PARAMETERS ESTIMATION OF FEEDBACK DELAY NETWORK REVERBERATION WITH ARTIFICIAL INTELLIGENCE ALGORITHMS

Alessandro Cerioli
Aalborg University
acerio19@student.aau.dk

ABSTRACT

The reverberation is one of the most famous and well-known acoustic phenomena, which consists in the propagation of a sound, produced by a source, across an environment, which could be a room or the sound box of a musical instrument such as a guitar, a violin or a piano. In particular, in this paper different approaches will be used in order to estimate the reverberation parameters given the impulse response of the sound box of a guitar. Such approaches are based on genetic algorithm and gradient descent algorithm, which results were then integrated into a hybrid reverberation approach. At the end, the results were implemented in Matlab and SOUL programming languages and the properties of the two different approaches are compared and analysed.

1. INTRODUCTION

The reverberation is that acoustic phenomena for which sound propagates across a room or the sound box of an instrument. The "collisions" against obstacles (such as walls and objects) create a modification of the properties of the sound, because they can absorb part of the sound, creating an attenuation in the amplitude as most evident consequence, but also a change in the timbre, since different materials tend to absorb better different ranges of frequencies (generally high frequencies are absorbed faster than low ones) [1].

As can be observed in figure 1, the reverberation is composed essentially by three main stages [1,2]. The first stage is the direct sound, or rather the time that the sound takes to go from the source to the listener. Generally, it is the most powerful part of the reverberation, because it doesn't collide against obstacles, the only attenuation is given by the mean across it travels, usually the air, that can be often considered not so relevant. Obviously, it is the first component that reaches the listener, within a time T_0.

$$T_0 = \frac{d}{c} \tag{1}$$

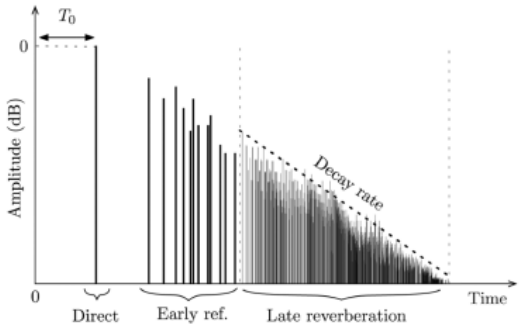


Figure 1. Structure of a reverberation over the time. It is clearly possible to notice that it is subdivided in three stages: direct reverberation, early reverberation and late reverberation. Anyway, often the boundary between early reverberation and late reverberation is not very well defined.

where T_0 is the time the direct sound takes to go from the source to the listener, d is the distance between the source and the listener and c is the sound propagation speed (around 330 m/s in the air). The second stage is the so-called "early reflection" and it is characterized by the soundwaves that reach the listener after a relatively small number of reflections (generally, second and third order of reflections are considered). Since the number of collisions is small, these soundwaves tend to maintain the properties of the sound generated by the source, in other words their frequency components are not too much attenuated, so the sound tends to be still loud and with the original timbre, moreover it is possible to perceive the direction from which they arrive, since the amount of soundwaves reaching the listener is not so dense yet. Finally, the late reverberation is the last part of the reverberation phenomena, so, in this stage, the soundwaves that reach the listener are very dense (as a result, it is impossible to distinguish clearly the direction) and they are subjected to a high number of collisions and reflections against walls and objects. Consequently, this stage results much more attenuated than previous ones and the timbre tends to be different as well. A particularly meaningful property of the reverberation is the T60, or rather the time the reverberation takes from the end of the steady part of the sound (at 0 dB) to the instant of time in which it decays below -60 dB (figure 2).

The most famous technique for reverberation reproduction is undoubtedly by means of convolution with the impulse response of the system (or rather, the room or, as in

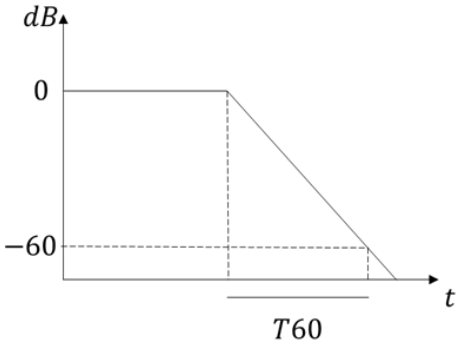


Figure 2. Graphical representation of the T60 of a signal. The first section is the steady part and the period of time between the end of the early portion and the moment in which the signal decays under $-60dB$ is called T60.

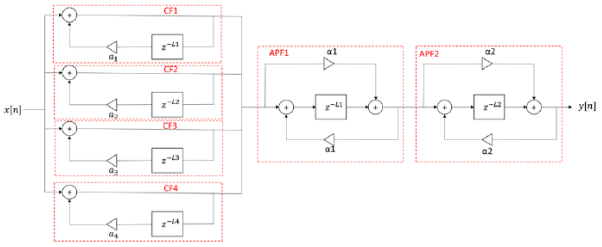


Figure 3. Scheme of the Schroeder's model: the input signal is passed into four comb filters in parallel and, successively, the result is passed through two all pass filters in series.

this case, the sound box of the instrument). As a matter of fact, there is a family of systems called "LTI systems" (Linear Time Invariant systems), which impulse response contains all the information of the same system, with a consequent very accurate reverberation reproduction. A given system is defined as LTI when the linear and time invariance properties are true [3]:

$$T(\alpha\, x[n]) = \alpha T(x[n]) = \alpha y[n]$$

$$x_3\,[n] = x_1\,[n] + x_2\,[n] \implies$$
$$T\,(x_3\,[n]) = T\,(x_1\,[n] + x_2\,[n]) =$$
$$= T\,(x_1\,[n]) + T\,(x_2\,[n]) = y_1\,[n] + y_2\,[n] = y_3[n]$$

$$y_1\,[n] = x_1\,[n] = x\,[n - L] = y[n - L]$$

(2)

Respectively, scalability, superposition and time-shift invariance properties. Scalability and superposition define the linearity property. Consequently, its impulse response is simply the output of the system given an impulse (Dirac delta function) as input.

$$\delta(n) = \begin{cases} 1, & \text{if } n = 0 \\ 0, & \text{otherwise} \end{cases}$$

(3)

The impulse response of the room or of the sound box can be then convolved with and input signal to create the reverberation effect. In our case, the impulse response of the sound box of a guitar was used, so that the convolution with a plucked string as input signal can reproduce the effect of reverberation of the sound box.

$$y[n] = x[n] * h[n] = \sum_{m=-\infty}^{\infty} x[n] \cdot h[n - m] = \sum_{m=-\infty}^{\infty} h[n] \cdot x[n - m]$$

(4)

Even if this approach brings to very accurate results (since impulse response is measured from a real world environment) maintaining all the nuances, it is actually not well suited for real-time applications because convolution is very onerous from the point of view of the computation. Moreover, the parametrization is very difficult, consequently just few changes bring to a relevant amount of work in order to adjust the model [1]. This is the reason for which, in the last century, many attempts to reproduce reverberation by means of algorithms were accomplished [4]. In particular, the most relevant one was Schroeder's model, that was conceived in the '60, with which he tried to reproduce the reverberation by using four comb filters in parallel and two all pass filters in series in order to simulate respectively echo density and attenuation and diffusion of the signal [5] (figure 3).

During the following decades, another model progressively replaced Schroeder's model: the feedback delay networks, that are essentially vectorized comb filters. They are composed of delay lines to simulate the echo density, which outputs are then filtered by lowpass filters to simulate the attenuation and the quicker absorption of higher frequencies of the sound. Successively, the outcome acts in retroaction with a feedback matrix to simulate the damping. In this experience, in particular, a four lines Jot's feedback delay network was used [6] (figure 4). These algorithmic reverberations are more flexible and efficient than convolution, but on the other hand they are much less accurate as well. Then, feedback delay networks are more suited to simulate the dense and flat late reverberation than the early reverberation.

Consequently, a good compromise between the two models is trying to mix them together with a hybrid model [1, 7]. It consists in using the convolution to represent the early reverberation and the feedback delay network to simulate the late reverberation (see figure 5). In this paper, two different algorithms will be used to estimate the parameters of the feedback delay network and then the results will be compared.

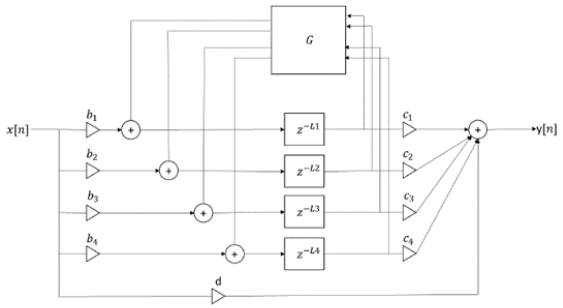


Figure 4. Jot's four lines feedback delay network. The input signal is multiplied for different coefficients (b_0, b_1, b_2 and b_3) and the result is summed with the output of the damping matrix. Successively, the delayed output is the input for the feedback matrix and it is multiplied for other coefficients (c_0, c_1, c_2 and c_3). The resulting signals are summed to generate the output of the network.

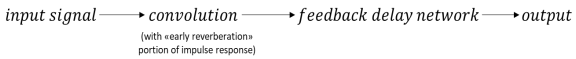


Figure 5. Hybrid reverberation structure. The first portion of the signal is convolved with impulse response for early reverberation representation, the second portion of the signal is instead sent in input to the feedback delay network for the representation of late reverberation. Finally, the early and late portions are attached together to get the output.

2. SIMULATION MODALITIES

Concerning the modalities of the simulations of reverberation, two programming languages were used for this purpose: Matlab [8] and SOUL [9]. Matlab is a general purpose programming language, consequently very suited for tasks of analysis and estimation of parameters. Nevertheless, a model to hear the final result was implemented in this language as well. Conversely, SOUL is a strongly specialized new programming language for audio modelling and synthesis of sounds. As a result, the feedback delay networks with estimated parameters were implemented in such language, while any kind of analysis of the signal was not possible. In SOUL was not possible to develop efficiently a convolution operation, neither in the early portion of the impulse [10]. Therefore, the final result of hybrid reverberation was exclusively developed in Matlab.

3. PLUCKED STRING SIMULATION

The aim of this experience is simulating the reverberation of the sound box of a guitar, consequently for the final simulation (both in Matlab and SOUL programming languages) was used the algorithm model of a string obtained by means of Finite Discrete Schemes (FDS) [11], to which was successively added the reverberation. The difference equation is:

$$u_{tt} = c^2 u_{xx} - k_e u_{xxxx} - 2\sigma_0 u_t + 2\sigma_1 u_{txx} \tag{5}$$

Where u is the transversal displacement of the string, depending on position over the same string (x) and the time (t). As a matter of fact, in this notation, for example u_{tt} represents the second order derivative with the time dimension.

$$c = \sqrt{\frac{T}{\rho A}} = \sqrt{\frac{T}{\rho \pi r^2}} \tag{6}$$

Where c is the wave speed, T is the tension of the string, ρ is the density of the material of the string, A is the cross-section area of the string and, consequently, r is the cross-section radius of the string. Moreover, σ_0 is the damping frequency-independent coefficient, σ_1 is the damping frequency-dependent coefficient and k_e is the stiffness coefficient. Proceeding with extension rules, the result is:

$$\begin{aligned} u_l^{n+1} &= 2u_l^n - u_l^{n-1} + \tfrac{c^2k^2}{h^2}(u_{l+1}^n - 2u_l^n + u_{l-1}^n) + \\ &- \tfrac{k_e^2k^2}{h^4}(u_{l+2}^n - 4u_{l+1}^n + 6u_l^n - 4u_{l-1}^n + u_{l-2}^n) + \\ &- 2\sigma_0 k(u_l^n - u_l^{n-1}) + 2\sigma_1 \tfrac{k}{h^2}(u_{l+1}^n - 2u_l^n + u_{l-1}^n + \\ &- (u_{l+1}^{n-1} - 2u_l^{n-1} + u_{l-1}^{n-1})) \end{aligned} \tag{7}$$

Where k is the discrete space step and h is the discrete time step. The stability condition of the system is the following:

$$h = \sqrt{\frac{c^2k^2 + 4\sigma_1 k + \sqrt{(c^2k^2 + 4\sigma_1 k)^2 + 16k_e^2k^2}}{2}} \tag{8}$$

4. SOUL PROGRAMMING LANGUAGE

4.1 Introduction to SOUL

SOUL is a new programming language strongly specialized for audio modelling and synthesis. As a matter of fact, it was designed to make easier the synthesis the sound but it is not possible to analyse the sound. Consequently, for this purpose other non-real time programming languages

are more suited. SOUL is a programming language essentially based on procedural paradigm, in which different modules called "graphs" containing processor units are connected together to form a net to model the sound. The data is passed from one module to another by means of streams: it is possible to define input events to control the input signals and output streams that are related to the data that leaves the module after the internal elaboration.

4.2 Type of variables

SOUL presents different types of primitive data:

- int: it is a type for integer values stored in 32 or 64 bits.
- $int32$: it is a type for integer values stored in 32 bits.
- $int64$: it is a type for integer values stored in 64 bits.
- $bool$: it is a type for boolean values.
- $float$: if is a type for floating point values stored in 32 or 64 bits.
- $float32$: it is a type for floating point values stored in 32 bits.
- $float64$: it is a type for floating point values stored in 64 bits.
- $wrap < size >$: it is a type for an integer constrained into a compile-time established range. The increment of the variable's value outside such range implies a wrapping (with an increment above the upper bound, the value is set to the lower bound; with a decrement below the lower bound, the value is set to the upper bound).
- $clamp < size >$: it is a type for an integer constrained into a compile-time established range. The difference with wrap is that instead of the wrapping operation, there is a clamping operation when the value is outside the range(with an increment above the upper bound, the value does not overcome the upper bound; with a decrement below the lower bound, the value is constrained to the lower bound value).
- $string$: the string type has the main function of debugging, since it is not possible to accomplish runtime operations.
- let: the type of this variables is defined at the moment of the initialization and they are set as constant.
- var: as in the case of let variables, the typed is defined at the moment of the initialization, but it is not constant.

A variable can be set as $const$, so that won't be possible to change its value after the initialization.

4.3 Input event

The input events make possible to the user to control the parameters of the model. As a matter of fact, such events are declared as variables that can be modified by a factor external to the module. The input events values can be modifyed by sliders automathically generated by SOUL's environment. The declaration of input events is generally inside an input event block, but can be defined also in a single line of code. The declaration can be "bare", or rather without any parameter definition (even if, in such case the definition of the parameters is delegated to the input events of other graphs which are connected to the input events of the current graph. Infact, it is not possible to define the input events without the specification of the parameters in any point of the program) or with the definition of the parameters. An easy example of definition of an input event is the following:

```
input event {
  float amplitudeIn [[ min: 0, max: 1,
    init: 1, name: "amplitude", unit: "dB",
    step: 0.01 ]];
}
```

This input event is useful to control the dB of the amplitude of a signal. it is possible to notice how "min", "max" and "init" fields define the range in which it is possible to modify the variable. The "name" is useful as (not univocal) ideftifier, "unit" to communicate the user the unit of measurement used to calibrate the variable and, finally, the field "step" defines the minimum variation of the input event variable's value. An event method is associated to each input event declared. This is particularly useful, since in this way it is possible to define the portion of code that will be executed whenever the user will change, by means of the slider, the value of the variable. An example of event method for the previous input event can be the following:

```
event amplitudeIn (float amp) {
  amplitude = amp;
  // amplitude is the state variable
  // that controls the amplitude of
  // the signal inside the run method.
}
```

4.4 Streams

The streams are particular instances of variables that are not directly manipulated by means of events. In fact, the input streams can be considered the "input ports" of the graph (or processor) and the output streams as the "output ports". The output port of the instance of a graph can be connected to the input port of another instance of a graph inside the connection block of a graph. As in the following example of code:

```
graph main [[ main ]] {
  ...
```

```
  connection {
    ...
    sine1.output  > sum.input1;
    sine2.output  > sum.input2;
    ...
  }
}
```

The output can be sent on the output stream by means of << operator inside the *loop* block of the processor's run method. Since it is possible having more than one output port and, consequently, more than one ouput stream, it is possible to use << operator more than once on the different output streams defined in the processor. Here a brief code of explanation:

```
processor processor_name {
  ...
  output stream float output1;
  output stream float output2;
  ...
  void run() {
    loop {
      output1 << ... ;
      output2 << ... ;
    }
  }
}
```

4.5 Structure

The structure of the programming language is strongly inspired by a tree graph model, where the "leaves" are defined as "processors" and the "branches" as "graphs". The processors are the blocks of code related to the actual audio synthesis and data elaboration and the general structure is shown in the following pseudo-code:

```
processor processor_name {
  // input streams declarations ...
  // output streams declarations ...

  input event {
    // input events declarations...
  }

  // variables declarations...

  void run() {
    loop {
      // audio synthesis ...
      advance();
    }
  }
}
```

The declaration of input and output streams are generally set at the beginnig portion of the block and they represent the input a output gates of the processor, respectively. The run method is the portion of code run for the audio synthesis task, moreover the "loop" block is the portion continuously iterated over the time. Here is proposed an easy example of processor, which output is the sum of two input signals:

```
processor adder {
  output stream float audioOutput;
  input stream float input1;
  input stream float input2;

  void run() {
    loop {
      audioOutput << input1 + input2;
      advance();
    }
  }
}
```

The graphs are instead the nodes that allow to connect together different processors or also different graphs (as a result, in this case the graph is the root of a tree substructure). Hence, graphs are not units where audio synthesis is specifically accomplished (as a matter of fact, there is not the run method), but they are useful to define the structure of the net that connect the different processors, which actually generate the sound. One convenient structure of the graph is the following:

```
graph graph_name {
  // input streams declarations...
  // output streams declarations...

  input event {
    // input events declarations...
  }

  let {
    // variables declarations...
  }

  connection {
    // connections...
  }

}
```

Also in this case, it is possible to define the input and output streams and the input events at the beginning. The connection block is a fundamental portion of code typical of the graph and it allows to connect the different units by means of their input streams, output streams and input events. Here is proposed an easy example of a graph to sum two different sinusoids, connecting the two sinusoid processors to a summer processor:

```
graph main [[ main ]] {
  output stream float audioOutput;
  input event {
```

```
    float amplitudeIn1 [[ min: 0, max: 1,
      init: 1, name: "amplitude 1", unit: "
      step: 0.01 ]];
    float frequencyIn1 [[ min: 400, max: 5(
      init: 450, name: "frequency 1",
      unit: "Hz", step: 1 ]];
    float amplitudeIn2 [[ min: 0, max: 1,
      init: 1, name: "amplitude 2",
      unit: "dB", step: 0.01 ]];
    float frequencyIn2 [[ min: 400, max: 5(
      init: 450, name: "frequency 2",
      unit: "Hz", step: 1 ]];
  }

  let {
    sine1 = sinusoid;
    sine2 = sinusoid;
    sum = summer;
  }

  connection {
    amplitudeIn1 > sine1.amplitudeIn;
    frequencyIn1 > sine1.frequencyIn;
    amplitudeIn2 > sine2.amplitudeIn;
    frequencyIn2 > sine2.frequencyIn;
    sine1.audioOutput > sum.input1;
    sine2.audioOutput > sum.input2;
    sum.audioOutput > audioOutput;
  }
}
```

The main graph (defined with $[[main]]$, is the graph where the computation begin. Essentially, it is the main of the program.

5. FEEDBACK DELAY NETWORK IMPLEMENTATION IN SOUL

The first thing to consider regarding the implementation of the feedback delay network in SOUL programming language, is its algebraic structure, already shown in figure 4. It is easy to notice how different components had to be developed before connecting together the different parts. In particular, without considering the "container" graphs, the following processors were necessary: multiplier, adder, delay, feedback matrix and lowpass filter. In this section the different implementations and how they were interconnected will be described.

5.1 Multiplier

The multiplier is the easiest and most used processor (with the adder), inside the feedback delay network structure. It was designed in order to allow the user to control the value of the coefficient which multiplies a given signal. As a result, the processor was composed of an input stream (the input signal), an event input (to control the value of the coefficient) and an output stream (the result of the multiplication between the input signal and the coefficient). The structure can be seen in figure 6.

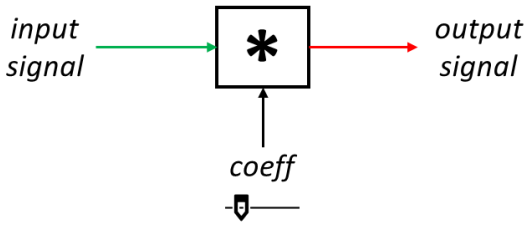


Figure 6. There is only one input signal that, inside the block, is multiplied by a coefficient $coeff$ that can be controlled by the user. The result is successively sent in output.

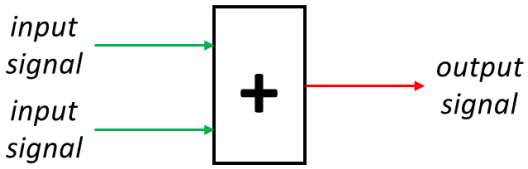


Figure 7. There are two input signals, which sum is sent in output.

5.2 Adder

The adder is a processor used to sum two input signals. Consequently, the structure was designed with two input streams (the two input signals) and one output stream (the sum of the two input signals). Hence, any parameters can be controlled by the user in this case. The structure is shown in figure 7.

5.3 Delay Line

The delay line processor is essentially composed of an input stream, an output stream, a buffer where data is stored and two heads: one for reading and one for writing. The samples come through the input stream and they are stored into the buffer (that is an array) and every time a sample is stored, both the heads are incremented of one unit (if the position of one head exceed the dimension of the array, it is simply wrapped and set at the beginning of the array, to have a circular behaviour). At each step, the reading head reads the sample in its current position and sends it to the output stream. The user can manipulate the length of the delay, as a matter of fact, in order to change such parameter, it is enough changing the distance between the writing head and the reading head. Obviously, to have a correct behaviour, it is necessary that the distance between the two heads is not higher than the length of the array buffer. The structure of the delay line processor and the buffer management mechanism is illustrated in figure 8.

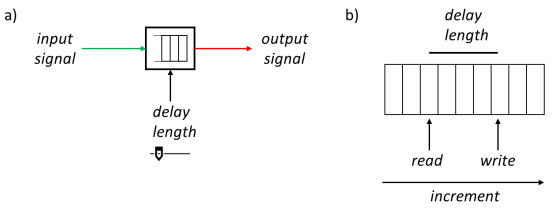


Figure 8. a) the input stream is the current signal, while the output stream is the delayed version of the input. The delay can be changed in real-time by the user. b) the delay can be manipulated by changing the distance between the writing head and the reading head. To guarantee the correct behaviour of the delay line, it is necessary that the delay length (or rather, the distance between the two heads) is not higher than the dimension of the buffer.

5.4 Feedback Matrix

The feedback matrix (necessary for damping) is composed of one input stream for each input of the input vector and one output stream for each output of the output vector. At the moment of the implementation of this project, SOUL didn't make available the use of multiple arrays, consequently the matrix was simply represented as one array for each row of the matrix. In this case, since a Jot's feedback delay network with four delay lines was used, the matrix was represented with four arrays of four elements, as can be seen in the following code:

```
float[] matrixRow1 =
  (0.0449, 0.0449, 0.0449, 0.0449);
float[] matrixRow2 =
  (0.0449, 0.0449, 0.0449, 0.0449);
float[] matrixRow3 =
  (0.0449, 0.0449, 0.0449, 0.0449);
float[] matrixRow4 =
  (0.0449, 0.0449, 0.0449, 0.0449);
```

The elements of the matrix cannot be changed in real-time, as a result there are not input events in this processor, but only four input streams and four output streams. The structure is described in figure 9.

5.5 Lowpass Filter

The lowpass filter was implemented in the form of [12], or rather based on the following equations:

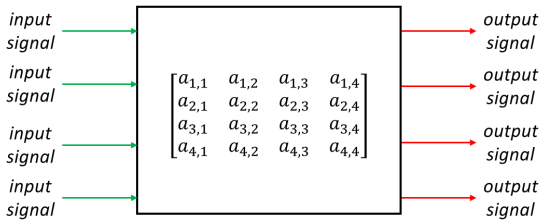


Figure 9. It is possible to see how the processor takes in input the single values separately. Hence, the dot product with the damping matrix is computed and the result is sent on the separated four output stream lines.

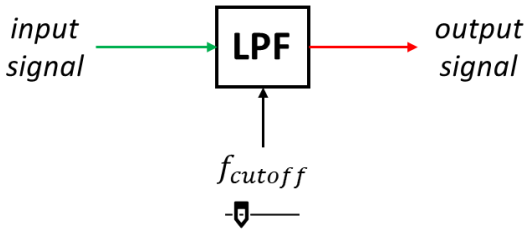


Figure 10. the input streams is the signal we want to filter, the output stream is the filtered signal and the cutoff frequency is a parameter of the filter that can be manipulated by the user.

$$t = 1 - \cos(2\pi \frac{f_{cutoff}}{f_{sampling}})$$

$$\alpha = -t + \sqrt{t \cdot t + 2 \cdot t}$$

$$y[n] = \alpha \cdot x[n] + (1 - \alpha) \cdot y[n-1]$$

(9)

Hence, the processor is composed of an input stream, an output stream and an input event with which it is possible to control the value of the cutoff frequency, or rather the variable f_{cutoff} in the equation. The structure is described in figure 10.

5.6 Connections

The different components described in the previous paragraphs needed being interconnected each other to create the more complex structure of the feedback delay network.

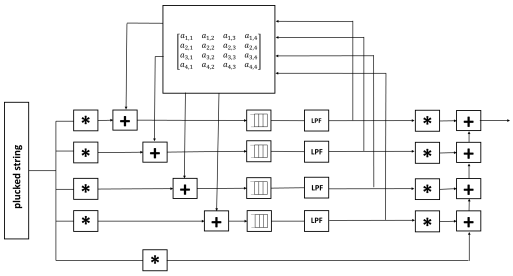


Figure 11. The output of the plucked string is the input signal of the feedback delay network structure. It is possible to notice that the input is, at first, multiplied for different coefficient and then the result is summed with the feedback coming from the damping matrix. Hence, there are delay lines and lowpass filters, which output is the input signal for the damping matrix. Finally, the results are multiplied for other coefficient and summed toghether to determine the output of the system. For the sake of clarity, the input events were not reported in the scheme.

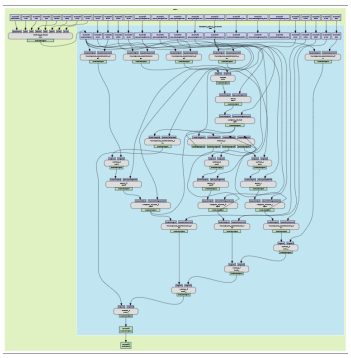


Figure 12. The structure of the feedback delay network model can be also analyzed by means of the "Audio Graph" modality availabe in SOUL. The connections between the different components are completely equivalent to the ones reported in the scheme in figure 11

The model considered is, as already mentioned, a four delay lines Jot's feedback delay network. Since in this case, we want to simulate the sound box of a guitar, the plucked string obtained with finite difference schemes method was used as input of the system. In figure 11 it is possible to see the scheme of the structure as it was implemented in SOUL, considering the blocks described in the previous sections. In an equivalent fashion, the same structure can be analyzed by means of the "Audio Graph" modality available on SOUL web editor (see figure 12).

5.7 Plucked String

Even if the plucked string simulation is not part of the feedback delay network structure, but only the input, it is nevertheless worth to give a rough description regarding how it was developed in SOUL. Since it was implemented with finite difference schemes technique, it was necessary to define different arrays, representing the state of the string in different times. In particular, three arrays were used because of the maximum derivative with the time was a second order derivative, that was discretized with the central method. As a result, the three arrays used were one for the discrete previous time, one for the current discrete time and one for the discrete next time.

```
processor string_plucked {
  // streams declaration ...
  // input events declaration ...

  // events implementation ...

  // state variables declaration ...

  void run() {
    loop {
      // next string state update ...

      // send result in output:
      output << uNext[pos];

      // previous string state update ...
      // cuurrent string state update ...
    }
  }

  void excite() {
    ...
  }

  void deexcite() {
    ...
  }
}
```

As described in the plucked string simulation paragraph, the differential equation of the plucked string depends on many parameters such as length, tension, density, radius, stiffness, frequency dependent damping and frequency independent damping. Consequently, they were all implemented in the SOUL processor as input events that can be manipulated in real-time by the user. Another relevant input event implemented was "excite". As a matter of fact, when it passes from the "off" state to the "on" state, the excite method is called and the arrays representing the current and the previous states of the plucked are modified to excite the system. On the other hand, when the excite parameter passes from the "on" state to the "off" state, the deexcite method is called and all the arrays are re-initialized with 0 values.

```
void excite() {
  wrap<maxLength> i = 0;
  for(i=0; i<N; i++) {
    u[i] = excitation shape
    uPrev[i] = excitation shape
  }
```

```
}


void deexcite() {
  wrap<maxLength> i = 0;
  for(i=0; i<N; i++) {
    uNext[i] = 0;
    u[i] = 0;
    uPrev[i] = 0;
  }
}
```

6. GENETIC ALGORITHM APPROACH

6.1 Genetic Algorithms Introduction

Genetic algorithms are a family of algorithms in artificial intelligence category. Their use is mostly exploited in all those cases in which it is known the existence of a solution, but it is too difficult or too expensive to obtain in an analytical fashion. These algorithms are iterative and at each generation they constantly try to improve the population, making it always more similar to a target, in other words always more suited. They are called "genetic algorithms" because they are clearly inspired by the genetic field and Darwin's evolutionary theory. The genetic algorithm follows a well-defined pattern: there is a random initial population that is generated at the beginning of the algorithm, then such individuals are evaluated by means of a fitness function (that essentially defines how the individual is "suited", or rather how it is similar to the target). Only a percentage (usually between 20% and 50%) is maintained, while the rest of the population "not suited" is discarded. The "survived" individuals are successively passed in functions of mating and mutation. The mating function has the aim of matching together two random individuals in order to create a new individual, essentially a "child" of the "couple". Such new individual could be subject to mutations on the basis of the mutation function. If the mutation brings to an improvement of the child's fitness, then the mutated child is maintained and the other one discarded. Mating and mutation functions are iterated until the number of individuals of the new population is smaller than the dimension of the initial population (in essence such iteration stops when all the discarded individuals were replaced with new individuals). The last stage of the algorithm is the so called "degeneration": if the percentage of the individuals with the fitness closer to the average fitness than a given threshold value is bigger than a given percentage initially established, then only the best individual (the one with the best fitness) is maintained, while the rest of the population is discarded and replaced with re-initialized random individuals. At this point, the generation is concluded and the algorithm will be re-iterated with the current population as initial population. The algorithm will be iterated for a number of times equals to the number of generations established at the beginning of the algorithm [2, 13, 14]. An explicative diagram is shown in figure 13.

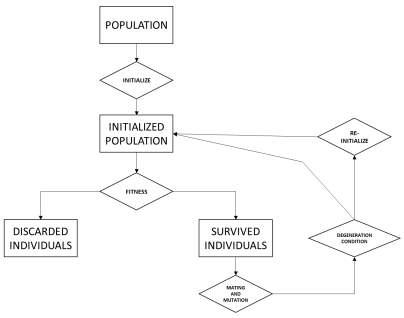


Figure 13. genetic algorithm structure diagram. It is possible to see the initialization and, after that, the iterated generations composed of the stages: fitness evaluation, mating, mutation and degeneration.

6.2 Genetic Algorithms Implementation

6.2.1 Population Initialization

In order to avoid a too high dispersion of values and consequently to help the algorithm to find faster the most suited solution, random start values in an already acceptable range were adopted. Considering the structure of a feedback delay network, the $\vec{b}$ vector and the $\vec{c}$ vector were both initialized with values between 0 and 1. The d value was initialized in a range between 0 and 1 as well. Instead, delay lines were initialized with buffers with a length between 1 and 100 samples. It was decided to maintain the upper bound of this range relatively small considering the sound box is much smaller than a room and consequently the echo density is much shorter (as a matter of fact, it was noticed empirically with SOUL model that with high delays, the plucked of the string tends to be repeated over the time as an echo). The cutoff frequencies of the lowpass filters were initialized with values between 0 and 1000 Hz. The matrix values were chosen between four different categories of matrices: there was the 25% of probability to be initialized as a Stutner-Puckette matrix, the 25% to be initialized as a 4x4 Hadamard matrix, the 25% to be initialized as a random circular matrix with values between -1 and 1 and the final 25% to be initialized as a random matrix of values between -1 and 1. The four different types of matrices can be seen in figure 14. Even if circulant and random matrices are not necessarily stable, they were included in the options for a matter of diversification. Moreover, since the target impulse response is obviously stable, all the unstable models will be probably discarded in the first generations of the algorithm, as a result there is no way they can compromise the result.

6.2.2 Fitness Function

The fitness function is useful to define how a given individual is suited to survive, or rather how it is close to the target. Consequently, in the current case, the fitness function can be described as the computation of the distance, given a metric, between the real impulse response of the sound box and the artificial impulse response generated by the feedback delay network with a particular set of parameters.

$$g\begin{bmatrix} 0 & 1 & 1 & 0 \\ -1 & 0 & 0 & -1 \\ 1 & 0 & 0 & -1 \\ 0 & 1 & -1 & 0 \end{bmatrix} \qquad g\begin{bmatrix} 1 & 1 & 1 & 1 \\ -1 & 1 & -1 & 1 \\ -1 & -1 & 1 & 1 \\ 1 & -1 & -1 & 1 \end{bmatrix}$$

$$\begin{bmatrix} a & b & c & d \\ d & a & b & c \\ c & d & a & b \\ b & c & d & a \end{bmatrix} \qquad \begin{bmatrix} r_{1,1} & r_{1,2} & r_{1,3} & r_{1,4} \\ r_{2,1} & r_{2,2} & r_{2,3} & r_{2,4} \\ r_{3,1} & r_{3,2} & r_{3,3} & r_{3,4} \\ r_{4,1} & r_{4,2} & r_{4,3} & r_{4,4} \end{bmatrix}$$

Figure 14. At the top left Stutner-Puckette matrix, at top right 4x4 Hadamard matrix, at bottom left the circulant matrix, at bottom right random matrix. For stability, in Stutner-Puckette and Hadamard matrices, it is necessary that $|g| < 1/\sqrt{2}$. In circulant and random matrices, stability is not guaranteed.

The more the parameters return a suited result, the lower is the distance between the target impulse response and the artificial impulse response and consequently the better will be the fitness of the individual, with a higher probability to "survive" and attending the next generation. One first approach could be to make the summation sample-by-sample between the target impulse response and the artificial impulse response as in the equation:

$$fitness = \sum_{i=1}^{N} x_{target}[i] - x_{artificial}[i] \tag{10}$$

But such method is not very effective, since the single samples in the impulse response are quite disperse and they don't follow a precise line, as a result the information concerning the general behaviour is lost and not considered in the final result. For this reason, an approximation of the target impulse response and of the artificial one is necessary to compute adequately their distance. In the paper Chemistruck at al [15], a method based on the power envelope was proposed. It is a particular approximation based on the reproduction of the impulse response's attack and release stages, following the equations:

$$y[n] = \begin{cases} a_0(y[n-1] - x[n]) + x[n], & \text{if } n = 0 \\ r_0(y[n-1] - x[n]) + x[n], & \text{otherwise} \end{cases}$$

(11)

Where a_0 and r_0 are two empirical parameters respectively for attack and release, x[n] is the current value of target impulse response and y[n] is the current value of the approximation.

Nevertheless, as outlined in the mentioned paper, the approximation is good mostly from a global point of view, not considering the nuances due to low local representation. Essentially, it has the opposite problem of difference sample-by-sample approach. As a result, a new approach to compute the distance between the target impulse response and the artificial impulse response is proposed in this paper. Essentially, it is based on the computation of linear regression on consecutive equidistant points, in order to approximate the impulse response as a sequence of consecutive segments. Therefore, the single sections were computed according to the following estimation:

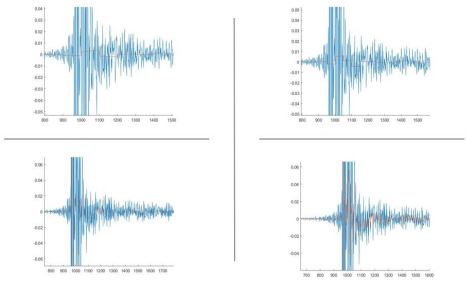


Figure 15. From the top-left to the bottom-right: approximations with respectively segments of 100, 50, 10 and 5 samples.

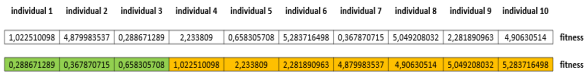
$$a = \frac{(\sum_{i=sL}^{s(L+1)} y_i)(\sum_{i=sL}^{s(L+1)} x_i^2) - (\sum_{i=sL}^{s(L+1)} x_i)(\sum_{i=sL}^{s(L+1)} x_i y_i)}{L(\sum_{i=sL}^{s(L+1)} x_i^2) - (\sum_{i=sL}^{s(L+1)} x_i)^2} \tag{12}$$

$$b = \frac{L(\sum_{i=sL}^{s(L+1)} x_i y_i) - (\sum_{i=sL}^{s(L+1)} x_i)(\sum_{i=sL}^{s(L+1)} y_i)}{L(\sum_{i=sL}^{s(L+1)} x_i^2) - (\sum_{i=sL}^{s(L+1)} x_i)^2} \tag{13}$$

$$y_{[sL,s(L+1)]} = bx_{[sL,s(L+1)]} + a \tag{14}$$

Where s is the index of the current segment and L the number of samples for each segment, or rather the frame size (except eventually the last one, considering the rest of the division between the total number of samples of the impulse response and L). Consequently, the result of the approximation is a series of segments that follows the general trend of the impulse response. It is possible to notice in figure 15 that it is necessary to find a convenient trade-off between the length of the segment and the accuracy with which information is represented: with an interval too short, information is represented very accurately for each sample, but the information inherent the general behaviour is lost; conversely, too large segments bring to the loss of local behaviour (as a matter of fact, for high lengths, the segments tend to be flat, inasmuch the impulse response is quite specular on the x axis). It is worth to notice that this approach degenerates into the sample-by-sample approach if the length of the segment is $L = 1$.

The segment length chosen was 10 samples, since from a qualitative point of view was the best balance between local and global information. Finally, individuals' fitness value was computed as the difference between the approximation of the target impulse response and the approximation of artificial impulse response.



individual 1	individual 2	individual 3	individual 4	individual 5	individual 6	individual 7	individual 8	individual 9	individual 10	
1,022510098	4,879983537	0,288671289	2,233809	0,658305708	5,283716498	0,367870715	5,049208032	2,281890963	4,90630514	fitness
0,288671289	0,367870715	0,658305708	1,022510098	2,233809	2,281890963	4,879983537	4,90630514	5,049208032	5,283716498	fitness

Figure 16. The percentage of survival is rounded on the floor, therefore a rate of 35% on a population of 10 individuals maintains "alive" the three individuals with the best fitness.

$$fitness = \sum_{i=1}^{N} |\hat{x}_{target}[i] - \hat{x}_{artificial}[i]| \tag{15}$$

Where $\hat{x}_{target}$ is the approximation of the target impulse response and $\hat{x}_{artificial}$ is the approximation of the artificial impulse response.

6.2.3 Discarding Criteria

As in every genetic algorithm, once the population has been evaluated by means of the fitness function, the set of individuals is sorted. In this case, the sort is for increasing values of the fitness and only the 35% of the individuals "survives" to the next generation, or rather the 35% with the lowest fitness value (that consequently is more similar to the target since in this case the fitness function is the computation of a distance, as explained in the previous paragraph)(see figure 16).

6.2.4 Mating Function

The mating function is useful in order to re-fill the population to the initial dimension. It consists in extracting randomly two individuals between the 35% of survived ones and then mix their "genes", or rather their features such as vector $\vec{b}$, vector $\vec{c}$, the cut-off frequencies of the lowpass filters, the delay lines buffer's length and the damping matrix. Chemistruck at al [15] proposed to generate the new parameters of the "child" as the average of the parameters of the "parents" and in particular only for lowpass filter's cut-off frequencies and for the elements of the damping matrix.

$$f_c[i] = \frac{f_{c1}[i] + f_{c2}[i]}{2} \tag{16}$$

Where i is the i^{th} lowpass filter, f_c is the cut-off frequency of the new generated individual, f_{c1} and f_{c2} are instead the cut-off frequencies of the two "parents".

$$[g]_{ij} = \frac{[g_1]_{ij} + [g_2]_{ij}}{2} \tag{17}$$

Where $[g]_{ij}$ is the element of the damping matrix in position (i, j). The two elements involved in the average operation belong to the parents. Here in this paper, some modifications were accomplished to increase the possibility of the algorithm to reproduce a more reliable model of the target. First at all, not only cut-off frequencies and the damping matrix elements were involved in the mating function, but vector $\vec{b}$, vector $\vec{c}$, the scalar d and the delay lines buffer's length as well. In particular, the last one is fundamental to try to reproduce accurately the echo density, therefore could be inadequate to suppose to establish a priori delay lengths without trying to adapt them to the model.

$$b[i] = \frac{b_1[i] + b_2[i]}{2} \tag{18}$$

$$c[i] = \frac{c_1[i] + c_2[i]}{2} \tag{19}$$

$$d = \frac{d_1 + d_2}{2} \tag{20}$$

$$len(buffer[i]) = \frac{len(buffer_1[i]) + len(buffer_2[i])}{2} \tag{21}$$

Moreover, the most important modification adopted here in this paper to Chemistruck at al [15] approach was the modality with which the elements of the damping matrices of the parents were involved to produce the elements of the damping matrix of the child. As previously explained, it is based entirely on the average of the values. Nevertheless, this approach generally brings to undesired results. This is why, from an algebraic point of view, the set of unitary matrices is not closed to the product element-by-element and consequently the resulting child could present a divergent behaviour even if both parents are convergent. A given matrix is said to be "unitary" when the dot product with its transpose results in the identity matrix.

$$G \cdot G^T = G^T \cdot G = I \tag{22}$$

The unitary matrix is a necessary and sufficient condition for the stability of the feedback delay network [1], because it causes a convergent damping of the reverberation. As a proof that the average between unitary matrices is not necessarily a unitary matrix, we can consider the average between Stutner-Puckette matrix and Hadamard 4x4 matrix:

$$\frac{1}{2} \cdot \begin{bmatrix} 0 & 1 & 1 & 0 \\ -1 & 0 & 0 & -1 \\ 1 & 0 & -0 & -1 \\ 0 & 1 & -1 & 0 \end{bmatrix} \cdot \begin{bmatrix} 1 & 1 & 1 & 1 \\ -1 & 1 & -1 & 1 \\ -1 & -1 & 1 & 1 \\ 1 & -1 & -1 & 1 \end{bmatrix} = \begin{bmatrix} \frac{1}{2} & 1 & 1 & \frac{1}{2} \\ -1 & \frac{1}{2} & -\frac{1}{2} & 0 \\ 0 & -\frac{1}{2} & \frac{1}{2} & 0 \\ \frac{1}{2} & 0 & -1 & \frac{1}{2} \end{bmatrix} \tag{23}$$

Now, if we compute the dot product between such matrix and its transpose, it is easy to notice that the result is not an identity matrix and consequently there is the loss of the unitary property:

$$\begin{bmatrix} \frac{1}{2} & 1 & 1 & \frac{1}{2} \\ -1 & \frac{1}{2} & -\frac{1}{2} & 0 \\ 0 & -\frac{1}{2} & \frac{1}{2} & 0 \\ \frac{1}{2} & 0 & -1 & \frac{1}{2} \end{bmatrix} \cdot \begin{bmatrix} \frac{1}{2} & -1 & 0 & \frac{1}{2} \\ 1 & \frac{1}{2} & -\frac{1}{2} & 0 \\ 1 & -\frac{1}{2} & \frac{1}{2} & -1 \\ \frac{1}{2} & 0 & 0 & \frac{1}{2} \end{bmatrix} = \\ = \begin{bmatrix} \frac{5}{2} & -\frac{1}{2} & 0 & -\frac{1}{2} \\ -\frac{1}{2} & \frac{3}{2} & -\frac{1}{2} & 0 \\ 0 & -\frac{1}{2} & \frac{1}{2} & -\frac{1}{2} \\ -\frac{1}{2} & 0 & -\frac{1}{2} & \frac{3}{2} \end{bmatrix} \tag{24}$$

It is worth to notice that, always from an algebraic point of view, the set of unitary matrices forms a group (the so called "unitary group") considering the operation of matrix row-column multiplication [16, 17]. Since unitary matrices form a group with dot product operation, this means that the dot product between two unitary matrices is still a unitary matrix, that implies the conservation of the stability. Here, the algebraic demonstration that unitary matrices form a group with dot product is given:

At first, we have to consider the unitary condition of the starting matrices:

$$\begin{aligned} A^H \cdot A = A \cdot A^H = I \\ B^H \cdot B = B \cdot B^H = I \end{aligned} \tag{25}$$

Consequently,

$$\begin{aligned} &(AA^H) \cdot (B^H B) = I \\ &A^{-1} A A^H B^H B B^{-1} = A^{-1} B^{-1} \\ &A^H B^H = A^{-1} B^{-1} \\ &(BA)^H = (BA)^{-1} \end{aligned} \tag{26}$$

Finally, we just need to apply again the definition of unitary matrix on the product between the two matrices to conclude the demonstration:

$$\begin{aligned} (BA)(BA)^H = (BA)(BA)^{-1} = I \\ (BA)^H (BA) = (BA)^{-1} (BA) = I \end{aligned} \tag{27}$$

As a result, here dot product operation was proposed instead of element-by-element average in order to define the damping matrix of the new individual.

$$G = G_1 \cdot G_2 \tag{28}$$

The mating function generates a new individual at each iteration from the best individuals and it stops only when the population went back to the initial dimension.

6.2.5 Mutation Function

The mutation function is strictly related to mating function, inasmuch it is executed inside it. It consists in causing a random mutation (inside a previously established range) of the parameters of the models. Chemistruck at al [15] proposed the mutation exclusively of lowpass filters' cut-off frequency and damping matrices' elements. In this case, instead, other values were involved as for the mating function case, except for the damping matrix, as will be soon explained. Here, the values adopted were 5% of mutation probability for each element of the $\vec{b}$ vector with a new random value between 0 and 1, 5% of mutation probability for each element of the $\vec{c}$ vector with a new random value between 0 and 1, 1% of mutation probability of the d scalar with a new random value between 0 and 1, 5% of mutation probability for the cut-off frequency of each lowpass filter of the feedback delay network and, finally, mutations of delay lines buffer's length and damping matrix were not implemented. In particular, the decision to not mutate the damping matrix is strictly related to the explanation in mating function paragraph. As a matter of fact, with random mutation on matrix parameters, the damping matrix would probably switch to a non-unitary matrix with a consequent loss of its convergence. Instead, the decision to not mutate the length of the delay lines was not deduced from a conceptual problem as in damping matrix case. As a result, in a future work will be possible to propose the same algorithm with the possibility to mutate this parameter as well. The values that were chosen here are arbitrary and based on the reasonability of their efficiency to bring a random variation on individuals to differentiate them to find new solutions that otherwise couldn't be found, following the algorithm in a completely deterministic way. Nevertheless, it is worth to mention that the choice of the parameters presents a trade-off between how deep you want to explore the solutions generated from the initial population (if you opt for low mutation probabilities) or how many solutions you want to find from more random individuals (and then more starting points) but with a less deep exploration (if you opt for high mutation probabilities). This experiment can obviously be done again in the future with different probabilities for the mutations. This mutation operation is accomplished systematically for every mating process: when the new individual is generated by means of the rules described in the previous paragraph, the mutation procedure is executed on the model's parameters and a mutated new individual is generated in this way. The fitness function is applied on both new individual and mutated new individual and their resulting fitness is compared: only the best one is maintained and passed into the population set (therefore can be immediately used for mating in the next mating step), while the other one is definitively discarded.

6.2.6 Degeneration Conditions

In genetic algorithms, degeneration is a situation for which most of the population degenerates in a given range of values, with a consequent deadlock condition in which new individuals are very similar to the parents with a low improvement of the fitness function, making useless the computation of the next generations. A possible solution to the degeneration situation problem could be to maintain only the individual with the best fitness and re-initialize the rest of population [15]. In this way, solutions from new differ-

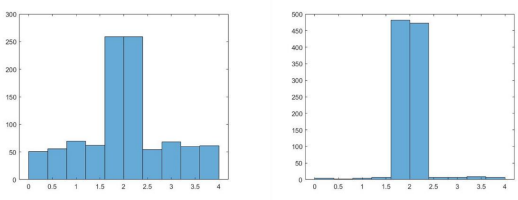


Figure 17. Given a population of 1000 individuals, a threshold of ± 0.05 and a percentage of 95%, the distribution of fitness is represented in the histograms. To the left a case of non-degeneration and to the right a case of degeneration.

```
population = initialize()
for each generation:
        computeFitness(population)
        discard(population)
        matingAndMutation(population)
        if degenerationCondition:
                reinitialize(population)
```

Figure 18. Generative algorithm pseudocode.

ent starting points can be found. In this case, degeneration was computed considering a threshold of deviation equal to ± 0.05 and a percentage of 95% inside such a range (see figure 17). This means that if at least the 95% of individuals have the fitness close to the average population fitness until 0.05, then degeneration occurs. Also, these parameters were chosen arbitrarily too, since considered reasonable, but the experiment could be repeated in the future with other values of the threshold and of the percentage. In this particular case, with these values, degeneration have never occurred, probably because the very small value that was chosen as threshold and the relatively high percentage of population in such range (considering the fitness of almost all population into an interval of 0.1).

6.3 Algorithm Complexity

In this paragraph, temporal and spatial complexities will be analysed according to the classic theory of computation [18]. As a starting point for the analysis, a very simplified pseudocode comprehensive of the most significative and computationally relevant steps of the algorithm will be given in figure 18.

6.3.1 Temporal Complexity

The initialization of the population (excluding cases of degenerations) occurs only once at the beginning of the algorithm. Even if the computation of the fitness is inside the for loop, the most onerous task inside it (or rather the computation of the fitness) occurs only once because it computes the fitness only if the individual's fitness is not computed yet. Since for the fitness of new individuals is computed in "matingAndMutation" portion of code and in case of degeneration the fitness of the new population is computed in "reinitialize" portion, then the computation of fitness in "computeFitness" only happens at the first step. The discard of the population, or rather the process of sorting and discarding the individuals not suited, can be described on the basis of the algorithm used as $N+(N-1)+(N-2)++1 = \frac{N(N+1)}{2}$ for Gauss formula (therefore not a very optimized sorting algorithm). Nevertheless, the time step of the sort algorithm in considerably lower than the time step to compute the fitness function (probably many orders of magnitude), consequently this term can be neglected and approximated to 0. The mating and mutation functions have the onerous task of computing the fitness function of the new individuals and of the new mutated individuals (their generation can be neglected), they are obviously affected by the percentage of individuals discarded for each generation: the higher is such percentage, the higher is the number of the fitness functions to compute. The computation of the degeneration condition can be neglected, on the other hand, in case it is positive, the re-initialization of the population is an extremely onerous task. Finally, we can resume the algorithm's worst case in the following formula:

$$f_c \cdot N + g \cdot d + 2 \cdot g \cdot (N - dPerc \cdot N) \cdot f_c + \\ + g \cdot (N-1) \cdot f_c \tag{29}$$

Where g is the number of generations, N is the number of individuals of the population, f_c is the fitness function computation cost, d is the discard computation cost, $dPerc$ is the percentage of population survived at each step. Considering the discarding task neglectable and $N - 1 \approx N$, we can write the formula as follows:

$$f_c \cdot N + 2 \cdot g \cdot N \cdot (1 - dPerc) \cdot f_c + N \cdot g \cdot f_c = \\ = N \cdot (f_c + 2 \cdot g \cdot (1 - dPerc) \cdot f_c + g \cdot f_c) = O(N) \tag{30}$$

Therefore, the algorithm is linear, as can be seen in figure 19.

We can consider the best case as well, or rather when degeneration never occurs (as in the case of the experiment described in this paper). It is shown in figure 20.

$$N \cdot f_c + g \cdot d + 2 \cdot g \cdot (N - dPerc \cdot N) \cdot f_c = \\ = N \cdot (f_c + 2 \cdot g \cdot (1 - dPerc) \cdot f_c + g \cdot f_c) = o(N)$$

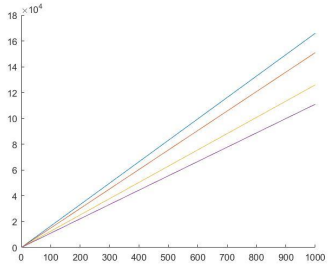


Figure 19. The worst case of temporal complexity evolution for increasing population, considering 50 generations, a fitness function computation cost equals to 1 and survived percentage respectively 35%, 50%, 75% and 90%. It is possible to see how the higher is the percentage of population that survives at each generation, the lower is the computational cost of the algorithm. As a matter of fact, $dPerc \approx 0 \Rightarrow (1 - dPerc) \cdot f_c \approx f_c$, conversely $dPerc \approx 1 \Rightarrow (1 - dPerc) \cdot f_c \approx f_c$. As a result, since f_c is always a positive number, the slope is higher when the survive percentage is low.

(31)

6.3.2 Spatial Complexity

The case of spatial complexity is much easier to analyse, since at each iteration of the algorithm the same percentage of the population is discarded and successively re-filled by means of mating function. Only the memory occupied to store the individual is considered relevant, consequently other temporary variables are neglected. As a result, worst case and best case coincide and they uniquely depend on the initial dimension of the population. In particular, it is worth to consider that during the mutation of the last iteration of the mating function, both new individual and mutated new individual are generated, consequently the effective maximum is $N + 1$. The worst case is:

$$N + 1 = O(N) \tag{32}$$

The best case is:

$$N + 1 = o(N) \tag{33}$$

6.4 Implementation and Results

The generative algorithm to estimate the parameters of the feedback delay network was entirely developed in Matlab programming language, since it easily allows to analyse and manipulate accurately both the impulse response signal and the feedback delay network model, that is not possible with other more specific programming languages such as SOUL (that was used here only for the synthesis of the final result). The code of Matlab implementation of the genetic algorithm can be found at this link:
`https://drive.google.com/drive/folders/1gYXLX3g7QE94okMqqWHW0jxOIxyGfidj?usp=sharing`

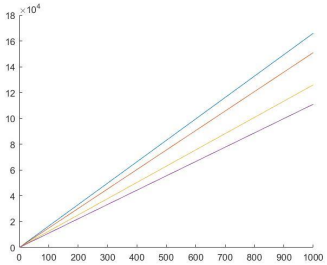


Figure 20. The best case of temporal complexity evolution for increasing population, considering 50 generations, a fitness function computation cost equals to 1 and survived percentage respectively 35%, 50%, 75% and 90%. It is possible to see how the higher is the percentage of population that survives at each generation, the lower is the computational cost of the algorithm

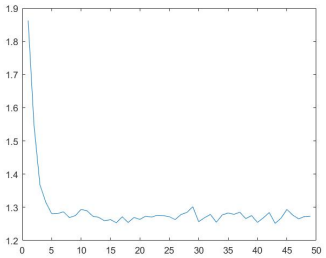


Figure 21. Average fitness of the population at each generation. It is possible to notice that the trend is not a monotonic non-decreasing function, as a matter of fact it is not guarantee that the child's fitness will be necessarily better than parents' fitness. Instead, obviously the trend of the best individual is a not increasing function.

while the implementation of the model in Matlab can be found here:
`https://drive.google.com/drive/folders/1e5BVOTVWKugoJfwwHC49xKQMLg0_sOQi?usp=sharing`
Which results (both in terms of estimated parameters and audio file generated) can be found at this link:
`https://drive.google.com/drive/folders/1tU4gf9uMIjAZQobULhXHhe-tsf1cAQQH?usp=sharing`
The parameters that were chosen for this specific simulation were: a population of 40 individuals, 50 generations and a percentage of survived population at each generation equals to 35%. Concerning the algorithm convergence, it is easy to notice in figure 21 that the average fitness quickly converge in the first 5 generations and then it tends to maintain a value around 1.3.

Once the algorithm finished the computation, the best individual was extract from the population, obtaining the estimation of the following parameters:

$$\vec{b} = [0.4243, 0.6220, 0.1653, 0.7658] \tag{34}$$

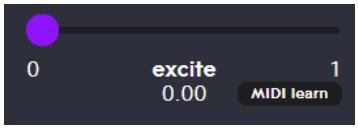


Figure 22. The excitation of the string can be controlled by means of "excite" slider.

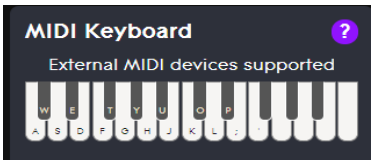


Figure 23. The excitation of the string can be controlled by means of a MIDI keyboard.

$$d = 0.0463 \tag{35}$$

$$\vec{delay} = [658, 942, 348, 556] \tag{36}$$

$$\vec{cutoff} = [946.2887, 226.0057, 547.5537, 500.7412] \tag{37}$$

$$\vec{c} = [0.1587, 0.6848, 0.5774, 0.3031] \tag{38}$$

$$G = 0.0723 \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & -1 & 1 & -1 \\ 1 & 1 & -1 & -1 \\ 1 & -1 & -1 & 1 \end{bmatrix} \tag{39}$$

$$fitness = 1.1207 \tag{40}$$

Once the parameters were estimated with Matlab, the feedback delay network and the plucked string were proposed in SOUL programming language, in order to provide a real time implementation. The parameters are initialized to the values that were found with genetic algorithm and can be changed into a specific range of values to accomplish modifications to the reverberation (it is also possible to change the parameters relative to the string). Two different implementations were proposed: one with the excitation as a parameter(see figure 22), where it is possible to change the length of the string to control the pitch and one with MIDI keyboard, where the length of the string is defined as a consequence of the frequency of the key pressed (see figure 23). Such implementations can be found at the following link:
`https://drive.google.com/drive/folders/12y3TXwW0QAKoPMPMLYjwvqJ2-UMvsAPP?usp=sharing`
Which results can be found at:
`https://drive.google.com/drive/folders/1CP-hqGQVUdUU-fn00ZPWrrz-S4UoWeJ0?usp=sharing`

7. GRADIENT DESCENT ALGORITHM

7.1 Gradient Descent Algorithm Introduction

In this paper, an alternative approach for the estimation of feedback delay networks is proposed: the gradient descent algorithm. Initially, a brief description of the algorithm is given. Essentially, it is an algorithm of optimization used to find the points of minimum and maximum in functions with several variables which analytical solution is difficult to compute. This method was proposed for the first time by Augustine Cauchy in 1847, for heavenly bodies' orbit estimation [19, 20] and here is reported the mathematical principle on the base of such algorithm. Given a function with several variables:

$$u = f(x, y, z, ...) \tag{41}$$

The aim of this method is to find the solution $(x_0, y_0, z_0, \dots)$ for which $u = 0$. Considering an initial point $(x, y, z, \dots)$ and the derivatives of $f(x, y, z, \dots)$:

$$\begin{aligned} u_x &= f'_x(x, y, z, ...) \\ u_y &= f'_y(x, y, z, ...) \\ u_z &= f'_z(x, y, z, ...) \\ &\dots \end{aligned} \tag{42}$$

We consider following the direction of maximum decrease for small steps $\alpha, \beta, \gamma, ...$

$$\begin{aligned} &f(x + \alpha, y + \beta, z + \gamma, ...) = \\ &f(x, y, z, ...) + \alpha u_x + \beta u_y + \gamma u_z + ... = \\ &u + +\alpha u_x + \beta u_y + \gamma u_z + ... \end{aligned}$$

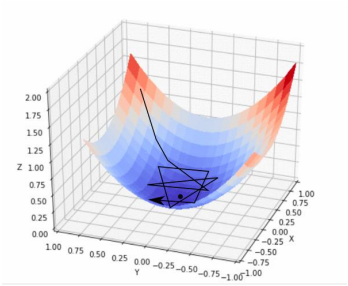


Figure 24. An illustration of the gradient descent algorithm. It is possible to see how the path follows the gradient direction of the function at each step of the algorithm.

(43)
Considering a small step value $\vartheta > 0$

$$\alpha = -\vartheta u_x$$

$$\beta = -\vartheta u_y$$

$$\gamma = -\vartheta u_z$$

...

(44)
Considering now the function:

$$\theta = f(x - \vartheta u_x, y - \vartheta u_y, z - \vartheta u_z, ...) \tag{45}$$

To find the minimum it is necessary to compute:

$$\theta_{\vartheta'} = 0 \tag{46}$$

In such a way, at each step of the algorithm, the "path" to find the minimum of the function follows the direction of maximum decrease (see figure 24).

7.2 Algorithm Description

Gradient descent algorithm was adapted to our specific problem to estimate the parameters of the feedback delay network, in order to make it similar to the target impulse response. Consequently, in this case the function to minimize is the "distance" between the target impulse response and the model. Such distance was computed at the same way as for the genetic algorithm, or rather with the approximation of the function by means of a series of adjacent segments obtained with linear regression of the single windows of samples. Also in this case, the window chosen was 10 samples. Clearly, the function depends on the parameters of the feedback delay network: the vector $\vec{b}$, the scalar d, the delay lines buffer's length, the lowpass filters' cut-off frequencies, the vector $\vec{c}$ and the elements of

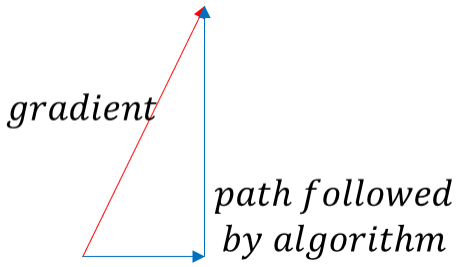


Figure 25. It is possible to notice that the algorithm here proposed didn't follow the direction of maximum decrease, but instead at each step the direction of maximum decrease over the variables direction. This is not necessarily less efficient than considering the direction of maximum decrease, since also such computation has to be considered and it is more onerous.

the damping matrix. Consequently, in order to estimate the parameters, we need:

$$\min_{\vec{b},d,\vec{del},\vec{f_c},\vec{c},G} (d(fdn_{IR}(\vec{b}, d, \vec{del}, \vec{f_c}, \vec{c}, G), target_{IR})) \tag{47}$$

Where d is the distance function (based on linear regression), fdn_{IR} is the impulse response of the feedback delay network and $target_{IR}$ is the impulse response of the target.

As for the genetic algorithm, a feedback delay network with four delay lines was used, consequently the number of variables is 33 (4 for the $\vec{b}$ vector, 1 for d scalar, 4 for buffers' length, 4 for cut-off frequencies, 4 for $\vec{c}$ vector, 16 for the damping matrix). The approach to find the minimum is following the direction of maximum decrease, even if only along one direction per time (so it doesn't follow exactly the gradient, but the direction of the single variable with the higher decrement, the result is the same, but a higher number of steps is necessary), see figure 25.

Therefore, the aim of this algorithm is following the "canonical" direction with the maximum decrease (see figure 26).

This method is applied to the vector $\vec{b}$, to the d scalar, to the delay lines buffer's length, to the cut-off frequencies and to the vector $\vec{c}$, but not to the damping matrix that is considered as a single variable, because the modification of a single element could make the matrix not unitary anymore with the consequent loss of the stability of the damping as explained in the Mating function paragraph of the genetic algorithm section. As it is well-known, gradient descent algorithm doesn't reach necessarily the optimal solution (or rather, the global minimum) as can be seen in figure 27.

Consequently, it was decided to run the algorithm on many different starting points, to increase the probability that one of them "falls" on the global minimum region (see figure 28).

Another aspect that is worth to point out is that the algorithm, as it was developed here, stops only when there

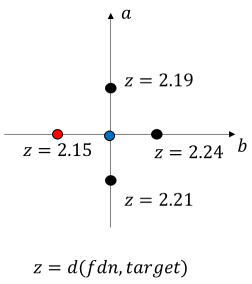


Figure 26. In this case the algorithm will follow the negative verse of the b variable, because in such direction there is the maximum decrease of the distance between the feedback delay network impulse response and the target impulse response. The blue point is the starting point, the red point is where the algorithm moves in the current step to minimize the function.

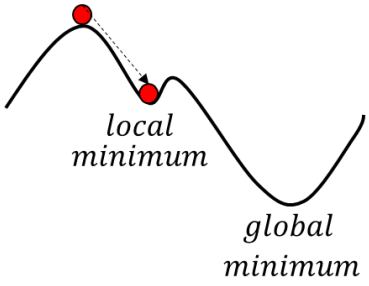


Figure 27. The solution found by the gradient descent algorithm strongly depends on the initial position considered for the solution. Hence, it is possible that the solution deduced by the algorithm is not the global minimum of the function, but a local one.

is not an improvement in the minimization of the distance anymore or when the number of iterations exceeds a maximum number established a priori. This means that the current point maybe is moving to the global minimum, but it doesn't reach it because too many iterations are necessary. This could be useful for the performance of the algorithm, but rarely could really reach the best solution with a small number of iterations. A work for the future could be removing such limit or trying again the algorithm as it was proposed here but with a bigger number of iterations, in order to make the limit irrelevant.

7.3 Algorithm Complexity

In this paragraph, temporal and spatial complexities will be analysed according to classic theory of computation [18]. As a starting point for the analysis, a very simplified pseudocode comprehensive of the most significative and computationally relevant steps of the algorithm will be given (see figure 29) as for the genetic algorithm.

Notice that the notation was maintained from the genetic algorithm. Terms as "individual", "population", "fitness"

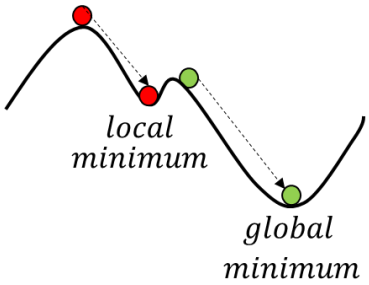


Figure 28. In order to increase the probability to find the global minimum, a good strategy is trying with different initial positions. In this case, the solution represented with the red point "falls" into a position of local minimum, while the solution represented with the green point reaches the global minimum position.

```
population = initialize()
computeFitness(population)
for each individual ∈ population
        for each generation
                newIndividuals = moveParameters(individual)
                bestIndividual = findBestIndividual(newIndividuals)
                if bestIndividual.fitness == individual.fitness
                        break;
                else
                        individual = bestIndividual
        bestIndividuals.add(bestIndividual)
definitiveBestIndividual = findBestIndividual(bestIndividuals)
```

Figure 29. Gradient descent algorithm pseudocode.

are here inappropriate, but since both the algorithms share some data structures and some algorithms, the decision was to maintain the same terminology.

7.3.1 Temporal Complexity

The initialization of the population at the beginning of the algorithm and the consequent computation of the fitness of the single individuals is a first, constant, onerous computational step. For each individual and for each generation, the parameters of the feedback delay network are moved for a total of 38 new individuals, which fitness have to be computed (8 for vector $\vec{b}$, 2 for scalar d, 8 for the delay lines, 8 for the cut-off frequencies, 8 for the $\vec{c}$ vector, 4 for the damping matrix). The search for the best individual is a trivial task of 39 steps, which single step is largely below the computation cost of the fitness function, as a result it is possible to neglect it. In the worst case, all the generations are computed for each individual and the values of the parameters are always far from the boundary (for example, $\vec{b}$ is constrained to be between 0 and 1, therefore if its value is really close 1 or equals to 1, the upper movement of the parameter is not computed). For this reason, this is the complexity in the worst case:

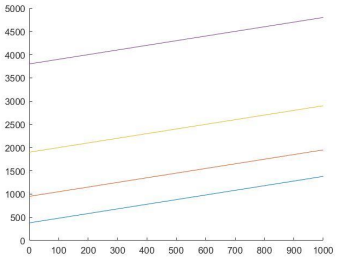


Figure 30. The complexity of the worst case, considering the computational cost of the distance equals to 1, for an increasing population up to 1000. Different cases with respectively 10, 25, 50 and 100 generations are shown.

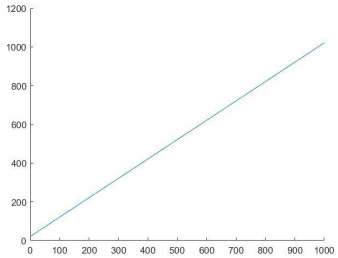


Figure 31. The complexity of the best case, considering the computational cost of the distance equals to 1, for an increasing population up to 1000. Different cases with respectively 10, 25, 50 and 100 generations are shown. It is possible to notice that it is not influenced by the number of generations, as a matter of fact the best case presupposes that the solution is in the minimum position since the first iteration.

$$N \cdot f_c + g \cdot f_c \cdot (4 \cdot 2 + 2 + 4 \cdot 2 + 4 \cdot 2 + 4 \cdot 2 + 4) =$$

$$= N \cdot f_c + 38 \cdot g \cdot f_c = O(N)$$

(48)

Therefore, also in this case the algorithm is linear (see figure 30).

To define the best case, we have to suppose that each individual is in the best position of the function at the first generation, consequently $g = 1$, moreover all the parameters are really close to the boundaries, therefore except the case of the damping matrix, it is possible to consider half the cases for vector $\vec{b}$, scalar d, vector $\vec{c}$, delay lines buffer's length and lowpass filters' cut-off frequency (see figure 31). For this reasons, the complexity in the best case is:

$$N \cdot f_c + 1 \cdot f_c \cdot (4 + 1 + 4 + 4 + 4 + 4) =$$

$$= N \cdot f_c + 21 \cdot f_c = o(N)$$

(49)

7.3.2 Spatial Complexity

Concerning the spatial complexity, it is easily possible to analyse the maximum number of individuals stored at the same time, or rather:

$$N + 38 \tag{50}$$

Where N is the size of the population and 38 is the number of individuals generated when parameters are moved. It represents both the worst and the best case. It could be easily better optimized by defining the new starting individual at the beginning of the first for loop, without the initialization of all the population at the beggining of the algorithm. In this way, the complexity would be reduced to $1 + 38 = 39$, therefore a constant value.

7.4 Implementation and Results

As for the implementation of genetic algorithm, the algorithms for feedback delay network's parameters estimation were developed in Matlab programming language because of its flexibility. Successively, real-time applications were developed in SOUL programming language. Here, there is the code used for gradient descent algorithm implementation in Matlab:
`https://drive.google.com/drive/folders/1EPHfhce1iTWMAlZvTiIrOhWk5Kw-RyoO`
The code model is instead at:
`https://drive.google.com/drive/folders/15KSjYe4DIYWdgpk0295Xnmoig86oLqnu?usp=sharing`
Finally, the result, intended as the sound generated (considering a plucked string with reverberation) is at:
`https://drive.google.com/drive/folders/1ZjKWFNin0WZE-8-qBODbPWBfYRsmBnZM?usp=sharing`
Regarding the execution of gradient descent algorithm, a number of 5 starting points (the "population", or rather the feedback delay network models) was chosen and 40 iterations ("generations") for each one. In figure 32, it is possible to notice the trend of the distance ("fitness") over the increase of the iterations.

Here, there are the parameters estimated with gradient descent algorithm:

$$\vec{b} = [0.0038, 0.0417, 0.0133, 0.8972] \tag{51}$$

$$d = 0.0467 \tag{52}$$

$$\vec{delay} = [94, 308, 456, 102] \tag{53}$$

$$\vec{cutoff} = [995.0928, 332.0928, 297.3468, 72.0452] \tag{54}$$

$$\vec{c} = [0.0482, 0.0464, 0.8554, 0.0114] \tag{55}$$

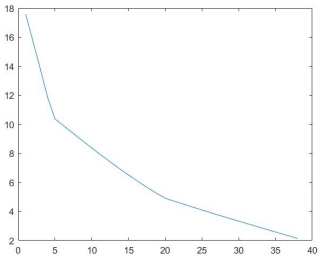


Figure 32. Trend of the distance between target and best feedback delay network model of one of the individual. Clearly, it must be a non-positive monotone function. This trend is not related to the individual that reached the best fitness, since here the line stops to a value near two, while the best one reached a distance of 0.5794. It is worth to notice that in this case the number of iterations stopped before the allowed maximum number, as a result the algorithm brought to a local minimum and the computation was interrupted since any further improvement was revealed.

$$G = 0.0449 \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & -1 & 1 & -1 \\ 1 & 1 & -1 & -1 \\ 1 & -1 & -1 & 1 \end{bmatrix} \tag{56}$$

$$fitness = 0.5794 \tag{57}$$

After the parameters were estimated, the SOUL program of the feedback delay network was initialized with the such values. A code where the excitation is accomplished by means of a slider and it is possible to control the length of the string was developed:
`https://drive.google.com/drive/folders/1QhwUqW1lSr8eOiGXp0Fq6axIf2DDqBvV?usp=sharing`
Another code where the excitation occurs by means of MIDI keyboard was developed too:
`https://drive.google.com/drive/folders/1QTf9E5n38o5CZThRCXwk23FSPVEvucGp`
Finally these are some of the recordings extracted from this model in SOUL programming language:
`https://drive.google.com/drive/folders/1HhYqMpAdJ7hKxf0Yawrc9cHOcLhT5TFA?usp=sharing`

8. HYBRID APPROACH

8.1 Algorithm Description

As described in the introduction, convolution with impulse response if the best solution to simulate the reverberation of a system, if it is a linear time-invariant system. The impulse response allows to reproduce all the nuances, but even if it is very reliable and realistic, it is not flexible

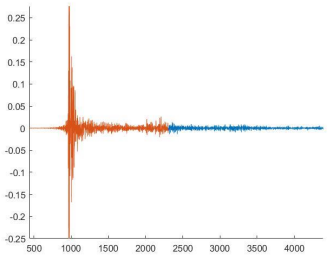


Figure 33. The orange part is the early reverberation, while the blue one is the late reverberation. The point of truncation was obtained with Gaussian truncation method. Therefore, the end of early reverberation is the first point of the first window for which the al least the 30% of the samples were outside the range $[\mu - \sigma, \mu + \sigma]$.

at small differences imply great problems to re-adapt it to the new conditions. Instead, reverberation reproduced with algorithms is less realistic and more approximative, but on the other hand changing parameters and features to re-adapt it to other conditions is very easy. For this reason, the so-called "hybrid" approach was conceived in order to exploit the potentialities of both. As a result, the early reverberation is simulated with convolution, while the late reverberation is simulated with feedback delay network. Finally, the two parts are attached together to form a complete impulse signal.

8.2 Algorithm Implementation

8.2.1 Early Reverberation Implementation

As the name suggest, the early reverberation is the referred to the first part. Therefore, a first problem is to establish what exactly are the early reverberation and the late reverberation. The early reverberation is characterized by small attenuation and well-defined direction and timbre. Conversely, the late reverberation is much more attenuate because of walls and objects absorption, it is dense and without specific direction and timbre. It is possible to divide the two components only by means of the intuition and the ear, but this approach is very subjective and arbitrary, inasmuch different people could define different point of separation. Consequently, here an algorithmic approach was developed to separate the two parts; in particular the Gaussian method was chosen [1]. It is based on the hypothesis that during the early reverberation a high percentage of the samples falls between the mean and the standard deviation, since there is less dispersion. For this reason, the signal of the reverberation is analysed by means of rectangular windowing of 20 milliseconds per window and the first window which doesn't present this feature anymore is considered to be the initial point of the late reverberation (and then the end of the early reverberation), see in figure 33 the division point for the impulse response used in this experiment.

The portion of the impulse response referred to the early

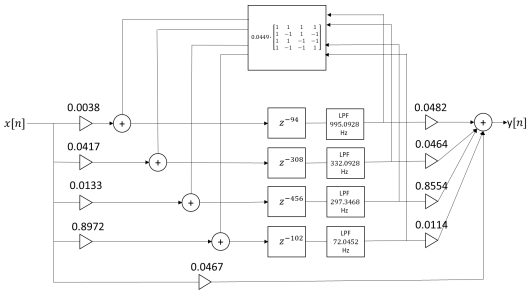


Figure 34. The scheme of the feedback delay network with the parameters estimated by means of gradient descent algorithm.

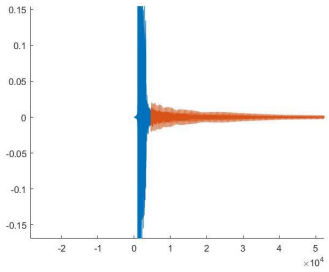


Figure 35. The blue part is the early reverberation obtained with convolution, while the orange one is the late reverberation obtained with feedback delay network.

reverberation was then convolved with the first portion of the plucked string.

8.2.2 Late Reverberation Implementation

The late reverberation, as already mentioned, is referred to the last part. It is more dense than early reverberation and reflections on walls and objects (or the sound box of the instrument, in our case of study) cause attenuation and modification of the timbre. In hybrid reverberation, this part is simulated algorithmically and here was used the feedback delay network that was previously obtained with gradient descent algorithm (see figure 34) (it was preferred to the one obtained with genetic algorithm since the distance from the target impulse response of the guitar sound box was smaller).

The plucked string sound that was not convolved for the early reverberation, was instead passed into the feedback delay network to simulate the late reverberation.

8.3 Results

The final result was obtained by attaching together early reverberation and late reverberation, obviously before the early reverberation simulated by means of convolution with target impulse response and then the late reverberation simulated through feedback delay network. The Matlab code can be found at the following link:
`https://drive.google.com/drive/folders/1ehBjQkfYAQrUxp5wfNoW3jioFepMe6e8?usp=sharing`

The plucked string with reverberation obtained is shown in figure 35.

The recorded simulation of plucked string with hybrid reverberation can be found at:
`https://drive.google.com/drive/folders/1o0aieubblCbZfo3TB88IyPF6gmDucJc0?usp=sharing`

For hybrid reverberation, only the Matlab implementation was developed, since SOUL is a programming language which was born for real time sound computing, as a result a convolution, even if relatively short, is inconvenient or even a little illogical considering the structure of the same programming language, at least for the moment, since SOUL is currently a new born programming language.

9. FUTURE WORKS

9.1 Genetic Algorithm

The improvement of the genetic algorithm is strictly related to the different tries that is possible to accomplish by changing the values of the parameters in the different stages of the algorithm. There is not a “safe” way to improve it, but only to make different trials with different settings, therefore here there are some possibilities. It is possible to decrease the percentage of survived individuals. In this case, a percentage of 35% was used, but is possible to decrease it to values as 20% or 25%. This allows to maintain only the best individuals, but on the other hand it would increase the computation cost because of a greater number of individuals generated in mating function. In mating and mutation functions would be possible to search new kind of transformations in order to maintain as much as possible the fitness of the parents also in case there is not an improvement. Here, an average function was used for mating and a random variation of parameters in a given range for mutation. It is not proved that, even if parents have a good fitness, the children will have a good fitness as well (the trend, anyway, tends to improve as shown in figure 21). A different percentage and a different range for degeneration conditions could be used. In this case, with a percentage of 95% of individuals in a fitness function range of 0.1, the degeneration never occurred. Different attempts with different parameter settings can be used to find the best model possible.

9.2 Gradient Descent Algorithm

Gradient descent algorithm is less articulated than genetic algorithm since there are less stages, as a result the parameters to change and the different settings are in a smaller number. Anyway, a work for the future could be find a way to “move” each element of the damping matrix. Here, as explained, not to lose the condition of unitary matrix (that implies the stability of the damping), it was decided to apply the row column multiplication for other unitary matrices in order to diversify the damping matrix. Anyway, this is not really a “movement” as intended for the other parameters. Consequently, a way to move the damp-

ing matrix would be an improvement of the algorithm. Another improvement could be stopping the execution of the search of the minimum only when the distance from the target doesn't change anymore, without establish a priori a maximum number of iterations. In this way, from a starting point, the algorithm would always converge to a minimum (that could be local or global), without stopping the descent in a halfway point.

10. CONCLUSIONS

In the future, the use of algorithmic reverberation will be even more spread because of its production cheapness (no recordings are needed to implement it), its flexibility and the always more precise reproduction. The flexibility is due to the possibility to change the parameters easily to obtain a new reverberation, without the necessity to have new recordings. This is not possible instead for the convolution simulation, where new impulse responses are necessary to change the parameters of the model. The improvement of artificial intelligence algorithms, such as the genetic algorithm and the gradient descent algorithm explained here, for the estimation of the model parameters will be fundamental in order to obtain always more reliable and realistic results. Another important factor to consider is the reusability of these algorithms to obtain different reverberations. For example, here these algorithms were used to estimate the parameters of the sound box of a guitar, but in the future the same algorithms could be used to simulate the reverberation of other systems such as a room or the sound box of another musical instrument.

11. REFERENCES

[1] M. Steimel, "Implementation of a hybrid reverberation algorithm," 2019.

[2] J. S. Coggin, "Automatic design of feedback delay network reverb parameters for perceptual room impulse response matching," Ph.D. dissertation, University of Miami, 2015.

[3] T. H. Park, *Introduction to digital signal processing computer musically speaking*. World Scientific Publishing Company, 2010.

[4] V. Välimäki, J. Parker, L. Savioja, J. O. Smith, and J. Abel, "More than 50 years of artificial reverberation," in *Audio engineering society conference: 60th international conference: dreams (dereverberation and reverberation of audio, music, and speech)*. Audio Engineering Society, 2016.

[5] M. R. Schroeder, "Natural sounding artificial reverberation," *Journal of the Audio Engineering Society*, vol. 10, no. 3, pp. 219–223, 1962.

[6] D. Rocchesso, "Maximally diffusive yet efficient feedback delay networks for artificial reverberation," *IEEE Signal Processing Letters*, vol. 4, no. 9, pp. 252–255, 1997.

[7] S. Browne *et al.*, "Hybrid reverberation algorithm using truncated impulse response convolution and recursive filtering," *University of Miami Master's Thesis*, 2001.

[8] "Matlab," https://se.mathworks.com/products/matlab.html.

[9] "Soul," https://soul.dev/.

[10] "juce forum," https://forum.juce.com/t/soul-lang/30480/182.

[11] S. Bilbao, *Numerical sound synthesis, finite difference schemes and simulation in musical acoustics*. John Wiley Sons, 2009.

[12] "dsp.stackexchange.com," https://dsp.stackexchange.com/questions/54086/single-pole-iir-low-pass-filter-which-is-the-correct-formula-for-the-decay-coe?fbclid=IwAR0PkeIu930sHVQbPASkid-kj81SEzGIb0t6lyq1RndglKtASe9B77gizpQ.

[13] D. Whitley, "A genetic algorithm tutorial," *Statistics and computing*, vol. 4, no. 2, pp. 65–85, 1994.

[14] "introduction to genetic algorithm," https://towardsdatascience.com/introduction-to-genetic-algorithms-including-example-code-e396e9

[15] M. Chemistruck, K. Marcolini, and W. Pirkle, "Generating matrix coefficients for feedback delay networks using genetic algorithm," in *Audio Engineering Society Convention 133*. Audio Engineering Society, 2012.

[16] I. N. Herstein, *Algebra*. Editori Riuniti, 1982.

[17] A. Frigeri, S. Adami, A. Cherubini, C. Nuccio, L. Mauri, and E. Rodaro, *Logica e algebra esercizi svolti e richiami di teoria*. Apogeo, 2018.

[18] D. Mandrioli and P. Spoletini, *Informatica teorica*. CittàStudi, 2011.

[19] A. Cauchy, "Méthode générale pour la résolution des systemes d'équations simultanées," *Comp. Rend. Sci. Paris*, vol. 25, no. 1847, pp. 536–538, 1847.

[20] C. Lemaréchal, "Cauchy and the gradient method," *Doc Math Extra*, vol. 251, p. 254, 2012.