ECOLE CENTRALE DE LILLE

Physical modelling library in Faust

by

Pierre-Amaury GRUMIAUX

Second-year internship report

in the
MINES ParisTech
Centre de recherche en informatique

December 2016

Declaration of Authorship

I, Pierre-Amaury GRUMIAUX, declare that this report titled, 'Physical modelling synthesis in Faust' and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while an second-year internship as part of my engineering curriculum.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this report is entirely my own work.

Signed: ______________________________

Date: ______________________________

ECOLE CENTRALE DE LILLE

Abstract

MINES ParisTech
Centre de recherche en informatique

by Pierre-Amaury GRUMIAUX

This report is an explanation guide of what was done during my second-year internship. It is divided into three parts :

1. A presentation of the company where the internship took place, who they are, what they do, and what is the project I took part in.
2. A technical review of the mission of the internship, written as clear as possible for the lecturer. This chapter is itself divided into three parts :
   (a) A quick review of what is physical modeling synthesis
   (b) Presentation of Faust, a functional programming language for implementing signal processing blocks, with some elements to understand how it works
   (c) The description of what was produced during the internship : `pm.lib` (a Faust library), `IR2dsp.py` (a python script to generate a modal model in Faust from an impulse response), and `mesh2dsp.py` (a python script to generate a modal model based on FEM analysis).
3. An assessment of the internship, and what can be done in future work.

Acknowledgements

First, I would like to thank Pierre Jouvelot and Emilio Jess Gallego Arias who allowed me to do my second-year internship at the Centre de recherche en informatique at MINES ParisTech. They also gave me the opportunity to choose the topic of my internship, and I thank them a lot for that, as it is a part of my career plan. They were really pleasant during my internship, and also helpful when I needed some advice.

Next, I would like to thank Romain Michon from the CCRMA at Stanford University, without whom the topic of the internship would not have been proposed to me. He provided me a lot of advice and directions for my work, and took a part in the review of it, although the time difference was not our ally.

I would also like to thank all the people involved in the Feever project, of which my work is a part, as well as each member of the Centre de recherche en informatique with whom I shared some time during my internship.

I thank the administration of MINES ParisTech for providing me a accommodation during those three months of internship, which was a privilege.

Finally, I thank all the people who I shared some moments during this internship with, my co-workers at MINES ParisTech, my kins and friends.

Contents

List of Figures

Chapter 1

Company presentation

This internship took place at the Centre de recherche en informatique (CRI) which is the computer science laboratory of the MINES ParisTech school. The CRI is located in Fontainebleau, a lovely French former royal city, around 50 km away from Paris.

1.1 The MINES ParisTech school

The MINES ParisTech school is an engineering school, one of the most prestigious in France, created in the year 1783. It provides a high quality education to train new high level engineers. It also hosts an important research activity within its laboratories. The MINES research activity is shared into five research departments :

- Earth science and environment
- Energetics and process
- Mechanics and materials
- Economics, management and society
- Mathematics and systems

The internship took place in that last department. This department is itself divided into several parts [1] :

- The mathematical morphology center (CMM) whose topic is the science of morphology which is helpful to analyze images.
- The robotics center (CAOR) which studies 3D scenes real-time analysis algorithms.

- The automatics and systems center (CAS) whose activities are about physical systems control.
- The applied mathematics center which shows its skills in modelling and decision theory about climate change.
- The computer science research center (CRI) where I did my internship. Its structure will be more detailed in the following section.
- The bio-informatics center develops some machine learning methods to analyze biological and chemical data.

1.2 The Centre de Recherche en Informatique (CRI)

The CRI is a computer science laboratory located in Fontainebleau which is around 50 km away in the south of Paris. The CRI dedicates its activity to languages for information technology, and develop some techniques for semantics analysis and automated transformations in order to answer to industrial needs (performance, energy consumption, security, development costs) as well as societal and administrative needs (consistent information share, data normalization, information access, heritage safeguard) [2].

ulIts activity rests upon several current projects in the computer science area, among which we can quote MINDs (a musical therapy for Alzheimer suffering patients), ACOPAL (analysis and compilation of parallel programming languages), or LEGIVOC (it provides a terminology database to help for understanding EU laws). This internship took part in the FEEVER project.

1.3 The FEEVER project

FEEVER (**F**aust **E**nvironment **Ever**yware) is a project funded by the ANR since October 2013. The motivation of the FEEVER project is the belief that there are a real improvements to make in the audio digital world [3]. Indeed, today's technologies are not that convenient for the audio engineer. Audio technologies depend on the device on which it is used, and it is not standardize or compatible.

Feever defines itself to be :

- portable, to allow program-once, deploy-everywhere economic advantages

- easily programmable, to narrow the gap between specifications and implementation,
- able to deal with multiple platforms, for seamless integration within the users listening environments,
- efficient both in terms of computing time, since audio processing is a highly compute-intensive activity, and energy, if only to permit mobile applications
- secure, since audio processing activity performed on the client side must not jeopardize the user system

[3]

Chapter 2

Mission

In this section, the internship mission will be explained and developed. Some technical elements will be detailed : first the lecturer will have a look into the physical modelling synthesis, then he will be given some elements to understand how the Faust language works. In the third section, the core of the mission will be introduced : the *pm.lib* file, along with the two scripts `IR2dsp.py` and `mesh2dsp.py`.

2.1 Physical modelling synthesis

Sound synthesis is a subject of research which occupies the time of a lot of computer music researchers. A nice view of the different digital synthesis methods is provided in [4], but let's quickly present some synthesis methods in this paper. The most-known synthesis technique is the additive synthesis. It is obtained by adding together several waveforms (usually harmonically related). The subtractive synthesis is obtained by filtering complex waveforms in order to remove some specific frequencies. Subtractive synthesizer can produce specify waveforms such as triangle waves, sawtooth waves or square waves which have complex frequency spectrum, so that the waveform can be filtered. The FM synthesis (frequency modulation synthesis) requires at least two signal generators. One signal is used to modify one of the characteristics of another signal, which can lead to new sound undoable in additive or subtractive synthesis. The granular synthesis is a method which manipulates very small samples to create a sound. The physical modelling synthesis aims to describe the phenomena of the real world in the best possible way by using physical models. In this section are explained some points of the theory, in particular how digital waveguide models work.

2.1.1 Principle

The physical modelling synthesis refers to a plenty of methods which enable to generate the waveform of a sound based on a mathematical model, with equations and algorithms. This model aims to describe how the sound is produced thanks to the laws of physics. The results will depend on several parameters such as the material, the instrument size or even how the musician wants to excite its instrument (plucked, stricked, bowed ...). There are many kinds of mathematical models for describing how the sound is generated.

2.1.2 Delay lines

A delay line is an elementary element which models an acoustic propagation of the sound. This element is present in most of delay-effects and digital waveguide synthesis models. A delay line inserts a time delay between its input and output, as shown in figure 2.1. If the input signal is denoted $x(n), n = 0, 1, 2...$, then the output signal $y(n)$ of an M-length delay-line is :

$$y(n) = x(n - M), n = 0, 1, 2 \tag{2.1}$$

where $x(n) = 0$ for $n < 0$.

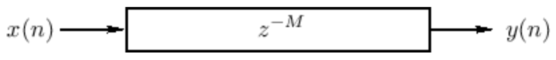

FIGURE 2.1: An M-sample delay line

We can see that the output signal is a kind of delayed version of the input signal.

An example : Damped traveling waves

Figure 2.1 also represents a simulation of a traveling wave which propagates in one direction. However in this form the traveling wave would never stop, so we had to add an element so take the damping factor into account. This damping factor appears at each propagation stage but it can be gather in one final damping factor. In figure 2.2, the damping factor g is merely a constant lower than zero $g < 0$ so that $g^M < 0, \forall M \in \mathbb{N}$ as well. If we need internal signals within this delay line, they can be tapped out and processed with correcting gains.

This damping factor gives good results but is not that representative of the real world. A better approach is the use of digital filters $G(z)$ which implements frequency-dependent attenuation, where $|G(z)| < 1, \forall z \in \mathbb{C}$, as figure 2.3 shows.

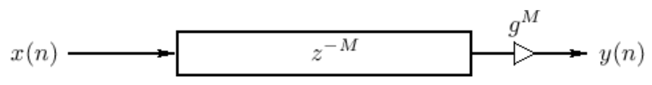

FIGURE 2.2: A damped travelling wave with $g^M < 1, \forall M \in \mathbb{N}$

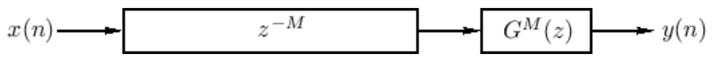

FIGURE 2.3: A damped travelling wave using a digital filter, where $|G(z)| < 1, \forall z \in \mathbb{C}$

2.1.3 Digital waveguides

A digital waveguide is defined as a bidirectional delay line with at some wave impedance R, as shown in figure 2.4.

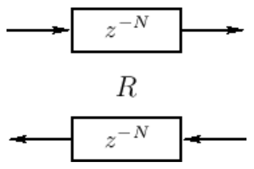

FIGURE 2.4: A digital waveguide N samples long at wave impedance R

As we can see a digital waveguide represents two traveling waves in opposite directions, since it has been proved that the vibration of an ideal string can be decomposed in two traveling waves going in opposites ways. This model can be used for any one-dimensional acoustic system such as strings (guitar, violin ...), bore (clarinet) or pipe (flute, trumpet).

Output of a digital waveguide

The two traveling waves in a digital waveguide are just components of an acoustic vibration. We need to sum these two components to output the needed physical variable, as shown in figure 2.5

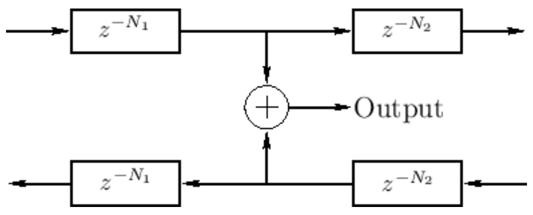

FIGURE 2.5: Output of a digital waveguide using taps (more detailed)

Digital waveguide inputs

Most of musical instruments need to be excited in order to generate a sound, which is modelled in digital waveguides as physical inputs. It corresponds as a disturbance of the 1D propagation medium, and has to be taken into account in both left and right directions of the digital waveguide. It can be a superimposed input (simple vision, figure 2.6) or an interaction input (more relatistic vision, figure 2.7). For example, plucking a string which is already vibrating (interactive input) has to be different than plucking a string at rest (interactive input with no interaction, so additive input).

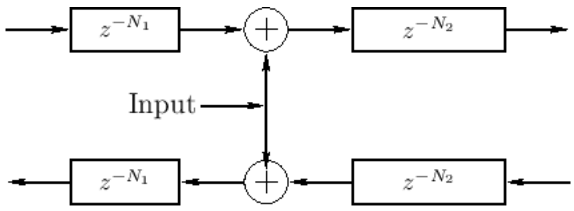

FIGURE 2.6: An input signal is added in one point to both waveguide components

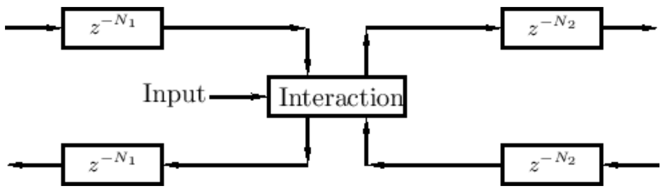

FIGURE 2.7: An interactive input which takes into account the incoming waves at one point and add the result of the interaction with the input at the same point

2.1.4 Ideal string

An ideal vibrating string, if plucked, will vibrate forever in one plane. The wave equation which describes its movement is well-known in physics :

$$K\frac{\partial^2 y}{\partial t^2}(t,x) = \epsilon\frac{\partial^2 y}{\partial x^2}(t,x)$$

where y is the string displacement, K the string tension and ϵ the linear mass density. This form comes from Newton's first law, $force = mass \times acceleration$.

Since d'Alembert in 1747, the solution of the later equation is known and can be expressed as :

$$y(t,x) = y_r(t-\frac{x}{c}) + y_l(t+\frac{x}{c})$$

where $c \triangleq \sqrt{\frac{K}{\epsilon}}$.

We can see here that the equation 2.1.4 looks like the waveguide approach, but we need to put it in a digital way. This continuous solution can be sampled to the equation 2.2 :

$$\begin{aligned} y(nT, mX) &= y_r(nT - mT) + y_l(nT + mT) \\ &\triangleq y^+(n-m) + y^-(n+m) \end{aligned} \tag{2.2}$$

where T is the sampling interval, and $X = cT$ the spatial sampling interval in meters.

This final solution looks like the bidirectional waveguide we have seen sooner.

2.1.5 Rigid terminations

To achieve our string (or bore) physical model, we need to provide some information about both ends of the string, which are called terminations and which are rigid. At these terminations, the string cannot move, which is written as $y(t, 0) = 0$ and $y(t, L) = 0$. Along with the wave equation, this yields to equation 2.3 and 2.4:

$$y(nT, 0) = y^+(n) + y^-(n) \Leftrightarrow y^+(n) = -y^-(n) \tag{2.3}$$

$$y(nT, NX/2) = y^+(n - \frac{N}{2}) + y^-(n + \frac{N}{2}) \Leftrightarrow y^+(n + \frac{N}{2}) = -y^-(n + \frac{N}{2}) \tag{2.4}$$

where $N \triangleq \frac{2L}{X}$ determines the time in samples for the wave to propagate from one end to the other, so the total delay. So we see that both waves are reflected with a negative coefficient which invert the waveform. Both terminations can be added to the bidirectional waveguide model in the following figure 2.8 :

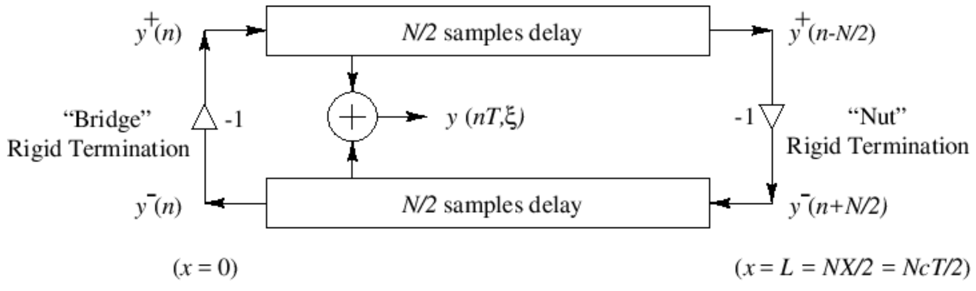

FIGURE 2.8: Rigidly terminated ideal string, with a displacement output at position $x = \xi$

2.1.6 Conclusion

This achieve the chapter about physical modeling synthesis. Only a small part of this theory is reviewed here, but it will be sufficient for understanding what will follow. The interested reader can look further information up in Julius O. Smith's book. [5], where most of the pictures used in this chapter come from.

2.2 Faust language

2.2.1 Introduction

Faust (Functional Audio Stream) is a functional programming language specifically designed for real-time signal processing and synthesis. [6]. Faust is a specification language which provides an adequate notation to describe signal processors. Most of audio devices can be described as signal processors as they take some input signals, and transform them into output signals. The Faust language is also block-diagram oriented. It combines functional programming with block-diagram algebra in order to enable the programmer to construct block-diagram easily with functions composition. Faust programs are fully compiled, the compiler translating the Faust code into C++ code. Faust source file take the .dsp extension, and library file take the .lib extension.

2.2.2 Syntax

A Faust program always contains a "main" function which is called `process` and specifies a block diagram. Here is a simple example which takes the input x and outputs the same signal x, as a simple wire in the physical world :

```
process(x) = x;
```

The simplest signal with only zero-value samples is written :

```
process = 0;
```

Operators

The "," symbol is used for putting diagrams in parallel, as if we want a stereo output signal :

```
process(x,y) = x,y;
```

Mathematical operators such as `+`, `-`, `*` or `/` are also seen as signal processors. For example, the `+` operator takes two input signals in parallel and provides one output signal :

```
process(x,y) = x,y:+;
```

The `:` symbol is used here for putting block diagrams in series. The first part consists in putting x and y into two wires and providing these two wires as inputs for the second part, the `+` operator. However, for convenience Faust enables the programmer to write `process(x,y) = x+y;`. The other symbols works the same way.

An interesting symbol is `_` which consists in a simple wire, so this does nothing to the input signal. Therefore `process(x) = x;` as seen before could be rewritten as

```
process = _;
```

Now let's have a look at an important symbols in block diagrams algebra : the feedback operator `~`, used for recursive composition. The code

```
process = + ~_;
```

provides the diagram in figure 2.9.

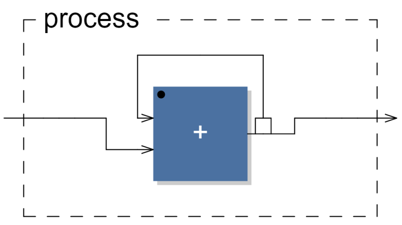

FIGURE 2.9: Block diagram generated for the code `process = + ~_;`

The feedback connects the output to the input and is a nice way for differential equations for example. The last diagram features the equation $y(n) = x(n) + y(n-1)$ where the one-sample delay (represented by the little square) is implicit and cannot be avoided.

There are two other block-diagram operator, the split composition and merge composition. The split composition with the operator `<:`

```
process = _ <:  _, _;
```

enables the programmer to duplicate a signal into two or more identical parallel signals.

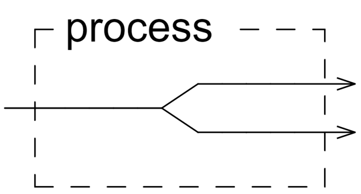

FIGURE 2.10: Block diagram generated for the code `process = _ <:  _, _;`

Inversely, the merge composition, with the operator `:>`

```
process = _, _ :> _
```

provides a way to join two signals by adding them up.

Finally, we can implement delay in an easy way with the `@` operator.

```
process(x) = x@4;
```

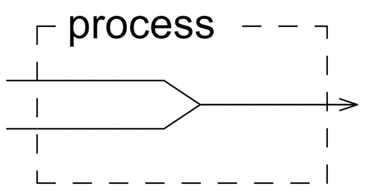

FIGURE 2.11: Block diagram generated for the code `process = _, _ <: _;`

is a processor which outputs the input signal `x` with a 4-samples delay. A one-sample delay can also be written as `x'`, equivalent to `x@1`.

Functions declaration and utilisation

Functions declaration is done in a common way :

```
average(x,y) = (x + y) / 2;
```

where `average` is the function name, and `x` and `y` are the parameters of the function. These parameters are input signals of the corresponding signal processor which transform it into an output. Naming parameters is optional but more readable : indeed the `average` function can be written also in the following way, without specify any parameter :

```
average = (_ , _ : +),2 : /;
```

Function parameters often do not appear in functional blocks as they are implicitly provided in the "wires" and taken into account as input for the functions. For example, using the `average` function with x^2 instead of `x` could be written as :

```
process = square,_:average;
```

where `square` would be the classic square function (`square(x) = x*x;`). The diagram 2.12 illustrates this later code. We see that the output of the `square` function implicitly becomes the first input of the `average` function.

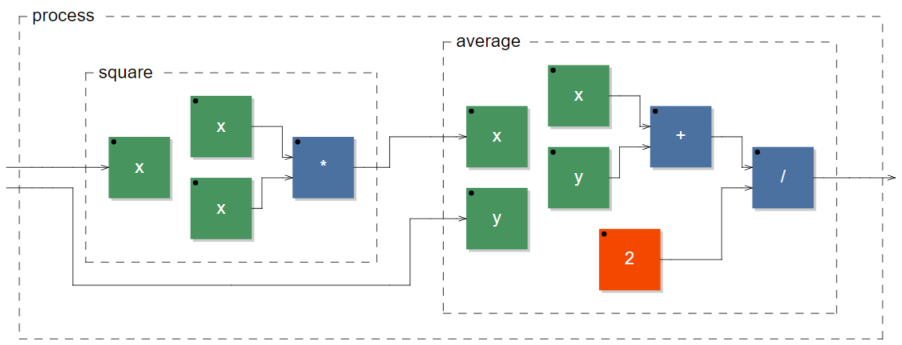

FIGURE 2.12: Block diagram generated for the code `process = square,_:average;`

2.2.3 Libraries

The Faust language enables to include libraries as it is possible in most of programming languages. For that purpose, the `import` function is used to import definitions from other source file:

```
import("math.lib");
```

This library includes a lot of useful math functions.

The work of the internship was mainly to go further with `pm.lib` file, which is a physical modeling synthesis library file for Faust. It will be explained in the next section. Moreover, some library files were used as well : `music.lib` (which imports `math.lib` and declares some useful functions for music applications as spacialisation, envelops, delays) and `filter.lib` (which declares some digital filters, reviewed in [7]).

2.2.4 Diagrams generation

A powerful tool provided with the Faust language is the possibility to generate block diagrams directly from the code. It takes on its full meaning as the Faust language is oriented towards processing blocks building. Such diagrams has been encountered in previous sections.

To generate a block diagram, simply use the `-svg` option while using the `faust` command :

```
faust -svg file.dsp
```

This will create a subfolder of the current folder which contains a `.svg` (scalable vector graphics) file for each block-diagram expression of the file in question. These several `.svg` files are useful for expanding some parts of the final process, or for coming back to a more general view.

2.2.5 Example : a sine oscillator

As an example to get the lecturer more used to the Faust language, here is the sine-oscillator example taken from the official Faust website [8]:

```
phasor(f)   = f/fconstant(int fSamplingFreq, <math.h>) : (+,1.0:fmod) ~ _ ;
osc(f)      = phasor(f) * 6.28318530718 : sin;
process     = osc(hslider("freq", 440, 20, 20000, 1))
                * hslider("level", 0, 0, 1, 0.01);
```

What we want to do here is to create all samples of the sine wave, one by one, infinitely. As we cannot specify samples infinitely we will need a loop, by the use of the ˜ (feedback) operator.

The phase generator creates a periodic signal between 0 to 1. The fmod function is a float modulo function, and we provide 1 as the "limit" of the modulo, i.e. `1.4 fmod 1.0 = 0.4`. So here we provide a constant `0.1` which goes through a `+` operator, then in a `fmod 1.0` operator, which output `0.1`, the first sample.. But we see the feedback loop that takes the output of `fmod` to provide it into `+`, which gives the second sample: `0.2` (after processing by fmod : `0.2 fmod 1.0 = 0.2`), and so on. When the feedback loop following by `+` gives a number more than 1.0 it is wrapped by `~`. It finally creates the signal : `(0.1, 0.2, 0.3, 0.4, ...)`.

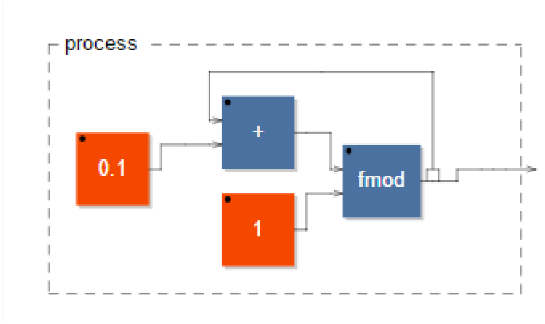

FIGURE 2.13: Block diagram generated for the code `0.1 : (+,1.0:fmod) ~_`

The function `fconstant` aims to extract the constant `fSamplingFreq` from `math.h`. The later signal is divided by the sampling frequency to obtain a signal at . Finally we multiple this signal by the frequency parameter `f` to produce a signal at frequency `f`.

```
osc(f) = phasor(f) * 6.28318530718 :  sin;
```

Now we have the phase of the oscillator, we multiply it by 2π before putting it into the sine function.

Finally, we want the user to be able to modify the input frequency, as well as the volume of the output. An interesting feature that was not presented in this section are GUI elements, like buttons, sliders, nentry (number input) ... Two sliders are used to control the input frequency `f` and the output volume. The input frequency slider becomes the input of the `phasor` function, and we multiply the output signal of the `oscillator` function by the volume slider (a value between `0` and `1`).

The figure 2.14 shows the block diagram generated for the whole code.

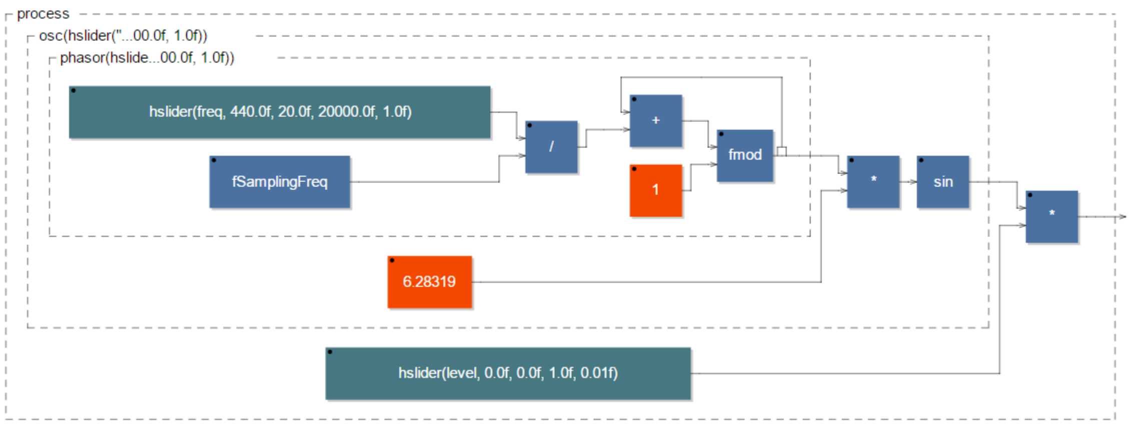
process
osc(hslider("...00.0f, 1.0f))
phasor(hslide...00.0f, 1.0f))
hslider(freq, 440.0f, 20.0f, 20000.0f, 1.0f)
fSamplingFreq
/
+
1
fmod
*
sin
*
6.28319
hslider(level, 0.0f, 0.0f, 1.0f, 0.01f)

FIGURE 2.14: Block diagram generated for the sine oscillator code

2.2.6 Conclusion

The last example shows a simple program using several tools of the Faust language. This language is really useful for processing blocks programming, so it is relevant for music application. We will now dive into the core of the internship mission, the realisation of the `pm.lib` file, which tries to implement some physical modeling synthesis features in the Faust language.

2.3 pm.lib, IR2dsp.py and mesh2dsp.py

In this section, we explain the main productive task of the internship which was the creation of the physical modelling library pm.lib in Faust, and two python scripts, IR2dsp.py and mesh2dsp.py. The entire code of these files can be found in the appendices.

2.3.1 `pm.lib`

The pm.lib file is a Faust library which includes functions to simulate some parts of instruments based on physical modelling synthesis theory, presented in the first section. Romain Michon, who is a Ph. D candidate at the Center for Computer Research in Music and Acoustics (CCRMA) at Stanford University, has written the Faust-STK based on the C++ synthesis tool kit in which there are some instruments based on physical models [9]. The library works well and is very useful, but Romain proposed me to rewrite it in a modular way : still upon physical models, the programming musician can connect different instrumental elements to recreate existing instruments or new ones, in order to extend the creative possibilities. The problem is that this kind of modular system imply bidirectional elements according to the digital waveguide model. The Faust language is built as a one-directional blocks processing language, so it was not possible to directly create bidirectional elements ready to be connected to each other. Romain started the work by cleverly simulating a chain of bidirectional elements, having input and output towards each side of the block diagram. We explain that simulation in the following subsection.

Note that `filter.lib` and `music.lib` is included as we use some of their declared functions.

Simulating bidirectional elements

The Faust language only proposes input from the left and output to the right, whereas we need one digital waveguide, with one input and one output from and towards each direction, as in figure 2.15

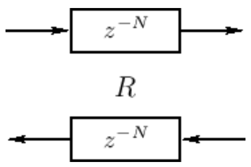

FIGURE 2.15: A digital waveguide as a bidirectional delay

So the idea to simulate it is to move both inputs from the left, and both towards the right. The communication between two elements will be specify in the `chain` function. We also add a third input/output which is the mix of the main signal output.

Given two blocks A and As, the way to chain them together is define in the `chain` function as following :

```
chain(A:As) = (crossnn(1),\_',\_ : \_,A : crossnn(1),\_,\_ : \_,chain(As) :
    crossnn(1),\_,\_) \~ \_ : !,\_,\_,\_;}
```

As this code is not easily readable, let's generate a diagram with the `-svg` option. The diagram 2.16 shows two elements A (adding 1 to each signal) and B (adding 2 to each signal) connected together. There are three inputs/outputs, two for the bidirectional concept and a third one that mix both signals. The first input goes through a `crossnn` function which is used here to switch two incoming signal (the first signal goes to the second input and inversely), and is processed by B before being processed by A, as the right-to-left line of the bidirectional line. The second signal is processed by A then by B as the left-to-right line, but there is a `mem` element at the beginning which puts a one-sample delay to the signal. This is unavoidable, for the simple reason that if there is no delay between the two lines, we could put one block looping into itself so that the first output goes to the second input and the second output goes to the first input : this would not be computable as the Faust language calculates sample by sample.

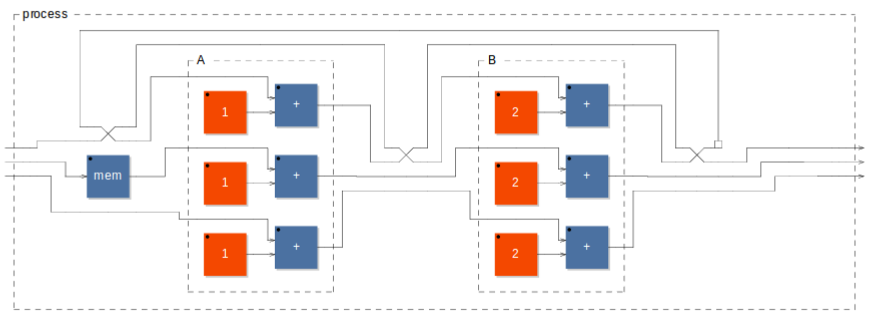

FIGURE 2.16: The `chain` function is used to chain block A and B where `A = +1,+1,+1` and `B = +2,+2,+2`

The `chain` function is also specified for one block only, which does nothing : `chain(A) = A;`.

Now we have managed to specify a bidirectional connection between to block with `chain`, the other fundamental elements are easy to set up.

The `input` and `output` function are also created according to the `chain` specification :

```
input(x) = +(x),+(x),_;
output(x,y,s) = x,y,x+y+s;
```

`input` merely adds a given signal to the two first inputs of one block. (figure 2.17) `output` does nothing to the first two inputs (it outputs them without processing) but it outputs the sum of the three inputs in the third wire. (figure 2.18)

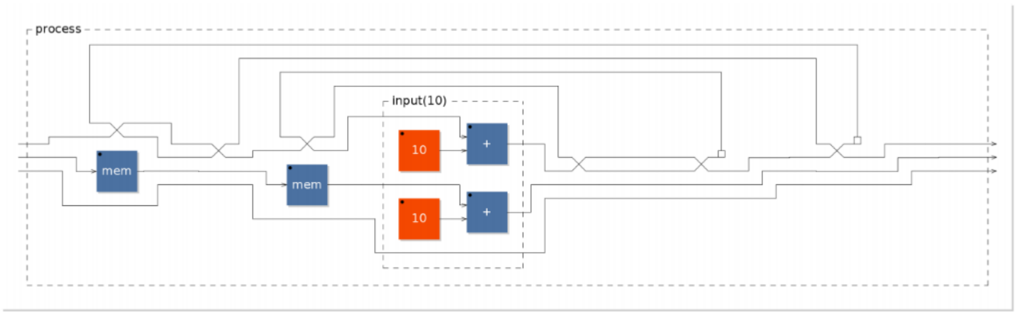

FIGURE 2.17: Diagram showing the `input` function within a chain : `process = chain(_,_,_ : input(10) : _,_,_);`

No we can input and output within a chain of blocks, we specify terminations to simulate the ends of instrument pieces such as strings or bores.

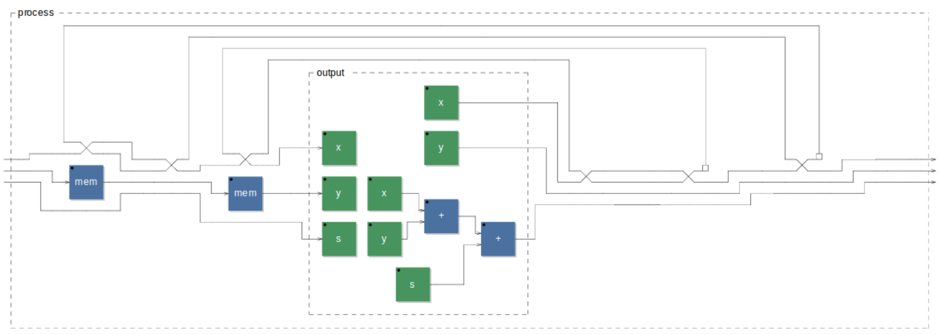

FIGURE 2.18: Diagram showing the `output` function within a chain : `process = chain(_,_,_ : input(10) : _,_,_);`

Terminations

An unspecified termination is created first :

```
terminations(a,b,c) = (_,crossnn(1),_,_ : +,+,_ : b) ~ (c,a : crossnn(1));
```

This function creates terminations on each side of a `chain` but the inputs and outputs are not closed (as we expect from a termination). Like `chain`, this function adds one sample delay to both signals of the bidirectional line. The `terminations` function is used that way :

`rightGoingWaves,leftGoingWaves,mixedOutput : terminations(a,b,c) : rightGoingWaves,`
where `a` and `c` are reflexion coefficients (for example -1) and `b` a chain of blocks.

Then a full termination is specified using the `terminations` function :

```
fullTerminations(a,b,c) = 0,0,0 : terminations(a,b,c) : !,!,_;
```

The `fullTerminations` function closes inputs and outputs of a chain, except for the third output which gives the mixed signal.

Both left termination and right termination are specified as well :

```
leftTermination(a,b) = 0,0,0 : terminations(a,b,*(0));
rightTermination(b,c) = terminations(*(0),b,c) : !,!,_;
```

Those functions cut one end like `fullTerminations` does but leave the other end unclosed.

Excitation functions

What we need to make an instrument playing a sound is an excitation. Two excitation functions have been created here : `impulseExcitation` and `acousticExcitation`.

```
impulseExcitation(gate) = gate <: _,mem : - : >(0);
```

provides a dirac impulse when a gate button is triggered.

```
acousticExcitation(gate,P) = noise : *(gate : trigger(P))
with {
     diffgtz(x) = (x-x') > 0;
     decay(n,x) = x-(x>0)/n;
     release(n) = + ~ decay(n);
     trigger(n) = diffgtz : release(n) : > (0.0);
};
```

creates a noise burst when a gate button is triggered. (full explaination can be found in [10])

Strings

Before specifying any string, we need to implement a `waveguide` function as we see in the first section.

```
waveguide(nMax,n) = fdelay4(nMax,n),fdelay4(nMax,n),_;
```

This uses a 4th order fractional delay implemented in `filter.lib` (see [7]). `n` represents the length of the waveguide in samples, and `nMax` the maximum length of the waveguide (two parameters needed for the filter). We this `waveguide` we can implement an `idealString` function :

```
idealString(length,reflexion,pluckPosition,x) = fullTerminations(term,wg,term)
with{
        nMax = 512; // each segment of the string can't be longer than that
        N = length*SR/320-8; // length (meters) to samples
        nUp = N/2*pluckPosition : max(1); // upper string segment length
        nDown = N/2*(1-pluckPosition) : max(1); // lower string segment length
        wg = chain(waveguide(nMax,nUp) : input(x) : output : waveguide(nMax,nDown
    )); // waveguide chain
        term = *(-reflexion); // terminations
};
```

This string expects some parameters :

- `length` : the length of the string in meters
- `reflexion` : the coefficient of reflexion (between 0 to less than 1 so that the string is damped)

- `pluckPosition` : where the string is plucked (between 0 and 1 excluded)
- `x` : the excitation input

This simulates an ideal string because of the reflexion parameter which is merely a positive coefficient less than 1. We see that the `fullTerminations` function is the main element of this `idealString` function. The functional block is a waveguide which has been defined previously, and both terminations reflect the wave negatively with the reflexion coefficient. We use two waveguides here : one of each side of the pluck position (two waves are created after plucking)

The `length` parameter provides the way to change the frequency of the string sound. The `pluckposition` changes a bit the timbre of the sound, and the `reflexion` coefficient makes the string sounding longer when it approaches 1 (and inversely).

This ideal string is interesting but not that close to the reality, because both ends of a real string instrument do not act merely as a negative coefficient. As every element in the nature, it acts as a filter, giving an output signal from an input signal. The bridge and nut of a string instrument can be simulated with a damping filter, which is a good approximation of the reality.

```
dampingFilter(rho,h0,h1,x) = rho * (h0 * x' + h1*(x+x''));
```

With this filter is created a bridge simulation :

```
bridge(length,B,t60,x) = dampingFilter(rho,h0,h1,x)
with{
        freq = 320/length;
        h0 = (1.0 + B)/2;
        h1 = (1.0 - B)/4;
        rho = pow(0.001,1.0/(freq*t60));
};
```

More natural parameter are transformed in a way that the `dampingFilter` can process with. This bridge reflexion will depend on the frequency of the string (its length) and two interesting parameters :

- `B` : (0-0.99) the more `B` approaches 1, the more brightly the string sounds
- `t60` : the decaying time of the string

The figure 2.19 (taken from `nonIdealString` diagram) helps to understand how the damping filter processes the signal.

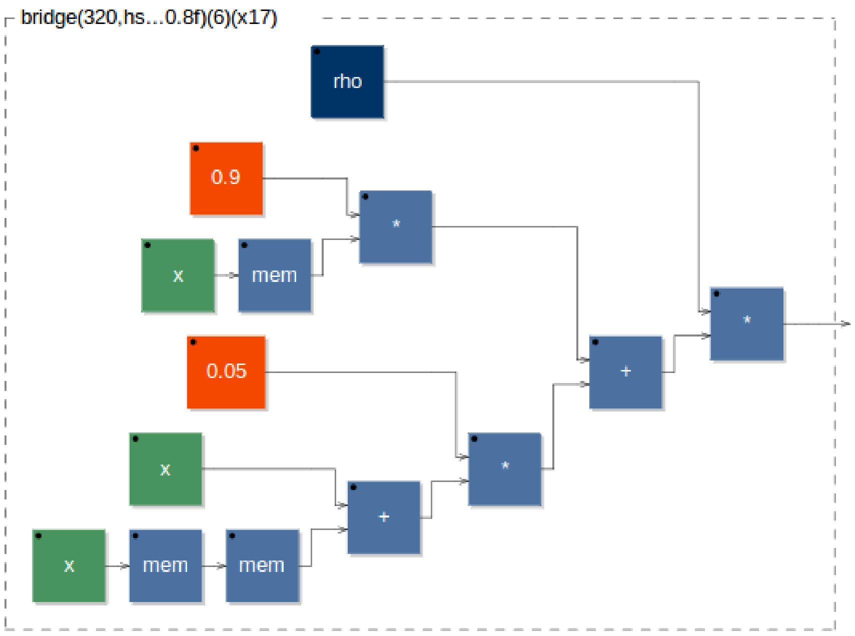

FIGURE 2.19: The bridge implementation using a damping filter

Now an non ideal string can be simulated using the `bridge` function :

```
nonIdealString(length,pluckPosition,B,t60,x) = fullTerminations(term,wg,term)
with{
        nMax = 512; // each segment of the string can't be longer than that
        N = length*SR/320-8; // length (meters) to samples
        nUp = N/2*pluckPosition : max(1); // upper string segment length
        nDown = N/2*(1-pluckPosition) : max(1); // lower string segment length
        wg = chain(waveguide(nMax,nUp) : input(x) : output : waveguide(nMax,nDown
    )); // waveguide chain
        term(x) = bridge(length,B,t60,x);
};
```

This is exactly the same as the `idealString` function except that the reflexion coefficient is replaced by the bridge which processes the incoming wave in a more naturally way. As the `bridge` function expect two new parameters that are `B` and `t60`, so does this function. This `nonIdealString` implementation sounds quite great, in a more acoustic perception.

With this `nonIdealString` function we can go further and directly create two specific strings : a `nylonString` and a `steelString` :

```
steelString(length,pluckPosition,x) = nonIdealString(length,pluckPosition,0.8,6,x
    );
nylonString(length,pluckPosition,x) = nonIdealString(length,pluckPosition,0.25,4,
    x);
```

`B` and `t60` were chosen empirically by ear, to recreate the sound of a steel and a nylon string in the best way.

`pm.lib` includes two other functions : `modeFilter` and `modalModel`, which are used by the two python scripts which are explained below, along with those two last functions.

Conclusion

`pm.lib` is in its beginning, and it is foreseen to add many more functions in order to have a great number of modules which can be assembled to create instruments. However the most important bases of the library are now established and enabled to go further in the work.

2.3.2 `IR2dsp.py`

The idea of this script is to provide a way for a musician to create the sound of an object in Faust, giving its impulse response (IR) in parameter. The impulse response of a system is its output when it is excited with a brief impulse, ideally a Dirac function. The script is written in Python, takes several parameter including the IR, and it creates a .dsp file recreating the system behaviour in Faust. The way to do it in the script is to find the frequency peaks of the signal and to store them. We can then estimate that the signal is the sum of several bandpass filters (which are called the modes of vibration) which central frequencies are those found previously, and to recreate that filter bank in a Faust script. Full code can be found in appendix B.

Power spectrum of the signal

After opening the expected sound file in a signal variable, a Fourier Fast Transform (FFT) is realised to compute the power spectrum of a given IR. The figure 2.20 shows that power spectrum computed for an IR of a glass.

The corresponding piece of code is usual :

```
#Reading file
(fs, x) = read(soundFile)

#Normalizing sound
x = x/max(x)

#FFT
X = np.abs(np.fft.fft(x))
X = X/(max(X))
```

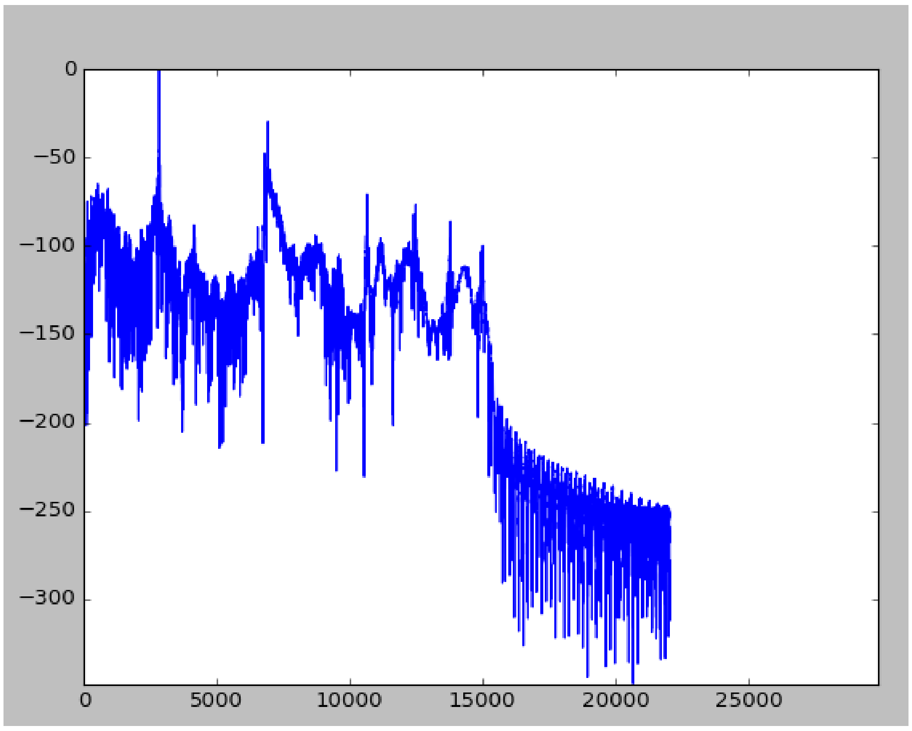

Figure 2.20: Power spectrum in dB of an IR of a glass

```
#computing corresponding frequencies
time_step = 1 / fs
freqs = np.fft.fftfreq(x.size, time_step)
idx = np.argsort(freqs)

#plot for debug
plt.plot(freqs[idx], 20*np.log(X[idx]))
plt.show()
```

Detecting frequency peaks

Now the power spectrum is computed, we need to detect at which frequencies the system responds the more. First we need some information such as the minimum peak amplitude to regard one frequency as a peak, and the minimum distance allowed between two peaks. Indeed, according to the sampling frequency, we could have the two highest frequencies within the same peak, and we do not want to count it as two peaks.

```
#detecting peaks
threshold = math.pow(10,float(peakThreshold)/20) #from dB to X unit
distance = int(peakDistance)/(fs/x.size) in sample count
indexes = peakutils.indexes(X[idx], thres=threshold, min_dist=distance)
```

The peakutils python library is used and allows to directly extract the indexes of the peaks positions in the signal. The two first lines aim to transform the user parameters into readable variable for the program. We expect the user to give more natural parameters, such as threshold in decibel and minimum distance in hertz.

Computing and storing all characteristics of the filters

With the indexes we are able to compute three parameters which describe a bandpass filter : the central frequency, the gain and the t60 (audio decay time, the time to decay to $-60dB$ see [11]).

```
#Storing frequencies and gains for each bp filters
peaksFreq = []
peaksGains = []
nbOfPeaks = 0
for i in indexes:
    if freqs[idx][i] > 0:
        peaksFreq.append(freqs[idx][i])
        peaksGains.append(X[idx][i])
        nbOfPeaks += 1
peaksGains = peaksGains/(max(peaksGains))

#Computing t60 values
peakst60 = []
for i in range(0,nbOfPeaks):
    offset = pow(10,-3/20) #conversion of -3dB in X unit
    peakIndex = indexes[len(indexes) - nbOfPeaks + i]

    n = peakIndex
    while X[idx][n] > (X[idx][peakIndex]*offset):
        n = n-1
    a = n

    n = peakIndex
    while X[idx][n] > (X[idx][peakIndex]*offset):
        n = n+1
    b = n

    bandwidth = (b-a)/(fs/x.size) #bandwidth in Hz
    print bandwidth
    peakst60.append(6.91 / fs/(1-math.exp(-math.pi*bandwidth/fs)))
```

The way to compute the t60 is quite complicated but more details can be found in Julius O. Smith's books. ([12] [13]).

At this stage we have three tables in the python script : a table which stores the frequency peaks, and two tables for corresponding gains and t60.

Creating the `.dsp` file

All needed data is computed from the given IR, so the creation of the corresponding model in Faust remains. For that part we use a usual way to open and write into a file in python, with sometimes calling the three computed tables. The modelisation in Faust uses a function from `pm.lib` : `modalModel` which generates a bandpass filterbank with frequencies, gains and t60.

```
//---modalModel--
// n : number of passband filters
// modeFreqs : list of modal frequencies
// modeGains : list of gains corresponding to the frequencies
// modeT60 : list of t60 corresponding to the frequencies
//--------------
modalModel(n,modeFreqs,modeGains,modeT60) = _ <: par(i,n,gain(i)*modeFilter(freqs
    (i),t60(i))) :> _
with{
        freqs(i) = take(i+1,modeFreqs);
        gain(i) = take(i+1,modeGains);
        t60(i) = take(i+1,modeT60);
};
```

`modeFilter` (see appendix A) is a 2nd order direct form filter (see [12]). `modalModel` simply puts some of this filter in parallel using the three given parameters. In that way, we have managed to reconstruct the IR signal using a bandpass filterbank. Below is an example of the Faust code generated by the `IR2dsp.py` script (see its diagram in figure 2.21.

```
import("architecture/pm.lib");
import("music.lib");

pi = 4*atan(1.0);
nModes = 6;
modeFrequencies = (532.601351351, 920.27027027, 2840.87837838, 6806.25,
    6934.96621622, 7042.90540541);
massEigenValues = (0.0388805406054, 0.0338179522812, 1.0, 0.0929655162617,
    0.228455685234, 0.0407576980285);
t60 = (0.0697422669941, 0.0465209840273, 0.139406145263, 0.0385136491875,
    0.0446632820508, 0.0211090511497);
modeFreqs = par(i,nModes,take(i+1,modeFrequencies));
modeGains = par(i,nModes,take(i+1,massEigenValues));
modeT60 = par(i,nModes,take(i+1,t60));
son = modalModel(nModes,modeFrequencies,modeGains,modeT60);
gate = button("gate");
process = impulseExcitation(gate) : son <: _,_;
```

When compiled, the program provide one button which triggers the impulse excitation into the modal model, and that reproduces the impulsive sound of the given object.

Besides, a help is provided while running the command `python -h script.py`.

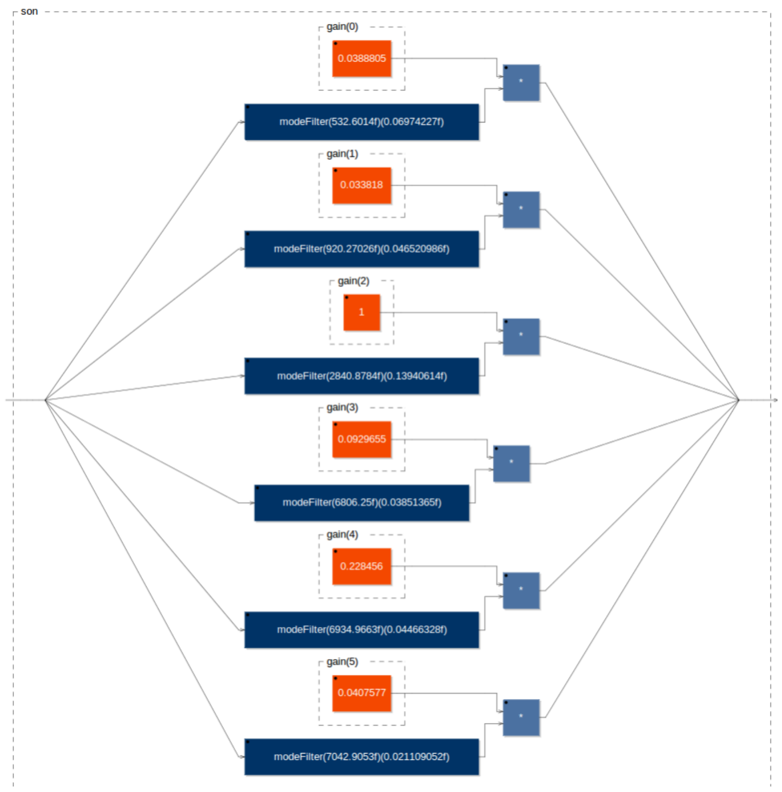
son
gain(0)
0.0388805
modeFilter(532.6014f)(0.06974227f)
gain(1)
0.033818
modeFilter(920.27026f)(0.046520986f)
gain(2)
1
modeFilter(2840.8784f)(0.13940614f)
gain(3)
0.0929655
modeFilter(6806.25f)(0.03851365f)
gain(4)
0.228456
modeFilter(6934.9663f)(0.04466328f)
gain(5)
0.0407577
modeFilter(7042.9053f)(0.021109052f)
*

FIGURE 2.21: Diagram of the filterbank generated by the script, for a glass sound

Conclusion

This script works well, and the several tests we did are satisfying : the different objets sounds quite closely from the reality, even if the raw tapped sounds were more or less similar for the ear.

2.3.3 `mesh2dsp.py`

The goal of this script is really interesting. Given a geometric file (`.stl` file) which only specify the shape of an object, we compute a modal analysis using the finite element method and then we write the faust file as done in `IR2dsp.py`. Before explaining how the script is written, we precise that it does not work the way we want due to a lot of difficulties we encountered. The conclusion cites those issues.

Elmer

The finite element method (FEM) is provided by Elmer, an open source multiphysical simulation software developed by CSC [14]. We can use Elmer in command line so its interesting for our script. For the analysis, we need to provide a `.sif` file which contains some directions for Elmer to do its FEM. To do the FEM analysis, Elmer needs a mesh file, a 3d object which divides the geometric object into small elements. The software ElmerGrid can convert a mesh file into a Elmer supported mesh.

First of all, we need to convert the `.stl` file into a Elmer supported mesh file. This was a first issue in the process : for now no method was found to convert a `.stl` file into a mesh without doing it manually in a specific software. Moreover, ElmerGrid was not completely reliable when used in command line, so we had to do it manually. Finally the input file for the script has to be the name of the folder containing four Elmer supported mesh files (`mesh.boundary`, `mesh.elements`, `mesh.header` `mesh.nodes`).

Then the `.sif` file is created with all information needed for the FEM analysis. When executing the script the user has to provide 3 parameters to inform about the considered material to consider for the analysis, which will appear in the `.sif` file :and

- the Young modulus
- the Poisson coefficient
- the density

Finally without a terminal we can call ElmerSolver to go for the FEM analysis, giving the `.sif` file (which contains the path to the mesh files) :

```
output = subprocess.check_output(["ElmerSolver case.sif"], shell=True)
```

Extracting modal frequencies from ElmerSolver output

ElmerSolver output is quite verbose so we need to parse it in order to extract the eigen values computed by the FEM analysis.

```
#Extracting the list of eigen values
eigenVS = re.findall("\d+\.\d+", output)
eigenV = [float('%.3f'%float((x))) for x in eigenVS]
eigenV = filter(lambda s: s != 0.0, eigenV)
```

We use python functions from strings and a regular expression to find our useful information. Finally, `eigenV` contains all computed eigen values. Then we just transform those eigen values into frequencies.

```
frequencies = [math.sqrt(x)/(2*math.pi) for x in eigenV]
```

Now we have the frequencies, the bandpass filters need gains and t60 values. As the t60 is not possible to compute from a FEM analysis, we let the user provide it as an input for the script, so he can play with the duration of the sound. For the gains which represents the mass of each eigen value in the FEM analysis, we were not be able to find out how to compute it with ElmerSolver unfortunately.

Creating the `.dsp` file

This is exactly the same way we did in `IR2dsp.py` script, as we have three tables of frequencies, gains and t60.

Conclusion

This script is based on a nice idea : ideally we provide a geometric shape and we are able to generate the sound of it when struck, depending on the concerned material. However a lot of issues remain in the script :

- We did not find a effective way to transform a `.stl` file into Elmer supported mesh files so that part has to be done manually for now
- We did not find how to extract eigen value mass participation from ElmerSolver in the FEM analysis. It is an important issue because when reconstructing the behaviour of the object with a bandpass filterbank, each filter will have the same participation so the final result will not sound closely enough.
- Another important issue is that in the `.sif` file we provide some boundary condition. For example in the case of a string (a very long rectilinear cylinder), both end of the cylinder will not move during during the FEM analysis. This is very important to well specify those boundary conditions, but depending on the mesh files provided to ElmerSolver the boundary number will vary, so it is complicated to automatically find whichone are concerned by the conditions, and how many boundaries need this treatment.

The fact that Elmer is a free open source software play an important role in the fact that we did not manage to succeed in every step of the script.

Chapter 3

Future work and internship review

In this section we discuss what directions can be taken in the future to further with the physical modeling library, and a review of the internship is presented.

3.1 Future work

What was done during the internship is the beginning of a project more consistent. The ambition is to finally propose a full physical modeling modular library for programmer musician.

First, a lot of modules have to be added to the library. Other kinds of excitation need to be specified for strings, as only a plucking excitation is modelized : struck strings (piano), bowed strings (violin). We also need to code the resonant part of musical instruments, the body, which takes different forms and sizes according to the instrument. Besides string instruments, with the waveguide model it is possible to model wind instruments (flute, saxophone, oboe, tuba ...). The excitation is a bit different too as you have different manners to blow into the instrument. The end also depends on the concerned instrument so several kinds have to be specified. All these instrument parts have to consist in a module so there are still a lot to implement. The more modular the library is, the more possibilities it offers for the musician.

Concerning the scripts, the one transforming a geometric object into a modal model has to be reconsidered from the beginning as for me. The idea is really interesting but not easy to do in practise. The use of Elmer may be to reconsidered, as I did not manage to find all information we need for the script to run well, and the help provided on the web was not really useful. It is the counterpart of an open source software.

3.2 Internship review

First of all, I really appreciated this internship. I looked for a mission which would be part of my professional plan. I am really interested by all music technologies and my co-workers Pierre and Emilio enabled me to choose the subject I prefer, so it was fully benefit for me.

First of all, this internship was really benefit for me as it is a part of my professional plan. I looked for an engineering or research internship with a musical goal. The subject of this internship took into account music, computer science, signal processing and a bit of physics, which is exactly what I wanted.

I discovered the deep part of one kind of sound synthesis, the physical modeling synthesis. As I am interested in the field of sound synthesis I was really glad to work on that topic and I have gathered some knowledge of that. I also discovered the Faust language, which is a really useful language for musical signal processing. I am sure to reuse that in my future career in music computer science. Moreover, I practised a lot and earned some skills in the computer science area : the use of Linux, the python language, some signal processing consideration ... I am also glad to have discovered the research field. The music computer area is not a huge field and I might work in a research place in my future career. This internship was a nice experience in a computer research center, and I have learned of how the work is done in such a place. I have learned that in a research job you are quite free to organize your work, so it is not that easy to be autonomous and to get things done. You have to keep being motivated by yourself as there is no authority to get you work. My conclusion is that the research area is fantastic if you choose a subject you are fond of, otherwise it will be a pain to make progress in your work.

There also were some more difficult points in this internship, and I learned a lot from them. As I was quite autonomous I had to fight issues by myself, by searching on the Internet or in books. I think it is an good habit to take as a future engineer, we have to find solutions to problems, and it is important to learn how and where to look for. The script `mesh2dsp.py` was quite a failure because it does not work finally. But I learned that sometimes software we can find on the Internet are not that reliable so we have to do with or find another one.

There is one thing that upset me at the end of my internship. It is that the work in far from being done, as there is still a lot of instrument modules to be implemented. I wish I did more during the internship. Another annoying thing is the impression that all the work done is quite useless, because for now the library is not finished so unusable, and there is no one to continue it full-time. But I put things into perspective and what I understand from that is that when you are in the research area, of course only a couple of people would know or use your work, as it is state-of-the-art technology.

I finally ended my internship with new skills and new knowledge. I have gained experience and vision about music computer field, so that internship was really benefit for me.

Appendix A

pm.lib

```
// # ‘pm.lib‘: Physical Modeling Library

import("filter.lib");
import("music.lib");

//-------chain(A:B:...)----------
// Creates a chain of bidirectional blocks.
// Blocks must have 3 inputs and outputs. The first input/output correspond to
    the left
// going signal, the second input/output correspond to the right going signal and
     the
// third input/output is the mix of the main signal output. The implied one
    sample delay
// created by the ‘~‘ operator is generalized to the left and right going waves.
    Thus, n
// blocks in ‘chain()‘ will add an n samples delay to both the left and right
    going waves.
// ### Usage
// ‘‘‘
// rightGoingWaves,leftGoingWaves,mixedOutput : chain(A:B) : rightGoingWaves,
    leftGoingWaves,mixedOutput
// with{
//              A = _,_,_;
//              B = _,_,_;
// };
// ‘‘‘
// ### Requires
// ‘filter.lib‘ (‘crossnn‘)
//----------------------------
chain(A:As) = (crossnn(1),_’,_ : _,A : crossnn(1),_,_ : _,chain(As) : crossnn(1),
    _,_) ~ _ : !,_,_,_;
chain(A) = A;

//-------input(x)--------------
// Adds a waveguide input anywhere between 2 blocks in a chain of blocks (see ‘
    chain()‘).
// ### Usage
// ‘‘‘
// string(x) = chain(A:input(x):B)
// ‘‘‘
// Where ‘x‘ is the input signal to be added to the chain.
//-----------------------------
input(x) = +(x),+(x),_;

//-------output()--------------
// Adds a waveguide output anywhere between 2 blocks in a chain of blocks and
    sends it
// to the mix output channel (see ‘chain()‘).
// ### Usage
// ‘‘‘
// chain(A:output:B)
// ‘‘‘
//-----------------------------
output(x,y,s) = x,y,x+y+s;

//-------terminations(a,b,c)--------------
// Creates terminations on both sides of a ‘chain()‘ without closing the inputs
    and
// outputs of the bidirectional signals chain. As for ‘chain()‘, this function
    adds a 1
// sample delay to the bidirectional signal both ways.
// ### Usage
// ‘‘‘
// rightGoingWaves,leftGoingWaves,mixedOutput : terminations(a,b,c) :
    rightGoingWaves,leftGoingWaves,mixedOutput
// with{
//              a = *(-1); // left termination
//              b = chain(D:E:F); // bidirectional chain of blocks (D, E, F, etc
    .)
//              c = *(-1); // right termination
// };
// ‘‘‘
// ### Requires
// ‘filter.lib‘ (‘crossnn‘)
//----------------------------------------
terminations(a,b,c) = (_,crossnn(1),_,_ : +,+,_ : b) ~ (c,a : crossnn(1)); //
    adds 2 samples to the loop!

//-------fullTerminations(a,b,c)----------
// Same as ‘terminations()‘ but closes the inputs and outputs of the
    bidirectional chain
// (only the mixed output remains).
// ### Usage
// ‘‘‘
// terminations(a,b,c) : _
// with{
//              a = *(-1); // left termination
//              b = chain(D:E:F); // bidirectional chain of blocks (D, E, F, etc
    .)
//              c = *(-1); // right termination
// };
// ‘‘‘
// ### Requires
// `filter.lib` (`crossnn`)
//----------------------------------------
fullTerminations(a,b,c) = 0,0,0 : terminations(a,b,c) : !,!,_;

//-------leftTermination(a,b)----------
// Creates a termination on the left side of a `chain()` without closing the
    inputs and
// outputs of the bidirectional signals chain. This function adds a 1 sample
    delay near
// the termination.
// ### Usage
// ```
// rightGoingWaves,leftGoingWaves,mixedOutput : terminations(a,b) :
    rightGoingWaves,leftGoingWaves,mixedOutput
// with{
//              a = *(-1); // left termination
//              b = chain(D:E:F); // bidirectional chain of blocks (D, E, F, etc
    .)
// };
// ```
// ### Requires
// `filter.lib` (`crossnn`)
//----------------------------------------
leftTermination(a,b) = 0,0,0 : terminations(a,b,*(0));

//-------rightTermination(b,c)----------
// Creates a termination on the right side of a `chain()` without closing the
    inputs and
// outputs of the bidirectional signals chain. This function adds a 1 sample
    delay near
// the termination.
// ### Usage
// ```
// rightGoingWaves,leftGoingWaves,mixedOutput : terminations(b,c) :
    rightGoingWaves,leftGoingWaves,mixedOutput
// with{
//              b = chain(D:E:F); // bidirectional chain of blocks (D, E, F, etc
    .)
//              c = *(-1); // right termination
// };
// ```
// ### Requires
// `filter.lib` (`crossnn`)
//----------------------------------------
rightTermination(b,c) = terminations(*(0),b,c) : !,!,_;

//-------waveguide(nMax,n)----------
// A simple waveguide block based on a 4th order fractional delay.
// ### Usage
// ```
// rightGoingWaves,leftGoingWaves,mixedOutput : waveguide(nMax,n) :
    rightGoingWaves,leftGoingWaves,mixedOutput
// ```
// With:
// * `nMax`: the maximum length of the waveguide in samples
```

```
// * ‘n‘ the length of the waveguide in samples.
// ### Requires
// ‘filter.lib‘ (‘fdelay4‘)
//----------------------------------
waveguide(nMax,n) = fdelay4(nMax,n),fdelay4(nMax,n),_;

//-------idealString(length,reflexion,xPosition,x)----------
// An ideal string with rigid terminations and where the plucking position and
    the
// pick-up position are the same.
// ### Usage
// ‘‘‘
// 1-1’ : idealString(length,reflexion,xPosition,x)
// ‘‘‘
// With:
// * ‘length‘: the length of the string in meters
// * ‘reflexion‘: the coefficient of reflexion (0-0.99999999)
// * ‘pluckPosition‘: the plucking position (0.001-0.999)
// * ‘x‘: the input signal for the excitation
// ### Requires
// ‘filter.lib‘ (‘fdelay4‘,‘crossnn‘)
//----------------------------------------------------------
idealString(length,reflexion,pluckPosition,x) = fullTerminations(term,wg,term)
with{
        nMax = 512; // each segment of the string can’t be longer than that
        N = length*SR/320-8; // length (meters) to samples
        nUp = N/2*pluckPosition : max(1); // upper string segment length
        nDown = N/2*(1-pluckPosition) : max(1); // lower string segment length
        wg = chain(waveguide(nMax,nUp) : input(x) : output : waveguide(nMax,nDown
    )); // waveguide chain
        term = *(-reflexion); // terminations
};

//-------dampingFilter(rho,h0,h1,x)--------------------
//----------------------------------------------------
dampingFilter(rho,h0,h1,x) = rho * (h0 * x’ + h1*(x+x’’));

//-------bridge(length,B,t60,x)-----------
// Simulate a bridge using a dampingFilter.
// ### Usage
// ‘‘‘
//
// ‘‘‘
// With:
// * ‘length‘: length of the string in meters
// * ‘B‘: the brightness of the string (0-0.99)
// * ‘t60 : decaying time
// * ‘x‘: the input signal for the excitation
// ### Requires
// ‘filter.lib‘
//----------------------------------------------------
bridge(length,B,t60,x) = dampingFilter(rho,h0,h1,x)
with{
        freq = 320/length;
        h0 = (1.0 + B)/2;
        h1 = (1.0 - B)/4;
        rho = pow(0.001,1.0/(freq*t60));
};

//-------nonIdealString---------------------------
nonIdealString(length,pluckPosition,B,t60,x) = fullTerminations(term,wg,term)
with{
        nMax = 512; // each segment of the string can't be longer than that
        N = length*SR/320-8; // length (meters) to samples
        nUp = N/2*pluckPosition : max(1); // upper string segment length
        nDown = N/2*(1-pluckPosition) : max(1); // lower string segment length
        wg = chain(waveguide(nMax,nUp) : input(x) : output : waveguide(nMax,nDown
    )); // waveguide chain
        term(x) = bridge(length,B,t60,x);
};

//-------steelString-----------------------------
//-----------------------------------------------
steelString(length,pluckPosition,x) = nonIdealString(length,pluckPosition,0.8,6,x
    );

//-------nylonString-----------------------------
//-----------------------------------------------
nylonString(length,pluckPosition,x) = nonIdealString(length,pluckPosition,0.25,4,
    x);

//-------impulseExcitation(gate)--------------
// Creates an impulse excitation of one sample, from a gate button when it is
    triggered.
// ### Usage
// ```
// gate = button('gate');
// impulseExcitation(gate) : chain;
// ```
// With:
// * `gate : a gate button
//--------------------------------------
impulseExcitation(gate) = gate <: _,mem : - : >(0);

//-------acousticExcitation(gate,P)-------------
// Creates an acoustic excitation (a noise burst) when a gate button is triggered
    .
// ### Usage
// ```
// gate = button('gate');
// P = SR/freq;
// acousticExcitation(gate,P) : chain;
// ```
// With:
// * `gate : a gate button
// * `P   : fundamental period in samples
// ### Requires
// 'music.lib' (noise)
//--------------------------------------
acousticExcitation(gate,P) = noise : *(gate : trigger(P))
```

```
with {
     diffgtz(x) = (x-x') > 0;
     decay(n,x) = x-(x>0)/n;
     release(n) = + ~ decay(n);
     trigger(n) = diffgtz : release(n) : > (0.0);
};


//-----modeFilter-----
modeFilter(f,t60) = tf2(b0,b1,b2,a1,a2)
with{
        b0 = 1;
        b1 = 0;
        b2 = -1;
        w = 2*PI*f/SR;
        r = pow(0.001,1/float(t60*SR));
        a1 = -2*r*cos(w);
        a2 = r^2;
};

//---modalModel--
// n : number of passband filters
// modeFreqs : list of modal frequencies
// modeGains : list of gains corresponding to the frequencies
// modeT60 : list of t60 corresponding to the frequencies
//--------------
modalModel(n,modeFreqs,modeGains,modeT60) = _ <: par(i,n,gain(i)*modeFilter(freqs
    (i),t60(i))) :> _
with{
        freqs(i) = take(i+1,modeFreqs);
        gain(i) = take(i+1,modeGains);
        t60(i) = take(i+1,modeT60);
};
```

Appendix B

IR2dsp.py

```
from __future__ import division
import math
import numpy as np
import matplotlib.pyplot as plt
from sys import argv
import subprocess
from scipy.io.wavfile import read
import peakutils
import peakutils.plot as pkplt
import operator
import argparse

#Help for use of the script
parser=argparse.ArgumentParser(
    description="The IR2dsp.py script is a python script that generates a .dsp
    file from an impulse response file (a .wav file). The impulse response is
    analyzed in order to rebuild the sound of the vibration of the object based
    on this impulse response.",
    epilog="See http://faust.grame.fr/images/faust-tutorial2.pdf for more
    information about Faust")
parser.add_argument('soundFile', type=str, help="Path of the sound file")
parser.add_argument('modelName', help='The name of the model created in the .dsp
    file')
parser.add_argument('peakThreshold', help='Minimum value of peaks in dB (between
    - infinity and 0)')
parser.add_argument('peakDistance', help='Minimum distance between two peaks in
    Hertz')
ars=parser.parse_args()

#Arguments
script, soundFile, modelName, peakThreshold, peakDistance = argv

#Reading file
(fs, x) = read(soundFile)

#Normalizing sound
x = x/max(x)
```

```
#FFT
X = np.abs(np.fft.fft(x))
X = X/(max(X))

#computing corresponding frequencies
time_step = 1 / fs
freqs = np.fft.fftfreq(x.size, time_step)
idx = np.argsort(freqs)

#plot for debug
#plt.plot(freqs[idx], 20*np.log(X[idx]))
#plt.show()

#detecting peaks
threshold = math.pow(10,float(peakThreshold)/20) #from dB to X unit
distance = int(peakDistance)/(fs/x.size)
indexes = peakutils.indexes(X[idx], thres=threshold, min_dist=distance)

#Storing frequencies and modes for each bp filters
peaksFreq = []
peaksGains = []
nbOfPeaks = 0
for i in indexes:
    if freqs[idx][i] > 0:
        peaksFreq.append(freqs[idx][i])
        peaksGains.append(X[idx][i])
        nbOfPeaks += 1
peaksGains = peaksGains/(max(peaksGains))

#Computing t60 values
peakst60 = []
for i in range(0,nbOfPeaks):
    offset = pow(10,-3/20) #conversion of -3dB in X unit
    peakIndex = indexes[len(indexes) - nbOfPeaks + i]

    n = peakIndex
    while X[idx][n] > (X[idx][peakIndex]*offset):
        n = n-1
    a = n

    n = peakIndex
    while X[idx][n] > (X[idx][peakIndex]*offset):
        n = n+1
    b = n

    bandwidth = (b-a)/(fs/x.size) #bandwidth in Hz
    print bandwidth
    peakst60.append(6.91 / fs/(1-math.exp(-math.pi*bandwidth/fs)))


print "peaks frequencies :"
print peaksFreq
print "corresponding gains :"
print peaksGains
print "corresponding t60 :"
print peakst60

# Writing the dsp file #
########################
file = open(modelName + ".dsp", "w")

file.write("import(\"architecture/pm.lib\");\n")
file.write("import(\"music.lib\");\n\n")
file.write("pi = 4*atan(1.0);\n")
file.write("nModes = ")
file.write(str(len(peaksGains)))
file.write(";\n")
file.write("modeFrequencies = ("); #writing the frequencies list
k = 0
for i in peaksFreq :
    file.write(str(i))
    if(k+1 < len(peaksGains)):
        file.write(", ")
    k += 1
file.write(");\n");
file.write("massEigenValues = ("); #writing the masses list
k = 0
for i in peaksGains :
    file.write(str(i))
    if(k+1 < len(peaksGains)):
        file.write(", ")
    k += 1
file.write(");\n");
file.write("t60 = ("); #writing the t60 list
k = 0
for i in peakst60 :
    file.write(str(i))
    if(k+1 < len(peaksGains)):
        file.write(", ")
    k += 1
file.write(");\n");
file.write("modeFreqs = par(i,nModes,take(i+1,modeFrequencies));\n")
file.write("modeGains = par(i,nModes,take(i+1,massEigenValues));\n")
file.write("modeT60 = par(i,nModes,take(i+1,t60));\n")
file.write(modelName)
file.write(" = modalModel(nModes,modeFrequencies,modeGains,modeT60);");
file.write('\ngate = button("gate");')
file.write('\nprocess = impulseExcitation(gate) : ' + modelName + ' <: _,_;')

file.close();
```

Appendix C

mesh2dsp.py

```
from sys import argv
import os
import subprocess
import string
import re
import math

script, stlFile, modelName, young, poisson, density, t = argv

fileName = stlFile[0:len(stlFile)-4]
print fileName

#subprocess.call(["gmsh", "-2", "-o", stlFile])
#mshFile = fileName + ".msh"
#subprocess.call(["ElmerGrid", "14", "2", mshFile)

# Writing of the sif file#
##########################

sifFile = open("case.sif", "w")
sifFile.write("Header")
sifFile.write('Mesh DB "." "'+fileName+'"')
sifFile.write("""
Include Path ""
Results Directory ""
End

Simulation
Coordinate System = "Cartesian 3D"
Coordinate Mapping(3) = 1 2 3
Simulation Type = "Steady State"
Steady State Max Iterations = 1
Solver Input File = "eigen_values.sif"
Output File = "eigen_values.dat"
Post File = "eigen_values.ep"
End

Body 1
Equation = 1
Material = 1
End
Material 1
""")
sifFile.write("Youngs Modulus = ")
sifFile.write(young)
sifFile.write("\nPoisson Ratio = ")
sifFile.write(poisson)
sifFile.write("\nDensity = ")
sifFile.write(density)
sifFile.write("""
End

Equation 1
Stress Analysis = True
End

Solver 1
Equation = "Stress Analysis"
Eigen Analysis = Logical True
Eigen System Values = Integer 20
Linear System Solver = "direct"
Variable = "Displacement"
Variable Dofs = 3
Linear System Iterative Method = "BiCGStab"
Linear System Max Iterations = 1000
Linear System Convergence Tolerance = 1.0e-08
Linear System Abort Not Converged = True
Linear System Preconditioning = "ILU0"
Linear System Residual Output = 1
Steady State Convergence Tolerance = 1.0e-05
Nonlinear System Convergence Tolerance = 1.0e-05
Nonlinear System Max Iterations = 1
Nonlinear System Newton After Iterations = 3
Nonlinear System Newton After Tolerance = 1.0e-02
Nonlinear System Relaxation Factor = 1
Linear System Precondition Recompute = 1
End

Boundary Condition 1
Target Boundaries(1) = 1
Displacement 1 = 0
Displacement 2 = 0
Displacement 3 = 0
End

Boundary Condition 2
Target Boundaries(1) = 2
Displacement 1 = 0
Displacement 2 = 0
Displacement 3 = 0
End

""")
```

```
sifFile.close()

output = subprocess.check_output(["ElmerSolver case.sif"], shell=True)
index1 = string.rfind(output, "Computed Eigen Values")
index2 = string.rfind(output, "EigenSolve")
output = output[index1:index2]

#Extracting the list of eigen values
eigenVS = re.findall("\d+\.\d+", output)
eigenV = [float('%.3f'%float((x))) for x in eigenVS]
eigenV = filter(lambda s: s != 0.0, eigenV)

frequencies = [math.sqrt(x)/(2*math.pi) for x in eigenV]
print frequencies
nModes = len(frequencies)
masses = [1] * nModes
print masses
t60 = t;



# Writing of the dsp file #
###########################
file = open("modalModel.dsp", "w")

file.write("import(\"architecture/pm.lib\");\n")
file.write("import(\"music.lib\");\n\n")
file.write("pi = 4*atan(1.0);\n")
file.write("nModes = ")
file.write(str(nModes))
file.write(";\n")
file.write("eigenValues = ("); #writing the frequencies list
k = 0
for i in frequencies :
    file.write(str(i))
    if(k+1 < nModes):
        file.write(", ")
    k += 1
file.write(");\n");
file.write("massEigenValues = ("); #writing the masses list
k = 0
for i in masses :
    file.write(str(i))
    if(k+1 < nModes):
        file.write(", ")
    k += 1
file.write(");\n");
file.write("modeFreqs = par(i,nModes,sqrt(take(i+1,eigenValues))*2*pi);\n")
file.write("modeGains = par(i,nModes,take(i+1,massEigenValues));\n")
file.write("t60 = ")
file.write(str(t60))
file.write(";\n\n")
file.write(modelName)
file.write(" = modalModel(nModes,modeFreqs,modeGains,t60);");
file.close();
```

Bibliography

[1] Mines paristech. URL http://www.mines-paristech.fr/Recherche/Domaines-de-recherche/Mathematiques-et-systemes/.

[2] Centre de recherche en informatique. URL https://www.cri.ensmp.fr/index.html.

[3] Feever. URL http://feever.fr/.

[4] Julius O. Smith III. Viewpoints on the history of digital synthesis. 2005. URL https://ccrma.stanford.edu/~jos/kna/kna.pdf.

[5] Julius O. Smith III. *Physical Audio Signal Processing.* . ISBN 9780974560724. URL https://ccrma.stanford.edu/~jos/pasp/.

[6] Grame. Faust quick reference. 2014. URL http://faust.grame.fr/images/faust-quick-reference.pdf.

[7] J. O. Smith. Virtual electric guitars and effects using faust and octave. *Proc. 6th Int. Linux Audio Conf (LAC2008)*, 2008. URL https://ccrma.stanford.edu/~jos/pdf/LAC2008-jos.pdf.

[8] Grame. A sine oscillator, 2015. URL http://faust.grame.fr/examples/2015/09/30/oscillator.html.

[9] Romain Michon. The faust-stk. URL https://ccrma.stanford.edu/~rmichon/faustSTK/.

[10] Orlarey Gräf Kersten. URL http://lac.zkm.de/2006/papers/lac2006_orlarey_et_al.pdf.

[11] Julius O. Smith III. *Mathematics of the Discrete Fourier Transform*, chapter Sinusoids and Exponentials. . URL https://ccrma.stanford.edu/~jos/mdft/Audio_Decay_Time_T60.html.

[12] Julius O. Smith III. *Introduction to Digital Filters.* . ISBN 9780974560717. URL https://ccrma.stanford.edu/~jos/fp/.

[13] Julius O. Smith III. *Mathematics of the Discrete Fourier Transform.* . ISBN 9780974560748. URL https://ccrma.stanford.edu/~jos/mdft/.

[14] URL https://www.csc.fi/web/elmer.