Evolutionary Audio Synthesis

Etienne Cella

June 2018

Contents

1 Introduction

Disclaimer: i'm by no means an academic, and this document simply describes an experiment i've been working on in my free time, as part of a learning process. Any comment / criticism is more than welcome.

1.1 Context

I've had a strong interest in electronic music as a teenager, and have now found myself more and more interested by the field of Artificial Intelligence. This experiment gives me an opportunity to combine both these interests, as we'll attempt to build an Evolutionary Audio Synthesis System.

We do not *evolve* sounds directly. Rather, we evolve programs that will generate those sounds, that is **CSound** audio synthesis programs.

1.2 Tools

This experiment is written in Python, which we picked because of its elegance, the fast iterations it allows, and the excellent scientific computing libraries it comes with, namely **numpy** and **scipy**. For vizualisation, we use the **matplotlib** and **graphviz** libraries. We also use the **CSound** for audio synthesis, we elaborate on this in the next section.

1.3 Why CSound?

CSound can be thaught of as the assembly of audio synthesis. It is an extremely simple yet powerful language, and its longevity speaks for itself. When describing a signal processing graph, it doesn't get any simpler:

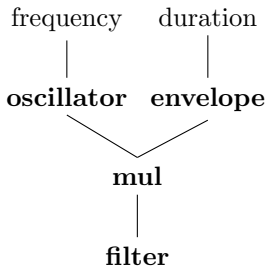


Figure 1: Signal Processing Graph

aenv	**envelope**	duration
aosc	**oscillator**	frequency
asig	**mul**	aosc, aenv
out	**filter**	asig

Figure 2: CSound implementation

2 Generating Programs

Generating programs in our case boils down to generating random trees by picking nodes that can be connected together.

2.1 Variable Types

When building the tree, we make sure that the input type of the parent node matches the output type of its child. In CSound, types are denoted by the prefixes used for variable names, for example, with `asomething`, the type is `a`, that is, an audio rate stream.

2.2 Node Types

As mentioned, we store programs as trees. These trees are made of nodes, who may hold **CSound opcodes** (see section 1.3) or numerical constants.

In these trees, we'll find three types of nodes, that we'll call **internal**, **terminal** and **constant** nodes. Let's quickly go over these terms:

- **internal node**, a node that holds an opcode whose inputs may be the output of other opcodes
- **terminal node**, a node that holds an opcode whose inputs, if any, are numerical constants
- **constants**, a node holding a numerical constant

let's recall figure 1, and vizualise these various node types:

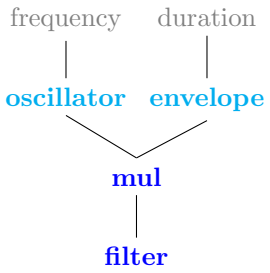


Figure 3: Node types, internal, terminal and constant

2.3 Opcode Signature

As mentioned in section 3.1, we have three types of nodes, internal, terminal and constant. Internal and terminal nodes hold CSound opcodes. We have retrieved from CSound's reference the description of a set of nodes that we want to support. Typically in CSound's reference, opcodes are described by their signature, such as:
`ares oscil xamp, xcps, ifn, iphs`
We augment these signature with information helping us to make better educated guesses when picking constants, such as:
`ares oscil xamp:[0,1], xcps:f, ifn:w, iphs:[0,1]`
Here we precise that **amp** (amplitude) and **phs** (phase) should be picked in the [0, 1] interval, **cps** (frequency) is a frequency and **fn** (function) is a wavetable

2.4 Opcodes Distribution

We support a set of CSound opcodes, some opcodes may appear multiple times with different signatures, think of it as overloads.

The distribution of nodes in the supported set does not reflect their usefulness. For instance, we may have 3 oscillators and 12 envelopes, making the system way more likely to pick an envelope than an oscillator. We would like the odds of picking an opcode to depend on its functionality and the the functionality of its parent opcode in the tree.

To do so, we associate tags to opcodes, reflecting their functionality, for example, wether it is an envelope, an oscillator, and so on. We then handcraft a matrix storing the probabilities of picking an opcode with a specific tag, given the tag of its parent.

When handcrafting the matrix, what we are doing is try to improve the quality of generated programs by specifying *rules*. For example, *avoid plugging an envelope to the audio output*, or *avoid connecting an envelope output to a reverb*. These rules convey part of the know-how of a sound designer (in a very, very primitive manner). There is a balance to be found between specifying enough rules for the system not to waste too much computation, while preserving enough freedom for the system to come up with surprising solutions.

3 Evolutionary Program Synthesis

3.1 Introduction

The goal of the system is to find programs who are good at generating sounds that are *close* to an audioclip selected before the genetic simulation starts. We'll call that audioclip the *target* sound.

Our system evolves programs. During most of this process, programs are stored as trees representing signal processing graphs. Only when generating audio from the programs to evaluate their fitness do we convert them to executable CSound code.

3.2 Initialization

We generate an initial population of random programs, constrained by the maximal depth a program may reach and the likelyhood of picking a terminal node (the higher this likelyhood, the less *bushy* the tree will be).

3.3 Reproduction

For a predetermined number of generations, we evaluate the fitness of the programs we have, and use the best ones to create the next generation. These offsprings are generated by several methods:

- **exact copy** of their parent
- **mutate constants**, we randomly tweak the numerical constants of the parent, the **lerp_factor** parameter lets us control by how much those constants may change

- **subtree mutation**, we pick a node in the parent and append a new random subtree at this node while preserving the overall depth of the tree
- **random generation**, some programs are created randomly, the same way we created the initial population

Figure 4: Parent program

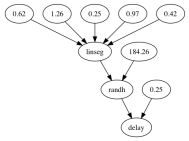


Figure 5: Child program after constant mutation, note that opcodes are preserved while numerical values have changed

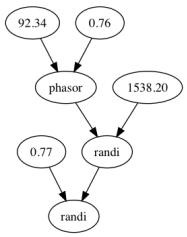


Figure 6: Parent program

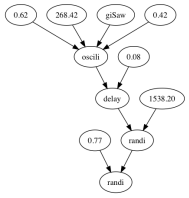


Figure 7: Child program after subtree mutation, note the replacement occured at the *phasor* opcode

3.4 Selection

3.4.1 Overview

To evaluate the fitness of programs, we take the following in account:

- **sound similarity**, that is, how close is the spectrum of the sound generated by the program to our *target* sound's spectrum.
- **nodes count**, how much nodes does the tree have. That way we can reward trees with lower nodes count, that is, simpler programs.

3.4.2 Sound Similarity

The main goal our system tries to achieve is to produce programs whose output is close to a given audio *target* file.

To evaluate what we'll call *sound similiarity*, we compare the spectrum of the generated audio file with the spectrum of the target file. We define sound similiarity **s** so that:

$$\mathbf{s} = \sum_{i=1}^{m} \sum_{j=1}^{n} (x_{i,j} - t_{i,j})^2$$

Each spectrum being an $m \times n$ matrix, whose axes represent time and frequency, x being the spectrum of the generated sound and t the spectrum of the target sound.

3.4.3 Preserving Diversity

By taking multiple factors in account when evaluating fitness, we can preserve diversity within our programs population. Using a single factor, we risk that any *innovation* quickly takes over the whole population, reducing its diversity and lowering the odds of coming up with good solutions by exploring multiple paths simultaneously.

4 Experimentation

4.1 Code Organization

Our code is organized in the following manner:

- **analysis**, sound analysis, spectrum computation and comparison
- **code_gen**, CSound code generation from program trees
- **csound_reference**, holds CSound opcodes descriptions
- **elements**, tree elements generation based on CSound reference data
- **experiment**, glue code between every other elements, responsible for running the overall experiment
- **genetic_operators**, genetic operators used to generate offsprings from trees
- **tree**, create and manipulate trees
- **util**, miscellaneous utility function
- **vizualisation**, vizualise sounds and program trees (using GraphViz for the latter)

There also are some test files that are not part of the system itself and are only here to support development.

4.2 Usage

To run an experiment, execute the file `experiment.py`, experiment parameters are exposed at the bottom of the file. These parameters have the following meaning:

- **file**, the audio file we'll use as a *target* sound
- **intern_op_set, term_op_set**, opcodes sets, simply returned by `read_op_set()`
- **num_generations**, number of generations the simulation will run for
- **init_population_size**, the size of the random population generated at the beginning of the simulation
- **selected_population_size**, the number of best performing individuals that will be selected from the current generation to create the next one
- **max_depth** the maximal depth of program trees
- **terminal_likelyhood**, the probability of picking a terminal opcode when generating a tree, the higher this probability the sparser the trees generated
- **lerp_factor**, when using constant mutation, how close do the new numerical values stay from their original values
- **complexity_factor**, when evaluating fitness, how much do we reward simpler trees, that is, trees who have a lower node count

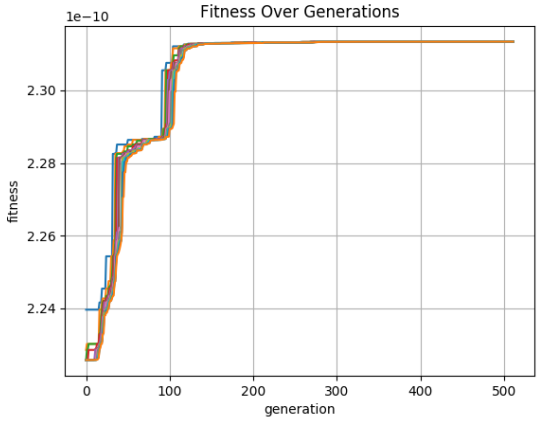
At the end of the experiment, all results will be stored in an `output` folder, including:

- generated CSound files for the best individual, `.orc` and `.sco` files.
- `graphviz` representation of the best individual
- audio rendering of the best individual
- plot of the best candidates fitness over generations
- GIF animation showing the best individuals' graph over generations

4.3 Results

We'll take a look at an experiment in which the target sound is a clap sampled from a Roland TR-808 drum machine.

We have plotted below the fitness of the best individuals over 512 generations.



We observe the curve tends to have a logarithmic shape, as the system makes discoveries and tries to improve on them, making smaller and smaller improvements. We notice that when a significant discovery is made (around generation 100 in the above plot), the *logarithmic pattern* restarts.

We vizualise below the best program after 512 generations:

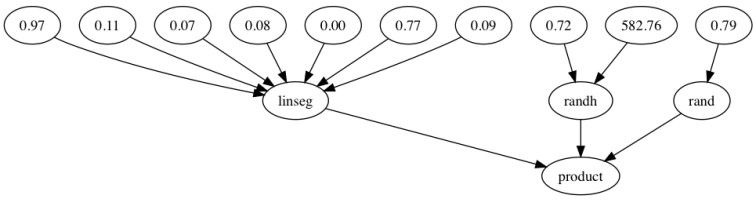


Figure 8: Best program after 512 generations

Let's compare the waveforms of the target sound and the best generated sound after 512 generations:

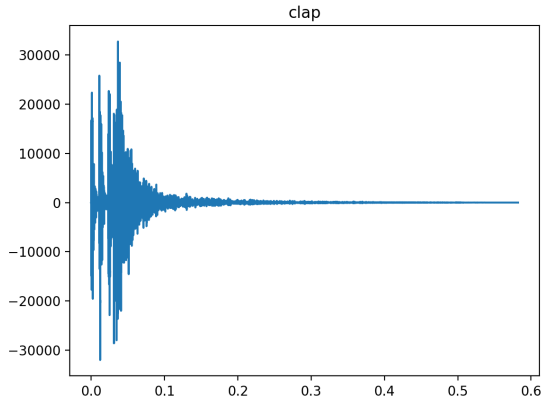


Figure 9: Target sound waveform (Roland TR-808 clap)

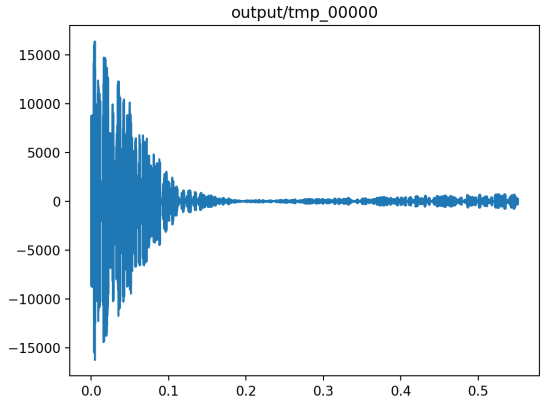


Figure 10: Best candidate after 512 generations

In this selected experiment, the system manages to produce a usable clap sound, close enough to the target sound, within a reasonable amount of generations (of limited size, a couple dozen individuals per generation).

Note that the system has successfully discovered a good strategy for synthesizing clap sounds: apply an envelope on noise generators.

4.4 Use Cases, Future Developments

With respect to obtained results and the limits of the computational resources at hand, this system is not aimed at generating audio programs generating sounds similar enough to the target sound to fool the human ear. We're in fact very

far from it. But it provides an interesting way to generate new, usable, audio programs, in a supervised way.

As far as future development is concerned, an obvious way to improve the system would be to support more opcodes, and make better educated guesses when building program trees.

We have mentioned in 2.4 that we use a handcrafted matrix to influence the tree building process. We could try to learn this matrix by parsing existing, human-made CSound programs. Or even other programs, say Pure Data or something else, as the tag system is relevant in any audio synthesis environment. We could also try a more radical approach, by updating the weight matrix as the experiment runs, rewarding *good* connections, that is, tag associations that tend to occur in better performing programs. Technically, this would introduce reinforcement learning to the experiment, besides the genetic learning we already have.