INTRODUCING FINITE DIFFERENCE SCHEMES SYNTHESIS IN FAUST: A CELLULAR AUTOMATA APPROACH

Riccardo RUSSO (rrusso19@student.aau.dk) [1], **Stefania SERAFIN**[1], **Romain MICHON**[2,3], **Yann ORLAREY**[2], and **Stéphane LETZ**[2]

[1] **Aalborg University**, Copenhagen, Denmark
[2] **GRAME-CNCM**, Lyon, France
[3] *CCRMA*, **Stanford University**, USA

ABSTRACT

In this paper we propose a technique for formalizing Finite Difference Schemes (FDSs) physical models in the Faust programming language. Faust libraries already allow for the implementation of several kinds of physical modeling techniques; however, to our knowledge, FDSs have never been integrated into this language. In fact, their implementation in imperative programming languages is typically achieved using data structures, which are not available in Faust. First, a method for coding FDSs in a functional programming way is introduced, starting from previous works on mass-interaction models. Then, we draw a connection between FDSs and cellular automata, and exploit it for building a library that eases the implementation of FDS synthesis in Faust.

1. INTRODUCTION

Physical modeling has quite a long history in the field of sound synthesis. Over the years, many different techniques have been proposed [1], such as digital waveguides [2], mass-interaction models [3], modal synthesis [4] or finite difference schemes [5]. Even though sometimes more computationally demanding and difficult to control than other synthesis methods [6], physical modeling techniques offer many advantages. Indeed, creating a model of a vibrating system provides full control on its properties and, as a consequence, the output sound. These approaches theoretically allow us to synthesize natural and realistic sounds, tunable in every detail.

Faust [7] is a high-level, domain-specific, functional programming language, with a strong focus on the development of digital signal processing algorithms for sound and music. Faust code can be compiled with the Faust compiler, in order to directly generate standalone audio applications or plugins in different formats, or translate it to other languages such as C++, Java and others. The compiler includes options for automatically applying optimizations to the generated code. This feature is extremely useful for physical modeling synthesis, which requires fast code to perform big amounts of computation in real time.

1.1 Physical Modeling in Faust

Given the features stated above, its unique syntax, and the wide range of functions already available [8], Faust has been extensively used for implementing physical models. Smith [9] was among the firsts to exploit it in the context of physical modeling, by implementing simulations of a virtual electric guitar and various audio effects with digital waveguides. The latter, along with modal synthesis techniques, were also employed in the context of the Faust-STK [10], a collection of physical models based on some of the algorithms in the Synthesis ToolKit [11]. The Faust Physical Modeling Library was also recently implemented [12]. It contains models of various instrument parts that can be assembled together, and it introduces a new bi-directional algebra allowing for the implementation of coupling between the modules, at the cost of adding a one-sample delay. In addition, it formalizes a way to generate custom instrument parts by using `mesh2faust`, [1] a tool that performs finite element analysis on a 3D model and automatically generates a modal physical model.

In addition to modal and waveguide synthesis, mass-interaction physical models were also implemented in Faust: this technique consists of modeling physical systems in the form of lumped mass-spring networks [3]. Faust for mass-interaction was first explored by Berdahl and Smith with *Synth-A-Modeler* [13], a tool providing a high-level graphical environment to generate physical models by using a combination of mass-interaction and digital waveguides. More recently, Leonard et al, extending Berdahl's work, introduced *mi_faust*. This project contains the scripter MIMS [14], a high-level, graphical or command line tool that can be used to describe a physical model and automatically generate Faust code. [2] Along with the scripter, *mi_faust* includes `mi.lib`, a Faust library that can be used to assemble mass-interaction models directly in Faust in a modular way.

As seen above, several works employed Faust for developing physical models; however, to the authors' knowledge, Finite Difference Schemes (FDS) synthesis has never been integrated in this programming language. The implementation of FDS models with imperative languages is

[1] `ccrma.stanford.edu/~rmichon/pmFaust`
[2] `mi-creative.eu/tool_MIMS-Online_V2.html`

typically achieved using data structures such as vectors and matrices; which are not available in Faust. This might be a characteristic that has discouraged users interested in developing FDSs from trying to use this language. As a matter of fact, before this work, it was probably easier to code a FDS model directly in a lower-level language such as C++ rather than doing it in Faust. As such, this paper has two purposes: the first one is to introduce a routine for the development of FDS physical models in Faust, without making use of arrays; the second one is to provide a tool that allows for an easier implementation of such models.

2. FINITE DIFFERENCE SCHEMES

FDS synthesis consists in developing a full mathematical description of the system at hand, usually by employing partial differential equations, and then discretising the mathematical model using finite-difference time-domain (FDTD) methods, thus obtaining a finite difference scheme. This technique requires more computational power than other physical modeling methods such as the ones described in the previous section, and is more prone to numerical dispersion than, for example, digital waveguides [15]. However, it allows for a better spatial accuracy if frequency-dependent losses and dispersion are present [16] and it is more flexible, as FDTD methods do not make any assumptions on the system's solution.

FDTD methods essentially work by performing a discretization of the partial derivative operators. To do that, first a sampling grid for space and time has to be defined, thus we can write: $t = nk$ and $x = lh$ where $n \in \mathbb{N}$ and $l \in \mathbb{Z}$. The numbers k and h are the sampling steps of the system for time and space respectively; they are not independent and are bonded through a stability condition, which depends on the system equations. Given a system of PDEs in space and time with one-dimensional solution $u(x,t)$, it is possible to define the discrete function u_l^n which approximates it using the sampling steps above. Having defined a time grid, the time difference operators can be written as:

$$\begin{gathered}\delta_{t+}u_l^n = \frac{u_l^{n+1} - u_l^n}{k} \quad \delta_{t-}u_l^n = \frac{u_l^n - u_l^{n-1}}{k} \\ \delta_{t\cdot}u_l^n = \frac{u_l^{n+1} - u_l^{n-1}}{2k}\end{gathered} \tag{1}$$

These are the *forward*, *backward* and *center* difference operators respectively, which approximate the partial derivative operators. By combining them it is possible to obtain the definition for the second-order time difference operator:

$$\delta_{tt}u_l^n := \delta_{t+}\delta_{t-}u_l^n = \frac{u_l^{n+1} - 2u_l^n + u_l^{n-1}}{k^2} \tag{2}$$

The first and second order spatial operators are obtained in a similar way.

2.1 Example 1: 1-D Wave Equation

The discretization of the 1-D wave equation represents the simplest possible FDS. In the continuous case we have:

$$\ddot{u}(x,t) = c^2 \frac{\partial^2}{\partial x^2} u(x,t) \tag{3}$$

A FDS is given by:

$$\delta_{tt}u_l^n = c^2 \delta_{xx} u_l^n \tag{4}$$

If we expand the operators we obtain:

$$u_l^{n+1} = 2\left(1 - \frac{c^2k^2}{h^2}\right)u_l^n - u_l^{n-1} + \frac{c^2k^2}{h^2}\left(u_{l+1}^n + u_{l-1}^n\right) \tag{5}$$

2.2 Example 2: 2-D Wave Equation

The equation above can be extended in multiple space dimensions:

$$\ddot{u}(\mathbf{x},t) = c^2 \nabla^2 u(\mathbf{x},t) \tag{6}$$

if $\mathbf{x} \in \mathbb{R}^2$ a FDS can be written as:

$$\delta_{tt}u_{l,m}^n = c^2(\delta_{xx} + \delta_{yy})u_{l,m}^n \tag{7}$$

where l, m are the grid indexes in the two space dimensions. If the medium is isotropic then the two space sampling steps are equal: $h_x = h_y := h$. The operator expansion then yields:

$$\begin{aligned}u_{l,m}^{n+1} &= 2\left(1 - 2\frac{c^2k^2}{h^2}\right)u_{l,m}^n - u_{l,m}^{n-1} + \\ &+ \frac{c^2k^2}{h^2}(u_{l+1,m}^n + u_{l-1,m}^n + u_{l,m+1} + u_{l,m-1})\end{aligned} \tag{8}$$

The space-time dependencies (stencils) of equations (5) and (8) are depicted in Fig. 1.

Both schemes are linear and explicit; in fact the only unknown is one future point, which depends on a linear combination of the states of some spatial side points, itself and its delayed version. For more details on finite difference schemes, refer to Bilbao [5].

3. FDS IN FAUST

In imperative programming languages, FDSs are typically implemented using vectors and matrices: each time step is represented by a matrix of dimension equal to the space dimension. For instance, in the 2-D wave equation case, three 2-D matrices would be needed, for time steps $n+1$, n, $n-1$. The time states would then be updated at audio rate by cycling between the elements and applying the mathematical equations. Since all this is not possible in Faust, a different method had to be developed.

3.1 From Mass-Interaction to FDS

As before mentioned, in *mi_faust*, Leonard et al. introduced a library that allows for the implementation of mass-interaction physical models in Faust [14]. The `mi.lib`

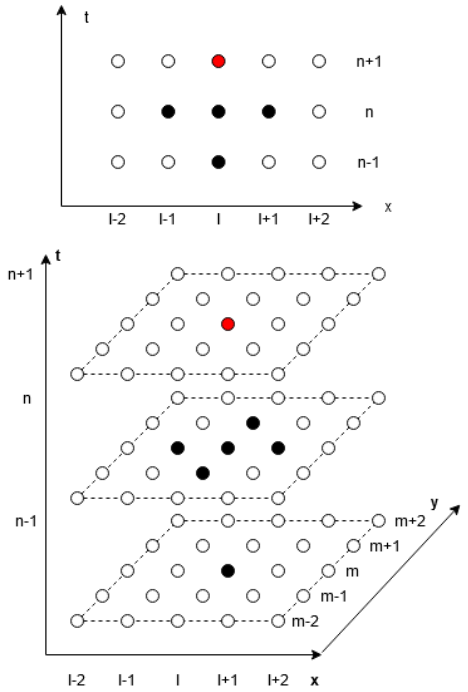

Figure 1. Stencils for the 1-D (top) and 2-D (bottom) wave equations. The red (top) point in each stencil is the unknown, the black points are the next points' states needed for the update equations

works this way: several *mass equation* blocks are stacked in parallel, followed in series by *spring-damper equation* ones; the number of these blocks is determined beforehand by the user when the mesh size is defined. The mass equations calculate the position of each mass given a force by discretising Newton's law: $\ddot{x} = F/m$. These are then fed into the dampened spring equations (from now on, *spring* will be used as a synonym for dampened spring), which calculates the force produced by each spring block depending on the position of the side masses, using Hooke's law and a linear damper equation: $F = -z(x_2 - x_1) - \sigma\frac{d}{dt}(x_2 - x_1)$, where z is the spring stiffness and σ the damping coefficient. The forces hereby calculated are then fed back into the mass equations, that output the new mass positions and so on. Each spring provides a force that only depends on two masses (the ones at the opposite sides of the spring) or one mass and a fixed point, which does not move. A mass position is then updated taking into account the forces coming from all the springs connected to it. The connections are previously defined by the user, depending on his mesh choices. Routing functions are used to bring the signals to the correct blocks.

If a uniform mesh is considered (namely, a mesh where each mass is connected with a spring to each neighbour, 2 in 1-D, 4 in 2-D, making up a string in 1-D or a regular matrix in 2-D), each mass receives a force from a constant number of springs, except the boundaries. In this case it is easy, by doing some algebraic calculations, to merge the mass and spring operations in one block. Given a time discretization as the one performed in section 2 and applying operator (2), Newton's law for a single mass at position 1 in the mesh becomes:

$$x_1^{n+1} = 2x_1^n - x_1^{n-1} + \frac{F_1^n k^2}{m_1} \tag{9}$$

This equation provides the horizontal position of m_1 at time step $n+1$. Using the backward operator (1), it is possible to discretize the force provided by the spring connecting m_1 to m_2, which will be applied to m_2:

$$\begin{aligned} F_{1\to2}^n = &-z_1(x_2^n - x_1^n) + \\ &- \frac{\sigma_1}{k}((x_2^n - x_2^{n-1}) - (x_1^n - x_1^{n-1})) \end{aligned} \tag{10}$$

And, for Newton's third law, the force applied to m_1 will be $F_{2\to1}^n = -F_{1\to2}^n$. If we consider a 1-D mesh, the total force applied to the single mass m_1 will be the sum of the forces provided by the two side springs: $F_1^n = F_{0\to1}^n + F_{2\to1}^n$. Therefore, it is possible to generalize equation (9) for a generic mass m_l not situated at boundaries:

$$\begin{aligned} x_l^{n+1} = &\, 2x_l^n - x_l^{n-1} + \frac{k^2}{m_l}\{-z_j(x_l^n - x_{l-1}^n) + \\ &- \frac{\sigma_j}{k}[(x_l^n - x_l^{n-1}) - (x_{l-1}^n - x_{l-1}^{n-1})] + \\ &+ z_{j+1}(x_{l+1}^n - x_l^n) + \\ &+ \frac{\sigma_{j+1}}{k}[(x_{l+1}^n - x_{l+1}^{n-1}) - (x_l^n - x_l^{n-1})]\} \end{aligned} \tag{11}$$

where z_j and σ_j are the stiffness and damping coefficient for the j-th spring.

If seen from another point of view, what we obtained here is the update equation for a linear, explicit FDS model. In fact, in equation (11) the future state (a horizontal position in this case) of a spatial point is given by a linear combination of the present and past states of itself and its *neighbours*. The fact that we started by considering masses and springs reflects here only in multiplications by the scalar coefficients m, z, and σ.

3.2 Faust Implementation

Having proved that there is a connection between mass-interaction and FDS, it is now possible to implement a FDS model in Faust, taking inspiration from the *mi_faust* approach. Using equation (11) we can merge the mass and spring blocks into one: now, each block will output its state, feed it back and receive in input the states signals from its neighbours and itself. The Faust syntax allows us to get the past versions of the states by simply using the delay operator `'`. The compiler automatically allocates the needed memory; therefore it is not necessary to implement multiple data structures for saving previous data, as it is in imperative languages. With this configuration we are also not limited to implement only equations (11), but whichever explicit update equation depending on a fixed number of neighbours' states. As an example, code listing 1 shows the Faust function for the 1-D wave equation (5):

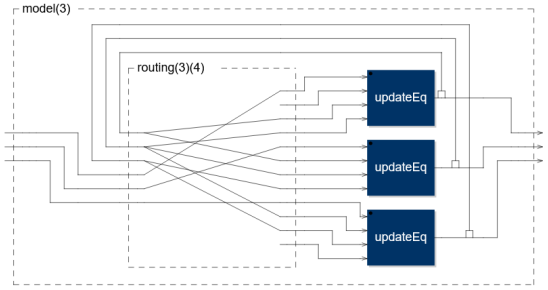

Figure 2. Block diagram for a 3-point-long 1-D wave equation scheme

```
lambda = c*k/h;
updateEq(fIn,u_w,u,u_e) =
   2*(1-lambda^2)*u-u'+lambda^2(u_w+u_e)+fIn;
```

Listing 1. 1-D wave update equation

This block takes as inputs the left and right neighbours' current states `u_w` and `u_e`, its own current state `u` and an external force `fIn`, and outputs the next state of the spatial point. Many blocks of this kind can be stacked in parallel to form meshes of the desired size (as done in listing 2): each one of these can be thought of as a single cell of the arrays that are usually implemented in imperative programming languages.

```
build1DScheme(nPoints)=par(i,nPoints,updateEq);
```

Listing 2. 1-D model builder function

As an example, Fig. 2 shows the Faust diagram for a 3-point-long 1-D wave equation scheme (the scheme was made very small in order to improve the image readability), coded by stacking in parallel the function in listing 1 three times. Here the signal paths described above are clearly visible. It can be noticed that the side blocks have an empty connection each; in fact, being at boundaries, there is no neighbour to take the signal from, so here they receive a zero value. It is possible to specify a block with a different equation that takes into account boundary conditions; if this is not done, as in this case, the zero signal automatically implies a clamped boundary condition. While the neighbour states are taken from the feedback loop, the force signals are open connections, these can be calculated outside the model and sent to the desired block with a selector function or an interpolator. This allows us to excite the mesh in the proper position.

The routing function is an essential part of the algorithm, and can be implemented using the Faust `route` primitive, which allows to drive the correct signals into the wanted spots in an optimised way. Code listing 3 shows the routing used for the algorithm in Fig. 2. Here `nPoints=3` is the number of blocks (points) stacked in parallel and `nInputs=4` is the number of inputs for each block. The primitive is designed so that the number of connections is 1-indexed: if a connection is numbered zero, or falls outside the maximum number of connections specified as an argument, a zero value is sent. This feature allows us to implement the "empty" connections for the boundaries shown above; the functions `F, W, C, E` take care of doing this.

```
routing(nPoints,nInputs) =
  route(nPoints+nPoints, nPoints*nInputs,
    par(x, nPoints, connections(x)))
with
{
  connections(x) =
    P(x)+nPoints, F(x),
    P(x), E(x-1),
    P(x), C(x),
    P(x), W(x+1);
  P(x)=x+1;
  F(x)=(1+0+(x*nInputs))*(x>=0)*(x<nPoints);
  W(x)=(1+1+(x*nInputs))*(x>=0)*(x<nPoints);
  C(x)=(1+2+(x*nInputs))*(x>=0)*(x<nPoints);
  E(x)=(1+3+(x*nInputs))*(x>=0)*(x<nPoints);
};
```

Listing 3. Routing function for 1-D wave equation.

The same algorithm structure can be employed for coding 2 or 3-D models. Listing 4 shows the Faust implementation for the 2-D wave update equation obtained in (8).

```
lambda = c*k/h;
updateEq(fIn,u_n,u_s,u,u_w,u_e) =
  2*(1-2*lambda^2)*u-u'+lambda^2*(u_e+u_w+u_n+
    u_s)+fIn;
```

Listing 4. 2-D wave update equation.

Again, many copies of this block can be stacked in parallel to form a 2-D mesh. Faust does not provide multi-dimensional structures such as matrices; therefore, this operation is not as straightforward as it was before. A nested for-loop can be simulated by nesting two par iterations, resulting in a piece of code that looks similar to what is used in imperative languages for parsing matrices:

```
build2DScheme(X,Y) = par(x,X,par(y,Y,updateEq));
```

Listing 5. 2-D model builder function

where `X` and `Y` are the total number of points in the x and y dimensions. Nevertheless, this algorithm will not form a 2-D structure; on the contrary, it will unroll it and build a 1-D block sequence with length `X*Y`, where consecutive rows are put one after the other. Hence, the code above is only useful to keep the double indexing convention.

Not having multi-dimensional structures is only a partial issue; in fact, with a proper routing, it is possible to drive the correct feedback signals into the right blocks. The routing function for the 2-D wave equation can be implemented as in listing 6:

```
routing2D(X, Y, nInputs) =
  route(X*Y*2, X*Y*nInputs,
    par(x, X, par(y, Y, connections(x,y))))
with
{
  connections(x,y) =
    P(x,y) + X*Y, F(x,y),
    P(x,y), S(x,y-1),
    P(x,y), N(x,y+1),
    P(x,y), C(x,y),
    P(x,y), E(x-1,y),
    P(x,y), W(x+1,y);
  P(x,y)=x*Y+y+1;
  F(x,y)=(1+0+(x*Y+y)*nInputs)*(x>=0)*(x<X)*(y
    >=0)*(y<Y);
  N(x,y)=(1+1+(x*Y+y)*nInputs)*(x>=0)*(x<X)*(y
    >=0)*(y<Y);
  S(x,y)=(1+2+(x*Y+y)*nInputs)*(x>=0)*(x<X)*(y
    >=0)*(y<Y);
  C(x,y)=(1+3+(x*Y+y)*nInputs)*(x>=0)*(x<X)*(y
    >=0)*(y<Y);
```

```
18   W(x,y)=(1+4+(x*Y+y)*nInputs)*(x>=0)*(x<X)*(y
       >=0)*(y<Y);
19   E(x,y)=(1+5+(x*Y+y)*nInputs)*(x>=0)*(x<X)*(y
       >=0)*(y<Y);
20 };
```

Listing 6. Routing function for 2-D wave equation.

Again, `X` and `Y` are the number of points in the two spatial dimensions, and `nInputs=6` is the number of inputs of each block.

Comparisons between these algorithms coded in Faust and the same versions in Matlab showed that the outcoming data was exactly the same; therefore it is possible to state that this method allows us to solve explicit finite difference schemes.

4. FDS LIBRARY

In the previous section, a method for implementing FDS physical models in Faust has been introduced and it was showed that what is usually achieved with array indexing can be equivalently accomplished in Faust by specifying signals routings. However, we can say without a doubt that coding FDSs this way is not as straightforward as it is in imperative languages. The main difficulty comes from the routing function, which can become very complicated for some models. In fact, the examples reported above are the simplest ones to implement, and things can become more intricate when a larger number of neighbours is needed. Moreover, these functions have to be re-written from scratch for each scheme. On the other hand, routing functions are not needed in languages where data structures are available, as they are replaced by explicit signal indexing. This makes Faust not competitive, at the moment, when it comes to implement FDS synthesis. For this reason, a second goal was set, consisting in the development of a library, `fds.lib`, allowing for a faster implementation of FDSs in Faust. Since this aims to be an introductory work and the intention was to give a coherent structure to the code, it was decided to only focus on linear and explicit schemes, leaving the implementation of other kind of simulations for future work. While it may seem a limitation, it has to be considered that linear schemes are sufficient for many cases of musical interest [5]; in fact, the majority of FDS models that run in real-time nowadays are of this kind, and Faust is specifically oriented towards the development of real time applications.

4.1 Cellular Automata

A cellular automaton (CA) is an algorithm that operates on a grid of cells, which can be in a finite number of states. For each cell, a set of cells is defined and called *neighbourhood*: at each time step t, the next state of a cell is determined by its present state and the state of its neighbours. The rule determining the new state is called *transition rule* and can be linear or nonlinear. The number of neighbours is defined by a coefficient r, called the neighbourhood *radius*; this indicates the number of cells at each side of the current cell that are taken into account. For instance, if $r = 1$ and the scheme is 1-D, the transition rule will depend on the current cell, one cell on the right and one on the left. Therefore, a neighbourhood is made of $(2r + 1)^D$ elements, where D represents the scheme dimension.

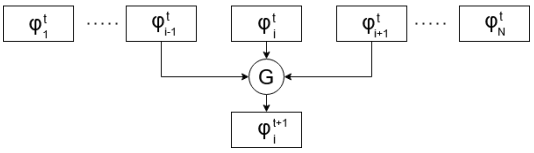

Figure 3. Scheme of a 1-D CA algorithm with transition rule G

If we consider a 2-D system, we can call the system's state at time t $\mathbf{\Phi}^t$; therefore the i-th, j-th single cell's state can be written as $\phi^t_{i,j}$. We can then define a transition rule G that brings $\mathbf{\Phi}^t$ to $\mathbf{\Phi}^{t+1}$ such that $G : \mathbf{\Phi}^t \mapsto \mathbf{\Phi}^{t+1}$. If G is linear we can write:

$$\phi^{t+1}_{i,j} = \sum_{\alpha=-r}^{r} \sum_{\beta=-r}^{r} a_{\alpha,\beta} \phi^t_{i+\alpha,j+\beta} \tag{12}$$

where $a_{\alpha,\beta}$ are the coefficients of the rule. A cellular automaton can be completely defined by its radius r and a transition rule; moreover, if the latter is linear, the coefficient matrix $\mathbf{A}$ and r are only needed. As an example, Fig. 3 depicts the scheme of a simple 1-D cellular automaton algorithm, made of N cells, radius $r = 1$ and transition rule G. For more details on cellular automata refer to [17, 18].

It is straightforward to identify a connection between FDSs and CA; in fact, several studies have been published on this topic [17]. Some of them have a general focus on the simulation of PDEs [19, 20], and others more particularly on the discretization of waveforms equations [21, 22]. Both FDSs and CA deal with an evolution of state variables on a discrete space-time grid, with the only difference being the fact that cellular automata operate on discrete states, while differential equations look for a continuous domain solution. However, in computer simulations numerical solutions are always discrete, since the processors' resolution is limited by the number of bits. Expressing a FDS as a CA might seem counter intuitive. In fact, the latter are mostly used with a small number of states (sometimes even 2), as they allow us to obtain complex behaviour from very simple rules [23], while in the case of PDEs simulation we deal with an enormous number of possible states. Nevertheless, the CA formalism allows us to obtain a standardized way for expressing FDS in Faust, as it will be shown in the next section.

4.2 The Library

The cellular automata formalism can help us coding standard routing functions for different space dimensions, that properly route the needed neighbours' signals relying only on the information provided by a predefined radius r. Moreover, as it was decided to focus on linear schemes, the transition function for the CA can be simply defined by some coefficient matrices. Contrary to cellular automata, a FDS usually depends on delayed versions of the neighbours' states; therefore, a *time coefficient* t had to be also

taken into account, which indicates how much steps back in time are needed (i.e., if $t = 1$ it means that the maximum delay needed for a neighbour state is 1 sample). With a little abuse of notation, we can express the operation performed by each cell (point) in a D-dimensional scheme as:

$$u_{\mathbf{x}}^{n+1} = \sum [\mathbf{A}_0 \odot \mathbf{N}_r(u_{\mathbf{x}}^n) + \\ + \mathbf{A}_1 \odot \mathbf{N}_r(u_{\mathbf{x}}^{n-1}) + \\ + \cdots + \mathbf{A}_t \odot \mathbf{N}_r(u_{\mathbf{x}}^{n-t}) \,] + F_{in} \tag{13}$$

where $\mathbf{x} = (x_1, \ldots, x_D)$ represents a spatial multi-index, depending on the scheme dimension D, $\mathbf{N}_r(u_{\mathbf{x}}^n)$ is a matrix containing the states of the neighbourhood of $u_{\mathbf{x}}$ at time step n, $\mathbf{A}_i$ are the coefficient matrices for each delayed version of the states, F_{in} is an external signal used to interact with the mesh, the operator $\odot$ represents element-wise matrix multiplication and the non-indexed sum indicates a summation over all the matrix entries. Both $\mathbf{N}_r$ and $\mathbf{A}_i$ contain $(2r+1)^D$ elements. The next paragraphs will detail how this approach can be used to build a FDS model with `fds.lib`.

4.2.1 Defining the Model

In order to code a new model, the user needs to provide: a neighbourhood radius r, a time coefficient t, the number of scheme points (size of the mesh) and the coefficient matrices relative to each point. The latter then need to be ordered in parallel to build a *coefficients scheme*, similarly to what was done in listings 2 and 5. This operation allows us to provide different coefficients for different scheme points, which is essential both for providing boundary conditions other than the clamped ones, and for building physical models with spatially-varying characteristics. As an example, listings 7 and 8 show the definition of the coefficient matrices for equations (5) and (8).

```
r=1; t=1; lambda=c*k/h;
A=2*(1-lambda^2); B=lambda^2; C=-1;
midPoint=B,A,B;
midPointDel=0,C,0;
leftPoint=0,A,2*B;
leftPointDel=0,C,0;
scheme(nPoints) = leftPoint,leftPointDel,
  par(i,nPoints-1,midPoint,midPointDel);
```

Listing 7. 1-D wave equation coefficient matrices

In this code a different coefficient matrix has been set to the leftmost point, in order to apply a Neumann free condition. Since the order of the points goes from left to right, the boundary condition is placed first inside the `scheme` function. Notice that the matrix for the non-delayed states is placed first; this order is very important for the correct functioning of the scheme.

```
r=1; t=1; lambda=c*k/h;
B = lambda^2;
A = 2*(1-2*lambda^2); C = -1;
midPoint = 0,B,0,
           B,A,B,
           0,B,0;
midPointDel = 0,0,0,
              0,C,0,
              0,0,0;
scheme(X,Y) = par (i, X,
  par(j,Y, midPoint,midPointDel));
```

Listing 8. 2-D wave equation coefficient matrices

In this case no boundary conditions have been specified; therefore clamped conditions are implied.

Looking at the code, a visual similarity between the coefficient matrices `midPoint` and `midPointDel` and the stencils in Fig. 1 might be identified. In fact, what we defined here are exactly the coefficients to be applied to the black points in the stencils. In the 2-D case, the CA neighbourhood is necessarily squared; therefore, zeros need to be placed at the corner points in this equation case, as seen in listing 8.

4.2.2 Model Construction

Once the coefficients are defined, the user can simply call `model1D` or `model2D` in order to obtain a fully working physical model. These functions take as inputs the number of points, the radius r, the time coefficient t and the coefficients scheme, and build a model with the same technique detailed in section 3. The built model will have open connections for the forces (or for a generic external signal), one for each scheme point, and will output each point's current state. Interpolation functions can be used in order to correctly select the zone of the mesh to excite or to read the signal from. Fig. 4 depicts the block diagram for a 3 points long version of the scheme in listing 7, built with `model1D`. This diagram will produce the same C++ code as the one represented in Fig. 2.

4.2.3 Interpolation

The library provides linear interpolation operators in 1 and 2 dimensions: `linInterp1D` and `linInterp2D`, which can be used to drive the force to the correct blocks. The index can be a float number varying at run time. These are essentially Faust implementations of the $J(x_i)$ operator, the linear interpolator described by Bilbao [5], not scaled by the spatial step, and work in a similar way as the Faust function `selectoutn` (included in the Faust *basics* library), except that they have the same number of input/output connections; and allow us to use float indexes. The library provides also stairs selectors functions, which only permit to use integer indexes: these are useful in case interpolation is not needed and require less computational power. All these functions are present also in an "out" version that sums all the outcoming signals together, in order to get a mono output signal.

4.2.4 Routing

The functions `route1D` and `route2D` are used to route the forces, the coefficients scheme and the neighbours' signals in the correct places. These are essentially versions of the functions reported in listings 3 and 6, modified to be automatically built starting from the values r and t, and to include also the coefficients matrices in the routing. The routing functions take as input, in this order: the coefficients block, the feedback signals and the forces. In return they provide for each scheme point (in order): the force signal, the coefficient matrices, and the neighbours' signals.

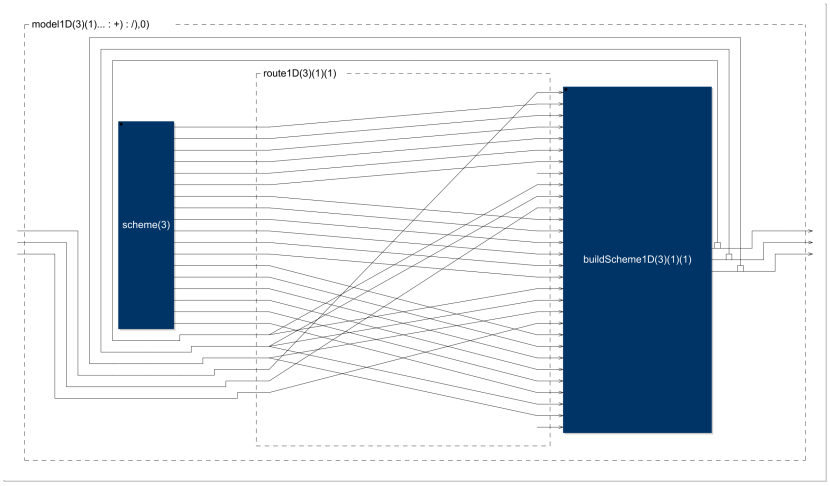

Figure 4. Block diagram for a 3 points long 1-D wave equation scheme, built with the library functions

4.2.5 Scheme Operations

As done previously in section 3, the actual equations are calculated inside some *scheme points blocks* stacked in parallel. The calculation is performed by the `schemePoint` function, which is built based on r, t and D. This function takes as inputs (in order): the force, the coefficient matrices and the neighbours' signals and outputs the next point state according to equation (13). The functions `buildScheme1D` and `buildScheme2D` are used to stack in parallel the proper number of scheme points.

4.2.6 Interaction Models

Even if only linear schemes were considered, nonlinear interaction models can be implemented. The library provides two force functions: a bow and a hammer. The first one is based on the formulation of the Helmholtz motion by Bilbao, the second one is modeled as a dampened oscillator interacting with the mesh in a nonlinear way [5]. Both models need to be coupled with the scheme, in particular, the hammer model involves solving another FDS (the oscillator) in parallel with the mesh. Coupling can be implemented by placing another feedback loop outside the model which drives back the mesh and the oscillator outputs; the interpolation functions can be used to pick the desired output signal from the mesh. Then, the two signals are used to calculate the force, which is subsequently sent as an input to the mesh and the hammer again. The hammer oscillator is integrated inside the hammer function, so that the two force models can be implemented with the same structure. The `bow` block takes as an input (in order): the bow velocity, the force scaling coefficient, the nonlinear parameter and the time step. The `hammer` block takes, in order, the force scaling coefficient, the oscillator frequency ω_0^2, the oscillator damping coefficient σ_0, the hammer stiffness parameter, the nonlinear parameter, the time step and the initial distance between the hammer and the mesh. Both blocks output a force times a scaling coefficient.

5. DISCUSSION & FUTURE WORK

The `fds.lib` library comes with a few examples serving as use-cases (in particular for what concerns the use of the interaction models) and ready-to-use virtual instruments. The modular structure of the library makes it convenient for different purposes. On one hand, the user interested in implementing standard linear schemes can simply write an equation with the desired numerical coefficients and use the model construction functions to easily obtain a working mesh. On the other hand, the library functions can be employed or modified individually to obtain a different result. For instance, the scheme point function could be easily adapted for calculating nonlinear equations, with the only constraint being that the nonlinearities would still need to depend only on the neighbors' states.

The 1-D schemes that were tested performed reasonably well: the CPU load, on an Intel i7-4710HQ processor, was always less than 10% even with a high number of points (schemes with up to 400 points compile and run even on the Faust online IDE). Nevertheless, more systematic performance tests need to be conducted in the future. The 2-D schemes, however, presented an issue. The generated C++ code is completely unrolled, resulting in very long algorithms. As a result, if the number of points is too high, the C++ compiler crashes, an issue similar to the one encountered by Barkati et al. [24]. Our experiments showed that the GCC compiler could not handle meshes with more than 20-by-20 points, which is not enough for many models. This is an issue inherent to the Faust compiler, which needs to be addressed in order to make Faust a complete language for coding FDSs. One way to solve this problem would be to make the compiler able to recognize the parallel structures inside the code and roll them up in the process of translation into C++. However, this topic needs to be more thoroughly investigated. Possible future work includes the implementation of initial conditions for the mesh points and, in the long run, support for implicit schemes.

6. CONCLUSIONS

This work aimed at introducing FDS synthesis in Faust. First, a routine for coding linear explicit physical models in a functional programming way was presented. Then, an overview on the connection between FDS and CA was provided, and the `fds.lib` was introduced: a library that takes advantage of this connection to ease the development FDSs. Together, the two tools allow users interested in implementing these models to exploit Faust potentialities in terms of code translation, optimization and wrapping system. The performances reported by 1-D schemes are promising; however, more methodical tests need to be performed. On the contrary, Faust presented several limitations for compiling 2-D models; these may be addressed by modifying the Faust compiler, making it able to recognize parallel structures.

7. REFERENCES

[1] G. De Poli and D. Rocchesso, "Physically-based sound modelling," *Organised Sound*, vol. 3, pp. 61–76, 04 1998.

[2] J. Smith, "Physical modeling using digital waveguides," *Computer Music Journal*, vol. 16, p. 74, 1992.

[3] C. Cadoz, A. Luciani, and J. L. 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.

[4] J. Adrien, "The missing link: modal synthesis," *Representations of Musical Signals*, 1991.

[5] S. Bilbao, *Numerical Sound Synthesis*. Chichester, UK: John Wiley & Sons, Ltd, 2009.

[6] S. Bilbao, C. Desvages, M. Ducceschi, B. Hamilton, R. Harrison-Harsley, A. Torin, and C. Webb, "Physical modeling, algorithms, and sound synthesis: The ness project," *Computer Music Journal*, vol. 43, no. 2-3, pp. 15–30, 11 2020.

[7] Y. Orlarey, D. Fober, and S. Letz, "FAUST : an Efficient Functional Approach to DSP Programming," in *New Computational Paradigms for Computer Music*, E. D. France, Ed., 01 2009, pp. 65–96.

[8] R. Michon, J. Smith, and Y. Orlarey, "New Signal Processing Libraries for Faust," in *Linux Audio Conference*, Saint-Etienne, France, 05 2017, pp. 83–87.

[9] J. O. Smith, "Virtual electric guitars and effects using faust and octave," in *Proc. 6th Int. Linux Audio Conf. (LAC2008)*, vol. 33, no. 3, Toronto, Canada, 01 2008, p. 1–10.

[10] R. Michon and J. 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 2011*, Paris, France, 01 2011, pp. 199–204.

[11] P. Cook and G. Scavone, "The synthesis toolkit (stk)," in *Proceedings of the ICMC*, Beijing, China, 10 1999.

[12] R. Michon, J. Smith, C. Chafe, G. Wang, and M. Wright, "The Faust Physical Modeling Library: a Modular Playground for the Digital Luthier," in *International Faust Conference*, Mainz, Germany, 07 2018.

[13] E. Berdahl and J. Smith, "Modular and open-source sound synthesis using physical models," in *Proceedings of the Linux Audio Conference*, Stanford, USA, 05 2012.

[14] J. Leonard, J. Villeneuve, R. Michon, and Y. Orlarey, "Formalizing mass-interaction physical modeling in faust," in *Proceedings of the Linux Audio Conference*, Stanford, USA, 03 2019.

[15] C. Erkut and K. Matti, "Digital waveguides versus finite difference structures: Equivalence and mixed modeling," *EURASIP Journal on Advances in Signal Processing*, vol. 2004, pp. 1–12, 06 2004.

[16] C. Erkut and M. Karjalainen, "Finite difference method vs. digital waveguide method in string instrument modeling and synthesis," *Proceedings of the International Symposium on Musical Acoustics (ISMA-02), Mexico City, 2002*, 12 2002.

[17] P. Narbel, "Qualitative and quantitative cellular automata from differential equations," *Lecture Notes in Computer Science*, vol. 4173, pp. 112–121, 10 2006.

[18] X.-S. Yang and Y. Young, *Cellular Automata, PDEs, and Pattern Formation*. Chapman & Hall/CRC, 09 2005, ch. 18, pp. 271–282.

[19] B. Strader, K. Schubert, E. Gomez, J. Curnutt, and P. Boston, "Simulating spatial partial differential equations with cellular automata." in *Proc. International Conference on Bioinformatics & Computational Biology, BIOCOMP*, Las Vegas Nevada, USA, 07 2009, pp. 503–509.

[20] G. Y. Vichniac, "Simulating physics with cellular automata," *Physica D: Nonlinear Phenomena*, vol. 10, no. 1, pp. 96 – 116, 1984.

[21] D. N. Ostrov and R. Rucker, "Continuous-valued cellular automata for nonlinear wave equations," *Complex Systems*, vol. 10, pp. 91–119, 1996.

[22] D. Barkley, "A model for fast computer simulation of waves in excitable media," *Physica D: Nonlinear Phenomena*, vol. 49, no. 1, pp. 61 – 70, 1991.

[23] S. Wolfram, "Cellular automata as models of complexity," *Nature*, vol. 311, pp. 419–424, 1984.

[24] K. Barkati, H. Wang, and P. Jouvelot, "Faustine: A vector faust interpreter test bed for multimedia signal processing," 06 2014, pp. 69–85.