Automatic Generation of Sound Synthesis Techniques

by

Ricardo A. García

B.S., Electrical Engineering (1996)
Universidad Pontificia Bolivariana

M.S., Music Engineering Technology (1999)
University of Miami

Submitted to the Program in Media Arts and Sciences,
School of Architecture and Planning,
in partial fulfillment of the requirements for the degree of

Master of Science in Media Arts and Sciences
at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 2001

Author_____
Ricardo A. García
Program in Media Arts and Sciences
August 21, 2001

Certified by _____
Barry L. Vercoe
Professor of Media Arts and Sciences
Thesis Supervisor

Accepted by_____
Dr. Andrew B. Lippman
Chair, Departmental Committee on Graduate Students
Program in Media Arts and Sciences

Automatic Generation of Sound Synthesis Techniques

by

Ricardo A. García

Submitted to the Program in Media Arts and Sciences,
School of Architecture and Planning,
on August 21, 2001, in partial fulfillment of the
requirements for the degree of
Master of Science in Media Arts and Sciences

ABSTRACT

Digital sound synthesizers, ubiquitous today in sound cards, software and dedicated hardware, use algorithms (Sound Synthesis Techniques, SSTs) capable of generating sounds similar to those of acoustic instruments and even totally novel sounds.

The design of SSTs is a very hard problem. It is usually assumed that it requires human ingenuity to design an algorithm suitable for synthesizing a sound with certain characteristics. Many of the SSTs commonly used are the fruit of experimentation and a long refinement processes.

A SST is determined by its "functional form" and "internal parameters". Design of SSTs is usually done by selecting a fixed functional form from a handful of commonly used SSTs, and performing a parameter estimation technique to find a set of internal parameters that will best emulate the target sound.

A new approach for automating the design of SSTs is proposed. It uses a set of examples of the desired behavior of the SST in the form of "inputs + target sound". The approach is capable of suggesting novel functional forms and their internal parameters, suited to follow closely the given examples.

Design of a SST is stated as a search problem in the SST space (the space spanned by all the possible valid functional forms and internal parameters, within certain limits to make it practical). This search is done using evolutionary methods; specifically, Genetic Programming (GP). A custom language for representing and manipulating SSTs as topology graphs and expression trees is proposed, as well as the mapping rules between both representations. Fitness functions that use analytical and perceptual distance metrics between the target and produced sounds are discussed.

The AGeSS system (Automatic Generation of Sound Synthesizers) developed in the Media Lab is outlined, and some SSTs and their evolution are shown.

Thesis Supervisor: Barry L. Vercoe
Title: Professor of Media Arts and Sciences

Automatic Generation of Sound Synthesis Techniques

by

Ricardo A. García

Thesis Reader_____

Dr. Joseph A. Paradiso
Principal Research Scientist
MIT Media Laboratory

Thesis Reader_____

Dr. Una-May O'Reilly
Research Scientist
MIT Artificial Intelligence Laboratory

ACKNOWLEDGMENTS

This thesis leaves more unsolved questions than answers. That was the goal: to immerse myself in the problem that I chose, and to find out that there are still many things that have to be explored. But during the process of understanding the problems and proposing solutions, I learned, and learned things well.

The most important thing that I have learned during my time at MIT is that there are more things to be explored and solved in the world than I used to believe. We are just starting to understand this world, and we are now capable of even creating new worlds of knowledge that claim to be studied.

There are many people and institutions that have contributed in some way to my personal growth and the development of this research:

- MIT Media Lab and the Digital Life consortium, for giving to me the opportunity and the means to do this research.
- My advisor, Barry Vercoe and the past and present members of the Machine Listening Group, for their example
- Paris Smaragdis for encouraging me to defy the world, even though the consequences are strong.
- Youngmoo Kim, for his support in the analysis by synthesis ideas.
- Wei Chai, my officemate, for supporting my music, phone talks, and jokes!
- Rebecca Reich for her role of technical editor in the final manuscript, and counselor in tough times.
- Connie, Elizabeth and Orla, that helped me to not to go crazy (crazier) with so many details in life!
- My thesis reader, Joe Paradiso, for his inspiring comments
- My thesis reader, Una-May O'Reilly, who I consider a great advisor, not just for the enormous guidance with evolutionary computation; but specially for her support in this complex fitness landscape called student life.
- Professor Withman Richards, for his ideas about models, structures and representations. The original idea for this thesis started as a project for his class.
- My friends at MIT and Boston, those who have shared good and bad times with me.
- My classmates and friends at University of Miami: keep trying to take over the world with our music engineering.
- Professor Ken Pohlmann, for his insightful guidance.
- My professors at Universidad Pontificia Bolivariana in Medellin, Colombia; for nourishing my beliefs that audio is high-level engineering.

Special dedications in Spanish:

- A mis padres, que me enseñaron a soñar, y a luchar fuertemente por alcanzar mis sueños.
- A mi familia, que siempre ha estado conmigo, así la distancia nos separe.
- A mis amigos, que siempre soñamos en hacer cosas grandes, y hoy, hacemos cosas grandes que nos permiten soñar.
- A aquellos amores que, así sea por un solo instante, llenaron mi vida; y por los amores que seguirán inspirándome.
- Dedico esta tesis a todos aquellos que saben que la música es magia que llena y eleva el espíritu; y cada cosa que hacemos por elevar el espíritu tiene un poco de divino.
- También, un aplauso por las arepitas de chócolo con quesito, especialmente si son de un rancherito en las palmas.

TABLE OF CONTENTS

1 INTRODUCTION

1.1 CONTRIBUTIONS

This thesis describes the concepts needed to develop an approach for *automating the design of Sound Synthesis Techniques (SSTs)*. Key ideas introduced with this research include:

A custom developmental representation of a Sound Synthesis Technique: Sound synthesis Techniques are usually represented as topology graphs (cyclic graphs). A mapping from an expression tree (acyclic graph) that encodes the "development" of an embryonic topology is proposed and used. This representation allows easy management and manipulation of SSTs, especially when used in conjunction with evolutionary methods.

Design as a search in the SST space: The space spanned by all the possible functional forms and internal parameters for SSTs is introduced. Design is treated as a search in the SST space.

Evolutionary methods to search the SST space: The advantage of using evolutionary methods to search the SST space is analyzed. A Genetic Programming technique is outlined and used as the core search method in the developed system.

Analytical and perceptual fitness Functions: The performance of suggested SSTs is measured using custom fitness functions. Two analytical and one perceptual fitness function are suggested and used.

Automatic Generation of Sound Synthesizers (AGeSS) system: A set of computer programs and scripts termed AGeSS was developed to show an empirical proof of the concepts introduced in this research. Different experiments and their results are outlined.

1.2 MOTIVATION

In a very general sense, a sound synthesizer is regarded as any device capable of producing sound. Digital computers can be used to produce digital sound samples using digital to analog converters (DAC). In electronic music parlance, a digital sound synthesizer is an algorithm implemented in a digital computer, with the goal of producing digital sound samples (waveforms). These algorithms for sound generation are termed *Sound Synthesis Techniques (SSTs).*

An SST can be decomposed into a *functional form* and *internal parameters*. The functional form describes the relationship between the functions and elements in the algorithm, while the internal parameters are variables that take a particular value at the moment of implementation of the algorithm (depending on the desired behavior). SSTs are usually represented using *instruction lists* or *topology graphs* (flow diagrams). "Classic" SSTs are a set of algorithms that have been used through the years in digital and analog synthesizers for emulating the sound of classical instruments, or to create totally novel sounds. Some of them have been studied in depth by researchers and musicians.

Design of a SST is customarily limited to the selection of a functional form from a set of algorithms (i.e. “classic” SSTs) followed by application of a mathematical technique for estimation of the internal parameters to match a target sound. The design of SSTs, more specifically their functional form, is a very hard problem. It is usually assumed that it requires human ingenuity to design an algorithm suitable for synthesizing sound with certain characteristics. Many of the SSTs commonly used are the fruit of experimentation and a long refinement processes.

The design of a SST is driven by the *design specifications*. They usually include a desired configuration for the number and type of inputs, some kind of error metric to measure the performance of the SST, and some implementation considerations (i.e. complexity of the algorithm). One of the most important aspects in design of a SST is the number and type of inputs. Some SST algorithms perform well in theory, but require a high amount of control information that makes them not very practical. Also, the meaning of the inputs plays an important role when using a SST. It makes more sense to use inputs related to “real world” parameters like brightness, or depth, than more obscure and difficult to understand parameters.

Our goal is to propose a general approach capable of suggesting valid functional forms and internal parameters for a SST to synthesize a target sound, using a known set of inputs (time varying signals). This problem is related to the system identification, or symbolic regression problem stated in control theory. The inputs and outputs of the system are known, but the system is unknown.

1.3 BACKGROUND AND RELATED WORK

Horner et al. (Horner, Beauchamp et al. 1993) proposed an approach for automating the internal parameter estimation of FM synthesizers using evolutionary methods, in particular Genetic Algorithms (GA). Before this approach, the selection of parameters for FM synthesizers was usually realized by trial and error, using some analysis methods to compare (by a human) the sounds produced by the synthesizer and the target. The goal of the algorithm is to find parameters for a fixed functional form synthesis technique (single modulator, multiple carriers FM synthesis) to match the spectrum of a target sound with harmonic partials. The search for the parameters was done using a genetic algorithm, that represented them as a fixed length bitstring, and using genetic operations it was possible to explore the parameter-space of the problem.
A fitness metric that calculated the Least Squared error between the magnitude spectrums of the output and target sounds was used. This measure was performed in selected frames across the duration of the sound, usually with more frames in the attack section.

This research is of relevance to us in indicating the possibilities of using evolutionary methods for exploring complex spaces; more specifically, sound spaces. The fitness metric proposed is also of interest for us. It was the original fitness function used in the first stages of our research. Also, this study shows in an indirect way the limitations when using a single functional-form when designing SSTs.

Johnson (Johnson 1999) proposed an interesting approach to use evolutionary methods and human listeners in an interactive system to explore the parameter space of (*Fonction d’Onde Formantique*) FOF synthesis. In his research, he proposes an interactive interface that lets the

user select the “best” synthesizers from a population, and the system “proposes” new individuals for the next population. This approach proved to be efficient for problems involving the estimation of many complex parameters (FOF synthesis has many non-intuitive parameters to explore).

Wehn (Wehn 1998) has an approach that includes part of the functional form of the synthesis techniques in the search space. His approach describes a set of basic functional elements (noise, sinusoid, filter) and a connection matrix that will ultimately outline the functional form of the synthesizers. The fitness function is similar to the one used by Horner. A GA is used to explore the connections and their weights between elements. In part, the goals of When’s research are shared with our research, but the methodology is very different. The representation of the problems is the main difference, followed by the type of manipulations done to the individuals and the usage of the fitness functions.

1.4 APPROACH

For us, the space of the SSTs is defined as the space of all the possible combinations of a given set of functional elements and their connections. Design of a SST is regarded as a search in the SST space. The goal is to “find” a point in the SST space that minimizes the error in the fitness function, and therefore, satisfies the specifications of design. Points in the SST space define a functional form and internal parameters for a specific SST.

The search in this space is performed using a class of evolutionary computation method called Genetic Programming (GP). GP has shown outstanding empirical performance in searching complex multidimensional spaces (Koza 1992; Koza 1994; Koza 1999).

Custom SST representations in the form of *topology graphs* and *expression trees* are developed along with their required mapping rules. Topology graphs are the most widely used representation for SST, but they are difficult to manipulate. Expression trees facilitate the level of manipulation required to use GP for exploring the SST space.

Different types of fitness functions are analyzed, and a perceptual fitness function is proposed to guide the search in the SST space.

An implementation of the approach called AGeSS (Automatic Generation of Sound Synthesizers) is outlined. The main software elements and their relationships are described. Experiments with AGeSS and their results are explained.

2 SOUND SYNTHESIS TECHNIQUES (SST)

2.1 SOUND SYNTHESIS

In the most general terms, *to synthesize* a sound is to produce a sound using artificial means. When speaking in the context of electronic music and sound production, a sound synthesizer is usually regarded as an electromechanical device capable of producing sound in a controlled manner. The most generic device capable of doing this is a loudspeaker connected to an amplifier and a sound source. In the early days, sound synthesizers were comprised of analog electronic circuits that exchanged voltage and current signals between filters, amplifiers, oscillators and other simple (but powerful) components. Even though analog synthesizers could emulate just partially the sound of classic acoustic instruments, artists have used their distinctive sound in very creative ways (Roads 1994).

In more recent years, with the advent of inexpensive and ubiquitous digital computers, a whole new trend of digital music synthesizers has flourished. A digital computer can manipulate sound in the form of "sound samples". Sound is sampled into digital form using an Analog to Digital Converter (ADC), and converted back to analog audio using a Digital to Analog Converter (DAC). This process (sampling) creates a stream of digital audio samples (waveform) that can be modified using computer programs.

2.1.1 Algorithms

"A procedure consisting of a finite set of unambiguous rules, which specify a finite sequence of operations that provides the solution to a problem, or to a specific class of problems, is called an algorithm" (Kronsjö 1987). Algorithms (or techniques) are usually implemented as computer programs in digital computers. With all this in mind, it is straightforward to define a Sound Synthesis Technique (SST) as: *"a step-by-step set of instructions designed to produce sound samples."*

2.1.2 Functional form and internal parameters:

The *functional form* (structure) of an algorithm is defined by the characteristic relationship between its constitutive functional elements. The *internal parameters* are the numbers that could take a different value for a specific implementation of an algorithm, but their change doesn't alter the functional form of the algorithm. This can be better seen with one example. In the simple relation:

$$y = 2\sin(x) + 0.5 \qquad (1)$$

The functional form is given by $y = A\sin(x) + B$, and the internal parameters are *A* and *B*, that in the particular example have the value of 2 and 0.5 respectively.
Usually, the functional form of an algorithm is specified at design time, but the internal parameters are left to be specified at implementation time with values that will fulfill the requirements of the particular problem. During this research, the term SST will refer mainly to the actual functional form of the algorithm, without concern about whether or not the value of the internal parameters is known.

2.1.3 Inputs and outputs:

Every meaningful algorithm has zero or more inputs and one or more outputs. The inputs are defined as known quantities that are given to the algorithm, and the outputs are quantities that are computed when executing it. For the particular case of SSTs, the outputs are usually in the form of digital sound samples. Inputs of SST are divided in these two types:

- *Static*: Fed into the algorithm at initialization time, and kept unaltered during the operation of the SST. Internal parameters will be treat as part of the input given to a SST.
- *Time varying*: Change their value during the execution of the SST. Usually called *control signals.*

One of the biggest challenges in the field of sound synthesis is to devise SSTs where the inputs are related to "real world" parameters, and have an intuitive meaning of their function (and effects) in the operation of an SST.

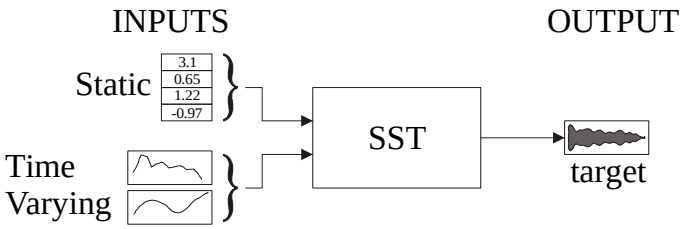


Figure 1 Inputs (static and time varying) and outputs (target) for a SST

2.2 LANGUAGE AND REPRESENTATION

Algorithms are meant to be conceptually transparent to humans until translated into a format that can be understood by a machine for execution. It is assumed that human-level abstractions are "easy" for manipulation by humans, and later, can be implemented easily on a machine. That is, the language for an algorithm that a human designs is intuitive for humans.

An algorithm language must express functional form and functional elements. It is composed of primitive elements that isolate levels of detail away from the actual user. These primitive elements can be combined to create compound and hierarchical representations of functional form and functional elements. In choosing (or creating) a language, one has to draw a line demarking primitive elements and hidden details from higher level composable elements, while understanding that this line influences the nature of the functional form and functional elements in an algorithm. Later in this chapter we will analyze some functional form and functional elements used in several SSTs and propose a set of "general functional elements" that could be used in the design and description of SSTs.

2.2.1 Instruction list (pseudocode) and formulas:

The definition of algorithm suggests that it can be represented as an ordered list of instructions. It is important to remember that each instruction should have a defined meaning (an specific action associated with it). If the instructions are in plain written English, the representation is usually called "*pseudocode*" (Juliff 1986). If the instructions are in a computer language it is called

"*source code*", and they have to be pass thru a "*compiler*" that will translate them into a computer program that can be run in a digital computer. Mathematical formulas are special cases of this type of representation. They express a step-by-step procedure that can be unambiguously evaluated to get an answer. This is a highly symbolic representation, but the meaning of each element and the rules of evaluation are defined beforehand as mathematical operations.
In example, the formula of equation (1) can be expressed with the instruction list:

```
evaluate sin(x)
assign result to P
evaluate 2*P
assign result to Q
evaluate Q+0.5
assign result to Y
output Y
```

Often, the evaluation rules for formulas are not completely specified, making some evaluations ambiguous. In the previous example, the formula doesn't specify the evaluation order of the two addends; but this is completely specified in the instruction list.

2.2.2 Topology graph (flow diagram):

Graphic representations of algorithms are always an invaluable tool for representing them. They are easy to manipulate and present a lot of information in a compact way. One of their strongest assets is that the relationship between elements is clearly shown. The functional elements are represented by boxes, and the data flow (execution order) is represented by directed arrows connecting the boxes.

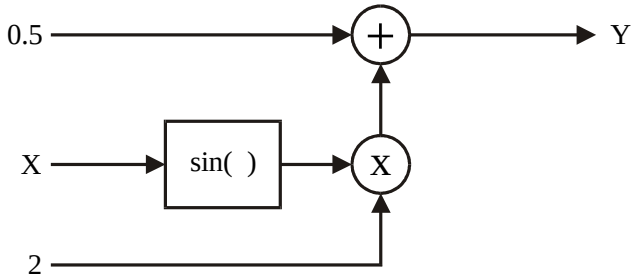


Figure 2 Flow diagram (topology graph) for equation (1)

Flow diagrams are often called *topology graphs* in the sound synthesis world, and we will employ this terminology from now on. The reason is because in the early days, the sound synthesizers were implemented with physical devices interconnected using patch cords, and their configuration was recorded as a "topology" or "patch". An advantage is that the topographical distribution of functional elements and their connections offers an intuitive approach for working with SSTs.
In this document, the term SST and *topology* would refer to the same concept and they could be used interchangeably.

2.2.3 Equivalence in representations:

The above two types of algorithm representations are equivalent: they are simply different schemes to represent the same information. Any algorithm expressed in terms of one can be mapped to the other without lost of information.
The manipulations that are possible in one representation have their equivalent counterparts in the other representations, but usually some manipulations are easier to perform in one of them.

2.3 "CLASSIC" SOUND SYNTHESIS TECHNIQUES:

So far we have established that SSTs are algorithms that produce sound samples, implemented in digital computers, and are represented by instruction lists, formulas or topology graphs. Now, we can focus on more detail in the functional elements that compose a SST. A good starting point is to analyze the termed *classic SST*. They are a set of techniques that have been used in the computer music field for many years, and some of them studied in depth by researchers and musicians.

It is important to be clear about the intentions with this analysis: the goal is to describe the *functional form* of each SST family and the main *functional elements* that compose them. The suggested taxonomy tries to organize them in families that share functional forms and functional elements. The goal is not at any time to make an exhaustive bibliographical research in the theory, uses and implementation of the mentioned SSTs, neither to have a comprehensive taxonomy with all the SSTs available in the literature.
The analyzed SSTs are not usually used in isolation. Commercial (practical) sound synthesizers employ a mixture of these techniques to improve the sound quality and performance. For a more complete taxonomy and explanation on the SST it is useful to read (Depoli 1983; Roads 1994; Boulanger 2000).

Some concepts that should be explained before the proposed taxonomy are:
Sound source: an object or functional element that "produces" a sound output.
Digital oscillator: a particular type of source that creates an output that repeats itself over time (oscillates). The origin of the sound samples in an oscillator can come from a mathematical formula that is evaluated every time that an output is needed, or from reading a wavetable stored in memory.
Wavetable oscillator: this is the most common way of implementing a digital oscillator. A wavetable with the desired output values is stored in memory. The process of reading it is called "indexing", and involves knowledge about what was the previous sample index read, and the "update" for the next index to be read. The indexing update can change over time.
Digital filter: a particular operation on a waveform that alters the magnitude of the frequencies present in the input.

The following SSTs are described using topology graphs, and mathematical formulas when possible.

2.3.1 Additive techniques:

Any method that synthesizes sound by adding different sound source objects together can be considered under the category of additive synthesis. There are three main aspects that help to differentiate the many types of additive synthesis methods: type of objects to add, role and type of inputs/internal parameters and finally, the analysis technique used to derive the actual objects and the internal parameters.

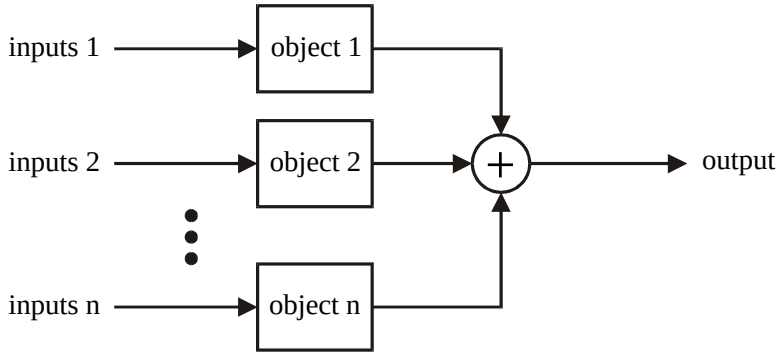


Figure 3 General Additive synthesis

2.3.1.1 Fourier:

One of the most important and used tools in sound synthesis comes from the so called Fourier Transform (Oppenheim, Schafer et al. 1999). The Fourier transform states that any sequence (digital sound samples in our case) can be fully represented as a superposition of infinitesimally small complex sinusoids of different frequency, amplitude and phase. The synthesis formula for the Fourier transform is given by:

$$x[n] = \frac{1}{2p}\int_{-p}^{p} X(e^{jw})e^{jw}\,dw \qquad (2)$$

where the $X(e^{jw})$ are complex coefficients that state magnitude and phase for the corresponding sinusoid of frequency w. This synthesis formula can be represented by a topology graph as shown in Figure 4.

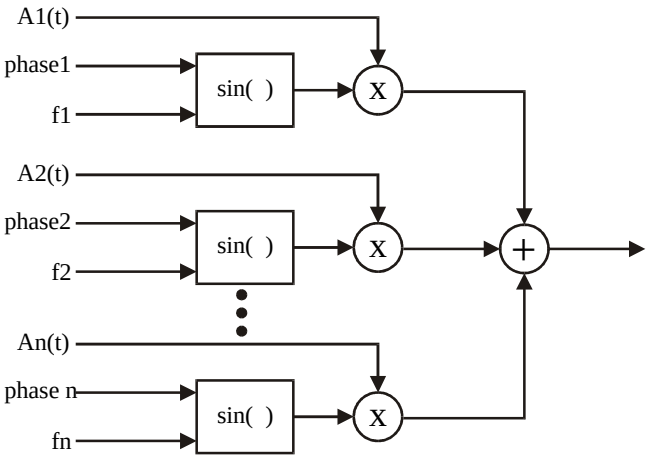


Figure 4 Fourier Synthesis. Additive synthesis with fixed frequency sinusoidal oscillators.

The sound sources are sinusoidal oscillators with fixed frequency and phase. The amplitude of the oscillators is usually varied over time to simulate the varying characteristics of the spectra of real acoustic instruments.
The importance of Fourier synthesis/analysis is related with the fact that there are very efficient tools (mathematically and computationally) to realize direct and inverse Fourier transformations.

The types of functional elements present in Fourier synthesis are:

- Sinusoidal oscillators with variable amplitude and fixed phase and frequency
- Addition
- Multiplication

2.3.1.2 Peak Tracking and Spectral Modeling Synthesis:

Unlike Fourier analysis where the goal is to represent a time signal with fixed frequency sinusoids, peak tracking finds a representation of a signal as the sum of sinusoids that vary their amplitude and frequency over time. The synthesis formula is very similar to Fourier synthesis, but the sinusoids are allowed to change their frequency over time. Usually, the number of sinusoids required to represent a given signal is significantly lower than in Fourier synthesis.

$$s(t) = \sum_{r=1}^{n} A_r(t)\cos(q_r(t)) \qquad (3)$$

A variant of peak tracking called Spectral Modeling Synthesis (SMS) (Serra and Smith 1990), includes some components that are difficult to express as a sum of sinusoids. This approach represents a signal as a sum of sinusoids that vary their amplitude and frequency over time, plus a time-varying stochastic component (noise).

$$s(t) = \sum_{r=1}^{n} A_r(t)\cos(q_r(t)) + e(t) \qquad (4)$$

These types of synthesis can be represented by the topology graph:

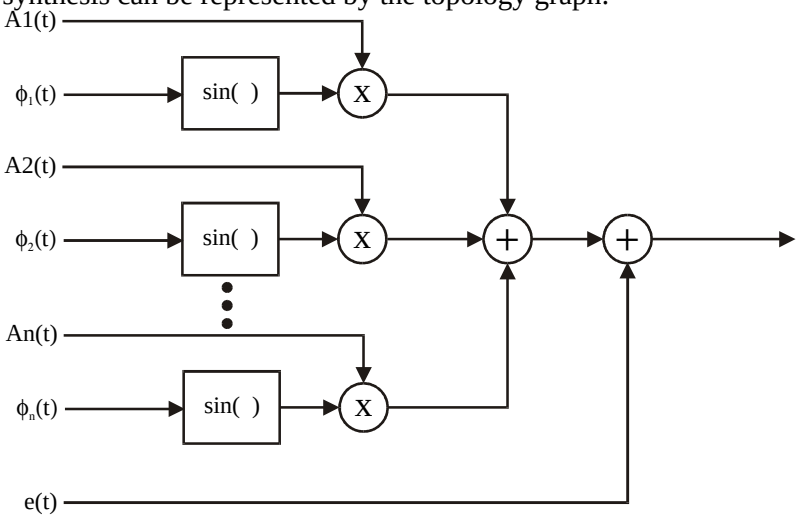


Figure 5 Peak tracking and SMS synthesis

The types of functional elements present in peak tracking and SMS techniques are:

- Sinusoidal oscillators with variable amplitude, frequency, initial phase.
- Addition
- Multiplication
- Stochastic signal (noise generator) (SMS)
- Time varying filter (to shape the noise) (SMS)

2.3.1.3 Sampling, Multiple wavetable, wavestacking:

Sound sources have been so far represented as simple oscillators (sinusoids) with fixed or variable frequency, amplitude and phase. A source can be thought as a function that returns a value every time that is executed. This value is usually calculated in one of two ways: evaluating a mathematical function (i.e. *sin(x)*) or by reading from a table stored in memory (usually called *wavetable*)

-Sampling

The process of recording a sound into a waveform is termed *Sampling*. As a synthesis technique, sampling reproduces the waveform stored in memory using a source that reads a wavetable. This source can be implemented as a wavetable oscillator that reads the wavetable just one loop. An advantage of sampling is that it can reproduce "any" waveform, as long as it is stored in memory.

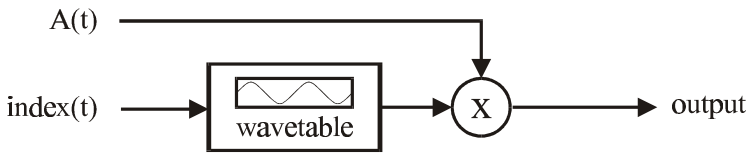


Figure 6 Sampling as a controlled reading from a wavetable

-*Wavestacking*
This technique is built on the concept of sampling, but relies on the addition of multiple waveforms to form the final output. For our purpose it can be regarded as a similar method to Fourier synthesis, but having sources with waveforms of different types (not just sinusoids).

- *Multiple wavetable*
The idea here is to have "chunks" of sound that play in succession one after the other. To achieve this effect the amplitude of several sources is cross-faded to allow just one wavetable to be played at a given time.

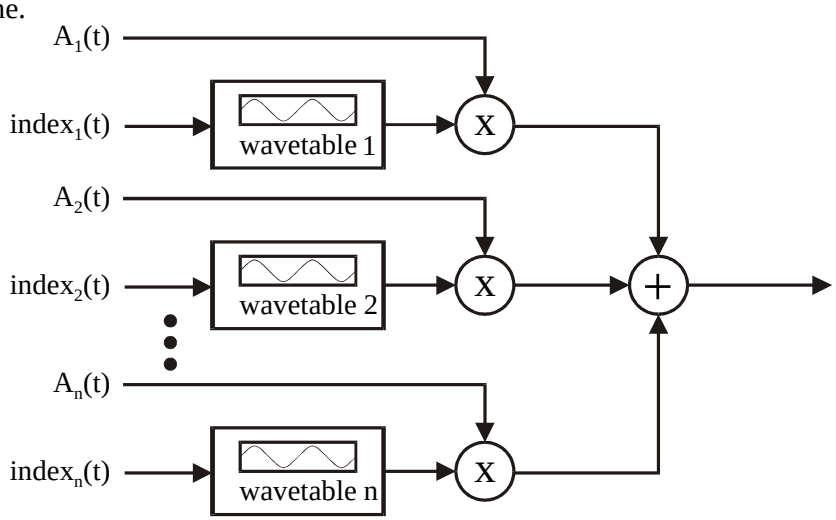


Figure 7 Multiple wavetable, wavestacking and ICA. Multiple Wavetable simply adds together different wavetables. Wavestacking incorporates more complex time envelopes. In ICA, each wavetable was extracted from an optimization procedure that separated a waveform in several "independent" components.

-*Independent Component Analysis ICA:*
Several mathematical analysis tools aim to decompose a sound in elements that can be added, multiplied or used to re-create the original sound (or a close approximation). From all these tools, ICA is one of the most commonly used (Casey 1998). Its most attractive feature is that it can decompose a sound in statistically "independent" components (this translates perceptually as elements "belonging together"). These components can be added together to form the reconstructed sound. It is possible to modify some aspects of some of the components in ways that preserve the structure of the original sound.
ICA synthesis is a straightforward additive synthesis, using the "objects" that were found in the analysis stage as sources in the wavestacking method. The analysis for ICA requires an iterative optimization process that can lead to diverse results.
Equation (5) shows a signal *X* formed by the additive sum of some independent components c_i and some noise components g_j.

$$X = \sum_{i=1}^{r} c_i + \sum_{j=1}^{k} g_j \qquad (5)$$

The main types of functional elements present in additive synthesis techniques are:

- Wavetable oscillators: Instead of a sinusoidal oscillator, these read from a wavetable with "any" waveform stored in memory.
- Addition
- Multiplication.

2.3.2 Subtractive Methods:

The previous methods used addition of "simple" waveforms to form the desired sound. Subtractive synthesis uses the opposite principle: it creates a very complex sound, and removes unwanted elements from it. It makes heavy use of digital filters to shape the frequency content of a complex source signal (usually rich in frequencies). The frequency response of the filters is selected to match the spectrum of the desired sound, and the source is driven to generate a rich spectrum. The types of sound sources found in Subtractive Synthesis techniques have the special characteristic that produce complex waveforms rich in frequencies. Subtractive Synthesis usually follows the source + filter model. The filters that are proposed are linear, and quite simple in the case of the vocoder, and more complex (more poles/zeros) in the LPC case, but they can be decomposed into a system of simple filters in series or parallel (Oppenheim, Schafer et al. 1999).

2.3.2.1 Vocoders:

A bank of narrow band-pass filters (distributed along the frequency axis) is excited with a broadband signal (i.e. white noise, a square wave, a train of pulses, etc). The gain value in each filter is controlled and varied over time. With the right set of parameters this gives a very good approximation of the spectrum of the desired signal (Rabiner and Schafer 1978).

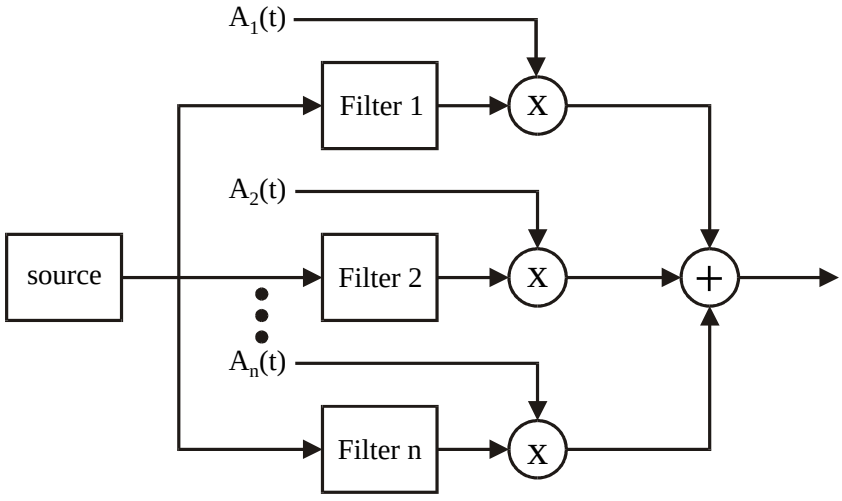


Figure 8 Simple vocoder. A source signal (rich in frequencies) is filtered by controlled bandpass filters.

2.3.2.2 Linear Predictive Coding (LPC)) synthesis

LPC implements a vocoder but replaces the narrow band-pass filter bank with a single and more complex filter that approximates the frequency response of some instrument (usually human voice, or an instrument with resonant cavities). The coefficients of this filter vary over time, and are updated to follow the changes in the spectrum of the original instrument. The types of sound sources employed to excite it vary from a pitched pulse train, to broadband noise (Rabiner and Schafer 1978)

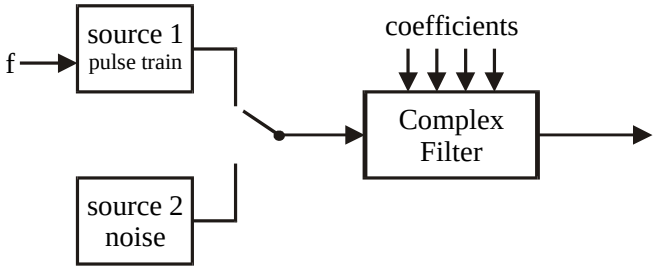


Figure 9 LPC synthesizer. Two types of sources can fed a complex filter, whose coefficients vary over time.

The main types of functional elements present in subtractive synthesis techniques are:

- Time varying sound sources: sometimes complex waveforms stored in a wavetable source.
- Filters: From simple narrowband filters with controlled gain, to more complex filters with coefficients that vary over time.
- Addition.
- Multiplication.

2.3.3 Modulation Methods:

Any property of a SST that can be varied over time is regarded as a *modulation*. In classic techniques, modulation has been used mainly to change amplitude, frequency or phase of a simple oscillator, with the objective of creating a rich and complex set of frequencies that vary over time. Most of the architecture of the modulation methods involves two (or more) controllable oscillators (usually with sinusoidal waveforms, but other simple waveforms are common: square, triangular, sawtooth).

2.3.3.1 Amplitude Modulation (AM) and Ring Modulation (RM):

As its name indicates, Amplitude Modulation modulates the amplitude of an oscillator with an unipolar time varying signal. Ring Modulation uses the same architecture, but the modulating signal used is bipolar.

$$s(t) = A(t)\sin(f) \quad (6)$$

with: $A(t) \geq 0$ (unipolar) Amplitude Modulation (AM)

$A(t)$ (bipolar) Ring Modulation (RM)

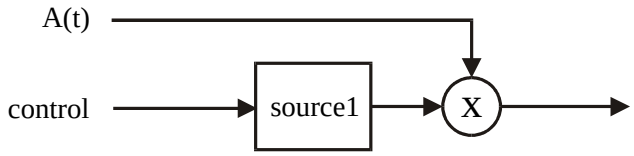


Figure 10 Amplitude and Ring Modulations (AM/RM)

2.3.3.2 Frequency Modulation (FM):

The frequency of an oscillator varies as a function of time. This synthesis technique (and its variants) have been deeply explored. Summaries of these can be found in (Roads 1994; Boulanger 2000).

$$s(t) = A\sin\left(C(t) + I\sin\left(M(t)\right)\right) \quad (7)$$

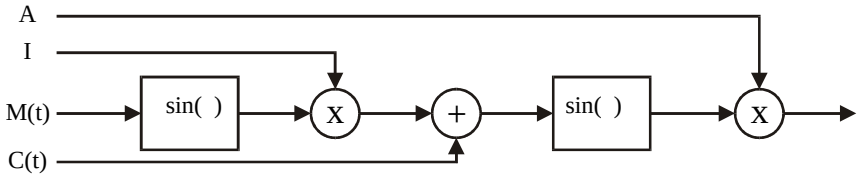


Figure 11 Frequency Modulation (FM) synthesis. One of the most used synthesis methods

FM has proved to be a very powerful (and computationally inexpensive) method for creating complex waveforms that offer very simple but meaningful controls.

2.3.3.3 Waveshaping:

This method uses a wavetable oscillator to control another wavetable oscillator. If both wavetables are sinusoids, this reduces to simple FM, but the more complex the wavetables, the more complex the produced sounds.

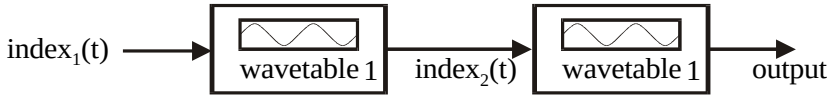


Figure 12 Waveshaping: controlled wavetable oscillators in series. Generalization of modulation techniques.

Waveshaping can include also a non-linear function instead of a wavetable module.
The main types of functional elements found in Modulation Synthesis techniques are:

- Simple wavetable oscillators (with simple waveforms like sinusoidal, square, etc) in series or parallel.
- Controlled wavetable oscillators (allow control of the way the wavetable is accessed).
- Non-linear function
- Addition
- Multiplication

2.3.4 Physical Modeling:

Physical models are mathematical models that describe the behavior of physical systems. They are used to model the behavior of the air and some mechanical elements in acoustic systems. Because of the vast field of mathematical modeling, it is not feasible to encapsulate all the possible models under one category, but it is possible to outline some of the most common techniques and tools.

2.3.4.1 Waveguides:

Waveguides are based on the solution of the wave equation, that describes the behavior of a wave in a given medium. Their implementation usually involves a delay line that simulates the transmission and reflection of a wave in a given direction of a medium.
A very basic waveguide instrument, Karplus-Strong model (Boulanger 2000), is implemented using a delay line, filters and feedback.

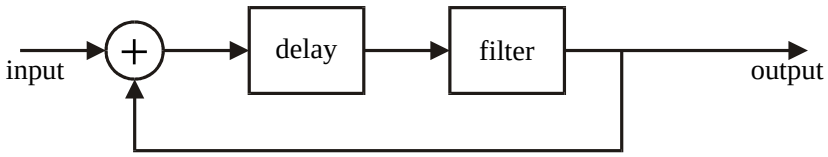


Figure 13 Generic Karplus-Strong string physical model.

2.3.4.2 Difference equations:

Physical systems are often described using differential equations, that ultimately describe the relationship between the inputs and outputs of a system. Symbolic solution of many of these equations is often very expensive computationally, not general enough, or simply not possible. Numerical methods based on difference equations give an accurate approximation of the behavior of these systems. Vibrating objects and their interaction are represented by a set of difference equations that are evaluated to produce sound samples. (Roads 1994). Difference equations are commonly used in fields like model analysis and signal processing. A common class of systems analyzed using difference equations are the Linear-Time Invariant systems where the input x(n) and the output y(n) satisfy a linear constant-coefficient difference equation of order N.

$$\sum_{k=0}^{N} a_k y[n-k] = \sum_{m=0}^{M} b_m x[n-m] \qquad (8)$$

Vast amounts of research in the analysis and use of these types of systems has been done (Oppenheim, Schafer et al. 1999). A first order system described by a simple difference equation is shown in equation (9) and Figure 14.

$$y[n] = 2y[n-1] + x[n] - 3x[n-1] \quad (9)$$

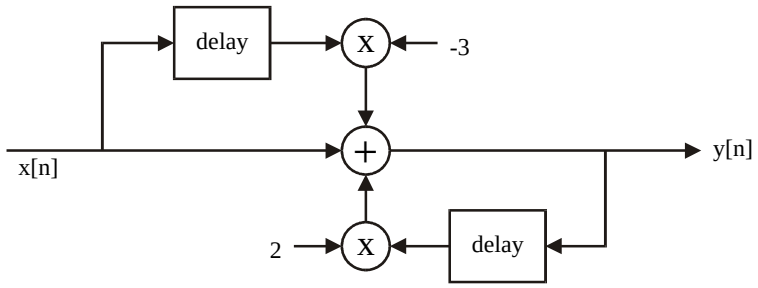


Figure 14 First order difference equation depicted as a topology graph.

The most common functional elements found in physical modeling are:

- Delay by one sample (multiple samples delays are done using several one-sample delays).
- Addition
- Multiplication

2.3.5 Summary of functional elements

This is a list of the main types of functional elements found in our taxonomy:

- Sinusoidal oscillators (variable amplitude, frequency, phase over time)
- Wavetable oscillator (variable amplitude, read index)
- Delay (memory) for one or more samples
- Controlled gain filter
- Noise generator
- Time varying filters (coefficients can change over time)
- Addition
- Multiplication

3 DESIGN OF SOUND SYNTHESIS TECHNIQUES

3.1 DESIGN:

Lets us define the design of an algorithm as *"the process that conceives the structural form and internal parameters of an algorithm, capable of producing a desired set of outputs, using a known set of inputs"*. For SSTs, which are algorithms for audio synthesis, the outputs are usually in the form of digital sound samples (waveforms), and the inputs in the form of time varying control signals and static parameters.

3.1.1 Specifications of design for SSTs:

Different elements should be specified when designing a new SST. The final design has to follow strictly some of them but it can have more error tolerance than some others.

3.1.2 Inputs

The type and number of inputs essential. The number and type of inputs can be given as part of the design specifications (i.e. the SST is a sub-module from a bigger design). The inputs to a SST can be of many types, but they are usually classified in two main groups: *static parameters* (updated at initialization time) and *control signals* (time varying signals that change their value during the evaluation of the algorithm).

One of the goals of SST design is to conceive algorithms where the inputs have "meaning" for the users. If a musician is using a SST that simulates the sound of a piano, it makes sense to have inputs that relate to the control of the "brightness" of the sound, the weight of the keys, or other piano-related variables that are present in the real world.

3.1.3 Outputs (TARGET)

For SSTs, the type of output is usually a sequence of digital audio samples (waveform), but other types of outputs could be easily specified if needed. The desired output is called *TARGET*, while the obtained output from any stage during the design is simply called *output*.

3.1.4 Error metric

All design processes need to measure the difference between desired and obtained results. In SST design the error is measured between the output and target waveforms in a given design. The error is usually measured using an *error function*, also called *fitness function* during this paper. This function computes some kind of "distance metric" between both waveforms and returns a value related to this distance. Because is a distance metric, it has the convention of being zero if both waveforms are identical, and a number greater than zero if they differ.
Human hearing is far from perfect, and it has been shown that very different waveforms (in a sample by sample sense) could sound "identical" or "similar" for a majority of listeners.

This redundancy can be exploited in the fitness function, and allow some "non perceived error" to be present in the output. In a future section we will discuss in detail some proposed fitness metrics. Therefore, the error metric (fitness function) should be completely described in the specifications of design. This is the "ruler" that will be used to decide if a design fulfills the requirements.

3.1.5 Implementation/algorithmic constraints:

Because SSTs are usually implemented in a digital computer that has limited resources, it is normal to include some implementation constraints in the specifications of design. Some of the constraints take the form of maximum allowed number of functional elements, size of the algorithm, execution time, real time capabilities, etc. Regardless of the platform where the SST will be implemented, all the designs have some boundaries and restrictions (i.e. an SST that requires infinite or too much memory is not realizable)

3.2 PROPOSED APPROACH FOR DESIGN OF SSTS

Design of an SST usually requires of two stages: first *selection of a functional form* for the SST, and then *parameter estimation* to find the right internal parameters for matching the performance of the SST to the desired target sound.

3.2.1 Classic design

In classic Sound Synthesis design a human realizes the first stage of the process. Functional forms are usually never conceived from scratch for a particular target sound. Instead, the designer selects one "template" from a set of known functional forms (i.e. the classic synthesis techniques) based on the characteristics of the target sound, and the known capabilities of the tentative functional forms.

In the second stage, the designer selects an approach for parameter estimation, and uses it to find the internal parameters that better "fit" the selected functional form to produce a sound "close" to the target sound. This part of the process has been automated with high success. (Horner, Beauchamp et al. 1993). The designer usually tries a handful of functional forms to select the one that results in a better match to the target sound.

In practice, it is common to find simple mixes of two or more synthesis techniques in a design. For example, to simulate the transient part of a sound is very complex with most of the synthesis techniques, but the steady state (sustained) part of the sound is fairly simple, so, a "hybrid" synthesis method could use sampling for the first part of the sound, and FM synthesis for the sustained. But is important to note that these kind of hybrid functional forms don't fuse in an intimate way, they just do a simple mix (usually additive) of their produced outputs.

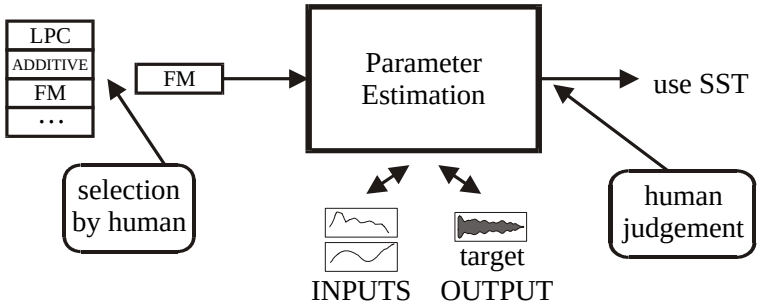


Figure 15 Classic design of SST. Human selects a fixed Functional Form from a pre-defined set (i.e. "classic" SST) and uses a parameter estimation method. Human judges if the results are satisfactory or repeats process with new functional form

A notable exception for this classic design process occurs in physical modeling, where a functional form is derived for each type of sound. The resulting functional form is usually related to the physical dimensions and properties of the system being emulated. But this physical model has to be as well handcrafted by a human designer.

3.2.2 Proposed approach for design

Our proposed approach tries to remove as much human intervention from the design process as possible. The first change (and maybe the most important) is to replace the first stage of selection of a pre-made functional form, by a "functional-form suggesting mechanism". This mechanism will suggest valid functional forms that can be tested to see if they are good or not for the desired goal. The second stage remains the same, and it consists in the parameter estimation for the "selected" functional form to try to match the target sound. Another point where the human intervention can be reduced is in the "error comparison" between the output sound and the target sound. This comparison (error function) will return a value that will be used for suggesting a new functional form, and that will try to minimize this error function. The procedure is repeated until the error falls within acceptable limits.

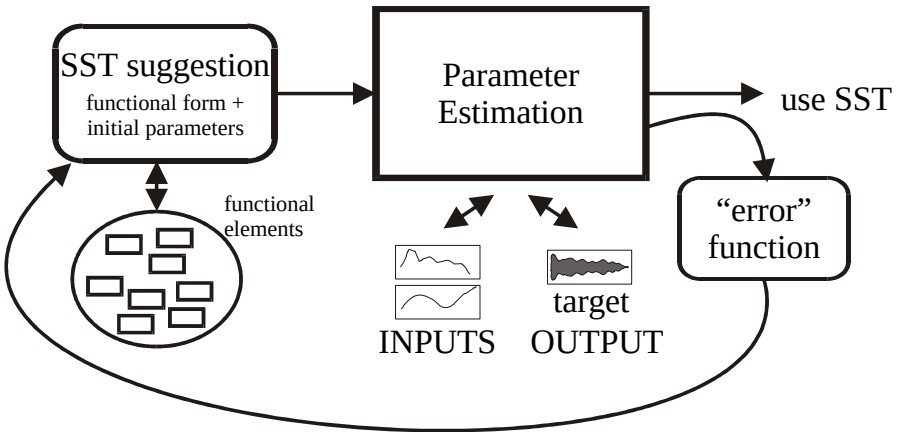


Figure 16 Suggested approach for design. Automated suggestion of Functional Forms, parameter estimation, and automated comparison of target/output sounds fed back in suggestion block.

It is important to note that the mentioned "functional-form suggestion mechanism" could operate in many forms, as long as the functional-forms that it suggests are valid. A simplistic approach could use a database of functional-forms, and just simply select one by one and wait for results. In this research, we decided to take a more adventurous path and elaborate a real "suggestion" mechanism that doesn't know precisely about the classic functional-forms, but it knows how to assemble many functional forms (classic and novel). This mechanism will be described later.

3.3 PARAMETER ESTIMATION

This stage has been extensively researched. We will give an overview of the techniques usually employed and their main goals. After this, we will introduce our approach for automation of the functional-form suggestion.

Once a functional-form has been selected (or suggested), the number and type of internal parameters remains fixed. A technique for parameter estimation can be used to find a set of values for those parameters that will reduce the error between the produced output and the target sound. This can be done using one of two approaches: *mathematical analysis* or *optimization methods*. The approach selected depends mainly in the type of functional form to be optimized and in the precision needed for its internal parameters. The fitness function (error metric) is used to guide the optimization process.

3.3.1 Mathematical analysis:

Many of the mathematical tools used to analyze signals can be used to extract meaningful information about our target waveforms, and in some cases that information can be used to infer the value of the internal parameters of a given SST. It is usual to find these tools in couples called *transforms* (Lindquist 1989). Transforms are couples of mathematical procedures that transform information (waveforms for us) from one domain into another. When they don't lose information in the process, it is possible to go from one domain into another and back without altering the original signal. When this is the case, the couple of procedures to go back and forth are called analysis/synthesis formulas.

3.3.1.1 Fourier:

Some functional-forms (SSTs) are intimately related to mathematical analysis/synthesis methods. The most notorious example is Fourier synthesis. The internal parameters for this SST are the same parameters returned by a Fourier analysis: magnitude and phase of the composing sinusoids. But Fourier analysis is a very valuable tool that can be used in conjunction with others to facilitate the parameter estimation process. A modification of Fourier analysis called Short Time Fourier tranform (spectrogram) (Rabiner and Schafer 1978) can be useful for analyzing frequency variations over time, onset/offset of components, and relative amplitudes of different components in frequency and time.
One example of the usefulness of Fourier analysis is when deciding the value of some filters to be used in subtractive methods, where a Fourier analysis can give an initial estimate of the desired frequency contour.

3.3.1.2 Cepstral Analysis:

Another popular analysis tool is called *Cepsrtal Analysis*. It gives an estimate of the broadband vs narrowband energy in the spectrum of a signal. (Rabiner and Schafer 1978) This analysis is useful to "isolate" the effects produced by a source, from the effects produced by a resonant system. It is often used in synthesis techniques that follow the source + filter model (i.e. LPC, Physical Modeling). Even though the results are just approximations of the real values, they are very good estimates.

3.3.2 Optimization methods:

Many interesting functional-forms of SSTs have no associated mathematical analysis tool that could be used to estimate their parameters. In these cases, optimization methods borrowed from systems, signals and control theory are useful. They are sometimes referred to as *search methods*, because they search for an optimal set of parameters in the parameter space associated with the SST. (Gershenfeld 1999).

3.3.2.1 Parameter space:

Because the number of parameters is fixed (once the functional form of the SST has been selected), the space spanned by these parameters is fixed.
The size of this space (total number of different possible combinations) can be computed as follows. It is assumed that the SST will be implemented in a digital computer and the parameters will be represented by a fixed bit-size word. Then, the total number of possible different combinations *C* of parameters is given by:

$$C = 2^{BP} \qquad (10)$$

with P number of parameters, and *B* the number of bits per parameter.

Using equation (10), for a SST with 10 parameters, and each parameter with 12 bits, a total of $2^{10\times12} = 2^{120} \approx 1.3\times10^{36}$ combinations can be found in the space.

3.3.2.2 Fitness Function (error function):

Each point in parameter space corresponds to a set of parameters that can be used with the selected SST to produce an output sound. This output can be fed into the fitness function specified for the optimization (error function) and compared to the desired target sound to compute an error value. This value is an indirect measure of the performance of the SST in that particular point of the parameter space. The task of the optimization method can be seen as "to find a point in parameter space where the related error value is minimal". If we map all the points of the parameter space into the related space of the error values, the resulting is called "fitness landscape" or "error landscape"(Press 1992; Gershenfeld 1999). It is possible to use the value of the error in the fitness landscape to decide where in the space are better points, and in that way guide the optimization process. But this is only possible if the fitness landscape is smooth. If the fitness landscape is rough, no inference can be done about the error value of the neighbors in any given point. This can be seen in the Figure 17 where three types of fitness landscapes (one-dimensional) are plotted.

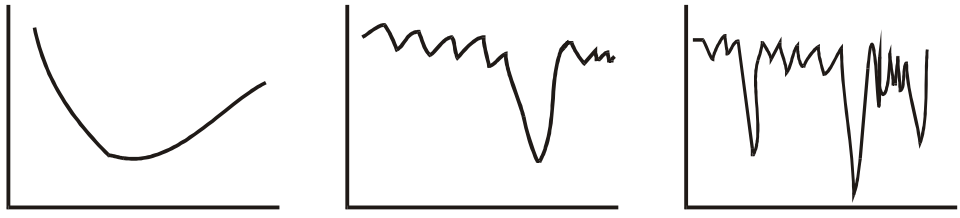


Figure 17 Fitness Landscapes. a) smooth, easy to be searched using an iterative method like gradient descent. b) medium, more difficult for normal iterative methods. c) rough, iterative methods fail completely here. Parallel methods are necessary

3.3.2.3 Enumeration:

Enumeration has to evaluate the fitness value for each point in the parameter space. This is equivalent to running the SST and fitness functions as many times as points in this space. The "best" error value can be found by comparing all of them and selecting the lowest one. This is a very expensive method and usually impossible to compute, because the number of possible combinations even in a small parameter space are too big. A more clever way to explore this space has to be devised. The only advantage of this method is that it guarantees that the lowest error value and its associated parameters will be found.

Iterative methods:
These methods start in a randomly selected point in the parameter space and evaluate the fitness function in some neighboring points. With this information, they select a new set of points, and keep doing this until they advance into a region where the fitness specifications are satisfied.

Iterative methods are numerical methods that rely on evaluation of a formula, instead on solving it symbolically. All these methods start in a "random" point in the parameter space, and evaluate the error value of some neighboring points. With this information they make an "educated guess" of the direction that they should take to start "walking" on this space towards a minimum. The process is repeated iteratively until no step improves the error value.

3.3.2.4 Gradient descent, simplex, downhill:

The common idea of these methods is to take two or more points in the parameter space, evaluate their fitness functions, and with this information suggest a new point to be evaluated. Usually, the new point is evaluated and accepted only if it has a lower error value than the old point.
These methods perform well when applied in cases where the fitness landscape is smooth, "convex", or moderate Figure 17 a). If the fitness landscape presents too many local minima, it is possible that they get stuck in one of these and never find the global minima. If the search allows a "good" solution, but doesn't requires "the best" solution, they perform a good job. If the fitness landscape is rough, these methods don't work at all.

3.3.2.5 Simulated Annealing:

Simulated Annealing can be thought of as an "extension" or improvement over one of the classical methods like gradient descent.
The main idea is to accept sometimes some points with "poor performance" in the update procedure of a given iteration. The acceptance of "bad choices" is regulated by a probabilistic factor that decreases over time, accepting more wrong choices at the beginning and just accepting good ones at the end. The effect of this is that the method can sometimes "climb up" some deep valleys in the fitness landscape. This decision allows better exploration of more regions of the space, and at the end, the solution will converge in a valley, hopefully one that is "very deep" compared with others (and therefore, a better solution).

3.3.2.6 Evolutionary methods:

Many good methods used to search large and complicated spaces borrow concepts from real, natural systems. While these methods lack some of the mathematical rigor that characterizes more classic techniques, their empirical performance has shown that they are apt to solve many

problems that otherwise would go unsolved. The field of evolutionary computation is based on concepts found in the natural systems like reproduction and “survival of the fittest”.

3.3.2.6.1 Genetic algorithms:

Genetic Algorithms (GA) borrow from the idea of Darwinian evolution of the “survival of the fittest” (Koza 1992; Gershenfeld 1999). The parameter space is represented as a fixed length bitstring. A random population of individuals (bitstrings) is created for the first generation. The population is then evaluated and a fitness value assigned to each individual. A new population is created using genetic operations in randomly selected individuals from the actual generation. The process is repeated until an individual that fulfills the specifications is found or a maximum number of generations are reached.
Genetic operations are performed in randomly selected individuals. The selection probability is based on the fitness value of that individual (the better the fitness, the more probability of being selected). The genetic operations create new individuals using: copy (exact copy of the bitstring), mutation (mutation of randomly selected bits) and crossover (copy of different bits from different bitstrings).

GAs have shown outstanding empirical performance in optimization problems where the fitness landscape is very rough Figure 17 c). The reason is because it starts exploring in parallel many different points in the parameter space, and has a higher probability of finding one of the many local minima that are present, but selecting the best option of all of them.

3.4 DESIGN AS A SEARCH IN THE TOPOLOGY SPACE

3.4.1 Automated functional-form suggestion mechanism

The core of our research lies in the automation of the functional-form suggestion mechanism. The goal of this module in the proposed SST design approach is to suggest valid functional-forms that will go through a process of parameter estimation, and then feed back the error information to suggest a new (and hopefully improved) functional-form. The process for suggesting functional-forms has been traditionally done by a human with knowledge in the field of sound synthesis, who selects a functional-form from a set of “classic” SST based on the known properties of the SST and the desired properties of the target sound. Our goal is to automate this suggestion mechanism, and to integrate it in a completely automated approach for SST design.

The functional forms are particular compositions of functional elements. We propose a set of “basic functional elements” that we believe are general enough to express many possible SSTs. They were derived from our analysis of the “classic SST” (see section 2.3).

The SSTs are represented as topology graphs (interconnected functional blocks). The selected functional blocks include:

- Controlled Wavetable oscillator (simple waveforms: sinusoid, triangular, square)
- Addition
- Multiplication
- Simple controlled filter (2 poles 2 zeros)

- Delay by one sample

All of these functional blocks represent linear elements, with the exception of the controlled wavetable oscillator. This means that all of the traditional linear Digital Signal Processing (DSP) systems can be incorporated as a set on our representations. The more complex elements like the wavetable oscillator and the simple filter are ubiquitous in the electronic music synthesis field.

3.4.2 SST space:

All the possible valid combinations of functional elements, connections and internal parameters compose the SST space. Each point in this space could be thought of as a different topology.

3.4.3 Hypothesis:

Given a set of inputs, a target sound and an error metric, it is possible to find the functional-form and internal parameters of a SST capable of synthesizing an output sound "close" to the target sound.

3.4.4 Design as a Search:

Our original goal of designing a SST (functional form and internal parameters) can then be stated as a search problem in the SST space. The next step is to define a search strategy to efficiently and adaptively explore this space and find an acceptable solution to our problem.

3.4.5 Size of the SST space:

The size of the SST space is related to the types of functional elements, the valid ways of interconnecting them, the total number of functional elements allowed, and their internal parameters. The numbers are huge, but a simple example will help to see the kind of spaces that we are dealing with. Assume that the topology representation will use 2 types of functional elements (x,y), with a total of three functional elements that can be interconnected in any order (A,B and then C).

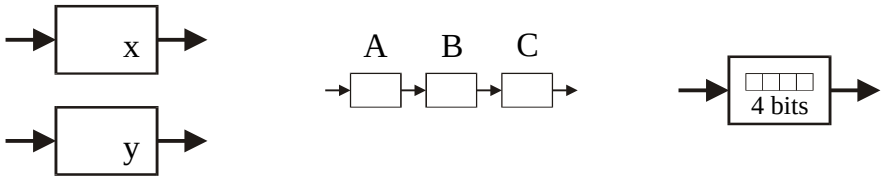


Figure 18 Functional blocks and their combinations. a) two types of blocks: x,y. b) three ordering schemes (connection) ABC, BAC, CBA, etc. c) Internal parameters quantization, in this case is 4 bits per parameter, 1 parameter per block.

The number of possible groups of blocks *G*, regardless of their ordering is given by the total number of types of blocks *T*, and the total number of blocks *N*:

$$G = T^N \qquad (11)$$

In the example, *T*=2, *N*=3, using equation (11), *G*=8.

The possible number of ways of connecting N elements is given by

$$M = N! \quad (12)$$

In our example, the possible number of ways of connecting 3 elements *M*=6.

The total number of possible different combinations of types of functional elements and their connections is given by the multiplication of equations (11) and (12).

$$top = T^N N! \quad (13)$$

In our example, *top*=6x8= 48. This is the total number of possible topologies that can be composed with this simple set of functional elements.
If now, we allow each functional block to have one internal parameter represented by a 4 bit word, that is a total of 12 bits per topology (3 parameters * 4 bits). The total number of possible parameters is given by equation (10) $C = 2^{4\times3} = 4096$

Finally, the total number of "points" in the SST space will be formed by the total of "all" possible topologies times the total of "all" possible parameters. Multiplying equations (10) and (13):

$$size\_SST\_space = 2^{BP} T^N N! \quad (14)$$

For our simple example, there are in total 196.608 different SSTs that span the space. In contrast, for a more realistic setup: 5 types of functional elements, 10 functional elements in total, 2 internal parameters per element, 10 bits per parameter there are approximately 5.7×10^{73} possible SSTs in the space.

3.4.6 Design as a search:

We will approach our problem of suggesting valid functional-forms and internal parameters as a search in the SST space. For the moment, we will try to search for both, the functional form and its internal parameters in the same problem, but in a later section we show a way to split this into two interleaved tasks that could be easily separated for improved performance.

3.4.6.1 Searching the SST space:

As was mentioned in the optimization of internal parameters, there are several methodologies that are usually employed to search in a parameter space to find a suitable solution. The main problem here is that our space is not composed of normal "models", but each point in space represents a different topology or SST (Functional Form + Internal Parameters).
This search space is complex in nature because of its high multidimensionality and non-linearity.

3.4.6.1.1 Exhaustive search:

This search method requires the evaluation of all the points in the search space. It ensures that the best point in the space will be found, but given the large number of possible points in these spaces it is not even remotely practical to try this approach.

3.4.6.1.2 Gradient descend, simplex, downhill:

As we know, the fitness landscape for this space is a very complex. Neighboring points in the space can represent radically different topologies. As a consequence, there is no guarantee that this complex search space has an acceptable local optima that this method can trap itself in.

3.4.6.1.3 Simulated Annealing:

The implementation of this search method requires working in conjunction with one of the methods of the previous section. As seen, these methods are not good for the kind of fitness landscapes with which we are dealing.

3.4.6.1.4 Evolutionary methods:

Evolutionary methods have been shown to perform very well in complex fitness landscapes. (Koza 1992; Gershenfeld 1999). The reason for this is that usually they perform a search in parallel, with simultaneous (and different) candidate solutions, that are usually located in different points in the fitness landscape. New individuals (candidate solutions) are evaluated based on the performance of the old ones, hence the better a solution, the higher probability of trying candidate solutions in that neighborhood.

The reason for this good performance is mainly because the search is in parallel at different points of the fitness landscape, and therefore, has higher probability of finding good global minima than other methods; but the cost is an increase in computation and memory requirements.

In any case, no guarantees about an exhaustive search can be done. The space is so huge that the kinds of solutions that are found are optimal in a local sense, but there is no guarantee that the global minima can be found.

4 SEARCHING THE SST SPACE

4.1 GENETIC PROGRAMMING

Genetic Programming (GP) is an optimization/search method that has been gaining popularity in the last decade. It is an extension of Genetic Algorithms, and both belong to the field of Evolutionary Computation. The idea with GP is to have a population of candidate solutions (in our case, suggested SST) that will be evaluated and a fitness value assigned to each.

The fitness function gives an analytical measure of the performance of the individual and its output. Once all the individuals in the population have computed their fitness value, a new population of candidate solutions is created by probabilistically selecting individuals and performing genetic operations on them. The probability of being selected to be part of a genetic operation is directly related to the fitness of the individual: the better the fitness, the higher the probability.

The genetic operations will create new individuals by: copy (identical copy of an individual), mutation (random alteration of an individual functional form and/or internal parameters), and crossover (characteristics of two individuals are fused together to create a new one).

The process is repeated until a candidate solution that shows a fitness value that fulfills the specifications is found, or a maximum allowed number of generations have been tested.

4.1.1 The genetic programming loop:

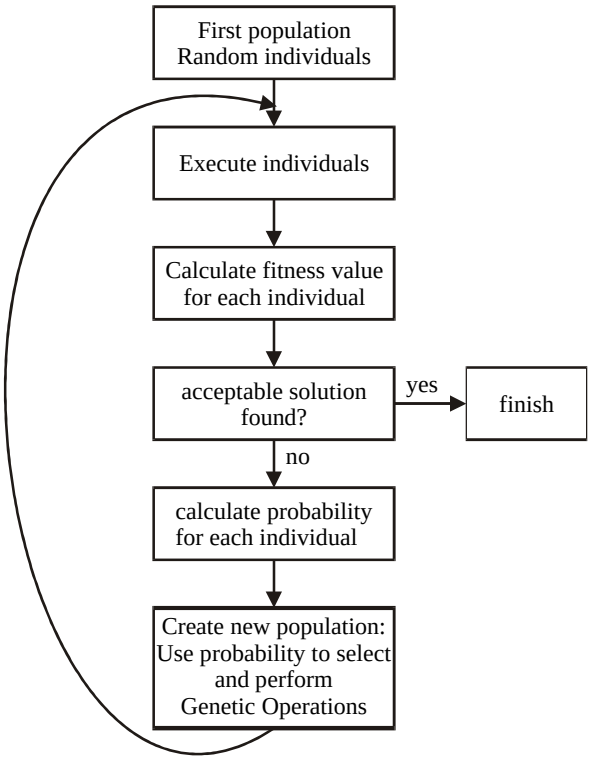


Figure 19 Genetic Programming loop

4.1.2 About the individuals, populations and candidate solutions:

A candidate solution can be seen as a particular point in our search space. For a SST this will represent exactly a particular Functional Form and a set of parameters.
For sake of clarity, we will discuss genetic programming as it was originally explained by Koza (Koza 1992), with individuals that represented simple computer programs (not our more complex SST).
A candidate solution is also called *individual,* and a group of candidate solutions or individuals is called *population*.
Every cycle in the GP loop is known as a *generation*. The influence of natural systems and biology is evident in all the terms used in the field of GP.

The individuals in GA were fixed length bitstrings, that could be easily mapped into a set of parameters to be plugged directly into the model that was being optimized. In GP individuals are computer programs (or executable sequences). They are not easily represented by a bitstring of fixed size. Computer programs are commonly represented by an *expression tree*. (Koza 1999). Expression trees are usually depicted as a rooted point-labeled tree with ordered branches, as shown in Figure 20. The trees are parsed from the upper node (root), and the branches from each node are evaluated from left to right.

$$0.3y + x(0.1 + 0.34) \qquad (15)$$

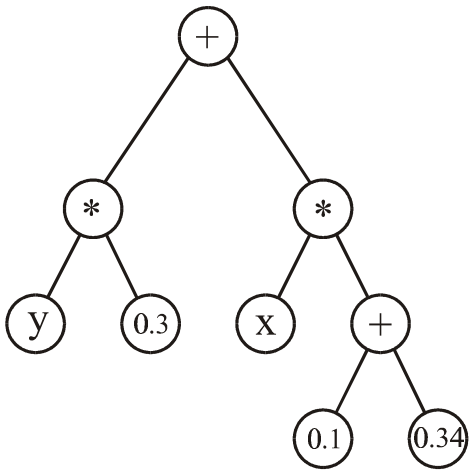


Figure 20 Expression Tree for equation (15)

This notation follows closely the one used in LISP programming (Steele 1984). We recommend the reader unfamiliar with representation and parsing of expression trees to read the references (Koza 1992). In the GP literature it is common to find the names *non-terminals* for the nodes in the tree that represent operations, and *terminals* for the nodes (leaves) in the tree that represent data (variables or constants), and have no further branches.

The rules that specify the types of nodes and connections that are allowed in an expression tree are called *production rules* (Marcotty and Ledgard 1987). A set of production rules has to be fully specified before generating or manipulating any tree, to keep it valid in any imaginable case. The advantage of using trees as representation for the individuals in GP is evident at the time of manipulation. Any manipulation requires the resulting individual to be compliant (valid). In the case of a computer program, valid means that the program can be executed in a digital computer. Non-valid individuals can slow down the process of convergence of a GP run, and add extra load in the housekeeping for testing individuals.

The manipulations that can be done to individuals are usually called *genetic operations*.

- Random creation of expression trees: A single expression tree is created from scratch. Some parameters such as tree depth and tree size can be specified. It is possible to assign some kind of repetition probability to each type of node or connection to shape the content of the tree.
- Copy: A copy of the individual into the next generation is made (see Figure 21).
- Mutation: This usually comes in two flavors: node mutation and branch mutation. Node mutation randomly selects a node in the tree, and changes the value or meaning of that node. If the node is a non-terminal, it is exchanged with an equivalent function. If the node is terminal, the variable or constant represented changes its value. Branch mutation randomly selects a node in the tree, and using the same algorithm as for random tree creation, it replaces everything below the selected node with a random branch (see Figure 22)
- Crossover: Two individuals, usually called *parents*, are used in this operation. A random node is selected in each parent (checking for compatibility of types) Once a pair of compatible type nodes is selected, a new individual (offspring) is created by taking all but the branch rooted in the selected node from parent number 1, and just the branch rooted in the selected node from parent number 2. This creates a new individual that has parts that belonged to two different expression trees, but still is guaranteed to be valid (because of the type check). It is usual to create a second different offspring by reversing the role of the parents (see Figure 23).

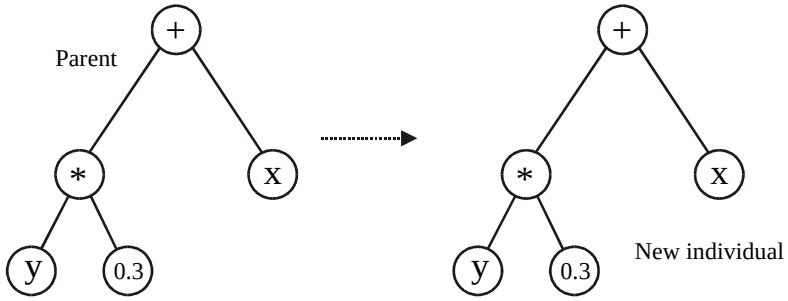


Figure 21 Genetic operations: Copy

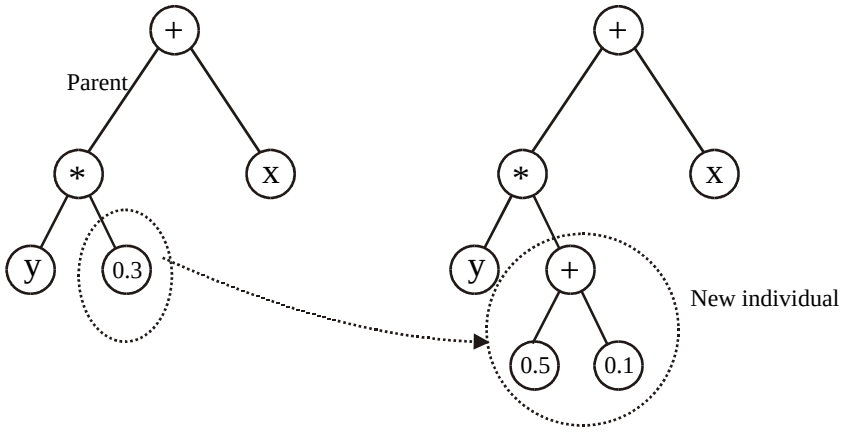


Figure 22 Genetic operations: Mutation

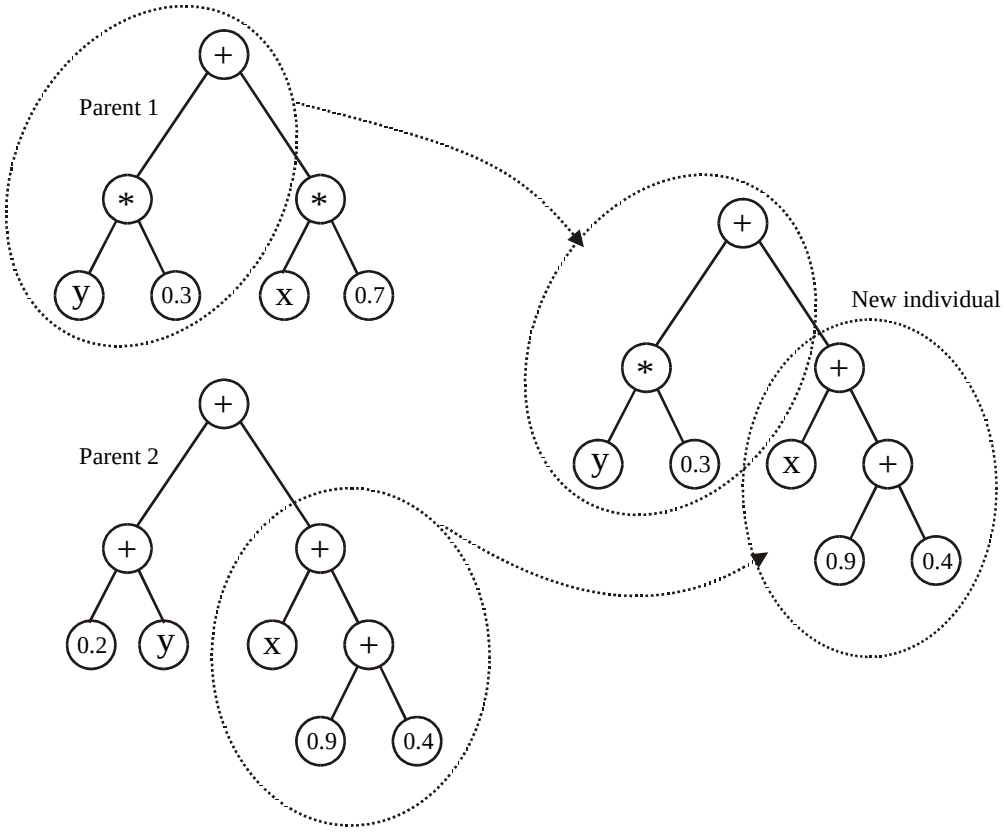


Figure 23 Genetic operations: Crossover

4.1.3 Execution and evaluation

Every individual (expression tree) has to be evaluated using the fitness function. But the fitness function takes information from the output produced by the individual (execution of the program)

and the program itself. Every individual has to be run, and this requires the expression tree to be compiled in an executable form. This compiled program is then run in a digital computer using the specified inputs and recording the produced output.

Fitness evaluation:
Once the output is computed for an individual, this output is then given to the fitness function to grade the performance of the individual. The fitness value will give an analytical measure of the "distance" between the desired target and the produced output. Ideally, a perfect match will give a distance of zero. As discussed before, for our goal of designing SSTs, and taking into account that our target and outputs are sounds that are meant to be heard by human beings, it is possible to use to our advantage the redundancy inherent in the human hearing mechanism, such as frequency masking, producing a more "flexible" target.

4.2 SST SPACE: TOPOLOGIES AND FUNCTIONAL ELEMENTS

4.2.1 Sound Synthesis Engines (SSE)

Our goal in designing a SST is to find a functional form and internal parameters that accomplishes our design specifications. Because of the nature of the problem and the GP approach, it is necessary to "suggest" possible solutions (in the form of SST topologies) and actually implement and run them on a digital computer to produce a sound output. A Sound Synthesis Engine is a platform dedicated to producing synthesized sound. The instructions are usually given in the form of instructions or opcodes that are assembled into programs to be run in a digital computer. There are several Sound Synthesis Engines that are popular in the electronic music field. Csound (Boulanger 2000) is one of the most powerful ones. It gives the user a very complete set of instructions for manipulating digital audio samples, and produces completely synthetic sounds from scratch. But Csound presents the problem that its grammar is too complex for being used in a GP approach. Other sound synthesis engines such as Cmusic (Moore 1990), FX2 (Jones 1998), MAX (Puckette and Zicarelli 1991), PD (Puckette 1996), are also very powerful, but all of them sacrifice simplicity for performance.

Because the SST that we are dealing with will be evolved by an autonomous entity (a computer program) it is better to opt for the simpler evaluation of the SST. For doing this (and to have control over all the unpredictables) we decided to implement a custom Sound Synthesis Engine. (see section 8.2).
The most relevant aspect of our synthesis engine at this point, is how the functional elements are encapsulated in functional blocks.

We need a Synthesis Engine that:
- Gives consistent results
- Is generic enough to implement the "classic" SST and many more
- Has no pre-built SSTs (we are trying to explore new SSTs, we don't want our GP to find answers using pre-fabricated modules).

In short, we need a very robust, very simple Sound Synthesis Engine.

Instead of taking one of the “commercial” SSE in the field, and reducing its capabilities (to fit our necessities), we decided to create our own SSE, based in a model of “connected blocks and messaging system”.

4.2.2 Proposed Topology Graph: Compatible Blocks

Our SSE was standardized to encapsulate the functional elements in “compatible blocks” that have a standard number of inputs and outputs. This has the extra advantage of facilitating the manipulation of the topologies. The types of compatible blocks are:

Block	Inputs	outputs	elements
Source	0	1	SOURCE
Render	1	0	RENDER
Type A	2	1	ADD,MULT, FILT,DELAY
Type B	1	2	KOSCIL,SPLIT

The rules of connection for valid topologies are:

- Outputs are connected to inputs
- Only a single connection is allowed for each output or input. Neither one can have more than one connection.
- All inputs/outputs in a topology have to be connected (no dangling connections).

No other restrictions are enforced.

The functional elements are encapsulated to have one-input/two-outputs or two-inputs/one-output. We are using a set of blocks derived from the summary of functional elements from the “Classic” SST analysis done in a section 2.3. A quick summary of the elements and their function follows.

FILT: (TYPE_A) A second order filter, whose coefficients are set at init time. One of the coefficients can be time varying.

KOSCIL: (TYPE_B) Encapsulates two functional blocks: a constant source (k), and a controlled wavetable oscillator. The type of wavetable, phase and k (constant) value are set at init time. The wavetable uses phase increment to calculate the next index value to read, but the increment can vary over time.

ADD: (TYPE_A) Adds two inputs and produces an output.

MULT: (TYPE_A) Multiplies two inputs and produces an output.

SPLIT: (TYPE_B) Splits an input into two identical outputs.

DELAY: (TYPE_A) Variable delay line. It delays the input by a variable amount of time.

With these proposed blocks, it is possible to implement most of the classic SSTs. An example using a few of them is in Figure 24 and Figure 25.

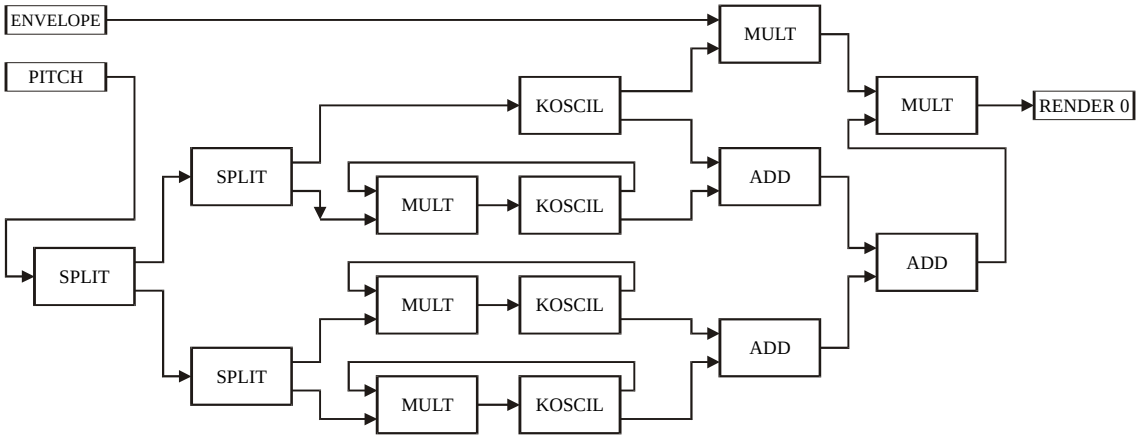


Figure 24 Additive synthesis using compatible blocks TYPE_A, TYPE_B, SOURCE and RENDER

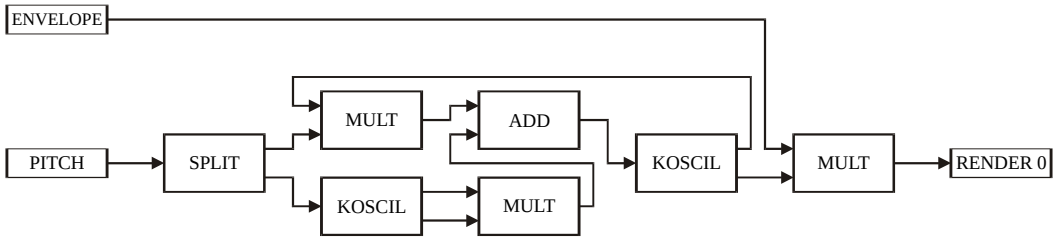


Figure 25 FM synthesis using compatible blocks TYPE_A, TYPE_B, SOURCE and RENDER

4.3 MAPPING EXPRESSION TREES TO TOPOLOGY GRAPHS:

Topology graphs are a good representation for SSTs, especially at the moment of design (conceptual relationship between functional elements) and implementation (many digital computer architectures work in an object-like fashion). But topology graphs are not a suitable representation when using genetic operators (copy, crossover, mutation) that are “blind” to structure and meaning. This can be shown with a simple example. Imagine a simple valid SST with few functional elements all interconnected. If a new functional element wants to be added, some existing connections have to be broken to “insert” the new element. In addition, it is possible that the number of connections (inputs and outputs) is not equal anymore and a “dangling” connection would be left. This would render the topology invalid. If we remember in addition that the goal is to have an automated system doing the manipulations, it is easy to see the magnitude of the problem. Our solution is to find a way of representing topology graphs by expression trees. Given an established mapping between the two, it is thus possible to use a GP for searching the SST space.

An ingenious idea borrowed from developmental biology suggests a way of mapping a topology graph representation into an expression tree representation. The idea is to encode in the expression tree the instructions for the "development" of an embryonic topology. The process begins with a very simple embryo, and following the instructions, it "grows" the fully developed topology. It can even include the development for the internal parameters associated with the functional blocks.

4.3.1 Background:

4.3.1.1 Automated Synthesis of Analog Electrical Circuits:

Koza et al. (Koza, Bennett et al. 1997; Koza 1999) proposed a representation for developing topologies of analog circuits, that later were used for a genetic program to evolve the functional form (topology) and the internal parameters (sizing of elements) of the analog circuits. The expression trees are called "circuit-constructing program trees" and have four kinds of functions on them: 1) connection-modifying functions (CMF's), that alter the topology of the circuit; 2) component-creating functions (CCF's), that insert components into the circuit; 3) arithmetic-performing functions that appear in subtrees as arguments for the CCF's and specify the numerical value (size) of a component; and 4) Automatically-defined functions (ADF's) that have the role of sub-programs within the tree.
A constrained syntactic structure is also proposed to regulate the creation and usage of the functions in the circuit-constructing program tree.
The embryo circuit is composed of one or more "modifiable wires". These wires can change their points of connection, or become new electrical components.When the expression tree was executed, the modifiable wires changed into a fully developed circuit. The different types of embryos are selected to match the number of inputs and outputs of the problem at hand.

The process for evaluating the individuals comprised some steps:
- Use circuit-constructing tree in the embryonic circuit (evolve a circuit)
- Translate circuit in a NETLIST (list of nodes, components and connections)
- Purge NETLIST of dangling components, isolated subcircuits, grounding isolated nodes, etc (cleaning elements that would render impossible to simulate circuit)
- Run the SPICE simulating software on the simplified NETLIST.
- Calculate fitness using output of SPICE program.

The electrical behavior of the system was simulated using the SPICE electrical circuit simulating software (Quarles, Newton et al. 1994). The fitness measure used is the absolute sum of the weighted deviation between the target and produced output, with different grades of penalization for small and big deviations from the target output.

4.3.1.2 Cellular Encoding of Genetic Neural Networks:

Gruau (Gruau 1992; Gruau 1993) faced the problem of encoding the topology for Neural Networks. In his approach he also choose to represent the development of an embryonic Neural Network topology (NN topology) by development instructions in an expression tree. In this research, seven theoretic properties to grade the efficiency of encoding schemes for neural networks topologies are proposed. An encoding scheme that satisfies all of the requirements was

studied. The most important point reinforced by Gruau's research is the usefulness of representing a topology graph (or topology-like problem) with an expression tree that encodes the development of an embryonic topology. It is possible to develop mappings from expression trees to topologies that satisfy a set of properties that allow for a great deal of manipulation and flexibility, especially if the expression trees will be suggested and manipulated by an autonomous entity (a computer program).

4.3.2 Proposed Mapping

The execution of the instructions on the expression tree will result in a fully developed topology graph. The initial topology graph is called *embryo*, and in our case it is a simple topology with four blocks, as seen in Figure 26.

The embryo has a single modifiable object TYPE_A with no functional element assigned yet, and connected to two sources and one renderer. The sources and the renderer will remain the same during the whole development process, but the modifiable object will change and new blocks and connections will be created. This configuration of the embryo could be different (to suit the design specifications, i.e. the number of time varying inputs), but has been chosen for explicatory purposes here. Figure 26 shows a simple expression tree, embryo and first steps of development of a topology. The first node of the expression tree is the *START* node, and this is ignored during the development process. The second node is "pointing" to the modifiable object number 1. When executed, this node will change the topology graph, more specifically the object that is pointed to in some way. In this case, the node has the instruction *MULT*, so the type *MULT* is assigned to the particular block in the topology graph.
The next node has the instruction *SERIES1*. The effect of this instruction is to add some new blocks and connections to our topology graph, and to create more "pointers" to different nodes in different new branches of the expression tree. After executing this Topology Modifying Function, several new blocks and connections are introduced into the topology graph. Each one of the new objects is modifiable, and has an associated node pointing to it. The rest of the nodes are executed, and this add, modify, or change blocks and their connections into the topology graph.

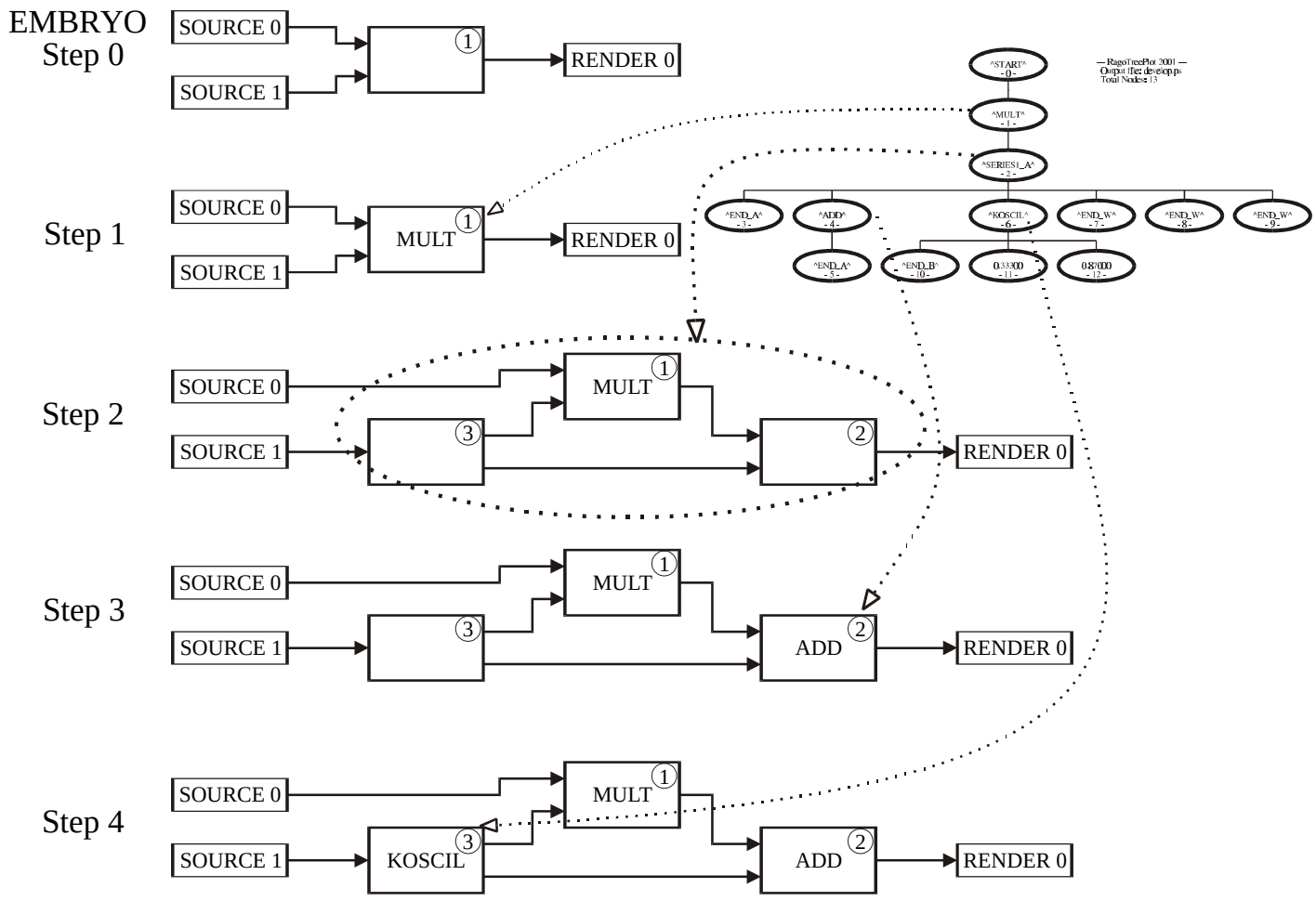


Figure 26 Embryo and example of development

4.3.2.1 Development process:

Every node in the expression tree has a function associated with it. At the beginning, a single node is pointing to the modifiable block of the topology graph. After this, all the remaining nodes on the expression tree are executed, one by one, in breadth-first order, to produce intermediate topologies. When the last node is executed, the final topology is the result of the development of the embryo with the particular expression tree.

The topology-creating functions are defined in a way that preserve the validity of the topology after their execution. This ensures that any valid expression tree will produce a valid topology graph.

4.3.2.2 A simple repertoire of topology developing functions:

For more information, please refer to section 8.1 that explains the creation rules, precedence procedure, and more details about this proposed set of functions.

- Each node in the expression tree has a related function, that somehow affects the topology graph.
- Each node is “pointing” to one modifiable object or wire.
- Some functions assign types to the related modifiable object (i.e. MULT, ADD), while some others change the topology and add new objects and connections (i.e. SERIES1, PARALLEL1), or just control the execution of the actual expression tree.

TYPE functions: assign a type to the object they point to. Also, if the related functional block has internal parameters, they are found in the next level after the type node.

TYPE_A = ADD,MULT,FILTER;
TYPE_B=SPLIT,KOSCIL

Topology Modifying Functions (TMF): break existent connections, add new objects, and re-connect them to accomplish a valid topology. Every node of the next level is pointing to the newly created objects or the wires.

TMF_A= SERIES1_A, SERIES2_A, PARALLEL1_A, PARALLEL2_A
TMF_B= SERIES1_B, SERIES2_B, PARALLEL1_B, PARALLEL2_B
TMF_W= RECONNECT1, RECONNECT2, RECONNECT3

Development Control Functions (DCF): don’t alter the topology in any way, but delay or stop the execution of one branch in the expression tree.

DCF_A = NOP_A, END_A
DCF_B = NOP_B, END_B
DCF_W = NOP_W, END_W

Construction Continuing Subtrees (CCS): As the DCF don’t alter the topology, but are in charge of selecting one of the previous sets of functions to execute.

CCS_A=TYPE_A, TMF_A, DCF_A
CCS_B=TYPE_B, TMF_B, DCF_B
CCS_W=TMF_W, DCF_W

4.4 INTRODUCING HYBRID-OPTIMIZATION: LAMARCKIAN EVOLUTION

A biology theory called Lamarckism, proposes that individuals evolve by the inheritance of traits that were acquired or modified through the use or disuse of body parts. A parallel to Lamarckism can be introduced into Genetic Programming, by allowing the individuals (programs) to locally optimize their internal parameters after a functional form has been suggested and then use the optimized values as inherited traits.

This can also be seen as the independent task of sub-optimization or parameter estimation using any of the previously discussed techniques. The approach uses GP to suggest a functional form, and then it uses any of the discussed techniques to optimize the internal parameters of the suggested functional form. The optimized set of internal parameters is put back in the original individual used by the GP. These parameters will be inherited by future generations.

This can be seen in Figure 27, where an extra step for optimization of only the internal parameters for each individual is added. The optimization technique used can be any of the previously discussed.

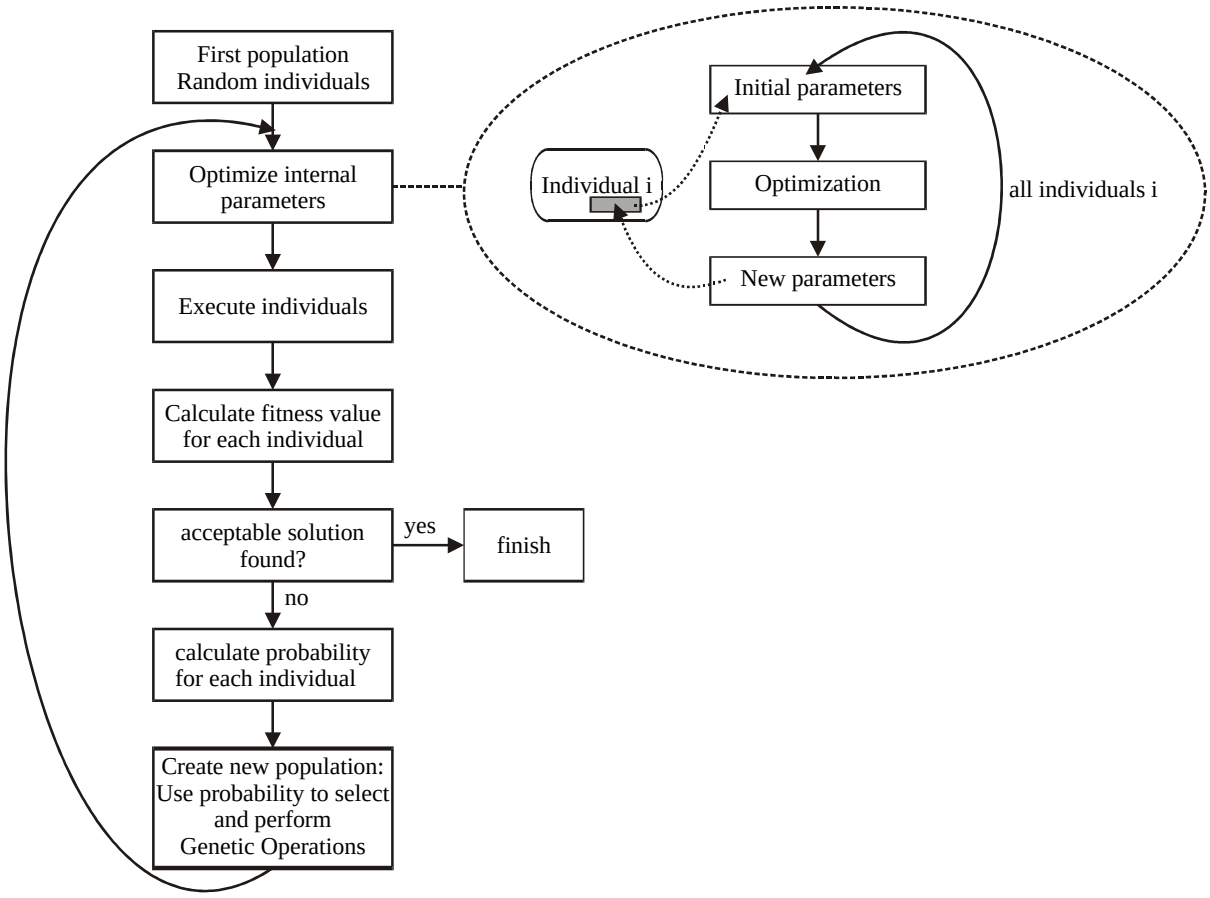


Figure 27 Hybrid optimization (Lamarckism): internal parameters have an extra optimization routine after a functional form has been selected.

It is important to note that the fitness function or error metric used during the optimization of the parameters should be the same used to guide the GP loop. If a different metric is employed, it is possible to stray too far from previous "good" solutions. The use of a sub-optimization stage has been demonstrated to give good empirical results, because in this way the burden of estimating an optimal set of parameters doesn't fall completely to the GP. Without it, GP may reject "good" suggested functional forms because they happen to have very bad parameters associated with them. If an optimization procedure is used, the probabilities of selecting at least a better set of parameters increases.

4.5 FITNESS FUNCTIONS

In any kind of optimization or search method, it is fundamental to have a way to measure the performance of the candidate solution. This performance metric is usually called a *fitness function* or *error metric*. Fitness functions (FF) give some numerical grade to the difference between the outputs of the system compared to a desired target. The features that are measured in a fitness function vary from application to application. In our case, for sound synthesis techniques and sound sample sequences (waveforms) as targets, it is usual to define fitness functions that measure the distance between two sounds, or how "similar" they are.

But that doesn't mean that the only fitness functions used have to measure how similar two sounds are. It is possible to fashion a FF that looks for certain features in a sound, For example, it is possible to design a FF that measures the spectral centroid, and penalizes the deviations from it. Even more, the FF is not limited to using the output and target information only, but can use any desired metric on the candidate solutions. It is possible to include implementation "desires" on the FF, i.e., "size" of the algorithm, complexity, memory requirements, etc.

Because of the characteristics of "distance metrics" it is accustomed to give a value of zero to a "perfect match", and a positive number to worse matches. The bigger the FF value, the farther away from a perfect match.

For our research towards the goal of assisting the design of sound synthesis techniques, it makes sense to define a FF that measures the distance between sounds. This then could be used to search for SST capable of producing sounds "similar" to the target.

4.5.1 Analytical vs Perceptual:

An analytical FF will use an objective measurement of the selected feature. Perceptual FFs will incorporate a subjective model of the selected feature. For example, an analytical measure would be the *fundamental frequency* of a signal, and the perceptual measure would be the *perceived pitch* of the signal. While the former has a very simple analytical form to be computed (and will always give the same type of results), the latter requires a model of how humans perceive pitch. This subjectivity can cause discrepancies between different models employed.

In any case, the FF has to return a numeric value associated with the "performance" of the sound related to the target. Subjective grades like: "good", "better", "bright", "dark" are not allowed, and have to be converted to a numerical value.

4.5.2 Analytical FF: Least Squared Error (LSE)

The LSE of a given sound and target is usually computed in one of two domains: time or frequency. In the time domain, the sum of the squares of the difference of the signals is calculated using equation (23):

$$F_{LSE\_TIME} = \sum_{n=1}^{N} \left(o(n) - t(n)\right)^2 \qquad (16)$$

with:

$o(n)$ = output waveform from SST under test.

$t(n)$ = target waveform

N = number of samples to measure

This measures the sum of the square of the error between both waveforms. The closer to zero, the less the error between waveforms.

A somewhat more complex measure, but at the same time more useful can be acquired using a time-frequency distribution of the waveforms. One of the most used time-frequency distributions is the *spectrogram*. To compute the spectrogram, the desired time sequence (waveform) is windowed in short segments, and a DFT is computed ever each one of them. It is common to have some overlap between segments. The spectrogram is a 2 dimensional representation, were one axis spans the frequency, and the other spans the frames (time). The spectrogram can be a "magnitude" spectrogram (throwing away the phase information on the DFTs), or a "complex" spectrogram that also takes into account the phase.
The LSE is measured after computing the spectrograms for both signals (with exactly the same parameters), and comparing frame by frame, and bin by bin each of them. The squared error is computed for all the components and added together to give the total error. The analytical fitness function using magnitude spectrograms is then calculated using equation (17):

$$F_{LSE\_FREQ\_MAG} = \frac{1}{F}\sum_{j=1}^{F}\sum_{i=1}^{B}\left[\left(\left|O(i,j)\right| - \left|T(i,j)\right|\right)^2 W_M(i,j)\right] \quad (17)$$

where:
F = Number of frames in the spectrogram
B = Number of frequency bins in each frame
$\left|O(i,j)\right|$ = Magnitude of the complex spectrogram of *o(n)*
$\left|T(i,j)\right|$ = Magnitude of the complex spectrogram of *t(n)*
$W_M(i,j)$ = Weight matrix for each component (i,j). It's value ranges from [0.1]

This fitness function has been used successfully by Horner et a. (Horner, Beauchamp et al. 1993) to optimize the parameters of a FM sound synthesizer.

A logical enhancement is to include phase information along with the magnitude. Another FF used during this research computed the fitness for the magnitude and phase spectrograms, and combined them in a linear fashion to achieve the final fitness. The fitness function for the phase spectrogram is calculated using equation (18):

$$F_{LSE\_FREQ\_PHASE} = \frac{1}{F}\sum_{j=1}^{F}\sum_{i=1}^{B}\left[\left(angle(O(i,j)) - angle(T(i,j))\right)^2 W_P(i,j)\right] \quad (18)$$

where:
F = Number of frames in the spectrogram
B = Number of frequency bins in each frame
angle(O(i,j)) = Phase component of the complex spectrogram of *o(n)*
angle(T(i,j) = phase component of the complex spectrogram of *t(n)*
$W_P(i,j)$ = Weight matrix for each component (i,j). It's value ranges from [0.1]

And combined linearly gives:

$$F_{LSE\_FREQ} = F_{LSE\_FREQ\_MAG} + F_{LESE\_FREQ\_PHASE} \quad (19)$$

This kind of FF ensures that the components of the output waveform are aligned in phase with the components of the target.

4.5.3 Perceptual FF: Simultaneous Frequency Masking (SFM)

SFM is used to analyze a frame of sound (usually 5-15 ms long) and determine the "threshold of masking" (TM) (Pohlmann 1995; Roederer 1995; Cook 1999; Garcia 1999). This is a value that determines which frequency components are "heard" by an average human listener (above the TM), and which components are "meaningless" for the same (below the TM). Encoders usually eliminate the components below the TM, and achieve a data reduction of up to 15 times the original data rate. In our case, our goal is to create a sound synthesizer that will "produce" a sound in which:

- The components in the output sound that corresponds to the components above the TM in the target are as similar as possible to the target.
- The components in the output sound that correspond to the components below the TM in the target are also below the TM in the target, but don't have to be identical to the components in the target. (this is a subtle but important point).

In summary, our perceptual fitness metric will:

- Calculate the spectrogram of the target and output sounds
- Calculate the TM of the target sound
- Find the components above the TM of the target (frame, freq. bin)
- Calculate the full error between the components from previous step.
- Calculate a partial error between the components not used in the previous step. Partial error is the difference between the output sound component and the TM in that component (not the target sound).
- If an output component is below the TM, the error is zero

Figure 28 shows the graphic interpretation of these rules, and their computation is outlined in equations (20) to (22).

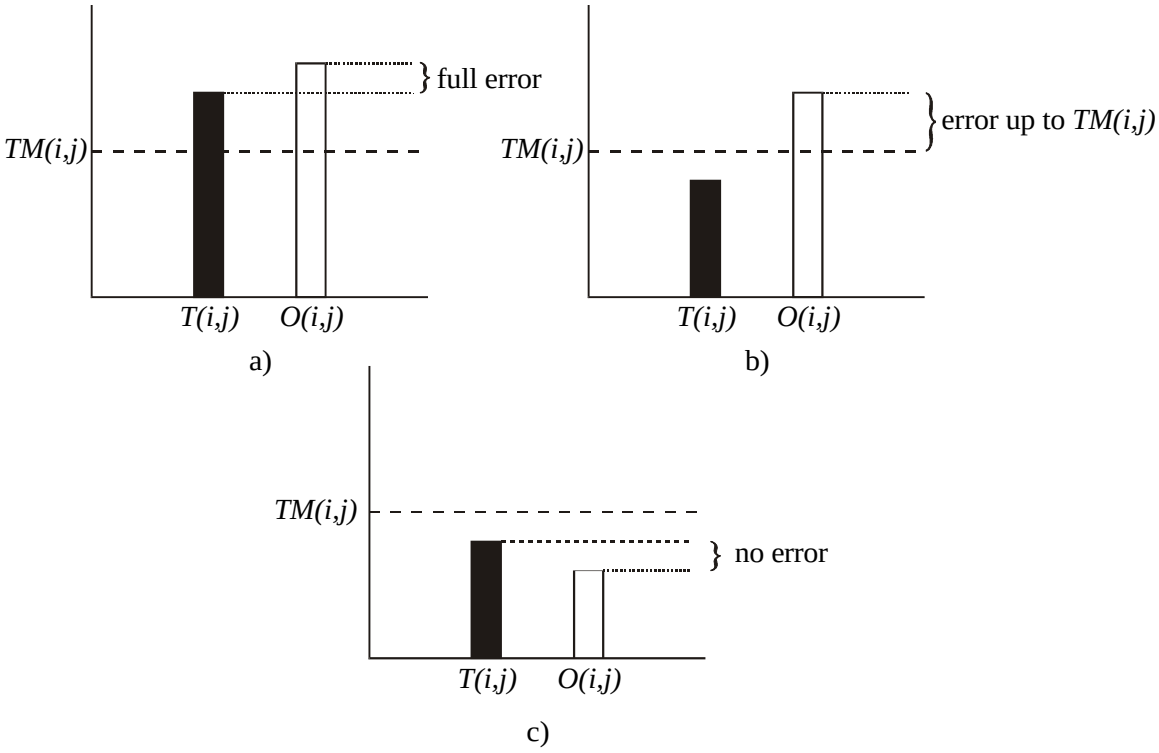


Figure 28 Computation of the perceptual fitness value: three cases a) *T(i,j)>TM(i,j)*, full error calculated; b) *T(i,j)<TM(i,j)* and *O(i,j) > TM(i,j)*, error up to *TM(i,j)*; c) *T(i,j) and O(i,j) <TM(i,j)*, error is zero.

T(i,j) = Complex spectrogram of t(n)
O(i,j) = Complex spectrogram of o(n)

TM(i,j) = Threshold of Masking for target sound, calculated from T(i,j).

$$MASK(i,j) = \begin{cases} 1 & T(i,j) \geq TM(i,j) \\ 0 & otherwise \end{cases} \quad (20)$$

$$MASK2(i,j) = \begin{cases} 1 & O(i,j) \geq TM(i,j) \\ 0 & otherwise \end{cases} \quad (21)$$

$$F_{LSE\_SFM} = \frac{1}{F}\sum_{j=1}^{F}\sum_{i=1}^{B} \left\{\left[\left(|O(i,j)| - |T(i,j)|\right)^2 MASK(i,j)\right] + \left[\left(|O(i,j)| - |TM(i,j)|\right)^2 MASK2(i,j)\right]\right\} \quad (22)$$

This process calculates the spectrogram of target and output sounds T(i,j) and O(i,j) respectively. Then, it calculates the threshold of masking for the target TM(i,j). The next step is to calculate the components above the TM of the target MASK, and the components above the TM of the output MASK2. Note that we are using the same TM for both calculations, the TM obtained from the target.

The fitness is then calculated as the sum of the squares of the differences of the components above TM of the target (MASK), plus the sum of the squares of the differences of the components that are both below TM for the target (1-MASK) and above TM for the output (MASK2). For the components that don't fit in any category (below TM for target and output (1-MASK, and 1-MASK2), the error contribution is zero. A similar FF is proposed by (Wun and Horner 2001).

5 DEVELOPED SYSTEM

A system implementing the proposed approach called AGeSS (Automatic Generation of Sound Synthesizers) has been developed and tested.

One goal of the AGeSS system is to be a platform for exploring the potential of this approach and allow some kind of isolation from some of the most repetitive and computationally hard problems, while allowing easy access to other modules. The system is implemented as a set of binaries (compiled for ANSI C++) and Matlab scripts.

5.1 USER:

The user is required to supply the parameters for the GP run, as well as the examples for the system.

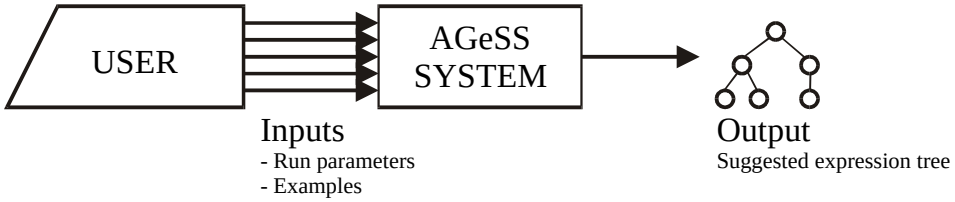


Figure 29 AGeSS system: user input (parameters, examples of control signals and Target). Output (suggested expression tree)

5.1.1 Inputs:

- *Number of individuals:* This is the size of the population. The bigger, the more SST space that could be explored in parallel, but more computation time and memory are required.
- *Random Seed:* A number selected by the user to seed the random number generator used for the runs.
- *Input/output folders:* A folder is created for every generation, and all the individuals of that generation are saved to the folder. This permits later recovery of information and individuals. Statistics of all the run (best individuals, best fitness, generation number) are stored for later retrieval. Initialization information, target files, rules files are stored in the input folder.
- *Name of the experiment:* To identify all the files created for the particular experiment.
- *Target/input file(s):* It is possible to use one or more sets of control inputs/target files.
- *Tree max levels and max nodes:* Maximum allowed number of levels and nodes in the individuals (trees). Once the maximum is reached, the termination rules are employed to "gracefully" terminate the tree (to keep validity). This doesn't guarantee that the total number of nodes or levels will be the specified, but will control the growth of the tree.
- *Probability for Genetic Operations:* Genetic Operations are selected randomly, and they can have different probabilities.
- *Sub-optimization parameters:* Once a functional form is suggested, the suboptimization can be done using any of the techniques described in the parameter estimation section. This can be selected here, as well as the specific parameters for each type of

optimization. Not all the individuals in a population have to be optimized, and this can be controlled using a probability of optimization.

- *Rule files:* The creation rules that the trees follow are specified in a set of rules, and end_rules files. The generation rules file (see section 8.1) can be edited to select the probability that each terminal and non-terminal will have. If any of them needs to be left out of the run, it is assigned a probability of zero. The termination rules file is another file that decides how to "terminate" a branch once the maximum number of levels or nodes is reached (to terminate gracefully). These rules files don't have to change from run to run, and are actually passed as a default parameter to the system.

5.1.2 Output:

- *output tree:* The output of the system is in the form of the "best" individual found during all the generations. In the actual implementation the best individual of the last generation is also the best individual of all the run. But it is important to note that because of the differences found in the fitness function, it is possible to have "desirable" individuals along the run, without them achieving the best scores in their fitness value.

5.2 SOFTWARE:

The AGeSS system was implemented as a set of ansi C++ binaries, and Matlab scripts. Many of the stages were divided into smaller but significant procedures that were executed upon need. The structure of the system can be seen in Figure 30 C++ binaries have the extension *.exe, Matlab scripts have the extension *.m.

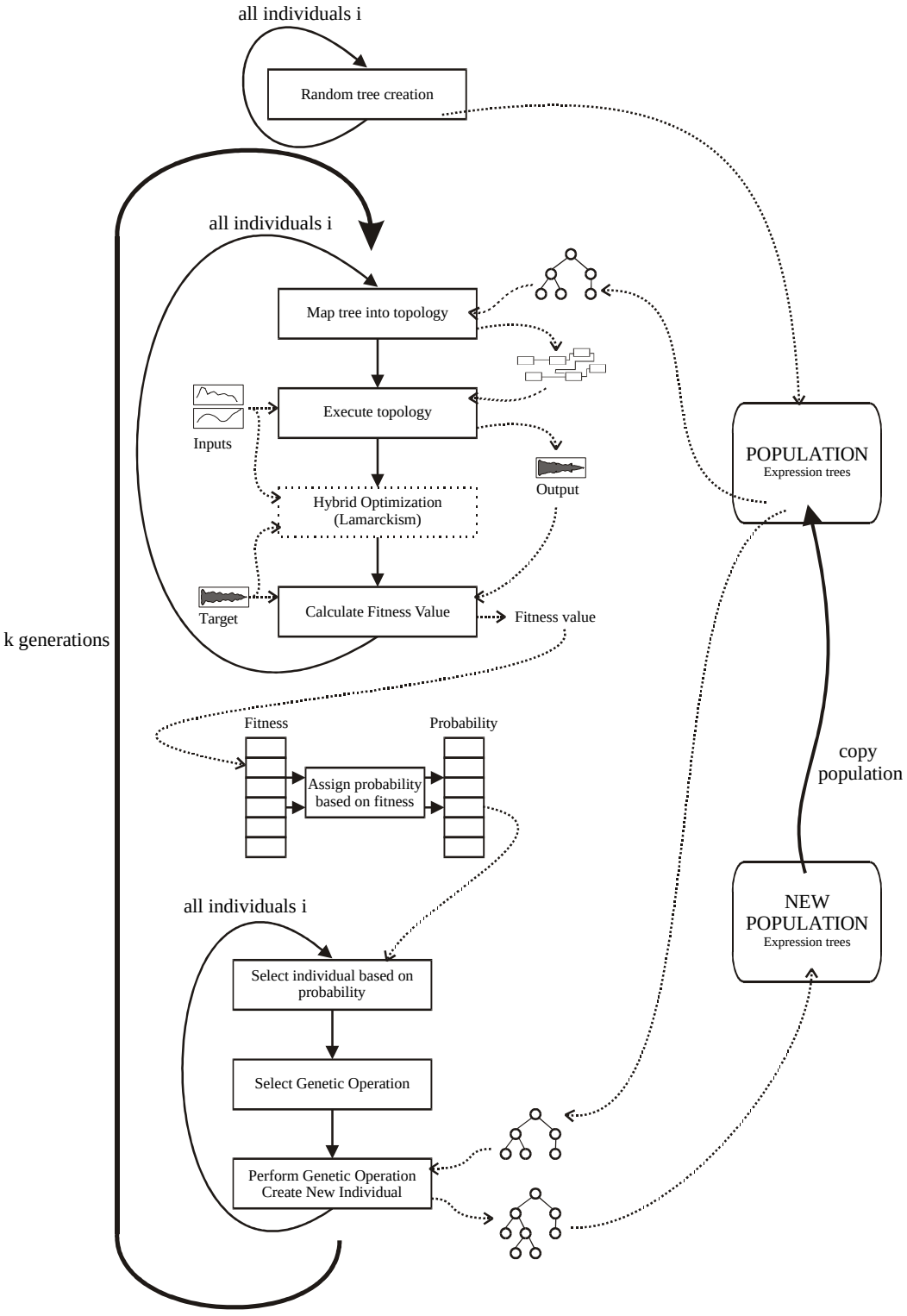


Figure 30 AGeSS system: Internal modules and their relationship

5.2.1 Tree manipulation:

Tree files have the extension *.tre.
tree_rules.exe : Reading and management of rules using BNF notation.
tree_random.exe : Creation of random expression trees, with size, node and rules control.
tree_copy.exe : Copy of actual tree to next population.
tree_cross.exe : Crossover of two trees to form a new tree in next population.
tree_mutate.exe : Mutation of one tree to form a new tree in next population

5.2.2 Topology:

Topology files have the extension *.td (topology description).
grow_tree.exe : Development of a given tree into a topology. The input is a tree file (*.tre) and the output is a topology file (*.td).
td_run.exe : Execution of a topology file. It is necessary to specify the input files (sources) and the output file. The output is a file with sound samples, or a standard *.wav wavefile.

5.2.3 Fitness function:

fitness.m : It takes an output waveform (from td_run.exe) and a target waveform, and performs a fitness calculation. This calculation could include extra information taken from the original *.tre file like size, number of elements, etc.
Our approach included three types of fitness functions: regular LSE on the magnitude spectrogram, LSE of magnitude and phase spectrograms, and the perceptual simultaneous frequency masking fitness function.

5.2.4 Genetic Programming Loop:

AGeSS.m : This is the main script, in charge of calling all the other programs and scripts. The main GP loop is here, as well as the initialization calls and result outputs. The initial user parameters are specified here.

lamarck.m : This is a script that takes a particular *.tre file, a set of inputs and a target, and uses a technique to do parameter estimation of the internal parameters. The selected technique in this implementation is a Genetic Algorithm, but any other parameter estimation technique could be used.

5.2.5 Edition and visualization:

A set of tools to edit and visualize the expression trees and topologies was developed or adapted for research purposes.

treeedit.exe : Command line based editor to add, delete, and modify nodes in expression trees. Several options are included such as: verification of validity on the tree, printing (to PostScript file), Development of tree, Execution of topology.

treeprint.exe : Utility to produce a PostScript drawing of the tree. It uses a custom tree drawing algorithm tailored to minimize space for the printing of nodes in multiple levels.

dot.exe : Software from AT&T (Koutsofios and North 1996) designed for “clean printing” of graphs. In our case it was adapted to print topology graphs, focusing on the connections between objects. The output is a PostScript graphic file.

grow_tree_dot.exe : Utility to develop a tree into a grown topology, and produce a dot script file that can be interpreted by dot.exe to produce a PostScript graphic file of the topology.

The waveforms produced by the topologies are recorded as standard wav files (Microsoft exchange format wav). This allows using any standard audio editor software package to open, visualize, play and edit the produced waveforms.

6 EXPERIMENTATION AND RESULTS

The developed AGeSS system was used to perform a series of experiments to explore the potential of the suggested approach.
Each experiment is divided in the following stages:

- Selection of a target SST.
- Creation of inputs/target sound using the selected SST
- Definition of fitness function
- Selection of parameters for AGeSS system
- Use parameters, inputs, target to feed the AGeSS
- Analysis of selected "best of generation" individuals
- Analysis of "final" SST suggested by system.

All of the experiments were run using a Pentium III dual processor 733 MHz with 256 MB in ram. For experiment number 1, the run time was 22 hours (using about 50% of processing power) for 220 generations and a population of 50 individuals. For experiment number 2, the run time was 100 hours (using about 50% of the processing power) for 1600 generations and a population of 50 individuals. For experiment number 3, the run time was 36 hours (using about 100% processing power) for 200 generations and a population of 40 individuals.

6.1 EXPERIMENT 1. FM SYNTHESIS (CHOW727)

6.1.1 Selection of a target SST:

A simple FM synthesis formula was chosen for this experiment (Roads 1994; Boulanger 2000), as shown in equation (23). This SST has been explored in depth by many researchers and musicians. The value of the internal parameters was taken from the original values suggested by Chowning for simulating a woodwind sound (Chowning 1973).

$$s(t) = A(t)\sin\left(2p\,\frac{C_f}{FS} + 2pIM_f\sin\left(2p\,\frac{M_f}{FS}\right)\right) \qquad (23)$$

With: C_f = Carrier frequency = 880 , 988 = *f(t)*

M_f = Modulator frequency = 880/3, 988/3 = *f(t)/3*

I = index of modulation = 2
FS = sampling frequency = 8000
A(t) = time varying envelope

6.1.2 Inputs/target:

This SST uses two time varying inputs: *A(t)* and *f(t)* and 3 internal parameters I, D, M. For the generation of the Target sound, two time varying signals were generated (using Matlab) to simulate the brass sound of two distinct notes (A880, B988) of 0.3 seconds each. These can be seen in Figure 31.

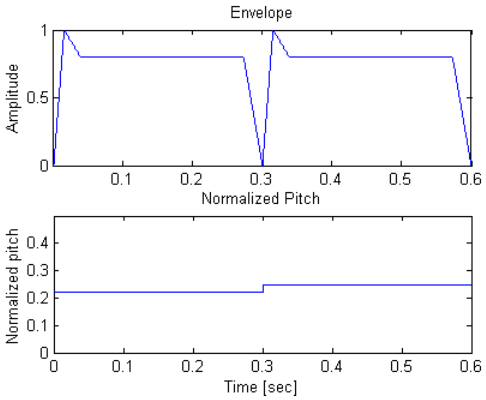


Figure 31 Input signals for experiments 1 and 2. (top) Envelope for two notes. (bottom) Normalized pitch for two notes (A880, B988).

6.1.3 Fitness function:

The selected fitness function uses the SFM LSE fitness function explained in section 4.5.3. It calculates the spectrogram of the target sound and uses this to calculate the threshold of masking of the target. This information, along with the spectrogram of the output sound is used to calculate a distance metric.

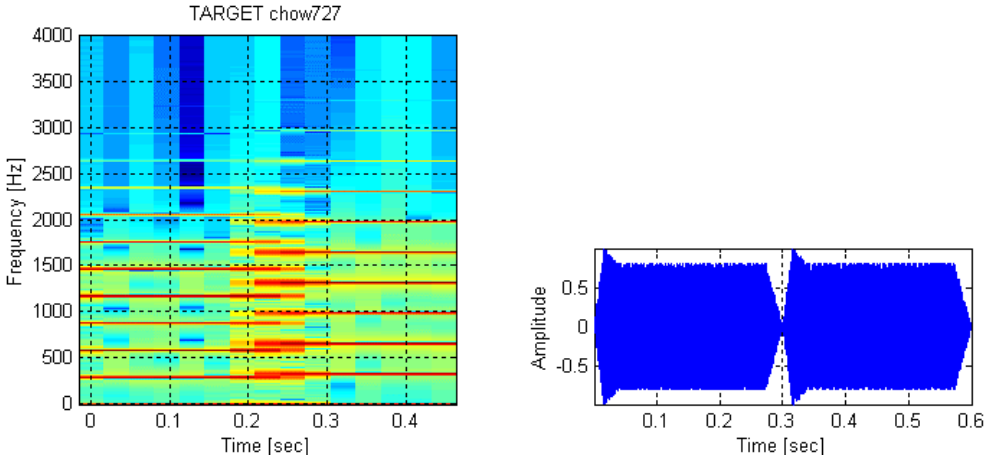


Figure 32 Spectrogram and waveform of TARGET signal for experiments 1 and 2, formed by two notes (A880, B988).

6.1.4 AGeSS parameters:

Experiment name: chow727
Allowed functional blocks: SPLIT, KOSCIL, ADD, MULT
Number of individuals: 50
Tree max levels: 10
Tree max nodes: 60
Probabilities for G.O: copy 10%, mutation 60%, crossover 25%, new random individual 5%;

Sub-optimization parameters: G.A. with 7 individuals max, 5 generations max.

6.1.5 Analysis of best of generation individuals:

Figure 33 shows the fitness of the "Best of generation" individuals across 220 generations. As expected, the fitness value of earlier individuals is higher than subsequent ones. It can be seen that the exploration stalls for several generations at a steady value, until a better functional form is suggested, then rapidly the fitness value decreases. It is said that the system "converged" to a topology when the fitness value stalls for a long time (preferably at a low fitness value).

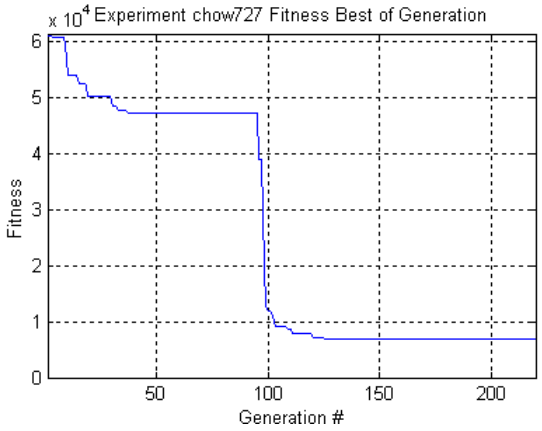


Figure 33 Fitness value for the "best of generation" individuals for experiment 1.

Generation 1 has a good match for a single frequency, that follows the original change in pitch.

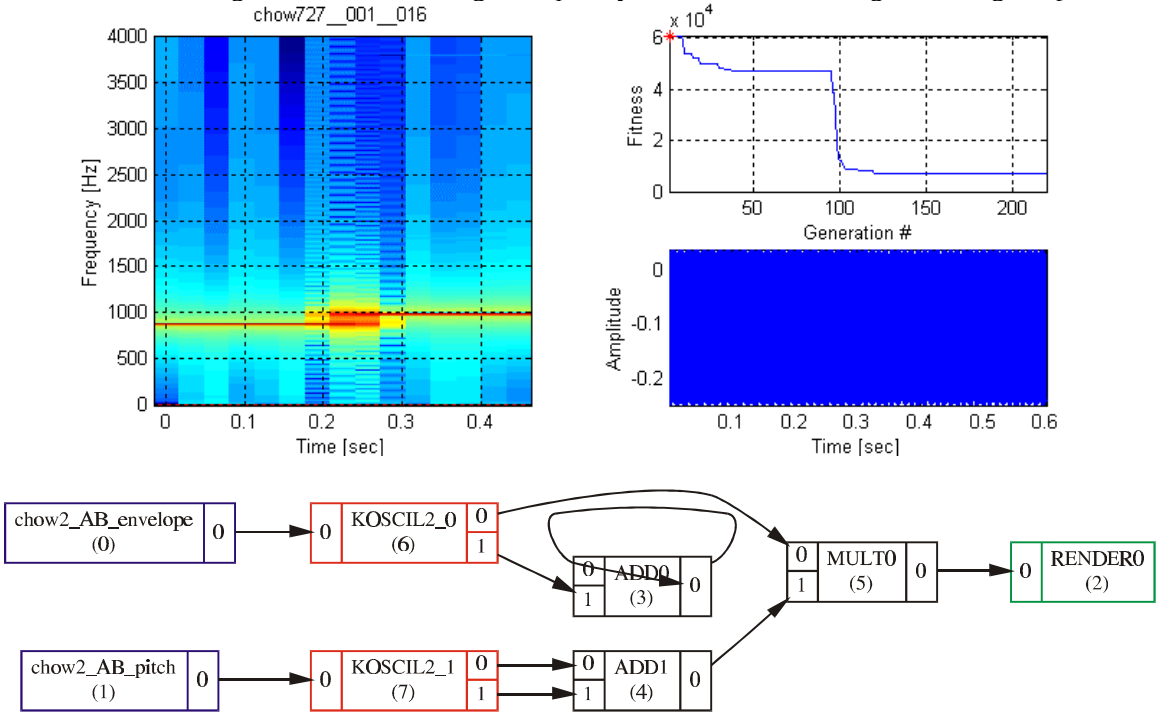


Figure 34 Spectrogram, waveform and topology for best individual of Generation 1, experiment 1.

Generation 10 presents a more complex topology, that produces a set of harmonics of different frequencies, and some of them are very close to the desired frequencies in the target. Note also the use of the pitch input to actually control the pitch change. This relationship was "found" by the algorithm, and it was never explicitly defined. The same applies to the envelope, is just used, but never enforced in a particular way.

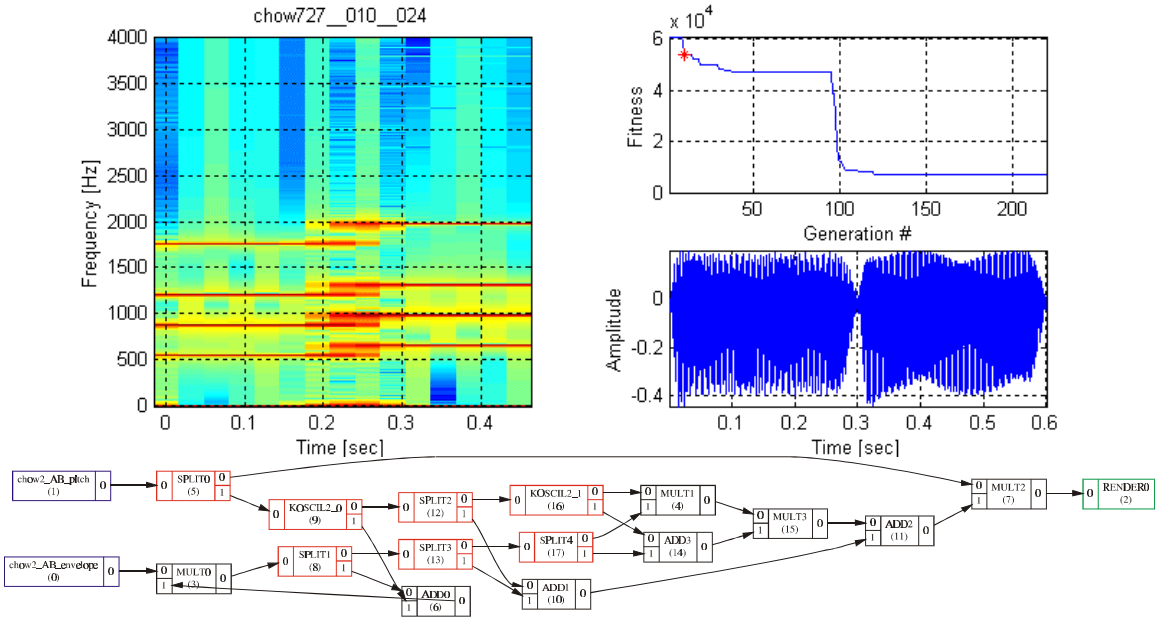
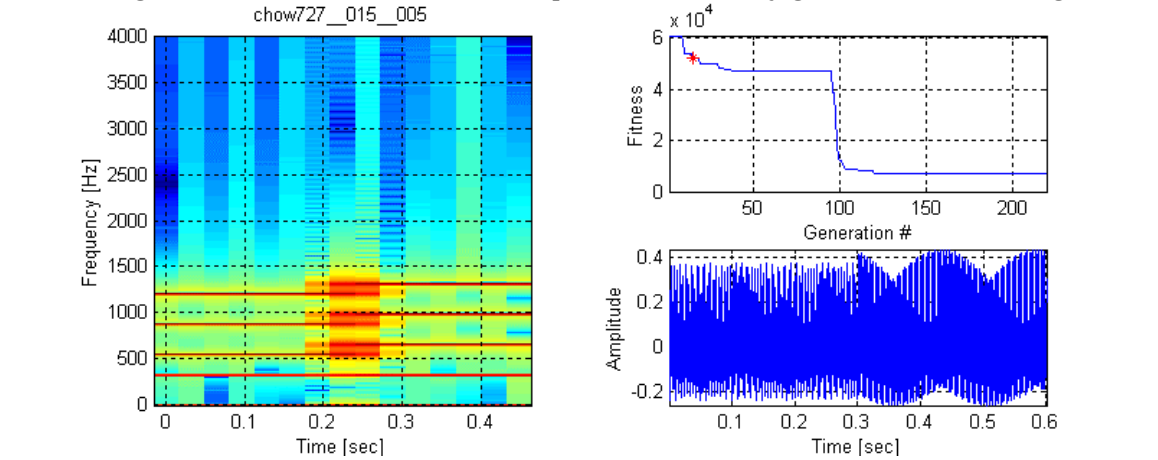


Figure 35 Spectrogram, waveform and topology for best individual of Generation 10, experiment 1.

Generation 15 has a "better" set of harmonics, but note that a frequency around 300 Hz stays fixed during the duration of the two notes. Despite this, it is a very good match for the target.



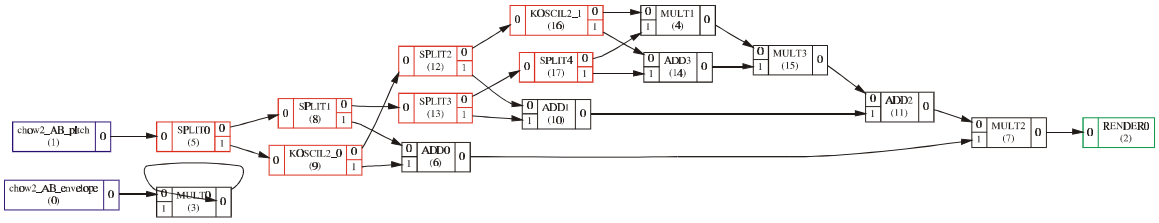


Figure 36 Spectrogram, waveform and topology for best individual of Generation 15, experiment 1.

Generations 20 to 95 explore the space without too much improvement of the "best individual". The produced sound has many of the characteristics of the one in Generation 15 with some extra harmonics, and the tuning improves over time, but with the steady 300 Hz tone present as well.

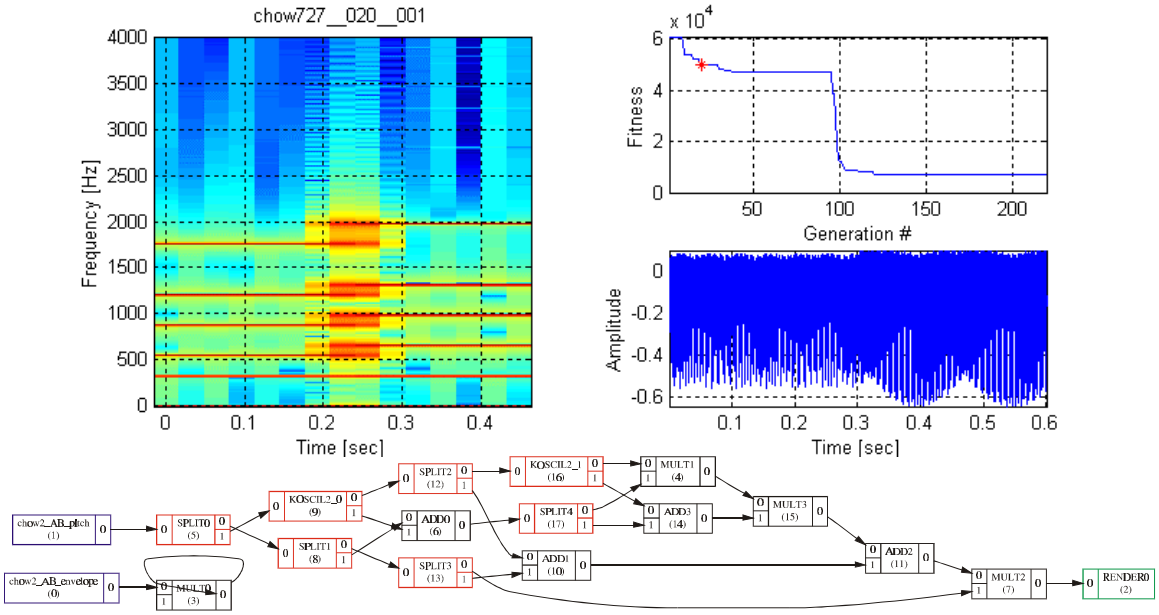


Figure 37 Spectrogram, waveform and topology for best individual of Generation 20, experiment 1.

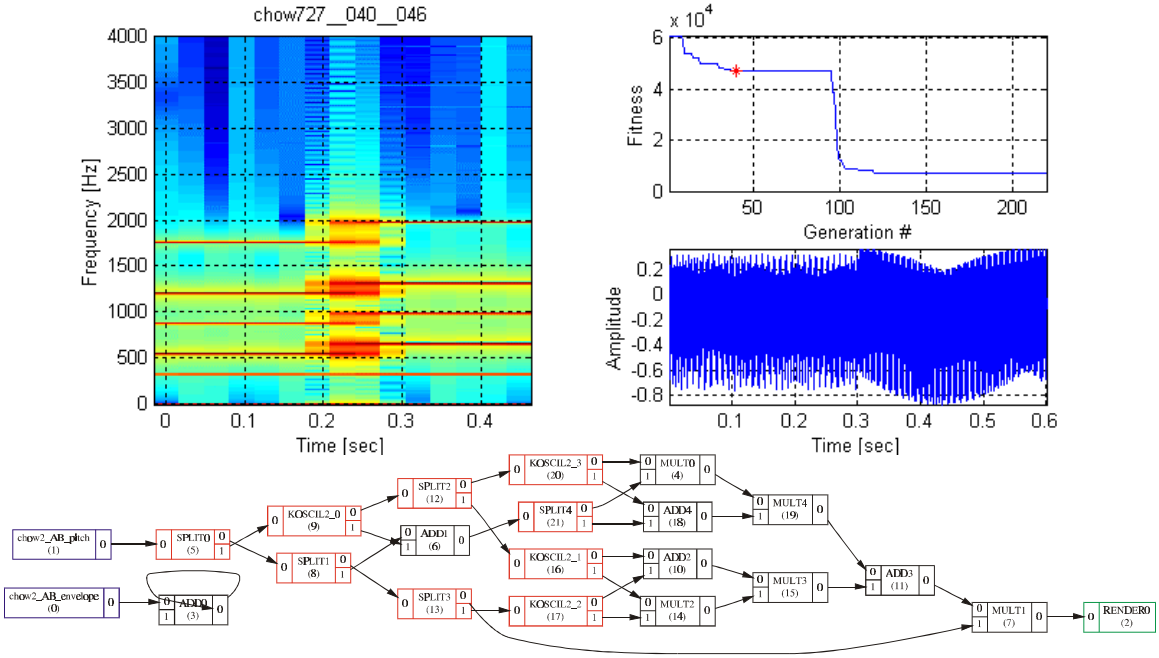


Figure 38 Spectrogram, waveform and topology for best individual of Generation 40, experiment 1.

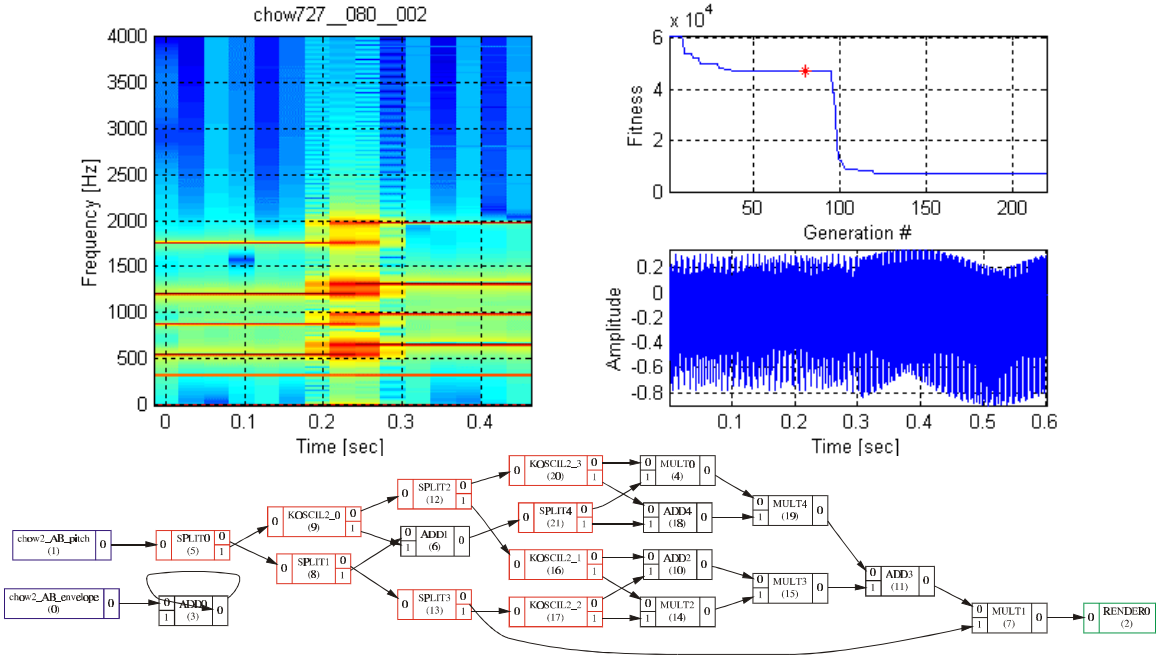


Figure 39 Spectrogram, waveform and topology for best individual of Generation 80, experiment 1.

Generation 96 has an interesting change in the topology suggested. Even thought that it is simpler (less functional elements), the spectrum produced is richer in harmonics. A very important feature is that there are no “constant” tones during the whole duration of the sound. In addition, all the

harmonics “change” at the same instant. This can be interpreted as being controlled in some way by the pitch information.
The number of harmonics in this spectrogram is higher than desired, but some of the harmonics lay at the right frequencies on the spectrum.

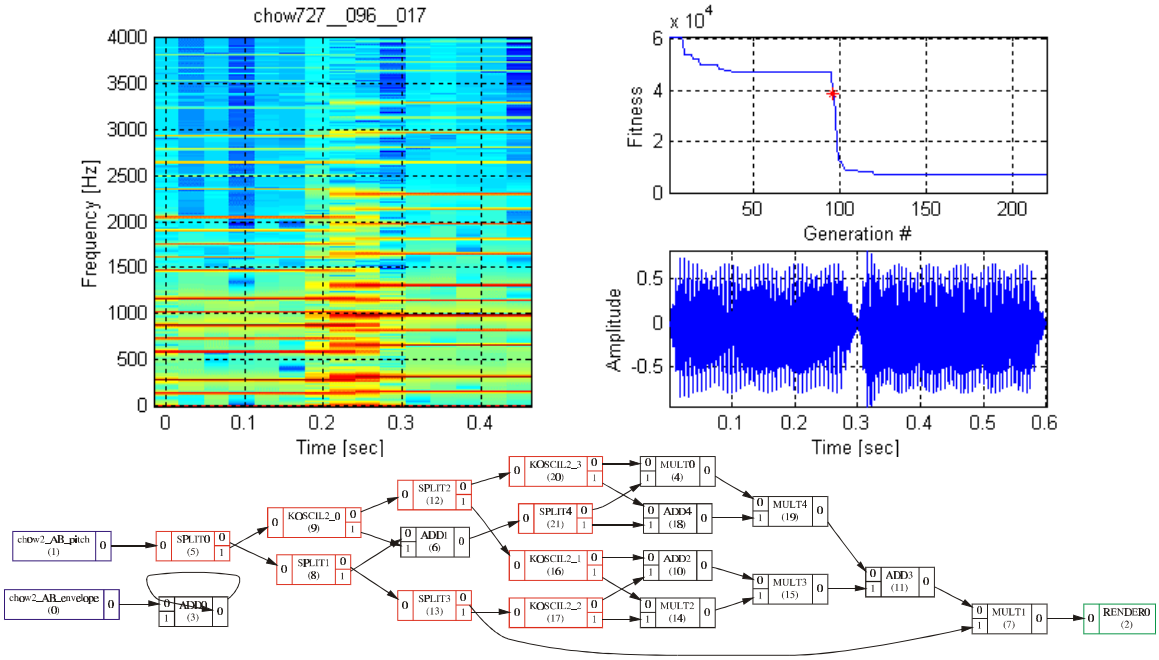
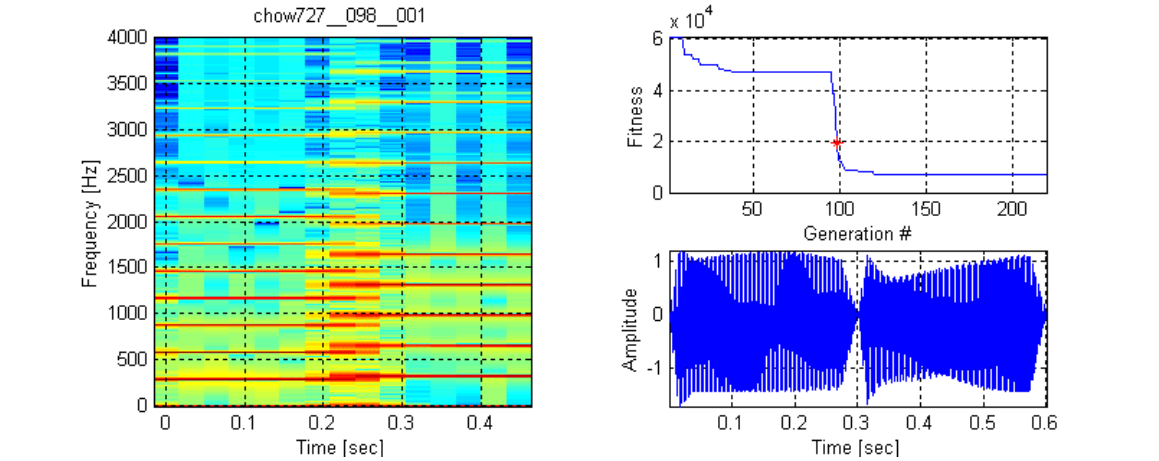


Figure 40 Spectrogram, waveform and topology for best individual of Generation 96, experiment 1.

Generation 98 preserves the same topology as 96, but the internal parameters were changed (during the optimization and selection procedures), and the double set of harmonics were “fine tuned” to be at the right place. The position of the harmonics compared with the target is very accurate, but some of their magnitudes are not correct.



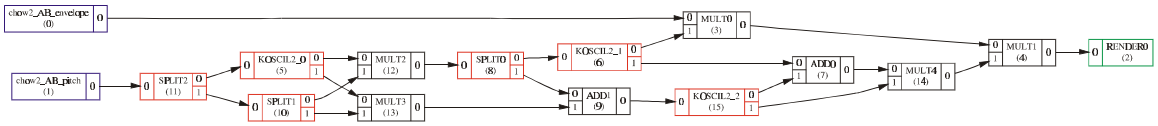


Figure 41 Spectrogram, waveform and topology for best individual of Generation 98, experiment 1.

At Generation 99 and in all subsequent generations the actual topology of the SST is preserved, but the exploration now "focuses" on the internal parameters.

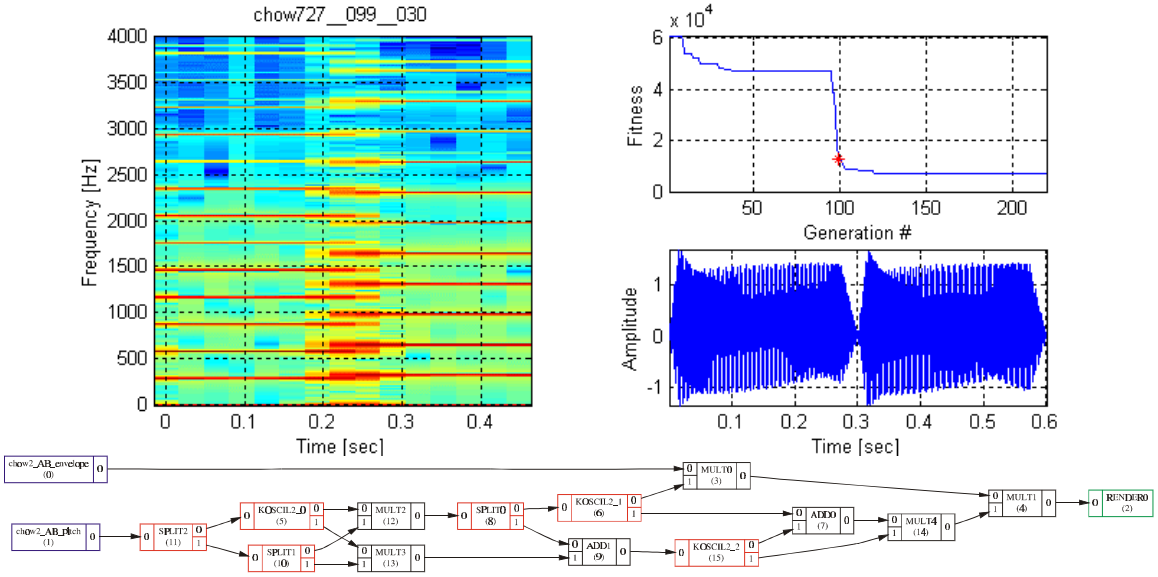


Figure 42 Spectrogram, waveform and topology for best individual of Generation 99, experiment 1.

Generation 100 – 220: During this interval the topology never changed, and the exploration was focused into finding a good set of parameters for fine tuning the frequency and amplitude of the harmonics.

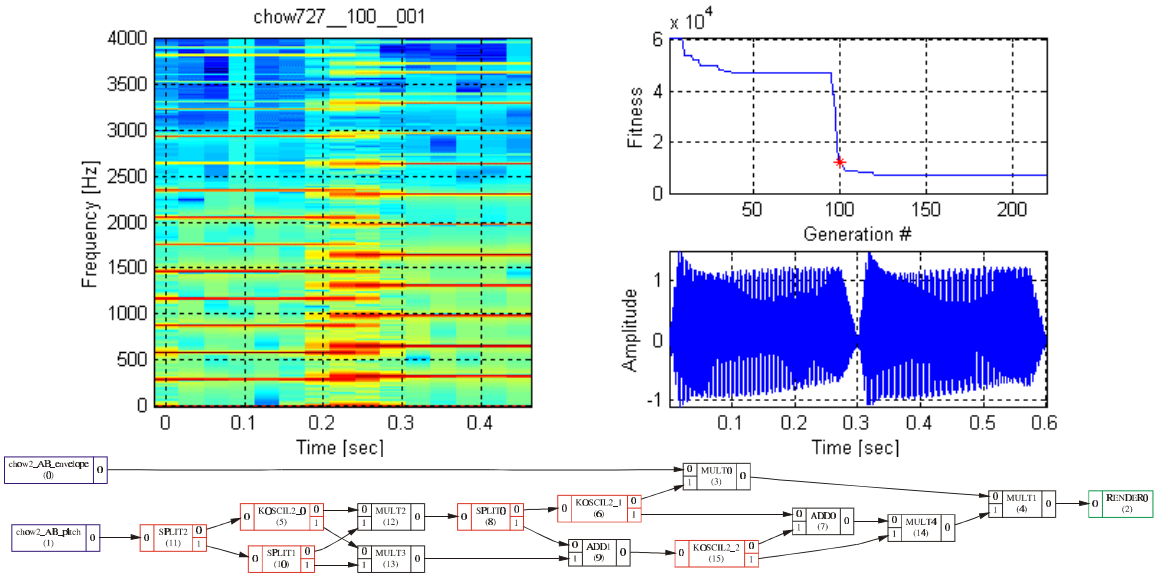


Figure 43 Spectrogram, waveform and topology for best individual of Generation 100, experiment 1.

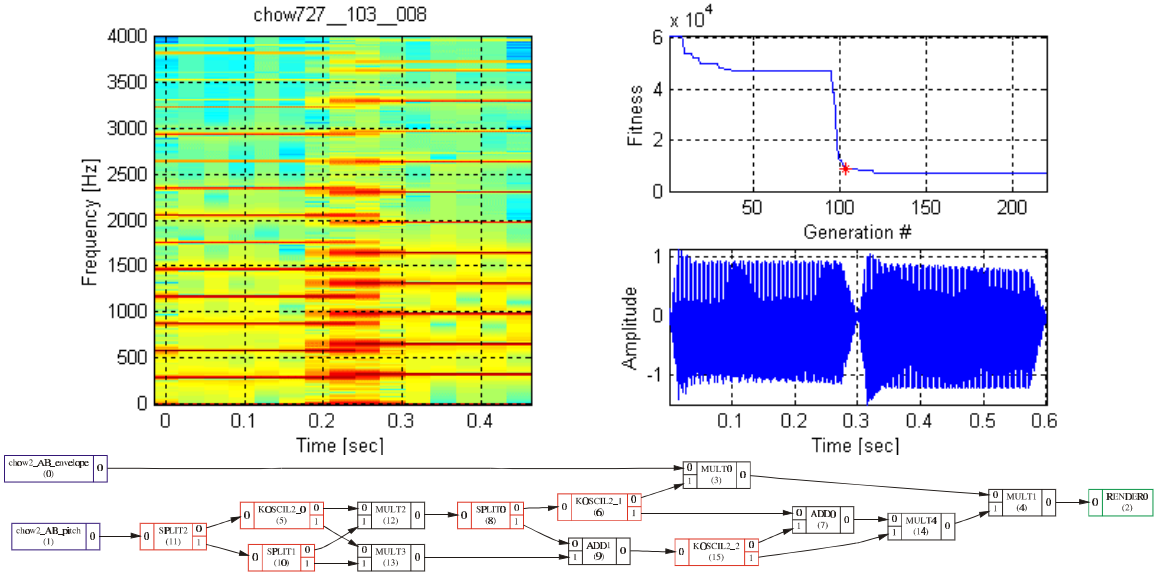


Figure 44 Spectrogram, waveform and topology for best individual of Generation 103, experiment 1.

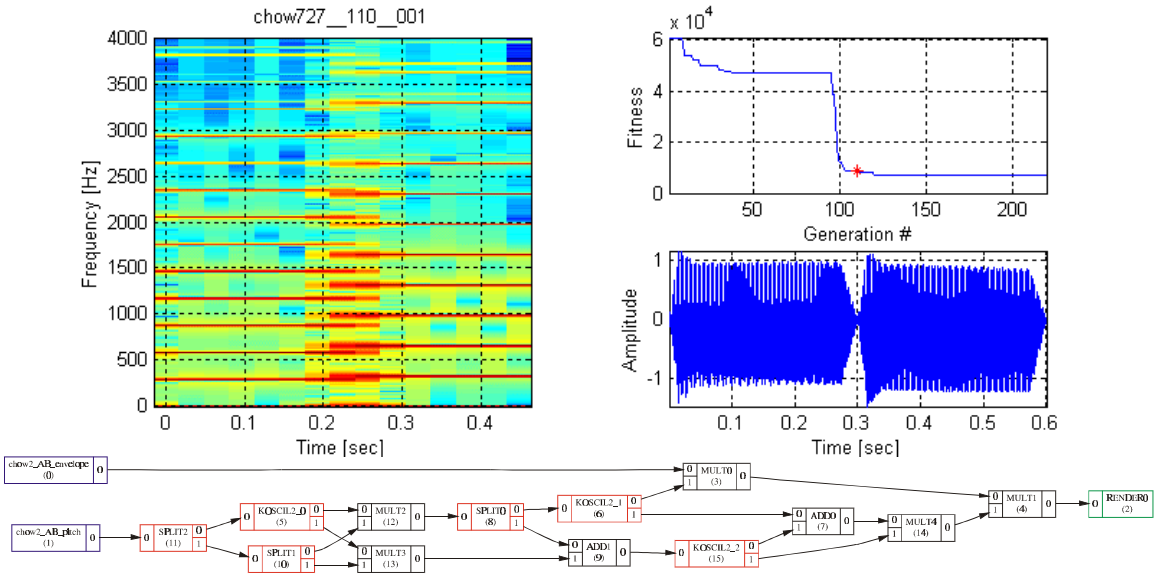


Figure 45 Spectrogram, waveform and topology for best individual of Generation 110, experiment 1.

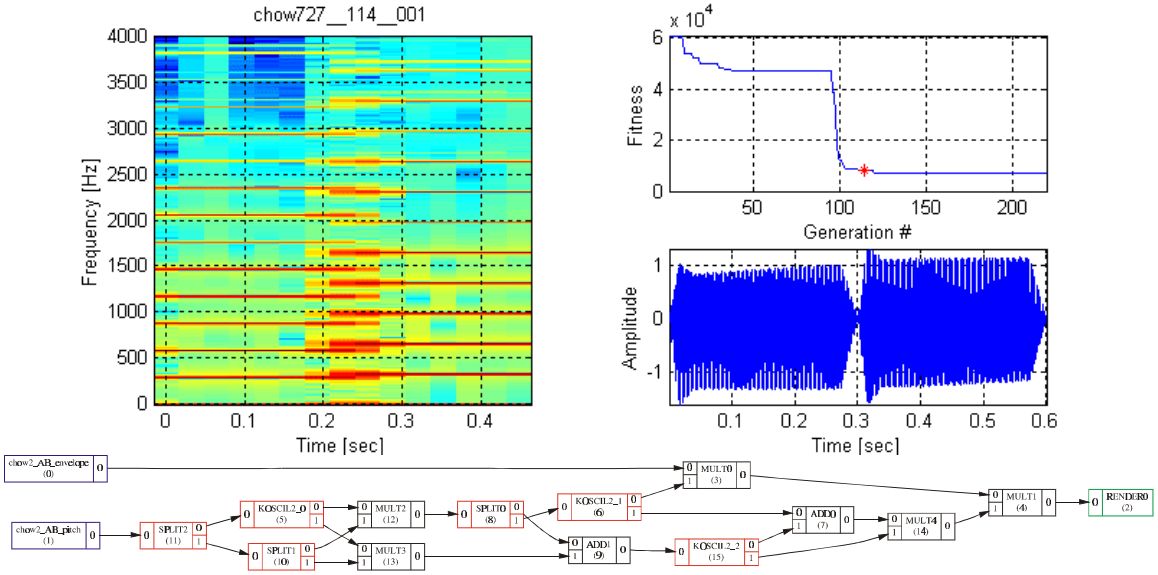


Figure 46 Spectrogram, waveform and topology for best individual of Generation 114, experiment 1.

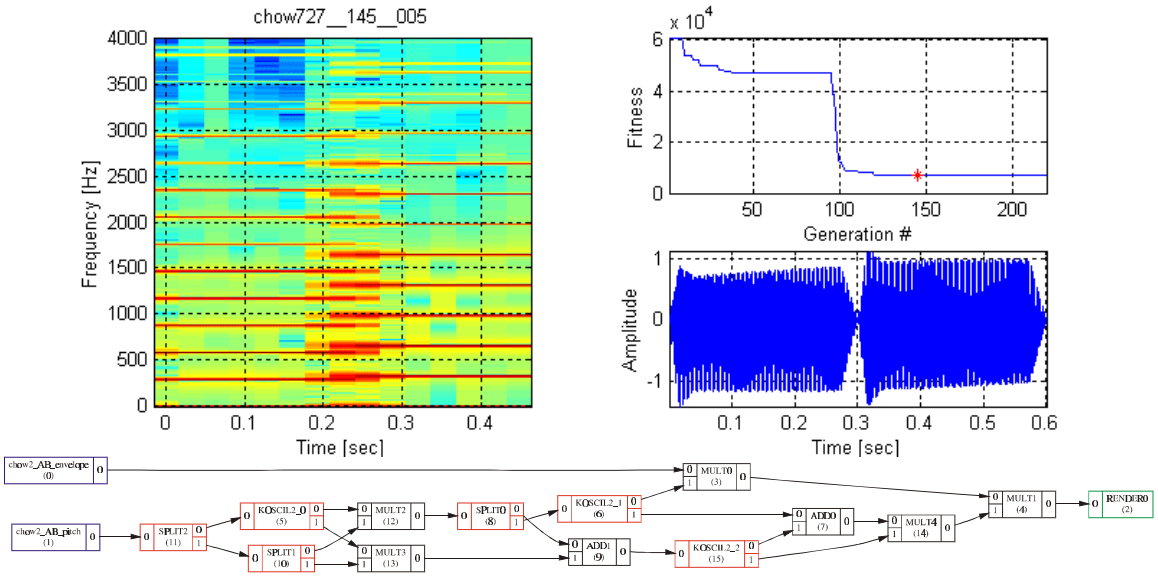


Figure 47 Spectrogram, waveform and topology for best individual of Generation 145, experiment 1.

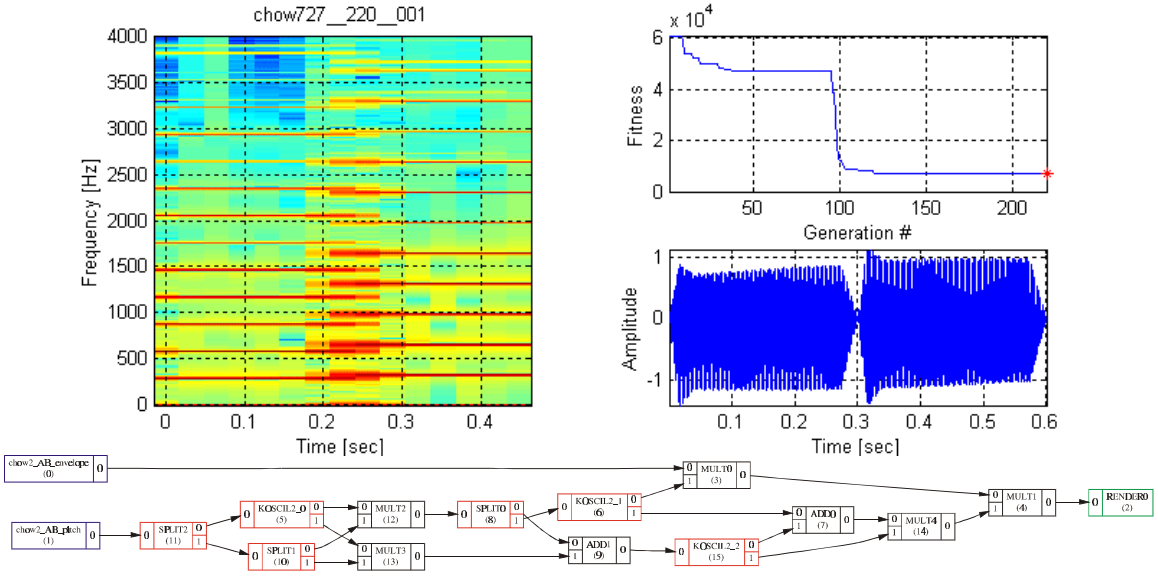


Figure 48 Spectrogram, waveform and topology for best individual of Generation 220, experiment 1.

Note that the final spectrum agrees with the target in all the frequencies of the harmonics. But the final spectrum has higher energy at the high end of the spectrum.

The "meaning" or right use for the inputs (in our case: envelope and pitch) was "found" by the system. This information is never fed into the system in any way. Evolution finds that the changes in the input are correlated to changes in the output, and finds a way of using these changes in a productive way.

6.1.6 Functional form analysis.

The topology evolved in generation 220 is shown in Figure 48. It is possible to analyze the functional elements and their connections to find the close form formula representation of the topology. In this case, it is represented in equation (24).

$$s(t) = k_1 A(t) oscil_2\left(k_0 f(t) + f(t) oscil_0\left(f(t)\right)\right)\left(oscil_1\left(k_0 f(t)\right) + k_2\right) \quad (24)$$

Comparison between equations (23) and (24) shows a close similarity in their functional form. The functional form of FM synthesis is characterized by a sinusoidal modulating the frequency of another sinusoidal. This structure is present in both equations. Focusing on equation (24), note how the envelope *A(t)* is used to actually serve as an envelope. Inside of *oscil2*, it is possible to see a linear term controlled by the frequency (pitch), and oscillator *oscil0* controlled by the frequency signal as well. There is a third oscillator *oscil1* that multiplies the whole system, but a closer analysis shows that it is also controlled by the frequency signal, and varies at the same rate as the first part of the oscillator, reinforcing the effect of the first.
This leads us to the conclusion that the AGeSS system found a solution that shares many of the elements of the original SST; entirely WITHOUT having information about the functional form of the original SST, but using a set of known inputs and the target sound. This is usually known as “system identification” or “system regression”.

AGeSS used only the known inputs, target and fitness function to explore the SST space and suggest the aforementioned SST. The target and inputs contained information about only two pitches (A880, B988). An interesting test would be to try the evolved SST with different pitches. Figure 49 shows a target scale (C523, D587, E659, F698, G784, A880 and B988) played with the original FM formula for experiments 1 and 2.
This test measures to some extent the capabilities of the evolved SST to extrapolate with the control signals, and produce good outputs. A good result would show a similar spectrogram between target and produced sound, as is the case in Figure 50, that was produced using the topology of the best individual of generation 220 from experiment 1.
With the results here, it is possible to say that the functional form evolved really captured the properties of the system, and using only 2 pitches was possible to reverse engineer the system and find a suitable model, capable of extrapolating to a broader set of pitches.

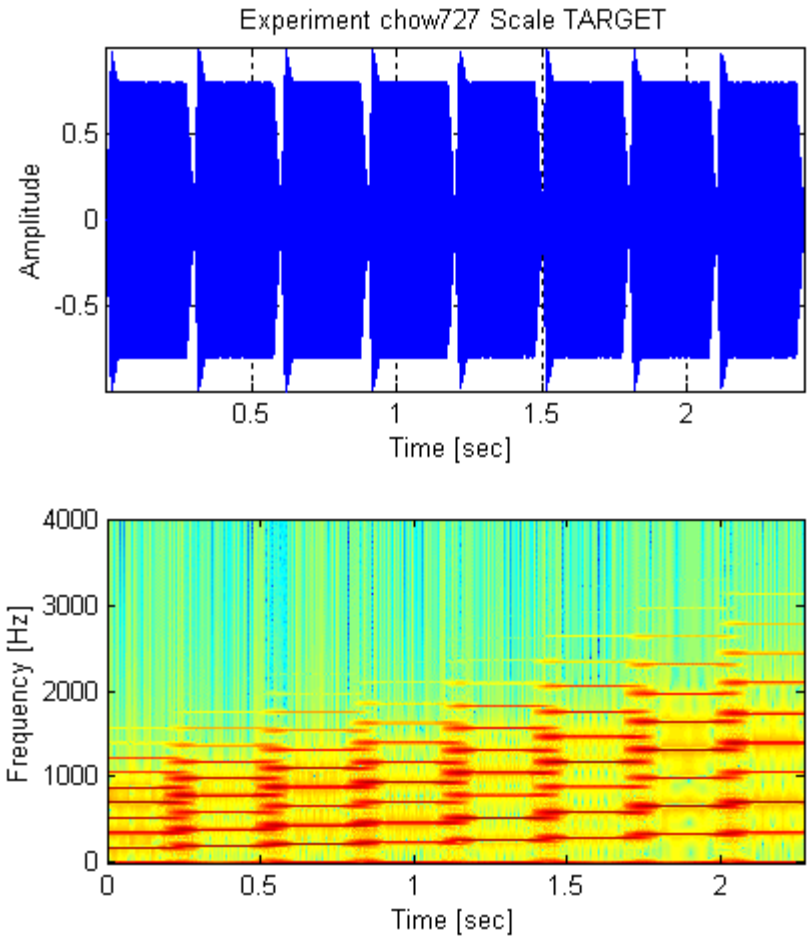


Figure 49 Waveform and spectrogram of scale produced with the FM synthesizer of experiments 1 and 2

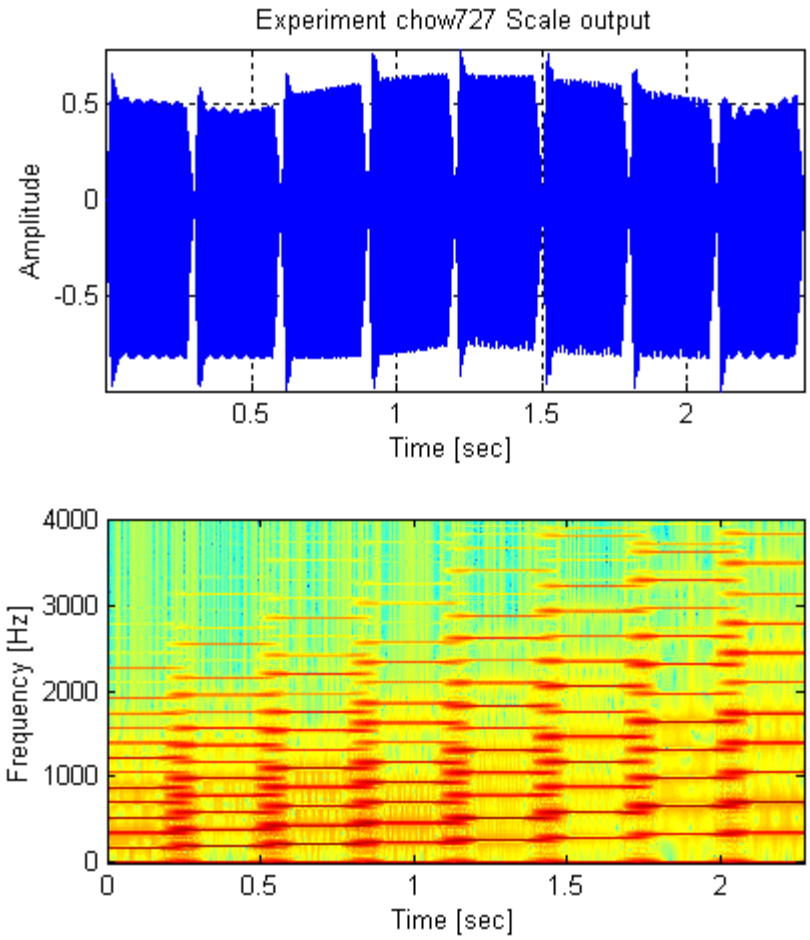


Figure 50 Waveform and spectrogram of scale produced using topology for best individual of Generation 220, experiment 1.

6.2 EXPERIMENT 2. FM SYNTHESIS (CHOW727B)

6.2.1 Selection of a target SST:

The target SST used is identical than the one employed in experiment 1 (section 6.1); a FM synthesizer simulating the sound of a woodwind instrument.

6.2.2 Inputs/target:

The same inputs/target from experiment 1 were used.

6.2.3 Fitness Function:

The same fitness function, SFM LSE than experiment 1 was used.

6.2.4 AGeSS parameters:

Experiment name: chow727b
Allowed functional blocks: SPLIT, KOSCIL, ADD, MULT
Number of individuals: 50
Tree max levels: 15
Tree max nodes: 80
Probabilities for G.O: copy 20%, mutation 40%, crossover 20%, new random individual 20%;
Sub-optimization parameters: G.A. with 7 individuals max, 5 generations max.

6.2.5 Analysis of best of generation individuals:

Figure 51 shows the fitness of the "Best of Generation" individuals across 1600 generations. Note the number of generations before the dramatic improvement in the fitness value, around 700, while this kind of improvement was noticed around only 100 generations during experiment 1.

This kind of behavior is common in genetic programming. There is no guarantee that a solution with the desired performance will be found, neither is possible to estimate the number of generations for having a particular fitness value. This is because the first populations are random individuals, that belong to different parts of the SST space, but there is no guarantee that they are uniformly distributed in this space, neither that they sample the most relevant sections of the space.

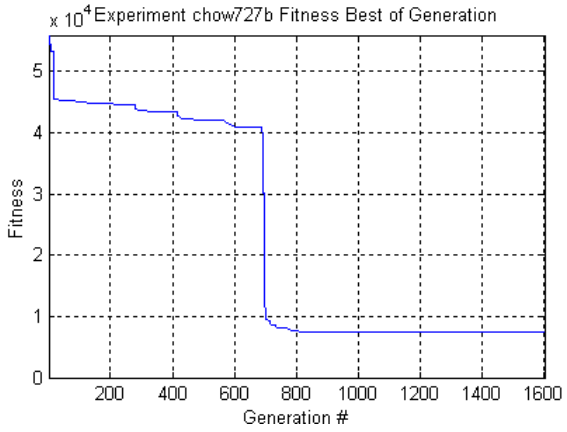


Figure 51 Fitness value for the "best of generation" individuals for experiment 2.

Following, we will show the waveform and spectrograms of some selected individuals during the run, and the topology of the last selected individual.

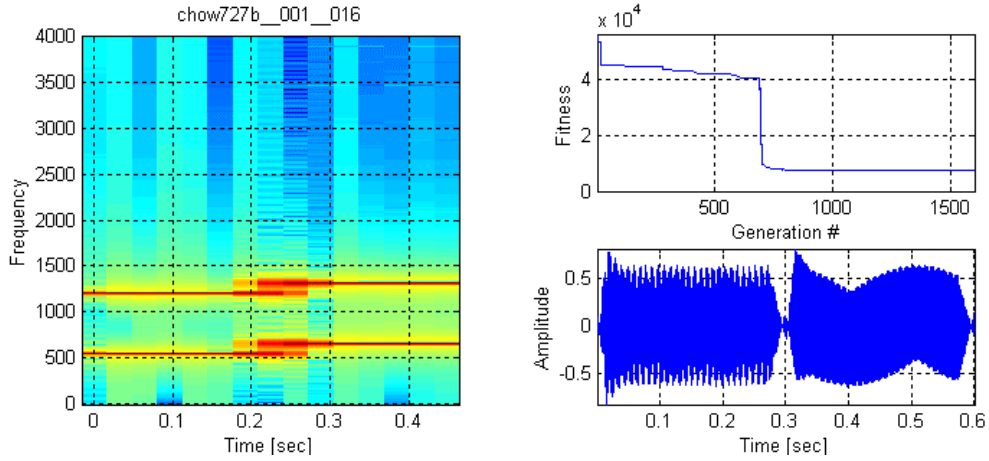


Figure 52 Spectrogram and waveform for best individual of Generation 1, experiment 2.

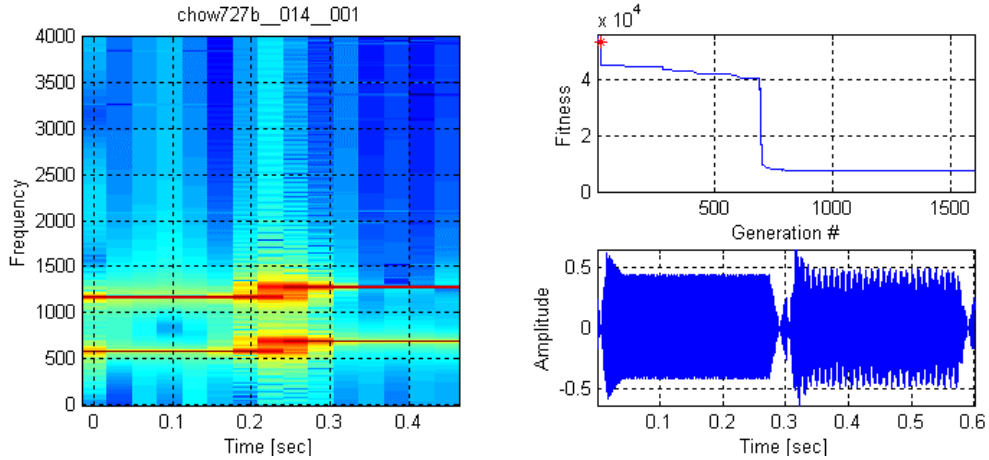


Figure 53 Spectrogram and waveform for best individual of Generation 14, experiment 2.

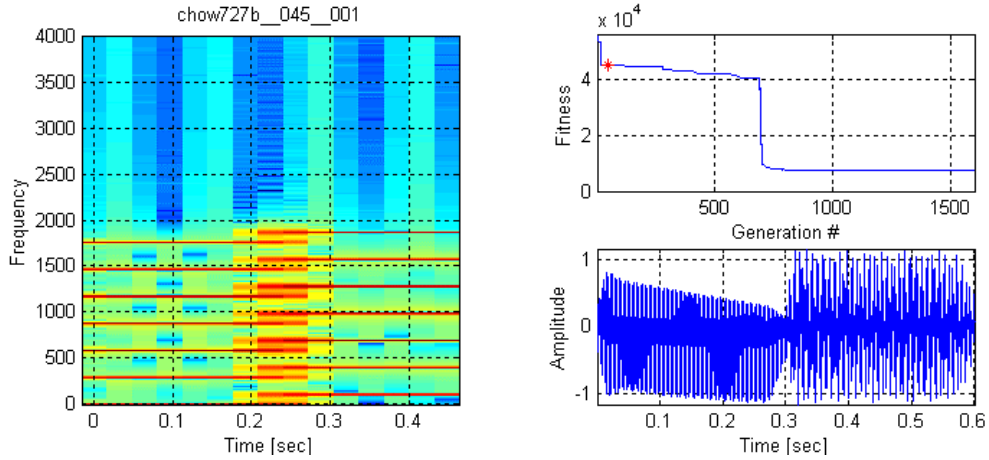


Figure 54 Spectrogram and waveform for best individual of Generation 45, experiment 2.

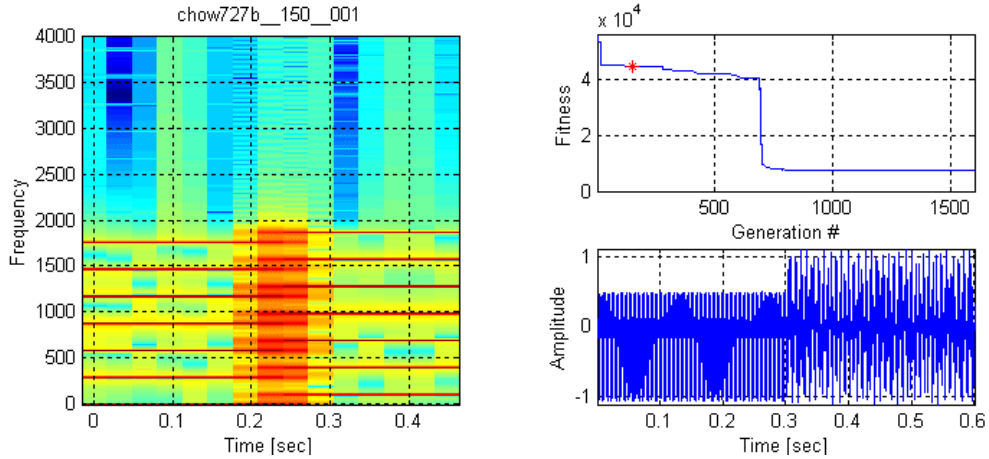


Figure 55 Spectrogram and waveform for best individual of Generation 150, experiment 2.

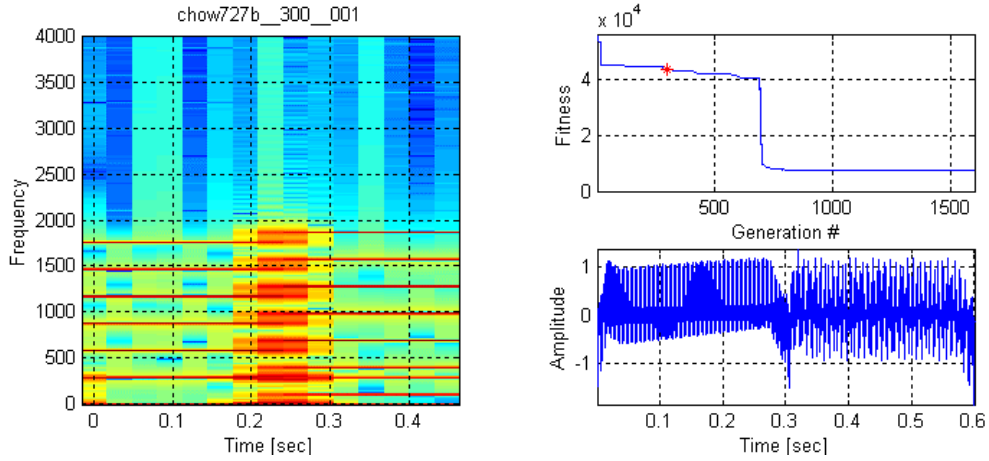


Figure 56 Spectrogram and waveform for best individual of Generation 300, experiment 2.

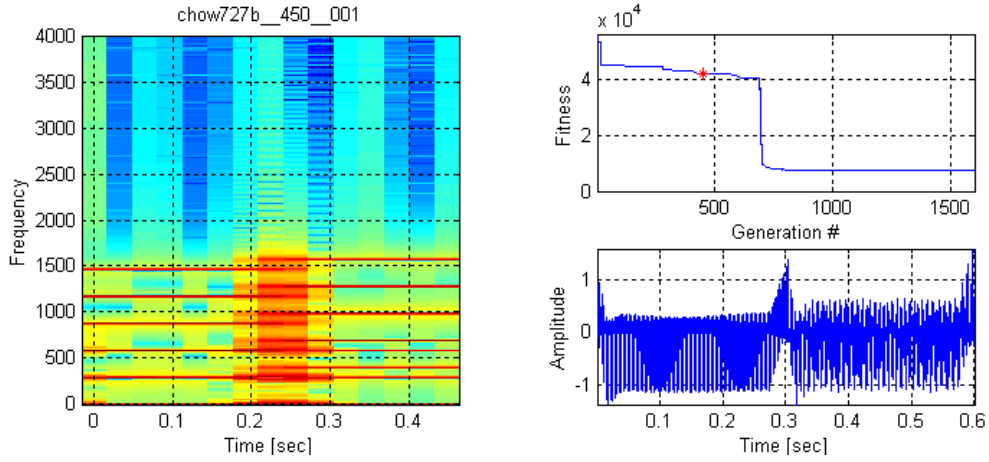


Figure 57 Spectrogram and waveform for best individual of Generation 450, experiment 2.

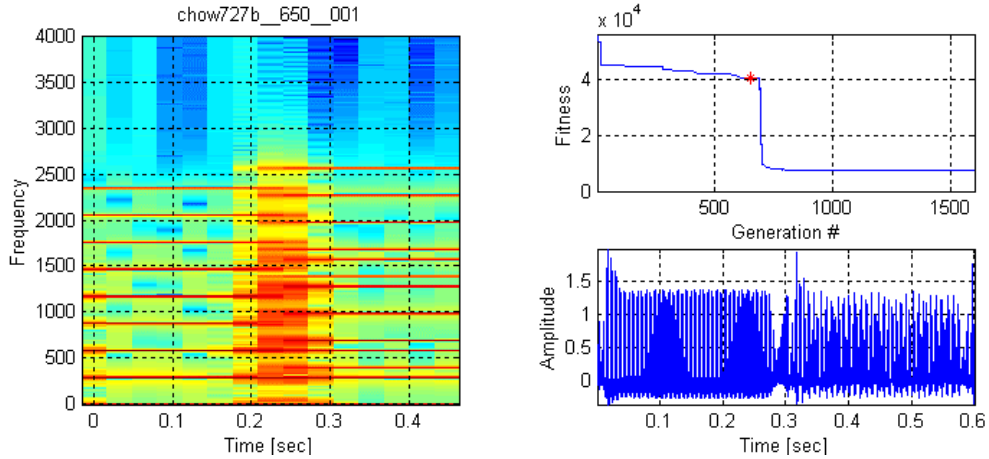


Figure 58 Spectrogram and waveform for best individual of Generation 650, experiment 2.

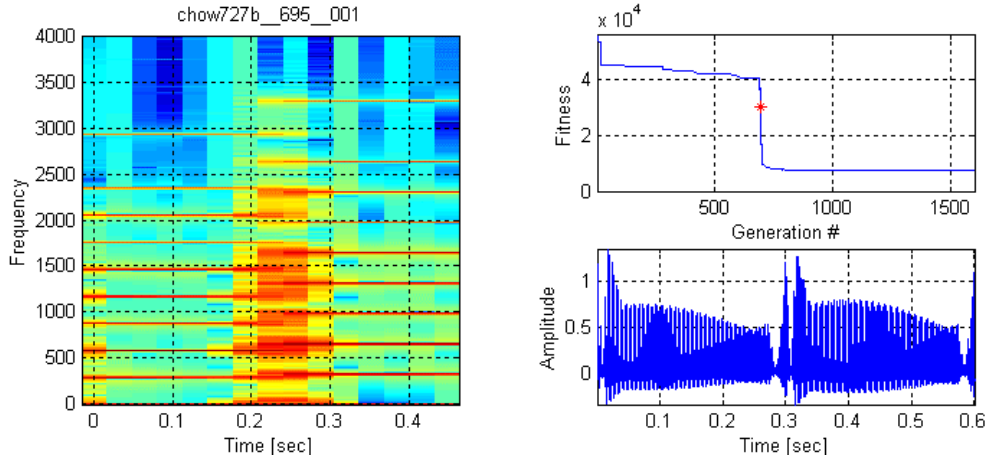


Figure 59 Spectrogram and waveform for best individual of Generation 695, experiment 2.

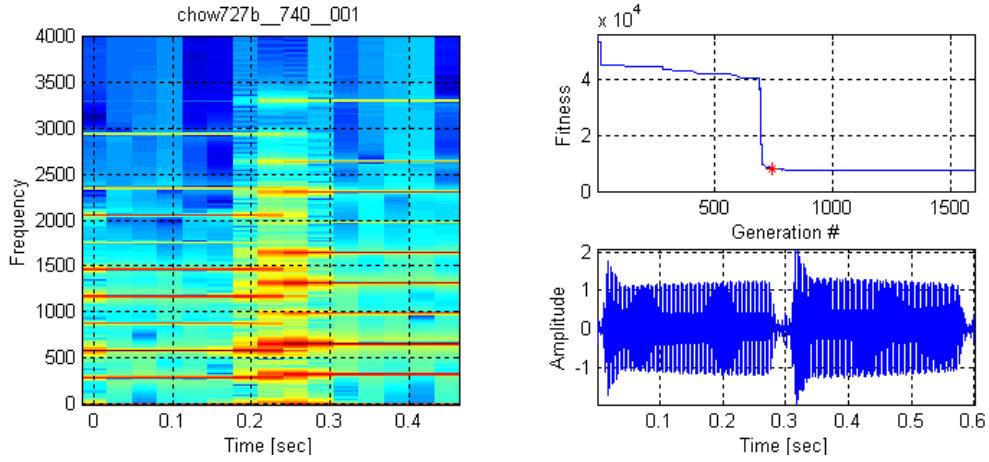


Figure 60 Spectrogram and waveform for best individual of Generation 740, experiment 2.

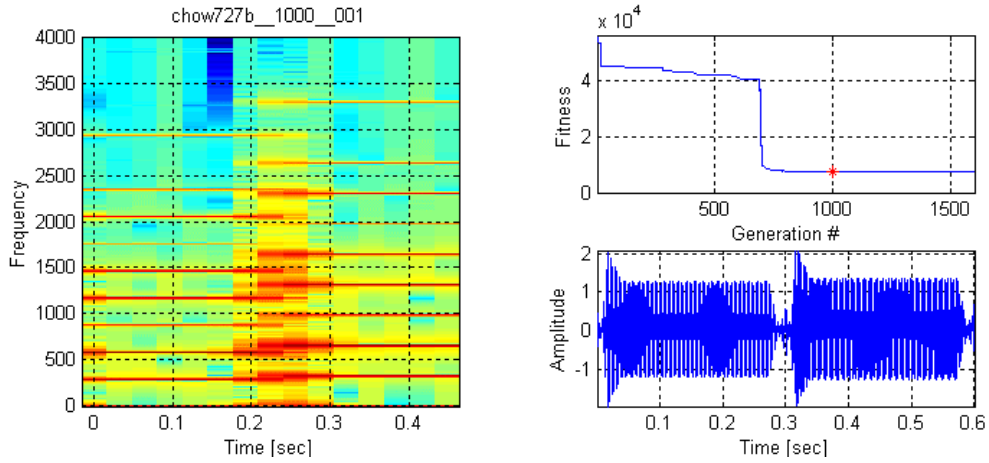


Figure 61 Spectrogram and waveform for best individual of Generation 1000, experiment 2.

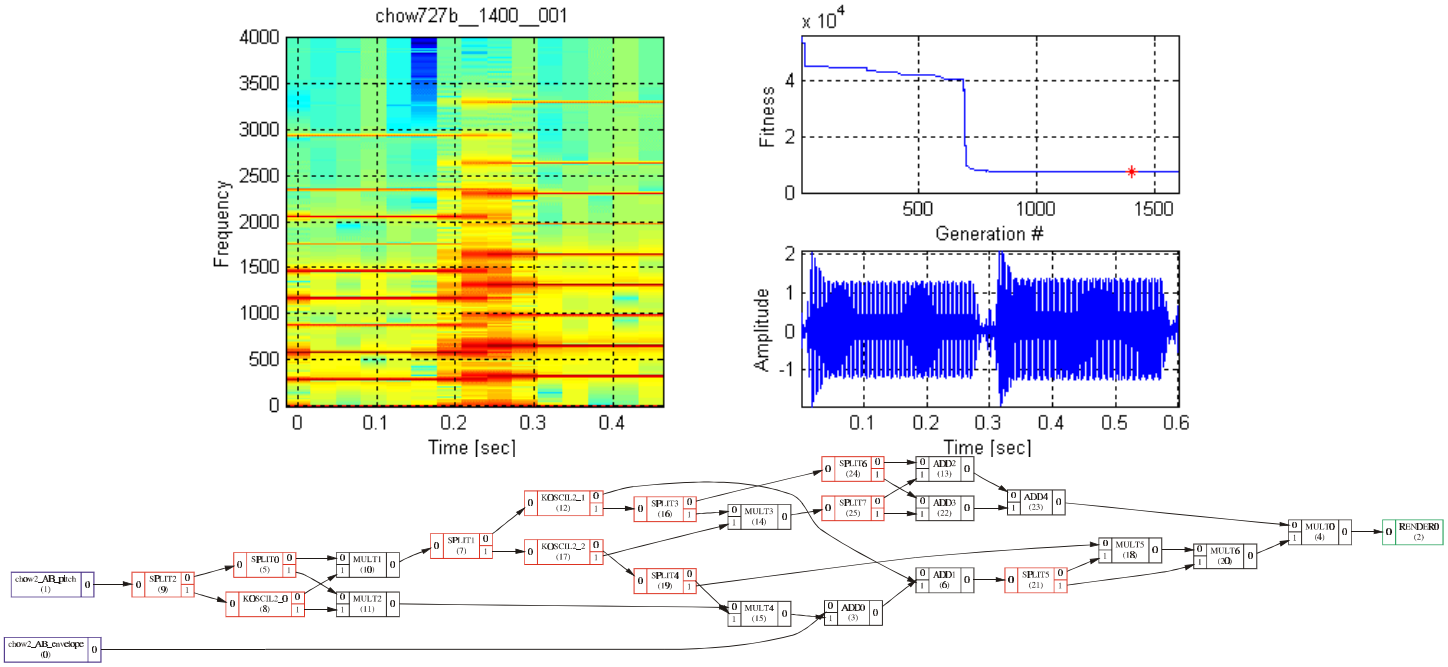


Figure 62 Spectrogram, waveform and topology for best individual of the Generation 1400, experiment 2.

6.3 EXPERIMENT 3. PIANO (DES44)

6.3.1 Selection of a target SST:

For this experiment there is no know target SST. The target sound was recorder from a commercial sound synthesis module.

6.3.2 Inputs/target:

The selected target was a piano note (C261) of about 1 second of duration. The note was sampled from a synthesis module, and the source algorithm was not know.
The inputs were computed from the actual target, as the envelope and extracted pitch.

6.3.3 Fitness Function:

A simple LSE fitness functions was used for this experiment, as discussed in section 4.5.2.

6.3.4 AGeSS parameters:

Experiment name: des44
Allowed functional blocks: SPLIT, KOSCIL, ADD, MULT
Number of individuals: 60
Tree max levels: 30
Tree max nodes: 300
Probabilities for G.O: copy 20%, mutation 60%, crossover 20%,
Sub-optimization parameters: G.A. with 10 individuals max, 4 generations max.

6.3.5 Analysis of best of generation individuals:

Figure 63 shows the fitness of the "Best of Generation" individuals across 120 generations.
In this experiment, the improvement is gradual, and not as sudden as in the first two experiments experiment 1.

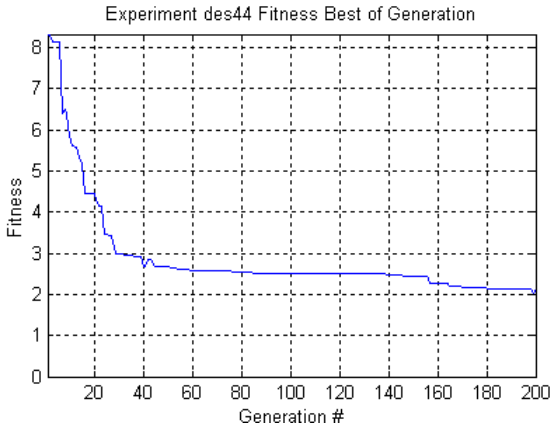


Figure 63 Fitness value for the "best of generation" individuals for experiment 3.

A short summary of spectrograms and waveforms of some of the best individuals of some generations will follow:

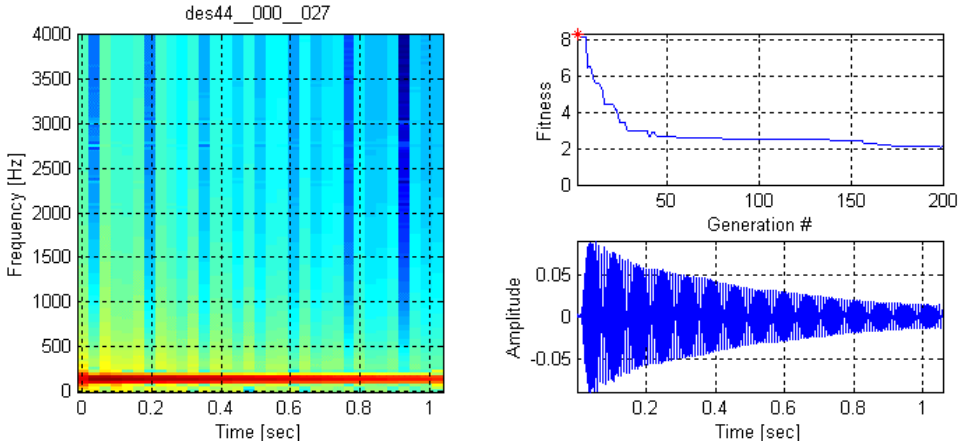


Figure 64 Spectrogram and waveform for best individual of Generation 1, experiment 3.

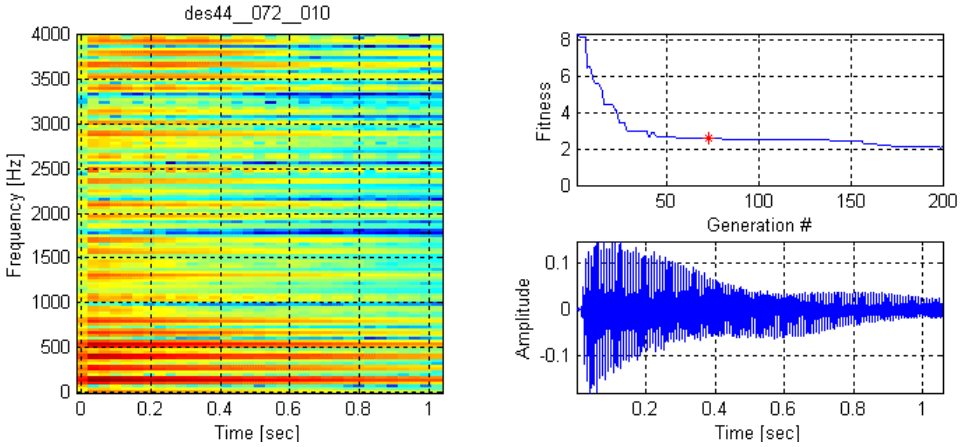


Figure 65 Spectrogram and waveform for best individual of Generation 71, experiment 3.

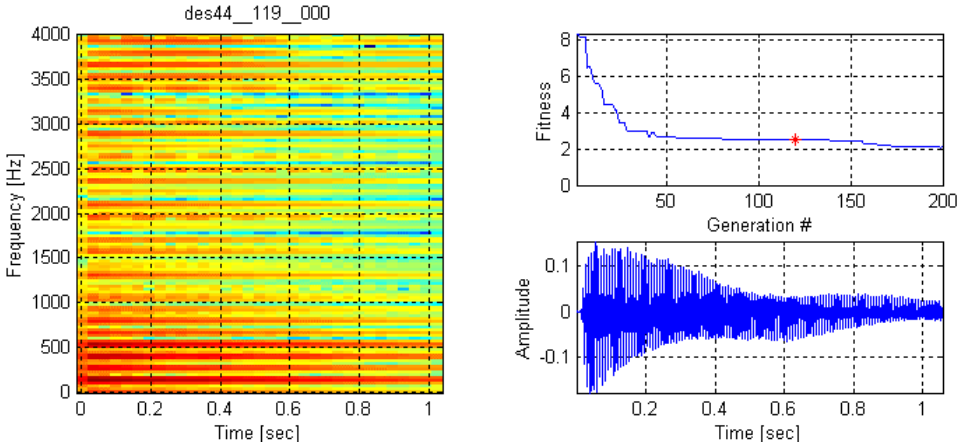


Figure 66 Spectrogram and waveform for best individual of Generation 118, experiment 3.

6.4 SUMMARY OF EXPERIMENTS

The first two experiments provided good insight of the potential of the approach, as well as the kind of results that can be expected with the AGeSS system.

The functional form of the SSTs suggested by AGeSS shows that the approach is capable of finding suitable algorithms capable of synthesizing a given target sound. In informal listening tests, the sounds produced by the first two experiments were remarkably close to the target sound (Chowning's FM instrument). The ability of play a scale, that also keeps the similitude between the example and the produced sound is a very important suggestion; but is important to note that there is no evidence that is the case in all the experiments.

The third experiment, based on an unknown source (piano), gave good results. The sound synthesized with the evolved SSTs had some of the characteristics that a sound designer will look for in a piano: a string hit by a hard object, and the string vibrating in a resonant enclosure. The high harmonics die faster than lower harmonics.

Even though that this experiment was based on examples of a single note (therefore, extrapolation was not enforced by the examples), the results are very encouraging.

7 CONCLUSIONS

Sound synthesizers are computer programs that are designed to produce digital sound samples that can (bur are not limited to) emulate the sound of a given musical instrument. The driving questions of this research were how are these sound synthesis algorithms conceived? Can this design process be automated? The goal was to explore the fundamentals of SSTs and propose an approach for automating their design.

This exploration started by analyzing a set of the so-called “classic SSTs” for regularities in their functional elements and functional form. Then, a representation for SSTs was proposed, capable of representing many of the classic SSTs and even novel ones. With this representation, the SST space is defined as the space spanned by “all” the possible SSTs capable of being represented using a given set of functional elements.
Finally, design is stated as a search in the multidimensional space of the SSTs.

This search is done using evolutionary methods, in particular Genetic Programming, which has proven useful to explore complex spaces like this one.

The following requirements are essential for using GP as a search tool in the SST space:

- A representation of the search space, in this case, the SST space
- A way of manipulating the selected representation.
- A function to evaluate the performance of each SST with regard to an example response.

Representation:

SSTs are algorithms for producing sound. Their representation should have the form of a language capable of expressing their constitutive parts, such as functional elements and functional form. For this language, it is desired that it exhibits the closure property, which states that any SST can be expressed using elements from this language. For our research it is very difficult to probe a closure on the SST space, but it is possible to define a general enough set and language that spans over many possible SSTs.

Another property that is desired is manipulability. SSTs will be “suggested” by an autonomous algorithm. The more structured their representation and ability of being manipulated, the easier it is for the algorithm to keep their validity across manipulations.

Design as a search in the SST space:

Design is stated as a search in the multidimensional SST space. Each point in this space will represent a different functional form and set of internal parameters.
The goal is then to find a point in the SST space that will fulfill the specifications of design. It is not clear how neighboring points are related in this representation. In addition, the number of possible points in this space is huge, making it impossible to do a thorough search of the space. These characteristics make the search of the SST space a very complex problem. Evolutionary

methods, such as Genetic Programming, have proved satisfactory when dealing with these types of problems.

Fitness Function (FF):

This is a measure of the performance of the SST. The FF should return an analytical value that reflects how the measured features in a SST follow a desired set of features in a target example. These features can be measured on the SST itself or in the results (sound samples) produced by the SST.

The analyzed FF measured a set of features extracted from the sound produced by the SST and the target sound from the training examples. Some of them were completely analytical (like the Least Squared Error magnitude spectrogram), but one of them included perceptual criteria by including a model of the human hearing and the simultaneous frequency masking phenomena.

It is possible to fashion different FFs that measure other features as well on the sound produced by the SST or even on the SST itself. An example of the latter is to measure the complexity or efficiency of the produced algorithm, memory requirements, etc.
It is even possible to mix these two (or more) types of FFs to achieve a meta-FF capable of measuring SSTs for an assorted variety of features.

Automatic Generation of Sound Synthesizers (AGeSS) system:

The developed AGeSS system shows the suggested approach for automatic design of SSTs to be practical and feasible.

The experiments show that the selected set of functional elements and the representation scheme are effective for the automated design of some common synthesis algorithms, especially the frequency modulation techniques.
But the major drawback of our implementation was the excessive computation time for each example (from 20 – 200 hours per experiment). This made the development process of the actual system a very extenuating procedure, from simple operations such as bug tracking, to major trials of different sets of functional elements, rules of assembling or number of training examples. A priority in the list for required improvements is the optimization of the system for lower computation times.

7.1 FUTURE DIRECTIONS

One of the most necessary improvements is the optimization of the code of the AGeSS system. This system accomplished the goal of showing the possibilities of the suggested approach, but at this stage is not very practical as a research tool, especially because of the long computation time that it is required for every run.

This can be accomplished using the parallelism inherent in the evolutionary methods. A population of SSTs can be distributed over many processors/computers and run independently for a while, and then exchange results to create a new population and distribute it on different processors.

As far as user interface is concerned, it is desirable to improve the means of parameter entry, pre-processing of the examples and visualization of results.

The experiments outlined in this research cover only some basic aspects of the many possibilities with this approach. New sets of experiments should be tried, such as:

- Evolution of SSTs for more complex sounds, i.e., natural acoustic instruments.
- Multi target evolution: evolution using different number (and type) of targets.
- Exploration with different functional elements: analyze if the type and number of basic functional elements is optimal. Realize experiments evolving SSTs with different types of elements and compare results.

The sets of examples (inputs/target pairs) can be seen as constraints that should be met by the evolved SST. In evolutionary computation, it is possible to change the number and type of constraints over the course of the run. Analysis of the type and number of constraints that could better guide the search in the SST space is needed. For example, some the experiments outlined in this research used constraints (examples) in the form of two pitches. A SST was evolved for these two pitches. It would make sense after having a “good” model for these two pitches, to change the constraints to three or four pitches. In that way, the search in the SST space starts from a point that is “optimal” for at least two pitches.
The point here is that the constraints can be given at a rate that helps to guide the search process in the SST space. If too many constraints are given from the beginning, it is possible that the search is not as successful.

Another course of research that merits attention is the expansion of the developed theory into a more general DSP framework. Even thought the used functional elements and their relationship can comprise the linear, time invariant DSP techniques, they were used from a generative viewpoint. It could be interesting to apply these methods to processing-like problems numerous in DSP.

8 APPENDIX

8.1 SYNTAX RULES FOR EXPRESSION TREES

These rules follow the Backus Naur Form (BNF) (Marcotty and Ledgard 1987). They are used by the proposed Expression Tree representation scheme.
The meta-symbols used are:

:= meaning "is defined as"
| meaning "or"
<> angle brackets used to surround category names (non-terminals, syntax rules names)
^^ hat used to surround terminals

() parentheses used to surround the probability of a given rule (in percentage, from 0 to 100)

RULES 10.txt

```
<START>         :=      (100)   ^START^         <TYPE_A>
                ;

<CCS_A>         :=      (33)    <TYPE_A>
                |       (33)    <TMF_A>
                |       (34)    <DCF_A>
                ;

<TYPE_A>        :=      (50)    ^ADD^   <CCS_A>
                |       (50)    ^MULT^ <CCS_A>
                |       (0)     ^FLTO^  <DCF_A>         <NUMVAL>   <NUMVAL>   <NUMVAL>
<NUMVAL> <NUMVAL> <NUMVAL>
                |       (0)     ^FILTER^        <DCF_A>         <NUMVAL>        <NUMVAL>
<NUMVAL> <NUMVAL> <NUMVAL> <NUMVAL>
                ;

<TMF_A>         :=      (50)    ^SERIES1_A^     <CCS_A> <TYPE_A> <TYPE_B> <CCS_W> <CCS_W>
<CCS_W>
                |       (0)     ^SERIES2_A^     <CCS_A> <TYPE_A> <TYPE_B> <CCS_W> <CCS_W>
<CCS_W>
                |       (0)     ^SERIES3_A^     <CCS_A> <TYPE_A> <TYPE_B> <CCS_W> <CCS_W>
<CCS_W>
                |       (50)    ^PARALLEL1_A^ <CCS_A>    <TYPE_A>    <TYPE_A>    <TYPE_B>
<TYPE_B> <CCS_W> <CCS_W> <CCS_W> <CCS_W> <CCS_W> <CCS_W>
                |       (0)     ^PARALLEL2_A^ <CCS_A>    <TYPE_A>    <TYPE_A>    <TYPE_B>
<TYPE_B> <CCS_W> <CCS_W> <CCS_W> <CCS_W> <CCS_W> <CCS_W>
                |       (0)     ^PARALLEL3_A^ <CCS_A>    <TYPE_A>    <TYPE_A>    <TYPE_B>
<TYPE_B> <CCS_W> <CCS_W> <CCS_W> <CCS_W> <CCS_W> <CCS_W>
                ;

<DCF_A>         :=      (50)    ^NOP_A^                 <CCS_A>
                |       (50)    ^END_A^
                ;


<CCS_B>         :=      (33)    <TYPE_B>
```

```
                |       (33)    <TMF_B>
                |       (34)    <DCF_B>
                ;

<TYPE_B>        :=      (50)    ^SPLIT^ <CCS_B>
                |       (0)     ^KOSCIL^        <DCF_B> <NUMVAL> <NUMVAL>
                |       (50)    ^KOSCIL2^       <DCF_B> <NUMVAL> <NUMVAL> <NUMVAL>
                ;


<TMF_B>         :=      (50)    ^SERIES1_B^     <CCS_B> <TYPE_B> <TYPE_A> <CCS_W> <CCS_W>
<CCS_W>
                |       (0)     ^SERIES2_B^     <CCS_B> <TYPE_B> <TYPE_A> <CCS_W> <CCS_W>
<CCS_W>
                |       (0)     ^SERIES3_B^     <CCS_B> <TYPE_B> <TYPE_A> <CCS_W> <CCS_W>
<CCS_W>
                |       (50)    ^PARALLEL1_B^ <CCS_B>    <TYPE_B>    <TYPE_B>    <TYPE_A>
<TYPE_A> <CCS_W> <CCS_W> <CCS_W> <CCS_W> <CCS_W> <CCS_W>
                |       (0)     ^PARALLEL2_B^ <CCS_B>    <TYPE_B>    <TYPE_B>    <TYPE_A>
<TYPE_A> <CCS_W> <CCS_W> <CCS_W> <CCS_W> <CCS_W> <CCS_W>
                |       (0)     ^PARALLEL3_B^ <CCS_B>    <TYPE_B>    <TYPE_B>    <TYPE_A>
<TYPE_A> <CCS_W> <CCS_W> <CCS_W> <CCS_W> <CCS_W> <CCS_W>
                ;

<DCF_B>         :=      (50)    ^NOP_B^         <CCS_B>
                |       (50)    ^END_B^
                ;

<CCS_W>         :=      (50)    <TMF_W>
                |       (50)    <DCF_W>
                ;

<TMF_W>         :=      (33)    ^RECONNECT^             <CCS_W>
                |       (33)    ^RECONNECT2^    <CCS_W>
                |       (34)    ^RECONNECT3^    <CCS_W>
                |       (0)     ^RECONNECT4^    <CCS_W>
                |       (0)     ^RECONNECT5^    <CCS_W>
                ;

<DCF_W>         :=      (50)    ^NOP_W^         <CCS_W>
                |       (50)    ^END_W^
                ;

<NUMVAL>        :=      (100)   ^CONSTANT^
                ;
```

End rules:

The size of the trees was controlled by restricting the maximum allowed number of levels (depth) and the maximum allowed number of nodes. These values are difficult to follow strictly, but it is possible to use them as general guidelines. The suggested tree generation algorithm was taken from (Koza 1992), and it was slightly modified to use two creation rules files. The firs file was used at the “beginning” of the tree, when no limit in nodes or depth has been reached. When any of these limits is reached, the end-rules file is used. This file is conceived in a way that will

"gracefully" terminate the branch of the tree. In short: no nodes with further branches will be allowed there.

End Rules 10.txt

```
<CCS_A>                 :=      (100)   <DCF_A>
                ;

<TYPE_A>        :=      (50)    ^ADD^   <CCS_A>
                |       (50)    ^MULT^ <CCS_A>
                |       (0)     ^FLTO^  <DCF_A> <NUMVAL> <NUMVAL> <NUMVAL> <NUMVAL>
<NUMVAL> <NUMVAL>
                |       (0)     ^FILTER^        <DCF_A>         <NUMVAL>        <NUMVAL>
<NUMVAL> <NUMVAL> <NUMVAL> <NUMVAL>
                ;


<DCF_A>                 :=      (100)   ^END_A^
                ;


<CCS_B>                 :=      (100)   <DCF_B>
                ;

<TYPE_B>        :=      (50)    ^SPLIT^ <CCS_B>
                |       (0)     ^KOSCIL^        <DCF_B> <NUMVAL> <NUMVAL>
                |       (50)    ^KOSCIL2^       <DCF_B> <NUMVAL> <NUMVAL> <NUMVAL>
                ;


<DCF_B>                 :=      (100)   ^END_B^
                ;

<CCS_W>                 :=      (100)   <DCF_W>
                ;

<DCF_W>                 :=      (100)   ^END_W^
                ;

<NUMVAL>        :=      (100)   ^CONSTANT^
                ;
```

By following these rules, it is possible to keep validity of the trees even under strong manipulation, or random generation.
An example of a tree generated using these rules can be seen in Figure 67.

Figure 67 Example of expression tree using syntax rules from rules10.txt

8.2 SOUND SYNTHESIS ENGINE

Messaging System
A custom engine for sound synthesis was developed. It uses an object-oriented approach of interconnected functional elements, which exchange sound samples.
Each functional element (block) encapsulates all the functionality required by a functional element drawn from the analysis of the SSTs performed in section 2.3. The communication between functional elements is done using a central messaging loop. Every time that a block has "all new" inputs in its input pins, the processing procedure is performed (using the inputs) and a new output is processed. Outputs are posted into the message loop to be delivered to their block destinations.
This type of system was inspired by Microsoft's Component Object Model (COM) (Rogerson 1997) used in technologies as Directshow and DirectX, but is important to note that the system was developed from scratch, without using any proprietary library or resource.

The types of blocks used where:

SOURCE BLOCK
With no inputs and one or more outputs. This block usually reads information in a file (sound samples) and output them.

PROCESSING BLOCK
One or more inputs and one or more outputs.
These blocks (most blocks are of this type) have the ability of reading sound samples at their inputs, and when "all" of the inputs have "new" information, a processing function is called. The processing function can perform any desired operation on the input samples, and produces output samples.

RENDERER BLOCK
One or more inputs and no outputs. These blocks write the sound samples that are input into a file.

9 REFERENCES

Boulanger, R. C. (2000). The Csound book : perspectives in software synthesis, sound design, signal processing, and programming. Cambridge, Mass., MIT Press.

Casey, M. A. (1998). Auditory group theory with applications to statistical basis methods for structured audio. Program in Media Arts & Sciences, Massachusetts Institute of Technology**:** 172 p.

Chowning, J. (1973). "The synthesis of complex audio spectra by means of frequency modulation." Journal of the Audio Engineering Society **21**(7): 526-534.

Cook, P. R. (1999). Music, cognition, and computerized sound : an introduction to psychoacoustics. Cambridge, Mass., MIT Press.

Depoli, G. (1983). "A Tutorial on Digital Sound Synthesis Techniques." Computer Music Journal **7**(4): 8-26.

Garcia, R. A. (1999). Digital watermarking of audio signals using a psychoacoustic auditory model and spread spectrum theory. Music Engineering Technology. Coral Gables, Florida, University of Miami**:** 133.

Gershenfeld, N. A. (1999). The nature of mathematical modeling. New York, Cambridge University Press.

Gruau, F. (1992). Cellular encodign of genetic neural networks. Lyon, Ecole Normale Superiéure de Lyon.

Gruau, F. (1993). Genetic synthesis of modular neural networks. Proceedings of the Fifth International Conference on Genetic Algroithms. S. e. Forrest. San Mateo, CA, Morgan Kaufmann Publishers**:** 318-325.

Horner, A., J. Beauchamp, et al. (1993). "Machine Tongues .16. Genetic Algorithms and Their Application to Fm Matching Synthesis." Computer Music Journal **17**(4): 17-29.

Johnson, C. G. (1999). Exploring the sound-space of synthesis algorithms using interactive genetic algorithms. Proceedings of the AISB Workshop on Articial Intelligence and Musical Creativity, Edinburgh, In G. A. Wiggins, editor.

Jones, N. (1998). FX2, http://www.njones.demon.co.uk/fx2index.htm.

Juliff, P. L. (1986). Program design. Englewood Cliffs, N.J., Prentice-Hall.

Koutsofios, E. and S. C. North (1996). Drawing graphs with dot. Murray Hill, NJ, AT&T Bell Laboratories**:** 23 p.

Koza, J. R. (1992). Genetic programming : on the programming of computers by means of natural selection. Cambridge, Mass., MIT Press.

Koza, J. R. (1994). Genetic programming II : automatic discovery of reusable programs. Cambridge, Mass., MIT Press.

Koza, J. R. (1999). Genetic programming III : darwinian invention and problem solving. San Francisco, Morgan Kaufmann.

Koza, J. R., F. H. Bennett, III, et al. (1997). "Automated synthesis of analog electrical circuits by means of genetic programming." IEEE Transactions on Evolutionary Computation **1**(2): 109 - 128.

Kronsjö, L. I. (1987). Algorithms : their complexity and efficiency. Chichester ; New York, Wiley.

Lindquist, C. S. (1989). Adaptive & Digital Signal Processing with Digital Filtering Applications. Miami, FL, Steward & Sons.

Marcotty, M. and H. Ledgard (1987). The world of programming languages. New York, Springer-Verlag.

Moore, F. R. (1990). Elements of computer music. Englewood Cliffs, N.J., Prentice Hall.

Oppenheim, A. V., R. W. Schafer, et al. (1999). Discrete-time signal processing. Upper Saddle River, N.J., Prentice Hall.

Pohlmann, K. C. (1995). Principles of digital audio. New York, McGraw-Hill.

Press, W. H. (1992). Numerical recipes in C : the art of scientific computing. Cambridge ; New York, Cambridge University Press.

Puckette, M. (1996). Pure Data. Proceedings, International Computer Music Conference, San Francisco, International Computer Music Association.

Puckette, M. and D. Zicarelli (1991). MAX Development Package Manual. Palo Alto, Opcode, Inc.

Quarles, T., A. R. Newton, et al. (1994). SPICE 3 Version 3F5 User's manual, Dept. Elect. Eng. Comp. Sci., Univ of California, Berkeley, CA.

Rabiner, L. R. and R. W. Schafer (1978). Digital processing of speech signals. Englewood Cliffs, N.J., Prentice-Hall.

Roads, C. (1994). The computer music tutorial. Cambridge, Mass., MIT Press.

Roederer, J. G. (1995). The physics and psychophysics of music : an introduction. New York, Springer-Verlag.

Rogerson, D. (1997). Inside COM Microsoft's Component Object Model. Redmond, WA, Microsoft Press.

Serra, X. and J. Smith, III (1990). “Spectral Modeling Synthesis: A Sound Analysis/Synthesis System Based on a Deterministic plus Stochastic Decomposition.” Computer Music Journal **14**(4): 12-24.

Steele, G. L. (1984). COMMON LISP : the language. Burlington, MA, Digital Press.

Wehn, K. (1998). Using ideas from natural selection to evolve synthesized sounds. Proceedings of the Digital Audio Effects DAFX98 workshop, Barcelona.

Wun, C. W. and A. Horner (2001). “Perceptual wavetable matching for synthesis of musical instrument tones.” Journal of the Audio Engineering Society **49**(4): 250-262.