications by the CORDIS-ANIMA system [1] at ACROE, mass-interaction physical modeling allows to formulate physical systems in the form of lumped networks, composed of two main components: masses, representing material points in a given space (1D, 2D, 3D) with a given inertial behaviour, and interactions, each representing a specific type of physical coupling (i.e., visco-elastic, collision, non-linear, etc.) between two mass elements. Mass-interaction systems are now used in a variety of contexts (musical & other), partly for the fact that arbitrarily complex virtual objects can be described simply as a construction of elementary physical components. A basic model is shown in Figure 1.

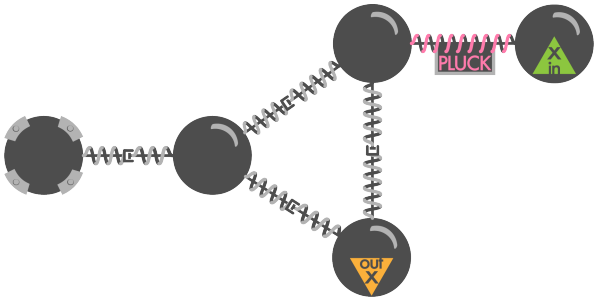

Figure 1: *Topological representation of a mass-interaction model. Here, a fixed point (represented on the left) is connected to a triangle composed of masses and dampened springs. An input module interacts with the top mass through a non-linear pluck interaction.*

Unlike FDTD methods [3], creating physical models with this formalism avoids the need to explicitly define a mathematical model (partial difference equations systems, boundary conditions, etc.) for a given physical structure beforehand. Therefore, it lends itself particularly well to iterative and exploratory design of "physically plausible" virtual objects, grounded in the laws of Newtonian physics but not necessarily limited to the mechanical constraints of the real world.

Mass-interaction physical models can contain anything from a couple of physical elements to tens or hundreds of thousands of them. Assembling and configuring the models element by element can be very time consuming. To this end, user-friendly modeling environments have been proposed, namely GENESIS [6] (and more recently *Synth-A-Modeler* [7]) for 1D audio applications. The former offers high level tools for generating topological structures, and analyzing/tuning physical constructions through modal analysis [8].

1.2. Current State of Physical Modeling in Faust

Various projects have been using FAUST to implement physical models of musical instruments.

The FAUST-STK [9] is a complete re-implementation of the waveguide and modal models of the Synthesis ToolKit (STK) [10]. It also contains various models from the *Soundius Project*. [2]

[1]The FAUST website contains an exhaustive list of all the FAUST targets: https://faust.grame.fr.

[2]Unfortunately, there is no documentation/publication on this project yet.

Julius O. Smith implemented a series of waveguide meshes that landed in the FAUST libraries[3] but that were never documented/published.

More recently, the FAUST physical modeling toolkit [11] was introduced. It is based on a library allowing for the implementation of bi-directional block diagrams in FAUST and containing a wide range of musical instrument parts that can be assembled in a modular way. It also comes with `mesh2faust`[12], a tool to generate modal physical models compatible with the FAUST physical modeling library using Finite Element Analysis (FEA).

The work presented in this paper was partly inspired by Ed Berdahl's *Synth-A-Modeler* [7] (which itself direcly draws upon CORDIS-ANIMA [1] and GENESIS [6]). This environment allows for the implementation of hybrid models combining mass-interaction systems with waveguide models using a graphical user interface (GUI). *Synth-A-Modeler* is based on a series of FAUST libraries and generates custom FAUST programs corresponding to the models implemented in the GUI. While it successfully combines various types of modeling techniques at a high level and facilitates their control using custom haptic interfaces such as the FireFader [13], it has, to our knowledge, never been used to implement large scale mass-interaction models.

Our proposed approach does not aim to supplant Berdahl's; rather, from a similar starting point it questions how the FAUST language's versatility can be used to formalize arbitrarily large mass-interaction models – and more generally speaking complex feedback networks – in a direct, concise and clear manner.

2. MASS INTERACTION PARADIGM IN FAUST

Before getting into implementation specifities, this section presents the basics of mass-interaction networks, in the case of 1D systems, in which all masses vibrate along a single z axis. These models are sometimes referred to as "zero-D", since they are purely topological and contain no direct geometrical information. First, discrete-time mass and interaction physical algorithms are presented and assembled into an explicit computational scheme.

Then, relying on a matrix-based representation of the topological network, we present a generic FAUST architecture that implements this computational scheme.

2.1. Discrete-Time Physical Algorithms

Below, we present finite difference implementations of two of the most basic elements in a mass-interaction network: punctual masses and springs.

2.1.1. Discrete-time implementation of a punctual mass

The motion equation for a continuous time mass is given by Newton's second law:

$$f = m.a = m\frac{d^2x}{dt^2} \tag{1}$$

Where f is the force applied to the mass, m is its inertia a its acceleration, and x its position. Applying the second-order central difference scheme, with the sampling interval noted ΔT, a discrete equation of the mass can be formulated as follows:

$$f(t) = m.\frac{x(t+\Delta T) - 2x(t) + x(t-\Delta T)}{\Delta T^2} \tag{2}$$

[3] `https://github.com/grame-cncm/faustlibraries`

Equation (2) can be normalized to unity, and rearranged in order to express the mass' position update scheme (discrete-time positions and forces are noted X and F) :

$$X_{(n+1)} = 2X_{(n)} - X_{(n-1)} + \frac{F_{(n)}}{M} \tag{3}$$

With M, the discrete time inertial parameter defined as :

$$M = \frac{m}{\Delta T^2} \tag{4}$$

Hence, the basic discrete-time mass module produces new position data based on its current position, previous position, the "discrete-time" mass parameter M, and the sum of forces applied to the mass from the previous interaction computation step.

The initial position $X_{(0)}$, delayed initial position $X_{(-1)}$ (which infers initial velocity) and initial force $F_{(0)}$ must be supplied at the start of the computation.

2.1.2. Discrete-time implementation of a dampened spring

The elastic force applied by a linear spring with a stiffness k and a resting length of $l_0 = 0$ connecting a mass $m2$ at the position x_2 to a mass $m1$ at the position x_1 is given by Hookes law:

$$f_{s1\to2} = -k.(x_2 - x_1) \tag{5}$$

The exact equivalent of this equation in discrete time is :

$$F_{s1\to2(n)} = -K.(X_{2(n)} - X_{1(n)}) \tag{6}$$

Where the discrete-time stiffness parameter $K = k$. The friction force applied by a linear damper with a damping parameter z connecting the same two masses is :

$$f_{d1\to2} = -z.\frac{d(x_2 - x_1)}{dt} \tag{7}$$

Using the Backward Euler difference scheme, the frictional force can be formulated in discrete-time as :

$$f_{d1\to2}(t) = -z.\frac{(x_2(t) - x_1(t)) - (x_2(t-\Delta T) - x_1(t-\Delta T))}{\Delta T} \tag{8}$$

Which after normalization becomes :

$$F_{d1\to2(n)} = -Z.((X_{2(n)} - X_{2(n-1)}) - (X_{1(n)} - X_{1(n-1)})) \tag{9}$$

With the discrete time inertial parameter Z defined as :

$$Z = \frac{z}{\Delta T} \tag{10}$$

The global equation of the force applied by the dampened spring is composed of F_s and F_d :

$$\begin{aligned} F_{(n)} = &- K.(X_{2(n)} - X_{1(n)}) \\ &- Z.((X_{2(n)} - X_{2(n-1)}) - (X_{1(n)} - X_{1(n-1)})) \end{aligned} \tag{11}$$

It is applied symmetrically to each mass (Newton's third law):

$$\begin{aligned} F_{2\to1(n)} &= -F_{(n)} \\ F_{1\to2(n)} &= +F_{(n)} \end{aligned} \tag{12}$$

2.1.3. Discrete mass - dampened spring - fixed point oscillator

A linear harmonic oscillator is obtained by combining equations (3) and (11), in the case where X_1 is a fixed point set to $X_{1(n)} = 0$, $n \in \mathbb{Z}$. This results in :

$$X_{(n+1)} = \left(2 - \frac{K+Z}{M}\right).X_{(n)} + \left(\frac{Z}{M} - 1\right).X_{(n-1)} + \frac{F_{(n)}}{M} \quad (13)$$

Since the basic oscillator is a very common element in modeling, the integrated form given in (13) can be implemented in the form a specific mass-type module (although it is identical to assembling a mass, dampened spring and a fixed point).

2.1.4. Generalization

Any element in a mass-interaction model follows the basic template of the elements described above. More complex interactions stem from conditional statements (e.g. springs only active during inter-penetration of two material points, as in visco-elastic collisions) or dynamic stiffness or damping parameters that depend on the position and/or velocity of the connected material points (e.g. through non-linear lookup tables, such as in plucking or bowing interactions [14]).

It is important to note that the M and Z parameters are dependent on the sampling interval. Hence, the oscillatory behaviour of physical models will be dependent on the sampling rate of the simulation.

2.2. Computation Scheme

Computing a mass-interaction model consists in calculating the *mass-type* and *interaction-type* algorithms in a closed loop. The explicit time step increment is carried by the masses, as shown in the discrete-time equation (3). The interactions in themselves are delay-less operations, but can be computed since their output is fed back into the masses for the next calculation step (cf. Figure 2). In other words, calculating a step of real-time audio requires to run all the masses' algorithms once, then all the interactions' algorithms.

2.3. Representing the Topological Network

The topological connections of a mass-interaction model can be formalized as a routing matrix of dimensions $J \times 2K$, where J is the number of material elements (or *M points*) in the network, and K is the number of interactions (each interaction module has two connections - or *L points* in the usual terminology[1]) :

$$\begin{array}{c} \\ m_1 \\ m_2 \\ \vdots \\ m_j \end{array} \begin{array}{c} \begin{array}{ccccc} i_{0\_l1} & i_{0\_l2} & \dots & i_{k\_l1} & i_{k\_l2} \end{array} \\ \begin{pmatrix} 1 \text{ or } 0 & 1 \text{ or } 0 & \dots & \dots & 1 \text{ or } 0 \\ 1 \text{ or } 0 & \dots & \dots & \dots & \vdots \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 1 \text{ or } 0 & \dots & \dots & \dots & 1 \text{ or } 0 \end{pmatrix} \end{array} \quad (14)$$

Each column in the matrix must have a single connection set to 1 and all others to 0, as an *L point* only connects to a single *M point* (partially connected interactions are not allowed). On the other hand, a material point could be connected to any number of interactions in a given model (many connections set to 1 for a single line).

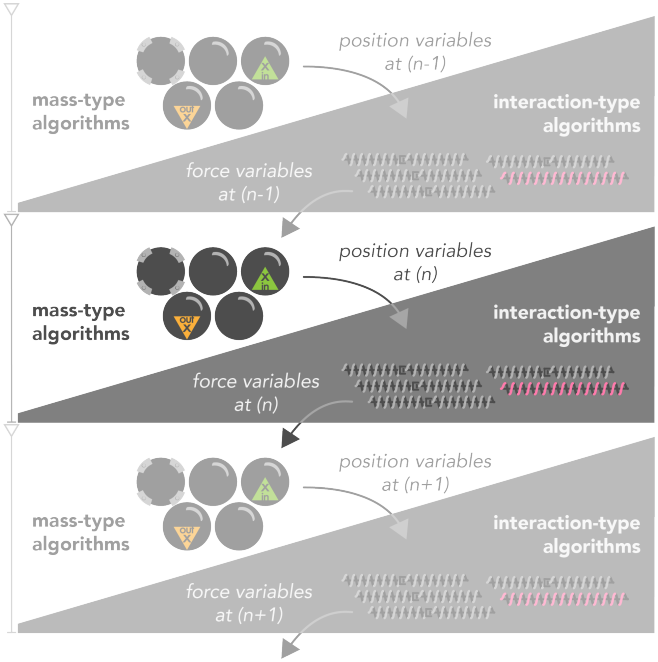

Figure 2: *Computation cycles of the model presented in Figure 1. At each time step, the mass-type algorithms are first computed using the forces calculated in the previous step, then the interaction-type algorithms are computed using the new positions.*

As an example, (15) presents the routing matrix for the topological structure shown in Figure 1. The material elements (fixed point, three masses and a position input module) are represented vertically and the *L points* of the four springs and the non-linear interaction are represented horizontally.

The closed-loop physical calculation scheme performed by FAUST is shown in Fig. 3. On the left, a LINKTOMASS connection function routes the force feedback signals produced by the interactions based on the routing matrix (thus calculating the sum of forces for each mass). The new positions of the material elements modules are then calculated. These positions are then fed into a MASSTOLINK connection function, that routes the signals to all of the concerned interactions. Finally, the pairs of force signals produced by the interactions are fed back for the next calculation step.

Position and force inputs are directly incorporated into the LINKTOMASS function, so that they are applied to the correct input module. Similarly, modules whose positions are observed as audio outputs are simply added as extra signals at the end of the MASSTOLINK function.

2.4. FAUST Implementation of Mass and Interaction Elements

The `mi.lib` library contains the FAUST implementation of most elementary mass-type elements (i.e., masses, fixed points, oscillators, etc.) and link-type elements (i.e., springs, collisions, non-linear plucking / bowing, etc.). Since the implementations are similar, we will explicit only the two simplest and most common elements below: the mass and the spring.

$$
\begin{array}{c} \\ s \\ m_0 \\ m_1 \\ m_2 \\ in \end{array}
\begin{array}{c}
\begin{array}{cccccccccc} r_{0\_1} & r_{0\_2} & r_{1\_1} & r_{1\_2} & r_{2\_1} & r_{2\_2} & r_{3\_1} & r_{3\_2} & nl_1 & nl_2 \end{array} \\
\left(\begin{array}{cccccccccc}
1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\
0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 1 \\
0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0
\end{array} \right)
\end{array}
\tag{15}
$$

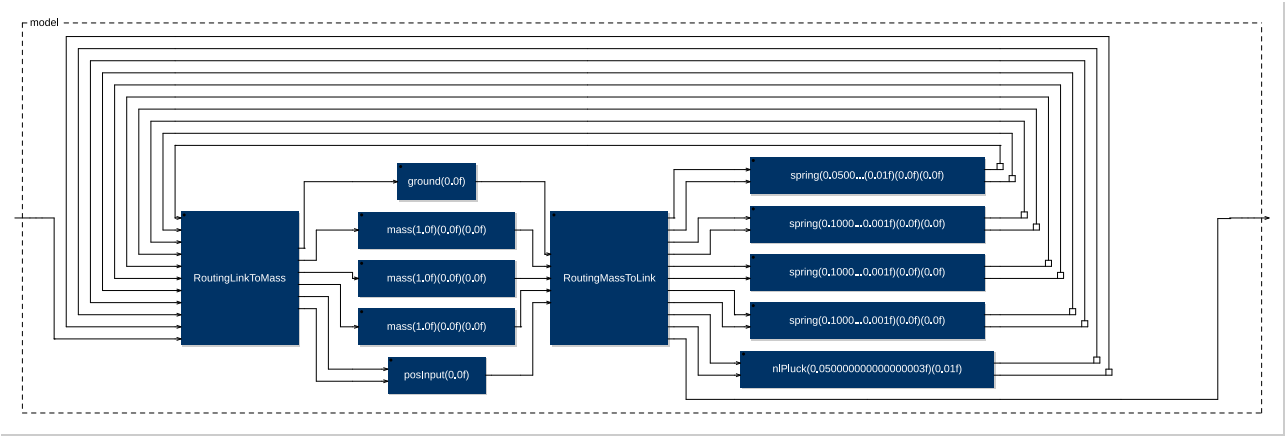

Figure 3: FAUST-*generated diagram corresponding to the model presented in Fig. 1.*

2.4.1. Mass

The discrete-time algorithm of a basic mass module described in (3) can be easily expressed with `letrec` environment expression in FAUST:

```
mass(m,x0,x1) = equation
with{
    A = 2;
    B = -1;
    C = 1/m;
    equation = x
    letrec{
        'x = A*(x : initState(x0)) +
             B*(x' : iniState((x0,x1))) +
             *(C);
    };
};
```

Listing 1: *The discrete-time mass algorithm in* FAUST.

The module takes an input signal (the sum of all forces fed back through the interaction feedback loop and routing function) and produces a position output. The initial position and delayed position of the module are dealt with using the `initState` function, which initializes the first step with the correct values.

2.4.2. Dampened spring

Similarly, a visco-elastic spring expressed in FAUST is shown in Listing 2. Interaction modules such as the spring take two input signals (the positions of the masses connected together by the link) and produce two identical and opposite force signals.

Attention must be paid to the correct initialization of velocity based interactions, especially when the initial position or speed of the masses is non-zero. To this end, the delayed initial positions of the two connected mass elements are supplied as arguments to the interaction function, which initializes them with the `initState()` function.

```
spring(k,z,x1r0,x2r0,x1,x2) =
    k*(x1-x2) +
    z*(
        (x1 - (x1' : initState(x1r0))) -
        (x2 - (x2' : initState(x2r0)))
    )
    <: *(-1),_;
```

Listing 2: *The discrete-time dampened spring algorithm in* FAUST.

3. CREATING MODELS WITH MIMS

MASS INTERACTION MODEL SCRIPTER[4] is a simple graphical or command-line tool written in Python to generate structured FAUST code from a textual description of a physical model.

Models are described in a format similar to the PNSL language [15]: each physical element has a specific label, specific physical parameters and/or initial conditions, etc. Parameters can be added to this description and shared by any number of physical modules, allowing global variation of the physical attributes (i.e., stiffness, damping, mass, etc.) of a subset of modules in real-time.

MIMS' `physics2faust` tool compiles the model by :

- parsing all of the physical modules and noting any specific elements (i.e., position or force inputs, audio outputs, etc.)
- creating the routing matrix and translating it into the two dual FAUST routing functions.

[4]https://github.com/mi-creative/MIMS

- ordering the resulting data into the output *.dsp* file. "Placeholder" functions are created for position / force inputs, allowing the user to describe his input functions directly in the FAUST code.

```
# Define global parameter attributes
@m_K param 0.1
@m_Z param 0.001

@nlK param 0.05
@nlScale param 0.01

# Create material points
@m_s0 ground 0.
@m_m0 mass 1. 0. 0.
@m_m1 mass 1. 0. 0.
@m_m2 mass 1. 0. 0.

# Create and connect interaction modules
@m_r0 spring @m_s0 @m_m0 0.05 0.01
@m_r1 spring @m_m0 @m_m1 m_K m_Z
@m_r2 spring @m_m1 @m_m2 m_K m_Z
@m_r2 spring @m_m2 @m_m0 m_K m_Z

# Inputs and outputs
@in1 posInput 0.
@out1 posOutput @m_m2

# Add plucking interaction
@pick nlPluck @in1 @m_m1 nlK nlScale
```

Listing 3: MIMS *description for the model presented in Fig. 1.*

The graphical UI version of MIMS also provides basic tools for generating certain categories of physical structures (i.e., strings, membranes, etc.) and performing modal analysis of linear structures.

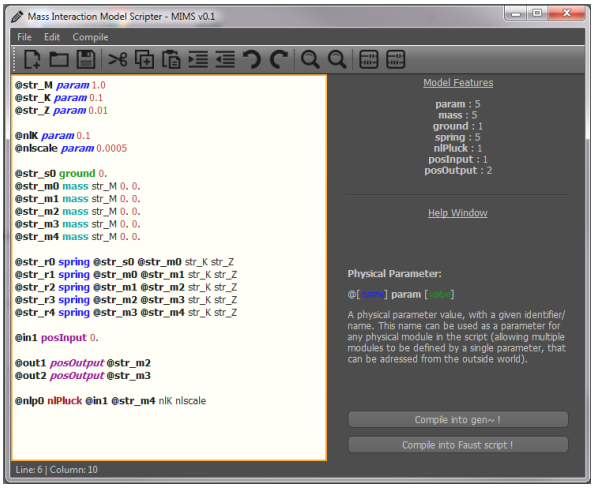

Figure 4: *The MIMS model editor prototype.*

The FAUST code generated from the model in Code Listing 3 is presented in Code Listing 4. The only hand-written element is the `inPos` function, that adds a graphical slider to control the position of the input mass. The control-rate output of the slider is smoothed to avoid artifacts.

```
import("stdfaust.lib");
import("mi.lib");

inPos = hslider("pos",1,-1,1,0.0001) : si.
    smoo;

OutGain = 10.;

m_K = 0.1;
m_Z = 0.001;
nlK = 0.05;
nlScale = 0.01;

model = (
    RoutingLinkToMass:
        ground(0.),
        mass(1.,0., 0.),
        mass(1.,0., 0.),
        mass(1.,0., 0.),
        posInput(0.) :
    RoutingMassToLink :
        spring(0.05,0.01, 0., 0.),
        spring(m_K,m_Z, 0., 0.),
        spring(m_K,m_Z, 0., 0.),
        spring(m_K,m_Z, 0., 0.),
        nlPluck(nlK,nlScale),
    par(i, 1,_)
)~par(i, 10, _): par(i, 10,!), par(i,  1, _)
with{
RoutingLinkToMass(l0_f1,l0_f2,l1_f1,l1_f2,
    l2_f1,l2_f2,l3_f1,l3_f2,l4_f1,l4_f2,in1)
    = l0_f1, l0_f2+l1_f1+l3_f2, l1_f2+l2_f1+
    l4_f2, l2_f2+l3_f1, l4_f1, in1;
RoutingMassToLink(m0,m1,m2,m3,m4) = m0, m1,
    m1, m2, m2, m3, m3, m1, m4, m2,m3;
};
process = inPos : model: *(OutGain);
```

Listing 4: MIMS *description for the model presented in Figure 1.*

4. EXAMPLES AND EVALUATION

The basic *mi_faust* package contains several examples of virtual instruments and use-cases of mass-interaction physics in FAUST. All of these examples can be compiled and executed directly as web applications via the FAUST online editor,[5] with generic user interfaces. They can also be found as pre-compiled web-apps on the *mi_faust* project web-page.[6]

- `IPlayTheTriangle`: the demonstration model discussed previously in this paper (Figure 1).
- `PolyTriangle`: the same model (with a direct force impulse applied instead of a pluck system), using FAUST's ability to automatically handle polyphonic voice allocation for MIDI instruments.
- `PluckedHarmonics`: a 150-mass string terminated by two fixed points. The first position input allows plucking the

[5] `https://faust.grame.fr/tools/editor/`

[6] `https://faust.grame.fr/community/made-with-faust/mi-faust`

string, and three others are used to press down lightly on the string at specific areas in order to bring out natural harmonics.

- `BowedString`: a bowed string, using the *nlBow* interaction. The user can control bow pressure and velocity, as well as the stiffness of the string.
- `LargeTriangleMesh`: a big triangular mesh, fixed at one summit, excited by a plucking system and damped by user input.
- `Resonator`: the audio input is fed into one end of a resonating physical model. The user can alter the properties of the resonator.
- `PhysicalLFO`: Using a physical model with slow dynamics as a control variable for another synthesis process. Here, the wave propagation observed along a very loose string is used to modulate the amplitude of a white noise source, generating AM modulation going from complex patterns at the onset to quasi-sinusoidal modulation as the higher modes of the string decay.

In addition to these examples, two large structures (a 20 by 30 mass mesh: `20x30mesh` and a 1000 mass string: `1000 massString`) were created for model complexity tests. The bench test results in Table 1 show the compile time and CPU load for various models. Large routing functions result in slower compilation, and maximum complexity is reached for approx. 1800 physical elements. Overall, fairly complex models run well, with a reasonable CPU load.

5. FUTURE WORKS

5.1. Faust

FAUST proves to be well adapted to implement mass-interaction physical models. The combination of connection matrices and of the use of the `letrec` environment expression allowed us to seamlessly implement the various elements of `mi.lib`. However, this raised some issues that will need to be solved in the future. They are presented below.

5.1.1. Specifying Initial States in `letrec`

The `letrec` environment expression doesn't allow us to specify an initial state (i.e., the value of $y(n-1)$, $y(n-2)$, etc. at $n=0$). We got around this problem by implementing the `initState` function which requires some unneeded computation. Hence, `letrec` could be modified to allow this type of expression to be written (rewriting Code Listing 1):

```
equation = x
letrec{
    x' = x0;
    x'' = x1;
    'x = A*x +B*x' + *(C);
};
```

We believe that this would significantly reduce computation for large scale models.

5.1.2. Optimizing Routing Matrices

The current "bare bone" implementation of connection matrices (e.g., `RoutingLinkToMass` in Code Listing 3) is hard to solve by the FAUST compiler, preventing large models to be generated (see §4). This could be solved by turning this operation into a primitive of the language. Compilation time would be significantly reduced since pattern matching [5] wouldn't be involved to solve this type of expression.

6. CONCLUSIONS

In this paper, we have presented early results of formal integration of 1D mass-interaction physical modeling into the FAUST environment, resulting in a new library. The MIMS and *physics2faust* tools allow to automatically generate FAUST dsp code for complex topological models, by expliciting the routing scheme for the model's position and force signals. Several basic models have been implemented and benchmarked, showing promising results. Furthermore, FAUST's capabilities offer an efficient solution for playing several dynamically allocated and parameter-mapped instances of a physical model across large ranges. More generally, this work extends beyond mass-interaction modeling and explores the possibilities for describing complex feedback networks and state space-models in FAUST.

7. REFERENCES

[1] 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, 1993.

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

[3] Stefan Bilbao, *Numerical Sound Synthesis: Finite Difference Schemes and Simulation in Musical Acoustics*, John Wiley and Sons, Chichester, UK, 2009.

[4] James Leonard, Nicolas Castagné, Claude Cadoz, and Annie Luciani, *The MSCI Platform: A Framework for the Design and Simulation of Multisensory Virtual Musical Instruments*, pp. 151–169, Springer International Publishing, Cham, 2018.

[5] Yann Orlarey, Stéphane Letz, and Dominique Fober, *New Computational Paradigms for Computer Music*, chapter "Faust: an Efficient Functional Approach to DSP Programming", Delatour, Paris, France, 2009.

[6] Nicolas Castagné and Claude Cadoz, "Genesis: a friendly musician-oriented environment for mass-interaction physical modeling," in *ICMC 2002-International Computer Music Conference*. MPublishing, 2002, pp. 330–337.

[7] 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.

[8] Jérôme Villeneuve and Claude Cadoz, "Understanding and tuning mass-interaction networks through their modal representation," in *40th International Computer Music Conference/11th Sound and Music Computing Conference*, 2014, pp. 1490–1496.

Model Name	N. Masses	N. Springs	FAUST Comp. Dur.	CPU Load
`1000massString`	1000	1002	-	-
`20x30mesh`	598	1151	20.576s	45%
`BowedString`	150	152	1.962s	14%
`IPlayTheTriangle`	3	5	0.029s	1%
`LargeTriangleMesh`	324	901	12.083s	48%
`PhysicalLFO`	10	12	0.032s	1%
`PluckedHarmonics`	150	152	2.192s	14%
`PolyTriangle`	3	5	0.027s	1%
`Resonator`	30	32	0.056s	4%

Table 1: *Number of masses and springs, compilation duration, and CPU load of the examples. Measurements were made on a Lenovo ThinkPad X1 Carbon with the following configuration: Linux Manjaro, Intel i7-7500U 4 cores at 2.7GHz, 16GiB of RAM, sampling rate of 48KHz, buffer size of 256 samples. Programs were compiled as ALSA applications with a GTK interface using* `faust2alsa`*.*

[9] 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.

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

[11] Romain Michon, Julius O. Smith, Chris Chafe, Ge Wang, and Matt Wright, "The faust physical modeling library: a modular playground for the digital luthier," in *Proceedings of the 1st International Faust Conference (IFC-18)*, Mainz (Germany), 2018.

[12] Romain Michon, Sara R Martin, and Julius O Smith, *MESH2FAUST: a Modal Physical Model Generator for the Faust Programming Language - Application to Bell Modeling*, Ann Arbor, MI: Michigan Publishing, University of Michigan Library, 2017.

[13] Edgar Berdahl and Alexandros Kontogeorgakopoulos, "The firefader: Simple, open-source, and reconfigurable haptic force feedback for musicians," *Computer Music Journal*, vol. 37, no. 1, pp. 23–34, 2013.

[14] James Leonard and Claude Cadoz, "Physical modelling concepts for a collection of multisensory virtual musical instruments," in *Proceedings of the Conference on New Interfaces for Musical (NIME15)*, Baton Rouge, USA, May 2015.

[15] Nicolas Castagné, Claude Cadoz, Ali Allaoui, and Olivier Tache, "G3: Genesis software environment update," in *ICMC 2009*. MPublishing, 2009, pp. 407–410.