Applications of genetic programming to digital audio synthesis

Chris Donahue
Advisors: Peter Stone, Russell Pinkston
Department of Computer Science
University of Texas at Austin

Abstract

The creation of digital audio synthesis algorithms has been a historically manual task. Tuning the numeric parameters of existing algorithms to produce a desired sound can take hours and still not produce the intended results. Electronic composers that choose to design their own digital instruments are typically trying to accomplish one of two tasks: mimicking an acoustic or electronic instrument that they have heard before or creating something entirely new. This thesis presents two applications of genetic programming to the domain of digital audio synthesis in an attempt to increase automation and accessibility of these processes. The first application is a command line tool which can automatically generate synthesis algorithms that mimic the timbre of an input sound file. The second application is a VST audio plugin that allows users to interact with genetic programming to search for pleasing or novel synthesis algorithms. The applications are designed to interface; synthesis algorithms discovered by the timbre mimicking application can be saved and loaded into the VST plugin to be played as MIDI instruments.
## Contents

1 Introduction  
  
2 The *Synthax* Language
  
  2.1 Symbolic Expression of Audio Synthesis Algorithms  
  2.2 Syntax  
  2.3 Rendering Audio From *Synthax* Code

3 Application 1: Automatic Timbre Mimicking
  
  3.1 Defining Timbre  
  3.2 Motivations  
  3.3 Related Work
    
    3.3.1 Early Investigations (1950-1970)  
    3.3.2 FM Synthesis (1970-1990)  
    3.3.3 Genetic Algorithms (1990-present)
  3.4 Method
    
    3.4.1 Initialization  
    3.4.2 Sub-optimization  
    3.4.3 Fitness Evaluation  
    3.4.4 Population Maintenance
  3.5 Implementation Details
  3.6 Experiments
    
    3.6.1 Human Study #1: Synthesized Piano Tone  
    3.6.2 Human Study #2: Audio From Known *Synthax* Program
  3.7 Results and Conclusions
    
    3.7.1 Human Evaluation of Fitness Functions  
    3.7.2 Synthesized Piano Tone  
    3.7.3 Audio from Known *Synthax* Program

4 Application 2: Interactive Evolution of Audio Synthesis Algorithms
  
  4.1 Motivation
  4.2 Method
  4.3 Implementation Details
  4.4 Experiment
  4.5 Conclusions

5 Future Work
  
  5.1 New Primitives
    
    5.1.1 Filters  
    5.1.2 Memory  
    5.1.3 Logic  
    5.1.4 Splines  
    5.1.5 Wave Table Generators
  5.2 General *Synthax* Enhancements
1 Introduction

Genetic programming is a type of genetic algorithm that operates on populations of symbolic expressions. Symbolic expressions (S-expressions) are highly adaptable structures that can represent mathematical expressions, computer programs and more depending on the set of domain-specific function operators and terminals provided (the primitive set).

The primary research contribution of this thesis is an investigation of applying genetic programming to timbre mimicking using a direct encoding of audio synthesis algorithms. Information from this investigation could also be found useful for applying genetic programming to problems with a heavy dependence on numerical parameters, using S-expressions for real-time generation of time-varying signals or arrays of data, introducing value range-aware primitives to genetic programming and interactive search in the space of digital synthesis timbres.

This thesis will begin by presenting the synthax audio programming language in Section 2, created for the purpose of directly applying genetic programming to the space of audio synthesis algorithms. Following this explanation, two concrete and interacting applications that utilize synthax through genetic programming will be presented in Sections 3 and 4. Finally, there will be a discussion of ideas for expanding the language and future applications in Section 5.

2 The Synthax Language

Synthax is a symbolic expression-based language containing domain-specific functions for audio signal generation and manipulation in addition to traditional genetic programming operators. Programs written in synthax are direct encodings of audio synthesis algorithms and can be repeatedly evaluated to fill 32-bit floating-point audio sample buffers. Synthax can be used to generate audio data but could also potentially be used to manipulate external audio data passed as input to terminal primitives. The name synthax is a portmanteau of the words “synthesizer” and “syntax”.

2.1 Symbolic Expression of Audio Synthesis Algorithms

Audio synthesis algorithms (ASA’s) are numerical algorithms designed to fill a buffer with an audio signal. The algorithm for rendering a pure sine wave tone is perhaps the simplest ASA and can be expressed mathematically as equation [1] or as an equivalent S-expression in equation [2]. The variable $f$ is the desired frequency of the sine wave in Hertz and $t$ is the time in seconds. To use this equation to fill an audio sample buffer, one would calculate its output at the desired frequency $f$ and values of $t$ corresponding to the start times of the samples. The value of $t$ in seconds for the $n$th sample is $\frac{n}{\text{sample rate}}$.

\[ \sin(2 \times \pi \times f \times t) \]  
\[ (\sin (\times (\times (2) (\pi)) (\times (t) (f)))) \]
2.1 Symbolic Expression of Audio Synthesis Algorithms

To demonstrate the relation between traditional audio synthesis diagrams and S-expressions I have depicted the sine wave rendering algorithm as each in Figures 1 and 2 respectively.

![Figure 1: Traditional synthesis diagram of sine wave rendering algorithm](image1)

![Figure 2: S-expression diagram of sine wave rendering algorithm](image2)

Expressions such as the one used to create a sine wave are so important to the domain of musical sounds that it will be convenient to encapsulate them as single units in synthax. This allows for a larger amount of programs to produce human-recognizable sound than would be the case if it was instead necessary to build the eight-unit sub-expression shown in Figure 2 to express a sine wave.

The sinuscb primitive in synthax is a basic implementation of the algorithm for rendering a sine wave, with additional parameters to assign waveform phase and partial (ratio of the center frequency $f$). The augmented sine wave rendering expression and its equivalent synthax encapsulation are shown below in Figures 3 and 4. For audio examples and more information on this subject, see Appendix C.2.
2.2 Syntax

Synopsis is a functional programming language with a symbolic expression-based syntax. Legal expressions begin with a function name followed by zero or more continuous (c) or discrete (d) numeric parameters and zero or more child expressions as shown in equation 3. Each node in the expression tree must keep track of its minimum and maximum output amplitude determined by the minimum and maximum output values of its children. These values are used internally by some nodes to map output values of a child to a suitable range of values used locally as a control signal.

\[
(function\ name\ \{\ c/d\ min\ val\ max\ \} \ldots (child\ expression) \ldots)
\]  \hspace{1cm} (3)

Synopsis expands on the concept of "ephemeral random constants" from Koza [15]. An ephemeral random constant in traditional genetic programming is a terminal primitive with a constant numerical value that is initialized uniformly at random within a specified range when it is created. This primitives is used to seed a variety of numerical constants throughout the program population. In synopsis, primitives both non-terminal and terminal can have internal parameters with pre-defined ranges that determine their mode of operation. For example, the lfo\* non-terminal primitive has a numerical parameter that determines its frequency, and the const terminal primitive has a parameter that determines its constant output value making it the synopsis version of the traditional ephemeral random constant primitive.

The current primitives in synopsis are listed in the table below. For more information about each primitive, see Appendix A.

Figure 3: S-expression diagram of augmented sine wave rendering algorithm

Figure 4: Synopsis encapsulation of Figure 3
2.3 Rendering Audio From Synthax Code

Audio is generated from synthax programs recursively using a post-order traversal. Any given expression must have all audio signals from its children before calculating its own output. The audio signal trickles up from the leaves towards the root node where the final output is rendered. At the present time synthax programs are strictly trees; no expression can have more than one parent.

In order to maximize efficiency the object representation of an expression must know certain conditions about the audio environment before rendering. These necessary conditions are the sample rate, the render block size, and the maximum render length in seconds, allowing the expressions to allocate necessary memory in advance of rendering.

This rendering paradigm is highly parallelizable. Each additional thread added to the rendering system requires only one additional floating-point buffer the size of the render block to be allocated. Rendering can be distributed amongst a pool of worker threads evaluating progressively shallower sub-expressions. Allocating the worker threads to sub-expressions with larger numbers of children may be the most efficient way to approach parallelization. At present time no parallel rendering implementation exists for synthax, though it is certainly possible and is an area for future investigation.

3 Application 1: Automatic Timbre Mimicking

The first application of synthax is a command-line tool for automatically generating audio synthesis algorithms that have a similar timbre to a target sound using genetic programming.
A machine-computable metric of timbre comparison that mimics human judgment will be necessary to allow the genetic algorithm to find timbres that would sound convincingly similar to a human. For this reason it is important to have a working definition of timbre and its human interpretation before beginning the investigation.

3.1 Defining Timbre

Timbre is a collection of physical properties of a sound and is usually explained in terms of its human utility rather than its underlying components. The American National Standards Institute defined timbre as “that attribute of auditory sensation in terms of which a listener can judge that two sounds similarly presented and having the same loudness and pitch are dissimilar” [1]. The ambiguity lies in determining what exactly “that attribute” is and what physical phenomena cause it. This psychoacoustic debate continues to this day; the advent of computer technology has introduced both a road block and an opportunity in our understanding of timbre [5]. Computers have the ability to create any sound representable by a discrete waveform and thus the umbrella term timbre can no longer be interpreted as a direct association between certain sounds and the concrete characteristics of the objects that produce them. Wishart said in his book Audible Design [25]:

The spectral characteristics of sound have, for so long, been inaccessible to the composer that we have become accustomed to lumping together all aspect of the spectral structure under the catch all term “timbre” . . .

Early attempts to characterize the physical components of timbre mostly focused on the relative amplitudes and ratios of partial tones. These assumptions were based perhaps too heavily on Fourier’s theorem that any periodic waveform can be decomposed into simple sin wave components [11][12]. Like most aspects of music however, humans interpret not the absolute values of individual notes or frequencies but the way their relations change over time. Characterizations of timbre that place full emphasis on instantaneous spectral relationships are antiquated and have been augmented by multidimensional descriptions made up of several components with codependencies [3]. One such description was proposed by J. F. Schouten in 1968 and involves five characteristics [19][5]:

1. The range between tonal and noise-like character
2. The spectral envelope
3. The time envelope in terms of rise, duration and decay
4. The changes both of spectral envelope (formant-glide) and fundamental frequency (micro-intonation)
5. The prefix, an onset of a sound quite dissimilar to the ensuing lasting vibration.

For the purposes of this application, these five perceptual characteristics will be used as guidelines to develop a method for comparing the timbral characteristics of two sounds. A computable metric of timbral comparison that is similar to human judgment is necessary for a genetic algorithm designed to produce synthesis algorithms capable of believably imitating target timbres.
3.2 Motivations

Chowning summarized the motivations for timbre mimicking research in his paper detailing the technique of frequency modulation audio synthesis.

Of interest in both acoustical research and electronic music is the synthesis of natural sound. For the researcher, it is the ultimate test of acoustical theory, while for the composer of electronic music it is an extraordinarily rich point of departure in the domain of timbre, or tone quality. [2]

Acoustic instruments are something of a rigor test for any new synthesis technique. They represent well-known “problems” that can be used to demonstrate the adaptability of new synthesis algorithms and the breadth of sounds that they can create. Researchers have traditionally used Fourier analysis to manually adapt functional structure and numeric parameters of synthesis techniques to mimic acoustic instruments. This thesis presents a system that allows one to supply a target sound and automatically receive a synthesis algorithm that mimics its timbral characteristics.

3.3 Related Work

Audio synthesis algorithms are a well studied aspect of computer music, and for as long as they have existed researchers have been trying to use them to mimic recognizable acoustic timbres.

3.3.1 Early Investigations (1950-1970)

Max Mathews created the first widely used computer program for audio synthesis in 1957 called MUSIC. MUSIC and its descendants in the MUSIC-N family of programs operate on high-level audio generators and functions. Mathews wrote that “there are no theoretical limitations to the performance of the computer as a source of musical sounds, in contrast to the performance of ordinary instruments.” [17] Jean-Claude Risset worked with Mathews in the late 1960’s to attempt to produce with a computer sounds that mimicked acoustic instruments. While they were successful in creating brass-like and string-like sounds by analyzing the spectral content of recordings, their conclusion was that “additional factors must be discovered to yield an impeccable match to a given [instrument]” [18].

3.3.2 FM Synthesis (1970-1990)

In 1973 John Chowning introduced a method for utilizing frequency modulation to efficiently synthesize sounds with rich harmonic spectra. He states in the introduction of the related paper that “temporal evolution of the spectral components is of critical importance in the determination of timbre” [2]. Perhaps one of the “additional factors” elusive to Mathews and Risset was temporal evolution of spectral components. The technique of FM audio synthesis opened up new possibilities for efficiently rendering complex spectra with temporal evolution. Chowning himself used FM synthesis to mimic brass, woodwind, percussive sounds, and later a singing voice [3] with impressive subjective results.
Work in the 1980’s on mimicking acoustic timbres was largely commercial after Stanford licensed Chowning’s work on FM audio synthesis to Yamaha. Audio researchers at Yamaha continued to mimic tones of acoustic instruments by modifying the parameters and carrier/modulation structure of FM synthesis.

3.3.3 Genetic Algorithms (1990-present)

Traditional Genetic Algorithms Genetic algorithms (GA’s) were only popularized in the 1980’s despite being studied theoretically for several decades prior. GA’s emulate the natural process of evolution over a population of genotypes, assigning higher likelihoods of reproduction to genotypes whose phenotypes perform better on a specified task.

In 1993, Horner successfully demonstrated a system to automate timbre mimicking using a genetic algorithm [13]. The approach optimized the parameters of a fixed FM synthesis structure with a single modulator and multiple parallel carriers. Later, Johnson used an interactive genetic algorithm to control parameters for a granular synthesis algorithm [14] and Wehn used ideas from natural selection to explore the parameter space and in a more limited sense the connection space of FM-like synthesizers [24]. Genetic algorithms are well-suited for searching large, multi-dimensional spaces such as the parameters of audio synthesis algorithms and as such are a good choice for the task.

Genetic Programming John Koza’s 1994 book defined a new class of genetic algorithms altogether [15]. Koza borrowed concepts from traditional genetic algorithms to evolve computer programs and called the technique genetic programming (GP). The computer programs that GP evolves are represented as symbolic expressions. The set of functional and terminal expressions that GP can use to create programs are called primitives. Genetic programming can be adapted to many domains by the inclusion of domain-specific primitives. In 2000, Ricardo Garcia successfully demonstrated the technique of GP in the domain of timbre mimicking [7].

Garcia’s approach evolved programs that represented a set of instructions for building audio synthesis topologies such as the topology depicted in Figure 1. This indirect encoding was motivated by the fact that audio synthesis topologies can be cyclical and S-expressions cannot be. However, these non-cyclical “instruction trees” could contain instructions to build a cyclical synthesis topology and therefore allow unmodified genetic programming to create cyclical synthesis algorithms.

Garcia made the following statement about his instruction trees: “every [instruction] tree maps into a single topology…but a topology can be represented by many different [instruction] trees” [8]. This implies that the genetic algorithm was searching through a larger space of instruction trees representations that map into a smaller space of synthesis algorithms. Despite lacking the possibility of feedback, a more direct encoding of synthesis algorithm genotypes may allow for more efficient searching of the audio timbre space.

3.4 Method

This approach to timbre mimicking uses genetic programming to manipulate a population of synthax programs. The primary difference between this method and Garcia’s is that the
population of programs that genetic programming is maintaining contains directly-encoded audio synthesis algorithms as opposed to instructions for building them. *Synthax* programs contain many numerical parameters that contribute significantly to their perceived timbre. These parameters are sub-optimized during evaluation and the best combination of parameters found for that structure is used during GP re-population, adding Lamarckism to this approach.

The inputs to this system are a recorded target sound stored as a WAV file, the fundamental frequency of the target sound, and in some cases an amplitude envelope calculated from the target sound. Providing the fundamental frequency allows the genetic algorithm to focus on recreating the timbre of the instrument, rather than searching for the center frequency. *Synthax* primitives operate on multiples of the fundamental allowing the user to test generalization of the timbre to other center frequencies. A visual summary of this methodology appears as a flowchart in Figure 5.

Figure 5: Experiment flowchart for timbre mimicking
3.4 Method

3.4.1 Initialization

The initial population of programs is generated by Koza’s “ramped half-and-half” method. This method creates an equal number of expressions of all height values between two and a maximum initial height parameter. For each height value, half of the expressions are created by Koza’s “full” method and half of them with the “grow” method. The “full” method creates expressions where all leaf nodes are at a specific depth. The “grow” method creates expressions of variable shape up to the max initial height.

I borrow the concept of “ephemeral random constants” to generate initial numerical parameters for the expression trees. In traditional genetic programming, nodes representing numerical constants are always terminal and can be initialized randomly or hard-coded to a particular value. Synthax programs contain several primitives both terminal and non-terminal which internally have one or more numerical parameters that determine their output. These parameters (which can be floating-point values or integer values) are initialized uniformly at random during initialization time.


3.4.2 Sub-optimization

Two methods of numerical sub-optimization are used for synthax programs in timbre mimicking experiments: repeated uniform randomization and CMA-ES.

Repeated Uniform Randomization  This method of sub-optimization serves as a control for evaluating other, more complex methods. In repeated uniform randomization, the \( n \) numerical parameters of a synthax program are uniformly randomized to new values a specified number of times. The best combination of parameters as reported by the fitness function is saved and reintroduced back into the GP population once the specified number of randomizations have been performed.

CMA-ES  The Covariance Matrix Adaptation Evolutionary Strategy (CMA-ES) is an evolutionary strategy designed to optimize a set of \( n \) numeric variables [10]. CMA-ES has been empirically shown to out-perform many numerical optimization methods and find global optima with fewer fitness evaluations [9]. This makes it a good choice for quickly optimizing the \( n \) numerical parameters in a fixed-structure synthax program.

In CMA-ES, candidate solutions are sampled using a covariance matrix which, at a high level, represents pairwise dependencies between the \( n \) random variables. The algorithm uses the reported fitness values of the top \( \mu \) candidates to update the values of the covariance matrix. During sub-optimization, the initial values of the \( n \) numerical parameters for a synthax program (created by uniform randomization or sub-optimization during a previous generation) are reported to the CMA-ES algorithm. The algorithm provides candidate solutions (sets of values for the \( n \) parameters) which are applied to the synthax program. The output of the program with the new parameters is then compared to the target audio using the
fitness function, and the fitness value is reported to the CMA-ES algorithm. This process repeats until a specified number of generations with population size $\lambda$ has been evaluated.

Unless otherwise specified, the proportion of $\mu$ to $\lambda$ for all experiments using the CMA-ES sub-optimization method is 0.5. This implies that the top half of the candidates from one generation is used to update the algorithm’s distribution parameters.

### 3.4.3 Fitness Evaluation

The two fitness functions employed for these tests were designed to be quickly-computable, human-inspired comparison metrics for two audio signals. In both fitness functions, the target samples are partitioned into $m$ groups of $n$ samples with $o$ overlap between groups. A discrete Fourier transform is used to calculate magnitude and phase values at $n/2$ frequency bins equally spaced between 0 and $\frac{\text{sample rate}}{2} \text{ Hz}$.

The following values are calculated for each bin $j$ of group $i$ in the target spectrum:

- $t\text{mag}_{(i,j)}$: Magnitude of the frequency bin for the target
- $t\text{phs}_{(i,j)}$: Phase of the frequency bin for the target

Each candidate S-expression is rendered to produce a candidate wave form with the same number of samples as the target. The candidate samples are partitioned into groups of $n$ with $o$ overlap in the same way as the target samples. The following values are then calculated for each bin $j$ of group $i$ from the candidate samples:

- $c\text{mag}_{(i,j)}$: Magnitude of the frequency bin for the candidate
- $c\text{phs}_{(i,j)}$: Phase of the frequency bin for the candidate

**Frequency-Time Squared Error**  One of the most basic fitness functions for comparing signals in the frequency domain is to compute the squared difference between the frequency spectra of the candidate and the target over time. This fitness function is not sympathetic to the subtleties of human sound perception but does provide a basic and easily-computable metric for timbre comparison. This function can be computed as follows:

$$mag\_error_{(i,j)} = (t\text{mag}_{(i,j)} - c\text{mag}_{(i,j)})^2$$
$$phs\_error_{(i,j)} = (t\text{phs}_{(i,j)} - c\text{phs}_{(i,j)})^2$$

$$fitness = \text{phs\_weight} \times \left( \sum_{i=1}^{m} \sum_{j=1}^{\frac{n-1}{2}} phs\_error_{(i,j)} \right)$$
$$+ \text{mag\_weight} \times \left( \sum_{i=1}^{m} \sum_{j=1}^{\frac{n-1}{2}} mag\_error_{(i,j)} \right)$$

**Frequency-Time Perceptual Error Weighting**  Still in the frequency domain, we can augment the squared error fitness function to weight the comparisons of some target/candidate bins more strongly than others. This can be done by calculating some form of moving average across the Fourier transform of each sample window and assigning comparison weights based on how far a particular bin deviates from the average.

- $MAC_{(i,j)}$: Moving average value at the bin
3.4 Method

- **under\(_{(i,j)}\)**: Penalty for candidate undershooting the bin
- **over\(_{(i,j)}\)**: Penalty for candidate overshooting the bin

The following values are calculated for each group \(i\):
- **max\(_{\text{above}}\(_{(i,j)}\)**: \(\max(MAG\(_{(i,j)}\) - MAC\(_{(i,j)}\))\)
- **max\(_{\text{below}}\(_{(i,j)}\)**: \(\max(MAC\(_{(i,j)}\) - MAG\(_{(i,j)}\))\)

\(\text{under}\(_{(i,j)}\)\) and \(\text{over}\(_{(i,j)}\)\) are calculated for each bin \(j\) of group \(i\) in the following way:

\[
\begin{align*}
  \text{under}\(_{(i,j)}\) & = \begin{cases} 
    \text{base}_p & \text{if } (tmag\(_{(i,j)}\) \leq MAC\(_{(i,j)}\)) \\
    \frac{tmag\(_{(i,j)}\) - MAC\(_{(i,j)}\)}{\text{max}\(_{\text{below}}\(_{(i,j)}\))} \times \text{additional}_p + \text{base}_p & \text{if } (tmag\(_{(i,j)}\) > MAC\(_{(i,j)}\))
  \end{cases} \\
  \text{over}\(_{(i,j)}\) & = \begin{cases} 
    \text{base}_p & \text{if } (tmag\(_{(i,j)}\) \leq MAC\(_{(i,j)}\)) \\
    \frac{MAC\(_{(i,j)}\) - tmag\(_{(i,j)}\)}{\text{max}\(_{\text{above}}\(_{(i,j)}\))} \times \text{additional}_p + \text{base}_p & \text{if } (tmag\(_{(i,j)}\) > MAC\(_{(i,j)}\))
  \end{cases}
\end{align*}
\]

Each candidate is assigned fitness in the following way:

\[
\begin{align*}
  \text{phs.error}\(_{(i,j)}\) & = (|tphs\(_{(i,j)}\) - cphs\(_{(i,j)}\)|)^{\text{phs},p} \\
  \text{mag.error}\(_{(i,j)}\) & = \begin{cases} 
    (tmag\(_{(i,j)}\) - cmag\(_{(i,j)}\))^{\text{under}\(_{(i,j)}\)} & \text{if } (cmag\(_{(i,j)}\) \leq tmag\(_{(i,j)}\)) \\
    (cmag\(_{(i,j)}\) - tmag\(_{(i,j)}\))^{\text{over}\(_{(i,j)}\)} & \text{if } (cmag\(_{(i,j)}\) > tmag\(_{(i,j)}\))
  \end{cases}
\end{align*}
\]

\[
\text{fitness} = \text{phs.weight} \times \left( \sum_{i=1}^{m} \sum_{j=1}^{\frac{n}{2}-1} \text{phs.error}\(_{(i,j)}\) \right) + \text{mag.weight} \times \left( \sum_{i=1}^{m} \sum_{j=1}^{\frac{n}{2}-1} \text{mag.error}\(_{(i,j)}\) \right)
\]

The motivations for this fitness function come from the attributes of timbre listed in Section 3.1. At a high level, the weighting system is giving a higher penalty for a candidate overshooting a target’s spectral valley and a higher penalty for undershooting a spectral peak. Penalty is constant for a candidate undershooting a target valley and overshooting a target peak. A spectrogram of a piano sample and its undershooting and overshooting penalty values for the first \(n\) samples are shown below in Figures 6, 7 and 8.
3.4 Method

This fitness function was empirically found early in the investigation to guide the genetic algorithm towards timbres a human would consider to sound similar to the target and away from local optima. Using the simple squared-error function, the genetic algorithm often became fixated on the local minimum of silence. The perceptually-inspired fitness function lessened the probability of silence convergence.

3.4.4 Population Maintenance

The genetic operations I use for this experiment are those typically associated with genetic programming. Individuals are selected with probabilities determined by the ratio of their fitness to the group (where a lower fitness value is better due to the fact that fitness is a calculation of error).

- Reproduction: Select one program and clone it without mutation into the next generation.

- Crossover: Select two parent programs, swapping two randomly selected sub-expressions from within them and adding both resultant programs to the next generation. This operation is quite expressive as the heights of the resultant children can range from 0 to the sum of the heights of the parents; it can produce radically different children even if the parents are identical. If one of the children is taller than the specified max height, we reproduce one of the parents into the next generation.

- Mutation: Select one program and select one sub-expression \( s \) from it uniformly at random. Replace this sub-expression with a new sub-expression generated by Koza’s “grow” method with height calculated by subtracting the depth of \( s \) in the original expression from the specified max height.

- New: Use Koza’s “grow” method to generate a random new program and add it into the population.
3.5 Implementation Details

The command line application implementing the described method was built using the JUCE C++ toolkit. Spectrum analysis for the fitness function was performed by Kiss FFT, a relatively fast and extremely lightweight implementation of the Fast Fourier transform. OpenBEAGLE was used to perform CMA-ES suboptimization as outlined in Section 3.4.2. Timbre mimicking experiments listed in Appendices B.1, B.2 and C.1 were run on the UT Condor Computing Cluster with five copies of each test (each with a different random number generator seed in the range of 0 to 4).

3.6 Experiments

Timbre mimicking experiments using a common set of parameters were performed on a wide variety of sounds. The set of parameters used was informally found to produce decent mimics of most sounds. The results of these experiments are online, and you can hear the resultant audio and synthax programs in Appendix C.1.

In addition to informal mimicking experiments, two human studies were conducted in which a group of human participants were asked to compare “target” sounds to “test” sounds produced by genetic programming. Participants were played a recorded audio script summarizing the instructions for the experiment. During this script, the target sound was played a total of five times to allow participants to become accustomed to hearing it. Participants subsequently listened to a series of sounds produced by a variety of techniques and parameter sets. The target sound was always played before any test sound, and each test sound was always played twice. The participants were asked to intuitively compare the test sounds to the target sound on a scale of 1 to 10, where 1 indicated no resemblance between the sounds and 10 indicated a perfect match. Two “calibration” sounds were always played before any test sounds to give participants a chance to familiarize themselves with the testing format. The data from these calibration sounds was discarded.

3.6.1 Human Study #1: Synthesized Piano Tone

The sample used for human study #1 was a recording of a synthesizer playing a C_4 piano timbre created by frequency modulation. This sample, along with an evolutionary lineage of sounds, was produced by Garcia’s AGeSS system and collected from a video on his website [6]. Unfortunately, the original sample used by AGeSS to produce the results on the video was not available and a compressed version captured from the audio of the video was used. The compression is evident in the spectrogram of the sound file shown in Figure 10.

![Figure 9: Waveform for human study 1 sample](image_url)
3.6 Experiments

Figure 10: Spectrogram ($FFT_{size} = 1024$) for human study 1 sample

Participants in this test were played a series of 25 sounds and asked to compare each of them to the target. The ordering and origin of the sounds is listed below:

- **1-2)** Two calibration sounds, #1 was not very similar to the target and #2 was a recording of a real piano playing $C_4$.

- **3-10)** The eight test sounds from #3 through #10 have strictly decreasing error as assigned by the Frequency-Time Squared Error fitness function described in Section 3.4.3 and were collected across all experiment runs in Appendix B.1. Sound #3 had an error value of 118335.7 and sound #10 had an error value of 19648.5. Sounds #4-9 were geometrically spaced in between #3 and #10 with a ratio of $r \approx 0.7738$ (calculated so that there would be six samples in between the worst and the best).

- **11-18)** The eight test sounds from #11 through #18 have strictly decreasing error as assigned by the Frequency-Time Perceptual Error Weighting fitness function described in Section 3.4.3 and were collected across all experiment runs in Appendix B.1. Sound #11 had an error value of 1000967.729 and sound #10 had an error value of 19490.45583. Sounds #4-9 were geometrically spaced in between #3 and #10 with a ratio of $r \approx 0.5698$ (calculated so that there would be six samples in between the worst and the best).

- **19-22)** Overall champions from all experiments listed in Appendix B.1 that used the Frequency-Time Squared Error fitness function. (Not included: champion of experiment 5 which was sound #10).

- **23)** Garcia generation 198 champion from his AGeSS website [6].

- **24-25)** Overall champions from all experiments listed in Appendix B.1 that used the Frequency-Time Perceptual Error Weighting fitness function. (Not included: champion of experiment 2 which was sound #18).
3.6.2 Human Study #2: Audio From Known Synthax Program

The second sample used for the human evaluation experiment was a recording of a synthax program rendered with a center pitch of $A_2$ (though because of the parameters of the structure the center frequency of the audio produced by the program is not an $A_2$). This test serves as a way to answer the question of whether or not this timbre mimicking method can successfully match timbres of sounds that it is certainly capable of producing.

![Figure 11: Waveform for human study 2 sample](image)

![Figure 12: Spectrogram ($FFT_{size} = 1024$) for human study 2 sample](image)

The synthax program used to create the sound is shown below as text:
\[
(\ast \left( \const \left\{ c \, -1 \, 0.849 \, 1 \right\} \right) \pm \left( \sinosc \left\{ d \, 0 \, 0 \, 0 \right\} \left\{ c \, 0.5 \, 2.308 \, 10 \right\} \left\{ c \, 0 \, 0.349 \, 1 \right\} \right) \left( \ast \left( \sinosc \left\{ d \, 0 \, 0 \, 0 \right\} \left\{ c \, 0.5 \, 1.139 \, 10 \right\} \left\{ c \, 0 \, 0.049 \, 1 \right\} \right) \left( \sinosc \left\{ d \, 0 \, 0 \, 0 \right\} \left\{ c \, 0.5 \, 5.766 \, 10 \right\} \left\{ c \, 0 \, 0.042 \, 1 \right\} \right) \right) \right)
\]

Participants in this test were played a series of 8 sounds and asked to compare each of them to the target. The ordering and origin of the sounds is listed below:

- 1-2) Two sounds produced by a commercial synthesizer, #1 was not very similar to the target and #2 was a real piano recording playing $C_4$.

- 3-5) Overall champions from all experiments listed in Appendix B.2 that used the Frequency-Time Squared Error fitness function.

- 6-8) Overall champions from all experiments listed in Appendix B.2 that used the Frequency-Time Perceptual Error Weighting fitness function.
3.7 Results and Conclusions

This section contains discussions and conclusions drawn from the results of the two human studies. This discussion must be prefaced by some informal comments I received from the participants. A frequent comment concerned a participant’s uncertainty with their evaluation metric. Participants were instructed to compare sounds to the target on a scale of 1 to 10, where 1 indicated that the sounds were as little alike as possible and 10 indicated a perfect match. I could have instructed them to compare the **timbres** of the sounds on the same scale, however even musicians with many years of experience have unclear or conflicting interpretations of timbre. For this reason I was intentionally vague, instructing participants to not think too much about their comparison metric and to answer intuitively.

Another consideration about the study is that the upper part of the scale is well-defined, while the lower side of the scale is incredibly hard to evaluate. If two sounds are almost or exactly the same, it is trivial for a human to recognize this and respond with a 9 or a 10. However, how could one know if a sound is as little like another as possible? Is the sound of a duck’s quack closer to a dog’s bark than a cow’s moo? Or should both the quack and the moo be scored a 1 when compared to a bark?

A better version of these human studies would have instructed participants to order groups of sounds from sounding the least like the target to sounding the most. Any unclarity in the low end of the fitness scale would have been resolved. However, such a test would require the creation of a user interface allowing participants to repeatedly listen to sounds until they sorted the sounds in their desired order. This would have significantly increased both the complexity of implementing the test and participant time commitment.

Regardless of these issues participants seemed to respond fairly consistently to the provided instructions, developing their own intuitive scale to evaluate the sounds as the study progressed. In addition to internal consistency, there was a surprising amount of group consistency amongst participants. The average standard deviation for all evaluation averages of non-calibration sounds in human study #1 was $s = 1.196$, and $s = 0.482$ in human study #2. There was even a sample in human study #2 that all participants gave the same score.

3.7.1 Human Evaluation of Fitness Functions

The human evaluation of fitness functions was part of human study #1 introduced in Section 3.6.1. Sounds #3 through #10 in the experiment had geometrically-decreasing error as determined by the Frequency-Time Squared Error function and sounds #11 through #18 had geometrically-decreasing error as determined by the Frequency-Time Perceptual Error Weighting function. There were $n = 8$ participants in this study, with average age of 22.5 years ($s = 2.83$) and average musical experience of 9.25 years ($s = 5.7$).

It would be difficult for a human to evaluate the sounds on the same scale that the computer fitness functions use. For this reason, it is necessary to transform the output values of the computer fitness functions to the same scale the human participants used in order to evaluate how well the functions approximate human judgement. The computer fitness functions output 0 when comparing identical audio however there is no easily-computable maximum value that they could output as the audio data could contain arbitrary floating point numbers. For this reason, the error of the least fit individual from each series was used
as the maximum error value for the transformation.

**Frequency-Time Squared Error** The values computed by the fitness function were linearly transformed from the error range of $[118335.719, 0]$ to the fitness range of $[1, 10]$ for the following chart.

![Frequency-Time Squared Error](image)

Figure 13: Human evaluation compared to Frequency-Time Squared Error on the same set of sounds. Error bars are standard deviations of the average of human evaluations for each sound.

The eight sounds that were part of this evaluation are sound #1 (least fit) through sound #8 (most fit). The average range between the minimum and maximum fitness scores from participants for all of the eight sounds was $3.125 (s = 0.835)$. The minimum range was 2 for sound #8 and the maximum range was 5 for sound #6. The highest score awarded to any sound was 9 assigned by one participant to sound #7. The lowest score assigned to any sound was 1 assigned by one participant to sound #2.

From the charted responses, it appears as though the Frequency-Time Squared Error fitness function generally corresponds to human evaluations. There is an increasing trend from least fit to most fit for human evaluation averages of sound #1 through sound #8, though interestingly the averages fluctuate between higher and lower scores when observed in pairs. This could possibly be attributed to the influence of the previous evaluation on the current evaluation.
3.7 Results and Conclusions

**Frequency-Time Perceptual Error Weighting** The values computed by the fitness function were linearly transformed from the error range of \([1000967.729, 0]\) to the fitness range of \([1, 10]\) for the following chart.

![Frequency-Time Perceptual Error Weighting](image)

Figure 14: Human evaluation compared to Frequency-Time Perceptual Error Weighting on the same set of sounds. Error bars are standard deviations of the average of human evaluations for each sound.

The eight sounds that were part of this evaluation are sound #1 (least fit) through sound #8 (most fit). The average range between the minimum and maximum fitness scores from participants for all of the eight sounds was 3.75 (\(s = 0.886\)). The minimum range was 3 for sounds #1, #3 and #6 and the maximum range was 5 for sounds #2 and #4. The highest score awarded to any sound was 9 assigned by one participant to sound #8. The lowest score assigned to any sound was 1 assigned by several participants to sounds #2, #5 and #6.

From the chart in Section 3.7.1, it appears as though the Frequency-Time Perceptual Error Weighting fitness function has poor correspondance to human evaluations. In fact, the sound with the lowest human evaluation average (sound #6) was rated the third most fit by the computer fitness function. Something to consider in analyzing correspondance between the human and computer evaluations is that the value of 1 was chosen rather arbitrarily for the adjusted computer fitness function, as the values had to be transformed from an error calculation to a fitness calculation. Additionally, the error value representing 1 on the fitness scale was completely different for both transformations in order to assign the least fit sound a 1.
3.7 Results and Conclusions

Conclusions  Regardless of potential issues with the human study or transformation of error values, it appears as that the Frequency-Time Squared Error fitness function is a better approximation of human timbral comparison than the Frequency-Time Perceptual Error Weighting fitness function. The perceptual error weighting fitness function was created initially to help the genetic algorithm not converge on the local minimum of silence, but as the numerical suboptimization system became more robust this premature convergence problem became a non-issue. This conclusion makes intuitive sense; the human ear performs an analogue spectral analysis of incoming audio over time and thus a calculation of error in the frequency-time domain makes for a good approximation. Any attempt at making this fitness function more sensitive to the subtleties of human timbre perception will require additional research in psychoacoustics.

To hear the sounds from these fitness function evaluations please see Appendix C.3.

3.7.2 Synthesized Piano Tone

In addition to evaluating the two fitness functions, participants in human study #1 evaluated the champions of all the experiments listed in Appendix B.1. Experiments #1 through #8 in the Figure 15 correspond to experiments #1 through #8 in the appendix, though they did not appear in this order in the human study.

Figure 15: Human evaluation of champions of the experiments listed in Appendix B.1. Error bars are standard deviations of the average of human evaluations for each sound.
3.7 Results and Conclusions

- “Garcia” (human average = 6.125, s = 0.835) in the above chart was a sound produced by Garcia’s AGeSS system trying to imitate the same target [7].

- Experiment #1 (human average = 5.625, s = 0.916) had the same parameter set that Garcia used to produce his result except for the sub-optimization method and synthesis algorithm encoding. I would have used the same sub-optimization method as Garcia in order to decrease the number of variables had that information been available in his paper. Regardless, it seems that the champion of this experiment was not as successful at convincing humans of timbral similarity as Garcia’s.

- Experiment #2 (human average = 5.625, s = 0.916) was the same as experiment #1 except that it used the uniform random sub-optimization technique instead of CMA-ES. Human evaluation produced nearly identical scores for these two experiments, possibly implying that CMA-ES is performing no better than random combinations of parameters. However, I would not necessarily expect to see CMA-ES outperform random initialization during low amounts of sub-optimization evaluations (population size of 10 and # generations 4 vs 40 random initializations).

- Experiment #3 (human average = 7.125, s = 0.835) was the same as experiment #1 except that it used no numerical sub-optimization, allowing for a much larger population size and more generations with the same number of fitness evaluations. Surprisingly, this technique produced the result with the highest human evaluation score. The score for Garcia’s mimic was more than a standard deviation below the score for the champion of experiment #3.

- Experiment #4 (human average = 6, s = 1.069) was the same as experiment #1 except that it had a larger population size and much fewer generations. Humans evaluated the champion of this experiment slightly higher than experiments #1 and #2, but not as high as Garcia’s mimic.

- Experiment #5 (human average = 6.25, s = 1.035) was the same as #1 except that it had an experimental set of primitives, including a node that performs frequency-modulation synthesis. Humans evaluated the champion of this experiment slightly higher than Garcia’s mimic.

- Experiment #6 (human average = 7, s = 1.309) was the same as #1 except that it used the Frequency-Time Perceptual Error Weighting fitness function. Even though it was concluded previously that this fitness function was not the most representative of human judgement, the champion of this experiment produced a remarkably pleasant mimic of the target.

- Experiment #7 (human average = 4.75, s = 1.389) was the same as #6 except that it had the same distribution of population size and generations as experiment #4. It had one of the lower human evaluation averages.

- Experiment #8 (human average = 4.375, s = 1.407) was the same as #6 except that it had no sub-optimization like experiment #3. It had the lowest human evaluation average.
3.7 Results and Conclusions

Conclusions  It is difficult to draw any definitive conclusions from the data gathered in this human study. Genetic programming is quite susceptible to random chance and as such, these data points represent results produced by one particular random seed. Ideally, human participants would have evaluated the champions of these parameters on the results of many random seeds, but I did not have sufficient data from Garcia to compare to and this would have greatly increased the length of the study.

That being said, it is a reasonable assumption that Garcia selected the best champion to demonstrate his results from a wide range of parameters and random seeds. Superior human evaluation on champions from several of my experiments (#3, #5 and #6) demonstrates that my timbre mimicking system at least has the capability to outperform AGeSS, though it is unknown whether or not it can consistently outperform it. In my opinion, the ability to audition synthax programs in real-time via MIDI gives my system an edge in utility.

To hear the sounds from these human evaluations and to view or use the associated synthax programs please see Appendix C.3.

3.7.3 Audio from Known Synthax Program

Participants in human study #2 (presented in Section 3.6.2) evaluated the champions of all the experiments listed in Appendix B.2. Experiments #1 through #6 in Figure 16 correspond to experiments #1 through #6 in the appendix, though they did not appear in this order in the human study.

![Audio from Known Synthax Program](image)

Figure 16: Human evaluation of champions of the experiments listed in Appendix B.2
3.7 Results and Conclusions

- Experiment #1 (human average = 8.0, s = 0.0) used pure genetic programming, a population size of 500, 50 generations and no sub-optimization. The primitive set was the same as the one used to generate the target program initially. For more detailed parameters, see Appendix B.2. Interestingly, the champion from this experiment was the only sound that all four participants scored the same.

- Experiment #2 (human average = 8.5, s = 0.58) was the same as experiment #1 but with a larger population size and more generations. Not surprisingly, the champion from this experiment had a lower error value and a higher human evaluation score.

- Experiment #3 (human average = 9.25, s = 0.55) was the same as experiment #1 but uses CMA-ES numerical suboptimization. The champion from this experiment had the highest human evaluation average.

- Experiment #4 (human average = 9.0, s = 0.82) was the same as experiment #3 except that it uses the Frequency-Time Perceptual Error Weighting fitness function. The champion from this experiment was scored highly.

- Experiment #5 (human average = 6.75, s = 0.5) was the same as #4 but with a larger population size and more generations. This experiment represented a much broader, longer search than experiment #4, but yielded much poorer results when evaluated by humans. This is a good example of how prone the timbre mimicking system is to poor performance due to random chance.

- Experiment #6 (human average = 8.75, s = 0.5) was the same as #1 except with uniformly random numerical sub-optimization. It did not score as highly as the champion of the same experiment using CMA-ES, suggesting that the targeted suboptimization might benefit the search algorithm even at low levels of suboptimization.

**Conclusions**  The purpose of this human evaluation was to confirm that my timbre mimicking system has the ability to rediscover a timbre it certainly has the capability of producing. This demonstrates that the system’s difficulty in mimicking certain sounds can possibly be attributed to the lack of necessary primitives. Due to the fact that numerical parameters in synthax can be floating-point, it is extremely unlikely that the target will be reproduced exactly even if the target was created by a synthax program. However, the results from the human evaluation demonstrate that the system can produce algorithms that are almost completely identical when observed by the human ear if the necessary primitives are certainly available.

To hear the sounds from these human evaluations and to view or use the associated synthax programs please see Appendix C.3.
4 Application 2: Interactive Evolution of Audio Synthesis Algorithms

To seek out new tonalities, new timbres...
To boldly listen to what no one has heard before. [21]

The second application of synthax is a VST plugin called evosynth that allows for interactive genetic programming. Users guide the genetic algorithm towards digital timbres that they find pleasant or intriguing in some way. Users can also save synthax programs they find as simple text files, creating a highly portable method of sharing audio synthesis algorithms. All parameters of evosynth can be controlled in real-time via MIDI through any digital audio workstation supporting the VST plugin format. The plugin also allows for real-time audition of synthax programs produced by the timbre mimicking application.

4.1 Motivation

The musical value of the computer does not lie, of course, in its ability to duplicate exactly what a real instrument can do, but rather in yielding an extended repertory of sounds, including and going beyond the classes of sounds of actual instruments. [18].

Figure 17: Screenshot of the evosynth VST plugin
Most composers stick to existing instruments when writing music. Acoustic composers typically restrict themselves to given sounds, such as the sounds that instruments in a wind ensemble or orchestra can produce. Electronic and electro-acoustic composers have a wider timbral palette to work with but usually end up working within the confines of available synthesizers and samples. It is challenging to discover novel sounds from thoroughly investigated synthesis techniques.

These issues suggest the need for alternative and accessible methods of timbre investigation for musicians and researchers. Many electronic musicians utilize synthesis types such as additive, subtractive, ring modulation, amplitude modulation, and frequency modulation to create sounds. An interactive genetic algorithm could allow producers to strategically search the space of audio synthesis algorithms of which these traditional synthesis techniques are a small subspace.

### 4.2 Method

This application uses genetic programming to evolve synthesis algorithms directly, utilizing a predetermined set of primitives. The output of this system is multiple synthax programs for the user to evaluate and/or modify.

Methodology for evosynth was heavily inspired by prior work in the field of interactive genetic art \[1, 20, 23\]. The user is presented with 10 initially random synthax programs (using the same initialization technique as detailed in Section 3.4.1) and can audition them using a real or virtual MIDI keyboard. The user can assign fitness values to programs he or she finds pleasing, or replace unpleasant programs with randomly generated ones. In this application a higher fitness value is better as we are rewarding nice-sounding synthesis timbres, not computing error. Once the user has assigned fitnesses for the current set of programs he or she can step to the next generation which is created by genetic programming operators where selection probabilities are determined via the user-assigned fitnesses. Currently there is no method of interactive sub-optimization outside of manually changing the numeric parameters in the synthax program.

The genetic operations I use in this application are a superset of those listed in Section 3.4.4. In addition to the operations listed there, the user can choose to save any program from one generation and load it in future generations. A user may also replace any individual with a random individual produced by Koza’s “grow” method \[15\]. These two techniques allow for human-interactive population maintenance.

### 4.3 Implementation Details

An early prototype for the interactive system was built using Cabbage, Csound and Python. After encountering performance limitations with this setup I completely reimplemented the application in C++ utilizing the JUCE library. The VST plugin was built and tested on Windows 7 x86, though JUCE makes porting the plugin to other operating systems and platforms quite simple. SAVIHost and Ableton Live 8.2.5 were used as test hosts for the VST plugin.
4.4 Experiment

The *evosynth* VST plugin is available online at the URL listed in Section 8. Users can download the plugin and interactively search the space of *synthax* programs for interesting or novel timbres. Users can also upload interesting programs they find to be stored and made available for public download. The text representation of the program is parsed upon submission to ensure that it is legal *synthax* code.

4.5 Conclusions

The *evosynth* VST plugin is a promising addition to this area of investigation. Unlike past attempts at applying genetic programming to timbre mimicking, discovered programs can be actually played as software instruments. The plugin acts as an intuitive platform for auditioning *synthax* programs.

*Evosynth* could be improved by the addition of more primitives. Time-varying primitives could help to add sonic variety to sounds produced by the plugin. It is also currently difficult for a user to interpret or edit a text representation of an S-expression outside of adjusting numerical parameters. A graphical representation of the programs allowing for structural and numerical manipulation would greatly increase the accessibility of the plugin.

At present time, *evosynth* is a promising area of active investigation. It represents an accessible way for electronic composers to interact with digital timbre search and is encapsulated in the most common audio plugin format. Popularization of the plugin could help contribute to “open-source” composition; *synthax* programs used in released music could be made available for public download to be used and manipulated by other composers.

5 Future Work

There are numerous applications and extensions of this research that I have left here as future work. Adding primitives to the *synthax* language is perhaps the most enticing and easiest direction of future investigation. It is fairly straightforward to add a primitive to the language as it only requires the creation of a new subclass and a small amount of modification to the parser. Additionally, there are some potential enhancements to be made to the core language structure. I will also present some possible extensions for current applications and ideas for future applications of the *synthax* language.

5.1 New Primitives

I have identified a few general categories of additional primitives that could be added to *synthax* and will elaborate on each category in the following sections.

5.1.1 Filters

Capturing digital audio filters as primitives would be particularly useful for creating more interesting sounds. Filters play a large role in subtractive synthesis which is impossible to achieve with the current set of primitives in *synthax*. Filter primitives could be represented
5.2 General \textit{Synthax} Enhancements

at a low level as randomly-generated difference equations or at a high level as filter types (low pass, high pass, notch, etc.) with children representing the center frequency and quality of the filter as well as the signal to be filtered.

5.1.2 Memory

Primitives that have memory could create some interesting delay and feedback capabilities for \textit{synthax}. For example, a delay primitive could store the previous $n$ samples and mix them in to the current $n$ samples with some amplitude multiplier. A memory primitive could store part of a signal in a buffer and then play that signal back at some time offset determined by a numerical parameter. A primitive could be introduced that references other nodes in the expression, creating the possibility of feedback. Care would have to be taken to ensure that such a primitive does not create an infinite loop.

5.1.3 Logic

There is currently only one primitive in \textit{synthax} that performs any logic operation. The \textit{switch} primitive switches between two child signals depending on whether or not a third child signal is above or below 0. Other logic primitives could be added representing “less-than,” “greater-than,” “equals-to,” etc. More logic gates combined with the addition of memory as discussed in Section 5.1.2 would create the exciting possibility of \textit{synthax} programs that can act as state machines operating on digital audio signals.

5.1.4 Splines

There is currently only one type of spline available in the \textit{spline} primitive, a simple linear interpolation between control points. Curved envelopes built using Bézier curves, Hermite curves and B-splines would be a useful addition to this primitive and allow for more complex enveloping.

5.1.5 Wave Table Generators

A primitive that generates a wave table in the same way that the \textit{spline} primitive generates a random linear envelope could be added to \textit{synthax}. Some primitives already exist with hard-coded wave tables: \textit{sawosc}, \textit{sinosc}, \textit{squareosc} and \textit{triangleosc}. It would be a fairly simple extension to add a primitive that generates a random wave table instead of using one of those predefined four.

5.2 General \textit{Synthax} Enhancements

Because of the nature of S-expressions, a given node cannot have multiple parents. This theoretically does not limit the capabilities of the \textit{synthax} programs produced by GP as an S-expression can have multiple identical terminals that would produce equivalent behavior to a node with multiple parents. However, this greatly increases search complexity and computation time to discover efficient algorithms such as FM synthesis using a single modulator and multiple carriers (the topology that \cite{13} optimized using a genetic algorithm).
A method for supporting feedback, whether added in at the language level or hacked in as a memory primitive, would add a lot of functionality to synthax. Some of the most interesting musical sounds are produced by feeding a signal back from the output of a system to an input of the same system. Additionally, an implementation of parallel rendering for a single synthax program would increase the utility of the language. Taller S-expressions can take a long time to render, and parallelization of this task could speed up render times on systems with slow but multi-core processors (such as processors on mobile devices).

5.3 Extensions for Current Applications

The two applications presented in this paper that use synthax have enormous potential for extension. A few of my own ideas for modifying and enhancing these applications are listed here.

5.3.1 Application 1 Extensions

1. Parallel evaluation and sub-optimization of candidates
2. More methods of sub-optimization
3. Fitness functions that better mimick human judgement
4. Further optimization of parameters that produce the best results on a wide variety of sound files

5.3.2 Application 2 Extensions

1. User can swap primitives in and out of the available set
2. User can change the likelihoods of particular primitives
3. User can modify the max initial height and max overall height of the population
4. User can modify the maximum amount of time that a program can render (currently hard-coded to 10 seconds)
5. User can specify a sound file and CMA-ES will be run to optimize the numeric parameters of the current program to get as close as possible to that sound file by some fitness function
6. User can run novelty search \cite{16} to discover synthesis algorithms with interesting harmonic content and development over time
7. Integration with Application 1, allowing the user to input a sound file and automatically receive a few synthax programs that mimick its timbre
8. Interactive sub-optimization of synthax program numerical parameters
9. Graphical representation and manipulation of synthax programs

10. Add center frequency as a time-varying variable so the plugin can process pitch mod-wheel MIDI data

11. Integration with a C++ REST library to allow interaction with the web service from within the plugin

5.4 New Applications

There are a few new applications of the synthax language that would serve as interesting extensions to the project.

1. Use CMA-ES or other sub-optimization methods on a fixed-structure synthax program for timbre mimicking. (Similar to Horner’s approach with an FM synthesis structure and a GA [13]).

2. Native mobile application for creating and modifying synthax programs (no genetic programming involved). Users would physically drag and drop synthax primitives into place and be able to adjust their parameters/audition the program in real time.

3. Compile evosynth on a Raspberry Pi to create physical hardware that can play synthax programs in a real-time, live performance setting.

6 Summary

This thesis has introduced the synthax programming language for specifying audio synthesis algorithms. Two applications of genetic programming on the synthax language with different fitness goals were also presented. Application one is a command line tool to find synthax programs that mimic the timbre of an input sound file. This application has potential benefits for the commercial music industry as an automatic method for mimicking acoustic and electronic instruments in light-weight software. Application two is an interactive VST program for musicians and composers to find digital synthesis algorithms with pleasing or novel timbres. This application benefits the modern “bedroom producer” who may be struggling to create a sound that he or she wants to use in a composition. A composer can also choose to share a synthax program that he or she has discovered for other composers to use or manipulate, promoting transparency in an otherwise opaque discipline. Application two can also potentially exist as a piece of commercial software as the online repository of sounds published by users becomes more robust. As synthax and evosynth themselves evolve, the range and types of sounds attainable through these technologies will grow enormously while accessibility and transparency will remain constant.

7 Acknowledgements

The author would like to thank the Computational Intelligence in Game Design lab, part of the University of Texas’s Freshman Research Initiative program, for providing the equipment
and workspace to implement this project. Thanks to Dr. Joel Lehman for his advice on the
genetic algorithm aspects of this project. Thanks to Ashvin Bashyam for his advice on the
subtleties of working with a frequency domain representation of an audio signal. Thanks to
Jules, creator of the JUCE C++ toolkit, for his attention to quality making this project a
pleasure to implement. Finally, the author wishes to express gratitude to his advisors and
committee members who were extremely helpful throughout the process.

8 Links

- Thesis Source Code: [https://github.com/chrisdonahue](https://github.com/chrisdonahue)
- Open BEAGLE Homepage: [http://code.google.com/p/beagle/](http://code.google.com/p/beagle/)
- KISS FFT Homepage: [http://sourceforge.net/projects/kissfft/](http://sourceforge.net/projects/kissfft/)
A Synthax Primitives

Any primitive that ends with a “*” implies that it functions the same as the non-“*” terminal version of the primitive except that it applies its envelope to a child signal. For example, the \texttt{adsr*} primitive produces an ADSR envelope which is then multiplied by its child’s signal. These primitives exist as a compressed representation of multiplying a signal by an envelope, an extremely useful operation to perform in digital audio synthesis.

- \texttt{adsr}
  - Description: Generates linear attack-decay-sustain-release envelopes of various shapes and sizes
  - Parameters:
    1. delay before attack in seconds
    2. attack in seconds
    3. attack final amplitude
    4. decay in seconds
    5. sustain in seconds
    6. sustain amplitude
    7. release in seconds
  - Children: 0 if \texttt{adsr}, 1 if \texttt{adsr*}

- \texttt{sinoscb_s}
  - Description: Generates a sine wave numerically (not using a wave table)
  - Parameters:
    1. input variable number used as center frequency \( f \) in Hz
    2. multiplier \( p \) for center frequency (partial number)
    3. phase value

- \texttt{const}
  - Description: Represents a constant numerical value
  - Parameters:
    1. constant value

- \texttt{val_s}
  - Description: Outputs a static input value (usually the desired center frequency)
  - Parameters:
    1. input variable number
    2. range of input value (e.g. \{c 0 0 22050\} if it’s a center frequency at a sample rate of 44100)

- \texttt{val_v}
  - Description: Outputs a variable input value (usually the desired center frequency)
  - Parameters:
    1. input variable number
    2. range of input value (e.g. \{c 0 0 22050\} if it’s a center frequency at a sample rate of 44100)

- \texttt{lfo}
  - Description: Acts as a sine wave low frequency oscillator
  - Parameters:
    1. frequency of the LFO in Hertz
  - Children: 0 if \texttt{lfo}, 1 if \texttt{lfo*}

- \texttt{pi}
  - Description: Represents the numerical value \( \pi \)
• **spline**
  - Description: Generates a piece-wise linear envelope using a variable number of points
  - Parameters:
    1. type of spline (currently only linear supported)
    2. number of control points \( n \)
    3. amplitude range
    4. segment length range in seconds
    5. \( n \) pairs of two parameters where the first represents an amplitude and the second indicates the length of time to get to the next amplitude
    6. final resting amplitude of the envelope
  - Children: 0 if **spline**, 1 if **spline**

• **silence**
  - Description: Generates silence

• **sawosc**
  - Description: Generates a sawtooth wave using a wave table
  - Parameters:
    1. input variable number used as center frequency \( f \) in Hz
    2. multiplier \( p \) for center frequency (partial number)
    3. phase value

• **sinosc**
  - Description: Generates a sine wave using an extremely simple wave table
  - Parameters:
    1. input variable number used as center frequency \( f \) in Hz
    2. multiplier \( p \) for center frequency (partial number)
    3. phase value

• **squareosc**
  - Description: Generates a square wave using a wave table
  - Parameters:
    1. input variable number used as center frequency \( f \) in Hz
    2. multiplier \( p \) for center frequency (partial number)
    3. phase value

• **triangleosc**
  - Description: Generates a triangle wave using a wave table
  - Parameters:
    1. input variable number used as center frequency \( f \) in Hz
    2. multiplier \( p \) for center frequency (partial number)
    3. phase value

• **time**
  - Description: Outputs the current time value in seconds

• **am**
  - Description: Performs a type of amplitude modulation on a sine wave. The primitive encapsulates the following expression, modified from [22]:
    \[
    [\omega + \alpha \cdot a_m(t)] \cdot \sin(2\pi fp t)
    \]
  - Parameters:
    1. input variable number used as center frequency \( f \) in Hz
2. multiplier $p$ for center frequency (partial number)
3. offset value $\omega$
4. modulation index $\alpha$
   - Children:
     1. amplitude modulation signal $a_m(t)$

• $fm_i$
   - Description: Performs a type of frequency modulation defined in the following expression using an index of modulation controlled by a child primitive:

   $$\cos((2\pi pf t) + (i(t) \cdot 2\pi r f t))$$ (5)

   - Parameters:
     1. input variable number used as center frequency $f$ in Hz
     2. multiplier $p$ for center frequency (partial number)
     3. ratio of modulator to carrier $r$
     4. center value of index of modulation
     5. radius of deviation from index of modulation center
   - Children:
     1. index of modulation signal, this primitive uses this child’s amplitude min and max to map its output values to the radius defined by parameters 3 and 4 for $i(t)$

• $pm$
   - Description: Performs a type of phase modulation on a sine wave. The primitive encapsulates the following expression, modified from [2]:

   $$\sin((2\pi pf t) + (i \cdot p_m(t)))$$ (6)

   - Parameters:
     1. input variable number used as center frequency $f$ in Hz
     2. multiplier $p$ for center frequency (partial number)
     3. modulation index $i$
   - Children:
     1. phase modulation signal $p_m(t)$

• $gain$
   - Description: Multiplies a signal by a gain value, another name for $const^*$
   - Parameters:
     1. gain value
   - Children:
     1. signal to apply gain to

• $sin$
   - Description: Applies the sine function to signal 1 (treated as radians)
   - Children:
     1. signal 1

• $cos$
   - Description: Applies the cosine function to signal 1 (treated as radians)
   - Children:
     1. signal 1

• $+$
   - Description: Adds two signals together
   - Children:
     1. signal 1
2. signal 2

- Description: Multiplies two signals together
  - Children:
    1. signal 1
    2. signal 2

- Description: Subtracts signal 2 from signal 1
  - Children:
    1. signal 1
    2. signal 2

- switch
  - Description: Audio logic primitive. Passes signal 1 if control signal is less than or equal to 0, otherwise passes signal 2
    - Children:
      1. control signal
      2. signal 1
      3. signal 2

B Experiment List for Human Study

B.1 Synthesized Piano $C_4$

This sound was recorded from a keyboard synthesizer by Garcia and used as an example in his doctoral thesis [7]. I recorded this sound as 16-bit floating-point audio at a rate of 44,100 hZ from a video on Garcia’s website [6]. The following is a list of experiments with different parameter combinations used to create the sounds in the human study of this sample.

1. Direct comparison to Garcia thesis [7]
   - Population Size: 60
   - # Generations: 200
   - Fitness Function: Frequency-Time Squared Error
   - Sub-optimization Method: CMA-ES (Population Size: 10, # Generations: 4)
   - Max Initial Height: 10
   - Max Height: 25
   - Inputs: Target Center Frequency, Target Envelope
   - Proportion of Population from Mutation: 0.6
   - Proportion of Population from Crossover: 0.2
   - Proportion of Population from Reproduction: 0.2
   - Proportion of Population from New: 0.0
   - Primitives:
B.1 Synthesized Piano $C_4$

$\left( + \text{ (null)} \text{ (null)} \right)$

$\left( \ast \text{ (null)} \text{ (null)} \right)$

$\left( \text{const \{c } -1.0 \text{ 0.0 1.0\}} \right)$

$\left( \text{gain \{c 0.0 0.0 1.0\} (null)} \right)$

$\left( \text{sinosc}_{b} \{d 0 0 0\} \{c 0.5 1.0 10.0\} \{c 0.0 0.0 1.0\} \right)$

$\left( \text{am} \{d 0 0 0\} \{d 1 1 10\} \{c 0 0 1.0\} \{c 0 0 1.0\} \text{ (null)} \right)$

$\left( \text{pm} \{d 0 0 0\} \{d 1 1 10\} \{c 0 1.0 5.0\} \text{ (null)} \right)$

$\left( \text{val}_{v} \{d 0 0 0\} \{c 0.0 0.0 1.0\} \right)$

2. Same as #1 except:
   - Sub-optimization Method: Random Parameters ($N = 40$)

3. Same as #1 except:
   - Population Size: 2000
   - # Generations: 240
   - Sub-optimization Method: None

4. Same as #1 except:
   - Population Size: 500
   - # Generations: 24

5. Same as #1 except:
   - Primitives:
     $\left( + \text{ (null)} \text{ (null)} \right)$
     $\left( \ast \text{ (null)} \text{ (null)} \right)$
     $\left( \text{const \{c } -1.0 \text{ 0.0 1.0\}} \right)$
     $\left( \text{gain \{c 0.0 0.0 1.0\} (null)} \right)$
     $\left( \text{sinosc}_{b} \{d 0 0 0\} \{c 0.5 1.0 10.0\} \{c 0.0 0.0 1.0\} \right)$
     $\left( \text{fm}_{i} \{d 0 0 0\} \{c 0.5 1.0 2.0\} \{c 0.0 0.0 3.0\} \{c 0.0 0.0 3.0\} \{c 0.0 0.0 0.5\} \text{ (null)} \right)$
     $\left( \text{lfo} \{c 0 0.0 4.0\} \right)$
     $\left( \text{val}_{v} \{d 0 0 0\} \{c 0.0 0.0 1.0\} \right)$

6. Same as #1 except:
   - Fitness Function: Frequency-Time Perceptual Error Weighting

7. Same as #6 except:
   - Population Size: 500
   - # Generations: 24

8. Same as #6 except:
   - Population Size: 2000
   - # Generations: 240
   - Sub-optimization Method: None
B.2 Meta Rediscovery Test

This sound was rendered from a *synthax* program as a test of whether or not the genetic programming system could find a synthesis algorithm that was definitely within its search space. The program used to generate the sound is an arithmetical manipulation of three sine wave generator nodes and is displayed below.

\[
(* \text{ (const } \{ c -1 0.849 1 \}) \) \text{ (+ (sinosc } \{ d 0 0 0 \} \{ c 0.5 2.308 10 \} \{ c 0 0.349 1 \}) \text{ (+ (sinosc } \{ d 0 0 0 \} \{ c 0.5 1.139 10 \} \{ c 0 0.049 1 \}) \text{ (sinosc } \{ d 0 0 0 \} \{ c 0.5 5.766 10 \} \{ c 0 0.042 1 \}))))
\]

The following is a list of experiments with different parameter combinations used to create the sounds in the human study of this sample.

1. Pure genetic programming, no sub-optimization
   - Population Size: 500
   - # Generations: 50
   - Fitness Function: Frequency-Time Squared Error
   - Sub-optimization Method: None
   - Max Initial Height: 5
   - Max Height: 8
   - Inputs: Target Center Frequency
   - Proportion of Population from Mutation: $\frac{1}{3}$
   - Proportion of Population from Crossover: $\frac{1}{3}$
   - Proportion of Population from Reproduction: $\frac{1}{6}$
   - Proportion of Population from New: $\frac{1}{6}$
   - Primitives:
     (+ (null) (null))
     (* (null) (null))
     (const \{ c -1.0 0.0 1.0 \})
     (gain \{ c 0.0 0.0 1.0 \} (null))
     (sinosc b s \{ d 0 0 0 \} \{ c 0.5 1.0 10.0 \} \{ c 0.0 0.0 1.0 \})

2. Same as #1 except:
   - Population Size: 4000
   - # Generations: 200

3. Same as #1 except:
   - Sub-optimization Method: CMA-ES (Population Size: 8, # Generations: 4)

4. Same as #3 except:
   - Fitness Function: Frequency-Time Perceptual Error Weighting

5. Same as #4 except:
   - Population Size: 4000
   - # Generations: 200

6. Same as #4 except:
   - Sub-optimization Method: Random Parameters ($N = 32$)
C  Online Appendix

This appendix serves as a pointer to an online appendix where a variety of data that could not be included in this paper is stored. This data includes audio files, visualizations of audio files and long synthax programs. To see and hear this information, please visit the “thesis” section of the evosynth website at \url{http://www.evosynth.tk/thesis/}.

C.1 A:  Synthax Programs for Mimicking Specific Timbres

\url{http://evosynth.tk/thesis/#A}

This online appendix stores results of various target mimicking experiments as described in Section 3 on a range of different sounds. Unlike those listed in Appendix B.1, the amplitude envelope of the target was not an input for these experiments. Any amplitude enveloping had to be done in the synthax program using primitives such as spline and gain. This allows for more portability of the synthesis algorithm without requiring the target envelope as input but increases the complexity of the search problem. All of the programs for mimicking specific sounds listed in the online appendix were produced by genetic programming with the following set of parameters:

- Population Size: 200
- # Generations: 60
- Fitness Function: Frequency-Time Squared Error
- Sub-optimization Method: CMA-ES (Population Size: 10, # Generations: 4)
- Max Initial Height: 8
- Max Height: 20
- Inputs: Target Center Frequency
- Proportion of Population from Mutation: 0.6
- Proportion of Population from Crossover: 0.2
- Proportion of Population from Reproduction: 0.2
- Proportion of Population from New: 0.0
- Primitives:

\[
\begin{align*}
&\text{(+ (null) (null))} \\
&\text{(* (null) (null))} \\
&\text{(const \{c -1.0 0.0 1.0\})} \\
&\text{(gain \{c 0.0 0.0 1.0\} (null))} \\
&\text{(sinoscb_s \{d 0 0 0\} \{c 0.5 1.0 10.0\} \{c 0.0 0.0 1.0\})} \\
&\text{(am \{d 0 0 0\} \{d 1 1 10\} \{c 0 0 1.0\} \{c 0 0 1.0\} (null))} \\
&\text{(pm \{d 0 0 0\} \{d 1 1 10\} \{c 0 1.0 5.0\} (null))} \\
&\text{(spline* \{d 0 0 0\} \{d 0 0 5\} \{c 0 0 1.0\} \{c 0 0 0.3\} (null))}
\end{align*}
\]
C.2  B: Symbolic Expression of Audio Synthesis Algorithms (expanded with Audio Examples)

This online appendix stores data useful for understanding how rendering and numerical parameters work in *synthax*.

C.3  C: Human Study Data

This online appendix stores data from various experiments used for the two human studies.
References


