REAL-TIME NOISE SYNTHESIS WITH CONTROL OF THE SPECTRAL DENSITY

Pierre HANNA, Anthony BEURIVÉ and Myriam DESAINTE-CATHERINE

SCRIME - LaBRI
Université de Bordeaux 1
F-33405 Talence Cedex, France
{hanna,beurive,myriam}@labri.fr

ABSTRACT

We propose in this paper a spectral synthesis model to generate noisy sounds with independent control parameters for spectral density and spectral envelope. Algorithms defining in a efficient way these spectral properties from the statistical parameters of the model are presented in details. Real-time implementation is composed of many methods which can be sequenced.

1. INTRODUCTION

1.1. Background

The existing models to synthesize noisy sounds are temporal or spectral models. Temporal models generate noises by randomly drawing samples using a standard distribution (uniform, normal, ...). Then they may be filtered (subtractive synthesis). The coefficients of this filter are the only control parameters. They are related to the *color* of the noise. The smoother the spectral envelope is defined, the more complicated the calculus of the coefficients is. In the same way, spectral models (for example [1]) are based on the Inverse-Fourier Transform and therefore propose only to modify spectral envelope.

We are interested particularly in spectral models to be homogeneous with other spectral models for harmonic sounds [2]. But we want to define other parameters than spectral envelope to control noisy sounds.

1.2. Spectral and statistic model

We introduce in this paper a synthesis model with statistical parameters in order to be able to modify other perceptually relevant properties like spectral density (defined as the ratio between the number of sinusoidal components N and the frequency bandwidth ΔF [3]).

We propose in [4] a statistical and spectral model. Sounds (sample rate F_e) are considered as random processes X. Each frequency component f_i is a random variable with fixed amplitude a_i and uniformly distributed phases ϕ_i:

$$X_k = \sum_{i=0}^{N} a_i \sin(2\pi f_i \frac{k}{F_e} + \phi_i) \qquad (1)$$

where the frequencies f_i are distributed in a band whose width is ΔF (Hz).

1.3. Parameters

Signal is defined from successive *frames*. Each frame is a sequence of parameters. The following statistical parameters are used to control the frequency distribution [5]:

- N is the number of frequencies which are randomly chosen in a band.
- $\Delta F = F_M - F_m$ is the bandwidth (Hz).
- M is the number of bins. Each bin length is $\frac{\Delta F}{M}$. No more than one frequency can be drawn in each bin ($N \leq M$).
- L is the width of the uniform distribution ($0.0 \leq L \leq 1.0$) in each bin.

2. NOISE CONTROLS

With this statistical parameters, users can control:

Bandwidth: By choosing F_m and F_M values, users can set the bandwidth of the synthesized noise. The band can be translated, broadened or narrowed.

Spectral density: The two parameters N and M allow the control of the spectral density of the sound [5]. $\frac{N}{\Delta F}$ has a maximum value over which no audible discrimination can be done (white noise). But if N decreases, our model leads to sounds which are perceptually different from usual filtered white noises.

M is also related to spectral density. If ΔF is small, increasing M increases intensity fluctuations. If ΔF is large, it improves the random characteristics of synthesized sounds. If $M = \infty$, synthesized sounds will be perceived as less dense, even if the number of sinusoidal components has not decreased.

Harmonicity: L is related to the harmonicity of the synthesized sound. If L is near 0, synthesized noise is quasi-harmonic (*machine* noises) because sinusoidal components are equally spaced. At the opposite, choosing L near 1 leads to *liquid*-like noises. This perceptual parameter cannot be modified using synthesis models based on filtered white noise.

Spectral envelope: Like the usual spectral models [2], spectral envelope can be directly controlled to modify the color of the synthesized noises.

3. IMPLEMENTATION

In order to test the real-time capabilities of the model, it was planned to implement it on one of the existing free software for real-time audio. The objective was to control all the synthesis parameters as fast as possible, while the sound is rendered. The first target was jMax ([6]) because we already had good experience with it and another sound model called SAS ([2]).

First of all, we needed to implement the model itself, independently of any other software. In other words, we needed a library. A name had to be found, and it became CNSS, for *Colored Noise by Sums of Sinusoids*. The library was named `libcnss` after it.

Eventually, the library had to be written so that it could be possible to wrap it with many other audio software (*clients*), be they real-time or non real-time. From the library point of view, the difference between these two kinds of clients essentially lie in the number of samples they want in a given amount of time. While a typical non real-time client would ask for a huge amount of samples (for instance, 10 seconds of samples) then never ask again, a real-time client asks for small amounts (say, 64 samples, which means 1.45ms at a rate of 44kHz), and often does that repeatedly.

To consider both kinds of clients means that we had to take care of the constraints of the most demanding, namely the real-time ones. These clients need efficient algorithms where most of the time spent should be spent on sample computation. Indeed, it is always good practice to develop efficient algorithms when possible, whatever the context. In our context, we already had fast synthesis algorithms inherited from the SAS model. These algorithms are able to compute *sine* incrementally with no memory access and very few processor instructions ([7]). It was straightforward to reuse them in the new library.

The experimental aspects of the model were also considered. The model can be viewed as several small components with different roles, bound together to produce noise. Some components deal with the size of the synthesis windows, while some others deal with the stochastic nature of frequencies or phases of the sinusoids. From time to time, new components were introduced to verify the audio consequences of new mathematical experiments. This was (and still is) challenging in a software engineering context where we want modular architectures that are clear and easy to extend.

In the current library, the following solution was adopted. All components are available to the client as CNSS *methods*. Table 1 enumerates the current available methods with short descriptions. Each method has its own synthesis parameters that can be controlled in real-time by the client. The client can build CNSS sound sources made of sequences of such methods. Thus, a sound source can be made with the sequence *[bins winsize nooffset ola]*. On that occasion, the parameters that the client can control are the ones offered by the methods in the sequence. For example, the *bins* method offers 6 parameters, among which we can find the *number of bins* that the frequency interval is split into, thus permitting to change this number during the real-time synthesis of the sound source.

4. DESCRIPTION OF METHODS AND ALGORITHMS

In this section we present the synthesis algorithms of the CNSS methods.

CNSS method	Description
uniform	Generates a set of sinusoids. All sinusoids have the same amplitude. Their frequencies are uniformly randomly drawn out of a frequency interval.
bins	Same as *uniform*, but frequencies are randomly drawn out of bins that split the frequency interval.
winsize	Sets the number of samples in the synthesis window.
dl	Randomly sets the number of samples in the synthesis window.
nooffset	Fills the synthesis window with samples computed by the forward synthesis of the set of sinusoids. Each sinusoid starts at the beginning of the synthesis window.
offset	Same as *nooffset*, but each sinusoid starts at a random offset from the beginning of the synthesis window, and is applied a triangular envelope.
ola	Overlaps consecutive synthesis windows, with fade-in and fade-out linear envelopes.
olasinus	Same as *ola*, but envelopes are sinusoidal.
phi	Randomly draws the phases of sinusoids..
silence	Fills the synthesis window with silence.
synchro	Setups the phases of sinusoids so that together they are at a maximum at a given percentage of the synthesis window's size. Eventually, they can also be randomly *unsynchronized*.

Table 1: *Current Implemented CNSS Methods*

4.1. Determination of frequencies

Frequency values of the sinusoidal components of each window have to be computed from the statistical parameters. Two methods are then proposed: *uniform* and *bins*.

4.1.1. Uniform distribution

In the *uniform* method, the number of bins is considered as infinite ($M = \infty$). Only the number of sinusoidal components and the width of the probability density function are input values and can be controlled. There is no need to choose any bin. Indeed the bin width is then null. Frequency are therefore drawn according to a uniform distribution between F_m and F_M:

$$f_i = W_{min} + \texttt{rand}(F_M - F_m) \tag{2}$$

where `rand` represents the *classical* random function which returns a pseudo-random real between 0 and the parameter of the function.

4.1.2. Drawing in bins

The *bins* method has three input values: the number of sinusoidal components N, the number of bins M and the width L of the probability density function.

The first step consists in defining M bins (denoted B_i) from the bandwidth values:

$$B_i = [i\frac{F_M - F_m}{M}; (i+1)\frac{F_M - F_m}{M}[\quad (3)$$

Then N frequencies are determined from these M bins. N bins have to be drawn from the M possibilities. A statistically correct algorithm to choose these bins is the classic algorithm to randomly define a permutation. One bin i is randomly drawn, then bins $M-1$ and i are interchanged. Another bin is randomly chosen from the $M-1$ last bins. This algorithm repeats until all N bins are chosen.

This algorithm has a cost if M is large compared to N because one large array has to be initialized and manipulated in each frame. But experiments show that defining M more than 1000 times N amounts to the *uniform* method previously described. So users would better use this one if M reaches a too large value in order to synthesize the perceptually same sound with less calculation time.

We could consider the special case where there is the same number of frequency bins than sinusoids ($N = M$). The calculus would be more efficient in that case, because we could directly associate sinusoid i with bin B_i ($i \in \{0, .., N-1\}$. But the fact that frequencies are stored in random order is important for other methods like *offset* (see section 4.5.5). So we don't consider that special case here. Furthermore, we believe that most of the time spent in `libcnss` is spent in partial synthesis, so improving the algorithm for that special case may not pay well enough.

Once the bins have been chosen, frequency values have to be determined from the parameter L. Another uniform draw is made in a band which is defined by the upper bound of the bin B_j ($j \in \{0, .., M-1\}$) and whose length is L multiplied by the bin length $\frac{F_M - F_m}{M}$ (see algorithm 1). :

$$f_i = F_m + (j + 1.0 - \texttt{rand}(L))\frac{F_M - F_m}{M} \quad (4)$$

Data : Frame parameters
begin
  for *each frame* **do**
    Define an array b of integers $\{0, .., M-1\}$ **for** $n \in 0, \ldots, N-1$ **do**
      Draw an integer k from the last $M-n$;
      Draw a real r in $[0; L]$;
      Calculate $f_n = F_m + ((1-r) + b[k]) * \frac{F_M + F_m}{M}$;
      $b[k] = b[M-n-1]$;
    end
  end
end

Algorithm 1: *general algorithm for determination of frequency values using the bins method.*

4.2. Determination of phases

The model of thermal noise described in [8] imposes on each component a phase to be uniformly distributed. At the opposite, noise synthesized with sinusoidal components with equal phases will result in noise with peaks of intensity. These peaks can be periodic depending on the length of the synthesis window. Such noises are described as *impulsive* noises. By changing the width of the probability density function of phases, users can control the amplitude of these peaks. By changing the length of the window size, users can modify the periodicity of these impulsions.

That's why we present in this section two methods to define the initial phases of each sinusoidal component and to take into account these properties.

4.2.1. Random phases

The *phi* method proposes to control the relative width P ($P \in \mathbb{R}$, $0 \leq P \leq 1$) of the probability density function of the phase:

$$\phi_i = \frac{\pi}{2} + \texttt{rand}(2\pi P) \quad (5)$$

If $P = 0$ all component's phases equal $\frac{\pi}{2}$. Thus all component's amplitudes sum together at the beginning of each temporal frame. If $P = 1$ all phases are uniformly distributed between 0 and 2π, so no intensity peak occur.

It is important to note that the intensity peak occurs at the *beginning* of each window. This implies that if users choose an overlap-add synthesis, this method becomes useless. Indeed the multiplication by the weighting window will reset the peak. That's why we propose another method giving users the choice of the time in the frame when the intensity peak occurs.

4.2.2. Synchronized random phases

The *synchro* method needs two parameters. The first one is the same as just above: the width P ($P \in \mathbb{R}$) of the probability density function of the phases. The second one is a relative value T_p ($T_p \in \mathbb{R}$ and $0 \leq T_p \leq 1$) which represents the instant in the window where the intensity peak occurs. Let $P = 0$: if $T_p = 0$ the peak is at the beginning of the synthesis window, if $T_p = 1$ the peak is at its end. Here is the equation to calculate the time τ from T_p and the synthesis window size W_s ($W_s \in \mathbb{N}$):

$$\tau = T_p \times W_s \quad (6)$$

This method must always be executed after the *uniform* or the *bins* method to use the frequency of the sinusoidal component. This frequency and the instant of the peak are thus known for each window. Each phase value are calculated from the equation:

$$\phi_i = \frac{\pi}{2} + \texttt{rand}(2\pi P) - \frac{2\pi f_i \tau}{F_e} \quad (7)$$

4.3. Determination of the amplitude of sinusoidal components

The amplitudes are simply defined from the frequency values and the spectral envelope. These parameters are the same as in the SAS model [2]. For now, it has not been implemented yet, but it will be in the next few weeks.

At the moment all sinusoidal components have the same amplitude. To avoid saturation each amplitude equals $\frac{1}{N}$. This value implies that general volume decreases when the number of components N decreases. Other solutions are now experimented to keep the volume independent of this number.

4.4. Additive synthesis

Once the frequency, amplitude and phase values are calculated, temporal samples are generated with additive synthesis. An efficient algorithm is presented in [7]. This algorithm can generate approximatively 2 partials per sample for each MHz of clock speed.

The algorithms we presented are being implemented to create a real-time sound synthesizer. The most CPU consumption is in the case of white noise (or filtered white noise). Synthesizing sounds with more sinusoidal components is useless, because no more difference can be heard by increasing N. That's why we define a maximum value for N depending on the synthesis window size. N cannot be greater than half of the synthesis window size (K in samples). This limit corresponds to the Inverse Fourier Transform technique:

$$N \leq \frac{K}{2} \tag{8}$$

For example, 128 sinusoidal components are needed to synthesize a 256 long white noise window. Furthermore the use of the OLA technique permits to diminish this number. As the OLA rate is always 0.5 to preserve the spectral density properties (the same number of components must compose sounds every sample), N can be divided by two. No more than 64 sinusoidal components have to be calculated to synthesize a white noise, which is very reasonable considering the speed of computers.

It is also important to observe that all other noises need less sinusoidal components and therefore less computation time.

4.5. Successive synthesis windows

We propose five methods to synthesize the successive windows. They are illustrated in figure 1. Some of them are efficient but may introduce distortion or clicks depending on the type of sounds produced. All the effects of these methods are detailed in [9].

4.5.1. No overlapped windows

The first method is the most trivial, but gives poor results. It consists in setting the window size constant and synthesizing successively one window after the other. This method is the most efficient but may lead to audible artifacts in the synthesized sounds at the boundaries of each window. One can hear periodic distortions.

4.5.2. Variation of the length of the synthesis window

The *dl* method proposes to vary the length of the synthesis window size W_s to avoid the periodic clicks. An input parameter Δ_l ($\Delta_l \in \mathbb{R}and0 \leq \Delta_l \leq 1$) gives the percentage of random size. The second input parameter is the mean size W_s ($W_s \in \mathbb{N}$). If $\Delta_l = 0$ the window size w_s is constant ($w_s = W_s$), whereas if $Delta_l = 1$ the window size w_s is randomly drawn according to a uniform distribution between 0 and W_s:

$$w_s = W_s - \Delta_l \times (1.0 - \texttt{rand}(2.0)) \times W_s \tag{9}$$

These variations improve the quality of sound by avoiding the periodicity due to the clicks.

4.5.3. Overlap-add

The *ola* method uses the *classical* overlap-add method. Each successive window is multiplied by a weighting window w. The OLA window is the Bartlett one (triangular). The resulting signal can be written as:

$$x[n] = \sum_{l=0}^{L-1} s_l[n - lH]w[n - lH] \tag{10}$$

where L is the number of frames s_l ($0 \leq l < L$) and H is the hop size (or the time advance).

The input parameter of this method is the relative size of the overlapped region $\frac{1-H}{W_s}$. It is limited to $[0; 0.5]$. In the worst case ($H = \frac{W_s}{2}$) this method may need to synthesize two times more windows than the previous ones.

We show in [9] that this usual OLA technique cannot sometimes be used for our synthesis method. Indeed multiplying each frame by a weighting window modifies the statistical properties of the resulting signal. That's why we propose two other methods to adapt the OLA technique to noise synthesis.

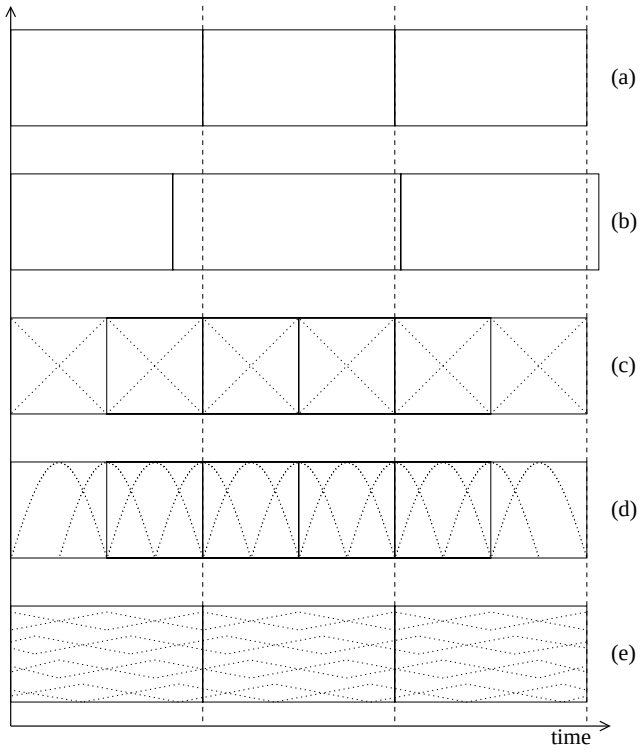

Figure 1: *Illustration of proposed additive methods: (a) constant synthesis window size (b) random window size (c) synthesis with overlap-and-add (Bartlett window) (d) olasinus (e) offset*

4.5.4. Sinusoidal window

The *olasinus* method is based on the overlap-add synthesis. But it differs from the previous one by using a sinusoidal weighting window. This window is defined as, for $n \in [0; W_s[$:

$$w[n] = \sin(\frac{2\pi n}{W_s - 1}) \tag{11}$$

There is no input parameter because the hop size H is set to 0.5. Other values are useless because they may imply distortions in the synthesized sounds.

4.5.5. Component offset

The *offset* method consists in time shifting each sinusoidal component in every window:

$$x'(n) = \sum_{l=0}^{L-1} \sum_{k=0}^{N-1} s_l^k(n - lH - d_k)\, w(n - lH - d_k) \qquad (12)$$

Experiments show that the time offset can always be the same for each sinusoidal component and does not have to be drawn or calculated for each frame. However this method needs more calculations because each partial has to be independently multiplied by a weighting window before being added to the sound. That implies one more multiplication per partial and per sample. The window is a Bartlett one.

This method takes into account properties of phases. Random phases ϕ_i^0 are phases at the beginning of windows. New phases are computed with offset values:

$$\phi_i = \phi_i^0 + \frac{2\pi f_i d_i}{F_e} \qquad (13)$$

5. SOFTWARE DESCRIPTION AND EXAMPLES OF METHOD SEQUENCES

As said above, we chose jMax as our first client for the library. An extension of jMax, `cnss4jmax`, has been developed to wrap some of the data types found in the library to the following object types on the jMax side:

- `cnss-methods`: objects of this type are able to output a list of the available methods in the library, as well as describe each of these methods in terms of functionality and controllable parameters.
- `cnss-frame`: objects of this type are placeholders for synthesis parameters related to sequences of CNSS methods.
- `cnss-source~`: objects of this type are CNSS sound renderers, meaning that they synthesize the samples according to their sequence of methods and the incoming frames containing synthesis parameters.

Figure 2 is a snapshot of a jMax patch containing CNSS objects. In this figure, we can see that a `cnss-frame` related to a CNSS method sequence *[bins dl]* permits the control of 8 parameters. The frame is connected to a single `cnss-source~` with method sequence *[bins dl nooffset ola]*, thus controlling the algorithm used in the synthesis of the source. The source itself is connected to a `dac~` object that represents the left and right outputs of the sound card.

The order in the sequence is very important. A permutation in a sequence may cancel some methods. For example, the sequence *[bins winsize synchro nooffset]* synthesizes sound with sinusoidal components whose frequencies are randomly chosen and whose phases can be calculated to produce an intensity peak in the temporal window. But if the *synchro* and the *winsize* method are permuted, the phase cannot be synchronized. Indeed the *synchro* method needs the synthesis window size.

Here are some other examples of sequence of methods to synthesize sounds:

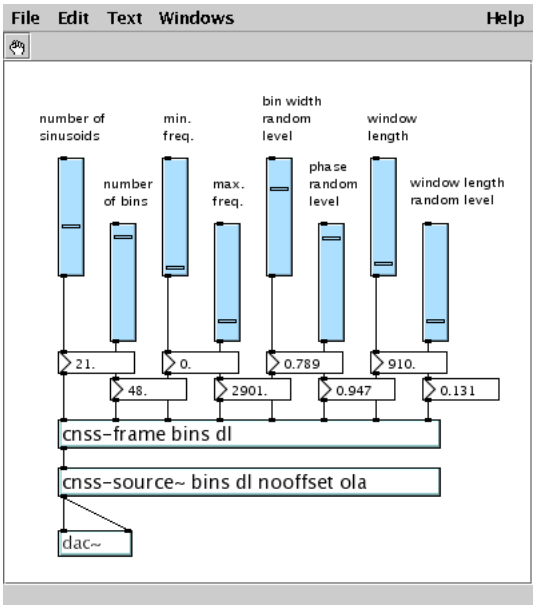

Figure 2: *Snapshot of a jMax Patch with CNSS Objects*

- *[uniform phi winsize nooffset]:*
  This example is a basic sequence. Random sinusoidal components are synthesized in successive windows. The only parameters are the number of components , the width of the probability density function of the phase and the window size.
- *[bins winsize dl synchro nooffset ola]:*
  This example shows that two synthesis methods can be associated. Here windows are overlapped *and* their lengths are randomly chosen. However all synthesis methods cannot be used together, for example *offset* and *ola* method.
- *[bins winsize dl synchro offset]* and *[bins winsize dl synchro nooffset olasinus]:*
  These two methods are the most useful ones to synthesize noise. They permit to control 9 parameters and thus lead to noise with many different properties.

6. FUTURE WORK

At that point, the implementation is fast enough to experiment effective real-time synthesis of many sounds in the model. On an "old-class" pentium III 500Mhz running GNU/Linux and jMax, we had no problem in the control and real-time synthesis of a few hundreds of sinusoidal components and relatively small synthesis windows. Our future experiments will lead us to several sound sources on different frequency ranges, probably requiring the synthesis of more than 500 sinusoidal components in real-time. But this certainly can be already achieved with up-to-date computers.

Anyway, this is still work in progress. While the algorithms computing the sinusoids are fast enough, it is quite obvious that other parts of the library could be improved to better suit the constraints of real-time audio software. And the fact that we want to control more than one frequency range with different parameters is a sign that the model has to be extended in new directions. The future will tell.

This implementation is mainly a research tool. Polyphony can be used to synthesize noises composed with different bands and different spectral properties. It may permit us to make psychoacoustic studies about the perception of bands of noise with different spectral densities.

The `libcnss` library and its jMax extension are free software developed on the GNU/Linux platform. They are (or soon will be) available on the website of the SCRIME ([10]). The code is currently in heavy development. It is likely that the available methods have been improved since the writing of this paper.

7. ACKNOWLEDGMENTS

This research was carried out in the context of the SCRIME[1] project which is funded by the DMDTS of the French Culture Ministry, the Aquitaine Regional Council, the General Council of the Gironde Department and IDDAC of the Gironde Department.

SCRIME project is the result of a cooperation convention between the Conservatoire National de Région of Bordeaux, ENSEIRB (school of electronic and computer scientist engineers) and the University of Sciences of Bordeaux. It is composed of electroacoustic music composers and scientific researchers. It is managed by the LaBRI (laboratory of computer science of Bordeaux). Its main missions are research and creation, diffusion and pedagogy thus extending its influence.

8. REFERENCES

[1] X. Serra and J. Smith, "Spectral modeling synthesis: a sound analysis/synthesis system based on a deterministic plus stochastic decomposition," *Computer Music Journal*, vol. 14, no. 4, pp. 12–24, 1990.

[2] S. Marchand, *Sound models for computer music: analysis, transformation, synthesis of musical sound*, Ph.D Thesis, LaBRI, Université Bordeaux I, 2000.

[3] A. Gerzso, "Density of spectral components: preliminary experiments," report, Ircam, 1978.

[4] P. Hanna and M. Desainte-Catherine, "Statistical approach for sound modeling," *Proceedings of the Digital Audio Effects Workshop (DAFX'00, Verona, Italy)*, pp. 91–96, 2000.

[5] P. Hanna and M. Desainte-Catherine, "Influence of frequency distribution on intensity fluctuations of noise," *Proceedings of the Digital Audio Effects Workshop (DAFX'01, Limerick, Ireland)*, pp. 125–129, 2001.

[6] "jmax," `http://www.ircam.fr/jmax`.

[7] R. Strandh and S. Marchand, "Real-time generation of sound from parameters of additive synthesis," *Proceedings of the Journées d'Informatique Musicale (JIM'99, Paris)*, pp. 83–88, 1999.

[8] W.M. Hartmann, *Signals, Sound, and Sensation*, Modern Acoustics and Signal Processing AIP Press, 1997.

[9] P. Hanna and M. Desainte-Catherine, "Noise synthesis using ola technique and sinusoidal component offset," *submitted to DAFX'02*, 2002.

[10] "Scrime," `http://www.scrime.u-bordeaux.fr`.

[1] Studio de Création et de Recherche en Informatique et Musique électroacoustique