Automatic Design of Sound Synthesizers as Pure Data Patches using Coevolutionary Mixed-typed Cartesian Genetic Programming

Matthieu Macret
School of Interactive Arts and Technology
Simon Fraser University
B.C., Canada V3T 0A3
mmacret@sfu.ca

Philippe Pasquier
School of Interactive Arts and Technology
Simon Fraser University
B.C., Canada V3T 0A3
pasquier@sfu.ca

ABSTRACT

A sound synthesizer can be defined as a program that takes a few input parameters and returns a sound. The general sound synthesis problem could then be formulated as: given a sound (or a set of sounds) what program and set of input parameters can generate that sound (set of sounds)? We propose a novel approach to tackle this problem in which we represent sound synthesizers using Pure Data (Pd), a graphic programming language for digital signal processing. We search the space of possible sound synthesizers using Coevolutionary Mixed-typed Cartesian Genetic Programming (MT-CGP), and the set of input parameters using a standard Genetic Algorithm (GA). The proposed algorithm coevolves a population of MT-CGP graphs, representing the functional forms of synthesizers, and a population of GA chromosomes, representing their inputs parameters. A fitness function based on the Mel-frequency Cepstral Coefficients (MFCC) evaluates the distance between the target and produced sounds. Our approach is capable of suggesting novel functional forms and input parameters, suitable to approximate a given target sound (and we hope in future iterations a set of sounds). Since the resulting synthesizers are presented as Pd patches, the user can experiment, interact with, and reuse them.

Categories and Subject Descriptors

H.5.5 [**Sound and Music Computing**]: Methodologies and techniques.

General Terms

Algorithms

Keywords

Cartesian Genetic Programming; Coevolution; Sound Synthesis

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
GECCO'14, July 12–16, 2014, Vancouver, BC, Canada.
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-2662-9/14/07 ...$15.00.
http://dx.doi.org/10.1145/2576768.2598303.

1. INTRODUCTION

Replicating sounds using synthesis techniques is a problem frequently addressed in the field of computer music [12]. A well-researched problem is the following: Given a target sound and a synthesizer, what set of input parameters can replicate or approximate this sound. Manually estimating input parameters of a particular synthesizer is usually difficult and time consuming, especially when there is no intuitive relationship between the input parameter values and the sound produced. This search for input parameters is then a good candidate for an automatic optimization scheme. Thus, diverse optimization methods have been used for automatic calibration, such as Particle Swarm [11], HMM [34], Neural Nets [27], Cellular Automata [29] and Genetic Algorithms [12]. It has been suggested that evolutionary approaches such as Genetic Algorithms (GAs) are performing the best to matching musical instrument tones [27]. GAs have been used extensively for estimating the parameters of various synthesis techniques [1, 8, 13, 14, 17, 22, 28, 31] and real-world synthesizers [16, 33]. For instance, we developed a GA-based system [17] able to automatically calibrate a ModFM synthesizer to replicate harmonic instrument tones. We then worked on a more complex problem: calibrating the OP-1 synthesizer [16]. This commercial synthesizer contains several synthesis engines, effects and Low-Frequency-Oscillators, which make the parameter search space large, complex, but also discontinuous. In order to tackle this difficult problem we proposed using a Non-dominated Sorting Genetic Algorithm-II (NSGA-II) with multiple objectives.

In this work, instead of only searching the space of input parameters for a specific synthesizer, we attempt to solve the more general sound synthesis problem: given a sound, what program (sound synthesizer) and set of input parameters can generate that target sound? We propose a system in which we represent sound synthesizers as directed acyclic graphs using the graphical programming language: Pure Data (Pd). We search the space of possible sound synthesizers using Coevolutionary Mixed-typed Cartesian Genetic Programming (MT-CGP). The proposed algorithm co-evolves a population of MT-CGP graphs, representing *functional forms* of synthesizers, and a population of GA chromosomes, representing their *inputs parameters*. A fitness function based on the Mel-frequency Cepstral Coefficients (MFCC) evaluates the distance between the target and produced sounds. We run experiments using contrived and recorded target

sounds. The resulting sounds and synthesizers presented as Pd patches are available online [5] and are discussed in Section 5.

2. RELATED WORK

Only a few attempts have been made at solving the general sound synthesis problem. The resulting systems differ from other GA sound matching systems [1, 8, 13, 14, 17, 22, 28, 31] in that they are not only searching the *inputs parameters* but also the *functional form* of the synthesizer.

Takala et al. [30] evolved what they called *Timbre Trees*. The nodes of these trees were arithmetic operations, analytic functions or noise generators. They defined a roulette-wheel selection operator as well as a crossover and mutation operator for trees. The user had to listen to the sounds generated using these trees and interactively evaluate them. Takala et al used the term Genetic Algorithm to describe their system but the term Genetic Programming (GP) seems more appropriate given the algorithmic process and genetic operators they defined. They implemented directly their system in C++ without using any generic sound synthesis library. While the authors reported that they successfully used their system to produce an entire class of bee-like sounds (ranging from mosquitos to chain saws), they did not present any proper evaluation.

More recently, Garcia [9] designed a Genetic Programming (GP) system for automatic generation of sound synthesis techniques. He encodes the instructions for building sound synthesis topology graphs in the GP expression trees. Production rules are defined to ensure that any valid expression tree will produce a valid topology. The nodes of the topologies are chosen from six different types of basis synthesizer components (oscillator, addition/multiplication operators, filters, etc). These topologies have two inputs parameters that are optimized by a canonical GA. He defined the fitness function as the least-squared error of the short-time spectra where the frequencies that are not heard by the average listener are ignored. The system was implemented directly in C++ and Matlab without using any generic sound synthesis library. Garcia tested his system with a FM woodwind instrument and verified that it could generate an expression tree with close similarity to the target FM equation used to design the instrument. However, there were higher energies at high frequencies. He also got modest results evolving a synthesizer for a sampled piano tone that sounded like a *string hit by a hammer*.

3. PURE DATA

Pure Data (Pd) [25] is a graphical programming environment for audio, video, and image processing popular among musicians, artists and sound designers. Pd programs (called *patches*) are made by arranging and connecting boxes within a visual *canvas*. There are four types of boxes: object, message, GUI (number boxes, toggles, sliders, etc.) and comment. Objects act as self-contained programs, each of which may receive inputs (through one or more visual *inlets*), generate outputs (through visual *outlets*), or both. Inputs and outputs can be of two types: *audio signal* or *control message* (trigger, integer or float). A connection can link the output of a given box and the input of another one (see Figure 1c). A connection can represent a control signal (represented in Pure Data by a thin link) or an audio signal (thick link). A control (audio) signal can only be produced by a control (audio) object. The presence of the sign $\sim$ in an object box indicates that it is an audio object (see Figure 1c). Audio (control) signals can only be transmitted between specifically defined audio (control) outlets and inlets.

We represent the synthesizers in Pd as directed acyclic graphs. Hundreds of object types are available in Pd. In order to limit the size of the search space, we select only objects that are the common building blocks of sound synthesizers in Pd. Table 1 shows a selection of Pd objects available to our search algorithm. This list includes arithmetic, trigonometric operations for controls and signals but also oscillators and filters. These objects are commonly used by Pd programmers [25] to implement major sound synthesis techniques such as additive, subtractive, FM, AM or PM synthesis. Each of these Pd objects has one to three inlets and only one outlet. Each inlet (outlet) is connected to one or more outlets (inlets). Figure 1c shows an example of Pd graphs evolved by our algorithm. In principle, any synthesizer can be represented as a Pd patch [25].

4. MIXED TYPED CARTESIAN GENETIC PROGRAMMING

Cartesian Genetic Programming (CGP) [21] is a form of Genetic Programming that encodes a directed acyclic graph representation of a computer program instead of a tree. The genotype, a string of integers, determines the functions of nodes in the graph, the connections between nodes, the connections to inputs and the locations in the graph where outputs are taken from. Using a graph representation is very flexible as many computational structures can be represented as a graph. Examples of this are artificial neural networks, electronic circuits or mathematical equations [21]. In this work, we represents Pd patches which are directed acyclic graphs in their general form.

This graph representation presents advantages over the classical tree-based GP representation. In a CGP graph, the output of a given node can be connected to one or more inputs of other nodes. It is not the case with a tree representation where a node output can only be connected to one other node. In addition, nodes of the graph are not necessary connected indirectly or directly to the output node of the program (e.g. second node see Figure 1b). As a result, nodes in the representation can have no effect on the output, a feature known in CGP as *neutrality*. This has been shown to be useful to the evolutionary process [21].

Mixed Typed Cartesian Genetic Programming (MT-CGP) is an extension of CGP [10] that handles multiple data types. Contrary to CGP, MT-CGP genotypes represent a linear string of nodes, without loss of generality. That is to say, only one row of nodes is used in contrast to CGP which can have a rectangular grid of nodes. The evolutionary algorithm usually used with MT-CGP is a $1 + 4$ Evolutionary Strategy [10].

To generate programs with multiple data types, MT-CGP uses an internal data type that can be cast to different data types. In MT-CGP, the functions inspect the types of the input values being passed to it, and determine the most suitable operation to perform. This in turn determines the output type of the function. In our implementation, we represent the functions as Pd objects and consider the two internal data types used in Pd: *signal* (for audio signal) and

control (for float or integer messages). The functions inspect the types of the input values being passed to them, and, given a set of predefined rules (see Table 1), determine the most suitable Pd object type to implement. For example, consider a function with a type gene value equals to 0. This function takes two inputs and returns one output. There are 3 input cases that need to be considered: two controls, two signals and a signal and a control. The sum of two control is performed by the + object, the signal addition and sum of the signal and a control by the $+\sim$ object. In Figure 1a, the node 3 has a type gene value equals to 0. It is interpreted as a $+\sim$ Pd object (see Figure 1c) because its 2 inputs are respectively of type *signal* and *control*. This node has then an output of type *signal* (thick connection).

Figure 1a shows an example of a CGP chromosome (genotype) corresponding to the CGP graph (phenotype) in Figure 1b and the Pd patch in Figure 1c. In this first implementation, the first 3 genes are not interpreted as we enforce Pd patch inputs of type *control*. In future iterations of this system, we plan to use these genes to code for the type of the Pd patch inputs. A gene would then code for either a *control* input (type 0) or for a*signal* input (type 1). More inputs could be added by simply appending new genes at the beginning of the chromosome.

The following genes are grouped by 5 in genesets (between brackets). Each of these genesets represents a node of the MT-CGP graph and are numbered from left to right starting from 0. Each MT-CGP node represents a Pd object and is encoded with the five genes (four integers and one double) of the corresponding geneset. The first gene of a geneset specifies the Pd object type that the node represents. The next three genes are used to specify the nodes from which the Pd object obtains its inputs. These connection addresses are defined relative to the current node and specify how many nodes backwards in the genotype/graph to connect. If a relative address extends beyond the extent of the genome, a new connection address is computed by taking the relative address modulo 3 in order to get the address of one of the 3 input nodes. We consider only 3 genes coding for connections because in our system, the Pd objects available to the search algorithm have an arity of maximum 3 (see Table 1).

The final gene for each node is a floating point number that can be interpreted as either a function parameter, or used to generate values within the program. In our case, this extra parameter is used as constant for arithmetic operations (see gene value 9 in Table 1) and also to handle special cases to determine the most suitable Pd object type to use. For example, for the gene value 6 that represents the low pass filter (*lop* $\sim$), if there is no signal input, a new gene value is computed by taking the extra parameter modulo 4 in order to get a arithmetic object (gene value from 0 to 3) compatible with these inputs.

Table 2 gives the range of each gene types. The extra parameter can take integer values between 0 and 22500. 22500 is the Nyquist frequency for a sampling rate of 44.1 kHz and also the upper bound of the human hearing frequency range.

The decoded CGP graph (Figure 1b) is then embedded in the engine Pd patch (Figure 1c). Nodes that are not connected directly or indirectly to the output in the CGP graph are are not visible in the Pd patch as they have no effect on the sound produced.

Gene value	**Input 1**	**Input 2**	**Input 3**	**Extra param.**	**Pd Object**	**Description**	**Output type**
0	**c**	**c**	-	-	+	Addition	c
0	**s**	**c**	-	-	+∼	Addition	s
0	**c**	**s**	-	-	+∼	Addition	s
0	**s**	**s**	-	-	+∼	Addition	s
...							
3	**c**	**c**	-	-	/	Division	c
3	**s**	**c**	-	-	/∼	Division	s
3	**c**	**s**	-	-	/∼	Division	s
3	**s**	**s**	-	-	/∼	Division	s
4	**s**	**c**	-	-	osc∼	Oscillator	s
4	**c**	**c**	-	-	osc∼	Oscillator	c
...							
5	**s**	**c**	-	-	phasor∼	Sawtooth oscillator	s
5	**c**	**c**	-	-	phasor∼	Sawtooth oscillator	c
...							
6	**s**	**c**	-	-	lop∼	Low pass filter	s
6	**c**	**s**	-	-	lop∼	Low pass filter	s
6	**s**	s	s	**v**	lop∼	Low pass filter	s
6	**c**	**c**	v	v	s.c.	Arithmetic function	c
...							
9	**s**	-	-	**v**	+∼	Add a constant	s
9	**c**	-	-	**v**	+	Add a constant	c
...							

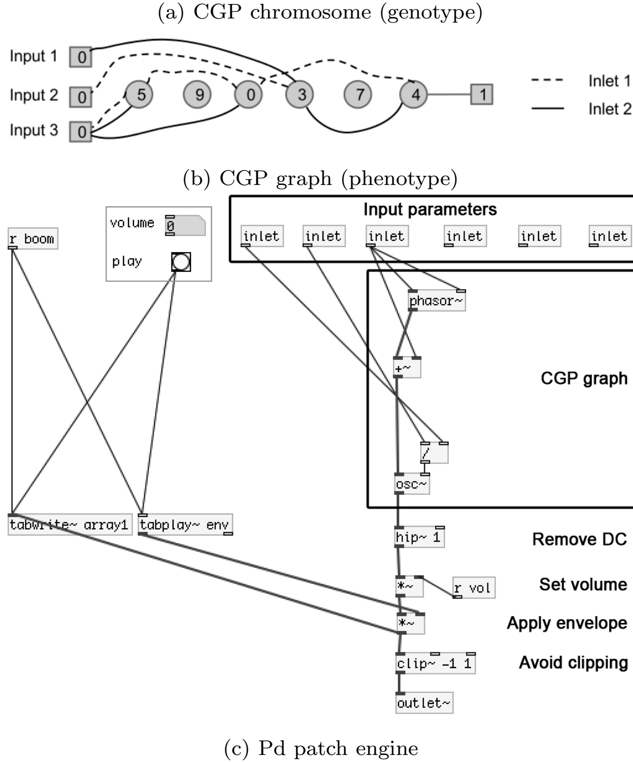
Table 1: Rules that determine the type of the Pd object that each node of the CGP graph represents given its respective gene value and its input types. s is for signal type, c for control type, v for the extra parameter and - for unused parameter. The inputs of the nodes that are considered for connection for the corresponding Pd object are set in bold. s.c. is a special case where the gene value is replaced by the value of the extra parameter modulo 4. The rule is then reapplied using the result as the new gene value.

4.1 Evolutionary algorithm and parameters

MT-CGP is used with a simple Evolutionary Strategy $1 + n$ with $n = 4$. In this setting, the best performing individual is selected as the parent. However, if the best performance is achieved by more than a single individual the ES chooses the one with the shortest program (i.e. the individual with the fewest connected nodes). When this choice is not unique, the newest individual is selected. Choosing the shortest program pushes evolution to find more compact programs. The preference to newer individuals has been shown to help MT-CGP find better solutions [10].

In keeping with the $1+n$ evolutionary strategy, MT-CGP does not use crossover. This *mutation-only* approach is typical in MT-CGP and it appears that a high mutation rate (10%) performs best [10]. The mutation rate is defined as the probability of each gene to be modified when generating an offspring. MT-CGP requires few parameters compared to other Evolutionary Algorithms [10], even in a distributed system.

$$[0, 0, 0, [5, 1, 4, 6, 295], [9, 2, 10, 5, 14773], [0, 2, 9, 7, 14300], [3, 5, 6, 1, 9520], [7, 9, 3, 3, 4108], [4, 3, 2, 4, 15775], [1]$$



(a) CGP chromosome (genotype)

(b) CGP graph (phenotype)

(c) Pd patch engine

Figure 1: Example of CGP chromosome decoding

Table 2 lists the parameters of our MT-CGP algorithm. These values have not been optimized, and therefore it may be possible to improve the performances of our system by conducting a parameter sensitivity analysis.

4.2 Similarity measure and fitness function

In this implementation, we consider the Mel-frequency Cepstral Coefficients (MFCC) to measure the similarity between the target sound and the sounds generated by the evolved patches. This sound feature has been initially developed for the task of speech recognition [20]. It also has been extensively used in Music Information Retrieval (MIR) systems [3, 4] and in automatic synthesizer calibration systems [27]. Yee-King and Roth [27] conducted a study to compare the performances of four parametric optimization techniques for automatic synthesizer calibration. The results of their study showed that the GA, with the MFCC fitness function, outperformed all the other optimization techniques in every experiments.

In the pre-processing steps of our algorithm, we extract the MFCC coefficients from the target sound on each time window. The same extraction is done on each candidate sound generated by the evolved Pd patches. Then, we calculate the Euclidean distances between the MFCC vector on each frame, sum them and finally divide this value by the number of frames.

Parameter (CGP)	**Value**
Max. number of generations	5000
Gene mutation rate	10%
Genotype length	50 genes
Number of inputs	3
Number of outputs	1
Type gene range	$[0, 14]$
Object input connections range	$[1, 50]$
Extra parameters range	$[0, 22050]$
Parameter (GA)	**Value**
Population size	150
Mutation probability	10%
Bitflip probability	10%
Crossover probability	80%
Tournament selection size	2
Elitism	3%
Bit-string size	$15 \times 3 = 45$

Table 2: CGP and GA parameters

Equation (1) shows how we compute this distance between the sound t and c.

$$d_{MFCC}(t, c) = \frac{\sum_{i=1}^{N_w} \sqrt{\sum_{j=1}^{N_c} (t_{i,j} - c_{i,j})^2}}{N_w} \tag{1}$$

The sampling rate is 44100 Hz and we set a window analysis size of 1024 samples (23 ms) and an overlapping of 512 samples (11.5 ms). N_w is the total number of windows. N_c is the number of MFCC coefficients that is set to 13. $t_{i,j}$ (resp. $c_{i,j}$) is the jth MFCC coefficient on the ith window for the sound t (resp. c).

The fitness function is then obtained by Equation (2) and gives a measure of similarity ranging from 0 (not similar) to 1 (perfect match).

$$fit = \frac{1}{1 + d_{MFCC}} \tag{2}$$

The goal of our optimization algorithm is then to find the synthesizer (Pd patch) and set of inputs parameters that maximize this fitness function.

4.3 Evaluation in Coevolutionary MT-CGP

In GP, CGP or MT-CGP, the fitness of a given generated program is, most of the time, evaluated given a dataset of inputs [10, 21]. For example, Harding and al. [10] used MT-CGP to generate programs that classify clinical data into two classes: cancer and non-cancer. They used the well known Wisconsin breast cancer dataset as dataset of inputs to evaluate theirs MT-CGP programs.

In our case, it is not possible to generate a generic dataset of inputs that would work for every synthesizers generated by MT-CGP. For a given synthesizer (MT-CGP program), it is not possible to know in advance what would be good input parameters to approximate the target sound. To deal with this issue, we coevolve a population of input parameters using a GA in parallel to the CGP population [15, 24]. In our system, the 3 input parameters of the Pd patches are integers ranging from 0 to 32767. These 3 integers are encoded as 15 bits bitstrings. Any floats or integers can then be obtained from a given input parameter using arithmetic or trigonometric Pd objects (see Table 1). Each CGP indi-

	Fitness (see Eq (2))				Nb. Pd Objects				
Contrived sound	**max**	**min**	**std**	**mean**	**max**	**min**	**std**	**mean**	**target**
additive	0.1572	0.1475	0.0055	0.1509	14	6	4.0415	10.333	20
FM	0.2467	0.1967	0.0273	0.2154	10	5	2.6458	8	7
phase	0.2171	0.1747	0.0225	0.1915	8	7	0.5774	7.3333	8
ring	0.2718	0.2407	0.0220	0.2563	14	10	2.8284	12	20
Recorded sound	**max**	**min**	**std**	**mean**	**max**	**min**	**std**	**mean**	**target**
op1	0.4265	0.4023	0.0580	0.4132	10	8	0.6534	9	X
transient	0.4126	0.4114	0.0001	0.4120	6	5	0.4171	5.3	X
clarinet	0.4212	0.4154	0.0041	0.4183	12	7	3.5355	9.5	X
cello	0.3977	0.3518	0.0325	0.3748	10	7	2.1213	8.5	X

Table 3: Experiment results

vidual is then evaluated using the inputs parameters taken from the GA population. The best fitness score obtained over the GA population is assigned to the CGP individual.

For each generation, once the 5 CGP individuals have been evaluated, each GA individual has been evaluated 5 times and consequently has 5 distinct fitness scores. The best score among these 5 fitness scores becomes the unique fitness score of the GA individual. This way, GA individuals, scoring high with CGP individuals that are not selected in the next generation, are more likely to be kept in the GA population. With this mechanism, we maintain diversity in the GA population (input parameters) and avoid to bias this population toward a specific CGP individual (synthesizer).

The GA population is then varied using a bitflip mutation and a 2-point crossover operator implementing an diversity preservation method (incest prevention strategy [26]). Table 2 lists the parameters used in the GA.

A neutral mutation is a mutation that does not change the CGP phenotype [21]. In order not to perform unnecessary evaluations, when neutral mutations is detected, the new CGP offspring is not evaluated but get the same fitness score than its parent. This offspring also replace its parents to encourage this kind of mutation that can lead to shorter graphs. Likewise, when the output of the last Pd object is of type control for a given CGP individual, the Pd patch does not produce any sound. The CGP individual is then not evaluated and it receives the minimal fitness score (0).

4.4 Pd patch design

Any sound synthesizer offers the functionality of changing the volume. In order to limit the complexity of the search, we normalize the sounds before evaluating them and provide a volume parameter on the Pd patch GUI for the user to adjust.

The difficulty in sound matching is mainly getting the right timbre and pitch [12]. The temporal envelope can easily be obtained using, for example, an envelope follower. Searching the temporal envelope in addition to the other parameters has shown to increase significantly the complexity of the problem [16]. Moreover, in most sound synthesizers, the temporal envelope is usually adjusted separately by the user. For example, the user may want a longer sustain and a shorter attack for the sound.

In our implementation, the temporal envelope is not optimized during the evolution. In the pre-processing steps of our algorithm, we extract the target envelope using an envelope follower. This envelope is then applied to the sound at the output of the CGP graph (see Figure 1c). One advantage of this approach is that the user can easily replace this envelope by an envelope of her choice: for instance, a *vline* $\sim$ object generating an ADSR envelope.

4.5 Implementation

The evolved CGP graph is embedded in a Pd patch consisting of two canvas. The first canvas is the canvas that the user sees when opening the patch. This canvas is common to every patches evolved and acts as GUI. On this canvas, the user can change the input parameters and the volume. A bang object has to be triggered to set the input parameters to the values optimized by our search algorithm. Another bang object triggers the generation of the sound. This sound can be listened and its waveform visualized on a graph.

The second patch called *engine* embeds the CGP graph evolved by our algorithm (see Figure 1c). The outlet of the last object of this graph is connected to a high-pass filter (*hip* $\sim$) to remove the DC component of the generated sound. It is then multiply by the envelope extracted from the target sound. This envelope is scaled by the *volume* parameter that can be adjusted on the GUI.

The implementation of the GA and MT-CGP uses the DEAP Python framework [6]. The MFCC coefficients are extracted using the Python wrapper for Yaafe [18]. The rules to determine the Pd object type given its inputs (see Table 1) are applied using the Python knowledge-based inference engine Pyke [7]. To speed up the execution of our search algorithm, we use the Bugaboo cluster that is part of Compute Canada/Westgrid [32]. We distribute the evaluation of the GA population for each CGP individual over 50 nodes. Pd is embedded in our algorithm using the Python wrappers offered by libpd [2]. CGP graphs from DEAP are converted into Pd file format that can be directly loaded with libpd from memory and processed to get the resulting sound. Using libpd makes possible to avoid parallel hard drive access. Instead of exporting the sound as a file, libpd directly provides Yaafe with the corresponding array of samples ready to be analyzed. This advantage can seem minor but it is actually significant in a distributed context where parallel hard drive access has to be avoided.

5. EXPERIMENTS

To demonstrate the validity of our approach, we have tested our algorithm with two types of target sounds: contrived sounds [23] and recorded sounds. The first category of sounds are sounds generated by Pd Patches. We generated four target sounds using additive synthesis, FM synthesis,

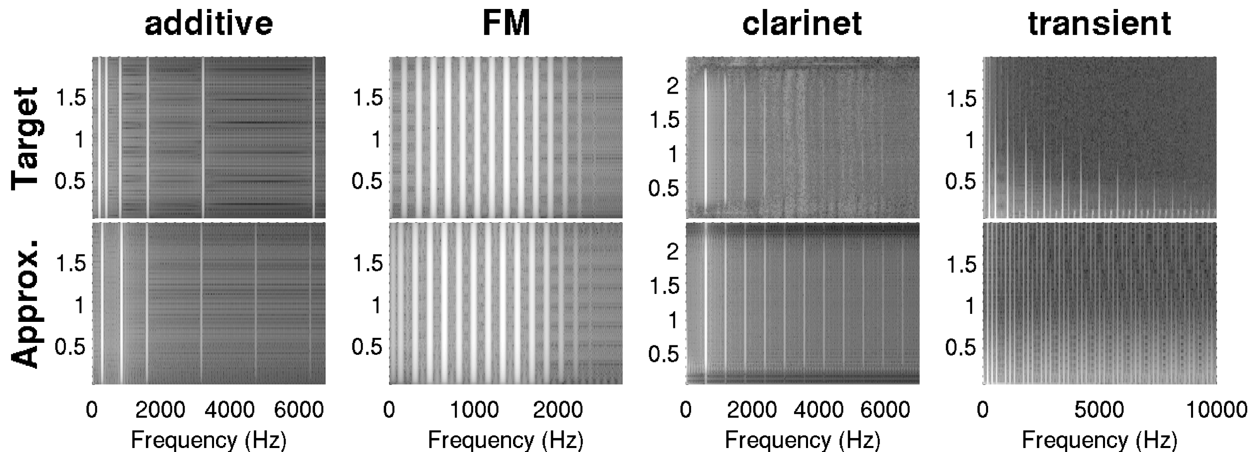


Figure 2: Spectra comparisons between approximated and target sounds

phase synthesis and ring modulation synthesis. In this experiment, we know that a solution exists in the Pd patches space that replicates perfectly this sound. We want to inquire whether our algorithm is able to reverse engineer the Pd patch or discover a novel way to approximate this sound.

The second category of sounds are recorded sounds that could be instrumental, natural or even synthesized by other means than Pd. We select four sounds: a clarinet sound, a cello sound, a transient sound and a sound synthesized by a commercial synthesizer (the OP-1). In this experiment, we want to see how well the synthesizers produced approximate these sounds, but also investigate what kind of program is found.

Table 3 gives the maximum, minimum, standard deviation and mean across 5 repetitions for each target sound. For the contrived sounds, one can notice that our system generates Pd patches that are more compact than the ones used to initially produce the target sounds. For a given target sound, the performance of our system is consistent across experimental repetitions ($std < 0.05$).

We report in the rest of this section the results of the most successful runs of our algorithm. Figure 2 compares the target spectra and their approximations by our algorithm for 2 contrived sounds (additive and FM) and 2 recorded sounds (clarinet and transient). The evolved Pd patches and sounds are available for download online [5].

Our system was unable to perfectly replicate the target sound in any of the experiments but it did approximate it using interesting synthesis methods. As the Pd patches are meant to be played with, it is relevant to look at the effect of changing an input parameter on the output sound. For example, in a Pd patch evolved for the flute sound, one input parameter was directly mapped to the pitch of the sound. It is then easy to generate another flute sound with a different pitch using this parameter.

For the additive target sound, the evolved synthesizer generates a spectrum that matches four of the five harmonics. When looking at the structure of the evolved patch, we can see that it is using additive synthesis. Oscillator objects ($osc \sim$) are added together using an addition object ($+ \sim$). Some patches even use Phasor objects ($phasor \sim$) (that generate sawtooth signals whose spectrum contains even and odd harmonics of the input frequency) instead of oscillator objects. Coupled with a low pass filter ($lop \sim$), it makes possible to generate spectra with a limited number of harmonics in a more compact way than adding together as many oscillators as there are harmonics.

For the FM target sound, the evolved synthesizer generates a spectrum very similar to a FM spectrum with a fundamental frequency and side bands frequencies of decreasing amplitudes. Looking at the structure of the evolved patch, it uses a combination of FM synthesis and ring modulation synthesis. Oscillator, Phasor and cosine waveshaper ($cos \sim$ objects output the cosine of two π times its signal input) objects are combined to approximate the target spectrum. The same observations can be made for the experiments using target sounds from ring and phase modulation.

We experimented with different recorded sounds. One of the most successful was the clarinet sound. Our system was able to find a very compact way of re-synthesizing this sound. It uses a Phasor centered on the fundamental frequency of the target clarinet sound and filters the high frequencies with a low pass filter. The result is a very suitable approximation of the target sound. As with the flute sound, the pitch of the sound was one of the input parameters evolved.

Our system was also to approximate sounds synthesized by other means than Pd. For example, for a sound generated by the OP-1, the evolved patch is able to match the frequencies of every harmonics but not their correct amplitudes. It is also able to match a transient sound even if there are some unwanted harmonics in the high frequency range. However, the results were less successful for sounds with time-varying spectrum such as piano sounds, voice sounds or synthesized sounds involving LFO.

6. DISCUSSION

Compared to Garcia's system [9], our system presents some advantages. First, the synthesizer structures are directly encoded in the CGP graphs while, in Garcia's system, the GP trees had to be mapped to the synthesizer structures using production rules. Second, acyclic directed CGP graphs are also more expressive representations than the GP trees. For example, in a CGP graph, an output of a given node can be connected to one or more inputs of other nodes. It

is not the case with Garcia's representation where a node output can only be connected to one other node.

In Garcia's system, the input parameters of each synthesizer functions had to be optimized by a GA. Computational resources were wasted when trying to optimized input parameters for unpromising synthesizer functions. Co-evolving a GA population of input parameters in parallel of the CGP population does a better use of the computational resources. Synthesizer functions and input parameters are evolved in parallel. The potential of promising synthesizer functions are unveiled using fitter and fitter input parameters while maintaining diversity in the GA population to discover novel functions. This approach also makes possible to better distribute the input parameters search between the GA and CGP. Arithmetic operations and constants (using the extra parameter) are available to the CGP algorithm (see Table 1). The search for the best inputs parameters is then made in cooperation between the GA that adjusts the input parameters and the CGP that changes the way these parameters are used in the synthesizer function. For example, given an input parameter with value 4, a target value 8 can be obtained either by, directly modifying bits in the GA bitstring that encodes the value, or by applying a suitable arithmetic operation in the CGP graph (multiply by the constant 2, add 4, etc.) or both. This balance between GA and CGP in the input parameter search can be adjusted by modifying the number of input parameters. For example, if we reduce the number of input parameters from 3 to 2, more complexity would be required in the CGP graph to regenerate these 3 initial parameters internally using only 2 input parameters.

Another advantage of our system is the representation of the synthesizers as Pd patches [25]. Pd counts a large user base and has a popular commercial version (MAX/MSP) commonly used in professional audio production. The Pd patches evolved by our system can easily be modified and reused in other contexts or applications. Garcia took a different approach and developed his own library to represent the synthesizers. This library was designed especially to work with his GP system and was not thought to be easily modified or reused in other applications. Moreover, more synthesizer's components are available with Pd than in Garcia's system making possible to generate more complex synthesizers and match more complex sounds.

Finally, Garcia's system required 20 to 200 hours per experiments. As our algorithm is distributed on a cluster, we are able to limit the time to perform an experiment to 5 hours. For each experiment, we evolve 5000 generations that represents at most 3.75×10^6 evaluations ($5000 \times 5 \times 150$). This number could be highly reduced by fine-tuning the GA and CGP parameters (see Table 2). We also constrained the types of Pd objects accessible to our algorithm to the basic building blocks of synthesizers. Adding more complex objects, such as more sophisticated filters or envelopes, could lead to novel function discoveries and a better and faster convergence.

7. CONCLUSIONS AND FUTURE WORKS

We propose a new approach to automate the design process of sound synthesizers using Coevolutionary Mixed-typed Cartesian Programming. The approach is capable of evolving synthesizer structures for generating sounds similar to the target sound (according to the fitness function based on MFCC coefficients). We represent the synthesizer structures using acyclic directed graphs and optimize the synthesizer's input parameters in parallel using a GA. We run a small number of experiments but their results are promising and show the potential of using Coevolutionary Mixed-typed Cartesian Programming as an approach to automate the design of sound synthesizers.

Our proposed method involves quite many parameters (see Table 2). We plan to perform a parameter sensitivity analysis to investigate the robustness of our method and also determine how to set these parameters in a procedural fashion. We also plan on performing a more rigorous quantitative and qualitative comparison with previous similar systems [9, 30] using, for example, a same benchmark of target sounds and reporting on their respective performance.

In this work, we limited our search to only one target sound. We plan to extend our system to search the synthesizer space for synthesizers able to reproduce not only one sound but a set of sounds. For example, one could search for a synthesizer able to produce clarinet sounds but also oboe sounds just by changing the input parameters. Considering sound as a multi-dimensional object has shown to be helpful to better explore the sound space [16, 19]. For future work, we plan to take into account more sound features (psycho-acoustic, spectral or temporal) instead of only MFCC coefficients.

8. ACKNOWLEDGMENTS

This research was funded by a grant from the Canada Council for the Arts, and the Natural Sciences and Engineering Research Council of Canada.

References

[1] Bozkurt, B. and Yüksel, K. Parallel evolutionary optimization of digital sound synthesis parameters. In *Proceedings of the Conference on Applications of Evolutionary Computation* (2011), pp. 194–203.

[2] Brinkmann, P., Kirn, P., Lawler, R., McCormick, C., Roth, M., and Steiner, H. Embedding pure data with libpd. In *Proceedings of the Pure Data Convention* (2011).

[3] Brown, J. C., Houix, O., and McAdams, S. Feature dependence in the automatic identification of musical woodwind instruments. *The Journal of the Acoustical Society of America 109* (2001), 1064–1072.

[4] Casey, M., Veltkamp, R., Goto, M., Leman, M., Rhodes, C., and Slaney, M. Content-based music information retrieval: Current directions and future challenges. *Proceedings of the IEEE 96*, 4 (2008), 668–696.

[5] Experiment results for contrived and recorded sounds. `http://metacreation.net/mmacret/GECCO2014/` [Last accessed: April 2014].

[6] Fortin F., De Rainville F., Gardner M., Parizeau M. and Gagné C. DEAP: Evolutionary algorithms made easy. *Journal of Machine Learning Research 13* (2012), 2171–2175.

[7] Frederiksen, B. Applying expert system technology to code reuse with pyke. In *Proceedings of the conference on the Python Conference (PyCon)* (2008).

[8] Fujinaga, I. and Vantomme, J. Genetic algorithms as a method for granular synthesis regulation. In *Proceedings of the International Computer Music Conference* (1994), pp. 138–146.

[9] Garcia, R. Automating the design of sound synthesis techniques using evolutionary methods. In *Proceedings of the conference on Digital Audio Effects, Limerick, Ireland* (2001), Citeseer, pp. 1–8.

[10] Harding, S., Graziano, V., Leitner, J., and Schmidhuber, J. MT-CGP: Mixed Typed Cartesian Genetic Programming. In *Proceedings of the international conference on Genetic and Evolutionary Computation Conference* (2012), ACM, pp. 751–758.

[11] Heise, S., Hlatky, M. and Loviscach, J. Automatic cloning of recorded sounds by software synthesizers. In *Proceedings of the Audio Engineering Society Convention 127* (2009).

[12] Horner, A. Evolution in digital audio technology. In *Evolutionary Computer Music*. Springer, 2007, pp. 52–78.

[13] Horner, A. and Beauchamp, J. Piecewise-linear approximation of additive synthesis envelopes: a comparison of various methods. *Computer Music Journal 20*, 2 (1996), 72–95.

[14] Horner A. and Beauchamp J. and Haken L. Machine tongues XVI: Genetic algorithms and their application to FM matching synthesis. *Computer Music Journal 17*, 4 (1993), 17–29.

[15] Hrbacek, R.and Sikulova, M. Coevolutionary Cartesian Genetic Programming in FPGA. In *Advances in Artificial Life, ECAL* (2013), vol. 12, pp. 431–438.

[16] Macret, M., and Pasquier, P. Automatic Tuning of the OP-1 Synthesizer Using a Multi-objective Genetic Algorithm. In *Proceedings of the Sound and Music Computing conference* (Stockholm, Sweeden, 2013), pp. 614–621.

[17] Macret, M., Pasquier, P. and Smyth, T. Automatic Calibration of Modified FM Synthesis to Harmonic Sounds using Genetic Algorithms. In *Proceedings of the Sound and Music Computing conference* (2012), pp. 387–394.

[18] Mathieu, B., Essid, S. and Fillon, T., Prado, J. and Richard, G. YAAFE, an easy to use and efficient audio feature extraction software. In *Proceedings of the International Society for Music Information Retrieval* (2010).

[19] McDermott, J., O'Neill, M., and Griffith, N. EC control of sound synthesis. *Evolutionary Computation Journal 18*, 2 (2010), 277–303.

[20] Mermelstein, P. Distance measures for speech recognition, psychological and instrumental. *Pattern recognition and artificial intelligence 116* (1976), 374–388.

[21] Miller, J., and Thomson, P. Cartesian genetic programming. In *Genetic Programming*. Springer, 2000, pp. 121–132.

[22] Mitchell, T. Automated evolutionary synthesis matching. *Journal on Soft Computing* (2012), 1–14.

[23] Mitchell, T. and Creasey, D. Evolutionary sound matching: A test methodology and comparative study. In *International Conference on Machine Learning and Applications* (2007), pp. 229–234.

[24] Popovici, E., Bucci, A., Wiegand, R. P., and De Jong, E. Coevolutionary principles. In *Handbook of Natural Computing*. Springer, 2012, pp. 987–1033.

[25] Puckette, M. Pure data: another integrated computer music environment. In *Proceedings of the Second Intercollege Computer Music Concerts* (1996), pp. 37–41. http://puredata.info/.

[26] Rocha, M. and Neves, J. Preventing premature convergence to local optima in genetic algorithms via random offspring generation. *Multiple Approaches to Intelligent Systems* (1999), 127–136.

[27] Roth, M. and Yee-King, M. A comparison of parametric optimization techniques for musical instrument tone matching. In *Proceedings of the Audio Engineering Society Convention* (2011), pp. 972–980.

[28] Schatter, G. and Züger, E. and Nitschke, C. A synaesthetic approach for a synthesizer interface based on genetic algorithms and fuzzy sets. In *Proceedings of the International Computer Music Conference* (2005), pp. 664–667.

[29] Serquera, J. and Miranda, E. Evolutionary sound synthesis: rendering spectrograms from cellular automata histograms. *Applications of Evolutionary Computation* (2010), 381–390.

[30] Takala, T., Hahn, J., Gritz, L., Geigel, J., and Lee, J. . Using physicallybased models and genetic algorithms for functional composition of sound signals, synchronized to animated motion. In *Proceedings of the International Computer Music Conference* (1993), pp. 180–185.

[31] Vuori, J. and Välimäki, V. Parameter estimation of non-linear physical models by simulated evolution-application to the flute model. In *Proceedings of the International Computer Music Conference* (1993), pp. 402–402.

[32] Westgrid/ComputeCanada: Bugaboo cluster. http://www.westgrid.ca/ [Last accessed: April 2014].

[33] Yee-King, M. and Roth, M. Synthbot: An unsupervised software synthesizer programmer. In *Proceedings of the International Conference Music Conference* (2008), pp. 1–6.

[34] Yoshimura, T., Tokuda, K., Masuko, T., Kobayashi T. and Kitamura T. Simultaneous modeling of spectrum, pitch and duration in HMM-based speech synthesis. In *Proceedings of the conference on Speech Communication and Technology* (1999), pp. 1315–1318.