A DISSERTATION
SUBMITTED IN FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF DOCTOR OF PHILOSOPHY
IN COMPUTER SCIENCE AND ENGINEERING

Applications of Evolutionary Algorithms to Digital Audio Signal Processing

by

Edward Sam Ly

March 2024

The thesis titled

Applications of Evolutionary Algorithms to Digital Audio Signal Processing

by

Edward Sam Ly

is reviewed and approved by:

Chief referee

Senior Associate Professor Date

Julián Alberto Villegas Orozco

Professor Date

Michael Cohen

Professor Date

Qiangfu Zhao

Professor Date

Konstantin Markov

THE UNIVERSITY OF AIZU

March 2024

Contents

List of Figures

List of Tables

List of Abbreviations

AI	Artificial Intelligence
ANN	Artificial Neural Network
ANOVA	Analysis of Variance
API	Application Programming Interface
AU	Audio Units
BR	Bass Ratio
CGP	Cartesian Genetic Programming
CPU	Central Processing Unit
dBFS	Decibels Relative to Full Scale
DC	Direct Current
DE	Differential Evolution
DF	Degree of Freedom
DFT	Discrete Fourier Transform
DSP	Digital Signal Processing
EA	Evolutionary Algorithm
EC	Evolutionary Computation
EDT	Early Decay Time
EMM	Estimated Marginal Mean
EP	Evolutionary Programming
ERC	Ephemeral Random Constant
ES	Evolutionary Strategy
FAUST	Functional Audio Stream
FIR	Finite Impulse Response
FLAC	Free Lossless Audio Codec
FM	Frequency Modulation
FSA	Finite-State Automation
GA	Genetic Algorithm
GLMM	Generalized Linear Mixed Model
GNU	GNU's Not Unix!
GP	Genetic Programming
GPLv3	GNU General Public License Version 3
GPU	Graphics Processing Unit
GUI	Graphical User Interface
HRIR	Head-Related Impulse Response
HSD	Honest Significant Difference
IDFT	Inverse Discrete Fourier Transform
IIR	Infinite Impulse Response

ILD	Interaural Level Difference
IQR	Interquartile Range
IR	impulse response
ISO	International Organization for Standardization
ITD	Interaural Time Difference
ITDG	Initial Time Delay Gap
JND	Just Noticeable Difference
LGPLv3	GNU Lesser General Public License Version 3
LISP	List Processor
LLVM	Low Level Virtual Machine
LMEM	Linear Mixed-Effects Model
LSD	Log-Spectral Distance/Distortion
MATLAB	Matrix Laboratory
MFCC	Mel-Frequency Cepstral Coefficient
MIDI	Musical Instrument Digital Interface
ML	Machine Learning
MP-GEP	Multi-Population Gene Expression Programming
MT-CGP	Mixed-Type Cartesian Genetic Programming
PBE	Programming By Example
Pd	Pure Data
PPT	Probabilistic Prototype Tree
RAM	Random Access Memory
RCGP	Recurrent Cartesian Genetic Programming
RIR	Room Impulse Response
SHARC	Sandell Harmonic Archive
SISO	Single-Input/Single-Output
SLF	Sonic Localization Field
SNR	Signal-to-Noise Ratio
UoA	University of Aizu
VST	Virtual Studio Technology
WAV	Waveform Audio File Format

List of Symbols

$\pm\infty$	Plus-minus infinity
$\mathbb{N}$	Set of natural numbers (ISO 80000-2 standard)
NaN	Not-A-Number
$\mathbb{R}$	Set of real numbers
$\mathbb{S}$	Set of signals
$\mathbb{Z}$	Set of integers

Acknowledgment

My biggest thanks goes to my supervisor, Prof. Julián Villegas, for his patience, guidance, and encouragement throughout my five years as a student at the University of Aizu. This dissertation would not be possible without his immense help and support.

I would also like to thank Prof. Qiangfu Zhao for introducing me to the field of evolutionary computation via his CSC06A "Introduction to Meta-heuristics" course, as well as taking the time to be a part of the review committee throughout the development of my research. Additional thanks goes to Profs. Michael Cohen and Konstantin Markov for their valuable comments and suggestions as part of the review committee as well.

Finally, I would like to thank my parents for providing me the opportunity to travel overseas to pursue this unique education. This entire endeavor was first and foremost a product of their financial and motivational support, as well as their belief in my fruitful career ahead and the opportunities that are now possible as a result of this degree.

Abstract

With Artificial Neural Networks (ANNs) being at the forefront of Artificial Intelligence (AI) research in recent decades, other Machine Learning (ML) models within the larger field of AI can sometimes be overlooked as viable alternatives for solving complex problems in many fields. This dissertation examines some possible use cases of Evolutionary Algorithms (EAs) by applying them to audio Digital Signal Processing (DSP) problems in particular, including those that have previously seen many proposed solutions using ML. We start with the creation of Room Impulse Responses (RIRs) that can be used within a real-time convolution reverb audio effect plugin. Using an Evolutionary Programming (EP) approach, a user is able to gain some control over the shape of the resulting RIRs through various parameters defined in the ISO 3382-1 standard (e.g., reverberation time, early decay time, and clarity), the values of which determine the fitness of potential RIRs. The resulting rooms can range from those whose RIRs can be recorded in the real world, to virtual spaces that may be physically impossible to represent. Results from a subjective evaluation show that such perceptual differences were reduced when the EA was executed for a sufficient number of generations, or when the input audio signals consisted of only speech. We then proceed to a more general case where entire software synthesizers are generated via Recurrent Cartesian Genetic Programming (RCGP): given a target audio signal, DSP programs expressed as directed cyclic graph structures are evolved to generate an approximation of the target audio with arbitrary accuracy. The candidate program with the smallest error, using a fitness function based on Mel-Frequency Cepstral Coefficients (MFCCs) quantifying perceptual differences, is returned as the best fit solution. After an experiment evaluating the effects of several RCGP parameters, we determined that a classical Cartesian Genetic Programming (CGP) model with a weighted function set that accounts for prior knowledge was able to generate steady state signals with minimal error. This method was eventually generalized even further to include the generation of DSP audio effect chains, with the Log-Spectral Distance (LSD) being used as a fitness metric to compare the frequency spectra of their respective impulse responses. We then evaluated our method by generating various Infinite Impulse Response (IIR) filters, with the accuracy of these filters being dependent on the order of the target filters to be replicated. Working prototype implementations for these methods are publicly available as free software for demonstration purposes.

概要

人工神経回路網（ANN）がここ数十年の人工知能（AI）研究の最前線にあるため、AI という大きな分野の中で他の手法が、多くの機械学習（ML）技法で複雑な問題を解決するための有効な選択肢として見過ごされてしまうことがある。本論文では、進化的アルゴリズム（EA）を、以前に ML で解決されたオーディオのデジタル信号処理（DSP）の問題に適用することで、EA の可能な利用シーンを検討する。まず、コンボリューションリバーブをリアルタイムで実行するプラグイン内で使用できる部屋のインパルス応答（RIR）を生成する。進化的プログラミング（EP）を用いて、ISO 3382-1 規格で定義された様々なパラメータ（残響時間、早期減衰時間、透明度など）によってコスト関数が決定される RIR が進化する。その結果、RIR を実世界で記録できる部屋から、物理的に表現できないような仮想空間まで、さまざまな部屋を作成できるようになる。主観評価の結果、EA を十分な世代数で実行した場合や、信号が音声のみの場合、このような知覚差が減少した。次に、リカレントカルテジアン遺伝的プログラミング（RCGP）を用いて、ソフトウェアシンセサイザを生成する一般的な利用シーンに進む。目標信号を入力すると、その信号を近似するために、有向巡回グラフ構造で表現した DSP プログラムが進化する。知覚差を定量化するメル周波数ケプストラム係数（MFCC）に基づくコスト関数を用いて、誤差が最小となる候補プログラムを最適解として出力する。いくつかの RCGP パラメータの効果を評価する実験を行った結果、事前知識を考慮した非巡回カルテジアン遺伝的プログラミング（CGP）を用いて、最小限の誤差で周期信号を生成できる。さらに、デジタル信号に変換を施す機能の生成を含むように一般化されて、インパルス応答の周波数分布を比較するためのコスト関数としてログスペクトル距離（LSD）を使用する。無限インパルス応答（IIR）濾波器を生成して、本手法を評価した。その結果、この濾波器の精度は目標濾波器の次数によって違った。これらの手法をデモするための実用的なプロトタイプの実装は、フリーソフトウェアとして公開されている。

Chapter 1

Introduction

1.1 Background

In recent years, Artificial Intelligence (AI) has been the backbone of many state-of-the-art solutions for numerous problems in just about every research field. In particular, Machine Learning (ML) techniques and Artificial Neural Networks (ANNs), computational structures modeled after the neurons in the human brain, have seen the most widespread adoption since the popularization of the back-propagation algorithm in the mid-1980s [1]. Digital Signal Processing (DSP) research is no exception in this regard, with various types of ANNs being used in, for instance, the generation of synthesized audio signals via parameter estimation [2] or waveform manipulation [3]. As a result, other techniques in AI can sometimes be overlooked as viable alternatives for solving such problems.

In this dissertation, we introduce one such sub-field, namely Evolutionary Algorithms (EAs), and explore their potential use cases in the creation of DSP programs, especially those that have previously seen many proposed solutions using ML. EAs are a family of optimization algorithms modeled after Darwin's theory of evolution [4]. As such, their applications in many research fields are still yet to be fully explored. This dissertation details our novel contributions in regards to DSP applications in particular, as well as discuss the potential and unique advantages EAs have over ML techniques in

this field.

1.2 Overview

In the next chapter, we give a brief overview of EAs as well as several variations to the algorithm that are relevant to our discussion, which includes Genetic Programming (GP) in particular and some of its variations, eventually ending with Recurrent Cartesian Genetic Programming (RCGP). The following chapters discuss how such EAs can be used to generate DSP programs, with chapter 3 detailing the generation of Room Impulse Responses (RIRs) using an Evolutionary Programming (EP) approach, while chapters 4 and 6 detail the generation of generalized DSP programs with RCGP, including both synthesizers and audio effect chains, respectively. The source code for these projects is publicly available in [5] and [6]. Chapter 5 then discusses the effects of various RCGP hyper-parameters on the output DSP programs. Finally, chapters 7 and 8 discuss the overall results and make concluding remarks, respectively.

Chapter 2

Overview of Evolutionary Algorithms

2.1 Types of Evolutionary Algorithms

Within AI, there is a sub-field known as Evolutionary Computation (EC), which consists of a collection of EAs [7] inspired by Darwin's theory of evolution [4] for global optimization problems. Evolutionary Algorithms are multi-point meta-heuristic optimization algorithms in that they take a "population" of "individuals" representing potential solutions, and evolve them using various evolutionary operations, including selection, crossover, and mutation, in order to find better solutions. Each individual is also assigned a numerical "fitness value" to serve as a metric of its "goodness" relative to the other individuals. These fitness values determine which individuals are selected, or "reproduced," as a basis for creating new "offspring" solutions, which are often created by combining (or "crossing over") the various characteristics from two or more selected individuals. Afterwards, each individual can be modified even further through mutation in order to increase the "diversity," or genetic makeup, of the solutions in the population, and avoid convergence towards local minima or maxima when other solutions with better fit may exist elsewhere in the solution space. Each selection, crossover, and mutation operation constitutes a "generation" in time, and can be repeated for as long as needed until a termination condition is satisfied, often due to either a limit on the number of generations to execute or the fitness of a candidate solution is deemed

"good enough" to be returned by the algorithm. A summary of this process is visualized in Figure 2.1.

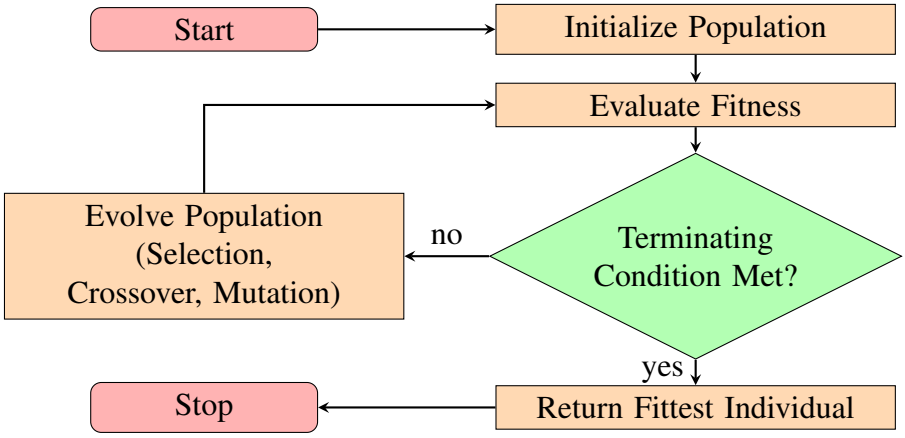


Figure 2.1: Flowchart summarizing the basic procedure for any EA.

Furthermore, these evolutionary algorithms can be divided into various categories according to the "genomes" they use, or the various ways that these solutions can be encoded or represented, and subsequently evolved. The most common types of EAs are summarized in Table 2.1.

Table 2.1: Types of EAs with their respective solution representations.

EA Type	**Genome**
Genetic Algorithm (GA)	sequence of binary digits
Differential Evolution (DE)	vector of real numbers
Evolutionary Programming (EP)	set of program parameters
Genetic Programming (GP)	the programs themselves

In this following sections, we will introduce in more detail GP in particular, as up until now, it has seen relatively little usage for solving problems in DSP research compared to other techniques such as EP.

2.2 Genetic Programming

GP, the idea of modifying computer programs through an evolutionary process, was first introduced by Koza [8]. He evolved a population of programs in the LISP programming language by encoding them as tree structures. Such structures were a

natural choice due to LISP's functional programming paradigm as well as the parenthetical notation of LISP's S-expressions, but almost any computer program, regardless of language, can be encoded in the same way. Mathematical expressions, for example, contain an operator function whose arguments can consist of both operands (i.e. variables and constants) as well as other operator functions. Figure 2.2 shows how a polynomial such as x^2-2x+1 can be encoded as a tree structure, of which an equivalent S-expression in LISP can be written as `(+ (- (* x x) (* 2 x)) 1)`.

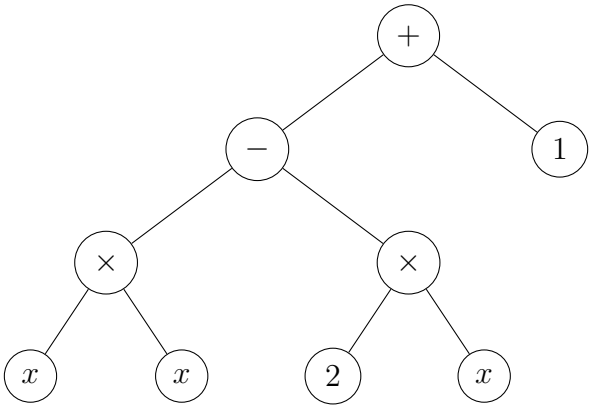


Figure 2.2: A tree structure representing the expression $x^2 - 2x + 1$.

In GP, various evolutionary operations can be performed on these tree structures in order to find the desired expressions. For example, the crossover operation may take a random sub-tree from one parent and replace it with a random sub-tree from another parent, or the mutation operation may replace an individual's sub-tree with a new random sub-tree entirely. Koza successfully used these methods in order to generate Boolean multiplexer functions that adhere to arbitrary truth tables, among many other applications.

A disadvantage to using tree structures, however, is that such representations can be inefficient as programs become more complex or make use of other functions as sub-trees. For instance, Figure 2.2 contains three nodes which all contain the same redundant variable x, but a single node cannot be reused as an argument to multiple different sub-branches. In general, the output of any sub-tree (function or operand) can only be used as an argument to another function at most once in any tree structure, forcing sub-trees to be duplicated each time they are used. Another example in DSP

is a feed-forward loop such as a comb filter, where a delayed audio signal is added to itself. In a tree structure, such a filter requires two "input signal" nodes when there ideally could be just one.

2.3 Cartesian Genetic Programming

Such limitations of tree-based GP would later be addressed thanks to the introduction of Cartesian Genetic Programming (CGP), which was first used by Miller [9] and then later proposed as its own paradigm by Miller and Thomson [10]. In CGP, computer programs are encoded as directed acyclic graphs instead, with the nodes being classified into three different types: input nodes for the program inputs, function nodes containing function primitives as in tree-based GP, and output nodes for the program outputs. The nodes are laid out in that order in an array, with all of the edge connections moving in a forward direction. An example chromosome provided by Turner and Miller [11] is reproduced in Figure 2.3.

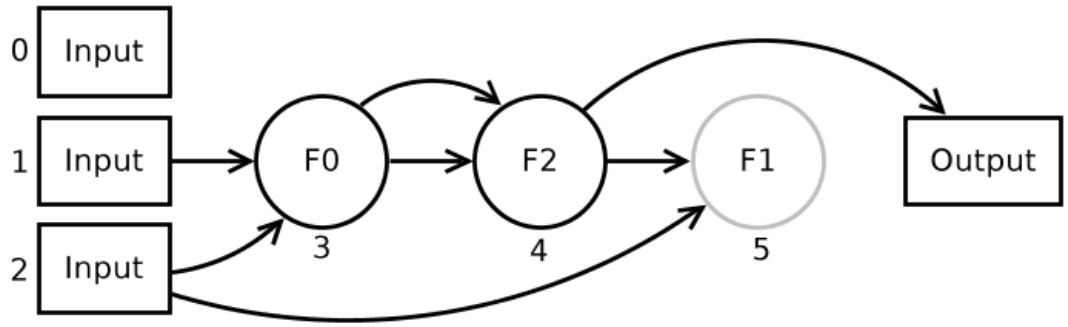


Figure 2.3: An example program encoded as a graph for evolution via CGP.

Note that not all input nodes and function nodes necessarily have to contribute to the outputs of the program, as is the case for node 5 in Figure 2.3. The fact that certain nodes can be "active" or "inactive" makes CGP an EA with a genotype-phenotype distinction, the genotypes in this case being the entire hereditary information of each individual, and the phenotypes being the expressed traits of each program through the active nodes. This is regarded as an advantage in the evolutionary search, as the resulting "neutral genetic drift" allows for the creation of offspring with differing genotypes but identical

phenotypes [11, 12]. Other advantages of CGP over tree-based GP include the reuse of data and function outputs, as well as the elimination of "bloat," a common phenomenon in tree-based GP where the programs in the population get larger over time without any apparent benefits in fitness [12–15].

Another trait of CGP that distinguishes it from other EAs is the use of a $(\mu + \lambda)$-Evolutionary Strategy (ES), where the μ best-fit individuals in the population are chosen as the parents for the next generation (i.e., elitist selection), and λ offspring individuals are generated via mutation-only reproduction, for a total population size of $\mu + \lambda$. Miller [9] found that small population sizes performed best for Boolean function learning problems, with $\mu = 1$ and $\lambda = 4$ being the most common values for many subsequent applications of CGP. He also determined that a crossover operation is not a necessary step in the evolutionary search, and in fact can even hinder the search in some cases.

2.4 Recurrent Cartesian Genetic Programming

In their paper introducing CGP, Miller and Thomson [10] had yet to explore the possibility of allowing the graphs to be cyclic. Such an extension of CGP where the graphs are allowed to be cyclic would later be introduced as RCGP by Turner and Miller [11]. This technique includes the addition of a "recurrent connection probability" parameter that controls the likelihood of recurrent connections, edges that connect from one node to itself or to a previous node in the chromosome, being created via mutation. Often, this can lead to the creation of feedback loops, but may not always be the case depending on the arities (the number of arguments) of the relevant nodes. Another example chromosome from Turner and Miller [11] that contains such a feedback loop is reproduced in Figure 2.4.

Turner and Miller also demonstrated that RCGP provides better performance compared to standard, acyclic CGP on two benchmark problems: time series prediction with sunspot data [16], and Artificial Ants, a Finite-State Automation (FSA) problem which

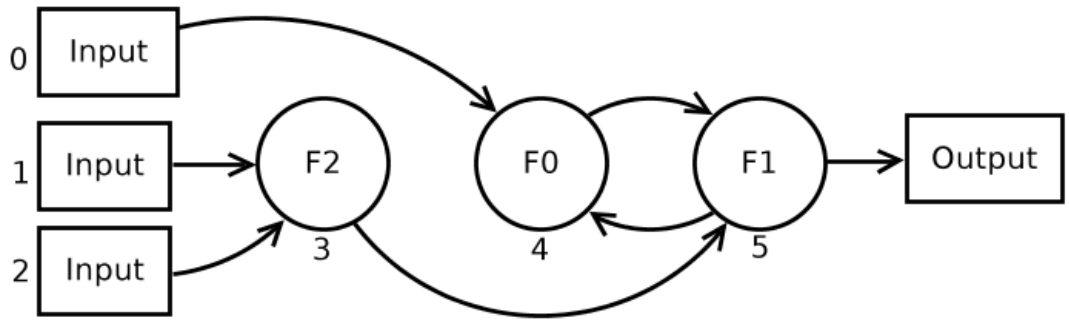


Figure 2.4: An example program encoded as a cyclic graph for evolution via RCGP.

was first proposed and tackled by Jefferson et al. [17] using ANNs and later tackled by Koza [8] using GP.

2.5 Evolutionary Algorithms in Sound and Audio

The idea of applying evolutive techniques to real-time DSP overall is not new (see, for example, [18]), but compared to other techniques in AI, there seems to be little research and few actual implementations so far. Collins [19] attempted to facilitate this by providing a software library built in SuperCollider [20] that can apply EPs to both sound synthesis and real-time DSP effects, including filtering and reverberation. Using SuperCollider's built-in Graphical User Interface (GUI) and programming tools, a framework for other developers to build software applications using EPs was provided. Still, Collins does not mention any ability to evolve a population of impulse responses (IRs) themselves for convolution reverberation, but rather other devices for reverberation techniques, such as multi-tap delay lines, feedback delay networks, and Schroeder reverberators, and modifying their parameters through EP. In addition, the SuperCollider environment requires the use of `scsynth`, a dedicated audio server for real-time DSP computation, which allows for SuperCollider-based systems to be more modular, but adds unnecessary network latency to and reduces the portability of such systems. As of June 2023, a *SuperColliderAU* wrapper is available to embed `scsynth` into a Audio Units (AU) plugin, but this technology is only available in the MacOS operating system. As for Virtual Studio Technology (VST) plugins, the language currently sup-

ports the ability to host and automate the parameters of existing plugins in an `scsynth` server, but not to export the entire project as a VST plugin itself.

Additionally, an EP approach for sound localization and spatialization has been proposed by Fornari et al. [21, 22], with implementations in both Pure Data (Pd) [23] and MATLAB [24] programming languages. With their EA, they first calculate the Interaural Time Difference (ITD) given sound intensity and azimuth angle as input parameters. Then, with this ITD, a population of sound mark cues in a Sonic Localization Field (SLF) with which sound can be perceived at particular locations in space are evolved and used to generate spatial sound.

DSP applications for sound spatialization are, in general, limited because a free-field is assumed; i.e., no reverberation is considered. In many cases, an audio signal is directly convolved with a Head-Related Impulse Response (HRIR) that captures only the transformations caused by the upper body of a listener to a sound in the absence of an enclosure (e.g., [25]). However, these applications can benefit from artificial reverberation since, when the direct sound is combined with reverberation or at least with late reflections (the tail of the reverberation), the externalization of audio sources seems to improve [26].

As for GP-based approaches, the only other known attempt comes from Macret and Pasquier [27], who represented DSP synthesizers as Pd patches and evolved them via Mixed-Type Cartesian Genetic Programming (MT-CGP), an extension of CGP where the inputs and outputs of the node functions can be limited to certain data types. This CGP variant has two distinct disadvantages: the first is that, without the possibility of creating recurrent connections, feedback loops cannot be introduced into the DSP graphs, and thus such programs are left out of the solution space. We believe RCGP is a more ideal framework for evolving these DSP programs because of the allowance of recurrent connections, since feedback loops are a common occurence in DSP for synthesizing and manipulating audio signals, such as in sawtooth oscillators or in Infinite Impulse Response (IIR) filters.

The second is that the introduction of multiple data types is merely a consequence

of Pd's distinction between audio and control signals, as well as objects that may accept as inputs one data type or the other (as of version 0.52-2). This means that extra computation is required to validate the inputs of each Pd object, and arbitrarily limits the solution space even further. For this reason, we believe that the FAUST [28] programming language is a better alternative for DSP programming as well. Although both FAUST and Pd use a block diagram-based paradigm for DSP development, FAUST is a functional, platform-independent DSP programming language that can be "translated" into a wide variety of other programming languages and compiled for various applications and frameworks, including Pd. From Chapter 4, we will explain how both RCGP and FAUST can be used to evolve and create just about any DSP program imaginable.

Chapter 3

Evolutionary Synthesis of Room Impulse Responses

3.1 Introduction

For our first foray into using EAs for DSP applications, we introduce a method for creating RIRs using an EP approach. This method produces unique RIRs with each iteration while satisfying some constraints imposed by the end user. These RIRs can then be used to perform real-time convolution reverb in "Genetic Reverb," a custom software plugin developed in MATLAB using VST 2, a popular cross-platform audio software interface widely used in music production. We also investigate the suitability of this approach by measuring the elapsed time, fitness, and subjective accuracy of the IRs thus generated.

Parts of this chapter were first published in [29], then expanded upon in my University of Aizu (UoA) Master's thesis and later in [30] within the duration of the doctoral program. The latter contains additional background information, updates to the methodology and subsequent experimental results, and an additional (subjective) evaluation method.

3.2 Background

3.2.1 Overview of Reverberation

In most situations, sound is perceived within enclosures such as rooms, large or small venues, caves, etc. These enclosures imprint their characteristics (i.e., their transfer functions) on the sound depending on where the sound source and the listener are located within the enclosure. The direct sound travels the shortest path between the source and the listener, while other paths may include reflections off the boundaries of the enclosure. Hence, reflected sounds arrive later than the direct sound and are, in general, weaker because of the energy absorption of the reflecting surfaces as well as the transmission medium. The collection of early and late reflections is usually referred to as "reverberation." While this phenomenon is often detrimental to speech intelligibility [31], it is frequently desired in music production as an expressive tool [32].

The most direct form of including reverberation in audio recordings is by capturing the sound directly in spaces with the desired acoustic characteristics. Alternatively, one can record the room's characteristics, or its IR, with which a dry audio signal (one that is anechoic or contains a very short reverberation) can be convolved. However, recording (even just the IRs) in a majority of venues can be expensive and may not always be possible, so other methods to add artificial reverberation have been devised. Later in this section, a few of these techniques are described in more detail, but in any case, these methods yield reverberations similar to those captured in venues without apparent quality detriment. Regardless of the alternatives for adding reverberation, uniqueness is a characteristic often sought for artistic purposes. However, creating new and unique IRs with desired room characteristics can be a difficult task without correct or adequate methods.

When producing reverberation artificially, however, it is often difficult to reproduce some acoustic characteristics of physical spaces. Even when the IR of a desired room is known, synthesizing a similar IR is still an open challenge. In this section, we will explain how artificial reverberation can be applied to an audio signal in real-time using IRs

or other techniques. Then, we will provide a brief overview of previous research in EC for musical applications, as well as recent advances in digitally simulating reverberation via convolution with IRs.

3.2.2 Reverberation Techniques

Some of the most common techniques for digitally simulating reverberation involve the use of delay lines, comb filters, feedback delay networks, etc. Fundamentally, all of these DSP devices work similarly. A simple delay-line-based reverberation modifies an audio signal by adding a delayed version of the same signal to itself. The delayed version is also attenuated, simulating a single sound reflection. This delay can be repeated many times to construct a synthetic IR, but having to program every reflection can be very inefficient. Comb filters, in feed-forward form, work the same way, but a feedback component can be included as well. In this case, the output signal can be recursively fed back into the comb filter as an input signal. Given adequate parameters, a comb filter can produce an infinite number of decaying reflections, or a series of echoes, at regular time intervals. The first known implementation of reverberation in software was developed by Schroeder [33, 34], who used a combination of both feedback comb filters and all-pass filters to simulate wall reflections with exponential decay, a system that would later be known as Shroeder reverberators. Depending on the number of filters used, however, the resulting IR can be difficult to modify, as the parameters of each delay object only control either a single reflection or a series of related reflections. This makes granular modification of the IR to match certain room characteristics cumbersome at best.

Convolution reverb works differently in that instead of representing the IR of the desired room as a series of delay objects, the IR information is stored directly as audio, an array of samples in the time domain, with each sample value representing the gain and polarity of the sound pressure signal after some delay. Then, assuming that an audio signal and the IR are sampled at the same rate, the two signals can be combined (via the convolution operation) to produce a reverberated signal.

In order for this process to be implemented in real-time, however, two modifications to the convolution operation have been proposed by Zölzer [35]. First, the convolution operation is replaced by frequency-domain filtering (also called "fast convolution"). In this operation, an audio signal and the IR are transformed from the time domain into the frequency domain via the Discrete Fourier Transform (DFT), the two transformed signals are multiplied element-wise, and the result is transformed back into the time domain via the Inverse Discrete Fourier Transform (IDFT) to produce the output signal. Second is the use of "partitioned convolution," where the audio signal and the IR are first partitioned into smaller blocks before fast convolution is performed on each block individually [36]. Each of the reverberated blocks are then recombined to construct the output signal.

Regardless of the specific algorithms used, current solutions (often software plug-ins) that implement convolution reverb are limited by their selection of pre-loaded IRs or enclosures that a user can choose from. While the ability to modify the existing IRs may be offered by some plugins, a plugin in which new IRs can be generated from scratch was yet to be found or implemented. That is the void that the "Genetic Reverb" plugin presented in this chapter is seeking to fill.

3.2.3 Simulating the IR of a Box-Shaped Room

An image method for simulating the IR of a box-shaped room was originally proposed by Allen and Berkley [37]. In their method, such a room was projected onto three-dimensional space to determine which sound reflections contributed to an IR. The same projection was also used to calculate the effective distance and associated gain for each contributing reflection. As a metaphor, one can imagine a listener and a sound source located arbitrarily within a room with mirrors on all sides. Such mirrors would create an infinite number of images of the source outside the initial boundaries of the room, and the listener could see the source reflected an infinite number of times. Then, each source image would contribute a single sound reflection, and the sound from each object image would then be delayed by some time before it reaches the listener. Such

a delay can simply be calculated as the perceived distance between the listener and the image divided by the speed of sound. Additionally, each reflection will lose power depending on the number of "walls" the sound has to "pass through" before reaching the listener's location. Practically, reflections greater than a certain order are ignored in software implementations, as such reflections become too quiet to be perceived or numerically represented.

Many extensions and applications based on the Allen and Berkley method would later be proposed. One such implementation is provided by Habets [38] in MATLAB. More recent revisions to the image method that tackled these performance issues have also been proposed, such as those by Kristiansen et al. [39] and McGovern [40]. Kristiansen et al. use the image method to extend the length of an existing IR by using known lower-level reflections to calculate higher-level ones, while McGovern improves on the original image method algorithm by using look-up tables and sorting to prevent unnecessary calculations, greatly reducing computation time.

Initially, we implemented the fast image method proposed by McGovern as a baseline model for generating an initial population of IRs in the "Genetic Reverb" plugin. However, the EA ultimately outputs IRs that no longer represent box-shaped rooms regardless of the nature of the initial population, so there was no reason why other (non-box shaped) models could not be used in the first place either. Thus, a new method for generating IRs modeled after recorded IRs was devised for our solution. The following section describes our EA method in detail.

3.3 Methods

Typical EAs often evolve a population of individuals that can be encoded as a series of binary or real-numbered values. As digital audio is also stored as an array of real values, it seems natural to attempt to evolve IRs or even entire audio signals as well. This section details each step of our evolutionary process for generating IRs as well as some notes regarding the implementation of our algorithm in software.

3.3.1 Evolutionary Process

An EA was implemented on a population of room IRs, where the genome for each IR individual consists of an array of real-valued genes in the range of $[-1, 1]$ in the time domain. The first gene at the start of the IR is also assumed to be non-zero and to represent the first reflection. Multiple applications of the genetic operations will then change the amplitude of existing reflections until a created IR closely adheres to the user constraints. Figure 3.1 provides an overview of the evolutionary process as it pertains to our solution.

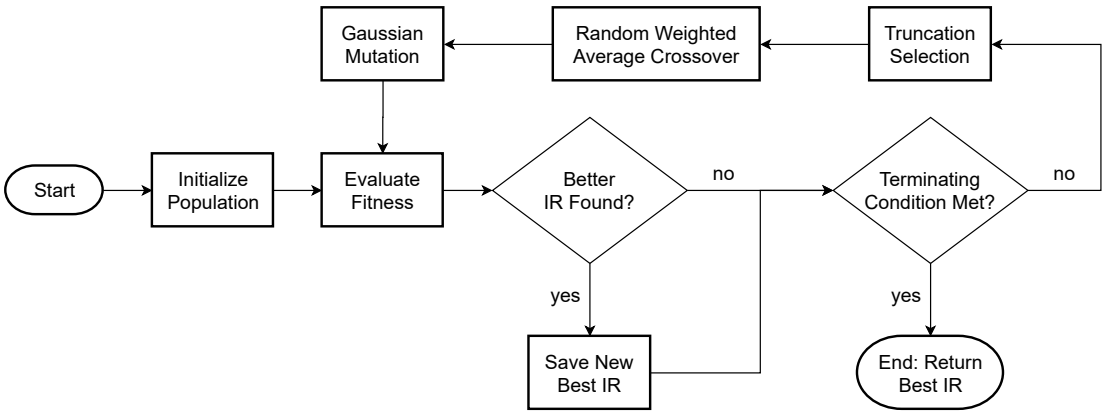


Figure 3.1: Flowchart outlining each step of the EA used to generate an IR in the Genetic Reverb plugin.

Five acoustic parameters that a user can impose on an IR in our solution are summarized in Table 3.1. The first three parameters (T_{60}, Early Decay Time (EDT), C_{80}) are defined under the ISO 3382-1 standard [41], while the other two parameters, Bass Ratio (BR) and Initial Time Delay Gap (ITDG), are among many additional parameters and attributes proposed by Beranek [42]. These parameters not only enable control over the overall shape of the IR, but are also parameters that a typical end user may understand and control.

The T_{60} parameter is perhaps the most important, since it specifies the overall length of the reverberation (i.e., how long the lingering sound is heard). In general, a long T_{60} correlates with large enclosures or enclosures with walls featuring high acoustic reflection coefficients. The ITDG is a similar measure for determining the "intimacy" of an enclosure, with longer ITDGs corresponding to wider or taller enclosures [42, pp.

Table 3.1: Acoustic parameters used in our solution, along with their colloquial names and definitions.

Parameter	**Common Name(s)**	**Definition**
T_{60}	Decay Time, Reverberation Time	Time for sound pressure level of the IR to decay by $60\,\mathrm{dB}$ from the initial magnitude
Early Decay Time (EDT)	Reverberance	Time for sound pressure level of the IR to decay by $10\,\mathrm{dB}$ from the initial magnitude
C_{80}	Clarity	Ratio (in dB) between sound pressure levels of direct sound plus early reflections ($< 80\,\mathrm{ms}$ after direct sound) vs. late reflections ($\geq 80\,\mathrm{ms}$ after direct sound)
ITDG	Predelay, Intimacy	Time between the arrival of the direct sound and the arrival of the first reflection.
Bass Ratio (BR)	Warmth	Ratio between T_{60} times in low-frequency (125 – $500\,\mathrm{Hz}$) and mid-frequency (0.5 – $2.0\,\mathrm{kHz}$) octave bands

513–516]. Meanwhile, the EDT characterizes the initial rate of decay of the IR, and it is generally considered more subjectively relevant than T_{60}, since the early reflections are more salient to the perception of reverberation than the late reflections [41].

Clarity (C_{80}) measures the relative "intelligibility" of the original audio source within its surrounding reverberations. High clarity indicates how "clearly" or "intelligible" a sound source can be heard. Generally, IRs with long T_{60} tend to have low C_{80} values. Finally, the BR or warmth parameter is a measure of the frequency content of an IR. Is is defined as the ratio of the reverberation times (RT) within four consecutive octave bands, with center frequencies at 125, 250, 500, and $1000\,\mathrm{Hz}$ [42, p. 512]:

$$BR = \frac{RT_{125\mathrm{Hz}} + RT_{250\mathrm{Hz}}}{RT_{500\mathrm{Hz}} + RT_{1000\mathrm{Hz}}}. \tag{3.1}$$

High BR values indicate that lower frequencies are more prominent in the reverberated sound compared to higher frequencies.

Initialization

In the initialization step, a population of IRs is generated using a custom Gaussian noise method based on the scheme proposed by Zölzer [35, p. 144]. This noise is typically present in observed IR recordings (mainly from the signal-to-noise ratio of the recording apparatus). The IR population is stored as a two-dimensional matrix, with each column representing an individual IR, and the number of columns is equal to the desired size of the IR population. The length of each column (i.e., the number of samples in the IR) is determined by the T_{60} reverberation time of the desired IR along with the sampling rate of the IR to generate. In our solution, the sampling rate was set to a constant $16\,\mathrm{kHz}$ in order to minimize computation time.

Within each IR, each sample is initialized as a random number from the standard normal distribution. Then, each IR is manipulated in such a way in an attempt to produce IRs that resemble actual IRs that have been recorded in the real world. This is done through the following steps:

1. Reduce the gain of the entire signal by a constant amount (a random factor between 0.2 and 0.7). This is to introduce some variation in regards to how much of the sound is absorbed by the enclosure.

2. Change the value of certain samples to be closer to ± 1. The probability that a sample is chosen at time t (in s) is

$$P(t) = 1 - p^t \tag{3.2}$$

   for some randomly chosen probability constant $p \in (0, 1)$, simulating the higher density of late reflections compared to that of early reflections. In addition, the new values of these samples are randomly generated from a normal distribution ($\mu = 1$, $\sigma = 0.05$), along with a $50\,\%$ probability of each of these values being positive or negative. This is to emphasize the presence of the room reflections over the diffused sound.

3. Apply exponential decay to the gain of the entire IR (the rate of which is inversely proportional to the input T_{60} value), simulating the absorption of sound from the

surrounding walls.

4. Apply a 2nd-order Butterworth band-pass filter, where the cutoff frequencies of this filter are randomly chosen (31.25 – $500\,\mathrm{Hz}$ for the lower bound and 0.5 – $8\,\mathrm{kHz}$ for the upper bound). Such a filter is normally applied to recorded IRs as well [43].

5. Additionally, low-pass Gaussian noise (cutoff frequency at $250\,\mathrm{Hz}$) with $12\,\mathrm{dB}$ per octave rolloff is applied. This spectral tilt was observed in the tail of real IR reverberations [43], and it is perceptually relevant [44].

The steps and values specified above were arbitrarily chosen and, at least impressionistically, these values produced IRs that best resemble IRs that have been recorded in the real world, such as those in the OpenAIR database [43]. At the end of this process, an initial population of IRs is seeded and used as a starting point from which other parameters can be sought.

Fitness Function

Each IR in a population is then assigned a fitness value based on how closely it matches user's specifications. In our implementation, an "error" or "loss" value is assigned, with figures toward zero representing a better "fit," since a value measuring the differences between IRs is desired. First, we calculate the z-score for each of the standard room acoustic parameters summarized in Table 3.1 (except for ITDG, since it can be generated with exact accuracy) using the desired IR parameter values as the mean for each z-score. The error value of each IR is then determined by the absolute sum of all the z-scores (i.e., the total number of standard deviations away from the optimal solution). Because of the differences in units and in numerical scale between IR parameters, this error value allows one to weigh the individual parameters equally and also detect and remove outlier IRs easily.

As for determining the acoustic parameter values for each IR, it is possible to compute them directly and then compare them with the desired values to find the closest match. For instance, T_{60} can be calculated using the IR's Schroeder curve $S(t)$ (also

known as the energy decay curve) [45], formally defined as the integral of the square of an IR $h(\tau)$ from time $\tau = t$:

$$S(t) = \int_t^\infty h^2(\tau)\, d\tau. \tag{3.3}$$

By expressing $S(t)$ in dB, it is possible to locate the instances where the sound energy decays $5\,\mathrm{dB}$ and $35\,\mathrm{dB}$ from the initial level. This time difference is known as T_{30}, and doubling it yields an accurate approximation of T_{60}. This is a common method of calculating T_{60}, as the Signal-to-Noise Ratio (SNR) of a measured IR is often less than $60\,\mathrm{dB}$, and samples within the noise floor should be avoided [41].

EDT can be computed in a similar manner, except that the time difference is estimated between the arrival of the direct sound ($0\,\mathrm{dB}$) and the time when the total energy decays $10\,\mathrm{dB}$. Clarity (C_{80}) can be determined using Schroeder curves as well, and is defined in terms of $S(t)$ as

$$C_{80} = 10 \log_{10} \left(\frac{S(0) - S(0.08)}{S(0.08)} \right) [\mathrm{dB}], \tag{3.4}$$

where $t = 0$ refers to the time of arrival of the direct sound.

For BR, instead of calculating the reverberation times of the relevant octave frequency bands, the IR is transformed into the frequency domain via the DFT before the sound energy ratio (in dB) between the 125 – $500\,\mathrm{Hz}$ and 0.5 – $2.0\,\mathrm{kHz}$ bands is computed. While the actual values between the two methods may differ, the implemented algorithm is faster to execute with no apparent detriment in accuracy. In addition, expressing the BR in dB (as opposed to a unit-less value) may help naive users to better understand this parameter.

Genetic Operations

In each generation, the IR population undergoes three successive operations (i.e., selection, crossover, and mutation) in order to search for a better match. For our solution, truncation selection, random weighted average crossover, and Gaussian multiplication

for mutation have been chosen for the EA. Other alternatives include: rank selection, tournament selection, n-point crossover, and permutation mutation (for a more comprehensive list of genetic operations, the interested reader is referred to [13]). These operations were chosen based on maintaining both computational efficiency as well as acoustic variety in the IR population. Additionally, some methods were simply not viable within the context of creating artificial IRs. One-point crossover, for example, would create a child IR where the size of the virtual room changes after some time, which, in most cases, would not be desirable in a plausible IR.

Truncation selection (also known as elitist selection in other contexts) is one of the simplest selection methods: The bottom percentile of IR individuals (those with the worst fit) are removed from the pool each generation. For our solution, a selection rate of 0.4 was chosen, meaning that individuals with error values below the 60th percentile are removed from the population. The removed IRs are then replaced with new ones through a random weighted average crossover operation. Given two random parent IRs $h_1(t)$ and $h_2(t)$, and a random weight factor $w \in (0, 1)$, the offspring IR $h_{1,2}(t)$ is defined as

$$h_{1,2}(t) = wh_1(t) + (1 - w)h_2(t). \tag{3.5}$$

This operation effectively blends the two parent rooms together to create a new room with its own acoustic parameter values from which a fitness value can be determined. Finally, in the mutation operation, each sample has a very small probability (1 out of every 1000 samples) of changing its value completely. To mutate the IRs in particular, a $50\,\%$ probability of increasing or decreasing in magnitude as well as a $50\,\%$ probability of switching sign for each sample to mutate is desired. To achieve this, mutation is performed via Gaussian multiplication, where each sample is multiplied by a normally distributed random number ($\mu = 0$, $\sigma \approx 1.483$).

Table 3.2: Parameter values used in the EA within our solution depending on reverb quality. Selection rate, fitness threshold, and mutation probability were constant at 0.4, 0.1, and 0.001, respectively.

Parameter	**Quality**			
	Low	**Med**	**High**	**Max**
Population Size	25	25	50	50
Max. Number of Generations	20	50	50	100
Plateau Length	4	10	10	20

Termination Conditions

After each round of genetic operations, the fitness of the new IR population is recalculated, but this is when the EA determines whether or not it should continue modifying the population or stop and return the best-fit IR found so far. This decision is based on three terminating conditions:

1. The fitness value of the best IR found is below a certain threshold value,

2. The fitness value of the best IR found does not decrease after a certain number of generations (specified by a "plateau length" parameter) has passed, or

3. A predetermined limit on the total number of generations to execute has been reached.

The fitness threshold, the plateau length, and the maximum number of generations, therefore, are all parameters within the EA intended to limit computation time. Since an end user might not understand what an EA does, however, including these parameters directly in the plugin's GUI might not be user-friendly. For this reason, a "Quality" setting was used to abstract these parameters away while still providing the user some control over the computation time of the EA and, hence, the fitness values of the IRs. Table 3.2 lists the current mapping between the "Quality" parameter and the EA parameters.

3.3.2 Implementation

Genetic Reverb was implemented as a subclass of MATLAB's built-in `System` and `audioPlugin` classes. These classes can be compiled into a VST 2 plugin compatible with both Windows and macOS operating systems. Table 3.3 presents a list of all parameters that can be controlled by a user, their valid ranges, and a brief description of each.

Table 3.3: IR parameters used in the plugin with their valid ranges and descriptions.

Parameter	**Valid Range**	**Description**
Decay Time	0.4 – $10\,\mathrm{s}$	T_{60} of the desired IR
Early Decay Time	5 – $25\,\%$	EDT of the desired IR (expressed as a percentage of T_{60})
Clarity	-30 – $+30\,\mathrm{dB}$	C_{80} of the desired IR
Warmth	-10 – $+10\,\mathrm{dB}$	BR of the desired IR
Predelay	0.5 – $200\,\mathrm{ms}$	ITDG of the desired IR
Quality	{"Low," "Medium," "High," "Max"}	Sets various parameters for the EA (see Table 3.2)
Mono/Stereo	{"Mono," "Stereo"}	Generate either one IR for both channels (mono) or a different one for each channel (stereo)
Normalize	{"On," "Off"}	In "stereo" mode, forces the RMS level difference in IRs to be zero ("On") or at most $20\,\mathrm{dB}$ ("Off")
Dry/Wet	0 – $100\,\%$	Balance between the dry input signal ($0\,\%$) and the processed one ($100\,\%$)
Output Gain	-60 – $+20\,\mathrm{dB}$	Gain of the mixed dry/wet signal
Generate Room	N/A	Pressing this button generates a new IR using the current parameters
Toggle To Save	N/A	Pressing this button saves the current IR as a binary file in the plugin directory

First, the user has a choice of producing either monotic or dichotic RIRs, labeled in the plugin as "mono" and "stereo" modes, respectively. In "mono" mode, a single IR is generated and then copied for both channels to use. In "stereo" mode, the EA is executed twice, each producing a different IR for the left and right channels. In either case, the "Generate Room" button controls when the EA is triggered to generate new RIRs that replace the existing ones after the completion of the EA. Due to the random nature of the EA, the RMS levels of the two IRs may also differ. We can

limit this difference to be less than the maximum Interaural Level Difference (ILD), approximately $20\,\mathrm{dB}$ [46, p. 230], to preserve the perception of sound localization. When the perceived loudness of each IR must be roughly the same, users have the option of equalizing the RMS levels of the IRs.

Then, we reinstantiate the predelay to the IRs, since a typical IR has some delay before the first reflection reaches the ear. While this predelay was previously used to determine the C_{80} value of the IRs, the delay itself is not added until after the IRs are generated by the EA. Finally, the new IRs are resampled from $16\,\mathrm{kHz}$ to the sampling rate of the host application before convolution with the input signal, since the sample rates of the IRs and the input signal must match.

For real-time convolution reverb, an object that performs partitioned fast convolution is readily available in MATLAB. In the partitioned convolution, the IR is broken up into blocks of 1024 samples each, which are convolved with the audio signal separately and then added back together (via the "overlap-save" method) to produce the output signal. This reduces the latency of the convolution from the entire length of the IR to approximately $1024/f$ s, where f is the sample rate of the audio signal. One limitation of this object is that the length of the array containing the filter coefficients (i.e., the number of samples in the IR) cannot be changed dynamically. As a workaround, we created multiple copies of this object, each with different lengths for the filter arrays. Then, depending on the current value of the T_{60} parameter, the object with the most appropriate length is chosen. Finally, the newly generated IRs can be saved into that specific object, which can then be set as active for use with partitioned convolution with the input signal. Figure 3.2 illustrates the basic flow of data and audio within the plugin, while Figure 3.3 shows its current user interface. Our implementation along with some demonstrations are available from [5].

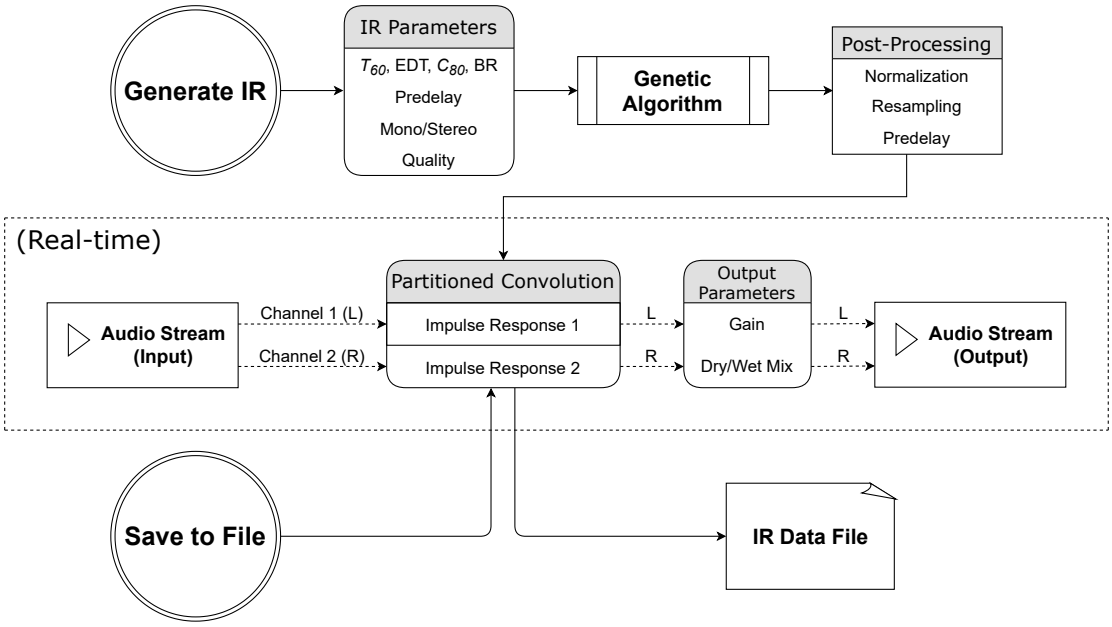


Figure 3.2: Flowchart illustrating the overall functionality of the plugin.

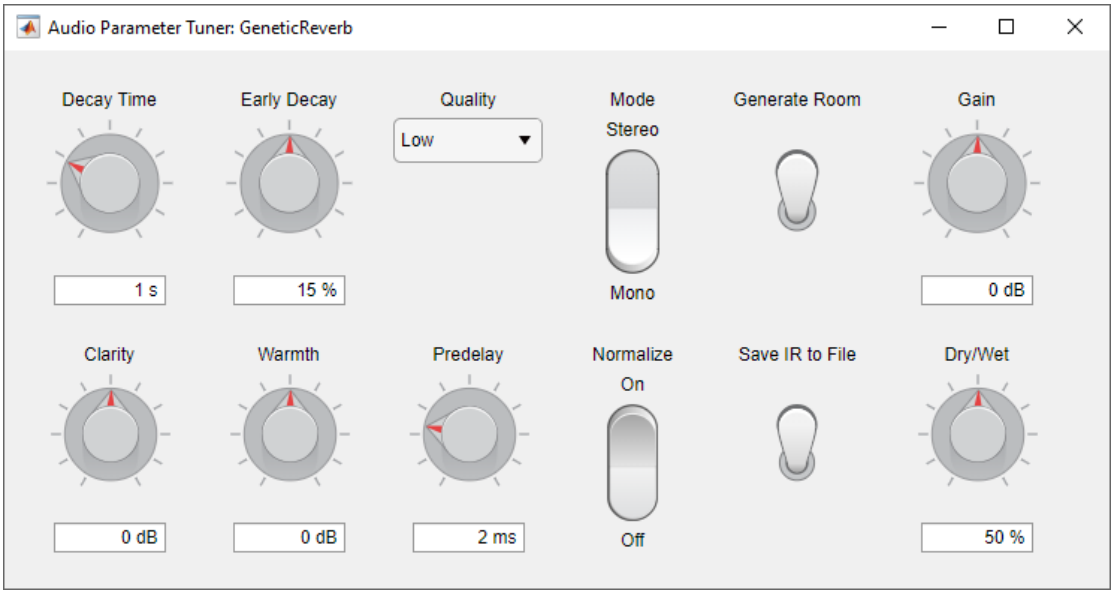


Figure 3.3: Program window displaying the plugin's GUI.

3.4 Evaluation

While we were successful in implementing an EA that can produce synthetic IRs, this alone does not tell us whether the EA can produce IRs that actually adhere to user constraints. In other words, how closely does the output IR match what the user expects in terms of the resulting reverberation? To that end, we performed one objective and one subjective evaluation. For the sake of simplicity, only monophonic IRs were considered

in these tests.

3.4.1 Objective Evaluation

Setup

We were interested in finding how close to zero the loss values could reach for every IR returned by the EA given certain acoustic parameters. We were also interested in the elapsed time to reach those loss values. These tests were run locally on a Windows 10 Home laptop computer equipped with an Intel Core i9-8950HK 6-core notebook CPU running MATLAB version R2020b. In these evaluations, MATLAB's built-in time-keeping tools (i.e., the `tic`/`toc` functions) were used to record the execution times of the EA.

In a first test, the values for EDT, C_{80}, BR, and ITDG were randomly selected within the valid range of the plugin for each IR, while T_{60} was restricted to 0.625, 1.25, 2.5, and $5.0\,\mathrm{s}$ in order to measure the relationship between T_{60} and computation time. This process was repeated four times, one for each of the four possible quality settings, with a population of 250 IRs being generated for each combination of T_{60} and quality setting.

In a second test, the acoustic parameter values were limited to those from real IRs to determine how well the artificial IRs from the plugin could emulate real IRs. For this, an IR was arbitrarily chosen from the OpenAIR database [43], and its acoustic parameter values were calculated using the same algorithms used in the EA. The selected IR was recorded inside the York Guildhall Council Chamber (source position 1, receiver position 1, channel 1), with its acoustic values summarized in Table 3.4. Then, 250 IRs were generated for each of the four quality settings, with the acoustic parameters set to the listed values.

Results

Figures 3.4 and 3.5 provide a summary of the output IRs from the first test, reporting the distributions of loss values and elapsed times, respectively, for each quality setting and T_{60} decay time via violin plots [47]. Like box plots, violin plots include a center

Table 3.4: Acoustic parameter values for an IR recorded in the York Guildhall Council Chamber.

Parameter	**Value**
T_{60}	$0.884\,\mathrm{s}$
EDT	$0.133\,\mathrm{s}$
C_{80}	$4.473\,\mathrm{dB}$
BR	$-1.233\,\mathrm{dB}$

white dot and surrounding gray bar to indicate the mean and Interquartile Range (IQR), respectively. Violin plots, however, are more informative than box plots with the inclusion of probability density functions that are smoothed via kernel density estimation. Figure 3.6 provides a similar summary for the output IRs from the second test.

3.4.2 Subjective Evaluation

Setup

For the subjective evaluation, the goal was to determine whether or not there were perceptual differences between audio signals convolved with real-world IRs and artificial IRs generated with the EA. For this evaluation, three IRs from the OpenAIR database with different T_{60} decay times were chosen. For each one, its acoustic parameter values were calculated, and then two artificial IRs were generated based on these values. One IR was generated using the `high` quality setting, while the other was generated using the `max` quality setting. These IRs were then convolved with two mono audio signals: one from a male speaker at $16\,\mathrm{kHz}$, and one from a dry riff of a synthesized drum kit at $48\,\mathrm{kHz}$, to produce 12 unique reverberated audio stimuli. The IRs were generated at a default sample rate of $44.1\,\mathrm{kHz}$, and then resampled to match the sample rate of the audio signals.

With these audio samples, an ABX test was created. An ABX test consists of a series of questions where two similar but different audio samples labeled A and B are presented, along with a third audio sample labeled X [48]. The audio sample labeled X is identical to either sample A or sample B at random, and participants were asked to determine whether X was equal to A or B. Each possible real IR and artificial IR

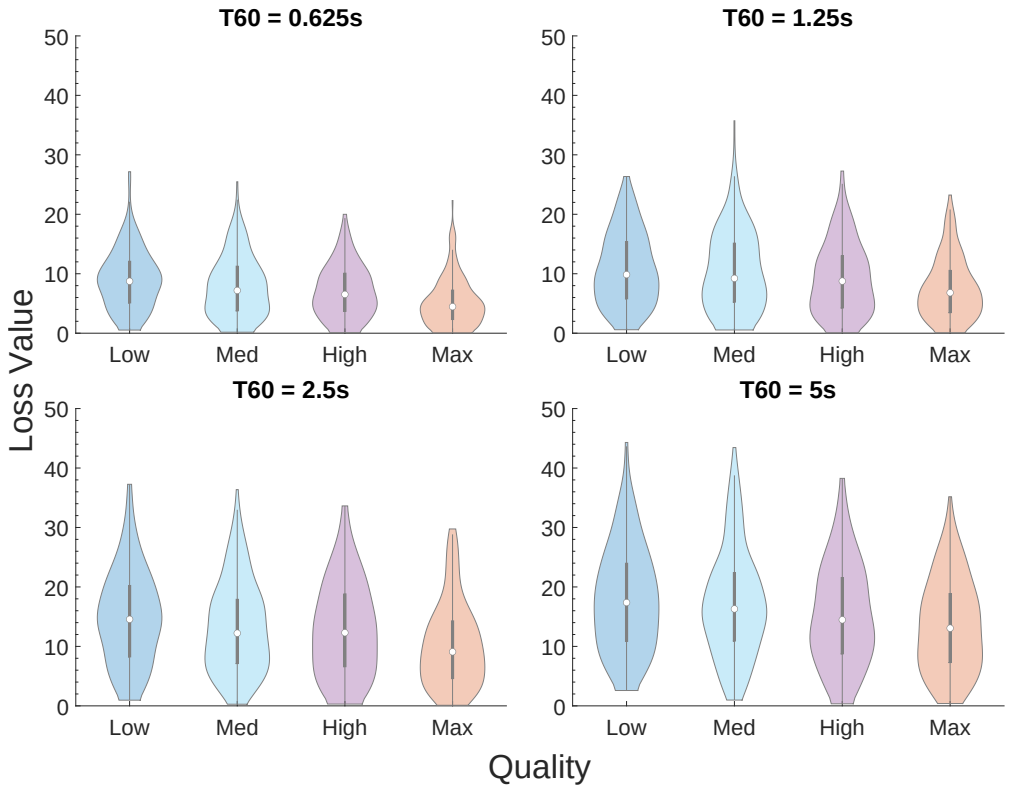


Figure 3.4: Violin plots of loss values for IRs generated with random acoustic parameter settings (lower is better). The center white dots and surrounding gray bars in these and other violin plots indicate the mean and IQR, respectively, of each distribution.

pair combination was presented six times in total, three with the speech stimuli and three with the drum kit stimuli, with random labeling of A, B, and X for each of the 36 total questions. The survey was split into two blocks, with the speech samples being presented in the first block before the drum kit samples in the second block. The order of the questions within each block was randomized for each participant. Participants were asked to conduct the experiment in an environment free of external noise or distractions, and to refrain from creating noise such as eating or chewing gum. However, since the survey was conducted online (as a COVID-19 precaution), such conditions could not be guaranteed.

After the conclusion of the survey period, 26 participants (21 male and 5 female) completed the survey, while five more volunteers started the survey, but did not complete it. The latter were removed from the statistical analyses. Participants who completed the survey were financially compensated for their collaboration. Before the start of the experiment, participants were asked about their age, sex, and amount of musical

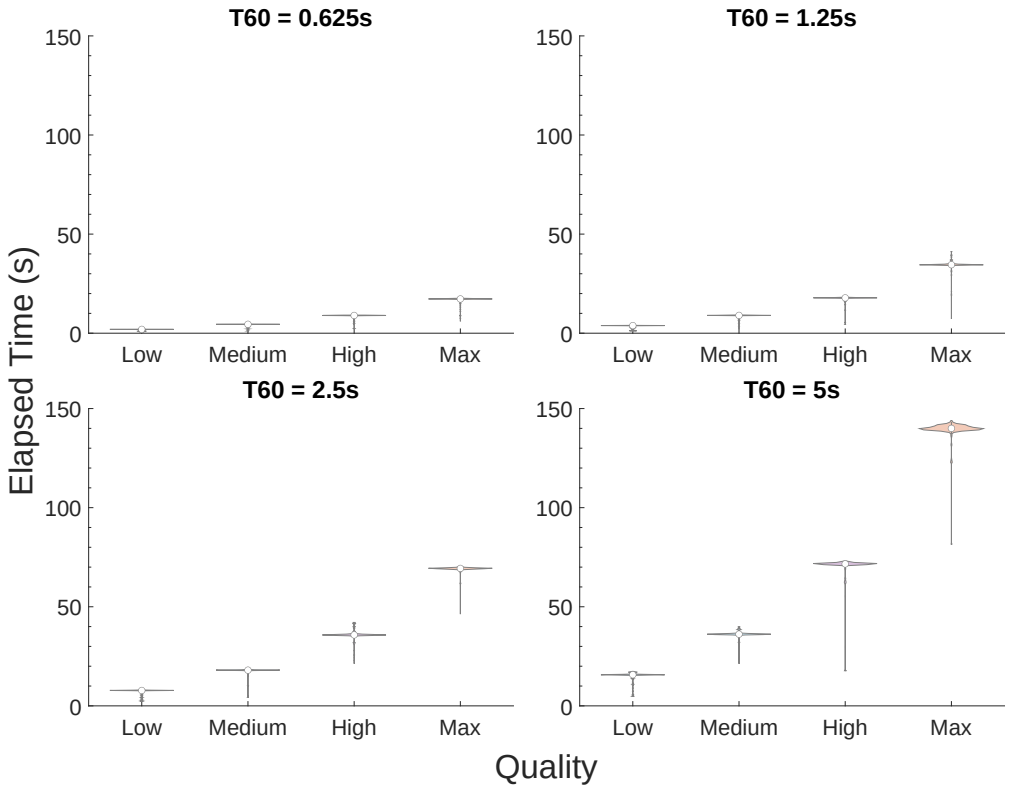


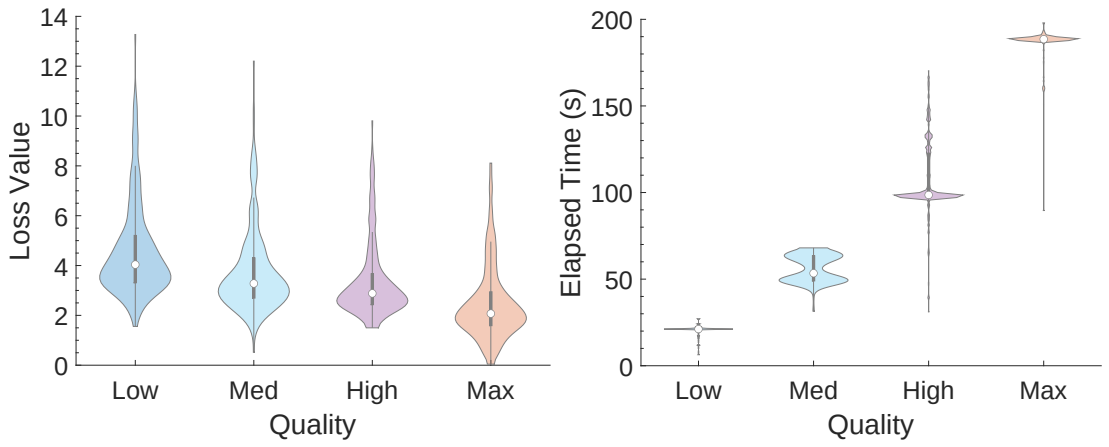
Figure 3.5: Violin plots of computation times for IRs generated with random acoustic parameter settings (lower is better). The center white dots and surrounding gray bars in these and other violin plots indicate the mean and IQR, respectively, of each distribution.

experience. The ages of the participants who completed the survey were in the range of 20 – 41, although 24 of the 26 participants were between 20 and 25 years old. Seven of the participants declared that they had no musical experience, ten responded with 1 – 3 years of experience, and the remaining nine declared four years of experience or more. Permission for performing this experiment was obtained following the UoA ethics procedure.

Results

Table 3.5: ANOVA Type II Wald chi-squared test results for the GLMM fitted to all responses.

Factor	χ^2	df	p
`program`	35.113	1	< 0.001
`quality`	5.827	1	0.016



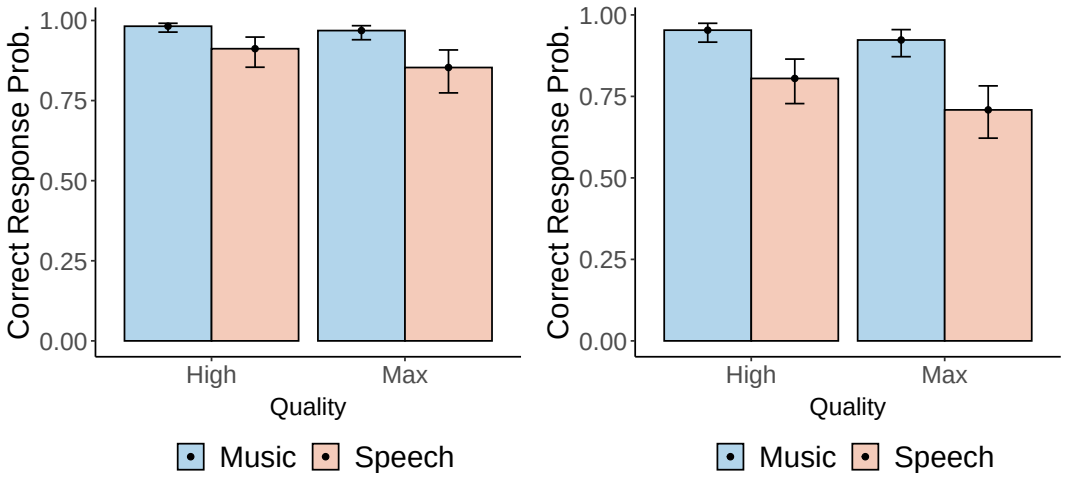
(a) (b)

Figure 3.6: Violin plots of (**a**) loss values and (**b**) computation times for IRs generated with fixed settings (lower is better).

A script was developed in R [49] to determine whether or not variables such as age or quality settings had a significant effect on the rate of correct answers in the experiment. A Generalized Linear Mixed Model (GLMM) was implemented to predict such effects for the binomially distributed data. Although there are other alternatives to analyze this kind of data, it has been shown that GLMMs are suitable for these analyses [50]. For this experiment, age, sex, years of musical experience ('exp'), the repetition order of a question ('rep'), and subjects (`id`) were considered random effects. Meanwhile, the `program` (speech vs. drum riff), IR `quality` (`high` vs. `max`), and whether sample X was created from a real or artificial IR (`x.type`) were deemed fixed effects. Starting with a model just containing each subject as a random factor, we conducted multiple pairwise Analysis of Variance (ANOVA) tests between models that differ by a single factor or interaction, updating the model each time to keep any factors that resulted in significantly better fit at a $95\,\%$ confidence level.

At the end of this process, it was determined that only `program` and `quality` had a significant effect on the subjective responses. We then conducted an ANOVA Type II Wald chi-squared test on the final model, of which the χ^2 values, degrees of freedom (DFs), and p-values for each significant factor are summarized in Table 3.5.

A post-hoc analysis based on Tukey's Honest Significant Difference (HSD) test comparing the Estimated Marginal Means (EMMs) of each level of the interaction was



(a) (b)

Figure 3.7: Means (dots) and $95\,\%$ confidence intervals (error bars) for the rates of correct responses by quality setting and program: (**a**) for all subjects, and (**b**) after removing responses from subjects deemed as super-classifiers.

conducted with the `emmeans` library [51], with the DFs being asymptotically computed for this analysis. Tukey's test showed that, regardless of quality setting, the rate of correct answers for the drum riff was significantly higher than that of speech ($z = 5.926$, $p < 0.001$). Additionally, regardless of `program`, the rate of correct answers given `high` quality IRs was significantly higher than that of `max` quality IRs ($z = 2.414$, $p = 0.016$). These findings are summarized in Figure 3.7a, which shows the mean and $95\,\%$ confidence interval of the probability of a correct answer by quality setting and program. We can also conclude from Figure 3.7a that the subjective responses were significantly different from random, since the rate of correct responses is significantly higher than $50\,\%$ in all cases.

A further inspection of the responses revealed that twelve participants returned perfect or near-perfect scores (those who answered $100\,\%$ correctly for at least three out of four question groups, where each question is classified based on the program and the quality setting of the artificial IR). We ran a similar statistical analysis without these participants (or "super-classifiers") to verify that the previous results still hold for the remaining participants.

Despite a decrease due to the removal of super-classifiers, the rate of correct re-

sponses is still significantly higher than $50\,\%$ in all cases. Furthermore, similar GLMMs with the reduced dataset still show that the simplest statistically significant prediction model remains the same as that shown previously, as summarized in Table 3.6. Tukey's

Table 3.6: ANOVA Type II Wald chi-squared test results for the GLMM fitted to all responses except those from super-classifiers.

Factor	χ^2	*df*	*p*
`program`	28.432	1	< 0.001
`quality`	4.140	1	0.042

HSD test also yielded similar results, with the rate of correct answers for the drum riffs being higher than that of speech ($z = 5.332$, $p < 0.001$, across levels of `quality`). Similarly, the rate of correct answers given `high` quality IRs was still significantly higher than that of `max` quality IRs ($z = 2.035$, $p = 0.042$, across levels of `program`). These findings are summarized in Figure 3.7b.

3.5 Discussion

In short, our solution is not able to produce an adequate IR $100\,\%$ of the time. However, this outcome was to be expected for reasons that will be explained below. Both the objective and subjective evaluations also provide some further insight into specific areas where the EA may be lacking, along with possible solutions.

3.5.1 Objective Evaluation

The mean loss value for a population of IRs given random parameter values is around the 10 – 18 range (about 2.5 – 4.5 standard deviations per parameter) on the `low` quality setting, with minimal improvements in fitness as the quality setting is increased. This indicates that not every combination of parameter values will be able to yield viable results. The relatively high error values for this first test can be partially attributed to the fact that certain combinations of parameter values would result in IRs that would be very difficult (or impossible) to produce. For instance, IRs with long T_{60}

tend to have lower clarity (C_{80}) values as well, since there are many more late reflections that can overpower the early ones. Otherwise, if reasonable values are chosen for these parameters, as is the case for the second test, then IRs that closely match the desired parameter values can be obtained through the EA, with a mean error value of about 4 at `low` quality and decreasing as quality increases.

3.5.2 Subjective Evaluation

As a whole, while the participants were able to distinguish between real and artificial IRs, two observations can be made: First, there is a significant decrease in the rate of correct responses when going from `high` quality to `max` quality IRs in both the speech and music sections of the experiment. So even though the artificial IRs have yet to be indistinguishable from real IRs, the fact that such a drop occurred suggests that it is possible to minimize perceptual differences even further by increasing the execution time (e.g., by increasing the size of the population or the maximum number of generations). Second, the IRs are more distinguishable when convolved with music as opposed to speech. One possible explanation for this outcome is that the original speech and music signals were sampled at different frequencies (16 and $48\,\mathrm{kHz}$, respectively). It is possible that the additional information (in frequencies above $8\,\mathrm{kHz}$) could be used to identify differences in music and gain an advantage over speech. Furthermore, the impulsive nature of the drum riff could also make the differences more apparent (regardless of the difference in sample rate).

One open question regarding these results would be: Given two different IRs with the same acoustic values, are they perceptually indistinguishable? According to Hak et al. [52], there is a Just Noticeable Difference (JND) for each of the ISO parameters in our solution: $5\,\%$ for T_{60}, $5\,\%$ for EDT, and $1\,\mathrm{dB}$ for C_{80}. Further experiments could determine the JND for combinations of these parameters and IRs in general.

3.5.3 Limitations

As previously mentioned, only monophonic IRs were evaluated. While binaural IRs can be produced by our solution, comparing them to real binaural IRs may hinder their perceptual evaluation, since spatial aspects, such as apparent location of the sound source (i.e., its azimuth and elevation), proximity, etc., may take relevance over the parameters we controlled (decay time, EDT, clarity, etc.).

As for the implementation of the plugin, while MATLAB allows the prototyping and generation of VST plugins with relative ease, there are a few setbacks for a full implementation: There are a limited number of data types for parameters, meaning that some GUI elements such as buttons cannot be directly implemented. In practice, a toggle switch can act as a substitute for triggering actions such as generating an IR or saving a file, but the resulting interface may be clunky for an end user. One feature that could be improved in our solution is the capability to save the IRs on the disk. As of June 2023, MATLAB does not allow saving IRs directly as Waveform Audio File Format (WAV) files (or in most other audio formats), so binary files are used instead. These can be converted into WAV files using an ancillary program (an example of which is also implemented and provided in the plugin repository).

An additional area of improvement pertains to how the ILD can be controlled in our solution. Currently, a user can choose to either balance the stereo image (i.e., equalize the RMS levels of the IRs) or leave the ILD value to random chance. In the latter case, the ILD is clipped to $\pm 20\,\mathrm{dB}$ only if it falls outside of this range. However, the ability to specify the ILD of the stereo image (e.g., from -20 to $20\,\mathrm{dB}$) is another possibility. A revised implementation and subsequent change to the GUI is deferred to a future version. Another issue is the amount of CPU resources needed to both generate the impulse responses and process the input audio stream in real time, especially as the length of the IR increases. This is why the sample rate of the IRs in the plugin is set to $16\,\mathrm{kHz}$, and why the IRs must then be resampled to match that of the input audio. Despite this, CPU overloading issues can still arise, especially when attempting to generate IRs with long T_{60} times. Even with a CPU as powerful as the one used in

this research, when running the plugin within Ableton Live 10 [53] at $44.1\,\mathrm{kHz}$, the application often crashed when attempting to generate T_{60} longer than about $2.5\,\mathrm{s}$ due to the long convolution. It is expected that, in the future, the full potential of multi-core processors becomes more accessible for real-time audio plugins so that either the partitioned convolution load can be distributed among several cores or the EA can be run as a background task while the convolution continues to run with the current IRs.

One more possible point of contention in our solution is the inclusion of the BR parameter, since it only accounts for frequencies in the 125 – $2000\,\mathrm{Hz}$ range, meaning there is no control over frequencies outside of this range in the IR. Even Beranek [42, p. 512] admits that BR is not very useful for measuring "warmth," even though this method has been around for the longest time. The decision to include the parameter was done for the sake of convenience to the user, who is thus able to partially control the frequency content of the reverberation with just a single parameter. One possible modification to our solution that could ameliorate this would be to allow the user to control the T_{60}, EDT, and C_{80} values for individual octave bands, and to modify the fitness function of the EA to take these parameters into account for all octave bands. This would require adding additional parameters to the plugin's GUI for each desired octave band, meaning that there is a trade-off between control over the artificial IRs and ease of use of the plugin.

3.6 Interim Conclusion

A working prototype was developed for a plugin that uses EAs to create artificial reverberation. The parameters of the plugin have been carefully chosen and designed so that anyone from music producers to game designers could understand and utilize the plugin with ease. Even if current technologies prevent us from producing IRs with good enough fit within a reasonable amount of time, there is not necessarily a correlation between fitness value and desirability either. Perhaps the random nature of the EA can be seen as a feature rather than a setback to some users, as an IR that sounds pleasing to

the ear may come at unexpected times. Thus, evolutionary algorithms can be a source of creativity in digital signal processing, audio design, and music production, as shown in our solution.

Further research and development could explore other methods that could decrease computation time and/or improve fitness, such as choosing different genetic operations or changing the genetic algorithm parameters. Parallelizing certain tasks with a multi-core CPU or a Graphics Processing Unit (GPU), or at the very least executing the EA in the background, is one of several improvements that could also be made with further development. Whether or not it is possible to generate synthetic IRs that are perceptually indistinguishable from real-world IRs or other synthetic IRs with acoustic parameters within the JNDs margin is another topic for further investigation.

Chapter 4

DSP Audio Synthesis via RCGP

4.1 Introduction

In audio DSP research, the replication of (perhaps unknown) sounds via software synthesizer programs has been a long-studied problem with many potential applications. One such application is the reproduction of musical instruments, whether they be recorded from real instruments or produced by other audio synthesizers: given a target sound, is it possible to generate an audio synthesizer whose output closely approximates (if not matches) the given sound?

In this chapter, we introduce the first part of our CGP method in which DSP synthesizers are represented as directed cyclic graphs, and are evolved via RCGP to produce candidate DSP programs, the output of which are compared to the target sound via Mel-Frequency Cepstral Coefficients (MFCCs) to determine fitness. The best-fit program is returned as source code in FAUST. We then evaluate our method, as well as determine the effects of various RCGP hyper-parameters, by replicating as closely as possible the Sandell Harmonic Archive (SHARC) database [54], a collection of steady state tone descriptions whose waveforms can be reproduced by additive synthesis. Finally, we discuss the results of such evaluation as well as future directions with this research.

4.2 Background

4.2.1 Previous Work

While RCGP can be used as a framework to create more general DSP applications, various other evolutionary techniques can be used for more specialized tasks. For instance, in the previous chapter, we described how a classic EA can evolve a population of RIR waveforms to generate a desired RIR subject to various room characteristics input by the user [29, 30]. Such IRs can then be mixed with an input audio signal via convolution reverb in real-time within a VST audio effects plugin to produce a signal with the desired reverberation.

However, there were two major limitations with our method. To summarize, the first is the fact that the EA itself is unable to generate IRs in real-time, rendering tasks such as the automation of room parameters impossible to perform. The other is due to the limitations of the MATLAB programming language with which "Genetic Reverb" was implemented. Furthermore, while the tools that MATLAB provides enables quick prototyping of real-time audio plugins, the fact is not only that MATLAB is proprietary software, but also that only the VST version 2 and AU interfaces, both proprietary themselves, are supported when compiling such plugins. Despite these limitations, we believe that our first attempt at using evolutionary techniques to generate DSP programs was a successful one, as it paved the way for further exploration into not only alternative methods for generating artificial reverberation (see, for example [55]), but also other potential applications of such algorithms in DSP research.

As such, we began to expand on the idea of generating DSP programs using evolutionary techniques to include any DSP program in general. In practice, however, DSP programs can be classified as either audio instruments/synthesizers, which can produce an audio signal, or audio effects, which can manipulate an input audio signal to produce a modified output signal, so each use case must be treated separately. For this chapter, we focus on the development of an implementation of RCGP that can generate DSP synthesizers subject to the characteristics of the target audio signal. An early version of

this implementation was demonstrated in [56], which used the Log-Spectral Distance (LSD) as a fitness measure to compare audio signals. Since then, we have improved our implementation to include bug fixes, additional features, and alternative fitness measures and selection schemes. In the next chapter, we will evaluate the effects of various RCGP hyper-parameters on the fitness of the best-fit solutions.

4.2.2 Related Work

In comparison, many of the previous attempts by others to use evolutionary algorithms in sound replication tackled the problem in terms of EP, or the optimization of a predefined set of parameters to produce values that closely match a target sound. These programs can focus on a single synthesis technique such as wave-table synthesis [57, 58] or Frequency Modulation (FM) [59–63], or they can be based on real-world synthesizers [64, 65], which include a combination of an arbitrary number of these techniques in a specified configuration.

In recent years, however, ANNs have overtaken such algorithms as a popular tool for tackling the problem as a whole. While various ANN architectures have been trained and used to replicate the target signal directly (see, for example [3, 66]), others have been developed to optimize various synthesizer parameters as well [2, 65]. Regardless, this EP paradigm for parameter optimization ultimately comes with the disadvantage of inflexible or inefficient program design, as different DSP algorithms or synthesis techniques may be more suitable for different sounds. GP (as opposed to EP or ANNs) solves this issue by evolving the program itself, easing the finding of a DSP algorithm suitable for a given sound without requiring any knowledge of the target sound itself.

Previous attempts to use GP to handle this problem, however, are far and few between. As mentioned in Section 2.5, the first and only other known attempt was conducted by Macret and Pasquier [27], representing DSP synthesizers as Pd patches and evolving them via MT-CGP. With the main disadvantages of MT-CGP being its arbitrary typing and its lack of recurrent connections, We believe that both RCGP and the FAUST language are better alternatives for the automatic induction of DSP programs,

and that our method, which we will describe in the next section, allows for more appropriate solutions to be found for any given sound.

4.3 Methods

4.3.1 FAUST Implementation

We now discuss the details around the implementation of RCGP as it pertains to our specific application. One of the first things to consider when creating an RCGP method is the function set, a collection of basic functions that can be composed together to form a more complex program. This includes what are known as "terminals," or functions that do not take any input values. Often, terminals take the form of constant values, but other functions such as random number generators can fall into this category as well.

In FAUST, all functions and primitives operate on a single "signal" data type, which is itself a function $s : \mathbb{Z} \to \mathbb{R}$, where $s(t) = 0$ for all $t < 0$. Because of this, FAUST primitives are also known as "signal processors," which are of the form $p : \mathbb{S}^n \to \mathbb{S}^m$ for $n, m \in \mathbb{N}$, where $\mathbb{S} = \mathbb{Z} \to \mathbb{R}$ is the set of all possible signals. For instance, the symbol '/' is the division function primitive in FAUST, defined as $/ : \mathbb{S}^2 \to \mathbb{S}$, where $y(t) = x_1(t)/x_2(t)$ for $x_1, x_2, y \in \mathbb{S}$.

This division is strictly speaking not a valid signal processor, however, as $y(t)$ can be undefined (or rather, take the NaN value) for some values of t whenever $x_2(t) = 0$. In reality, even addition and multiplication themselves are not valid signal processors either, as values of $\pm\infty$ can be produced if the output is too large to be represented as a floating-point number. Whenever any function is added to the function set, one must ensure that the "closure property" remains satisfied, whereby each function must be able to accept any value that is returned by any function in the function set as input [8]. Functions that do not satisfy the closure property such as division, then, are often modified in such a way that closure can be ensured. Sanfilippo [67] provides some techniques on how to protect these operations from $\pm\infty$ and NaN values in FAUST, thus avoiding any undesirable behavior that can arise from these values. The protected

division operation can be defined as:

$$y(t) = \begin{cases} \frac{x_1(t)}{\min(x_2(t),\, -\epsilon)}, & x_2(t) < 0 \\ \frac{x_1(t)}{\max(x_2(t),\, \epsilon)}, & x_2(t) \geq 0 \end{cases}, \tag{4.1}$$

where ϵ is a FAUST constant equal to the smallest positive value that can be represented as a floating point number. For additional safety, the result of the division can also be clipped to some fixed interval, (in this case, $[-1/\epsilon, 1/\epsilon]$ at the time of original publication) to ensure that future computations also cannot reach $\pm\infty$. In fact, all four of the basic arithmetic operations, among other functions, require their outputs to be clipped in order for them to both be considered valid signal processors and satisfy the closure property.

Once these protections are in place, we can now define a basic set of functions with which we can generate DSP programs. Figure 4.1 defines a list of basic functions, along with their corresponding primitives and FAUST definitions, for the function set in our implementation of RCGP. It includes a helper constant `MAX`, which is equal to $1/\epsilon$, as well as a helper function `max_clip`, which clips its inputs values to the range $[-1/\epsilon, 1/\epsilon]$.

The usual arithmetic operations are performed via the `add`, `sub`, `mul`, and `div` functions, with their results clipped and `div` defined as above. Constant terminals such as 0 and 1 are also manually added as these values cannot easily be reproduced with the current function set. Finally, the four oscillator functions that generate the various basic waveform shapes (sine, sawtooth, square, and triangle) make up the component signals that will ideally be added together to form the target signal. The one-sample delay function, or the `mem` primitive/keyword in FAUST, is also included as a way to partially control the phase of the oscillators as well as the combined signals.

4.3.2 RCGP Implementation

Recall from Section 2.4 that in RCGP, each individual program is encoded as an array of nodes, which are classified into three types: inputs, functions, and outputs.

```
import("stdfaust.lib");

EPS = ma.EPSILON;
clip = min(1 / EPS) : max(-1 / EPS);

// Constants (Terminals)
zero = 0;
one = 1;

// Fundamental frequency (Terminal)
freq = 440;

// Protected arithmetic operations
add = + : clip;
sub = - : clip;
mul = * : clip;
div(n, d) = ba.if(d < 0, n / min(-EPS, d), n / max(EPS, d))
    : clip;

// One-sample delay
delay = mem;

// Waveform oscillators
sine = os.osc;
sawtooth = os.sawtooth;
square = os.square;
triangle = os.triangle;
```

Figure 4.1: Preamble FAUST code defining basic functions for use in RCGP.

Typical RCGP usage includes at least one input node, one output node, and dozens of function nodes. A function node contains exactly one function from the function set with, depending on the individual function, n inputs and m outputs from which edges can be connected. During program execution, each function node also contains an internal value that is calculated based on the input values it is given. Each output node is connected to exactly one function node from which its value is returned as the output of the program. Not all function nodes necessarily have to contribute to the output of the program, however. Depending on the layout of the edges, a function node may be considered "inactive" if it does not contribute to the output of the program at all, but function nodes may become active or inactive at any time as the population is evolved. This allows for "neutral genetic drift," where the phenotype of an individual program

can be variable in length [11].

Finally, the input node contains a value that typically changes when calculating the fitness of the program, depending on the outputs of the program to be evaluated. While a DSP synthesizer can technically produce a signal without requiring any inputs, typical usage of DSP synthesizers allows at least for the pitch to be adjustable in order for them to be useful as musical instruments. To mirror this in our RCGP implementation, the fundamental frequency of the target sound can be provided as an optional argument. Then, for each individual in the population, we include one input node that will effectively serve as a function node instead, but whose value is a constant equal to the given frequency, thereby adding the fundamental frequency into the function set as a terminal function in the process.

In the CGP evolutionary process, a population of DSP programs is initialized, picking a random function for each function node and randomly drawing the connecting edges. Each "generation" consists of computing the fitness of each individual, selecting an individual to be reproduced for the next generation, and generating new "offspring" programs (via some modifications to the selected individual) to replace the population. Typically, an ES where only one parent is selected each generation and four offspring are produced from that parent is used, denoted as $(1 + 4)$ [10]. In RCGP, an additional "recurrent connection probability" parameter is added to control the frequency of feedback loops being created during this step [11]. This generational process is repeated a predetermined number of times or until the fitness value is satisfactory. Then, the program with the best fitness value across all generations is returned as the result. Table 4.1 summarizes the list of parameters used in RCGP as well as the values used in our DSP implementation.

In order to demonstrate the capabilities of RCGP, Turner and Miller developed and released `CGP-Library` [68], a library licensed under the GNU Lesser General Public License Version 3 (LGPLv3) which implements RCGP in C. This library is used as the basis for our RCGP implementation, with some modifications to produce working FAUST code. In support of the free software movement, these modifications as well

Table 4.1: List of RCGP parameters and the default values used in `FaustCGP`.

Parameter Name	**Value**
Evolutionary Strategy	$(1 + 4)$
Input Nodes	1
Function Nodes	15
Output Nodes	1
Error Threshold	0.1
No. of Generations	1000
Recurrent Connection Probability	0.1

as the source code for the RCGP implementation are also publicly available as free software, the latter of which also being released under the GNU General Public License Version 3 (GPLv3) [6].

4.3.3 DSP Synthesis

Before executing the evolutionary process, we begin with the target audio stored in Free Lossless Audio Codec (FLAC) format. After removing any Direct Current (DC) offset and normalizing the first two seconds of the signal to $0\,\mathrm{dBFS}$ (decibels relative to full scale), we calculate its power spectrum from $0\,\mathrm{Hz}$ to the Nyquist frequency via the DFT [69] for the first 4096 samples. The values are stored in a text file as (x, y) pairs, with x being the frequency of a frequency bin, and y, its magnitude. This power spectrum can then be read by the RCGP fitness function as a reference for comparison with the power spectra of signals generated by candidate programs.

Then, to evaluate the fitness of each individual, we first convert each individual's genotype (i.e., graph structure) into their respective phenotypes as text in the format of FAUST code. Ren et al. [70] provide an algorithm for precisely this task, which is also capable of handling an arbitrary number of feedback loops at any and all points within the graph structure. A simple example of how an arbitrary feedback loop can be coded in FAUST is shown in Figures 4.2 and 4.3, with the former showing an example feedback loop in Pd and the latter showing the corresponding FAUST code for this feedback loop.

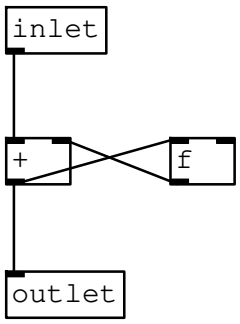


Figure 4.2: A Pd patch with a simple feedback loop.

```
main(rec_in, inlet) = rec_out, outlet with {
  add = inlet, rec_in : +;
  rec_out = add;
  outlet = add;
};

process = main ~ _ : !, _;
```

Figure 4.3: The FAUST code corresponding to the Pd patch in Figure 4.3, based on the algorithm by Ren et al. [70].

From there, we can compile the source code via the `libfaust` library, compute the output signal from the resulting executable, and calculate its power spectrum. Finally, we calculate the LSD value by comparing this power spectrum with that of the target signal (retrieved from the text file) from the given fundamental frequency up to $10\,\mathrm{kHz}$. Given two power spectra X_1 and X_2, the LSD d in the range $[a, b]$ is

$$d = \sqrt{\frac{1}{b-a+1}\sum_{k=a}^{b}\left(10\log_{10}\frac{X_1(k)}{X_2(k)}\right)^2} \quad [\mathrm{dB}]. \tag{4.2}$$

In other words, the LSD is the RMS of the level difference in corresponding frequency bins. This LSD serves as an error value quantifying the difference between the two sounds, where values closer to zero represent better fitness. A diagram summarizing this process is shown in Figure 4.4.

Once the best-fit individual is determined (i.e. the program whose output signal has the smallest LSD when compared to the reference signal), the FAUST source code corresponding to that individual is output.

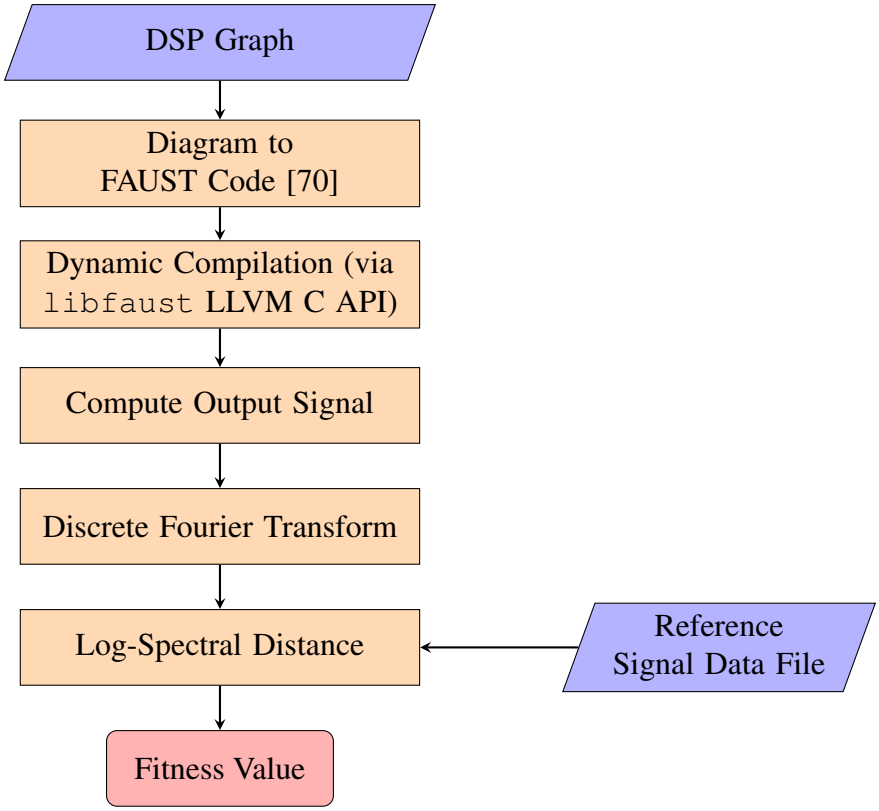


Figure 4.4: Flowchart summarizing the procedure for calculating the fitness value of any given individual in `FaustCGP`.

4.4 Evaluation

4.4.1 Setup

As a starting point for our evaluation, we attempt to replicate the sounds in the SHARC timbre database [54], a collection of steady state tone values analyzed from recordings of various orchestral instruments at various pitches, where each tone is identified by a fundamental frequency along with the magnitude and phase at each harmonic up to $10\,\mathrm{kHz}$. As such, the provided data makes it easy to both reproduce the steady state waveforms and compare our solution to the reference signals. With a total of 1338 tones in the database across 39 different instruments, we believe such a collection is a suitable enough challenge for our implementation, as such instruments are regarded as being able to produce a musically diverse set of timbres [71].

Using the set of functions as defined in Section 4.3.1, ten seeded runs of RCGP were executed for each tone, one for each combination of 15, 30, 60, 120, and 240

function nodes, stopping after 100 and 1000 generations. The LSD values of the best-fit individuals were recorded along with its FAUST code and corresponding waveform, for ease of comparison between the signals. The runs were executed with a Mac mini (running Arch Linux instead of macOS), with $8\,\mathrm{GB}$ of RAM and an 8-core Intel i7-2635QM CPU. For faster computation, all 8 CPU cores were utilized at the same time, with one RCGP process per core, via GNU `parallel` [72].

4.4.2 Results

The entire experiment took approximately 10 days to complete, even with the parallelism that GNU `parallel` provides. The mean execution times for all runs that stopped after 100 and 1000 generations were 69.1 and $769.4\,\mathrm{s}$, respectively. Of the 1338 tones in the SHARC database, only one was unable to be generated after the predefined number of generations, resulting in infinite LSD values. Among the remaining programs, the mean LSD value was $6.3\,\mathrm{dB}$ ($\sigma = 2.7\,\mathrm{dB}$) after 100 generations. After 1000 generations, it was $5.1\,\mathrm{dB}$ ($\sigma = 2.3\,\mathrm{dB}$). Similar values are obtained when broken down by the number of function nodes given. The minimum LSD values attained after 100 and 1000 generations were $0.81\,\mathrm{dB}$ and $0.52\,\mathrm{dB}$, respectively, although they are still above the previously set threshold of $0.1\,\mathrm{dB}$. Figure 4.5 reports the mean and standard deviation of LSD values via an error plot, broken down for each combination of the number of function nodes and the number of generations. The generated FAUST code is publicly available online [6], with audio samples generated on-the-fly via `Faust2WebAudio`, a JavaScript wrapper that can compile and run FAUST programs in a web browser.

4.4.3 Discussion

Our experiment shows that for a select number of tones, our method has the potential to reproduce the target sounds with sufficient accuracy. While early results show some promise, we believe the results can be improved in several ways. First, we plan to perform parameter sensitivity analysis in order to find a more optimal set of RCGP pa-

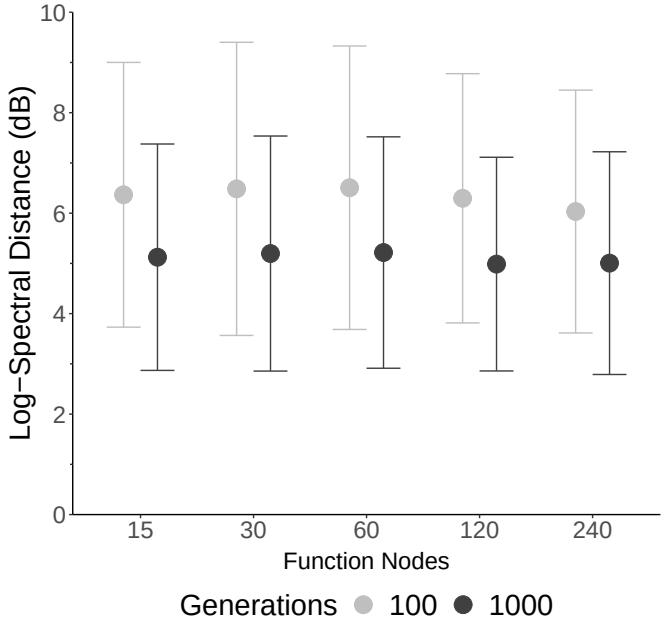


Figure 4.5: Means and standard deviations of LSD values by number of function nodes and number of generations.

rameters for our method. While the number of function nodes does not seem to have a significant effect on the LSD values, other changes such as increasing the number of allowed generations even further, increasing the population size, or changing the function set may help yield better results. Second, even though we provided to the RCGP basic component functions that are common in basic additive synthesis, the random nature of the connections between nodes means that not all DSP programs generated will adhere to the additive synthesis model, and that non-steady state signals can be produced. We intend to improve our fitness function by evaluating the output signals over longer periods, calculating the LSD as the mean of several LSD values at regular intervals. This should increase the likelihood that non-additive synthesizers and non-steady state signals would be removed from the population.

Another possibility to explore is the addition of even more component functions such as noise generators, filters, Boolean operators, and many more. This means that our approach could be extended to generate almost any DSP synthesizer in existence, including subtractive, wavetable, FM, and other synthesis techniques on top of additive synthesis, along with possibly even new types of synthesis that are yet to be discovered.

4.5 Interim Conclusion

We proposed a novel framework for generating arbitrary DSP programs via RCGP in order to replicate a target sound as closely as possible. These programs are encoded as directed graphs which can be cyclic or acyclic, and various similarity measures between each program's output signal and the target signal determine the fitness of these programs. The result is in the form of source code in FAUST, a functional programming language that can be exported to a multitude of other programming languages and applications. Preliminary evaluations show that the steady state spectra of a wide variety of orchestral instruments and pitches can be reproduced with varying degrees of accuracy. The advantages of this approach do not just lie in the flexibility of the FAUST language, which in itself can have practical uses across multiple platforms and architectures, but also lie in the flexibility of the RCGP, which is only limited by the function set that is provided.

Such a function set can be extended with the addition of even more component functions such as noise generators, filters, Boolean operators, and many more. This means that our approach could be extended to generate virtually any DSP program in existence, including subtractive, wavetable, and FM synthesis, along with even new types of synthesis that are yet to be discovered.

As previously mentioned, another possible application of this method the ability to generate audio effect chains as well. Given a dataset containing the desired frequency response of the effect chain, candidate programs generated via RCGP can be similarly evaluated based on the spectral characteristics of the IR of the audio effect. In Chapter 6, we will discuss in further detail how RCGP can be extended to facilitate such DSP applications as well.

Chapter 5

Effects of RCGP Parameters on DSP Programming

5.1 Introduction

In the last chapter, we demonstrated the real-world potential of RCGP in its applications to DSP synthesizer programming. Many open questions still remain regarding the performance and efficacy of CGP, however [12]. as many of the CGP parameters were still left to their default values as determined by Miller [9], among other similar papers, which may or may not be the most optimal values for our particular application.

To extend our knowledge on CGP, we evaluated the effects of three parameters in the context of digital audio synthesis: selection method, recurrence probability, and inclusion of prior knowledge in the evolutive process. In other words, we draw the probabilities of node selection and mutation from a known distribution as opposed to a uniform distribution, in a similar manner as informative priors are often preferred over non-informative priors in Bayesian data analysis.

This chapter was first published in [73].

5.2 Background

Many previous approaches to sound synthesis using EAs have been done through EP, where the structure of the program is fixed to one or more known synthesis techniques and only the parameter values are evolved and evaluated (see, for example, [74] for a survey of basic approaches). As for GP-based approaches, the first and only other known attempt was conducted by Macret and Pasquier [27]. As previously explained in Section 2.5, they used MT-CGP to evolve Pd patches, synthesizing sounds in a visual programming environment in real-time. The use of mixed typing, however, was only needed because of Pd's distinction between audio and control signals, arbitrarily limiting the solution space of DSP programs. In [56], we were able to remove this distinction by evolving DSP programs in FAUST instead, which operates on a singular "signal" data type, instead. In either case, extensive analysis of the CGP parameters used was yet to be conducted, which is what we aim to address here.

Arguably, the most common CGP configuration is a $(1 + 4)$-ES, as previously described in Section 2.3 [9], although other population sizes are also possible. Turner and Miller [11] also showed performance improvements of recurrent CGP over acyclic CGP on the Artificial Ants [17] and sunspot prediction [16] problems. Thus, $(1 + 4)$ in recurrent CGP became the de-facto standard in many implementations.

However, a recent study by Kalkreuth [75] shows that other evolutionary strategies, such as tournament selection in combination with subgraph crossover, outperform $(1 + 4)$ or even $(\mu + \lambda)$ in some symbolic regression and Boolean function problems. His findings suggests that the best CGP configuration may be problem-dependent.

The aim of this research is to determine whether the hitherto default settings of CGP achieve better accuracy than other configurations in the context of DSP. Chiefly, we compare the elitist selection method used in $(\mu + \lambda)$ with two other less elitist alternatives, as the former has been shown to perform best in other contexts [76]. We also compare acyclic vs. recurrent CGP, the latter having been shown to perform better than the former in other contexts [11].

Finally, we compare the traditional way of initializing and mutating programs by

drawing the function set primitives from a uniform distribution with drawing them from a prior known distribution. In a similar way that there are more function words (articles, conjunctions, etc.) than content words (nouns, adjectives, etc.) in speech, assigning functions to nodes in CGP could be done to reflect an underlying known distribution different from the uniform distribution. While our approach may reduce the likelihood that unexpected, but better, solutions are found, introducing prior knowledge to CGP could improve its efficiency (in convergence time) and its accuracy (in achieved fitness).

A similar approach was used by Ardeh et al. [77] for classical, tree-based GP, using Probabilistic Prototype Trees (PPTs), structures which encode different probability distributions for each tree node, to generate new GP individuals. They also demonstrated that using such distributions improves accuracy over traditional GP, especially in the initial population and subsequent generations. Because the depth of each function node is not fixed in CGP, however, our approach differs from theirs in that a single probability distribution is used for all nodes, and that such a distribution is based on known theoretical solutions rather than being based on prior, GP-based solutions.

Another related approach was recently proposed by Huang et al. [78], who devised a new GP variant called Multi-Population Gene Expression Programming (MP-GEP) which divides a population of individuals into multiple sub-populations and assigns a random probability distribution to each sub-population on initialization. Such distributions then undergo evolution themselves with each subsequent generation based on the makeup of the individuals in the population. As we evaluate our method using known synthesis techniques, however, such random distributions are not needed in our case. That way, a predetermined distribution based on known theoretical solutions can be used to randomly select primitives for all individuals in the population rather than continuously updating and applying it for multiple sub-populations.

5.3 Methods

Our simulations were performed with an updated version of `FaustCGP` [56], a software application that, up to this point, can generate DSP synthesizers in FAUST using CGP (the source code is publicly available at [6]). The updated version allows the setting of any recurrence probability (0 for acyclic CGP and 1 for only recurrent connections), and implements a modified function set that satisfies closure in the interval $[-1, 1]$, values that can be represented in a floating-point audio signal. The current function set is now comprised of arithmetic operations (`sum`, `sub`, `mul`, and `div`); 1st-order Butterworth low- and high-pass filters (`lop` and `hip`, respectively); sine, sawtooth, square, and triangle wave oscillators (`sinp`, `sawp`, `sqrp`, and `trip`, respectively) whose frequency and phase arguments are set as fractions of the Nyquist frequency and 2π, respectively; and Ephemeral Random Constants (ERCs), random values in the range $[-1, 1]$ which remain constant unless the mutation operation is applied to them (`const`). Any graph produced with `FaustCGP` is syntactically valid. Figure 5.1 summarizes the updated implementation of the function set in FAUST for all functions except for the generation of ERCs.

This software application also features two mechanisms to deal with "plateaus" after a local minimum is reached. After a number of generations defined by a "soft plateau," the mutation rate increases five percent each generation until either a new best-fit individual is found, resetting the mutation probability back to its original value, or the mutation probability reaches unity. After a number of generations defined by a "hard plateau," the CGP terminates. The default thresholds for "soft" and "hard" plateaus were set to 10 and 200 generations, respectively.

Different selection methods can also be set. In addition to elitism (dubbed `Elite` here), two more methods are available: one where the best-fit individual among the offspring is selected as the parent for the next generation (`Child`) and weighted random selection (`Rand`), where a parent is chosen with a probability proportional to its fitness relative to other individuals in the same generation. In any case, the best solution across all generations is output.

```
import("stdfaust.lib");

// Helper constants and functions

EPS = ma.EPSILON;
MAX_DELAY = 8192;
NYQUIST = ma.SR / 2;
freq = *(NYQUIST);

pos = abs : max(EPS);
clip(a, b) = min(b) : max(a);
sclip = clip(-1 / EPS, 1 / EPS);
fclip = clip(NYQUIST * 0.0001, NYQUIST * 0.99);
ffreq = freq : abs : fclip;

pdel(f, p) = 2 * ma.frac(p) / pos(f), _ : de.delay(MAX_DELAY);

// Function set (except "rand")

add = + : sclip;
sub = - : sclip;
mul = * : sclip;
div(n, d) = ba.if(d < 0, n / min(-EPS, d), n / max(EPS, d))
    : sclip;

sinp(f, p) = os.oscp(freq(f), p * ma.PI) : sclip;
sawp(f, p) = os.sawtooth(freq(f)) : pdel(f, p) : sclip;
sqrp(f, p) = os.square(freq(f)) : pdel(f, p) : sclip;
trip(f, p) = os.triangle(freq(f)) : pdel(f, p) : sclip;

lop(s, f) = ffreq(f), s : fi.lowpass(1);
hip(s, f) = ffreq(f), s : fi.highpass(1);
```

Figure 5.1: Updated preamble FAUST code defining basic functions for use in RCGP.

5.4 Evaluation

5.4.1 Setup

We attempted to replicate the steady state spectra of 250 entries randomly chosen from the SHARC [54]. Recall that this timbre database is comprised of 1338 entries, each containing frequency components (amplitudes and phases) up to 10 kHz from recordings of 24 orchestral instruments played with different styles at various pitches. The three aforementioned selection methods (`Elite`, `Child`, and `Rand`),

three recurrent connection probabilities (0, 0.25, and 0.5), and two function set distributions (equally probable—`Eq` and weights based on the proportions of an additive synthesizer—`Add`) were considered in the evaluation. A signal generated via additive synthesis is merely a sum of sinusoids, each with their own frequency, phase, and amplitude. For the weighted function case, functions present in the additive synthesizer (`sum`, `mul`, `sinp`, and `const`) were chosen 80 % of the time. `Const` is three times as likely as the other functions to be chosen so that enough numbers are available as arguments to the other functions. The remaining 20 % was equally divided among functions absent in the additive synthesizer. Table 5.1 summarizes the proportions and probabilities of each function in the additive synthesizer and the weighted function set case.

Table 5.1: Proportion of functions in an additive synthesizer ($\boldsymbol{p_1}$), and probabilities for each function to be selected in the weighted function set ($\boldsymbol{p_2}$).

Function	$\boldsymbol{p_1}$	$\boldsymbol{p_2}$
`const`	0.500	0.400
`sum`, `mul`, `sinp`	0.167	0.133
`sub`, `div`, `sawp`, `sqrp`, `trip`, `lop`, `hip`	0	0.029

Fitness was measured as the Euclidean distance between 40 MFCCs [79]. These coefficients were obtained from the CGP-generated audio and those of an additive synthesizer programmed in FAUST that perfectly replicates each SHARC tone. Such coefficients were taken from 48 triangular filter banks equally spaced in the mel scale between 20 – 10240 Hz. As opposed to LSD, we believe that MFCCs provide a better metric with which the perceptual differences between signals can be quantified and minimized, and this metric has been used in other academic literature for similar applications as well (see, for example, [27, 55]).

Other parameters such as the genotype length, initial mutation rate, distance threshold, and the maximum number of generations to evaluate remain constant throughout the evaluation (64, 0.05, 0.1, and 1000, respectively). Finally, a total of 4500 seeded simulations were conducted, one for each combination of timbre, selection method, function set weights, and recurrence probability. The runs were executed on the same hardware and software stack as in the previous evaluation: a Mac mini with 8 GB of

RAM and an 8-core Intel i7-2635QM CPU running Arch Linux, with GNU `parallel` [72] being utilized to minimize computation time.

5.4.2 Results

The results were analyzed with a series of Linear Mixed-Effects Models (LMEMs) eased with the `lme4` library [80] in R [49]. The MFCC distance was set as the dependent variable, while selection method (`select`), prior knowledge initialization (`weights`), recurrence probability (`rec`), and the MIDI note (`pitch`, ranging from C1 to F#7, or approximately $32.7\,\mathrm{Hz}$ to $2.96\,\mathrm{kHz}$) corresponding to a given timbre were considered as explanatory variables. Starting with a model only including timbre (entries in the SHARC database) as a random factor, more complex models were built and accepted if the additional complexity resulted in significantly better fit (with a $95\,\%$ significance level), as assessed with pairwise ANOVA tests. The final model included significant effects of all four explanatory variables and their two-way interactions except that between `select` and `pitch` ($\chi^2_{138} = 128$, $p = 0.720$), as summarized in Table 5.2. All three-way and four-way interactions were not significant.

Table 5.2: ANOVA Type II Wald chi-squared test results for the fitted LMEM.

Factor	$\boldsymbol{\chi^2}$	***df***	***p***
`pitch`	120.1	69	< 0.001
`rec`	1012.6	2	< 0.001
`select`	813.4	2	< 0.001
`weights`	33.3	1	< 0.001
`pitch:rec`	241.4	138	< 0.001
`pitch:weights`	121.0	69	< 0.001
`select:weights`	10.9	2	0.004
`select:rec`	23.4	4	< 0.001
`weights:rec`	14.4	2	< 0.001

A post-hoc analysis based on Tukey's HSD of the EMMs [51] shows that best mean fitness (shortest distance) was achieved when the elitist selection was applied to acyclic CGP, as shown in Figure 5.2a. Fitness worsened with recurrence probability, and differences in fit within levels of recurrence probability increased with less elitist selection

methods, with weighted random selection observing the greatest difference. Figure 5.2b shows that the default, equally distributed, function set yielded worse fit for non-zero recurrence probabilities, and that the differences in fitness achieved by the two function sets also increased with recurrence probability. Figure 5.2c shows that prior knowledge initialization yielded better fit for the elitist selection method, but not when recursion was allowed.

The interactions with pitch are shown in Figure 5.3. This figure indicates that best fits were achieved at low and high pitches, but not for mid-pitches. The weighted function set was more likely to yield better fitness than the default function set for mid-pitches, as shown in Figure 5.3a. In a similar manner, Figure 5.3b indicates that acyclic CGP was more likely to yield better fitness within the same frequency range.

5.4.3 Discussion

Our results are in agreement with previous findings for which the conventional elitist selection method outperforms others. In contrast, we found that acyclic CGP outperformed recurrent CGP in our problem, suggesting that the apparent benefits of RCGP are problem-dependent. We mentioned in Section 2.4 that feedback loops are used in DSP programming to create IIR filters, sawtooth oscillators, among other common functions. Despite this, they worsen the fitness achieved in our case. We speculate two reasons for this outcome: (a) acyclic graphs being more suitable structures for additive synthesis and steady state tones in particular, and (b) an increase in the solution space by introducing feedback loops into candidate programs and therefore increasing the possibility of exploring sub-optimal solutions.

When recursion was allowed, we observed better fitness for the weighted function set compared to the default function set. The largest differences in fitness occurred with elitist selection and a recurrence probability of 0.5 (i.e., when feedforward connections were equally as likely as feedback ones). Thus, our findings support the hypothesis that assigning functions to nodes in CGP based on an underlying known distribution (the proportions found in an additive synthesizer in our case) is more beneficial than assign-

ing them based on a uniform distribution, as it is traditionally performed. But, such benefits are more pronounced with increasing recurrence probability and more elitist selection methods. One possible explanation for this phenomenon is the elimination of length bias in recurrent CGP, where each function node is equally likely to be activated regardless of their position in the chromosome [81]. This in turn allows for longer program phenotypes, and thus more complex tones, to be produced, a finding that is supported by our evaluation. The mean number of active nodes was 7.56 when acyclic CGP was used, compared to 15.63 and 21.18 for recurrence probabilities of 0.25 and 0.5, respectively.

Such short phenotypes, as in our case, may still adversely affect the fitness achieved by the algorithm. As additive synthesis requires a number of nodes proportional to the number of desired harmonics. With the chosen function set, a chromosome length of 64 can ideally hold up to 10 harmonics, which is well insufficient for more complex tones, especially those corresponding to low pitches that may contain hundreds of harmonics. We argued that including harmonically rich functions (as those indicated in Table 5.1) could help to compensate for the lack of nodes. However, our results show no significant differences between the two function sets at low pitches, while the weighted function set was more likely to trump the default function set at mid-pitches. Recall that the latter could assign harmonically rich functions to nodes more freely than the weighted function set. Since the inclusion of these harmonically rich functions did not yield fitness gains for the default function set, we argue that the observed fitness gains of the weighted function set must come from the prior knowledge used in this case. The arbitrary 80 – $20\,\%$ split we imposed in the weighted function set then could hinder the capabilities of this set to find even better fitness. Whether to eliminate such split or modify it is deferred to future research.

Our findings are encouraging and far from being complete. Future investigations will explore the synthesis of non-steady state sounds to determine other CGP parameters that may be context-dependent, or whether other CGP variants perform better for DSP programming in general. Though our focus is on applications of CGP in audio DSP, we

believe that the ideas introduced here (such as the inclusion of prior knowledge through weighting of the function set primitives) may benefit other applications of graphical evolution in general.

5.5 Interim Conclusion

We determined that acyclic CGP, elitist selection, and inclusion of prior knowledge in the initialization and mutation of candidate programs significantly outperformed other evaluated options for steady state additive synthesis, indicating that the superiority of RCGP over CGP cannot be generalized to all problem domains. At least for our audio synthesis problem, increasing the recurrence probability worsens the resulting fitness, in a similar manner that selection methods less elitist than the $(1 + 4)$-ES also does. Modifying the distribution of function set primitives based on prior knowledge also shows promising results as it improves the fitness of candidate programs over the traditional (equal) distribution in our case.

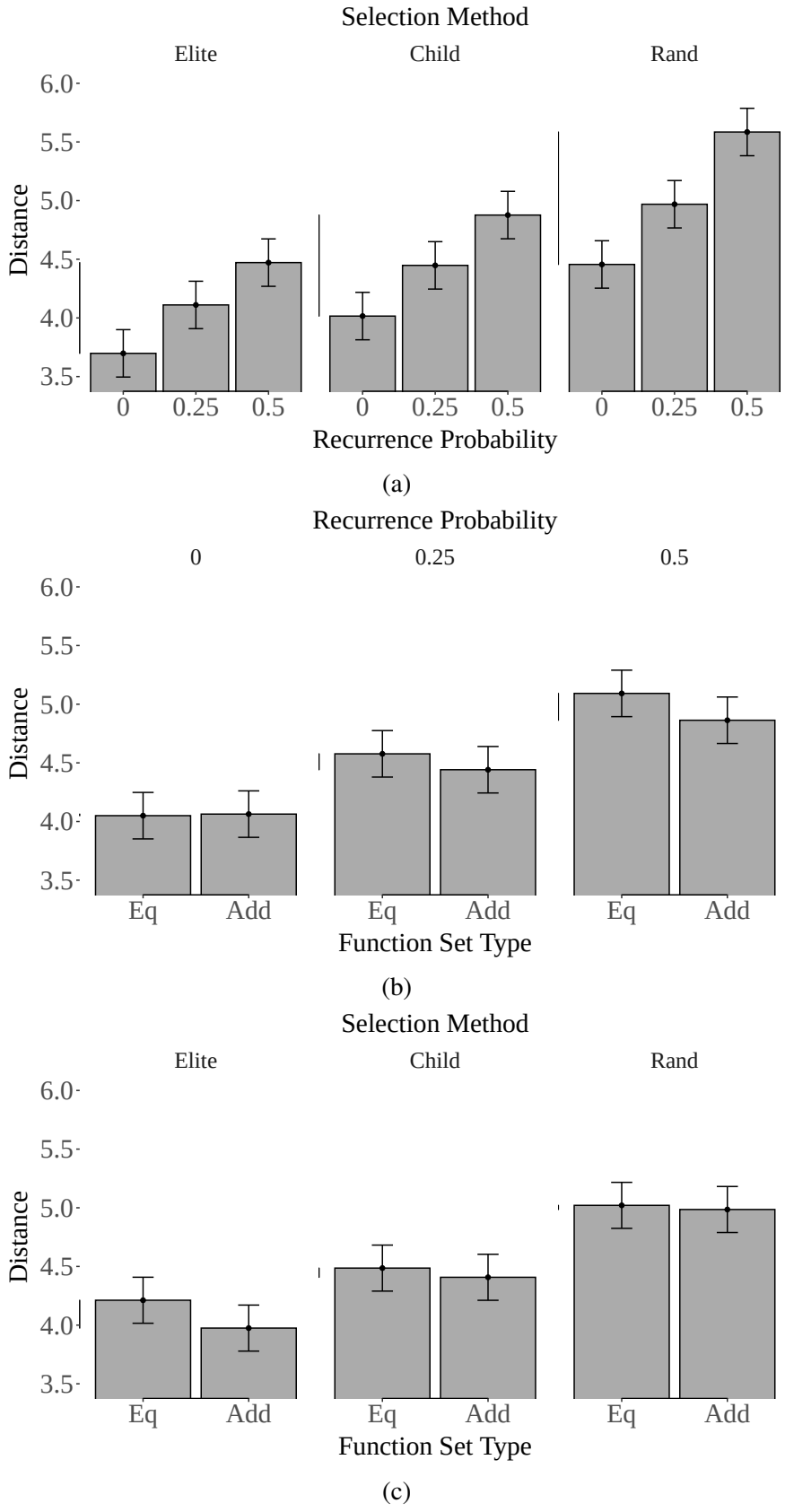
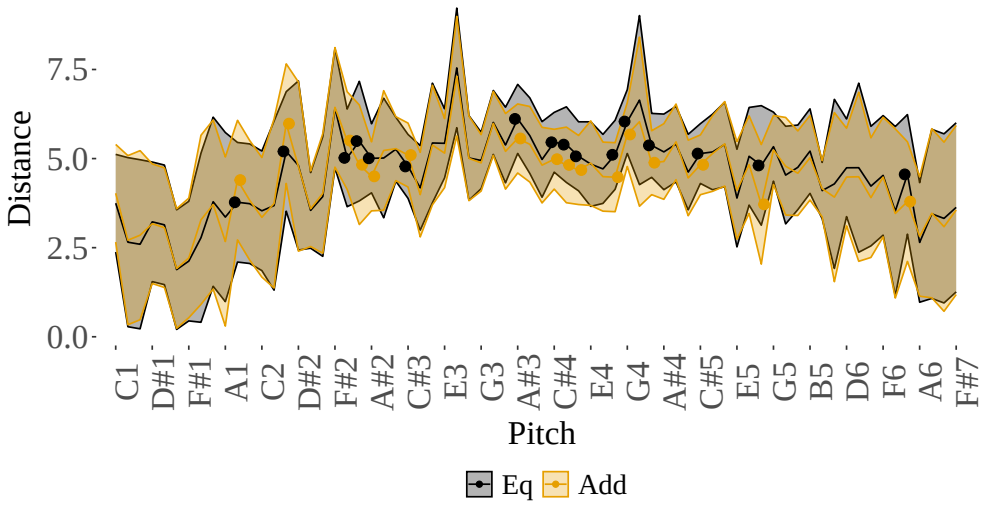
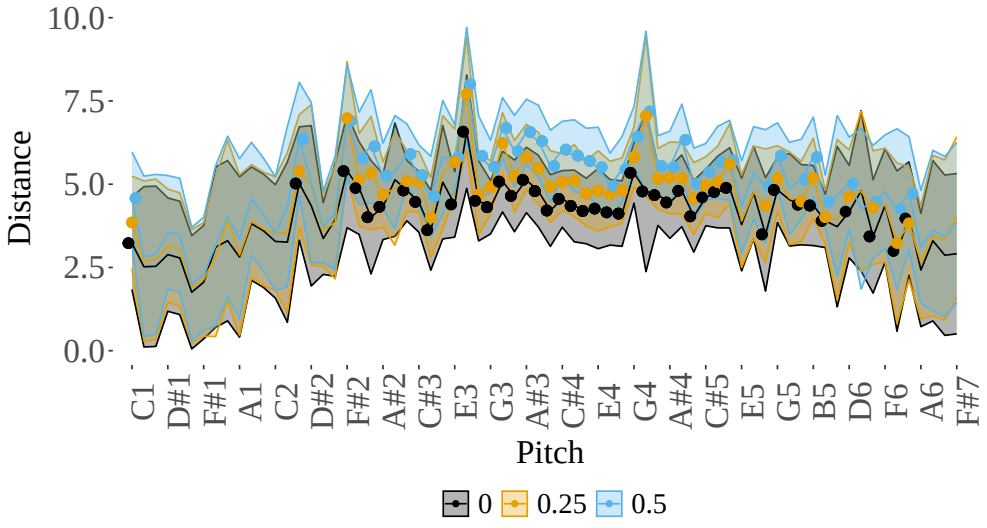


Figure 5.2: Mean MFCC distances and 95 % confidence intervals: (**a**) by recurrence probability and selection method, (**b**) by function set type and recurrence probability, and (**c**) by function set type and selection method.



(a)



(b)

Figure 5.3: Mean MFCC distance by pitch and function set type (**a**), and recurrence probability (**b**). Shaded areas around lines indicate 95 % confidence intervals. Dots indicate where significant differences within pitch were found. Distance between pitch marks are not scaled.

Chapter 6

DSP Filter Design via RCGP

6.1 Introduction

Filtering is a common technique in DSP that is used in various applications including room reverberation, noise reduction, and electronic music production. Such filters can be described in a number of ways, such as difference equations, transfer functions, IRs, or directed graphs visualizing signal flow. They can be used to modulate the magnitude and/or phase of an input signal. Often, these graphs adhere to one of four direct forms which implement the difference equations [82]. Such forms can be cyclic or acyclic, resulting in IIR or Finite Impulse Response (FIR) filters, respectively.

In Chapter 3, we provided an EP method for generating new RIRs in order to add the desired reverberation to an input signal, effectively creating FIR filters in the process. In Chapter 4, we investigated the application of RCGP to additive audio synthesis by using functional programming as a way to treat actual signals and control parameters of the synthesizer as equal. In Chapter 5, we showed that introducing prior knowledge for the initialization and mutation of RCGP was beneficial for the achieved fitness of the algorithm. In a similar manner, we demonstrate in this chapter how RCGP can be used to generate Single-Input/Single-Output (SISO) IIR and FIR digital filters by taking advantage of their directed graph representation.

We evaluate our method by replicating various filters with different (or perhaps

unknown) topologies and orders. As a result, we produce computer programs implementing these filters that can subsequently be used for real-time signal processing, and open a new path for the exploration of real-time digital audio effects. This research is also important because it demonstrates that combining the use of RCGP and functional programming solves some of the hurdles for applying the former in the context of DSP, and illustrates how, by selecting an adequate function set, arbitrary DSP programs can be evolved.

This chapter was first published in [83].

6.2 Background

Many different techniques have been devised to generate close approximations of desired filters, including those that tackle this problem using ML techniques such as neural networks (e.g., [84, 85]) and those using evolutive approaches (e.g., [86]).

A GP-based approach for filter design was proposed by Koza et al. [87] using a population of computer programs represented as tree structures. These programs generate syntactically valid analog electrical circuits, with analog filters being one example of the various circuits that were generated and evaluated. In contrast, CGP would commonly be used to generate digital circuits, often for use as Boolean logic functions or n-bit adder and multiplier circuits (e.g., [88]). In Chapter 3, we provided a rudimentary EP method for generating new RIRs in order to add the desired reverberation to an input signal, effectively creating FIR filters in the process.

A GP method that can generate digital DSP filters in general, however, is to the best of our knowledge yet to be devised. This chapter aims to not only fill this gap, but also take advantage of the recurrent properties of RCGP to theoretically create any desired FIR or IIR filter with ease.

Another attempt at using a quasi-evolutionary approach to generate basic filters was conducted by Santolucito et al. [89], who used a technique called Programming By Example (PBE), which takes a mapping provided by a user between various input signals

with their corresponding output signals, as a way to define and generate basic filters. Initial attempts, however, were restricted to a combination of a Butterworth low-pass filter, a Butterworth high-pass filter, and gain control. Stochastic gradient descent was also used to "evolve" the filter cutoff frequencies and gain amount. A fitness function based on the locations of peaks in the spectrograms of the output signals would then evaluate and find the combination of values that closely approximates the desired mapping. Thus, while PBE itself may not be an evolutionary approach, the evolution of these parameters is a heuristic approach regardless that is akin to EP, a type of EA where the population individuals consist of a set of parameters belonging to a program that is predetermined and remains fixed throughout the evolution.

One disadvantage of EP is that the design of the fixed program limits the variety of solutions that can be created. In the DSP domain, an additive synthesizer may have a difficult time trying to reproduce sounds made with subtractive synthesis, for example. Thus, the use of only Butterworth filters (as in the previous case) is merely an arbitrary limit imposed onto the solution space of IIR filters. Furthermore, Butterworth filters are a sub-class of analog IIR filters that can be transformed into digital filters using the bilinear transform [82]. This suggests that our method is capable of generating not only digital Butterworth filters, but also any filter with some arbitrary transfer function. In addition, while the use of a mapping between different input and output audio signals may be intended to make the system more accessible for end users, such a mapping is excessively large when, in reality, just a single IR is sufficient. Our proposed system addresses these issues by using a CGP-based approach instead, adding more flexibility to the kinds of solutions that can be generated while minimizing the amount of user input (information about the target filter) as much as possible.

6.3 Methods

We extend the functionality of our `FaustCGP` software application, previously discussed in Chapter 4, to include the evolution and generation of any target IIR or FIR

filter in FAUST. To replicate the direct form design of such filters, we start with a function set comprising four basic components for the difference equations of any direct form digital filter: addition, multiplication, one-sample delay, and ERCs. The latter are random values in the range of $[-1, 1]$ that will ideally be associated with the real poles and zeros of a filter. The output of each operation is also clipped to $[-1, 1]$ as necessary so that the function set satisfies the closure property within this interval, which is also the range of values that can be represented in a floating-point audio signal.

The direct forms often require a number of one-sample delays to be chained in parallel, which may be more difficult to replicate through evolution the more filter taps are required. Thus, we add an additional fractional delay line (with interpolation for non-integer delays) that takes as arguments both the signal to delay and the number of samples to delay expressed as an absolute fraction of the delay line length that must be arbitrarily set at compile time. Initially, we have set this maximum to 8192 samples.

To evaluate the fitness of candidate programs, the DSP graphs are first converted into FAUST code, using an algorithm based on that provided by Ren et al. [70] that is capable of handling any number of feedback loops in the graphs, before being compiled via `libfaust` using the LLVM backend API. The digital unit impulse function ($\delta : \mathbb{Z} \to \mathbb{R}$, where $\delta(n) = 1$ if $n = 0$ and $\delta(n) = 0$ otherwise) is then given as an input signal to the resulting executable. That way, the IR of each candidate program is output.

We then return to the LSD

$$d = \sqrt{\frac{1}{b-a+1} \sum_{k=a}^{b} \left(10 \log_{10} \frac{X_1(k)}{X_2(k)} \right)^2} \quad [\mathrm{dB}], \tag{6.1}$$

as a cost function to evaluate the magnitude spectra of the IRs of each candidate program. Unlike the previous chapters, we computed the LSD over the entire magnitude response (between the DC and the Nyquist frequency, exclusive). This metric is commonly used in applications that attempt to minimize such deviations as well (see [44] for a non-exhaustive list).

6.4 Evaluation

6.4.1 Setup

For this evaluation, We first implemented in FAUST a basic low-pass FIR filter sampled at $44.1\,\mathrm{kHz}$, with its transfer function given by $H(z) = 1 + 0.75z^{-1}$ (i.e., a single zero at -0.75 along the real axis of the z-plane). For the second filter, we selected the minimum-phase inverse filter of an Optimus Pro 7 loudspeaker, which is provided as part of a database of HRIR measurements released by Gardner and Martin [90]. In either case, their magnitude responses (in decibels relative to overload) over the first 512 samples were calculated, and the data was stored as text files to be used by the RCGP algorithm for comparison with the candidate filters.

The genotype length (the maximum number of function nodes allowed per chromosome) was set to 64 function nodes for the one-zero filter. Miller and Smith [91] suggested that the phenotype length (the number of function nodes that actively contribute to the output of the program) be $< 5\,\%$ of the genotype length for best performance. By this logic, a genotype length of approximately 41000 is recommended for the inverse filter assuming an FIR filter design of order 512. Such a length, however, results in DSP graphs that are currently too large to compile with our current hardware, so a limit of 1000 nodes was used instead.

Kalkreuth [75] also found that the optimal values for each RCGP parameter are dependent on the context of the problem to which they are applied. Thus, we decided to evaluate the effects of recurrence probability on the fitness of candidate filter programs to determine which values, if any, are optimal for each target filter. Such probabilities are also known to affect the amount of length bias [12, 81], a phenomenon in acyclic CGP where function nodes closer to the input nodes are more likely to be active than those closer to the output nodes. This probability subsequently affects the percentage of active nodes in the candidate graphs as well as their compilation time. For our evaluation, recurrence probabilities of 0 to 0.5, in increments of 0.1, were selected for each filter. Finally, 100 runs were executed for each combination of recurrence probability

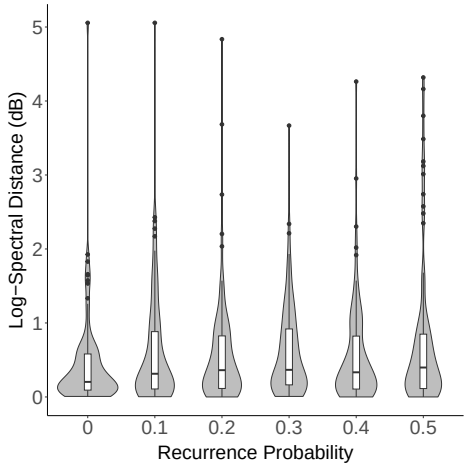


Figure 6.1: Distributions of LSD values by recurrence probability when replicating the one-zero filter. Violin plots show the shape of these distributions. Overlaid box-plots indicate intermediate quartiles.

and target filter. To minimize the overall computation time, these runs were executed via GNU `parallel` [72], allowing all CPU cores to be utilized at the same time with each core being given its own RCGP process. All runs were executed on the same hardware as in previous chapters: a Mac mini (running Arch Linux) with $8\,\mathrm{GB}$ of RAM and an 8-core Intel i7-2635QM CPU.

6.4.2 Results

Fig. 6.1 reports the distributions of LSD values achieved for the one-zero filter grouped by recurrence probability. This figure shows that our method was successful at replicating short filters, with the final LSD values of most runs being $< 1\,\mathrm{dB}$. Among the 600 total runs, 20 of them finished with LSD values of $< 0.01\,\mathrm{dB}$, which was the threshold set for early termination by fitness. Increasing the recurrence probability does not seem to affect the distances achieved in this case, suggesting that the RCGP process is still able to find appropriate solutions for short filters regardless of whether or not the solution space includes IIR filters.

Figs. 6.2 and 6.3 show the distances and best-achieved magnitude responses, respectively, given the Optimus Pro 7 minimum-phase inverse filter as the target. Here, the best filter found had an LSD $= 1.486\,\mathrm{dB}$, obtained from a run where the recurrence probability was set to 0.4. By comparison, these figures show that our method was

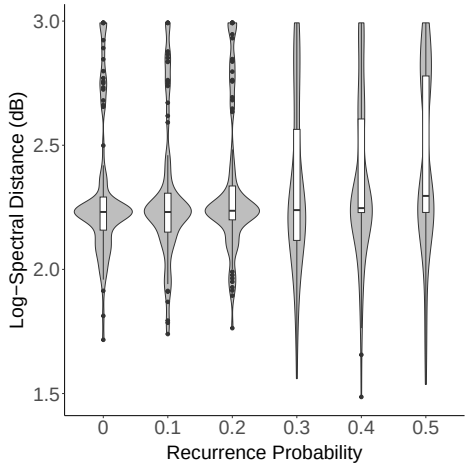


Figure 6.2: Violin plots of LSD values by recurrence probability for the Optimus Pro 7 minimum-phase inverse filter case.

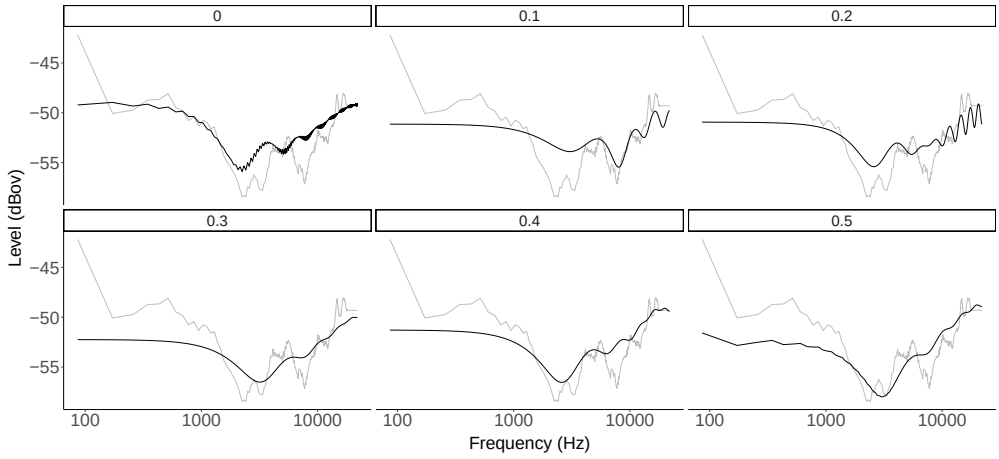


Figure 6.3: IR magnitude spectra of the best fit filters by recurrence probability (black lines) compared to that of the Optimus Pro 7 minimum-phase inverse filter (gray lines).

Table 6.1: Minimum LSD values in decibels achieved for each recurrence probability and target filter.

Recurrence Prob.	**Filter**	
	One-Zero	**Optimus 7**
0	0.007	1.716
0.1	0.003	1.740
0.2	< 0.001	1.763
0.3	0.007	1.560
0.4	< 0.001	1.486
0.5	< 0.001	1.537

less successful at replicating much longer filters. This is also evidenced by the distinct mismatch between the magnitude responses of the target and the artificial filters. As in

the case of the simpler filter, the recurrence probability setting does not seem to have a significant effect on spectral distances, nor on the overall magnitude responses of the artificial filters output (at least in terms of adequately matching the target filter). Table 6.1 reports the best LSD values obtained for each recurrence probability per filter.

6.4.3 Discussion

The results show that the distances achieved by our method depend on the order of the target filter, especially if the genotype length does not scale in accordance with such complexity as well. While the 1st-order filter can be replicated with relative ease, the same cannot be said for higher-order filters.

Genotype lengths that become too large will quickly become unfeasible to compute with our current hardware. As mentioned, the RCGP process seems to work best when the average phenotype length is $< 5\,\%$ of the genotype length, so a genotype length of 1000 nodes would suffice for IIR filters up to a maximum order of 8 (a direct-form II implementation with none of the poles and zeros being equal to 0 requires a minimum of 51 nodes). Thus, the chances that the RCGP process would find and recreate this exact structure (or any of the four direct forms) may also decrease as the number of dimensions in the solution space increases with every increase in filter order. Nevertheless, the observed results for the inverse filter are encouraging since the evolved filters approximate it reasonably despite its limited genotype length.

The recurrence probability seems to have no effect on the mean fitness of the generated filters. Although, for the inverse filter, we see a different amount of variance as well as smaller LSD values found for recurrence probabilities $>= 0.3$ compared to those < 0.3. While this finding supports the hypothesis that the optimal RCGP parameter values are problem-dependent, it may seem odd that this particular behavior is two-tiered rather than gradual. One could hypothesize that the feedback loops in IIR filters should reduce the number of nodes required to represent the filter as well as the total order compared to FIR filters. However, such reductions in filter representation may be counteracted by the fact that including recurrent connections increases the size

of the solution space due to the addition of IIR filters. This drastically decreases the likelihood of finding a correct solution through evolution.

The specified genotype length may also be a factor as it places a hard limit on the size or order of the filters that can be produced as well. For filters that are automatically programmed through evolutionary means, there are some properties such as the order of the generated filters that are more difficult to control. In situations where only the IR of the target filter is known, one could implement a matching digital filter as an FIR filter. In order to reduce the latency of this system, however, it would be ideal to return an equivalent IIR filter whose order is less than that of the target filter. Due to the random nature of the evolved solutions, however, there also exists the possibility that the order of the best fit solution is larger than that of the target filter. If desired, the fitness function may be modified to penalize higher-order filters, but such possibilities cannot be eliminated entirely in an evolutionary system.

The phase response is also currently not considered when evaluating the fitness of candidate filters. Therefore, all-pass filters cannot be synthesized with our current method since all-pass filters with different cut-off frequency will have the same magnitude response but different phase responses. Still, the fitness function can be extended so that both the magnitude and phase spectra are considered.

6.5 Interim Conclusion

We introduced an extension to the `FaustCGP` framework whereby IIR filters can be generated to approximate the magnitude response of any desired filter or IR. Two arbitrary filters of differing orders were chosen as targets for replication with our software, and our experiments showed that the accuracy of the genetic solutions is dependent on the complexity of the target solutions, which can be related to the order of the target filter or the parameter values chosen for the RCGP evolution. The recurrence probability setting in particular only affects the variance in the fitness of the evolved solutions rather than the mean, further supporting the context-dependence of RCGP parameters.

Chapter 7

Discussion

Over the past three chapters, we demonstrated how RCGP can be applied to the automatic program induction of both DSP synthesizers and DSP audio effects, as well as demonstrated a few research problems in the DSP domain to which this method can be applied. In particular, we evaluated the effectiveness of RCGP on the matching of both steady state sounds and IIR/FIR filters in the digital domain. Future work may extend these evaluations to other DSP applications and research problems Regardless, the simple fact that this single method can be used in a wide variety of problem domains highlights how flexible and extensible CGP can be compared to other AI methods, and this is all thanks to a cross-platform and cross-architecture programming language that can be described as flexible and extensible itself as well.

The goal of this research is to advance progress into the applications of EAs in DSP programming, and for that, we believe we have succeeded in that goal, making significant contributions in the field while inspiring others to do the same. That being said, it is also important to recognize that EAs have their own strengths and weaknesses in relation to ML techniques. For one, the fact that EAs do not require any training data can be seen as a double-edged sword. While the lack of such data eliminates any concerns regarding training bias or data security and privacy, this comes at a cost of executing all computations on-demand rather than in the training stage. This means that EAs are unlikely to be useful for real-time DSP applications in the near future, as they

may not be able to produce chromosomes with satisfactory fitness within the acceptable limit for audio latency, which is approximately $70\,\mathrm{ms}$ depending on the stimuli before the precedence effect can be felt [92]. This especially holds true for EP applications in particular where the parameters of a DSP synthesizer or audio effect are subject to real-time automation.

For DSP applications that do not require real-time synthesis, such as the programming of the synthesizers or audio effects themselves, such concerns no longer apply. While further evaluations may be needed to compare the performance of both evolutionary and ML methods given the same inputs, as previously demonstrated, we believe that EAs can be just as viable as ML for such tasks. Furthermore, we have demonstrated a general-purpose evolutionary framework that can be applied to multiple DSP research applications at once, something that ML cannot accomplish as they have no knowledge of any problem domains or data points outside of those they are trained upon.

Nevertheless, we do not claim that our framework is state-of-the-art either, as such a generalization comes at a cost of having a much larger solution space in which certain solutions must be found. Still, the advantage of having such a space to search through is that prior knowledge about the desired solution is helpful but not required in the evolutionary search. For instance, previous ML or EP methods for sound re-synthesis [57–65] are limited to a subset of synthesis techniques with which a target sound is assumed to use, so sounds generated using other techniques would be unsuitable for replication using such methods.

Our framework can also be easily extended by defining new functions to add to the function set as previously mentioned in Section 4.5. Just about any function that is available or can be implemented in FAUST can be added in a similar fashion to the existing functions, scaling the range of input parameter values and clipping the output signals in a similar manner as necessary. Adding such sub-modules to the function set would then allow for even more complex programs to be created and found. Furthermore, this function set can be as small or large as necessary to fit the known (or unknown) requirements of the solution to be found.

Chapter 8

Conclusion

We introduced a general-purpose evolutionary framework for the automatic program induction of any DSP program, and demonstrated several research applications to which this framework can be applied. We started with the EP of FIR filters as RIRs for use in a real-time convolution reverb application before generalizing the task to include the generation of any digital filter or synthesizer. We also publicly provide some prototype implementations of our applications via source code as free software to allow future researchers to both verify and build upon our work. While the accuracy of the output programs depends on many factors related to the parameters chosen for the framework, some of which may even be unfeasible to perform with current hardware, we have nevertheless demonstrated that this framework is a viable alternative to ML for solving many DSP research problems, removing the need for large datasets and eliminating all of the pitfalls associated with such data.

We hope that the introduction of these methods will lead to future research and new developments in both DSP program synthesis as well as Cartesian Genetic Programming, as there are still many open issues and questions concerning both DSP and GP that are yet to be solved. The intersection of these fields is one that is yet to be explored, but shows a lot of promise in the future as new DSP programs or synthesis techniques may be discovered as a result of this evolutionary search, leading to the advancement of new knowledge in ways that other AI methods cannot.

References

[1] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning representations by back-propagating errors," *Nature*, vol. 323, no. 6088, pp. 533–536, Oct. 1986, DOI: 10.1038/323533a0.

[2] O. Barkan, D. Tsiris, O. Katz, and N. Koenigstein, "Inversynth: Deep estimation of synthesizer parameter configurations from audio signals," *IEEE/ACM Trans. on Audio, Speech, and Lang. Process.*, vol. 27, no. 12, pp. 2385–2396, 2019, DOI: 10.1109/TASLP.2019.2944568.

[3] C. Donahue, J. McAuley, and M. Puckette, "Adversarial audio synthesis," in *Proc. 7th Int. Conf. on Learning Representations*, May 2019. [Online]. Available: https://openreview.net/forum?id=ByMVTsR5KQ

[4] C. Darwin, *On the Origin of Species by Means of Natural Selection, or the Preservation of Favoured Races in the Struggle for Life*. John Murray, 1859.

[5] E. Ly, "Genetic Reverb," Software, 2020, available from `https://github.com/edward-ly/GeneticReverb` (Sep. 23, 2022).

[6] ——, "FaustCGP," Software, 2022, available from `https://sr.ht/~led/FaustCGP/` (Sep. 23, 2022).

[7] A. Eiben and J. Smith, *Introduction to Evolutionary Computing*, 2nd ed., ser. Natural Computing Series. Berlin, Heidelberg: Springer, 2015, DOI: 10.1007/978-3-662-44874-8.

[8] J. R. Koza, *Genetic Programming: On the Programming of Computers by Means of Natural Selection*, ser. Complex Adaptive Systems. London: MIT Press, 1992. [Online]. Available: https://mitpress.mit.edu/books/genetic-programming

[9] J. F. Miller, "An empirical study of the efficiency of learning boolean functions using a Cartesian genetic programming approach," in *Proc. 1st Annual Conf. Genetic and Evol. Comput. - Vol. 2 (GECCO'99)*. Morgan Kaufmann, 1999, pp. 1135–1142, DOI: 10.5555/2934046.2934074.

[10] J. F. Miller and P. Thomson, "Cartesian genetic programming," in *Proc. European Conf. on Genetic Programming*. Berlin, Heidelberg: Springer-Verlag, 2000, pp. 121–132, DOI: 10.5555/646808.704075.

[11] A. J. Turner and J. F. Miller, "Recurrent Cartesian genetic programming," in *Parallel Problem Solving from Nature–XIII*, T. Bartz-Beielstein, J. Branke, B. Filipič, and J. Smith, Eds. Springer, 2014, pp. 476–486, DOI: 10.1007/978-3-319-10762-2_47.

[12] J. Miller, "Cartesian genetic programming: its status and future," *Genetic Programming and Evolvable Machines*, vol. 21, no. 1–2, pp. 129–168, 2020, DOI: 10.1007/s10710-019-09360-6.

[13] R. Poli, W. B. Langdon, and N. F. McPhee, *A Field Guide to Genetic Programming*. Lulu, 2008, with contributions by J. R. Koza. [Online]. Available: http://www.gp-field-guide.org.uk

[14] S. Silva and E. Costa, "Dynamic limits for bloat control in genetic programming and a review of past and current bloat theories," *Genetic Programming and Evolvable Machines*, vol. 10, no. 2, pp. 141–179, 2009, DOI: 10.1007/s10710-008-9075-9.

[15] A. J. Turner and J. F. Miller, "Cartesian genetic programming: Why no bloat?" in *Genetic Programming*, M. Nicolau, K. Krawiec, M. I. Heywood, M. Castelli, P. García-Sánchez, J. J. Merelo, V. M. Rivas Santos, and K. Sim, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, pp. 222–233, DOI: 10.1007/978-3-662-44303-3_19.

[16] SILSO World Data Center, "The International Sunspot Number," *International Sunspot Number Monthly Bulletin and Online Catalogue*, 1700–1987. [Online]. Available: http://www.sidc.be/silso/,

[17] D. Jefferson, R. Collins, C. Cooper, M. Dyer, M. Flowers, R. Korf, C. Taylor, and A. Wang, "Evolution as a theme in artificial life: The genesys/tracker system," in *Artificial Life II: Proc. 2nd Workshop on Artificial Life*, C. G. Langton, C. Taylor, J. D. Farmer, and S. Rasmussen, Eds. Redwood City, CA: Addison-Wesley, 1990, pp. 549–578.

[18] S. Todd and W. Latham, "The mutation and growth of art by computers," in *Evolutionary Design by Computers*, ser. Evolutionary Design by Computers, P. J. Bentley, Ed. Morgan Kaufmann, 1999.

[19] N. Collins, "Experiments with a new customisable interactive evolution framework," *Organised Sound*, vol. 7, no. 3, pp. 267–273, 2002, DOI: 10.1017/S1355771802003060.

[20] J. McCartney, "SuperCollider: a new real time synthesis language," in *Proc. Int. Computer Music Conf.*, vol. 1996. Michigan Publishing, 1996, pp. 257–258.

[21] J. Fornari, A. Maia, and J. Manzolli, "Interactive soundscape design with evolutionary sound processing," in *Proc. Int. Computer Music Conf.*, vol. 2007. Michigan Publishing, 2007, pp. 427–430.

[22] ——, "Interactive spatialization and sound design using an evolutionary system," in *Proc. 7th Int. Conf. on New Interfaces for Musical Expression*, ser. NIME '07. New York: ACM, 2007, pp. 293–298, DOI: 10.1145/1279740.1279803.

[23] M. Puckette, "Pure Data: another integrated computer music environment," *Proc. 2nd Intercollege Computer Music Concerts*, pp. 37–41, 1996.

[24] C. Moler, "Design of an interactive matrix calculator," in *Proc. National Computer Conf.*, ser. AFIPS '80. New York: ACM, May 1980, pp. 363–368, DOI: 10.1145/1500518.1500576.

[25] J. Villegas, "Locating virtual sound sources at arbitrary distances in real-time binaural reproduction," *Virtual Reality*, vol. 19, no. 3, pp. 201–212, Oct 2015.

[26] V. Best, R. Baumgartner, M. Lavandier, P. Majdak, and N. Kopčo, "Sound externalization: A review of recent research," *Trends in Hearing*, vol. 24, p. 2331216520948390, 2020.

[27] M. Macret and P. Pasquier, "Automatic design of sound synthesizers as Pure Data patches using coevolutionary mixed-typed Cartesian genetic programming," in *Proc. Annual Conf. Genetic and Evol. Comput. (GECCO '14)*, 2014, pp. 309–316, DOI: 10.1145/2576768.2598303.

[28] Y. Orlarey, D. Fober, and S. Letz, "FAUST: an efficient functional approach to DSP programming," in *New Computational Paradigms for Computer Music*, G. Assayag and A. Gerzso, Eds. France: Delatour, 2009, pp. 65–96.

[29] E. Ly and J. Villegas, "Genetic Reverb: Synthesizing artificial reverberant fields via genetic algorithms," in *Proc. 9th Int. Conf. on Artificial Intelligence in Music, Sound, Art and Design (EvoMUSART 2020)*, J. Romero, A. Ekárt, T. Martins, and J. Correia, Eds. Cham: Springer, Apr. 2020, pp. 90–103, DOI: 10.1007/978-3-030-43859-3_7.

[30] ——, "Generating artificial reverberation via genetic algorithms for real-time applications," *Entropy*, vol. 22, no. 11, pp. 1–19, Nov. 2020, DOI: 10.3390/e22111309.

[31] J. Lochner and J. Burger, "The intelligibility of speech under reverberant conditions," *Acta Acustica united with Acustica*, vol. 11, no. 4, pp. 195–200, 1961.

[32] V. Välimäki, J. D. Parker, L. Savioja, J. O. Smith, and J. S. Abel, "Fifty years of artificial reverberation," *IEEE Trans. on Audio, Speech, and Language Processing*, vol. 20, no. 5, pp. 1421–1448, 2012.

[33] M. R. Schroeder, "Natural sounding artificial reverberation," *J. Audio Eng. Soc.*, vol. 10, no. 3, pp. 219–223, Jul. 1962. [Online]. Available: https://www.aes.org/e-lib/browse.cfm?elib=18823

[34] M. R. Schroeder and B. F. Logan, "'Colorless' artificial reverberation," *IRE Trans. Audio*, vol. AU-9, no. 6, pp. 209–214, 1961, DOI: 10.1109/TAU.1961.1166351.

[35] U. Zölzer, Ed., *DAFX – Digital Audio Effects*, 2nd ed. New York: John Wiley & Sons, 2011.

[36] W. G. Gardner, "Efficient convolution without input/output delay," in *Proc. 97th Audio Eng. Soc. Conv.*, Nov. 1994. [Online]. Available: http://www.aes.org/e-lib/browse.cfm?elib=6335

[37] J. B. Allen and D. A. Berkley, "Image method for efficiently simulating small-room acoustics," *J. Acoust. Soc. of America*, vol. 65, no. 4, pp. 943–950, 1979.

[38] E. Habets, "Room impulse response generator," pp. 1–17, Jan. 2006. [Online]. Available: https://www.audiolabs-erlangen.de/fau/professor/habets/software/rir-generator

[39] U. Kristiansen, A. Krokstad, and T. Follestad, "Extending the image method to higher-order reflections," *Applied Acoustics*, vol. 38, no. 2–4, pp. 195–206, 1993.

[40] S. McGovern, "Fast image method for impulse response calculations of box-shaped rooms," *Applied Acoustics*, vol. 70, pp. 182–189, Jan. 2009, DOI: 10.1016/j.apacoust.2008.02.003.

[41] ISO 3382-1:2009, "Acoustics – Measurement of room acoustic parameters – Part 1: Performance spaces," International Organization for Standardization, Geneva, CH, Standard, Jun. 2009. [Online]. Available: https://www.iso.org/standard/40979.html

[42] L. Beranek, *Concert Halls and Opera Houses: Music, Acoustics, and Architecture*, 2nd ed. Springer, 2004.

[43] D. T. Murphy and S. Shelley, "OpenAIR: An interactive auralization web resource and database," in *Proc. 129th Audio Eng. Soc. Conv.*, Nov. 2010. [Online]. Available: http://www.aes.org/e-lib/browse.cfm?elib=15648

[44] S. Cecchi, A. Carini, and S. Spors, "Room Response Equalization—A Review," *Applied Sciences*, vol. 8, no. 1, p. 16, 2018.

[45] J. O. Smith, *Physical Audio Signal Processing*. `http://ccrma.stanford.edu/~jos/pasp/`, retrieved February 20, 2024, online book, 2010 edition.

[46] V. Pulkki and M. Karjalainen, *Communication Acoustics: An Introduction to Speech, Audio and Psychoacoustics*, 1st ed. John Wiley & Sons, 2015.

[47] J. L. Hintze and R. D. Nelson, "Violin plots: A box plot-density trace synergism," *The American Statistician*, vol. 52, no. 2, pp. 181–184, 1998, DOI: 10.2307/2685478. [Online]. Available: http://www.jstor.org/stable/2685478

[48] Søren Bech and Nick Zacharov, *Perceptual Audio Evaluation– Theory, Method and Application*. West Sussex, England: John Wiley & Sons, 2006.

[49] R Core Team, *R: A Language and Environment for Statistical Computing*, R Foundation for Statistical Computing, Vienna, Austria, 2023. [Online]. Available: https://www.R-project.org/

[50] A. Ferrari and M. Comelli, "A comparison of methods for the analysis of binomial clustered outcomes in behavioral research," *J. Neuroscience Methods*, vol. 274, pp. 131–140, 2016.

[51] R. V. Lenth, *emmeans: Estimated Marginal Means, aka Least-Squares Means*, 2023, R package version 1.8.5. [Online]. Available: https://CRAN.R-project.org/package=emmeans

[52] C. C. J. M. Hak, R. H. C. Wenmaekers, and L. C. J. van Luxemburg, "Measuring room impulse responses: Impact of the decay range on derived room acoustic parameters," *Acta Acustica united with Acustica*, vol. 98, no. 6, pp. 907–915, 2012, DOI: 10.3813/AAA.918574.

[53] Ableton, "Live," Software, 2020, available from `https://www.ableton.com/` (February 20, 2024).

[54] G. J. Sandell, "A library of orchestral instrument spectra," in *Proc. Int. Computer Music Conf.*, vol. 1991. Michigan Publishing, 1991, pp. 98–101.

[55] I. Ibnyahya and J. D. Reiss, "A method for matching room impulse responses with feedback delay networks," in *Proc. 153rd Audio Eng. Soc. Conv.*, Oct. 2022. [Online]. Available: http://www.aes.org/e-lib/browse.cfm?elib=21917

[56] E. Ly and J. Villegas, "Additive synthesis via recurrent Cartesian genetic programming in FAUST," in *Proc. 153rd Audio Eng. Soc. Conv.*, Oct. 2022, pp. 1–7. [Online]. Available: http://www.aes.org/e-lib/browse.cfm?elib=21954

[57] A. Horner, J. Beauchamp, and L. Haken, "Methods for multiple wavetable synthesis of musical instrument tones," *J. Audio Eng. Soc.*, vol. 41, no. 5, pp. 336–356, 1993.

[58] A. Horner, "Wavetable matching synthesis of dynamic instruments with genetic algorithms," *J. Audio Eng. Soc.*, vol. 43, no. 11, pp. 916–931, 1995.

[59] A. Horner, J. Beauchamp, and L. Haken, "Machine tongues XVI: Genetic algorithms and their application to FM matching synthesis," *Computer Music J.*, vol. 17, no. 4, pp. 17–29, 1993, DOI: 10.2307/3680541.

[60] A. Horner, "Double-modulator FM matching of instrument tones," *Computer Music J.*, vol. 20, no. 2, pp. 57–71, 1996, DOI: 10.2307/3681332.

[61] B. T. G. Tan and S. M. Lim, "Methods for multiple wavetable synthesis of musical instrument tones," *J. Audio Eng. Soc.*, vol. 44, no. 1/2, pp. 3–15, 1996.

[62] A. Horner, "Nested modulator and feedback FM matching of instrument tones," *IEEE Trans. Speech Audio Process.*, vol. 6, no. 4, pp. 398–409, 1998, DOI: 10.1109/89.701371.

[63] S. M. Lim and B. T. G. Tan, "Performance of the genetic annealing algorithm in dfm synthesis of dynamic musical sound samples," *J. Audio Eng. Soc.*, vol. 47, no. 5, pp. 339–354, 1999.

[64] K. Tatar, M. Macret, and P. Pasquier, "Automatic synthesizer preset generation with presetgen," *J. New Music Research*, vol. 45, no. 2, pp. 124–144, 2016, DOI: 10.1080/09298215.2016.1175481.

[65] M. J. Yee-King, L. Fedden, and M. d'Inverno, "Automatic programming of VST sound synthesizers using deep networks and other techniques," *IEEE Trans. on Emerg. Topics in Computat. Intell.*, vol. 2, no. 2, pp. 150–159, 2018, DOI: 10.1109/TETCI.2017.2783885.

[66] J. Engel, C. Resnick, A. Roberts, S. Dieleman, M. Norouzi, D. Eck, and K. Simonyan, "Neural audio synthesis of musical notes with WaveNet autoencoders," in *Proc. 34th Int. Conf. on Machine Learning*, vol. 70, Aug. 2017, pp. 1068–1077. [Online]. Available: https://proceedings.mlr.press/v70/engel17a.html

[67] D. Sanfilippo, "Handling infinity and not-a-number (nan) values in Faust and c++ audio programming," Dec. 2020, accessed: Jan. 28, 2024. [Online]. Available: https://www.dariosanfilippo.com/posts/2020/12/28/handling_inf_nan_values_in_faust_and_cpp.html

[68] A. J. Turner and J. F. Miller, "Introducing a cross platform open source Cartesian genetic programming library," *Genetic Programming and Evolvable Machines*, vol. 16, pp. 83–91, Mar. 2014, DOI: 10.1007/s10710-014-9233-1.

[69] M. Frigo and S. G. Johnson, "The design and implementation of FFTW3," *Proc. of the IEEE*, vol. 93, no. 2, pp. 216–231, 2005, Special Issue on "Program Generation, Optimization, and Platform Adaptation".

[70] S. Ren, L. Pottier, and M. Buffa, "From Diagram to Code: a Web-based Interactive Graph Editor for Faust DSP Design and Code Generation," in *Proc. 2nd Int. Functional Audio Stream (Faust) Conf.*, Dec. 2020. [Online]. Available: https://hal.inria.fr/hal-03087778

[71] C. Roads, "A conversation with James A. Moorer," *Computer Music J.*, vol. 6, no. 4, pp. 10–21, 1982. [Online]. Available: http://www.jstor.org/stable/4617880

[72] O. Tange, "GNU parallel: The command-line power tool," *;login: The USENIX Magazine*, vol. 36, no. 1, pp. 42–47, Feb. 2011. [Online]. Available: https://www.usenix.org/publications/login/february-2011-volume-36-number-1/gnu-parallel-command-line-power-tool

[73] E. Ly and J. Villegas, "Cartesian genetic programming parameterization in the context of audio synthesis," *IEEE Signal Process. Lett.*, vol. 30, pp. 1077–1081, Aug. 2023, DOI: 10.1109/LSP.2023.3304198.

[74] A. Horner, "Auto-programmable FM and wavetable synthesizers," *Contemporary Music Review*, vol. 22, no. 3, pp. 21–29, 2003, DOI: 10.1080/0749446032000150852.

[75] R. Kalkreuth, "A comprehensive study on subgraph crossover in Cartesian genetic programming," in *Proc. 12th Int. Joint Conf. on Computational Intelligence (IJCCI 2020)*, J. J. Merelo, J. Garibaldi, C. Wagner, T. Bäck, K. Madani, and K. Warwick, Eds. SciTePress, 2020, pp. 59–70, DOI: 10.5220/0010110700590070.

[76] J. F. Miller and P. Thomson, "Aspects of digital evolution: Geometry and learning." in *Evolvable Systems: From Biology to Hardware*, 1998, pp. 25–35, DOI: 10.1007/BFb0057604.

[77] M. A. Ardeh, Y. Mei, and M. Zhang, "Genetic programming hyper-heuristics with probabilistic prototype tree knowledge transfer for uncertain capacitated arc routing problems," in *2020 IEEE Congress on Evol. Comput. (CEC)*, Jul. 2020, pp. 1–8, DOI: 10.1109/CEC48606.2020.9185714.

[78] Z. Huang, J. Zhong, W. Liu, and Z. Wu, "Multi-population genetic programming with adaptively weighted building blocks for symbolic regression," in *Proc. Genetic and Evol. Comput. Conf. Companion*, 2018, pp. 266–267, DOI: 10.1145/3205651.3205673.

[79] H. Terasawa, M. Slaney, and J. Berger, "The thirteen colors of timbre," in *IEEE Wkshp. on Applications of Signal Process. to Audio and Acoustics*, 2005, pp. 323–326, DOI: 10.1109/ASPAA.2005.1540234.

[80] D. Bates, M. Mächler, B. Bolker, and S. Walker, "Fitting linear mixed-effects models using lme4," *Journal of Statistical Software*, vol. 67, no. 1, pp. 1–48, 2015, DOI: 10.18637/jss.v067.i01.

[81] B. W. Goldman and W. F. Punch, "Length bias and search limitations in Cartesian genetic programming," in *Proc. 15th Annual Conf. Genetic and Evol. Comput.* New York: ACM, 2013, pp. 933–940, DOI: 10.1145/2463372.2463482.

[82] J. O. Smith, *Introduction to Digital Filters with Audio Applications*. `http://www.w3k.org/books/`: W3K Publishing, 2007.

[83] E. Ly and J. Villegas, "Digital filter design via recurrent Cartesian genetic programming," in *Proc. 13th Int. Wkshp. Comput. Int. and Appl. (IWCIA2023)*. IEEE, Nov. 2023, pp. 7–12, DOI: 10.1109/IWCIA59471.2023.10335891.

[84] X. Wang, Y. He, and T. Li, "Neural network algorithm for designing FIR filters utilizing frequency-response masking technique," *J. Computer Science and Technology*, vol. 24, no. 3, pp. 463–471, 2009, DOI: 10.1007/s11390-009-9237-0.

[85] N. Allakhverdiyeva, "Application of Neural Network for Digital Recursive Filter Design," in *2016 IEEE 10th Int. Conf. on Application of Information and Communication Technologies (AICT)*, 2016, pp. 1–4, DOI: 10.1109/ICAICT.2016.7991720.

[86] D. Etter, M. Hicks, and K. Cho, "Recursive adaptive filter design using an adaptive genetic algorithm," in *IEEE Int. Conf. Acoust., Speech, and Signal Process. (ICASSP '82)*, vol. 7, 1982, pp. 635–638.

[87] J. Koza, F. Bennett, D. Andre, M. Keane, and F. Dunlap, "Automated synthesis of analog electrical circuits by means of genetic programming," *IEEE Trans. Evol. Comput.*, vol. 1, no. 2, pp. 109–128, 1997.

[88] Z. Vasicek and L. Sekanina, "Evolutionary approach to approximate digital circuits design," *IEEE Trans. Evol. Comput.*, vol. 19, no. 3, pp. 432–444, 2015.

[89] M. Santolucito, K. Rogers, A. Lombardo, and R. Piskac, "Programming-by-example for audio: Synthesizing digital signal processing programs," in *Proc. 6th ACM SIGPLAN Int. Wkshp. on Functional Art, Music, Modeling, and Design (FARM 2018)*. New York: ACM, 2018, pp. 18–25, DOI: 10.1038/323533a0.

[90] B. Gardner and K. Martin, “HRTF measurements of a KEMAR dummy-head microphone,” MIT, Cambridge, MA, Perceptual Computing Technical Report 280, May 1994. [Online]. Available: https://sound.media.mit.edu/resources/KEMAR.html

[91] J. Miller and S. Smith, “Redundancy and computational efficiency in Cartesian genetic programming,” *IEEE Trans. Evol. Comput.*, vol. 10, no. 2, pp. 167–174, 2006, DOI: 10.1109/TEVC.2006.871253.

[92] H. Wallach, E. B. Newman, and M. R. Rosenzweig, “The precedence effect in sound localization,” *American J. Psychology*, vol. 62, no. 3, pp. 315–336, 1949, DOI: 10.2307/1418275.