A Software Platform for Genetic Algorithms based Parameter Estimation on Digital Sound Synthesizers

Kamer Ali Yuksel
VPA Laboratory
Sabancı University
kamer@sabanciuniv.edu

Batuhan Bozkurt
MIAM
Istanbul Technical University
pyramind@gmail.com

Hamed Ketabdar
Quality and Usability Lab
T-Labs, TU Berlin
hamed.ketabdar@telekom.de

ABSTRACT

In this research, we introduce a general-purpose sound synthesis architecture for parameter estimation system, which is employing genetic algorithms (GA), to search the parameter space for different digital sound synthesizer topologies. Based on the architecture, we have implemented the software using the SuperCollider audio synthesis and programming environment and conducted several experiments. Primary consideration of the implementation was a modular and flexible structure of the framework that may open wide-range of opportunities for musicians and researchers in the field.

Categories and Subject Descriptors

H.5.5 [**Sound and Music Computing**]: Signal analysis, synthesis, and processing—*Methodologies and techniques*

Keywords

evolutionary computation, genetic algorithms, sound synthesis, parameter estimation, parallel computing

1. INTRODUCTION

Any attempt for sound analysis is also a form of endeavor for some sort of parameter estimation [2, pg 596]. The analysis task might be undertaken for obtaining the properties of some source sound that is to be resynthesized with different sound synthesis methods, or for observing those very qualities with the purpose of fitting them into a theoretical model. As an example, Roads [2, pg 596] points out that the Fourier Analysis can be considered as a parameter estimation method, because the results returned by such an analysis (namely frequency, amplitude and phase dissections for the analyzed signal), can be considered as parameters for a sine wave re-synthesis method that will approximate the source content veridically. Although it was originally conceived as a coding method for reducing the bandwidth of speech signals, the Phase Vocoder, being an example, uses Fourier Transform for analysis and re-synthesis at its heart, but the raw analysis data it produces is far more greater in size compared to the input data it is fed with [2, pg 549]. However, in our work we approach the problem of parameter estimation for effectively reducing the amount of data that is required to approximate a given sound with different synthesis methods, because we want to be able to control and alter various perceptual qualities of the resulting sounds from a higher level of abstraction, in an intuitive and interactive manner.

In this paper, we introduce our implementation of a modular genetic algorithm framework conceived inside the SuperCollider programming language. We describe how the framework can be utilized for doing parameter estimation work using user created arbitrary sound synthesizer topologies. The techniques we use for the task are built upon the previously referred research, and the process is also used as a test case for describing how and where the introduced framework sits inside the SuperCollider environment and hinting how and in what ways it can be useful. We also try to stimulate ideas for opportunities on creative usages of genetic algorithms, once their capabilities become available inside a highly connective real-time sound synthesis and algorithmic composition environment, especially when the products are available at interactive or near-interactive speeds to the user. To begin optimization, the user supplies a target sound file with desired attributes, a synthesizer definition, parameter names and defaults values of the parameters, which forms the initial population. After this initialization, the GA loop happens in generations for evolving towards better solutions of parameter sets. In each iteration, GA synthesize sounds for each set of parameter and compare the attributes of the output sound to the target sound by calculating the fitness of them. Then, multiple individuals are stochastically selected from the current population based on their fitnesses to breed a new generation. After selection, crossover and mutation operators are applied to the gene pool. This loop continues until satisfactory fitness level, which is set by the user, has been reached for the population. In addition to a target fitness value, the user the user can decide the fitness of the fittest member of the last generation by listening or can limit the maximum number of generations to be iterated.

2. MODULAR GA FRAMEWORK

Our parameter estimation system is built around this general purpose genetic algorithm framework and we will briefly introduce some of our key choices concerning its design. In order to fulfill the needs of creating a framework that

SAC'11 March 21-25, 2011, TaiChung, Taiwan.

is not bound to any specific task or approach using GA's, we needed to modularize the usual stages of a typical GA workflow. While the class itself declares and promotes default methods (which will be discussed briefly in a following section) for initialization, selection, crossover and mutation, methods for all these stages can also be supplied by the user as functions. We also took advantage of the interpreted and interactive nature of sclang. Each of the instance methods and variables of the running system can be queried and altered on demand at precise stages of the evolution. The looping of the "select, crossover, mutate" operators are not handled in a special method provided inside the framework so that the user is able to control the execution and alter the state of the instance(s) and the functions the operator methods call, whenever need arises. This flexibility, for example, allows implementation of increasingly demanding fitness functions (layered learning), or fine tuning of the bounds of mutation operators throughout evolution, among other standard or creative and novel alterations to the GA process. The functions for creating the initial gene pool, mutating chromosomes, and fitness evaluation are provided by the user at the creation time of an instance. The class however, includes built-in default methods for selection and crossover operators, both of which can also be dispatched to user supplied functions on demand. Building IGA systems around the framework is also possible. The default selection method we've included in the implementation is tournament selection. Tournament selection involves running several tournaments among a few individuals chosen at random from the population. After each tournament, the one with the best fitness (the winner) is selected for crossover. The user can override the default tournament size (2) for the selection operator. Tournament selection allows direct control of the selection pressure (by setting the tournament size) that can be tuned for different search domains and desired time constraints. Since the selection pressure is one of the main factors that affect the convergence time of GA search, this type of flexibility is desirable in music related domains where sacrifices might be chosen to be made over the optimality of the solution, in favor of obtaining more interactive search speeds. This selection method is also adequate for systems with noisy fitness functions. For the crossover operator, a multi-point crossover method is supplied where the number of split points determined proportionally to the crossover probability. This method, like the other operators, can dispatch the parent chromosomes to user supplied crossover functions if necessary.

3. PARAMETER ESTIMATION SYSTEM

The GAPmatch class provides a simple front-end for doing parameter estimation work inside the SC environment. Its functionality depends on facilities provided by the GAWorkbench framework we've introduced earlier. Arguments relating to logistic issues pertaining to GA's aside, the user supplies a sound file (target sound with desired attributes), a synthesizer definition (abstracted under the SynthDef class in sclang), the parameter names and a function that returns parameter values for individuals that will make up the initial population. SynthDefs are compiled by sclang, using a user generated function that defines the unit generator graph of the desired synthesizer. These functions describe how the unit generators composing the synthesizer are interconnected, what their inputs and outputs are, which arguments the synthesizers accept and how those arguments are utilized. The compilation of SynthDef is handled by sclang dynamically, and the resulting synthesizers can be sent to scservers via the OSC protocol, to be used immediately. Thus, it is also possible to build a system that wraps GAWorkbench functionality to encode synthesizer topologies as chromosomes, and evolve sound synthesis algorithms as Garcia [1] described in his research. However in our work, we are mainly concerned with user generated and supplied static synthesizer topologies, and GA search inside their parameter spaces for target sounds instead. Once the SynthDef is supplied to the system and the initial population of individuals providing the values for the arguments is created, the GA should try each set of arguments on the synthesizer, and compare the attributes of the resulting sound against the target sound that is provided by the user (i.e. calculate the fitness of the parameter sets for the given task). The same process should start over after selection, crossover and mutation operators are applied to the gene pool. This is a non-blocking loop working on scserver(s), and can be terminated anytime by the user operating the client (sclang) interactively. The user can listen to the fittest member of the last generation and can decide if the solution is fit enough, or can limit the maximum number of generations to be iterated, or set a target fitness value that will trigger the termination of the evolution. The initialization of the parameters and application of genetic operators are handled by the wrapped GAWorkbench class. The GAPmatch class is responsible for compilation of the SynthDef provided by the user, transmission of the compiled SynthDef to the supplied servers (local and networked), fitness evaluation of the parameters in the gene pool, and distribution of the computational load of fitness evaluation across the servers registered with the instance. The fitness evaluation stage in GA's should provide fitness scores for each member of the gene pool. Those scores will eventually influence the selection stage that will then steer the evolutionary process. Thus, our fitness function for the parameter estimation task should compare the attributes of the sound output by the synthesizer running with each set of parameters inside the gene pool, with attributes of the target sound we are interested in. A *fitness rating* that reveals the extent of similarity between two sounds is necessary. Synthesis operations are handled at server side in SC environment. As discussed earlier, scserver runs compiled unit generator graph functions, so we've implemented a unit generator that measures the analytical spectral distance of magnitudes between the complex spectrums of the source (synthesized) and target sounds. The default synthesizer unit that measures the MSE distance between two sounds can also be changed by the user, if need for using different analysis metrics arise. In our implementation, we've used the analytical distance metrics proposed by Garcia [1], but by disregarding the phase information, focusing only on the magnitudes.

4. REFERENCES

[1] R. Garcia. Growing Sound Synthesizers Using Evolutionary Methods. *European Conference in Artificial Life ECAL2001. Artificial Life Models for Musical Applications. University of Economics, Prague, Czech Republic.*, 2001.

[2] C. Roads and J. Strawn. *The Computer Music Tutorial.* MIT Press, Cambridge, Mass [u.a.], 4th edition, 1999.