UC San Diego
UC San Diego Electronic Theses and Dissertations

Title
Evolutionary Sound: a Non-Symbolic Approach to Creating Sonic Art With Genetic Algorithms

Permalink
https://escholarship.org/uc/item/0mq5z85r

Author
Magnus, Cristyn

Publication Date
2010

Supplemental Material
https://escholarship.org/uc/item/0mq5z85r#supplemental

Peer reviewed|Thesis/dissertation

eScholarship.org
Powered by the California Digital Library
University of California

UNIVERSITY OF CALIFORNIA, SAN DIEGO

Evolutionary Sound: a Non-Symbolic Approach to Creating Sonic Art With Genetic Algorithms

A dissertation submitted in partial satisfaction of the requirements for the degree Doctor of Philosophy

in

Music

by

Cristyn Magnus

Committee in charge:

Professor Miller Puckette, chair
Professor Charles Curtis
Professor Shlomo Dubnov
Professor Adriene Jenik
Professor Shahrokh Yadegari

2010

Copyright

Cristyn Magnus, 2010

All rights reserved.

The dissertation of Cristyn Magnus is approved, and it is acceptable in quality and form for publication on microfilm:

____

____

____

____

____

Chair

University of California, San Diego

2010

To P.D., for being a supportive,
patient, loving partner, who is always ready to crack geeky jokes, catch obscure references, hide stuffed monkeys around the house, and make me coffee.

Table of Contents

Signature Page iii

Table of Contents vi

List of Figures ix

Preface xi

Acknowledgements xiii

Vita, Publications, and Fields of Study xv

Abstract xvi

I Overview of Genetic Algorithms 1
1.1 Typical Applications 1
1.2 A Simple Genetic Algorithm 2
1.2.1 Representation 3
1.2.2 Fitness 4
1.2.3 Reproduction 5
1.3 Modification to Algorithm and User Bias 5
1.3.1 Fitness Function Bias 7
1.3.2 Representation Bias 8
1.3.3 Mutation Bias 11

II A Taxonomy of Genetic Algorithm Applications in Music 13
2.0.4 Introduction 13
2.0.5 Variations on Genetic Algorithms 14
2.1 Modeling and Simplified Applications 14
2.1.1 Rhythm 15
2.1.2 Harmonization 15
2.1.3 Melody 17
2.1.4 Other 19
2.2 General Tools for Artists 21
2.2.1 Timbre Exploration 21
2.2.2 Development of Melodic Material 22
2.2.3 Interactivity 24
2.2.4 Compositional Environments 25
2.3 Applications for Particular Works 27

III Framework . . . 30
3.1 Virtual Biology . . . 30
3.1.1 Genetic Representation . . . 30
3.1.2 Time . . . 38
3.1.3 Fitness . . . 42
3.1.4 Reproduction . . . 45
3.2 Virtual Ecology . . . 57
3.2.1 Population . . . 57
3.2.2 Worlds . . . 59

IV Sample Implementations . . . 63
4.1 A Simple Algorithm for Evolving Fixed Length, Time Domain Waveforms in a Single, Unchanging Environment . . . 64
4.1.1 The main program . . . 64
4.1.2 Initialization . . . 65
4.1.3 Calculate Population's Fitness . . . 66
4.1.4 Write Output . . . 67
4.1.5 Produce Next Generation . . . 67
4.1.6 Splice Parents . . . 69
4.1.7 Mutate Offspring . . . 69
4.1.8 Overwriting Duplication Mutation . . . 70
4.1.9 Swap Mutation . . . 70
4.1.10 Reversal Mutation . . . 71
4.1.11 Amplify Mutation . . . 71
4.1.12 Exponentiate Mutation . . . 71
4.2 An Algorithm for Evolving Variable Length, Time Domain Waveforms in a World With Multiple, Changing, Island Environments . . . 71
4.2.1 The Main Program . . . 72
4.2.2 Initialize . . . 72
4.2.3 Find Next Individual to End . . . 75
4.2.4 Reproduction . . . 75
4.2.5 Drop Mutation . . . 78
4.2.6 Lengthening Duplication Mutation . . . 78
4.2.7 Change Length Mutation . . . 79
4.2.8 Changing Island World . . . 79
4.2.9 Output . . . 82
4.3 An Algorithm for Evolving Variable Length, Time Domain Waveforms in a Real Time Ring World . . . 83
4.3.1 The Main Program . . . 83
4.3.2 Reproduction Timer . . . 84
4.3.3 Input and Output Weights . . . 85
4.3.4 Individuals . . . 85
4.3.5 Output . . . 88

V Results and Conclusions 90
5.1 Systematic Tests 90
5.1.1 Test Sets 90
5.1.2 Results 92
5.1.3 Seed 92
5.1.4 Crossover Point Without Mutation 94
5.1.5 Mutation Functions 99
5.1.6 Mutation by Amplification 100
5.1.7 Mutation by Exponentiation 103
5.1.8 Mutation by Reversal 105
5.1.9 Mutation by Swapping 107
5.1.10 Mutation by Duplication 110
5.2 Creative Works 114
5.2.1 Run5 & Run12 114
5.2.2 Run16 117
5.2.3 *Snapshots* 118
5.2.4 Real-Time Installation 120
5.3 Future Directions 124

Appendix 127

Bibliography 137

List of Figures

1.1 Simple chromosome . . . 3
1.2 Simple fitness calculation . . . 4
1.3 Sexual reproduction . . . 6
1.4 Mutation . . . 6
1.5 Representation of rhythm by time-slices . . . 10
1.6 Representation of rhythm by duration . . . 11

3.1 Time-Domain Chromosomes . . . 32
3.2 Time-Domain Chromosome spliced at a zero. crossing . . . 33
3.3 Time-Domain Chromosome spliced at a sample. . . . 33
3.4 Spectral-Domain Chromosomes . . . 36
3.5 Genes from Processed Sounds . . . 38
3.6 Other Information in Chromosomes . . . 39
3.7 Lifespan of fixed-length chromosomes . . . 40
3.8 Lifespan of arbitrary-length chromosomes . . . 41
3.9 Lifespan of real-time Critters . . . 41
3.10 Implicit fitness from length . . . 45
3.11 Implicit fitness from mating range. . . . 46
3.12 Asexual Reproduction . . . 48
3.13 Sexual reproduction with a fixed crossover point . . . 49
3.14 Sexual reproduction with a random crossover point . . . 50
3.15 Mutation by duplication . . . 52
3.16 Mutation by dropping . . . 52
3.17 Mutation by swapping . . . 53
3.18 Mutation by reversal . . . 54
3.19 Mutation by amplification . . . 55
3.20 Mutation by exponentiation . . . 56
3.21 Mutation by changing length . . . 56
3.22 Ring worlds mapped onto multiple speaker configurations . . . 61
3.23 Worlds with different geometries. . . . 62

4.1 Flow chart: for evolving fixed-length, time-domain waveforms . . . 64
4.2 Flow chart: initialize . . . 65
4.3 Flow chart: reproduction . . . 68
4.4 Flow chart: evolving variable-length waveforms in changing world . 73
4.5 Flow chart: reproduction . . . 76
4.6 Flow chart: change length mutation . . . 80
4.7 Flow chart: evolving variable-length, time-domain waveform in a real-time ring-world . . . 83
4.8 Flow chart: reproduction timer . . . 84
4.9 Flow chart: Individual . . . 86
4.10 Flow chart: initialize individual . . . 87

4.11 Flow chart: output 89

Preface

Genetic algorithms were developed by John Holland in the early seventies [Hol92]. They use an evolutionary metaphor to evolve solutions to problems or model the evolutionary processes. A population of potential solutions work in parallel to find optimal solutions in a large, complicated search-space. Genetic algorithms are typically used as function optimizers for various sorts of engineering tasks and to investigate a wide range of adaptive processes such as learning, economics, biology, etc.

Because genetic algorithms can effectively explore high-dimensional spaces, I believe they have potential application to algorithmic composition. As my literature review will show, previous work in this field have limited genetic algorithms to symbolic music: usually music that can be described using Western musical notation. The overwhelming majority of work in attempts to evolve entire pieces of music rather than incorporating evolution into the form of the music itself. This project investigates the algorithmic composition of non-symbolic music: the algorithm works on the direct digital representation of the sounds rather than on high level symbols. Both the compositional form of this music and the local details of particular sounds will be dictated by the evolutionary process. The compositional form is defined by an algorithmically generated world comprised of multiple, changing environments. Particular sounds will exist in this world. The local details are defined by the process of the sounds' migration between environments, their interaction with these environments, their reproduction, and the population's evolution over time.

This thesis is organized so that it can be read by someone with no background in biological modeling. The first chapter provides an explanation of genetic algorithms, develops a simple algorithm that evolves new tuning systems, and describes the impact of user bias on algorithm design. The second chapter reviews research applying genetic algorithms to creative musical applications. The third chapter describes a framework for evolving waveforms and a stochastic compositional framework that guides this evolution. Evolution in the time domain is examined in detail; an evolutionary framework that operates in the spectral domain is proposed. The fourth chapter presents three different implementations of the framework in flowcharts. The Fifth chapter characterizes the output of several different algorithms that employ the framework: first a systematic comparison of different sound populations and evolutionary parameters then a description of several works created using the framework.

Acknowledgements

I need to acknowledge too many people. It is inevitable that someone important will be left out. It's nothing personal. I'm going to compartmentalize in terms of family, friends, colleagues, and so on in hopes that I minimize the number of people I forget to acknowledge. By friends, I mean friends who don't fall into another category; I am grateful to everyone in all categories for their friendship.

Let's start with family. They've all provided tremendous moral support. P.D. Magnus has been amazingly supportive throughout this entire process. He has talked shop, read drafts, and brought me coffee in bed. How cool is that? Ceri Van Slyke and Lloyd Brown read drafts of the dissertation and gave feedback. Gwynyth Brown, Nathan Brown, and David Van Slyke all should get gold stars for putting up with the rest of us talking shop during family gatherings; they put on a good act but I know the geeking out wasn't as fun for them as it was for the rest of us.

My friends have also been tremendously supportive. Jen Edwards and Zack Vineyard have spent countless hours in coffee shops with me working on various projects. There's nothing like co-motivation to actually accomplish things. On a tangent, I should also thank the folks from Lestats, Twiggs, and Uncommon Grounds for providing the world with coffee shops where said projects can be worked on in the company of friends with the perfect combination of caffeine and low level distraction. Brenna McLaughlin, sundry UAlbany philosophers, and the Absurdity House and Random Stuff List folks have provided me with emotional

support and important breaks from work to eat lunch, dinner, go for walks, play games, and so on.

My colleagues at RPI who have provided me with an intellectual home away from home. Participating in Pauline Oliveros' Tintinabulate ensemble has kept me grounded. I won't thank members individually because so many people have been involved off and on over the last few years. But breaking from a text-filled computer screen to make actual music kept me reminded of why we do any of this. Without Jonas Braasch's encouragement and advice, I likely wouldn't have completed my degree. I might simply have gone crazy without Doug Van Nort. We've been going through ABD limbo together; a colleague at the same point in his career with similar interests, issues, concerns, etc. has made the entire process substantially less alienating.

Having completed my dissertation away from UCSD, I'll likely forget to thank many important people. Obviously my advisor and committee deserves thanks, especially my outside committee members whose degree of involvement was supererogatory. The community provided by my fellow graduate students has been invaluable.

Vita

Winter 2010	UCSD. Ph.D. in Music
Spring 2003	UCSD. M.A. in Music
Spring 2000	UCSD. B.S. in Cognitive Science

Publications

‘Aesthetics, Score Generation, and Sonificaiton in a Game Piece.’ *Proceedings of the 2006 International Computer Music Conference*, November 2006.

‘Evolutionary Musique Concrète.’ *Applications of Evolutionary Computing: EvoWorkshops 2006, Lecture Notes in Computer Science*, April 2006.

‘Evolving electroacoustic music: the application of genetic algorithms to time-domain waveforms.’ *Proceedings of the 2004 International Computer Music Conference*, November 2004.

Fields of Study

Computer Music

ABSTRACT OF THE DISSERTATION

Evolutionary Sound: a Non-Symbolic Approach to Creating Sonic Art With Genetic Algorithms

by

Cristyn Magnus
Doctor of Philosophy in Music
University of California, San Diego, 2010
Professor Miller Puckette, Chair

The goal of this research is to explore the use of genetic algorithms to evolve waveforms. Genetic algorithms are introduced with a simple application that evolves novel tuning systems. Research involving the application of genetic algorithms to musical situations is reviewed. A framework for applying genetic algorithms is described in terms of virtual biology and virtual ecology. Virtual biology applies the central concepts of genetic algorithms (genetic representation, reproduction, fitness, mutation) to waveforms. Methods for implementing virtual biology are described in detail for time-domain waveforms and are proposed for spectral domain waveforms. Virtual ecologies replace the simple fitness function of conventional genetic algorithms. These ecologies can be designed to produce formal musical structure in an algorithm's output. Several algorithms that employ the framework are described in detail. The output of a simple version of the algorithm is systematically evaluated by running several fixed sets of sounds and environments with active evolutionary parameters. Works produced by more complex algorithms are evaluated subjectively. The success of this project is not in this algorithm's ability to make a population of sounds sound more like their environment, but rather in its ability to create novel sounds that are intimately tied to the process of their creation.

I

Overview of Genetic Algorithms

This chapter provides an introduction to genetic algorithms.[1] Typical applications for genetic algorithms are briefly described, the implementation of a simple genetic algorithm is developed, the biases inherent in applying genetic algorithms to particular applications are explored, and some specific examples of these biases are examined.

1.1 Typical Applications

There are two common approaches to using artificial intelligence for problem solving. One is to use rule-based systems; the other is to use biological modeling. Rule-based systems have the benefit of being explicit and predictable. They can be powerful and efficient in well-defined situations, but not all situations are well-defined. Many problems are too complex to be easily defined; they change in unexpected ways; they have unclear boundaries; *etc.* Biological modeling tends to be more appropriate for such situations.

Biological modeling algorithms, such as genetic algorithms and neural networks, are capable of adjusting themselves to achieve increasingly correct out-

[1] The bulk of this characterization seems to be common knowledge in the field. Much of my personal background was acquired from undergraduate coursework in cognitive science. I'm not sure where to find original written sources for this characterization, although similar characterizations (also sans citations) can be found in the introductions of most of the sources in my bibliography.

puts. This saves users from the tedious task of adding more and more rules to cover special cases and to counteract the unexpected output discrepancies that result from not fully characterizing the problem space. Although some researchers have tried to use neural networks to produce music [TL91], in general neural networks are more appropriate for learning and pattern recognition; genetic algorithms are more appropriate for searching, optimizing, and generative tasks.

Genetic Algorithms were developed by John Holland [Hol92] in the early seventies. They use an evolutionary metaphor to evolve populations of potential solutions in parallel. Genetic algorithms are well suited for solving adaptive problems because they have the capacity to evolve and adjust to changing environments. An obvious application of genetic algorithms is biological simulation. For example, competing evolutionary models can be explored and compared. Optimization problems, particularly those with large-dimensional search-spaces, are another common application, since genetic algorithms are efficient at reducing error. More importantly, a genetic algorithm can be applied to a problem space about which we know little; this allows us to at least solve the problem and potentially discover something about the space by examining the solution.

1.2 A Simple Genetic Algorithm

Genetic algorithms model the evolution of a population in a particular environment. Each member of the population is represented by a *chromosome* that is comprised of a series of *genes*. Each gene has two or more possible values, called *alleles*, and is mapped onto a parameter of the problem space. The environment is represented by a *fitness function* that evaluates each individual and assigns it a *fitness* value. Following the evolutionary metaphor, fitter individuals have a higher probability of passing on their genetic material to future generations. The fitness values of the population members are normalized and converted into probabilities of *reproduction*. Each *offspring* is produced by sexually combining two randomly

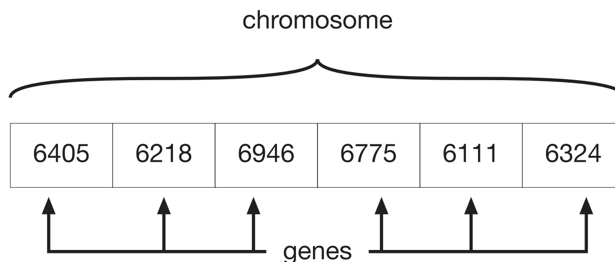

Figure 1.1: A chromosome with six genes.

selected parents. The parent chromosomes are split at some *crossover point* and the first part of one parent is spliced to the second part of the other. Reproduction is accompanied by some probability that *mutation*, usually a flipped bit, will occur in the offspring. Natural selection acts to preserve beneficial mutations; harmful mutations are lost over time. Evolution is allowed to continue for a fixed number of generations or until an individual that reaches some requisite fitness level is produced.

1.2.1 Representation

To clarify this process, suppose that we want to use genetic algorithms to come up with a new, interesting tuning system. To get this system, we will use a genetic algorithm to evolve an ascending scale of eight notes. We can assume that the first and last note of the scale will be fixed an octave apart. This means that each chromosome in population will have 6 genes, one for each intermediate note of the scale. The alleles of these genes will be integers between 6000 and 7200; when divided by 100, these can straightforwardly be mapped to MIDI between C4 (60) and C5 (72) with an offset in cents (figure 1.1). The genes of each chromosome in our initial population will be a random value between 6000 and 7200. This representation is useful because it represents pitch in a perceptually meaningful way but is flexible enough to allow evolution of something besides the logarithmic frequency relationships that we are used to hearing in tuning systems.

Figure 1.2: The negative intervals of this chromosome are summed and added to 3600 for a fitness value of 2578.

1.2.2 Fitness

The fitness function for this application will be quite straightforward. Because we have assigned possible alleles such that the first gene (k_0) cannot help but be equal to or greater than 6000, we do not have to include this first interval in our fitness calculation. For each subsequent gene k_i, subtract the value of the previous gene k_{i-1}. The result will be negative if the two genes are not in an ascending relationship. We will ignore all positive results, since we don't care about the size of the interval between note as long as they are ascending, then sum the values over the length of the chromosome. Given a worst case scenario in which the values oscillate between 6000 and 7200, this will give us a value of -3600. Because it will simplify later calculations if our fitness values are positive, we will add 3600 to the result, bringing our lowest possible fitness value up to zero (figure 1.2). This will give us the fitness F.

$$F_j = \sum_{i=1}^{5} (k_i - k_{i-1})^- + 3600$$

This gives us fitness values between zero and 3600. An individual that perfectly fulfills our requirements will have a value of 3600. In order to calculate each individual's probability of producing offspring, we will convert each fitness value F_j into a probability P_j. We do this by dividing the fitness value of each chromosome by the sum of all fitness values in the population.

$$P_j = \frac{F_j}{\sum_{l=0}^{N} F_l}$$

1.2.3 Reproduction

Once we know the probability of reproducing for each individual, we can produce the next generation. For each individual of the next generation, we randomly select two parents. Since the chromosomes are 6 genes long, we will set the crossover point to be between the 3rd and 4th genes, at the halfway point. The first three genes (0–2) from the first parent will become the first three genes of the offspring; the last three genes (3–5) from the second parent will become the last three genes of the offspring (figure 1.3). The final step of reproduction is mutation. For each gene, there is some probability, usually very low, that mutation will occur. I don't want to replace genes with random numbers, which might lose any information that the genes may already contain, so instead my mutations will add a random number between -50 and 50 to the mutating gene's value. If the result of the mutation would be less than 6000 or greater than 7200, the gene is replaced by a random value (figure 1.4).

1.3 Modification to Algorithm and User Bias

The choice of genetic algorithm used to explore a space reflect the designer's bias. Although the method is flexible enough to be applied to a wide array of problems, the form that this application takes will strongly reflect the designer's intuitions about the domain in question. This is most obvious in the designer's choice of fitness function. If I choose, for example, to use rules from western tonal theory to evaluate members of a population, I am already using a model that cannot be applied to world musics or even to much of the western art music written in the last century. The choice of representation compounds this situation. If I choose to represent music in terms of pitch, I place pitch in a hierarchical relationship to other musical variables that excludes application of the model to musics that stress rhythm, timbre, dynamics, *etc.*, over pitch. This bias is amplified by the designer's attempts to choose mutation methods that are likely to raise fitness,

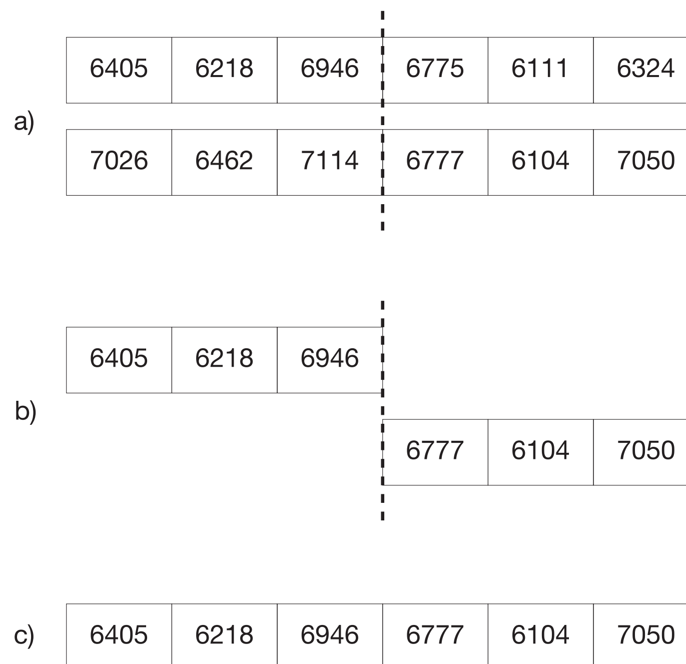

Figure 1.3: a) Two parent chromosomes with dotted line representing crossover point. b) Portions of parents to be transcribed to offspring. c) Offspring.

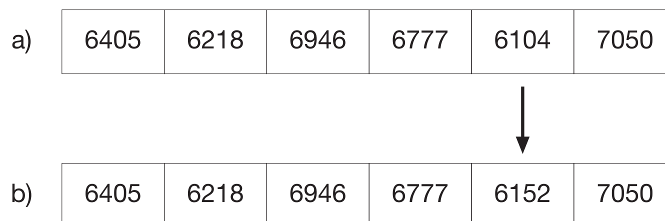

Figure 1.4: a) Offspring before mutation. b) Offspring after mutation. Arrow indicates mutated gene.

rather than introduce new flaws. Any mutation method I introduce in this way will reflect the features in which I am interested.

There are many biases built into the above sample algorithm. An obvious one is the assumption that a scale ought to have seven novel pitches, culminating with the first pitch repeated an octave up. This is an easy assumption for someone trained in western music to make—if I hadn't used seven pitches for simplicity's sake, I most likely would have been inclined towards twelve. This prevents my algorithm from coming up with scales of other lengths that might be interesting to work with. My decision to have a scale that starts and ends on C is a clear outgrowth of years of practicing scales on the piano—always starting on C of course; perhaps I wouldn't have chosen C if I'd played a different instrument. I can write this off as a practical consideration because I can always transpose the scale, but my decision says something about my worldview nonetheless. But why does my scale even have to be within an octave? If my goal is to create a new tuning system with it new perceptual experiences, why am I constraining it to an octave? Perhaps it would be interesting to develop a system that repeated every 2 octaves, or every tritone, or over some arbitrary interval that doesn't necessarily fit evenly into an octave. All of these assumptions about how tuning systems ought to behave are direct results of my training and aesthetic preferences.

1.3.1 Fitness Function Bias

Fitness function bias can be clearly seen by comparing the work of Moroni *et al.* [MMZG00] (see §2.2.4) and the work of McIntyre [McI94] (see §2.1.2). Both use genetic algorithms to create harmony, but with different goals that can be seen in the type of functions they choose.

Moroni *et al.* are concerned with developing a creative system and use their harmonic fitness function as one of its components. Their fitness function is based on physical theories of consonance and dissonance. This method attempts to be neutral in that it uses a model of what everyone can physically hear rather

than a particular style's definition of dissonance and consonance. Furthermore, it does not define functional relationships between consonant and dissonant chords, as these are subject to change with style as well. By attempting to avoid style-specific fitness functions, Moroni *et al.* hope to leave resulting compositions open to influence by the user.

McIntyre is concerned with modeling harmonic production. Because the concern is with modeling, an objective measure of success must be employed. This leads him to use a fitness function based on rules of baroque, four-part harmony. This is an ideal function for his purpose because the rules have already been described by theorists and are generally agreed upon. If McIntyre were to use a fitness function based on physical theories, there would be no clear answer as to whether or not his system were successful. A consonant but uninteresting harmonization might be declared successful for being consonant or it might be declared unsuccessful for being musically uninteresting. By using rules from a well-studied style of music, the resultant harmonizations will either clearly follow the rules or they will not.[2]

1.3.2 Representation Bias

Throughout the literature (§II), there are two ways in which pitch and rhythm are combined. In one, the chromosome's genes represent slices of time, usually sixteenth-notes or eight-notes, and each slice of time contains a pitch or a rest. In the other, each gene is represented by a pitch/duration pair.

In the first case, there is a strong notion of beat. All notes will fall on beats or beat subdivisions. There are two possible ways of interpreting neighboring notes of the same pitch. They can either be considered notes of longer duration, in which case repeated notes cannot be represented, or they can be considered as repeated notes, in which case notes of longer duration cannot be represented.

[2]This is not to say that this method captures the musical style, only that it effectively follows the rules. This is a reasonable way to test the success of the model; we can disagree about its ability to capture style but not about its ability to follow rules.

This method of representing time is effective at constraining notes to fall on beats, which is important for many styles of music. However, it will never permit triplets or other tuplets.[3] Each pitch will be tied to a particular location in time and mutations to particular genes will change pitches vertically but not disrupt the temporal location of later notes (figure 1.5). This means that it is highly unlikely that the melodic contour will remain the same while the rhythm changes.[4] When genes are tied to particular slices of time, a gene is either on the beat or not. If a gene starts out on the beat, it will always be on the beat, and mutation will not change this. This means that events can be consistently tied, for example, to strong beats.

Because the second case represents each note with a pitch/duration pair, the melodic contour and the rhythm can be evolved with more independence. If the durations permitted by the algorithm are restricted to beats or beat subdivisions, this representation will be constrained to the same sorts of divisions as the previous representation. However, this representation could potentially permit rhythms that are unrelated to some beat structure.[5] Another difference is that this representation allows neighboring notes to repeat pitches without losing the possibility for notes of varying duration. Unlike the first case, there is no clear indication of beat. When genes are tied to particular slices of time, a gene is either on the beat or not. If a gene starts out on the beat, it will always be on the beat, and mutation will not change this. This means that events can be consistently tied, for example, to strong beats. In the case where genes represent pitch/duration pairs,

[3]One can imagine extending this method by adding more subdivisions, but it will always be constrained. Since each subdivision adds to the chromosome length, adding subdivisions will explosively lengthen the chromosomes. The computer's available memory, not to mention additional processing time, will provide an upper limit to chromosome length and hence the number of subdivisions. Furthermore, by adding too many subdivisions, you lose the beat-constraining benefits of this type of representation without gaining the flexibility of the second type of representation.

[4]Since mutations, in this case, are applied independently to individual genes, several unlikely events would have to occur. Each gene after the changed rhythm would have to be selected for mutation. Each mutation gene would have to acquire the appropriate value to shift the melody in the appropriate direction rather than replace it with something else. This is something like expecting monkeys with typewriters to write Shakespeare.

[5]Unlike the first representation, this would not change the length of the chromosome. The available durations would be limited by the number of bits the computer allocates to the data type we use to represent duration (e.g. a float).

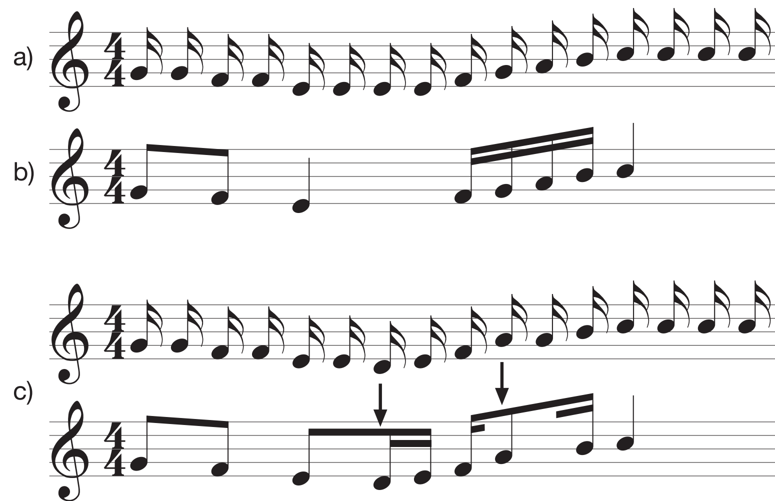

Figure 1.5: a) A 4/4 bar of a chromosome whose genes represent sixteenth-note slices of time. b) This chromosome's output interpreted so that neighboring notes of identical pitch represent longer durations. c) Two mutations to the previous chromosome, resulting in changes to both melody and rhythm. These changes do not affect other notes represented by the chromosome.

Figure 1.6: a) A 4/4 bar of a chromosome whose genes represent pitch/duration pairs. b) The gene after the first note's pitch and rhythm have been mutated. The mutated pitch results in a repeated note, whose repetitions have different durations. The mutated rhythm shifts the rest of the measure by a sixteenth-note, changing the position of each pitch while preserving the melodic contour and changing the length of the entire melody.

this is not the case. If the rhythm of the first note is altered by mutation, the shift in time will propagate down the gene, potentially causing drastic changes to the overall metrical structure (figure 1.6).

1.3.3 Mutation Bias

The introduction of bias through mutation results from the designer's attempts to introduce domain-specific mutations. Two contrasting examples are seen in the work of Wiggins *et al.* [WPPAT98] and McIntyre [McI94] (see §2.1.2). While both are attempting to use genetic algorithms to write baroque four-part harmony to an existing melody, Wiggins *et al.* attempt to build a great deal of knowledge into the mutation operations, whereas McIntyre allows much more freedom.

Wiggins *et al.* provide a list of mutation functions that severely curtail the potential values for each gene. These are *perturb*, which allows the voice to move up or down a semitone, *swap*, which swaps the values of two voices, *rechord*, which replaces a chord with a new one in which the melodic note is either root, 3rd, or 5th, *phrase-start*, which replaces each phrase beginning with a tonic in root position on the downbeat, and *phrase-end*, which replaces the end of each

phrase with a chord in root position. The bias is clear—the problem space has been arranged so that only high-level mutations relating specifically to chords can be used. Only the rechord and perturb mutations will allow the algorithm to explore new harmonies. The rechord mutation assumes that dominant sevenths will never occur, so these can only be reached by the perturb mutation. This is highly unlikely because it can only move by semitones and will have to traverse a great distance through low-fitness space in order to survive long enough to reach a higher fitness level. The phrase-start and phrase-end functions effectively force a specific structure on the members of the populations rather than allowing them to evolve to some point that might differ from this structure but still follow rules of baroque harmony.

McIntyre has one, simple mutation: the mutated pitch can be replaced by any other pitch. This assumes that with enough freedom and a well-designed fitness function, the rules of baroque harmony will be successfully implemented by the algorithm. Unlike the perturb mutation function used by Wiggins et al., this function does not require unfit individuals to be preserved for multiple generations in order to reach a fitter value because notes can jump from one position to another rather than having to move by semitone. Although many mutations will be negative, these will be unlikely to be passed on to future generations. Beneficial mutations will happen with enough frequency and be passed on to future generations that the population as a whole will progress towards a higher fitness.

In this section I have explained how implementing genetic algorithms inherently involves user bias. This will be relevant both in the literature review (chapter II) and in descriptions of my own work (chapters III–V).

II

A Taxonomy of Genetic Algorithm Applications in Music

2.0.4 Introduction

There are many researchers using genetic algorithms in computer music, both as tools for engineering and as methods for achieving creative goals. Engineering applications include designing synthesis systems [Gar00], setting synthesis parameters to match particular sounds [CYH96], [FV94], [Hor95b], [Hor95a], [Hor96], [HB96], [HBH92], [HBH93], [HC95], [VV93], [RV02], [Hor03], tuning [HA96], synchronization of sound and animation [THG+93], recognition [FF99], [Fuj96], polyphonic pitch detection [Gar01], and evolution of components in systems with more traditional architecture [Bey99], [Jac95]. We will not address these in more depth here, as our primary interest is in creative applications.

Because of the diverse approaches taken towards using genetic algorithms for musical applications, there are several useful ways in which these can be organized. Burton and Vladimirova [BV99] review this literature in terms of the type of fitness function used. Here we address the work in terms of the researchers' intents.

There are several approaches being taken towards achieving creative ends using genetic algorithms. One is to assume that we must first model creativity in

known musical situations before we can approach the much larger task of creating a more comprehensive creative system. Another is to develop tools that perform particular tasks that the designer believes will be useful to currently practicing musicians. A third is to design tools or processes that can be applied to specific creative ends; if they happen to be useful to other practitioners, that's an added bonus, but it is not the primary goal.

2.0.5 Variations on Genetic Algorithms

There are two variations on conventional genetic algorithms that are seen throughout the literature: the *genetic program* and the *interactive genetic algorithm*. Genetic programs [Koz92] differ from genetic algorithms in that they evolve sequences of functions that act to produce and modify data rather than directly evolving sequences of data. Interactive genetic algorithms replace the fitness function with a user who evaluates each member of the population and subjectively assigns a fitness value. Because of this, a recurring problem for designers of interactive genetic algorithms is the *fitness bottleneck*. This is essentially the limit of the human capacity to rate each member of the population, resulting from the requirement that each member of the population be listened to in real time—a time-consuming and often tedious process.

2.1 Modeling and Simplified Applications

Many researchers assume that we must tackle simple problems to decide what will and won't work before investing the time and effort in creating more comprehensive systems. More importantly, solving simple problems helps us to refine our understanding of the processes involved; if a comprehensive system is built from scratch, it is more difficult to discover why the system works the way it does than if it is built from simple parts that have all been individually examined.

2.1.1 Rhythm

Both Horowitz [Hor94] and Burton and Vladimirova [BV97] explore the generation of rhythm. Their initial approaches are very similar; both represent rhythm in terms of sequences of ones and zeros that occur on discrete time steps. Individuals are collections of voices with different timbres that coöccur to build more complex rhythmic textures. What distinguishes their approaches is their fitness functions. Horowitz uses an interactive genetic algorithm. Because interactive genetic algorithms require that users listen to and rate each individual in the population, this can be taxing for the user. To ease the user's work-load, higher-level algorithms, such as syncopation, density, downbeat, *etc.*, are used to shrink the search space. Burton and Vladimirova use an ART (Adaptive Resonance Theory) neural network, rather than a user, to evaluate fitness. ART networks are effective pattern recognizers—they categorize patterns and identify new pattern categories without supervision. Fitness is assigned based on similarity to patterns, with new patterns being designated for individuals that are sufficiently different from existing patterns.

2.1.2 Harmonization

Investigations that model harmonization focus primarily on baroque, four part harmony. Perhaps this is due to extensive effort that has been made by music theorists to document rules for this style: the bulk of the effort in writing a suitable fitness function has already been done. The standard representation is a fixed length chromosome with each gene a set of four values, one for each voice. The melody voice is pre-defined and is not subject to mutation.

A very simple sub-problem is that of finding the correct voicings of a chord progression. Horner and Ayers [HA95] take this approach. They build knowledge into their system by enumerating every possible voicing of each chord in the progression. The mutation function simply selects another voicing possibility. This enumeration might seem excessive, but the approach clearly emulates the method

taught to music theory students;[1] it has a certain ecological validity. Because of its simplicity, the system will almost always find an answer that satisfies the fitness constraints.

McIntyre [McI94] addresses the harmonization problem directly. Rather than encode extensive knowledge of theory into the mutation function, mutated notes are replaced at random by any other note. This approach allows the population to converge relatively quickly. Although the output is rarely completely true to the rules of baroque, four-part harmony, McIntyre seems satisfied that the results are believable harmonies, consistent with the baroque style. One strength is the ability of the algorithm to find multiple believable harmonies for a given melody—this is note-worthy since rule-based systems tend to find the first result that satisfies the rules and then stop before finding other potential results.

Wiggins *et al.* [WPPAT98] have attempted to build upon McIntyre's work. The authors clearly find the results dissatisfying and they conclude that only a conventional rule-based system is capable of proper harmonization. It is worth investigating this dissatisfaction in more detail, as the way in which a system fails can tell us much about the system.

Wiggins *et al.* go to great lengths to encode as much knowledge as possible into the mutation functions. Rather than use a simplified version of the codified rules of baroque, four-voice harmony, as McIntyre does, they extend the fitness function a great deal. The excessive complexity of their domain-specific mutations hamstrings the genetic algorithm by preventing it from exercising one of its primary strengths: the ability to find solutions when the path is unclear. By artificially limiting the spaces that can be reached by mutation to a list of highly-specific possibilities, Wiggins *et al.* are presuming that they have already completely codified in their mutation functions every possible operation that might be necessary to achieve correct, baroque, four-part harmony. Because of the precision of these rules, if any necessary rule was left out, the search space would be

[1] This might not be the method taught to *all* music theory students, but it was at least the way I was taught.

so constrained that the population could not reach portions of the space[2] that it may need to pass through in order to reach an acceptable fitness level.

Wiggins *et al.* make it clear that the only acceptable fitness level is perfection and, based on their algorithm's failure to achieve perfection, they conclude that genetic algorithms are not appropriate for the "simulation of human musical thought." This conclusion is unjustified. First, when their results were graded by a music theory professor, who was asked to grade them as he would his theory students, he gave them "a clear pass." Wiggins *et al.* do not make it clear why they are dissatisfied with results that seem acceptable to a music theorist. Second, they assume that modeling four-part harmonization in the baroque style, based on an intricate rule-system defined years after-the-fact by theorists, will somehow have clear implications for creativity. Despite the fact that baroque composers did not have the benefit of a few hundred years of theory to carve rules in stone for them, and therefore often failed to comply perfectly with these post-hoc rules, Wiggins *et al.* declare that any approach that results in less than perfect compliance with this artificial rule-system is an inappropriate model of creativity. For Wiggins *et al.*, any result that breaks even the most minute of rules is unsatisfactory. In taking this position, Wiggins *et al.* set themselves against contemporary popular wisdom, which holds that the ability to effectively break rules, rather than the ability to perfectly adhere to rules, to be the hallmark of creativity.

2.1.3 Melody

There is surprising similarity between the approaches taken to modeling melodic generation by Laine and Kuuskankare [LK94] and Johanson and Poli [JP98]; this is all the more striking given their apparent unawareness of one another's work. Rather than use a straightforward genetic algorithm, both use genetic programming (see §2.0.5). This allows melodies to vary in length. Because of the desire to simplify the problem to one dimension, melodic sequences are rep-

[2]Space here is the space of which the search-space is a subset.

resented as a sequence of pitches with no rhythmic representation; all notes are assumed to be the same length, although Laine and Kuuskankare grant that longer rhythms might be represented by sequences of identical pitches.

Johanson and Poli use a hybrid approach to fitness. The user can rate fitness, as with an interactive genetic algorithm (see §2.0.5), or use a neural network to evaluate fitness. Unlike the ART network used by Burton and Vladimirova (see §2.1.1), which relies on the properties of the network itself to assign a fitness, Johanson and Poli train back-propagation networks with the user's past ratings in order to create networks that will rate in a similar manner to the user. These networks use shared weights for scalability so that they can evaluate melodies that vary in length.

Brown [Bro04] uses the task of melodic generation to compare expert systems with genetic algorithms. He compares randomly generated melodies and melodies generated by rule-based algorithms. Then he uses populations of randomly generated melodies and melodies generated by rule-based algorithms to seed genetic algorithms that run for twenty generations. Each gene represents a note. The genetic algorithms either apply no mutation, random mutations, musically meaningful mutations, and combinations of random and musically meaningful mutations. The fitness function incorporates statistics from a library of analyzed melodies. Rule-based procedures, unsurprisingly, produced melodies that were consistent with the style in the fitness library, but Brown calls these melodies "conservative." Musical mutations make these melodies more interesting and surprising. Other mutations reduce coherence. Twenty generations turned out to be far too few to allow the populations seeded with random melodies to reach the level of fitness the rule-based metlodies started with, though in that amount of time musical mutations began to produce perceptible structural elements. Brown acknowledges the problem, but unfortunately has failed to re-run his experiments for long enough to produce results that could be meaningfully compared. A more appropriate version of the study would let each case run until it minimized its er-

ror. He could then compare both efficacy and the amount of processing necessary to achieve it.

2.1.4 Other

The papers discussed below go a step further than the papers previously discussed in this section. Although they are still modeling problems, they investigate multiple musical dimensions or alternative uses for genetic algorithms in the context of modeling musical creation.

In addition to their efforts in harmonization (see §2.1.2), Wiggins *et al.* use genetic algorithms to create simple jazz solos with harmonic accompaniment. The solo melodies are represented as a list made up of paired scale-degrees and durations, with non-scale notes forbidden. The harmonic accompaniments are represented by lists of triplets: root, chord type, duration. Unfortunately, as with their attempts at harmonization, the combination of extremely specific mutations and fitness functions comprised of large lists of rules led them to disappointing results.

Polito *et al.* [PDBB97] use genetic programming to create sixteenth-century counterpoint from a user-defined *cantus firmus*. Rather than evolving a single set of instructions, three sub-populations of instructions are evolved: one for polyphony, one for imitation, and one to select portions of the cantus firmus from which to generate new material.

Rather than evolve a lengthy excerpt along a single musical dimension, Gibson and Byrne [GB91] focus on evolving four bars of four-part harmony. Unlike the previous examples that evolve only harmony (see §2.1.2), they evolve rhythm, melody, and harmony. To simplify the problem, the space is restricted to notes in the C major scale and 3 chords: tonic, dominant, and sub-dominant. The problem is broken down and addressed in series. First, a rhythm is evolved, then a melody, then harmony. Instead of a rule-based fitness function, each sub-section is graded by a neural-network that has been trained with four-bar segments in the desired style. To provide for global structure, the melodic fitness assignment is made by

two networks, one that evaluates melodic intervals and the other that evaluates melodic structure.

Hörnel and Ragg [HR96] use genetic algorithms more indirectly. Their research focuses on building neural networks capable of producing and harmonizing simple folk melodies. A classic problem with neural networks is arriving at an ideal network size: a network that is too large will learn quickly but be incapable of generalizing; a network that is too small will generalize to some extent but be unable to solve the problem. A genetic algorithm is used to add and remove weights and units to neural networks to evolve an appropriately sized network that is capable of both learning at an acceptable rate and generalizing.

Todd and Werner [TW99] [MKT03] use the context of melodic production to explore the coëvolution of music producers and music critics. Producers represent melodies with a fixed-length chromosome of pitches; critics represent expectancy with a transition matrix. Coëvolution reduces the ability of creators to find easy ways to trick a stationary fitness function into giving a high score and enhances diversity within the population and over the course of time.

Kirby and Miranda [MKT03] use a modified genetic algorithm to model the cultural transmission of emotional meaning in music and the emergence of musical grammar. They use the metaphor of parents teaching children rather than a reproductive metaphor. Children initially know nothing; adults teach by example. The first generation of adults produce purely random combinations of riffs and emotions. Children hear many melodies produced by their parent and generalize their own musical grammars linked to emotional meanings. After training is completed, the adult is deleted. This is quite similar to asexual reproduction in GA, with the random elements from the adult's melodies and the child's generalization from the melodies taking the place of mutation. Eventually, offspring develop musical grammar and are able to produce increasingly complex music that is characteristic of the music produced by the population.

Federman [Fed03] develops a system for predicting the next pitch in a

melody and studies the rôle of representation in its efficacy. She compares four different pitch representations that encode pitches using binary versus grey coding and pitch versus a psychologically based representation of pitch. She concludes that the type of encoding does not matter, but the representation that incorporates relational features of previous notes performs much better than pitch alone.

Pazos *et al* [PSA+03] model collective music making from an anthropological perspective using an interactive genetic algorithm. Groups of artificial musicians collectively produce rhythmic themes. Each group of artificial musicians is represented by a grid; each line is an individual's chromosome. Chromosomes are strings of evenly-spaced rhythmic units that are either on or off. When rhythms are played, each individual in the group has it's own density probability; whether or not an on-note is played or not is based on this probability. Groups selected as parents mate on an individual basis—that is, each individual in the group mates with the corresponding individual in the other group, but with an independently generated random crossover point.

2.2 General Tools for Artists

2.2.1 Timbre Exploration

There are three basic approaches taken to using genetic algorithms to explore timbre. One is to modify some initial sound; another is to use genetic algorithms to set parameters for some synthesis technique; the third is to evolve populations of synthesizers that cooperate to create sound events.

The first approach is taken by Horner, Beauchamp, and Packard [HBP93], who begin with an initial sound made with additive synthesis and derive a population with a series of filtering and time-warping operations. These same operations are used as mutation functions. Because this application is designed to help the user find novel sounds, rather than known sounds, an interactive genetic algorithm is used to direct the course of evolution.

The second approach is taken by Johnson [Joh99] [Joh03] and Dahlstedt [Dah01] [BD03]. Johnson uses interactive genetic algorithms to set parameters for Csound algorithms [Bou00]. To allow the evolution to proceed more quickly, users are permitted to change parameters to help direct the course of evolution. Dahlstedt takes this process a step further by designing an interface that deals abstractly with synthesis parameters from any synthesis system. Two steps are taken to help counteract the fitness bottleneck (see §2.0.5). First, parameters that the user is satisfied with can be saved while allowing the rest of the genome to continue to evolve—this shortens the process by preventing successful settings from being evolved away. Second, an abstract, visual representation of each individual in the population is displayed, allowing the user to use visual information to speed the evaluation process.

Bowcott [Bow90] uses biological modeling to generate musical events by granular synthesis. Although this is not explicitly a genetic algorithm, it is sufficiently similar to be addressed in this section. It can be interpreted as a combination of population modeling, in which the population potentially supports some maximum number of individuals but the actual number of individuals fluctuates depending on the distribution of resources, and a modified genetic algorithm, in which reproduction is asexual and mutation is always in a favorable direction. Each grain has a chromosome comprised of its synthesis type, which cannot be modified by mutation, and a list of parameters used to generate an instance of the type. The environment defines factors relating to the survival of various types and how individuals will respond to other grains. The events generated by this process change in time as the evolutionary process unfolds.

2.2.2 Development of Melodic Material

Development of melodic material is a difficult prospect and its success depends a great deal on the algorithm-designer's success in defining a problem-space in which genetic algorithms can be fruitfully applied. Ralley [Ral95] attempts to

design an interactive genetic algorithm that will develop any melodic idea. It takes a user-supplied melody and uses it to seed a population that is generated based on its statistical properties. The hope is that this will allow the algorithm to be relatively style-neutral. Unfortunately, this approach results in a population with very little biodiversity; as a result, the user has difficulty making the subjective fitness evaluation.

By constraining their domain, Horner and Goldberg [HG91] have more success with their thematic bridging algorithm. It is designed specifically for minimalist-style phase music. Rather than use a subjective fitness function, the fitness function is based on the initial and target melodies and the amount of time that the user would like to have pass between them.

Biles [Bil94] also focuses on a particular style, allowing the use of mutation functions that are stylistically relevant. An interactive genetic algorithm, called GenJam, is used to generate jazz solos based on a tune. Two populations are used to construct the melody: a population of measures, each made up of eight eighth notes in 4/4, and a population of phrases that determine how the measures will be arranged. Biles describes the results as "competent 'with some nice moments' " but is clearly dissatisfied with the time-consuming nature of the training process. Biles and Eign [BE95] attempted, with equivocal results, to address the fitness bottleneck by designing a system that allows multiple users to work in parallel to rate the fitness of members of the population.

Instrument Design

Mandelis and Husbands [MH03] use interactive genetic algorithms to design virtual musical instruments—that is, control mappings for hardware interfaces such as data-gloves. The population is seeded with hand designed mappings. Crossover operations mix parameter values and mutation operators randomly change one or more parameter values. Additional individuals can be added to the population in between generational cycles.

2.2.3 Interactivity

Given current processor speeds, the use of conventional genetic algorithms for interactive applications is still impractical if the desired output is the result of some number of generations rather than the process itself. Biles [Bil98] attempts to do something of the sort by extending his previous work (see §2.2.2) to work in a real time situation in which the program trades four bars with a live musician. Although the program clearly grew out of work with genetic algorithms, in this instantiation there is a population of one that asexually reproduces, with no fitness measure and liberal mutation, to produce one offspring in real-time. It is essentially no longer a genetic algorithm; it is a process for directly deriving new material from from a seed by a series of mutations.

Further work by Biles [Bil03] results in a successful interactive performance system. Using a process similar to genetic programming, soloist agents are evolved either ahead of time or during a concert with audience feedback. These agents randomly select phrases from their phrase population to perform pieces. The phrase population is made of operators that are applied to the measure population. Mutation operators with musical meanings drastically accelerate the training process.

Spector and Alpern [SA94] approach the same problem using genetic programming. Because the evolved program can run in real time, the time-consuming evolutionary process does not have to occur between the program's input and output, as it would with a conventional genetic algorithm.

Nemirovsky and Watson [NW03] propose an three-layered audio-visual improvisation system. The bottom layer (input) maps and transforms input data from the outside world. The middle layer (structural) is a recurrent neural network that processes the input and sends output to the top (perceptual) layer. The perceptual layer consists of media operators that take input from each other as well as the structural layer. Each of the three layers can be independently evolved or the entire system can be evolved as a single chromosome. The user can choose

to use IGAs to evolve the system offline or during performance or choose to use a fitness function that can be described in terms of similarity to or difference from another designated network.

2.2.4 Compositional Environments

Designing compositional environments is an inherently more complex task than those previously described, as it involves some combination of the above elements. Some unified system for codifying the various musical elements must be developed, along with appropriate forms of mutation. Furthermore, it should be designed with the awareness that the environment will be appropriate only for a certain subset of musical styles.

A simple environment, designed by Degazio [Deg97], uses genetic programming to evolve MIDIFORTH processes that produce compositions. The fitness function combines objective fitness functions, such as following species counterpoint rules, harmonic correctness, or meeting particular statistical criteria, with the subjective user choice of the interactive genetic algorithm. For each generation, the first parent is chosen from a set of 3 individuals designated by the user; the second parent is chosen from the entire population based on the objective fitness function. This substantially decreases the fitness bottleneck while allowing the user to direct the course of evolution. Although this seems stylistically neutral in that the user can use statistical criteria rather than the style-specific fitness options, it is still restricted to styles of music that can be produced with MIDI and can be effectively defined by the statistics used by the system.

Thywissen [Thy96], [Thy97], [Thy99] tries to "define a comprehensive framework for musical evolution." This is clearly a futile process, as numerous decisions contrary to this goal must be made in the design process (see §1.3). The system uses genetic algorithms to evolve components of a composition such as melody, rhythmic structure, harmony, instrumentation, and form. Genes are mapped onto a grammar that has been supplied by the user either as a series of

rules or through importing an existing composition as a MIDI file. A hierarchical model separately evolves phrases and systems. A conventional genetic algorithm evolves phrases—sequences of notes and chords with their associated rhythms and dynamics. Systems are series of phrases evolved through genetic programming; the possible operators are transpose, retrograde, invert, augment/dimmute, and grow. Although this seems to be fairly neutral, in that a user has stylistic control over the resulting composition, it is clear that this system applies only to the subset of styles that deal centrally with melody, harmony, rhythm, and fixed forms—it is inappropriate for styles that focus on timbre rather than pitch, have open forms, or incorporate visual elements; it almost certainly can't handle text-setting, and *musique concrete* is out of the question.

The system designed by Manzolli, Moroni, *et al.* [MMZG99], [MMZG00] does not attempt to be stylistically neutral but instead embraces the notion of an algorithmic compositional system whose output, although guidable by the user, has a distinctive characteristic that is directly related to its processes. The system works in real time, sending output via MIDI; its rhythm is determined by the length of the generational cycle, which can be altered by the user in real-time. The population is made up of four-note chords, but unlike the harmonization attempts above (see §2.1.2), it relies on physical theories of consonance for its harmonic fitness function. Two other fitness functions are used in conjunction with the harmonic fitness function: melodic and vocal-range. The melodic fitness function is based on distance of notes from some tonal center or attractor; the vocal-range fitness function verifies that the notes of the chord fall within voice-ranges set by the composer. These two fitness functions can be set by the user in real-time, allowing the output to be sculpted by the user.[3] Although the user has considerable leeway in personalizing the output, the sound of the algorithm will always be heard in the rhythm (which comes directly from the generation length), in the types of harmonies evolved (which come directly from the consonance model

[3]Although its designers identify the system as primarily compositional, which it is, this system might well be considered as a successful implementation of an interactive performance system as well.

used), and in the tendency to converge towards the melodic fitness function's tonal center (when it remains stationary for a sufficient length of time).

Dahlstedt [BD03] describes a system that is more stylistically neutral than Thywissen's, but his output is still in MIDI, which restricts him to genres that can be described by MIDI. Instead of having a chromosome that sequentially translates into musical phrases, he evolves a tree that represents structural relationships. Nodes contain operators that either combine subtrees vertically into chords or horizontally into melodies. Leafs can store pointers to higher nodes in the hierarchy, allowing recursive self-reference. Individual trees reproduce asexually, with mutations affecting amplitude, duration, and transposition. All generations are stored and used to produce a single piece.

2.3 Applications for Particular Works

Unlike previous examples, the research presented here is not intended to result in models or tools that can be used and expanded upon by future research, but rather to create particular musical works that are generated by genetic algorithms that have been specifically designed for a work or series of works. My own investigations fall into this category and are described more thoroughly in chapter III. These algorithms are almost always heavily modified from conventional versions of the algorithm, relying often on complicated environments and interactions between individuals.

Waschka [Was99] describes a process used to produce a series of pieces. The population is seeded by some initial musical material. Two strategies are employed to retain biodiversity. First, fitness is unrelated to an individual's characteristics; it is assigned randomly. Second, individuals can skip generations. Each generation of individuals produced by the algorithm is heard in succession. Although a great deal of the form is produced by the evolutionary process, the resultant works are individual enough to merit the algorithm's use to create a series of

pieces rather than one because of the composer's influence over the initial musical material.

Brooks and Ross [BR96] describe a piece by Brooks in which the algorithm is modified by giving creatures activity-states. Creatures randomly rest, forage, or mate. A resting creature is silent, while a mating or foraging creature produces music. A creature's music is different for each activity-state and is derived from the set of characteristic chords and melodic phrases that are defined by its chromosome. Rather than each individual in the generation reproducing at once, an individual will only mate if its mating cycle coöccurs with the mating cycle of another individual in its generation. This causes the generations to overlap. This results in significant textural differences, since each generation is mapped onto a different instrumental voice.

Dahlstedt and Nordahl [DN01] describe not a musical work but a biological simulation[4] with a distinctive musical byproduct. The genetic algorithm is modified by giving each individual two genes and by replacing the fitness function with a square lattice in which individuals and resources occupy space. The two chromosomes are a sound chromosome, which is a list of the individual's preferred notes, and a procedural chromosome, which is a list of instructions defining the individual's actions in the world. One of these instructions is a sing instruction, which causes the individual to produce a note from its sound chromosome. Sound exists in a square and dies out over time. Creatures with more life points—these are acquired by entering a square containing food—produce louder sounds than creatures with fewer life points. Creatures with enough life points can mate, so long as a nearby creature has recently heard a sound from its sound chromosome. An individual can survive reproduction, but a portion of its life points is removed to create the offspring. As the process unfolds, patterns occur resulting from the layers of many individual movement and sound production patterns. These change gradually as the makeup of the population changes.

[4] I might call it an installation, but they don't.

Berry and Dahlstedt [BD03] describe a more sophisticated audio-visual version of the Dahlstedt/Nordahl simulation, by Dahlstedt, Berry, and Haw. The square lattice with randomly-distributed food is replaced by a randomly produced landscape containing food, music-producing trees, and geological features such as hills, valleys, and lakes. The population is seeded with both randomly-generated individuals and pre-evolved individuals (these serve to guarantee that at least some individuals in the population know how to eat). Genotypes determine appearance, physical behavior, and musical behavior. The physical behavior chromosome describes a neural network that processes information from the environment and uses that information to determine movements in the environment and the base pitch values. The musical behavior chromosome contains thirty-five sound-synthesis parameters, dynamic filters, and function generators that produce complex timbral and melodic patterns. The fitness function is implicit, rather than explicit—to reproduce, individuals must be able to find food and attract partners. To mate, two creatures must be close enough to each other physically, have high enough energy levels, and have similar enough musical behavior chromosomes.

III

Framework

This chapter describes several methods for applying genetic algorithms to time-domain waveforms. These methods cover virtual biology (how can time-domain waveforms be treated as genetic critters that mate and reproduce?), virtual ecology (what sort of worlds can these critters inhabit?), and representation (how are these critters in their world represented as art?).

The primary motivation for this work is to develop algorithms that act directly on digitized waveforms to create sound works whose local and global form are products of the evolutionary process in a changing environment. I believe that this process is capable of producing novel and interesting works and that the process will be perceptible in these works.

3.1 Virtual Biology

3.1.1 Genetic Representation

As discussed in chapter II, the chosen representation strongly reflects the biases[1] of an algorithm's designer. In this case, I have no desire to deal directly with pitches, loudnesses, or rhythms, but rather to deal with concrete sounds. Most of the algorithm variations proposed here use waveforms, read in from standard

[1] These can be both implicit assumptions and intentional applications.

audio files, as chromosomes. The waveforms can either be left in the time domain or be converted to the spectral domain.

A common misconception is that we can help a machine learning algorithm by preprocessing sounds and choosing only a few features on which to run the algorithm. However, machine learning algorithms should be able to use any feature present in the waveform—even features we don't know about and are therefore incapable of giving to the algorithm through preprocessing. Preprocessing can make the algorithm faster, by reducing the information it has to deal with, but by doing so we reduce its potential. For a real-time application of genetic algorithms to waveforms, we have to concern ourselves with computation time. For this reason, I propose some representations that use limited features instead of the entire waveform. There is less directional scope for evolution, but it will still occur.

Chromosomes and Genes in the Time Domain

Using time-domain waveforms as chromosomes has the benefit of leaving the waveform intact. Time domain waveforms can be mapped onto genes in two ways: each sample can be treated as a gene or segments of waveform can be treated as genes. Sometimes it is best to use different mappings for different parts of the algorithm: one mapping for fitness and another for reproduction. Because this representation is in the time domain, it doesn't allow for generalizations of features over time. That is, a waveform can share pitch, timbre, and amplitude characteristics with another waveform but be judged completely different from that waveform if the two waveforms differ greatly in length, while two very different sounding waveforms of the same length may be judged more similar. Sometimes it is desirable to have time be an important component of evolution: it allows the evolution of rhythms. A spectral domain representation should be used to make time a neutral parameter.

Treating individual samples[2] as genes has the benefit of being compu-

[2]Samples are the smallest meaningful unit of a digital waveform. A sample represents the amplitude

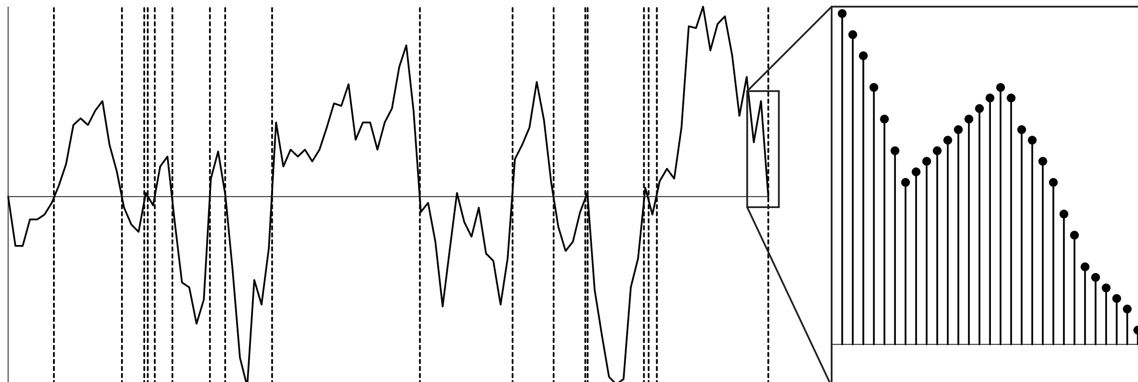

Figure 3.1: A time-domain chromosome's genes can either be mapped onto the segments between zero crossings (left) or the individual samples (right).

tationally easy; we don't have to figure anything out to end up with a string of genes (although the size of the chromosome leads to high computational overhead in any case). There are two major drawbacks to this representation. First, mutations applied to single samples will be unintelligible: they will only introduce noise. Second, sexual recombination will almost certainly produce discontinuities in the offspring waveform, which will be heard as clicks. We can deal with the first issue by always applying mutation functions to perceptually significant chunks of neighboring genes. This leaves us with both mutations and sexual combination causing clicks in the offspring. This can be remedied by adding an offset to modified segments of waveform to eliminate discontinuities. Usually, I use a linear function for the offset so that both the starting and ending points of the edited genes line up with the adjacent unedited genes. The alternative, which is to keep adding constants to the waveform, can result in severe DC[3].

The segments of waveform between zero-crossings[4] work well as genes.

---

of the waveform at a point in time. At the standard CD sampling rate, there are 44100 samples per second.

[3]DC is direct current. When DC is added to a waveform, it centers at some non-zero amplitude rather than zero. It won't sound any different, but unpleasant sounding artifacts (clipping) result from allowing the waveform to have a higher amplitude than the digital representation permits. The alternative, which is to shrink the amplitude of the waveform to prevent this from happening, would make the sounds too quiet to hear if there is too much DC.

[4]Zero crossings are the places where the waveform crosses the x-axis.

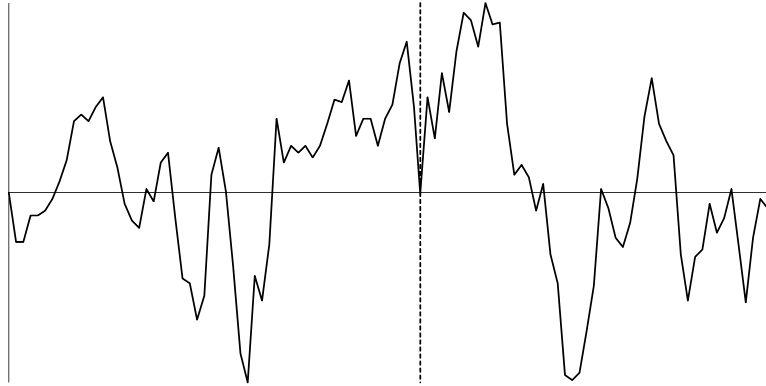

Figure 3.2: A time-domain chromosome spliced at a zero crossing.

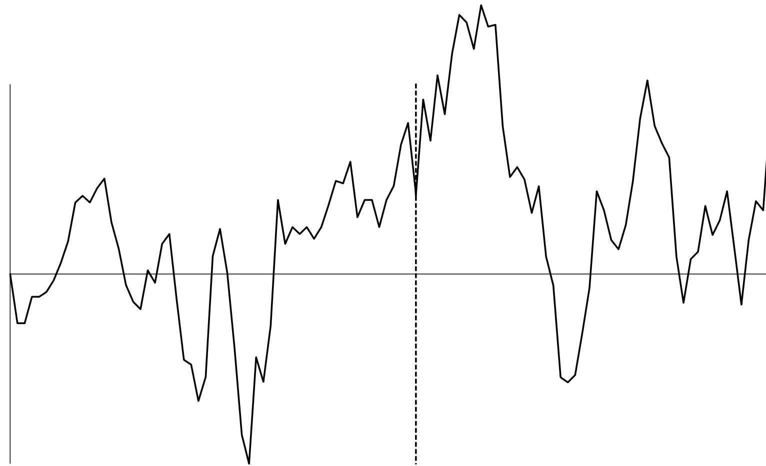

Figure 3.3: A time-domain chromosome spliced at a sample with an offset to prevent clicking.

This means that chromosomes will not have a fixed number of genes, even if all waveforms have the same length: a low pitched portion of a chromosome will have fewer genes than a high pitched portion of a chromosome. Offspring produced using this representation are relatively free from artifacts and sounds retain far more of their sonic identity than they do with the samples-as-genes representation.

These methods of treating genes for producing offspring have tangible effects on the sound of their offspring. Treating instantaneous samples as genes with offsets allows the algorithm to splice and alter the waveform at any point in the gene. When zero crossings are treated as gene boundaries, there are substantially fewer points for splicing and altering the waveform. On the one hand, the former allows much more potential for modification. It is possible, however unlikely, that a population evolving with this method could eventually produce an offspring that matched some target waveform. On the other, the restriction of the zero-crossing method will increase the likelihood that there will be recognizable chunks in the offspring waveform.

If edits only occur on zero crossings, the shortest edit possible will be some chunk of waveform that has some shape[5], otherwise edits can be as short as a sample. In the later situation, the shape of the resultant waveform is a succession of offsets and not based on any contributing sound at all. The resultant sound is rather like noisy gurgles. In the former, the edits can be affected by DC and inaudibly low frequencies. This will lengthen or shorten genes, depending on the interaction between the phase of the low frequency and the rest of the sound. I find the aesthetic result to be preferable in the zero-crossing case, since the resultant sounds are more closely related to the starting population.

I prefer to use the samples-as-genes representation for calculating fitness and the segment-between-zero-crossings-as-genes representation for reproduction. Fitness is easiest to calculate as a pure waveform (see §3.1.3) but offspring sound far better, particularly after many generations of evolution, when all edits occur

[5]Zero crossing edits can be one sample long only if the portion of waveform is at the Nyquist frequency or has DC offset such that only one sample protrudes through zero with no other frequencies.

at zero crossings.

Chromosomes and Genes in the Spectral Domain

My proposal for a spectral domain genetic algorithm is theoretical. Its implementation is one of the next steps in this research program. There are two fundamental ways of representing the spectral domain: either use the spectral analysis as-is or process the data further and use the results. There are many ways in which information can be processed to make genes. I will not go into the myriad of mapping options for processed data; I will instead focus on spectral genetic representation that uses the unprocessed spectral analysis result.

Genes in the spectral domain are three dimensional rather than two dimensional: time, phase, and magnitude. Phase and magnitude can be converted to frequency and amplitude, but this isn't necessary. The values are acquired through Fast Fourier Analysis (FFT). The FFT algorithm analyzes successive windows of time. The size of the window in samples determines the frequency resolution. The analysis of the window yields a series of bins of ascending frequency, each with a phase and a magnitude. There's a trade of between window size (temporal resolution) and frequency resolution. Fourier analysis is incapable of finding frequencies whose wavelengths are longer than the window. However, with long windows, temporal resolution is sacrificed. Other than this, most of the original waveform can be reconstructed by an inverse FFT. Windows can overlap to varying degrees. We can use different degrees of overlap for waveforms of different length to have fixed-lenght chromosomes. This allows a fitness function using a spectral domain chromosome to generalize fitness over different lengths of waveform.

Processed Genes

There are two reasons to process genes: focus and speed. By breaking down the waveform into one or two features, an algorithm can be designed that ignores other aspects of the sound and evolves along the designated dimensions.

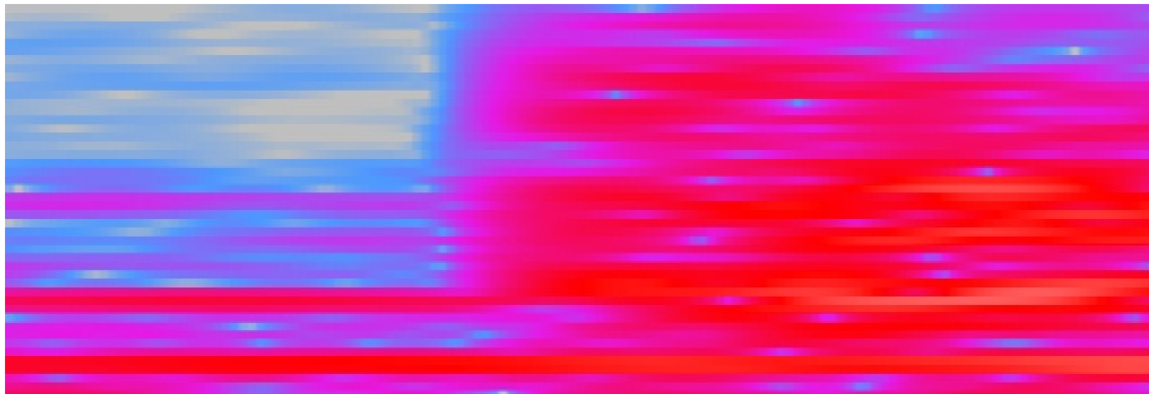

Figure 3.4: A chromosome in the spectral domain. The vertical access is frequency; the horizontal access is time; the shading represents magnitude. .

Processed genes have the potential to be faster for fitness calculation than waveforms. They might have a handful of genes per millisecond instead of 44.1 genes, which is the number of samples per millisecond with CD quality audio.

I can imagine artistic reasons for wanting to constrain the fitness function to well-defined and well-understood dimensions. One might want to evolve one or two easily perceived dimensions. One might also want to apply fitness function to a large number of variables (for instance, all of the various components of timbre that have been defined to date), but specify weights for each variable as an explicit part of the fitness function rather than use the weights implicit in the unprocessed waveform.

For real-time versions of the algorithm, it is desirable to process the waveform to produce simpler genes. This makes calculating fitness less computationally intensive. We can choose to tie our chromosomes to time, with the same rhythmic consequences as a time-domain representation. Or, we can sample time variably and have fixed-length chromosomes and allow the algorithm to generalize across time, as with a spectral-domain representation. Processed genes can be amplitude envelope, pitch of partials, or or some combination thereof.

Using the results of processing instead of the waveform itself only saves

us time under some circumstances. The computation involved in processing the waveform and applying the fitness function on the simpler chromosome has to be faster than applying the fitness function to the waveform itself. Otherwise, the good of the simpler chromosome is cancelled out by the trouble of getting it. An expensive process can be worth it if fitness will be calculated many times on each processed chromosome. For instance, in a world where critters look for self-similarity in a mate (see §3.1.3), every time an individual reproduces, it calculates the fitness of all other critters. This means that if there are n critters in the population, fitness functions are calculated n^2 times instead of n times.

It might also be possible to preprocess waveforms and maintain temporally linked processed and unprocessed chromosomes. Given the temporal shuffling that occurs during sexual reproduction and mutation, this method would almost certainly gradually desynchronize. Experimentation is necessary, but it might be possible to maintain *good enough* synchronization to allow evolution to continue for the duration of a work. I would certainly try this before processing each waveform if I wanted to experiment with a large number of intensive processes (see above regarding timbre).

Other Information in Chromosomes

In the simplest worlds, critters are only sound. In more complex worlds, as those described in §3.2.2, critters have an additional chromosome. The chromosome contains information on location, movement variables, and mating range. Location signifies an individual's location in the world. Movement variables encode an individual's possible movements. Mating range specifies the maximum distance from the individual that another individual can be in order to be considered a potential mate. For a ring world, I represent location as a single gene: degrees. Mating range is also represented with a gene for degrees. For a plane, I would need two genes: either x and y coordinates or angle and radius. My choice of planar representation would depend on how I wanted to compute movement.

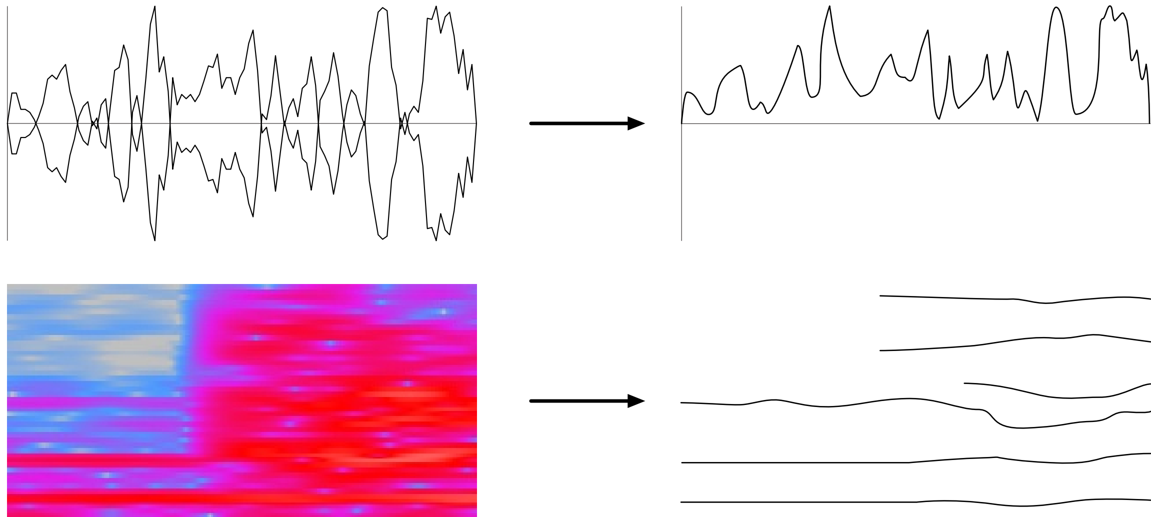

Figure 3.5: Examples of chromosomes made from processed genes: amplitude envelope (top); partials (bottom).

In my ring world, movement is a drunken walk around the ring. There are genes for maximum movement (in degrees) in either direction. There are two genes rather than one, since it allows the individual to prefer one direction over the other. In extreme cases, it is possible to have individuals who can't move at all, or who only move in one direction. There are also genes that give a range of time in which movement occurs, and a gene with the probability of resting for some amount of time so that the resulting sound is not in constant motion.

Depending on the complexity of the world, this chromosome could encode any number of relevant variables. Genes could encode sex, energy, place in the food chain, etc. Without experimentation, it's not clear how this would contribute to sound. But, it would definitely open up more possibilities for modeling social behaviors that might contribute to an implicit fitness function (see §3.1.3).

3.1.2 Time

In a conventional genetic algorithm, how chromosomes exist in time is not an issue. Whether or not chromosomes are fixed length (which is standard) or

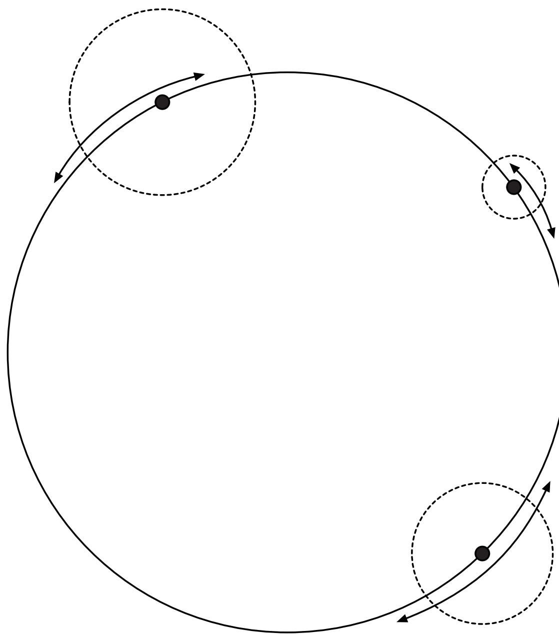

Figure 3.6: Critters with other information in their chromosomes: each critter has a mating range (dotted line) and a range of possible movement (arrows).

variable length, new generations replace old generations instantaneously. Because my goal is to have the musical artifacts produced by the algorithm *be* the evolution taking place rather than being the *product* of generations of evolution, time is an important aesthetic factor. I think of time differently in versions of the algorithm intended to produce musical works and versions of the algorithm intended for sound installations.

For compositional use, the length of the chromosome is it's lifespan. I've experimented with both fixed-length and variable-length chromosomes. Fixed-length chromosomes have discrete generations whose members all begin and end simultaneously (figure 3.7). Variable-length chromosomes have arbitrary lengths that lie within some fixed upper bound (figure 3.8). This allows individuals from different generations to overlap, since they are replaced independently of the rest of the population.

With this treatment of time, the primary difference between fixed-length

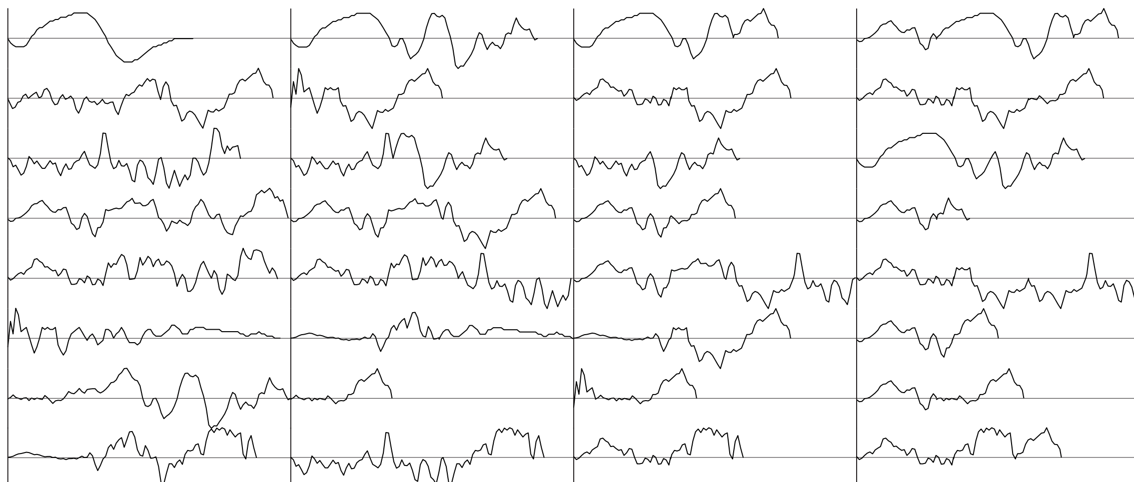

Figure 3.7: Critters with fixed-length chromosomes all have the same lifespan, regardless of waveform length.

and variable-length chromosomes is rhythmic. Pieces made using fixed-length chromosomes have an inherent rhythm that is dictated by chromosome length. Variable-length chromosomes tend to have an emergent rhythmic quality as well, but the rhythms change over the course of the piece—sometimes gradually, sometimes quite dramatically.

For sound installations, I have made a real-time version of the algorithm. Genetic algorithms, especially with the long chromosomes that result from using real-world recorded sounds, are rather computationally intensive. Therefore, it is impractical to treat time in the same way as I treat it for tape pieces. Sounds cannot all play and reproduce at once. Furthermore, it might be desirable to have a sparser soundscape than the soundscape that emerges from the tape version of the algorithm, which plays all sounds in the population at all times.

Critters are played periodically. The frequency at which they are played is based on their fitness, which is calculated when they are produced, rather than recalculating it each time they attempt to mate.[6] Fit individuals are played fre-

[6]This is relevant when critters evolve in changing, rather than static, environments (see §3.2.1). As faster computers become available, there is no computational reason that fitness couldn't be calculated

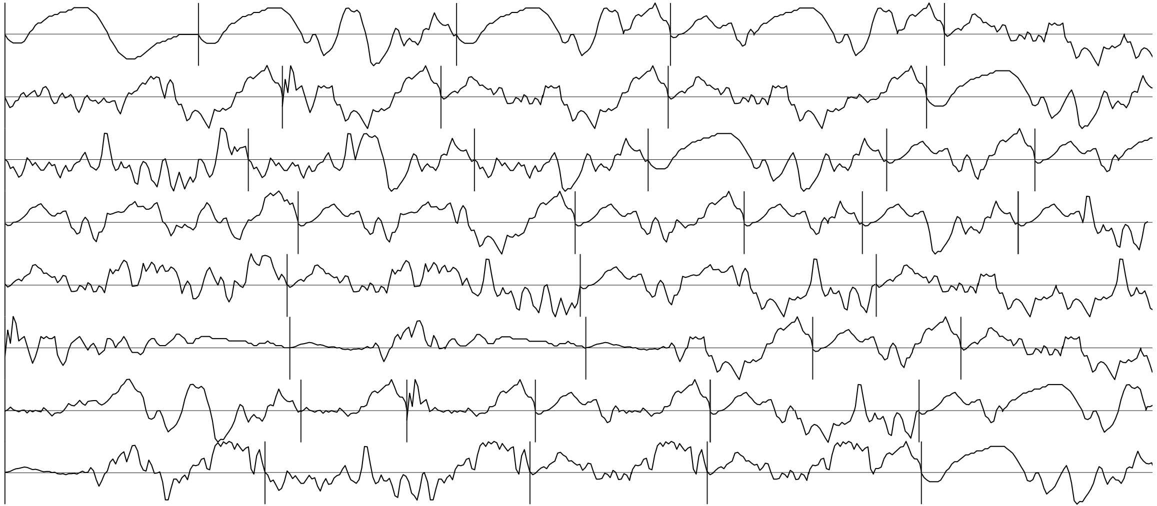

Figure 3.8: Critters with arbitrary-length chromosomes overlap in lifespan. Their waveform length is their lifespan.

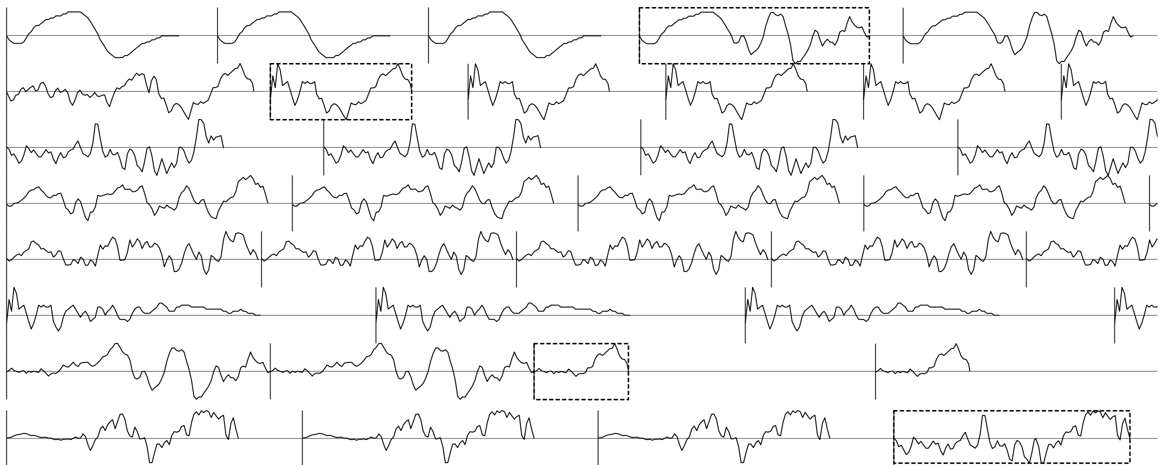

Figure 3.9: Critters in real-time play periodically but their lifespan is unrelated to their length. (Dashed boxes indicate the first playing of each sound.)

quently; unfit individuals are played rarely.

Since reproduction is computationally intensive, only one individual reproduces at a time. Each time an individual plays, it is added to the reproduction cue. When an individual reproduces, all other instances of that individual are removed from the cue. This guarantees that the offspring that occupies the individual's place in the population will be played at least once before it reproduces. The frequency of playback and reproduction are multiplied by variables. This gives me control over the relative sonic density of an installation.

3.1.3 Fitness

Explicit Fitness

The fitness function directs the course of evolution. It influences which critters reproduce successfully and ultimately determines what genetic material remains in the population as it converges. Fitness in genetic algorithms is usually explicit: that is, it is defined by a fitness function. In more complex ecological systems, there are other factors that result in implicit fitness.

A fitness function maps members of the population onto numbers representing fitness. The more fit a critter is, the more likely it is to reproduce. There are many ways we could calculate fitness for waveforms—the choice is primarily aesthetic. My goal is to present the process of evolution, not evolve some ideal critter or evolve a critter that meets certain criteria. In the absence of a set of criteria I'd like my critters to evolve towards, I have chosen a model that evolves based on comparison to a waveform. This comparison can be the correlation between chromosomes or the sum of the difference between corresponding genes over the length of the chromosome.

---

each time an individual reproduces. Although, general consensus from the nature/nurture debate seems to be that early development plays a substantial role in an individual's fitness, and that untapped genetic potential isn't readily activated by a better environment later in life.

Waveforms as Fitness Functions

Any waveform can be used in the fitness function, but the sort of waveform chosen will cause different behaviors in the algorithm. We can choose an arbitrary waveform, and the entire population will evolve towards that waveform. Depending on the representation we choose for the chromosomes (see §3.1.4), different characteristics from the target waveform might manifest in the population.

The fitness function does not need to be restricted to a single waveform. As discussed in §3.2.2, the function can change in time, in space, or by individual. To use real-world sounds for evolution, I place one or more mics in the room and map them onto the virtual space. An individual's fitness is calculated by taking the sample of real-world sound that begins when the sound is created and is the length of the sound. With real-world fitness functions, the audience can impact the course of evolution. Also, if the mics are placed in the vicinity of the speakers, there will be a feedback situation in which sounds of the population itself become part of the environment.

The fitness function doesn't have to be explicitly tied to space. Individuals can carry a chromosome that represents a fitness function that will be applied to potential mates: a personalized, social fitness, rather than an environmental fitness. A simpler application of this principle is to have critters look for self-similarity in a mate by applying themselves as a fitness function. This leads to more biodiversity, since no sound is inherently unfit. It can also lead to a sort of speciation, as sounds inbreed primarily with similar sounds. Often, I apply both an environmental fitness function and a self-similarity function by calculating the two fitnesses, applying weights to them so that one might have a stronger affect than the other, and adding them together.

Other Fitness Functions

Although I've chosen to use sounds as fitness functions in my projects, the possibility exists to use other things. Just as any sonic features can be used

as genes (see §3.1.1), they can also be used as fitness functions. Fitness functions can incorporate sonic features—tied to time or not.

Biodiversity

Sometimes the fitness function incorporates a biodiversity factor in addition to the measure of fitness. Some number between 0 and 1 is multiplied to a critter's fitness each time it reproduces. This makes it less likely for a few fit individuals to dominate the next generation.

Implicit Fitness

Implicit fitness is based on ecology. It includes any factor that might affect the ability of an individual to reproduce. For instance, a critter with a very small range in which it's willing to look for a mate might die without reproducing, regardless of its calculated fitness. When this happens, it's slot in the population will be taken up by the offspring of other individuals. Life-span is another contributor to implicit fitness. An individual that lives longer is more likely to be chosen as another critter's mate than a shorter individual with the same explicit fitness.[7] A critter with only one or two mating options will settle for an unfit critter that it probably wouldn't have matted with in a more densely populated area. More complex ecologies give rise to more factors that contribute to implicit fitness. An appropriately designed ecology doesn't require an explicit fitness function for

[7] Although it might seem counterintuitive for unfit individuals to have longer lifespans than fit individuals, this has desirable effects on both evolution and perception. Evolutionarily, this adds a mechanism to increase biodiversity in the population. Because unfit individuals are given more opportunities to compete to mate, they have an implicit fitness increase that isn't represented by the explicitly calculated fitness function. Perceptually, fit sounds will be heard frequently even though fit individuals may be replaced by offspring often. Although the exact character of the offspring will differ fro the parents, there will be a familial resemblance between the fit individuals and their offspring. The frequently occurring fit sounds will be seen as instances of a single type despite individual differences. Effectively, the fit type will be heard to play extremely frequently. Think of a field of clover—there are lots of different plants but we see them as a single mass of clover that occupies a large part of our visual field. The unfit sounds will occur rarely and will sound novel relative to the backdrop of fit sounds. Think of animals that like clover. You'll see some rabbits, bees, and butterflies. There are few enough of them relative to the cloverThat is, unless you lack coyotes to eat the rabbits, in which case rabbits become so fit that you see a field of rabbits on clover and individual rabbits lose significance.that you can tell individual kinds from the constant backdrop of clover.

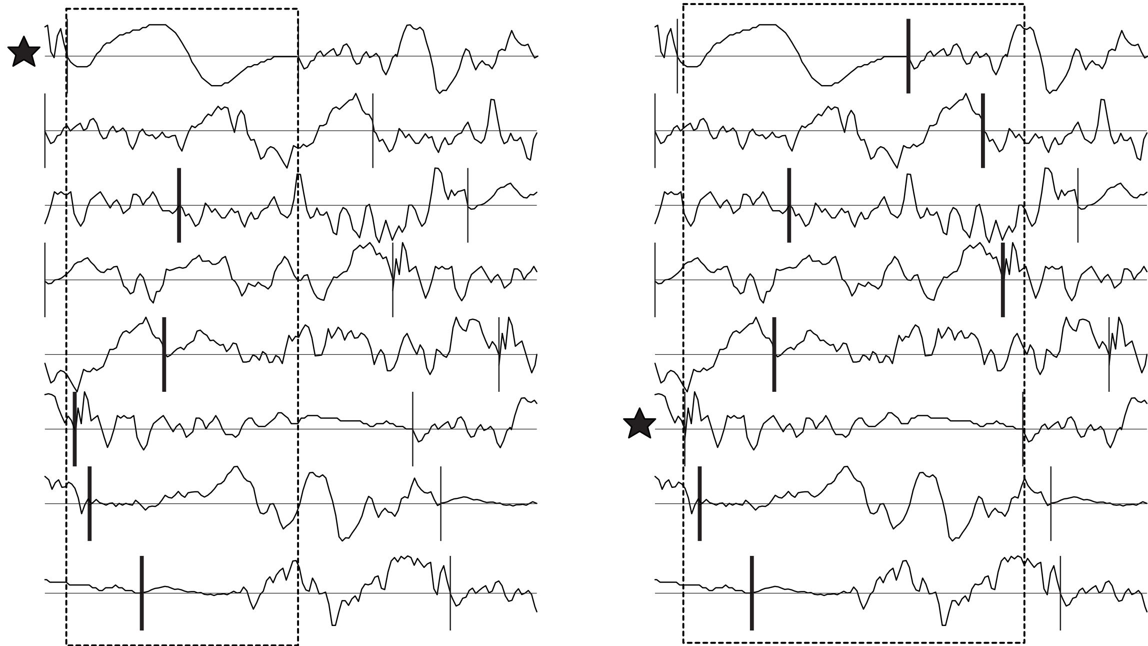

Figure 3.10: Implicit fitness can result from length. The starred waveform on the left is shorter than the starred waveform on the right. The heavy lines mark the beginning of waveforms that start while the starred waveforms are playing. The waveform on the left has five opportunities to reproduce, while the waveform on the right has seven.

evolution to occur.

3.1.4 Reproduction

Reproduction is the process by which a critter is replaced by a new critter. This section deals with the biological factors in reproduction. The details of who reproduces and when are dictated by the fitness (§3.1.3) and the ecological of the world in which reproduction occurs (§3.2.1).

Critters can reproduce either sexually or asexually. With asexual reproduction, new critters are the offspring of a single parent. Sexual reproduction is used more often. Critters are the offspring of two parents. Sexual reproduction has the potential to produce offspring that have beneficial aspects of both parents.

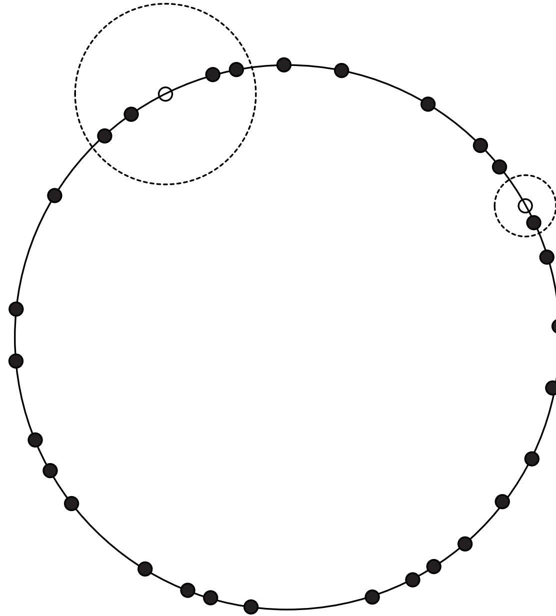

Figure 3.11: Implicit fitness can result from proximity of other critters. The small circles represent all the critters in the population. Consider the two outlined critters. The dashed lines represent the ranges over which they'll seek mates. The one on the left has four potential mates; the one on the right has only one. Assuming all critters have the same fitness, the potential mates on the left will have a one in four chance of being chosen as mates, while the potential mate on the right will be the guaranteed to be chosen as a mate.

It also has the potential to produce offspring with the least beneficial aspect of both parents, but these offspring will be unfit and will have a reduced chance of propagating. Evolution has the potential to occur much more quickly.

Sexual reproduction splices the first part of one parent's chromosome to the second part of the other parent's chromosome. Splicing occurs at the *crossover point*. In critters with multiple chromosomes, the process is repeated for each chromosome. Sexual reproduction manifests differently in the time domain and the spectral domain, since the time domain has one-dimensional chromosomes and the spectral domain has two-dimensional chromosomes.

New material can only be introduced by mutation. Mutation slightly alters a few genes of the offspring. Mutation probabilities are usually very low. Although mutations are the source of novelty, the underlying assumption is that the gene pool already contains good genes and that mutation is only needed to nudge the population out of local fitness maxima and to add biodiversity to the gene pool as it converges.

Asexual Reproduction

Asexual reproduction is the simplest form of reproduction. An individual simply is replaced by another version of a critter that has gone through the process of mutation (§3.1.4). Since the probability of mutation is usually low, there's no guarantee that the offspring will differ from the parent. Evolution only occurs if individuals are replaced by a random member of the population, with each individual's probability of taking a given slot in the population based on fitness. In this case, the population will be changed to have many copies of the fittest critters, some of which will have different mutations.

Time Domain Sexual Combination

Sexual combination is simply taking the first part of one parent's waveform and the last part of the other parent's waveform and splicing them together.

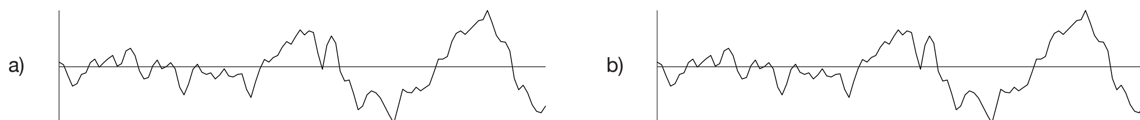

Figure 3.12: Asexual reproduction in the time domain. Note that a) parent and b) offspring are the same.

The crossover point is where the cutting and splicing occurs. The crossover point can either be fixed, usually at the half way point, or it can be random. In order to preserve length with fixed-length chromosomes, the random crossover point is the same for both parents.

What all this means varies with representation. We can calculate crossover point either with a samples-as-genes representation or with a waveform-between-zero-crossings-as-genes representation. I usually use samples to calculate the crossover point, then nudge the crossover point to the next zero crossing. With variable-length waveforms, the half-way point isn't necessarily the same for each parent. The algorithm can either be set up to take half of each parent or to take half of one parent and the same amount of the other parent. In the former case, this results in a waveform that averages the length of the two parents. Eventually the population will converge to a fixed length. In the later case, existing lengths can propagate, but new durations can never be introduced through crossover. There will be slight deviations in time from moving the crossover point to a zero crossing, but these will almost always be imperceptible.

With random crossover points and variable-length chromosomes, a different crossover point can be chosen for each parent. The resulting chromosome can be arbitrarily short, or potentially as long as the two parents spliced end to end. Musically, this results in much more rhythmic flexibility.

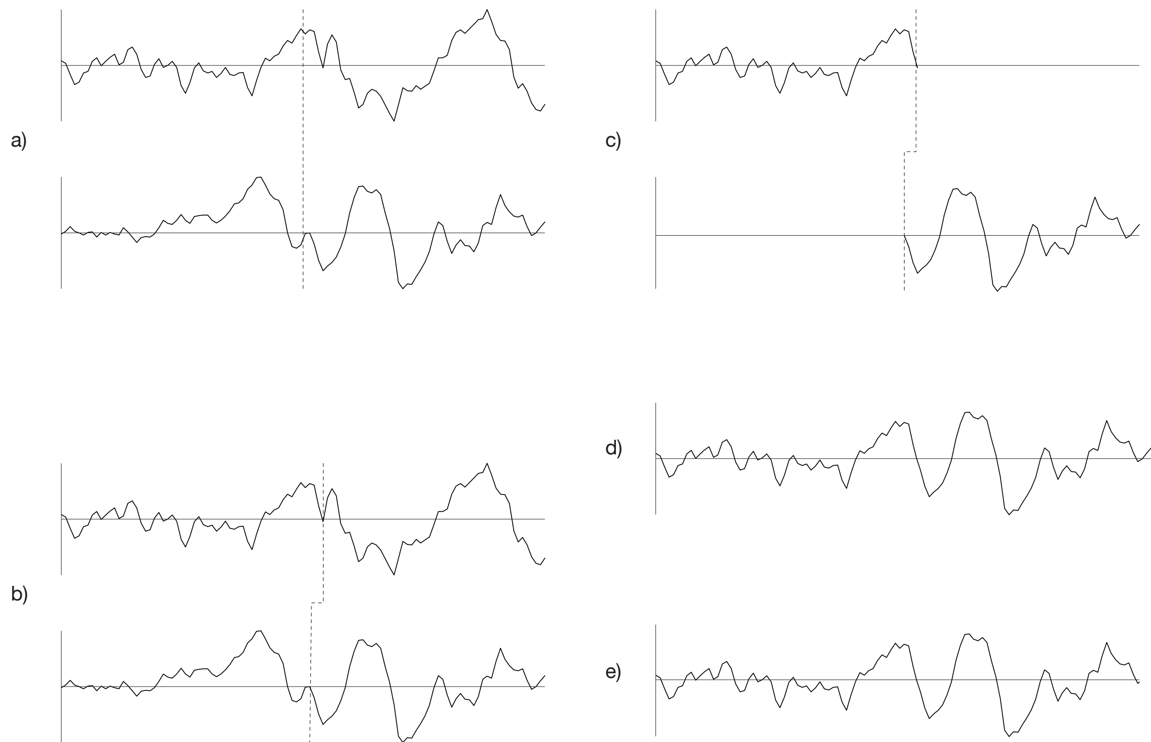

Figure 3.13: Sexual reproduction in the time domain with a fixed crossover point. a) The halfway point is designated by a dotted line. b) The crossover point is nudged to make edits occur at zero crossings. c) The beginning of one waveform and the end of the other are taken. d) Offspring.

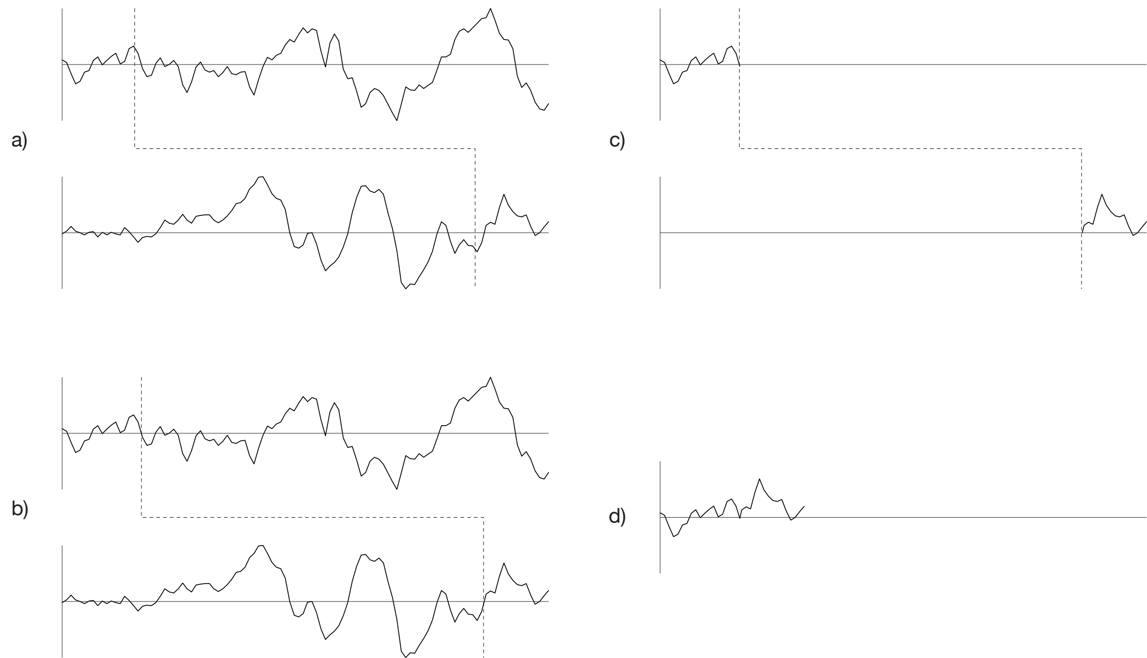

Figure 3.14: Sexual reproduction in the time domain with a random crossover point. a) The randomly selected crossover points are designated by a dotted line. b) The crossover point is nudged to make edits occur at zero crossings. c) The beginning of one waveform and the end of the other are taken. d) Offspring.

Time Domain Mutation

Mutation allows changes to be introduced into the population. This introduces genetic material that will allow the population to reach a higher level of fitness than it could using only the genetic material available within the initial population. Some mutations are based on common gene transcription errors: various splicings of chromosome segments. Chromosome segments can be duplicated, dropped, swapped, rotated, or reversed. Each mutation has it's own characteristic sound, but the splice itself has the same sonic quality as the sexual reproduction edit. Other mutations change the shape of the waveform: amplify, exponentiate, change length. They introduce change with a gradual quality that contrasts musically with the abrupt changes that come from splicing.

Duplicate. A segment of neighboring genes can be duplicated some random number of times (figure 3.15). Duplication preserves the sound of the mutated waveform, but it changes the sound of the waveform more dramatically than the other splicing mutations. If the mutation affects very short segments of waveform, it can introduce pitch, even to a waveform that consists of only white noise. If the mutation affects longer segments of waveform, it is perceived as rhythmic looping. For fixed-length chromosomes, the end of the waveform is truncated.

Drop. A segment of neighboring genes can be dropped from any point in the waveform (figure 3.16). This always shortens the waveform. For fixed-length chromosomes, the end of the waveform is filled in with silence.

Swap and Rotate. Two segments of waveform can be swapped or rotated(figure 3.17). These are computationally the same, but conceptually different. Swapping takes two neighboring, equal-lengthed chunks of waveform and swaps them. Rotating cuts one chunk of wave form and pastes it someplace else in the waveform, sliding the rest of the waveform into its position.

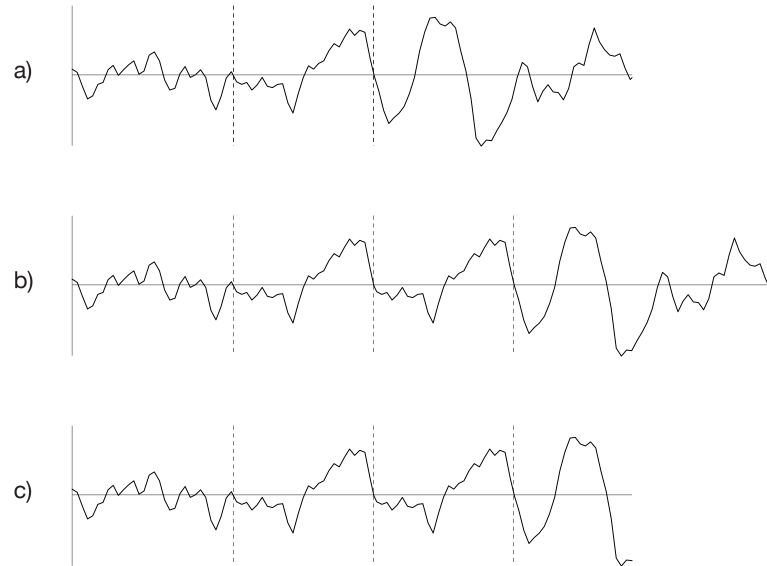

Figure 3.15: a) Offspring before duplication. The portion of the chromosome selected for mutation is delineated by dotted lines. b) Offspring after mutation in which selected genes have been duplicated. c) Offspring chromosome truncated for fixed-length case.

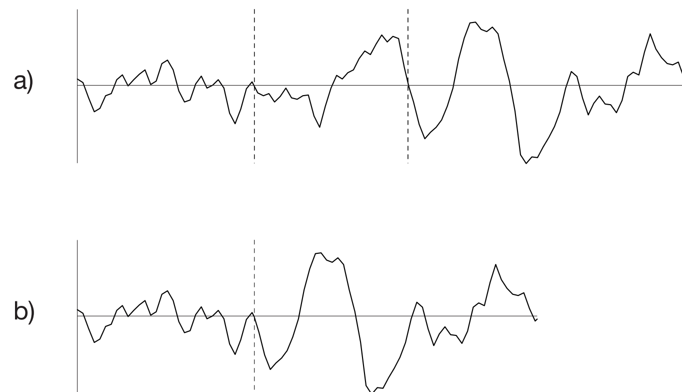

Figure 3.16: a) Offspring before dropping. The portion of the chromosome selected for mutation is delineated by dotted lines. b) Offspring after mutation in which selected genes have been dropped. This mutation type is not used in the fixed-length chromosome case.

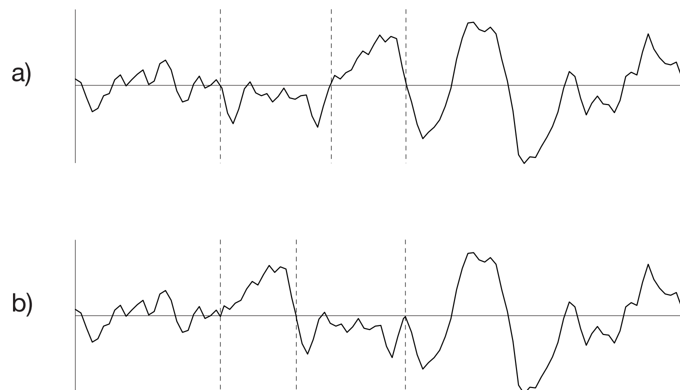

Figure 3.17: a) Offspring before swapping/rotating. The portion of the chromosome selected for mutation is delineated by dotted lines. b) Offspring after mutation in which selected neighboring genes have been swapped.

Reverse. A segment of waveform can be reversed (figure 3.18). It stays in the same place, but is played backwards. For very short chunks of waveform, there will be little perceptible change in sonic quality, but the fitness of the waveform will almost always slightly change. For long chunks of waveform, the quality of backwards playing will be perceptible.

Amplify. A segment of waveform can be amplified or attenuated (figure 3.19). If there is an unchanging environment, over time, amplitude modifications can sculpt the population to match the amplitude envelope of the target environment.

Exponentiate. A segment of waveform can be raised to a power (figure 3.20). This is the most distorting of the mutations. I usually give it a much lower probability than the other mutations, it has a distinctive sound.

Change Length. A segment of waveform can be lengthened or shortened (figure 3.21). This both changes both the length of the segment and its pitch.

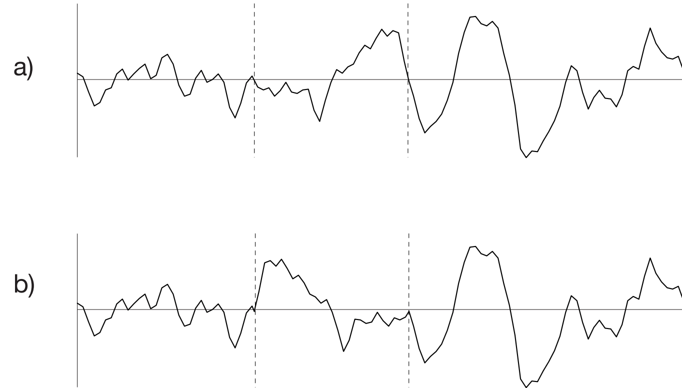

Figure 3.18: a) Offspring before reversal. The portion of the chromosome selected for mutation is delineated by dotted lines. b) Offspring after mutation in which selected genes have been reversed.

Spectral Domain Sexual Combination

Sexual Combination in the spectral domain will introduce artifacts. There is a particular sound to spectral domain processing. Reproduction in the spectral domain should only be used if the intention is to evolve a population of sounds that is increasingly overtaken by these artifacts.

Spectral domain chromosomes are three dimensional, and sexual combination can occur on any of the three dimensions. Time windows can be treated as the primary axis of the chromosome, with the first part of one parent and the second part of another parent combined. The result would very similar to the reproduction in the time-domain waveform. Frequency can be treated as the primary axis, taking the high bins from one parent and the low bins from the other. Or, the spectral information can be treated as the primary axis, taking the phase from one parent and the magnitude from the other parent.

Spectral Domain Mutation

There are a myriad of mutation options in the spectral domain. All of them will introduce artifacts. The spectral domain can borrow mutations from the

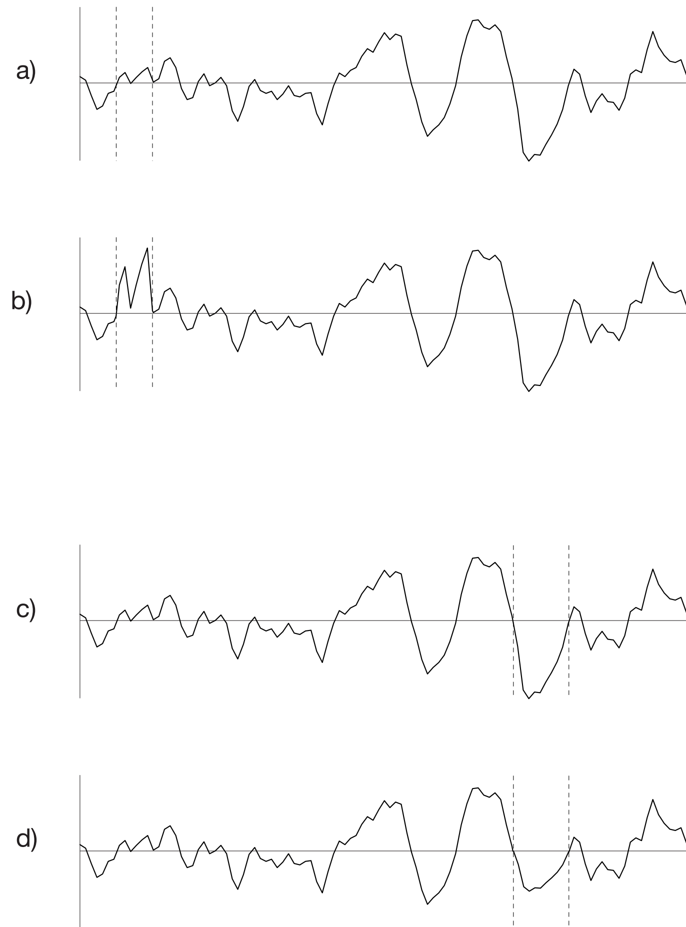

Figure 3.19: a) Offspring before amplification. The portion of the chromosome selected for mutation is delineated by dotted lines. b) Offspring after mutation in which selected genes have been amplified. c) Offspring before mutation. The portion of the chromosome selected for mutation is delineated by dotted lines. d) Offspring after mutation in which selected genes have been attenuated.

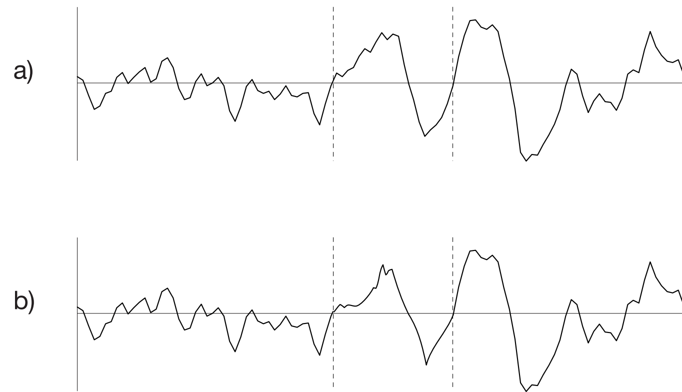

Figure 3.20: a) Offspring before exponentiation. The portion of the chromosome selected for mutation is delineated by dotted lines. b) Offspring after the amplitude selected genes has been raised to a power.

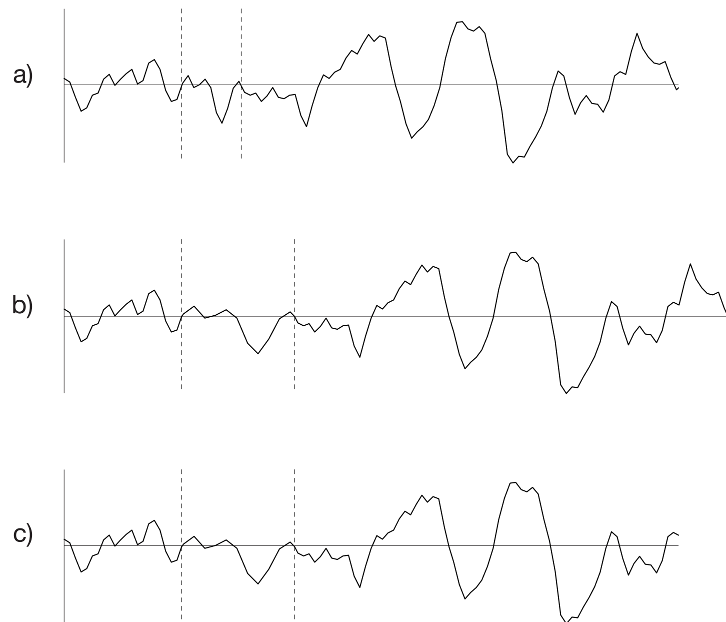

Figure 3.21: a) Offspring before changing length. The portion of the chromosome selected for mutation is delineated by dotted lines. b) Offspring after mutation in which selected genes have been resampled by a random amount. c) Offspring chromosome truncated for fixed-length case.

time domain; the same mutations can apply in the frequency domain. Chunks of spectral data can be shifted in time or frequency. Phase and frequency information can be swapped. Neighboring bins can be flipped in frequency so that the highest frequency becomes the lowest and vice versa. Etc.

3.2 Virtual Ecology

Virtual ecology refers to the worlds the sonic critters inhabit. In a conventional genetic algorithm, the population is fixed in size, generations are discrete, and the world is defined solely by the fitness function. All critters in the generation reproduce and are replaced by other critters at the same time. All critters are assumed to occupy a single point that has a fixed fitness function. Worlds don't have to be defined so simply. Generations don't have to be discrete. Critters can have different lifespans; generations can overlap. There doesn't have to be a single fitness function for the whole world. Fitness functions can vary by both location and time.

3.2.1 Population

Lifespan

In a conventional genetic algorithm, the population has a constant size and generations are discrete. In the real world, population size varies and and, for many species, generations overlap. If we are to have non-discrete generations, we need to have some notion of lifespan. Individuals need some reason to live for some amount of time and then die. There are several possible options for determining lifespan. Lifespan can be genetically determined, environmentally determined, or some combination of the two. A critter whose lifespan is the length of it's soundfile has a genetically determined lifespan. We could design a world in which critters die because of some interaction with the environment, with no regard to genetics. Or we could let lifespan be determined by explicit fitness, the interaction between

genetics and the environment. These issues are discussed further in §3.1.2 and §3.1.3.

Fixed vs. Variable Population Size

The computer's memory is a limiting factor: there is a fixed maximum size to any population. A world's population size can either be fixed or variable. If the population size is fixed, one individual must die for another individual to be born. Otherwise, this is only an issue if the population reaches its maximum size.

In future work, I would like to explore variable population size. If the population size is allowed to vary, something has to determine lifespan and something must determine how often individuals reproduce. We can eliminate fitness functions entirely and design reproduction pressures in the world itself. Individuals can look for particular features in a mate. I have experimented with self-similarity, but there's no reason individuals can't have another chromosome that represents the fitness function that defines their desired mate (see §3.1.3). The situation in the world itself can influence lifespan. A predation model would accomplish this. Features of the sound place individual sounds somewhere in the food chain, and some notion of energy could allow reproduction without death for efficient predators, while others might reproduce fewer times. An attractive feature of variable population size is the potential for unstable populations. There is no guarantee of equilibrium: overactive predators can wipe out their food supply and starve. A piece produced this way can have a natural ending instead of an arbitrary length.

Who Reproduces?

With a fixed-sized population, the algorithm must decide who gets to be a parent. It's not enough to have a fitness function; it has to be used. When an individual dies, the fitness function can go to work twice, picking two parents whose offspring will fill the individual's place, or it can go to work once, finding a mate for the dying individual. In the former case, fitter individuals will have greater

presence in the next generation. In the later case, biodiversity is increased because even the least fit individuals will reproduce at least once—fitter individuals will be chosen as mates. The former will cause the population of sounds to change more rapidly than the later.

3.2.2 Worlds

Although genetic algorithms are designed to explore large-dimensional spaces, we can think of them as functioning in a metaphorical world with some dimensionality. Classical genetic algorithms exist in an unchanging, zero-dimensional (single-point) world: there is only one environment, which is defined by a fitness function, and it doesn't change as a function of time. The real world isn't like that. We exist on a rough sphere. There are many biomes, and instances of the same biome are physically separated (hence the dearth of penguins in the arctic). We have a diversity of climates, sub-climates, and micro-climates. We have physical barriers that can be crossed by some species and not by others (neighboring bodies of water can share the same population of frogs, but their fish populations are isolated—unless a flood connects them temporarily.). We have seasons and a much slower cycle of global cooling and warming. We have rare sudden events that can drastically change the entire environment (volcanos, meteors), and sudden events that drastically impact some subset of species in an area that alter the micro-climate (forest fires can create meadows by eliminating trees) or otherwise affect implicit fitness (pollution can cause algae blooms that drastically reduce oxygen in affected waters).

Since I'm interested in the artistic results of the process and not in using genetic algorithms to solve some problem, different worlds are an attractive way to affect the overall process and produce different sorts of artistic works. Changing environments increase the possibility for musical surprises. Different worlds produce different large-scale forms. There are two fundamental ways in which a world can be altered: its shape (dimensionality and topology) and its changeability.

Dimensionality and Topology

A world can have any dimensionality the designer feels is worth implementing. Naturally, practical considerations have a huge impact on what one is willing to program. Since we're dealing with sounds, speakers and the space in which they're installed are the primary constraining features. My first experiments treated each speaker as an island. Critters had some probability of migrating between islands, but they were otherwise separate. Each speaker had its own environment that did not interact in any way with environments of other speakers. Finite but unbounded one-dimensional spaces allow environments and critters to exist on a continuum. I have implemented a ring world which can be mapped onto any speaker setup. One can imagine other one-dimensional topologies, such as a line world between two speakers, or multiple rings in neighboring rooms that are interconnected at the doors. Although it is hard to spatialize perceptibly with a standard speaker setup, one could imagine mapping critters onto a plane projected onto a room[8], or (with appropriately placed speakers) a sphere. But, two and three dimensional worlds have not yet been practical for one of my projects.

My ring-world maps a ring onto some number of mics and speakers. Each critter has an additional chromosome that contains it's location (in degrees), a distance in which it's willing to look for a mate (also in degrees), and variables that define the motion it makes over it's lifetime. Critters move with a drunken walk that is defined by an amount of time between choosing a type of movement, a probability of not moving at all (this is queried each time it looks for a new movement type), a range of degrees it might move up, and a range of degrees it might move down. The up and down ranges are different, so critters might move with different speeds and directional tendencies. Since the world is continuous but the mics are discrete, the fitness functions from each mic are spatially interpolated so that a slightly different fitness function is applied at each point in space. When

[8]This would be perceptually relevant if there was a grid of speakers the plane was mapped onto instead of a ring around the room. With the standard ring setup, the connectedness would affect evolution but the sounds would still sound like they came from the sides of the room.

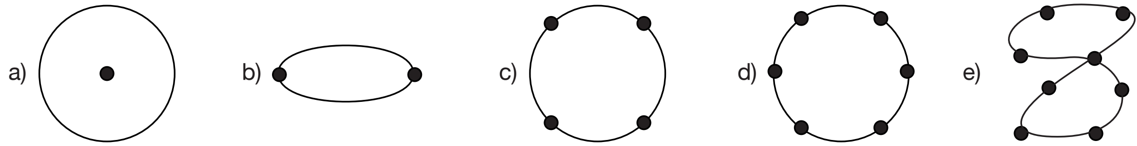

Figure 3.22: Ring worlds can be mapped onto multiple speaker configurations. a) mono b) stereo c) quad d) and so on, as many speakers as you like e) a 3-D space.

it comes time for an individual to mate, it will look for an individual within it's genetically encoded mating distance, rather than examine the entire population.

Changing Environments

I've explored several possibilities for changing environments. Changes can be gradual or sudden. Fitness functions can incorporate self-similarity or sounds from the real world. Fitness functions and types of change can be algorithmically combined to create more complex environments. One can also imagine algorithmically changing the world's topology or dimensionality, although I've not yet encountered an artistic situation that called for me to implement this.

I usually combine gradual and sudden environmental changes with a single algorithm. Gradual change can be implemented by algorithmically choosing two fitness functions. The algorithm starts with one function, then gradually interpolates it with the second fitness function. When only the second is left, the algorithm chooses a third fitness function and the process continues. The algorithm has variables that allow me to set a length of interpolation and an amount of deviation to that duration (since real-world change isn't like clockwork). Sudden change instantly replaces the existing function with another. There is some random amount of time between sudden changes. As with the interpolation duration, this time is represented by a constant and a range of deviation. Sudden changes pick two new fitness functions to interpolate between. Usually I choose deviations

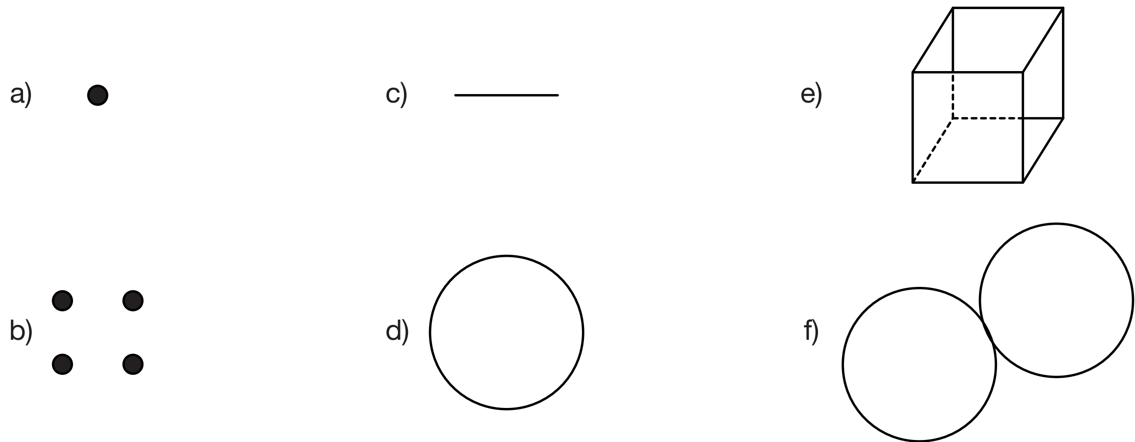

Figure 3.23: Worlds can have different geometries. a) the world of a conventional genetic algorithm is a single point. b) a world with multiple islands. c) a continuous, linear world. d) a ring world. e) a three-dimensional world. f) multiple rings worlds.

such that sudden changes happen with more variability than gradual changes.

IV

Sample Implementations

This chapter provides some sample implementations of the framework described in chapter III. Most of the material from chapter III is presented again, but in a more formalized format: flow charts for specific applications with prose explanations.

The first implementation is a simple algorithm for evolving fixed-length chromosomes. This is the algorithm whose results will be presented in §5.1. The other implementations expand on this basic algorithm. The second uses variable-length chromosomes and a world with multiple, changing, island environments, similar[1] to the one used to produce the tape pieces in §5.2. The third evolves variable-length chromosomes in a ring world and is similar to the real-time algorithm in §5.2.4. For clarity, these implementations are somewhat less complicated than the versions I've used for compositions.

[1] The algorithm is conceptually the same, but for the purpose of explanation, efficiency has been sacrificed for clarity.

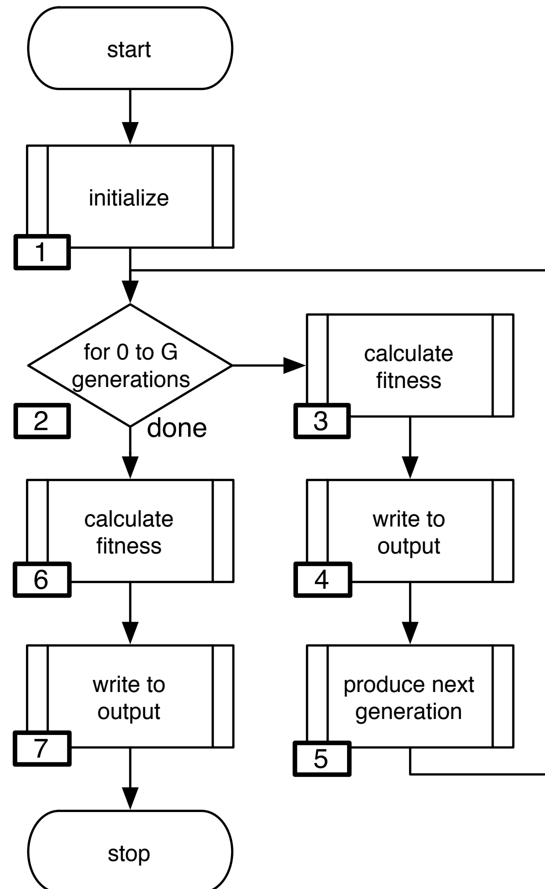

Figure 4.1: Evolving fixed-length, time-domain waveforms in a single, unchanging environment

4.1 A Simple Algorithm for Evolving Fixed Length, Time Domain Waveforms in a Single, Unchanging Environment

4.1.1 The main program

The main program (figure 4.1) initializes the algorithm (1) and evolves the population for G generations (2). A generation of evolution consists of calculating the fitness for all individuals in the population (3), writing a generation to output (4), then producing the next generation (5), then copying the offspring onto the

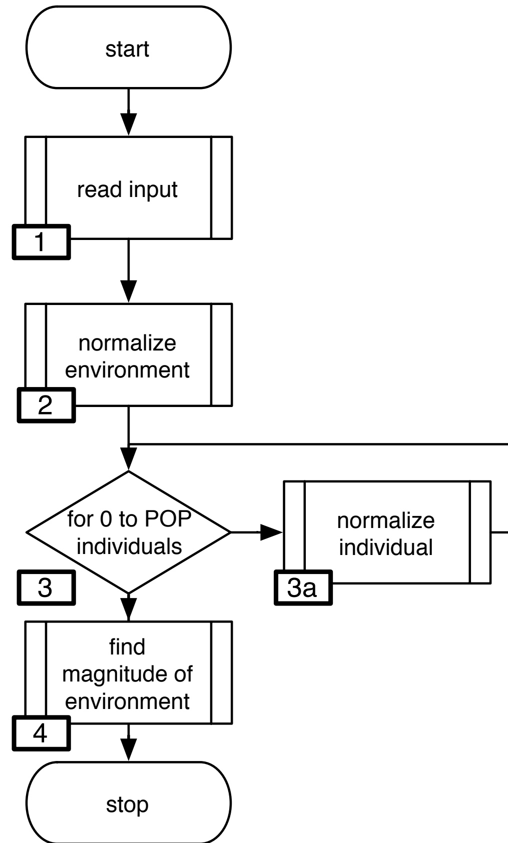

Figure 4.2: Initialization

parents (6).

4.1.2 Initialization

The initialization routine (figure 4.2) gets initialization parameters from the user (1) then lays the groundwork for evolution. The user has a great deal of control over the algorithm. The user chooses population files and an environment file—these are .wav files. The user chooses population size (POP), number of samples per generation (SAMP), normalization amplitude (N), number of generations of evolution (G), biodiversity modifier (B), probability of mutation (P), range for number of genes mutated (Smin–Smax), range for degree of mutation (Dmin–Dmax), and whether or not the algorithm uses a fixed or random crossover

point. For each mutation-related variable, there is a different analogous variable for each mutation. We store all of this information in an input file so we will not have to manually enter it every time. There are multiple instances of each of the starred variables (P, SMIN, SMAX, DMIN, and DMAX), associated with the different mutation types—obviously each variable has a different name in the program, but here we will refer to them more generally here to conserve space.

Once the user defined variables have been read in, the program reads in the environment waveform (2). Since the environment might be shorter than SAMP, we fill the rest of the chromosome with silence. We also want to normalize the environment so that it and the population have the same peak amplitude (2c). Then, we need to initialize the population (3). Finally, since we'll need to know it every time we calculate fitness, we find the MAGNITUDE OF THE ENVIRONMENT (4).

We make several assumptions here about the data. First, we assume that the population files and the environment files are no longer than SAMP samples long. Second, we assume that the user has preprocessed the files to make sure they start and stop at zero. If the user fails to supply appropriate files, there will be discontinuities in the output.

4.1.3 Calculate Population's Fitness

We calculate the fitness of each of the POP individuals to create an array FITNESS. We then normalize the array so its values can be used as probabilities. For example, if we start with 5 individuals, each with a FITNESS of 0.5, we will end up with 5 individuals, each with a fitness of 0.2.

Calculate Individual Fitness

Our fitness function treats chromosomes and the environment as SAMP-dimensional vectors and finds the cosine of the angle between the chromosome and the environment. This will give us higher numbers for more similar waveforms.

First, we find the dot product of the individual and the environment. Then, we use the same function as we used when initializing the environment to find the MAGNITUDE OF THE INDIVIDUAL WAVEFORM. Finally, we divide the dot product of the individual and the environment by the MAGNITUDE OF THE INDIVIDUAL times the MAGNITUDE OF THE ENVIRONMENT.

4.1.4 Write Output

After we've calculated each generation, we mix all of the individual waveforms together and append them to our output file we loop through each individual and add its SAMPLES times its FITNESS to the output. Since we have mutations that can change amplitude, we need to make sure our output doesn't clip. Finally, we write the output.

Prevent Clipping

Since we're going to add every member of the population together on output, we need to make sure it doesn't clip (exceed an amplitude of 1) when we write it to a waveform.

4.1.5 Produce Next Generation

To reproduce (figure 4.3), for each offspring (1), we select parents (2), splice the first part of one parent to the second part of the other parent (3), then apply mutations to the offspring (4). Once the entire next generation has been generated, we move the offspring to the population slots so they can be parents for the next generation.

Select Parents

Both parents are selected by the same method, which is based on FITNESS. We don't want to select the same parent twice; the algorithm selects two distinct parents.

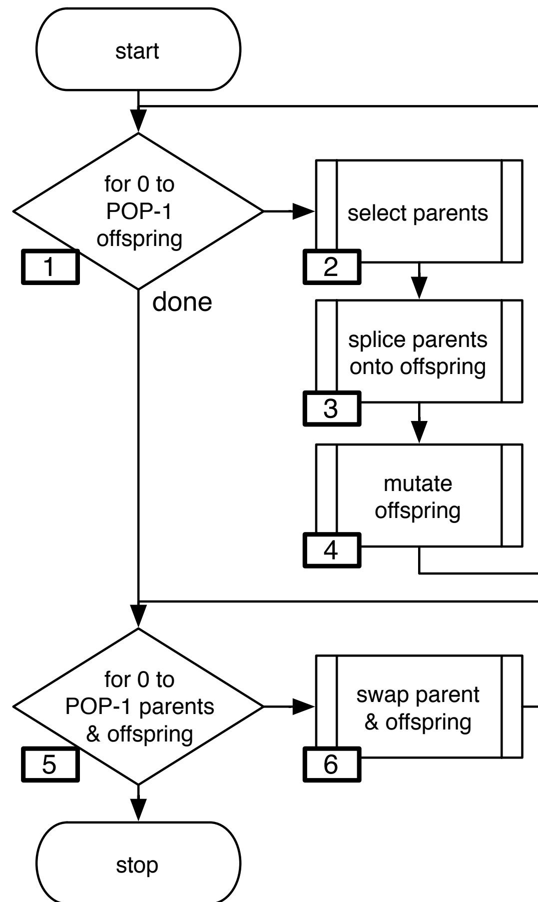

Figure 4.3: Reproduction

Select Individual Parent

Select an individual parent pseudorandomly out of the population with each individual's probability proportional to its fitness. To avoid using the same parent twice, the second parent is redrawn as necessary so that it is not the same as the first one.

4.1.6 Splice Parents

Sexual reproduction is essentially splicing the first part of one parent with the second part of the other. First, we check whether the user has specified a fixed or a random CROSSOVER POINT. If the CROSSOVER POINT is random, we set it to a random number between 0 and 1; otherwise, we set it to 0.5. Then, we convert the CROSSOVER POINT from a proportion of the chromosome to the actual number of samples by multiplying it by the number of samples. We do not want to splice where it will create a discontinuity in the resultant waveform, so we move the CROSSOVER POINT for the first parent to the next zero crossing and the CROSSOVER POINT for the second parent to the previous zero crossing. Then, we make sure our offspring won't exceed our chromosome length. If it would, we move the second CROSSOVER POINT back to the previous zero crossing and check the resultant length again. We continue this process until the offspring chromosome will be an appropriate length. Then, we copy the first parent from its start up through its CROSSOVER POINT and the second parent from its CROSSOVER POINT to its end. Finally, if the offspring is too short, we fill the chromosome with silence.

4.1.7 Mutate Offspring

To mutate the offspring we step through each type of mutation and check whether or not a random number between 0 and 1 is less than the MUTATION PROBABILITY. If it is, we apply the mutation; otherwise, we don't.

Find Segment to Mutate

For each mutation, the region the mutation applies to is chosen as follows. We set the beginning point of the mutation to a random integer between 0 and the difference between the CHROMOSOME LENGTH and the MAXIMUM MUTATION LENGTH for that mutation. Then, we move the beginning point to the NEXT ZERO CROSSING. We set the end of the mutation to the MUTATION STARTPOINT plus the MUTATION LENGTH—a random number between that mutation's MINIMUM MUTATION LENGTH and MAXIMUM MUTATION LENGTH.

4.1.8 Overwriting Duplication Mutation

We have fixed-length chromosomes, so we want to use a duplication mutation that overwrites samples as it duplicates rather than one that lengthens the entire chromosome. First, we find the region of the chromosome to which we'll apply the mutation Then, we set the NUMBER OF DUPLICATIONS to a random integer between the MINIMUM and MAXIMUM NUMBER OF POSSIBLE DUPLICATIONS. Then, we copy the portion of chromosome to be mutated into a buffer. Then, we loop through each duplication, copying the buffer into the next segment of chromosome. Once we're done duplicating, we need to find the NEXT ZERO CROSSING in the waveform after the duplicated segments. Starting at that zero crossing, we copy the rest of the waveform. Finally, we fill the end of the chromosome with zeros to avoid having a discontinuity at the end of the chromosome.

4.1.9 Swap Mutation

The swap mutation swaps two segments of chromosome, so we select two segments of chromosome for mutation. We make sure the two segments do not overlap; if they do, we move the ENDPOINT of the first segment so it maches the STARTPOINT of the second segment.

Then, we fill three buffers with our two designated segments and any samples in between them. We copy the first buffer back into the chromosome so

that it ends where the second segment ended. We copy the second buffer starting where the first samples was copied from. Finally, we copy the third buffer between the two.

4.1.10 Reversal Mutation

To reverse a portion of the chromosome, we start by selecting a segment of chromosome. The segment is copied into a buffer. Finally, we copy it into the chromosome backwards.

4.1.11 Amplify Mutation

To amplify, we choose a segment of chromosome to mutate, then select an AMPLIFIER by choosing a random number between the MINIMUM and MAXIMUM possible amplifications. Finally, we loop through the affected samples, multiplying the samples by the AMPLIFIER.

4.1.12 Exponentiate Mutation

Exponentiate works the same as amplification, except that instead of multiplying the samples by some constant, we raise the samples to a POWER.

4.2 An Algorithm for Evolving Variable Length, Time Domain Waveforms in a World With Multiple, Changing, Island Environments

The variable-length chromosome algorithm is based on the fixed-length algorithm (§4.1). It has many of the same building blocks, but changed slightly to deal with time (which we had been able to ignore in the fixed-length algorithm). There are several new mutation functions that take advantage of variable-length chromosomes (§4.2.5–4.2.7). The biggest structural change is in the introduction of

multiple, changing environments. This requires changes throughout the algorithm, as well as new functions to deal with the changing world (§4.2.8).

4.2.1 The Main Program

Individuals live in one of a fixed number of locations, each with its own specified environment. As before, the main program (figure 4.4) starts by initializing (1). In the fixed-length version, we specified length of evolution by number of generations. This doesn't make sense when we don't have discrete generations. Instead, we keep track of ELAPSED TIME, and evolve until we reach some user-specified TIME (T). Then, we make sure all waveforms left in the population are written to output (10).

Each individual has a lifetime equal to its length in samples and is replaced when it expires. Since their lifetimes vary individually, the times at which individuals are replaced are staggered. At each iteration the NEXT INDIVIDUAL TO END and how many SAMPLES ARE LEFT before it ends (3). The NEXT INDIVIDUAL TO END reproduces (4), the world is updated (6), and we write the next segment of output (6)—through the END of the NEXT INDIVIDUAL TO END. Finally, we do our temporal bookkeeping: the number of SAMPLES LEFT before the NEXT INDIVIDUAL ENDS is added to each individual's INDEX (7) and converted into the appropriate units and added to ELAPSED TIME (8).

4.2.2 Initialize

The initialization function starts by reading input files. We initialize each location and each individual. Individuals are assigned to a random location, and each individual's INDEX is set to 0. Both individual and environment waveforms are normalized using the same function as we used for the fixed-length version of the algorithm. The individual's INDEX keeps track of the point in the individual's waveform that corresponds to the CURRENT TIME in the world. Finally, the world's ELAPSED TIME is set to 0.

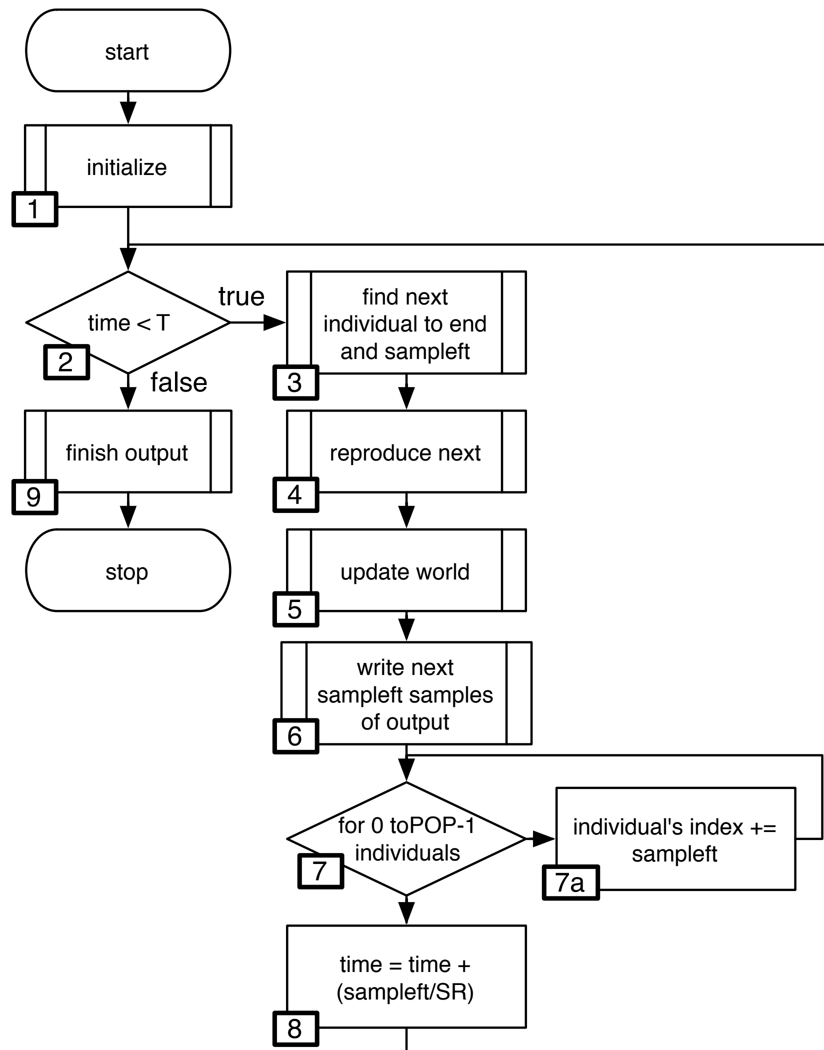

Figure 4.4: The main flowchart for evolving variable-length waveforms in a world with multiple, changing, island environments

Read Configuration File and Waveforms

The configuration file contains all user-defined variables and filenames of environment and population waveforms. First, we find out how many LOCATIONS (LOC) our world will have and open OUTPUT files for each location. Then, we read in user-defined variables: NORMALIZATION AMPLITUDE (N), BIODIVERSITY MODIFIER (B), CROSSOVER TYPE (C), PARENT TYPE (PAR), MINIMUM AMOUNT OF EVOLUTIONARY TIME (T), SAMPLE RATE (SR), SEEDS for the random number generator (S1–S3), PROBABILITY OF GRADUAL ENVIRONMENTAL CHANGE (PENV), PROBABILITY OF CATASTROPHIC ENVIRONMENTAL CHANGE (PCAT), INCREMENT OF ENVIRONMENTAL CHANGE (EINC), and PROBABILITY OF MIGRATION (PMIG). Then, we read in MUTATION PROBABILITIES for each mutation: PROBABILITY OF MUTATION (P), MINIMUM and MAXIMUM MUTATION SIZES (SMIN, SMAX), MINIMUM and MAXIMUM DEGREES OF MUTATION (DMIN, DMAX), and, for each variable, the PROBABILITY THAT IT WILL CHANGE (PCHANGE) and the AMOUNT OF VARIATION that variable can have if it changes (VPM). Finally, we read in our input files. We find out the MAXIMUM NUMBER OF SAMPLES PER WAVEFORM (SAMP) and how many environmental files there are so we can allocate memory. Then, we read in the environment files, the number of population files (POP), and the population files themselves.

Initialize Location

We want each location to have its own sound. A location's sound comes from three sources: its MUTATION PROBABILITIES (which influence the characteristic mutation sounds that are present in the location), its environmental waveform (which provides evolutionary direction), and its subpopulation (which provides the sonic materials the algorithm can act on). We randomly generate variables for each location based on the variables provided by the user. First, we initialize mutation variables: Each mutation's PROBABILITY OF OCCURRING is a random number between 0 and its USER-DEFINED PROBABILITY. The other mutation variables

(Smin, Smax, Dmin, Dmax) are random numbers between their user-defined minimum and maximum values. If the minimum is greater than the maximum we swap them. As described in §3.2.2, each locaton's environment will be defined by two waveforms that are mixed together to produce the environment. Each time the environment undergoes gradual change, the two waveforms are interpolated. We randomly choose an initial waveform (9) and a target waveform (10) for the location. We copy the initial waveform into the environment and set the degree of interpolation to 0 (12). We set the environment's length to the length of the longer of the two constituent waveforms (13) then find the environment's magnitude (14) using the magnitude function we used for fixed-length chromosomes.

4.2.3 Find Next Individual to End

When an individual ends, it leaves the population. It ends when its last sample has been written to output and it is analogous to biological death. Finding the next individual to end is simple, but the function is the lynchpin of the variable-length chromosome version of the algorithm. Instead of reproducing, writing output, and performing any bookkeeping the algorithm might require at the end of every generation, we do these things each time an individual ends.

4.2.4 Reproduction

The primary difference between reproduction in the variable-length chromosome algorithm (figure 4.5) and reproduction in the fixed-length chromosome algorithm (§4.1.5) is that only one individual at a time can reproduce. We select parents (1), splice them into an offspring (2), mutate the offspring (3), then replace the parent with its offspring (4). We set the individual's index to 0 so it will start playing from its beginning. Finally, we calculate the individual's fitness (5) using the same individual fitness function we used in the previous algorithm. Fitness relative to the environment will be calculated only once: We don't want to waste

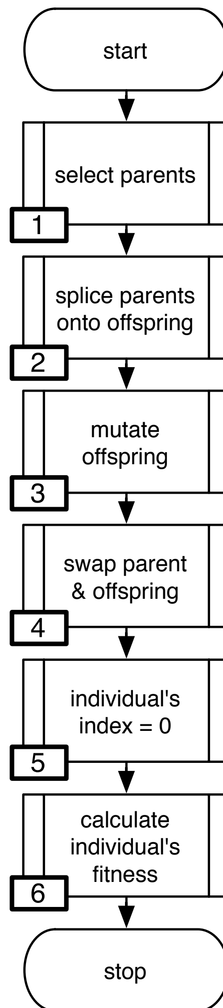

Figure 4.5: Reproduction

processing time by calculating FITNESS each time the environment changes, and we don't want discontinuities in the output when the environment changes.

The details of selecting and splicing parents are different, but the mutation function is essentially the same as the fixed-length mutation function. The mutation function loops through the possible mutations, checking which mutations are applied to the offspring. There are three new mutation functions that are introduced with variable-length chromosomes: dropping (§4.2.5), lengthening duplication (§4.2.6), and changing length (§4.2.7).

Select Parents

This version of the algorithm allows the user to choose between two methods for selecting parents.[2] We can use the same method as we used for fixed-length chromosomes, chosing both parents based on fitness, or we can designate the individual being replaced as the first parent and select the second parent based on fitness. If we use the latter method for chosing parents, we can calculate fitness of the second parent based on self-similarity.[3] To do this, we use the same fitness function as before, but replace the environmental waveform with the first parent's waveform. As before, we make it impossible for the first parent to be selected as the second parent. We normalize FITNESS, process it, and find the second parent, all using the same functions that we used for fixed-length chromosomes.

Splice Parents

The primary difference in splicing parents with variable-length rather than fixed-length chromosomes is how the algorithm deals with random crossover points. With fixed-length chromosomes, both parents have the same CROSSOVER POINT, regardless of whether it is FIXED or RANDOM. Since we no longer need to

[2]Unlike the three new mutation functions, this isn't a possibility introduced by variable-length chromosomes. Rather, the first function was simplified as much as possible, and we are taking this algorithm as an opportunity to introduce more possibilities.

[3]In practice, I often use fitness functions that combine self-similarity and environmental fitness.

conserve length, random crossover points can be different for each parent.

We start by checking whether the user has specified a FIXED or a RANDOM CROSSOVER POINT. If it is RANDOM, we select a random CROSSOVER POINT for each parent; otherwise, the CROSSOVER POINT is the halfway point for each parent. We map the crossover points onto the LENGTHS of the parents and nudge the crossover points onto zero crossings using the same next and previous zero crossing functions we used before. Then, we copy the first part of the first parent and the second part of the second parent into the offspring. Finally, we find the LENGTH of the offspring.

4.2.5 Drop Mutation

The drop mutation drops a segment of neighboring genes. The segment of genes to be dropped is selected. The remainder of the waveform is moved forward, overwriting the dropped portion of waveform. Finally, the individual's new LENGTH is calculated.

4.2.6 Lengthening Duplication Mutation

The lengthening duplication (figure 4.6) pushes the end of the waveform out as it duplicates rather than overwriting. First, we find a segment of waveform to duplicate, then we randomly select a NUMBER OF DUPLICATIONS. We find the offset that will be introduced by this NUMBER OF DUPLICATIONS and check that the resultant offspring won't exceed our MAXIMUM WAVEFORM LENGTH. If it does, we decrease the NUMBER OF DUPLICATIONS and make sure that still lets us have the minimum NUMBER OF DUPLICATIONS. If decreasing the NUMBER OF DUPLICATIONS gives us a legal NUMBER OF DUPLICATIONS, we double check the MAXIMUM LENGTH again. Otherwise, we find a a new mutation segment and try again.[4]

[4]We might throw in another step to prevent an infinite loop if the user has chosen unreasonable variables. This step has been left out for the sake of simplicity.

Once we have an appropriate segment for mutation, we copy it into a buffer. Then, we copy the end of the waveform so that it starts where our duplications will end. For each duplication, we copy the buffer into the mutation, updating the OFFSET after each duplication. Finally, we update the individual's LENGTH, incrementing it by the OFFSET.

4.2.7 Change Length Mutation

To lengthen a portion of waveform (figure 4.6), we start by selecting a segment to mutate (1). We randomly select a LENGTHENING MULTIPLIER (2), then figure out how much that will OFFSET the rest of the waveform (3). We copy the portion of the waveform that we're going to resample into a buffer (4) then move the rest of the waveform to account for the OFFSET (5). We copy three extra samples into the buffer, since we'll need them for interpolation. We also need indices to step through the buffer and the offspring while we interpolate (6): IMULT will be the fractional part of the INDEX, which increments by the MULTIPLIER, while I will be the integer part. We use four-point interpolation to resample the buffer as we copy it into the waveform (8). Since we're tracking the fractional and integer parts of our INDEX, we increment them by adding MULT to IMULT (9) and checking if the result is greater than one (9a). If it is, we add its integer part to I (9b) and keep the fractional part as IMULT (9c). Finally, we calculate the individual's new LENGTH (10).

4.2.8 Changing Island World

There are three components to the changing world: catastrophic change, gradual change, and migration. The changing world function is called each time an individual reproduces. It checks whether or not there is catastrophic environmental change in the individual's location; if there is, it reinitializes the location. If not, it checks if that location undergoes gradual environmental change and, if appropriate, implements that change. Finally, it checks whether or not the next individual—at

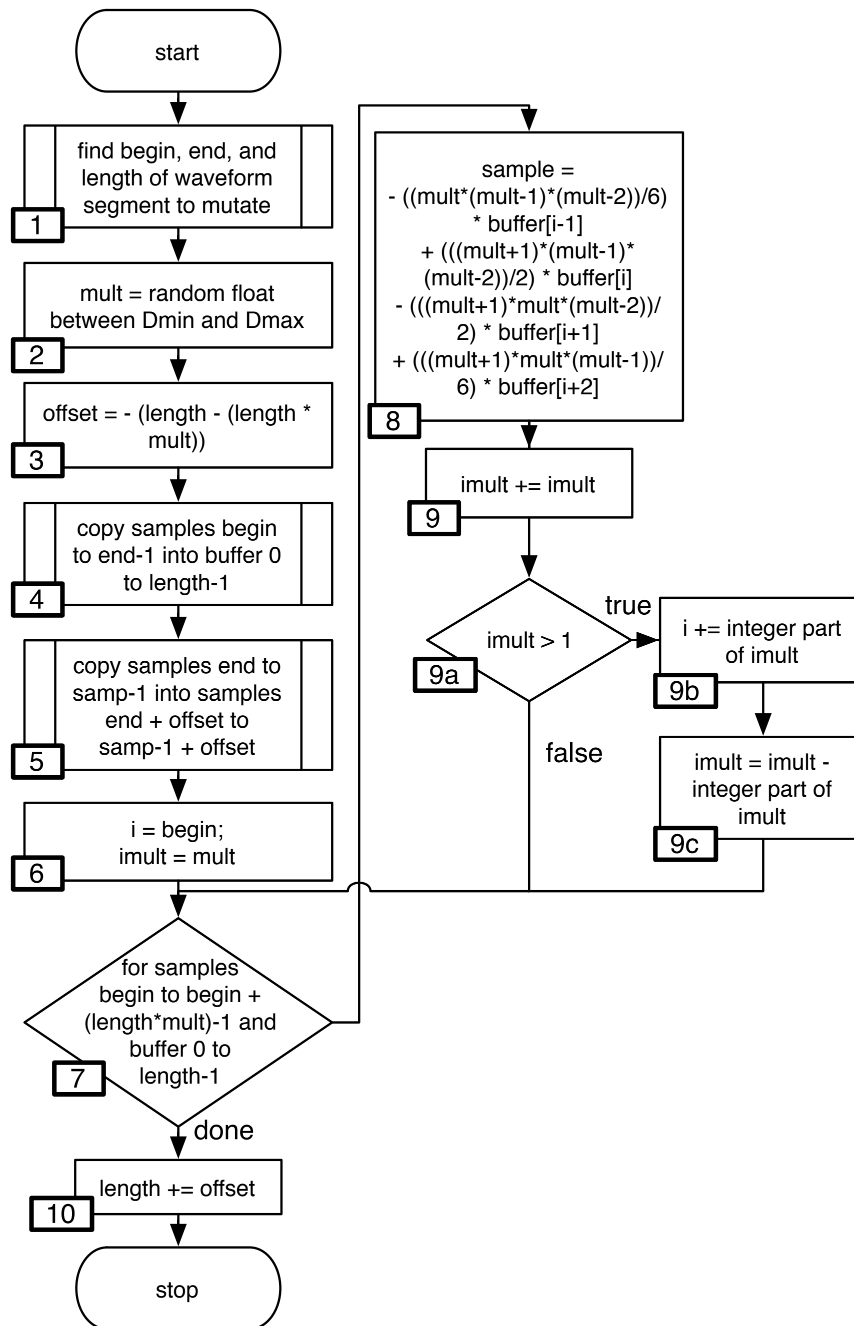

Figure 4.6: Change length mutation

this point, that individual's offspring has taken its place—will remain in the same location or spend its lifetime migrating.

Gradual Change

Gradual change makes each mutation variable take another step in a random walk and interpolates the environmental waveforms a little further.

To change our variables, we start by moving through each variable of each mutation and test whether or not that variable will change. Variables have a random number between -VPM and VPM added to them. Then, we make sure our variable is within the user-defined VARIABLE RANGE. If the variable is outside its designated range, we change the variable to its limit.

To interpolate waveforms, we start by increasing our DEGREE OF INTERPOLATION by our INTERPOLATION INCREMENT. Then, we check whether or our degree of interpolation has reached one. If it has, we need to interpolate to another waveform. So, we move our target waveform into the initial waveform position and select a new target waveform. We make our new initial waveform our new environment and set the DEGREE OF INTERPOLATION to zero. If our DEGREE OF INTERPOLATION is less than one, we mix the two waveforms to make our environment. We weight the target waveform by the DEGREE OF INTERPOLATION and the initial waveform by one minus the degree of interpolation. Finally, we find the MAGNITUDE of the new environment.

Migrate

We need to keep track of which individuals are migrating, where they are migrating to, and where they are migrating from. We start by setting the individual to MIGRATING and choosing the individual's destination. We don't want the individual to migrate to its current location, so we randomly choose from one fewer than the NUMBER OF LOCATIONS. The individual will be considered part of the DESTINATION LOCATION for reproductive purposes. Finally, we need

to set variables that will enable us to write the migrating files to output: it will need to fade out of its starting location and fade into its destination. We set the individual's fade to zero and set its fade increment to one divided by the individual's length.

4.2.9 Output

Write Segment

Every time an individual ends, we write the segment of OUTPUT between the last time an individual ended and the current time. We write each location to its own output file. We start by initializing the segment of output to zero. Then, we mix the waveforms at the location. For each individual, we check whether or not it is MIGRATING. If it is not MIGRATING, we check whether or not it is in the LOCATION; if it is, we write the individual, weighted by its FITNESS, to OUTPUT. If the individual is MIGRATING, we see if it is migrating to or from the LOCATION. In either case, we write the migrating individual to OUTPUT, weighted by both its FITNESS and its FADE. Finally, we prevent clipping using a function similar to the one we used in the fixed-length version of the algorithm, but operating between zero crossings instead of generation boundaries. Finally, we write the segment to output.

Write End

Since the individuals in the population don't end simultaneously, we need to write the remainder of any individuals left when the evolution stops. To do this, we loop through the population until all individuals have finished. We swap the next individual—which will have just finished playing—with the last individual and decrease the size of the population by one, so that the finished individual is no longer counted as part of the population. Then, we find the NEXT INDIVIDUAL TO END and write the intervening segment of OUTPUT.

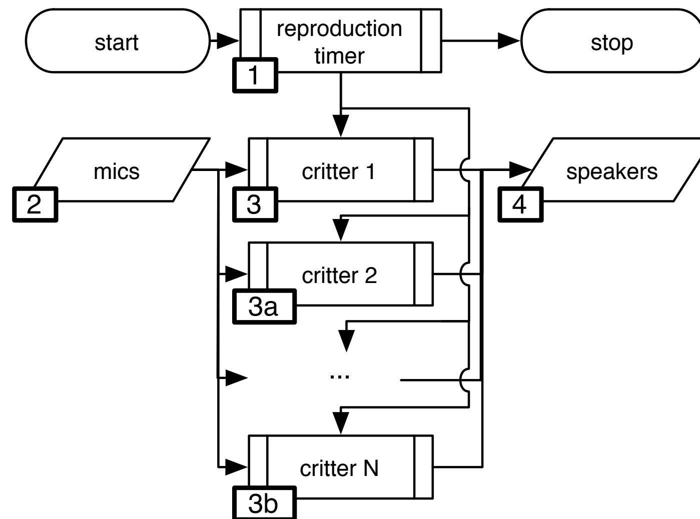

Figure 4.7: Evolving variable-length, time-domain waveform in a real-time ring-world

4.3 An Algorithm for Evolving Variable Length, Time Domain Waveforms in a Real Time Ring World

4.3.1 The Main Program

I have implemented the realtime version of the program in Pure Data, whereas the previous versions have been implemented in ansi c. Pure Data has its own scheduler, so this implementation doesn't deal with time in the same way as the previous two. In the first two versions of the program (§4.1–§4.2), it was not a problem if it took us several hours to generate a few minutes of music. Here, it is an issue. Instead of using idealized, independent agents, we need to schedule individuals to reproduce serially, since reproduction is computationally intensive (1). They can otherwise operate in parallel (3). Each individual takes input from microphones (2) and sends output to speakers (4).

This version of the program implements a ring-world topology, as described in §3.2.2. The world is defined as a ring. The ring is mapped onto speakers and microphones that occupy fixed locations in a performance space. Individuals

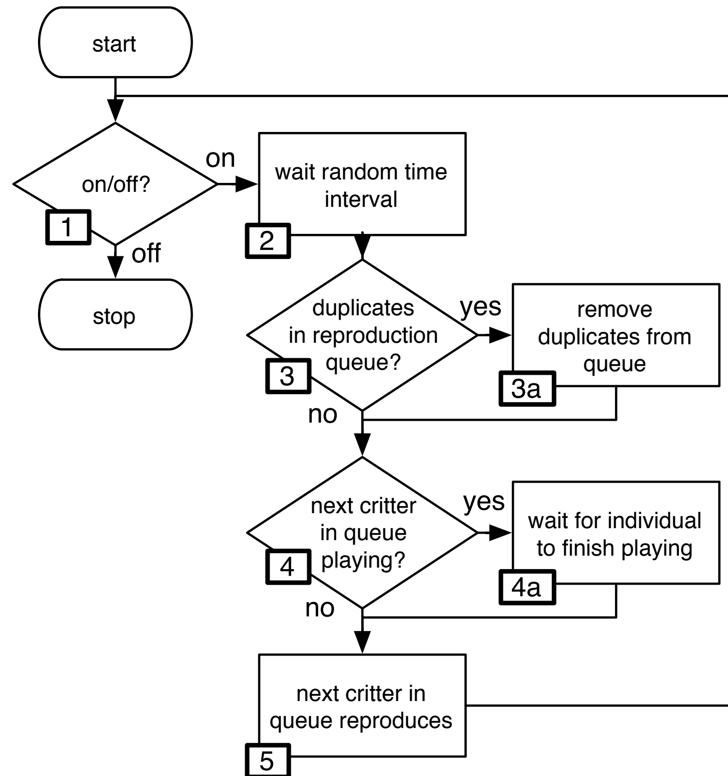

Figure 4.8: Reproduction timer

occupy locations, specified by angle, on the ring. Individuals can move over the course of their lifespan. Individuals' non-sonic behaviors are defined by a second chromosome (§4.3.4). Microphones define the environment in real time: each time an individual is produced, its fitness is determined based on the environmental waveform at its location that begins at its inception and is of the same length as it.

4.3.2 Reproduction Timer

We want to make sure each individual plays at least once before being replaced by its offspring. So, we add individuals to the end of a queue each time they finish playing (§4.3.4), and reproduce from the beginning of the queue. The reproduction timer operates when the program is running (1). It waits some ran-

dom amount of time (2) then looks at the queue. First, we need to make sure there are no duplicates of the next individual in the queue—if an individual reproduces, we don't want residual copies of that individual to cause the individual's offspring to reproduce before it's had a chance to play (3). If the next individual in the queue is playing, we wait for it to finish playing (4). Finally, we send a message to the individual, having it reproduce (5).

4.3.3 Input and Output Weights

The ring world is represented by input and output weights. Each individual will have a LOCATION on the ring. Both its INPUT, which will be used to calculate FITNESS, and its OUTPUT, are weighted to represent this LOCATION. The two IOs on either side of the individual receive weights that sum to one and that depend proportionally on their two distances from the individual, and all other IOs receive a weight of zero.

4.3.4 Individuals

With the exception of the reproduction scheduler, the individuals are independent agents. They are initialized when the program starts (1), and thereafter inhabit their world, mating and reproducing, for as long as the evolutionary process continues (2). There are two basic loops that produce an individual's behavior. Over the long term, the individual calculates its fitness (3) then waits until it is told to reproduce (4). When it is told to reproduce (4), it finds a mate (8), splices with the mate (9), and mutates (10). Since individuals now have two chromosomes (waveform and behavior), both chromosomes splice and mutate. The behavior chromosome mutations are conventional mutations (§1.2.3, §1.3.3) that add or subtract a tiny amount rather than the waveform-specific mutations described in chapter III. The loop will repeat and the offspring's FITNESS will be calculated (3).

Over the short term, the individual waits some amount of time that is

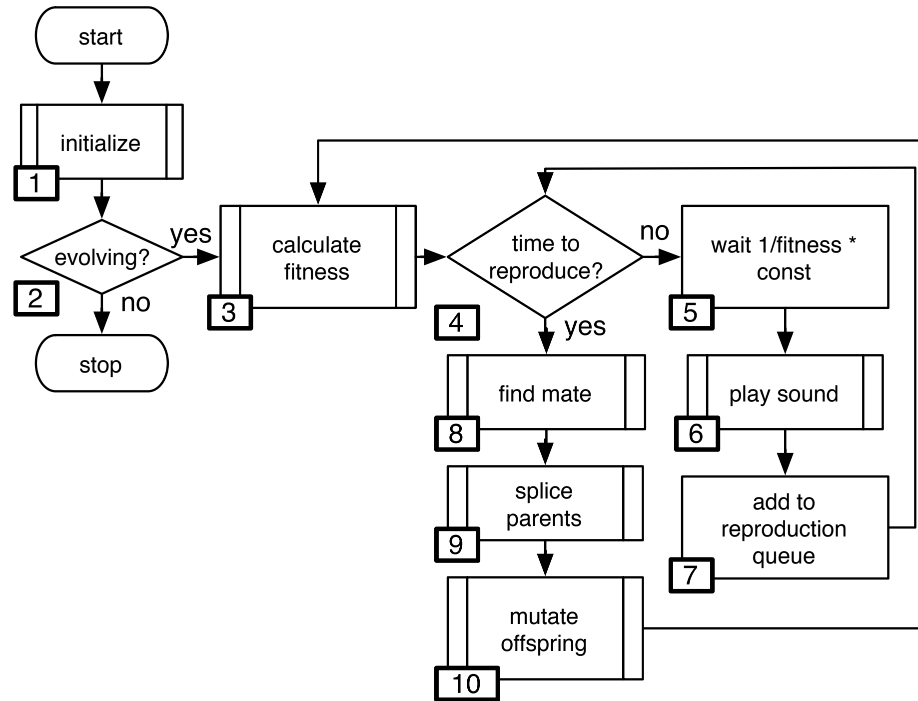

Figure 4.9: Individual

inversely proportional to its FITNESS—that is, a shorter amount of time if it is fitter and a longer amount of time if it is less fit (5)—then plays (6) and adds itself to the reproduction queue (7). The individual will play periodically until it reproduces (4).

Initialize Individual

Individuals have two chromosomes. The first is the waveform, which is read in by the initialization function (1). The second defines the individual's behavior in the world. Each individual has a RANGE OF MATE SEARCH (2), a LOCATION (3), a MOVEMENT DURATION (4), a PROBABILITY OF STANDING instead of moving (5), a MAXIMUM AMOUNT OF CLOCKWISE MOVEMENT (6), and a MAXIMUM AMOUNT OF ANTI-CLOCKWISE MOVEMENT (7). All variables in the second chromosome are randomly assigned when the initial population is created. Except for LOCATION (4), which is between 0 and 359 degrees, all of the variables

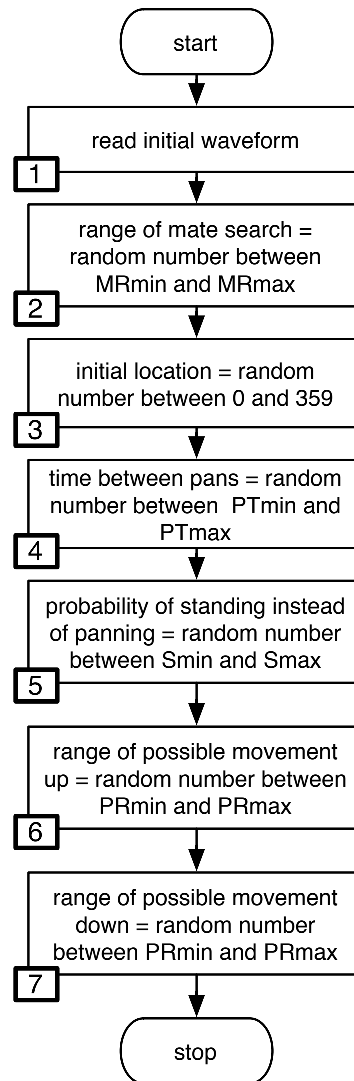

Figure 4.10: Initialize individual

fall in some user-defined range.

Fitness

To calculate FITNESS, we have to find the individual's input weights (1) and mix the inputs to produce a single channel of input that corresponds to the individual's location (2). We treat the segment of incoming audio beginning when we start testing fitness and having the same duration as the individual's waveform as the environmental waveform. Then, we can use the same fitness function as we used in the first two algorithms (3).

Find Mate

Instead of having discrete islands, we have a continuous world. The individual's behavior chromosome specifies its LOCATION and the RANGE in which it is willing to look for a mate. The individual's mate is chosen randomply from among all individuals within RANGE degrees of the individual's own LOCATION, with each such individual's probability proportional to its own fitness

4.3.5 Output

Since Pure Data handles time for us, we don't have to explicitly mix the OUTPUTS. Rather, each individual can play it's waveform and the details will be taken care of for us. When we play an individual's waveform, we find WEIGHTS for its outputs (1). We play the waveform (2), and, for each output, multiply the waveform by its weight for that output (3a).

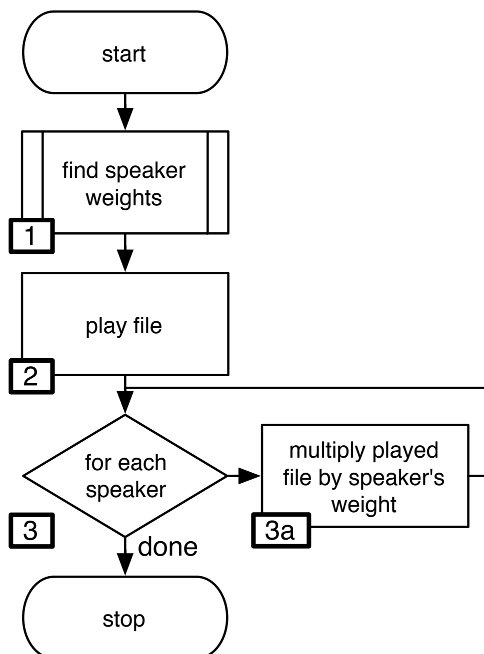

Figure 4.11: Output

V

Results and Conclusions

5.1 Systematic Tests

I have implemented a very simple version of the algorithm for systematic tests of the framework. This algorithm, described in §4.1, has fixed-length chromosomes and a single, unchanging environment. I will present the results of running the algorithm with all of the mutation functions available to it, with fixed and random crossover points, and with different random seeds for the same initial conditions. Some dimensions, such as degree of mutation and the biodiversity multiplier, are not dealt with here, as they are not unique to this framework and their behavior is well understood.[1] Other dimensions relating to the complex worlds developed for musical compositions are not dealt with here, since their components can't be meaningfully isolated (see §5.2).

5.1.1 Test Sets

To better isolate the behavior of different components of the algorithm, I used three test populations: noise, sinusoids, and real-world sounds. All samples

[1]Higher degree of mutation and higher mutation probability lead to faster evolution, but they tend to be more readily attracted by local minima. Increasing biodiversity slows evolution and decreases the probability that the population will fall into a local minimum. Typically, users of genetic algorithms choose initial values then run the algorithm and observe its behavior. If the algorithm evolves too slowly or falls into local minima too easily, these values can be changed until the algorithm reliably performs as desired.

were normalized to have a maximum amplitude of one. The chromosome length was one second. There were 101 individuals in the noise population, 121 samples in the sinusoid population, and 50 samples in the real-world population.

- The noise population is the most neutral test population. All waveforms are constant-amplitude white noise. We do not expect the population to change dramatically; all we can evolve from noise to noise. But, the quality of the evolved noise tells us something about the algorithm's behavior. Anything that changes the quality of the noise—i.e. its amplitude envelope, pitch characteristics, etc—will be a result of evolution and its representation.

- The sinusoid population is comprised of every audible, equal-tempered sinusoid.[2] As with the noise population, the sinusoid waveforms have constant amplitude. We expect to hear evolutionaryeffects on both pitch and loudness. Although none of the mutations available to this algorithm can change pitch, sinusoids provide evidence about which parents have been selected and how they have been spliced together.

- The real-world sounds are sampled from a composition for percussion and voice. This makes them a mix of harmonic, inharmonic, and noisy sounds of varying amplitude. We expect this test set to shed light on how the algorithm will behave with a varied initial population.

There are four environments: one synthesized sound and three real-world sounds. The synthesized sound is a constant-amplitude glissando that covers the same pitch range as the sinusoid population (gliss.wav). The real-world sounds are a struck pot (pot.wav), a cricket (cricket.wav), and footsteps (footsteps.wav). The struck pot is inharmonic and starts with a high amplitude and dies out over the course of a second. The cricket is a short, pitched, tremolo—a little less than half a second long—which will also serve to test how the algorithm deals with

[2] With A equal to 440Hz and assuming that 20Hz–20000Hz is the audible range

environments that are shorter than the generation. The footsteps are a couple of noisy impulses over a bed of background noise.

5.1.2 Results

All test sets were run with 10 different seeds for 300 generations. Representative examples of these runs are discussed below. The appendix contains the configuration files for all example files. For ease of listening, all examples have been edited to present the first 10 generations of evolution, generations , and generations 290–300. There is a second of silence between the each segment of output.

5.1.3 Seed

This algorithm is stochastic. We can make generalizations about how it will behave under particular conditions, but nothing is guaranteed. Genetic algorithms explore high-dimensional spaces, rife with local minima. Sometimes, several runs of the algorithm with same starting conditions will result in more-or-less interchangeable results; sometimes the results will be wildly different. They will generally share features, such as the characteristic sounds from the mutations available in the run and the sound of evolution itself, but on a more immediate level, they will be different.[3] I will present several runs from the same variables with different seeds here. I will discuss other examples as they arise below. Examples real-walking-all-c12.wav, real-walking-all-c13.wav, and real-walking-all-c18.wav have identical populations, environments, and variables, but different seeds.

Example real-walking-all-c12.wav starts the same way as the others, a mixture of recognizably real vocal and percussive sounds. Within a few generations the population develops an amplitude envelope that is loud in the first half and quiet in the second half. By the middle, it seems to mix a sung pitch with percussion. But, over the course of the 10 generations presented, the sung pitch remains static relative to the percussion, which increases in density and intensity.

[3] To achieve musical results, it is helpful to run the algorithm multiple times with different seeds. In 5.2.1, I discuss two pieces chosen from 15 runs with identical starting conditions.

The density increase is most likely from mutations (such as overwriting duplication) offsetting sections of similar individuals. Since the mutations do not line up exactly, we hear several similar sounds instead of one loud sound. This run of evolution seems to have evolved towards the noisy, background timbre of the environmental sound, ignoring its amplitude envelope.

The difference between real-walking-all-c12.wav and real-...-c13.wav is already apparent after a few generations of evolution. In real-walking-all-c13.wav, different sounds dominate the population. Instead of a relatively constant vocal pitch and fairly metallic percussive sounds, several vocal sounds are prominent in the population. The most audible of these is a glissando. There is a metallic percussion sound, but it does not rise to prominence. By the middle portion of the run, we start to hear the characteristic sound of several mutations. A repetitive pulse (overwriting duplication) dominates the first half of the waveforms and a gritty, distorted (exponentiation) voice dominates the second half of the waveforms. By the end of 300 generations of evolution, we have something with amplitude and timbre characteristics that are similar to the environment: background noise with a smattering of amplitude peaks. One of the peaks retains the clear sonic character of an original vocal sound.

Example real-walking-all-c18.wav starts like the others. Early on, different sounds from the original population rise to prominence. By the middle portion, its amplitude envelope more closely resembles the original amplitude envelope, with quiet background noise and footstep-like peaks that correspond temporally to the footsteps in the environment. However, by the end of 300 generations it has over evolved, losing its footstep-like peaks. Timbrally, it has a different sound from the other two runs, but the characteristic sound of the overwriting duplication remains.

5.1.4 Crossover Point Without Mutation

The algorithm lets the user choose a fixed or a random crossover point. Without mutations, with a perfectly fixed crossover point,[4] the population should converge to a population of identical individuals that are made from the first part of an initial individual and the second part of an initial individual.[5] The same population, evolving under otherwise identical conditions, but with a random crossover point, should converge more slowly. It will almost never converge to identical individuals in the time frame of a typical composition. The individuals in the population will be comprised of excerpts of many initial individuals.

Noise

Consider the population of constant-amplitude noise, with one example from each environment for each type of crossover point. Four examples have a fixed crossover point: noise-gliss-none-c1-6.wav evolving in the gliss.wav environment, noise-cricket-none-c1-3.wav evolving in the cricket.wav environment, noise-pot-none-c1-0.wav evolving in the pot.wav environment, and noise-walking-none-c1-0.wav evolving in the footsteps.wav environment. Four examples have a random crossover point: noise-gliss-none-c0-4.wav evolving in the gliss.wav environment, noise-cricket-none-c0-7.wav evolving in thecricket.wav environment, noise-pot-none-c0-0.wav evolving in the pot.wav environment, and noise-walking-none-c0-0.wav evolving in the footsteps.wav environment.

We have conflicting expectations. On the one hand, we predicted above that we would hear a difference between the fixed-crossover and random-crossover cases. On the other hand, with a population of noise, all individuals are virtually identical. Without mutation, we wouldn't expect to perceive a difference between the parts of various individuals, so we shouldn't perceive a difference at all.

[4]The fixed crossover point isn't completely fixed, since it's nudged to correspond to the nearest zero crossing.

[5]Since the waveforms are hermaphroditic, there's no guarantee that they won't be two halves of the same individual

The noise population behaves as expected with respect to crossover; it does not behave in the way we would expect mutation-less noise to behave. With a fixed crossover point, we hear two distinct halves for each generation (noise-gliss-none-c1-6.wav, noise-cricket-none-c1-3.wav, noise-pot-none-c1-0.wav, noise-walking-none-c1-0.wav). With a random crossover point, each generation has a smoother amplitude envelope (noise-gliss-none-c0-4.wav, noise-cricket-none-c0-7.wav, noise-pot-none-c0-0.wav, noise-walking-none-c0-0.wav). While the fixed-crossover examples sound like noise after 300 generations of evolution, the random-crossover examples sound filtered.

So, why do the populations of constant-amplitude noise not evolve to populations of constant-amplitude noise? In all of the examples, the first generation *does* have uniform amplitude. The fixed-crossover example noise-cricket-none-c1-3.wav is a particularly telling example. Its first generation is uniform amplitude noise; its last ten generations are uniform amplitude noise. But, in the middle it is loud for the first half of each generation and quiet for the second half.

Recall that the algorithm outputs all individuals at once. They are weighted by their fitness and mixed together. Also, with fixed-length chromosomes, if the individuals are too short they are padded with silence. Furthermore, if the individuals are too long, they are truncated and, if they don't end with a zero crossing, their last few samples are overwritten with silence from the last zero crossing to the end. This is essentially an implicit mutation that results from having fixed-length chromosomes. All individuals have something in their first half, since only the last half is affected by crossover-related shortening. Some individuals eventually evolve a silent second half. When they are all mixed together, we hear this as two distinct volume levels.

What has happened in noise-cricket-none-c1-3.wav, is that the population evolved through a period in which some individuals ended with silence and others were noise throughout. Then, it converged to individuals with noise throughout. noise-pot-none-c1-0.wav exhibits a similar trajectory. We can tell it hasn't quite

converged, because the second half of each generation is slightly quieter than the first half. This suggests that there are a few short individuals remaining in the population. Because this behavior results from fixed-length chromosomes, we wouldn't expect it to manifest in the variable-length algorithms; it does not manifest in any of the pieces produced with the variable-length versions of the algorithm.

The filtering exhibited by the random-crossover examples is explained by a loss of biodiversity, combined with slight temporal offsets that result from nudging segments of waveform to line up zero crossings. Eventually, we have multiple copies of the same waveform in the population, with different slight delays. Digital filtering is an emergent phenomena of the output representation.

Finally, it is worth noting that all examples are louder after 300 generations of evolution than they were initially. This is because the output is weighted by fitness. We expect a fitter population to be louder than an unfit population.

Sinusoids

Consider the population of constant-amplitude sinusoids, with one example from each environment for each type of crossover point. Four examples have a fixed crossover point: sin-gliss-none-c1-4.wav evolving in the gliss.wav environment, sin-cricket-none-c1-0.wav evolving in the cricket.wav environment, sin-pot-none-c1-0.wav evolving in the pot.wav environment, and sin-walking-none-c1-6.wav evolving in the footsteps.wav environment. Four examples have a random crossover point: sin-gliss-none-c0-3.wav evolving in the gliss.wav environment, sin-cricket-none-c0-3.wav evolving in the cricket.wav environment, sin-pot-none-c0-0.wav evolving in the pot.wav environment, and sin-walking-none-c0-0.wav evolving in the footsteps.wav environment.

With a fixed crossover point, we expect to hear the population evolve towards two-part individuals. With a random crossover point, we expect the population to evolve towards multi-part individuals. Since the initial population has 121 different pitches, we expect to hear this in terms of pitch. The fixed-crossover

examples should converge to two pitches; the random-crossover examples should converge to many pitches. Initially, we should hear a chord of 121 pitches. We should hear convergence as chords of decreasing complexity. As the population converges, we should less complex chords. Since we expect fixed-crossover evolution to converge more quickly, we expect monophonic, alternating pitches by the end of the examples. Since we expect random-crossover evolution to converge more slowly, we expect successions of simple chords in later generations. Finally, we expect to see the same fixed-length-chromosome shortening behavior that we saw in the noise examples.

Since the sinusoid population has more audibly different sonic building blocks to work with than the noise population has, we might expect to find behavior in the populations to reflect their environments. But, it is important to note that the fitness function considers phase, not just pitch. An individual with the same pitch as the environment, but completely out of phase will be less fit than an individual with a different pitch whose waveform occasionally corresponds to the environment. It would be naive to assume that sinusoids evolving with random crossover points in the gliss.wav environment with random crossover points would converge a sequence of ascending sinusoids.

The sinusoid population behaves as expected in all respects. The fixed-crossover samples start with one pitch and move to another halfway through. In most of these, the second parent is forced into silence by the final generation, but sin-walking-none-c0-0.wav retains some pitch from its second parent. The random-crossover samples become strings of pitches. They don't, however, fulfil the naive expectation that they will become ascending pitches. It would be surprising if they did.

The sinusoid population allows us to clearly hear biodiversity. With all of the examples, the initial population gives us a very complex chord: all members of the population have different pitches. With time, all of the examples become less complex chords. The fixed-crossover samples usually converge to identical

individuals before they reach the last 10 generations of their 300 generations of evolution. sin-cricket-none-c1-0.wav converges to two individuals. The random-crossover examples converge to a handful of individuals; their output is a succession of fairly simple chords by the end of 300 generations of evolution. The random-crossover examples converge .

Real-World Sounds

Consider one exemplar from each environment for each type of crossover point. We have four sounds with a fixed crossover point: real-gliss-none-c1-0.wav evolving in the gliss.wav environment, real-cricket-none-c1-4.wav evolving in the cricket.wav environment, real-pot-none-c1-2.wav evolving in the pot.wav environment, and real-walking-none-c1-8.wav evolving in the footsteps.wav environment. Four sounds have a random crossover point: real-gliss-none-c0-0.wav evolving in the gliss.wav environment, real-cricket-none-c0-4.wav evolving in the cricket.wav environment, real-pot-none-c0-2.wav evolving in the pot.wav environment, and real-walking-none-c0-3.wav evolving in the footsteps.wav environment.

It is difficult to tell, just by listening, which real-world examples have a fixed crossover point and which have a random crossover point. The fixed-crossover samples have more clarity—partly from loss of biodiversity and partly from containing material from fewer members of the initial population—but this doesn't confront the listener the way it does, for example, when two sinusoids are spliced together.

The patterns of loud beginnings and soft endings exhibited by the noise and sinusoid samples holds only for the random-crossover examples. Even then, it is much less pronounced. In general, the amplitude envelopes of real-world examples behave much more like one would expect based on the envelopes of the fitness functions. Real-world sounds evolving in the cricket environment have a fairly consistent amplitude relative to the same population with other environments. Sounds in the pot.wav environment tend to be louder in the beginning and quieter

towards the end. Sounds in the footsteps.wav environment tend to be fairly quiet, with peaks corresponding to the footsteps in the environment. These peaks don't always overlap exactly with the environmental peaks; for instance, real-walking-none-c1-1.wav has a much wider peak than real-walking-none-c1-8.wav. However, the algorithm can only work with the genetic material in the population, so without mutation, there is no way for it to do much better.

The real world sounds seem to take on other environmental qualities. For instance, real-walking-none-c1-8.wav sounds far more like the environment at it's final generation than it did at it's first generation. It is noisy, and it's amplitude peak sounds similar to the environmental footsteps. Compare this to real-pot-none-c1-2.wav, which evolved in the pot.wav environment, which is inharmonic but not at all noisy. The sound that emerges through evolution isn't noisy either. The population does not always take on the characteristics of the environment: sometimes necessary genetic material is lost through chance early in evolution; sometimes it was not there to start with.

5.1.5 Mutation Functions

Each of the mutation functions (§3.1.4) results in a characteristic output. In general, qualities from the initial population are preserved and qualities of the environment and of the mutation functions are added to the output as evolution progresses. Multiple mutation types can be used at the same time—in fact, I generally enable most mutations for compositions—but, for clarity we will only consider the results of their use when applied alone. Since we are experimenting solely with the single-location, fixed-length algorithm, we can only examine the mutation functions that operate on that algorithm: amplify, exponentiate, reverse, swap, and overwriting duplicate. These results provide enough insight into the algorithm's behavior that one can extrapolate how other mutations will function enough to use them compositionally.

5.1.6 Mutation by Amplification

Noise

The behavior of noise with the amplification mutation is disappointing if one expects the population to match the environment. This is hard to do with a population whose members initially sound the same: the only variation can come from mutation. When we have a small population and small mutation increments, the population is susceptible to random effects.

However, evolution occurs in both the fixed-crossover (noise-gliss-amp-c1-0.wav, noise-pot-amp-c1-0.wav, noise-walking-amp-c1-0.wav, noise-cricket-amp-c1-0.wav) and random-crossover (noise-pot-amp-c0-0.wav, noise-cricket-amp-c0-0.wav, noise-gliss-amp-c0-0.wav, noise-walking-amp-c0-0.wav) cases. In all of the fixed-crossover examples, the population evolves towards a loud first half and a quiet second half as described in §5.1.4. In all of the random-crossover examples, it evolves to a relatively constant amplitude sound by the end of 300 generations, although both noise-gliss-amp-c1-0.wav and noise-pot-amp-c1-0.wav are slightly louder early in the waveform. This is clearly the result of mutation. Another noteworthy feature, which I think adds musicality to the algorithm, is that evolution isn't always steady. Even with a population of noise, the 10 generations from the middle of noise-gliss-amp-c1-0.wav sound noticeably different from each other; some are muffled, some are brighter, etc.

The random-crossover examples sound more alike, both between environments and between generations. We can convince ourselves that noise-pot-amp-c0-0.wav, which evolves in the pot.wav environment, acquires an amplitude envelope reminiscent of a struck pot: a sharp onset with a decay. But, this is clearly confirmation bias: the other environments noise-cricket-amp-c0-0.wav, noise-gliss-amp-c0-0.wav, noise-walking-amp-c0-0.wav) evolve towards similar amplitude envelopes.

Sinusoids

The fixed-crossover sinusoid runs (sin-cricket-amp-c1-2.wav, sin-cricket-amp-c1-8.wav, sin-pot-amp-c1-2.wav, sin-walking-amp-c1-0.wav, sin-gliss-amp-c1-0.wav) exhibit the same amplitude behavior as the fixed-crossover noise runs: loud beginnings, quiet endings. Except, instead of evolving back towards a constant amplitude, all of the sample runs evolved towards silence for the second half. It's not clear if this is because of differences between the populations or the result of random chance. In the final 10 generations of evolution, we see slight amplitude variation in the resulting population. Since we started with a constant amplitude, this probably results from the mutation. These runs furnish us with another example of random seed effects: Examples sin-cricket-amp-c1-2.wav and sin-cricket-amp-c1-8.wav have the same environment and variables, but they converge to different pitches.

The variable-length sinusoid runs (sin-cricket-amp-c0-1.wav, sin-walking-amp-c0-0.wav, sin-gliss-amp-c0-0.wav, sin-pot-amp-c0-7.wav) are sonically more interesting. Initially, they resemble the variable-length runs without mutation: complex chords converging to sequences of pitches. The amplitude mutation has a more pronounced effect on these runs. The sinusoids evolved in the footsteps environment (sin-walking-amp-c0-0.wav) develop a similar amplitude envelope, with peaks near the footsteps and softer noise throughout. We never reach the point where the background sounds are as soft as they are in the environment, but the coincidence of amplitude peaks is noteworthy. Furthermore, we can safely assume that this isn't confirmation bias, since we don't see a peak corresponding to the last footstep from the other environments. In all of the runs, we evolved silence at the end of the samples. Again, this is probably the result of mutation, since even though the fixed-length runs without mutation tended towards silent endings, the variable-length runs without mutation did not have this pronounced an effect. Given the population of sinusoids, we would expect the population to tend towards silence both when the environment is silent and when the sinusoids are out of phase

with the environmental sounds. We see some of this behavior in the middle of the glissando run (sin-gliss-amp-c0-0.wav), which has pronounced peaks and troughs throughout the resultant waveform. The pot (sin-pot-amp-c0-7.wav), cricket (sin-cricket-amp-c0-1.wav), and footsteps (sin-walking-amp-c0-0.wav) evolve towards sounds that are loud at the beginning and soft at the end. This has some correspondence to their environment's amplitude envelopes: the cricket.wav is a short sample, the pot.wav fades out, and the footsteps.wav waveform is quiet throughout except for the footsteps themselves.

Real-World Sounds

In general, the real-world population exhibits the same mutation behavior as the noise and sinusoid populations. The primary difference is that the sonic diversity within individuals gives the algorithm more to work with. Consider the random-crossover examples real-gliss-amp-c0-0.wav, real-pot-amp-c0-2.wav, real-walking-amp-c0-5.wav and real-cricket-amp-c0-4.wav and the fixed crossover examples real-pot-amp-c1-0.wav, real-pot-amp-c1-4.wav, real-pot-amp-c1-5.wav, real-cricket-amp-c1-0.wav, real-pot-amp-c1-8.wav, real-gliss-amp-c1-1.wav, real-walking-amp-c1-3.wav, real-gliss-amp-c1-5.wav, real-walking-amp-c1-9.wav. In all cases, the resultant amplitude envelopes vary widely. The behavior exhibited in almost all of the examples from the sinusoid and noise populations, in which they evolve loud beginnings and quiet endings, is absent. This suggests that the behavior from the other populations results from lack of amplitude diversity the initial populations. This lack of diversity makes it impossible for small changes in amplitude to make enough of a difference to the individual's fitness to allow the population to leave whatever local minimum it's stuck in and acquire a very different amplitude envelope.

This set of starting conditions also gives us some interesting examples of different evolutionary outcomes resulting from changes to the random seed. First, the differences are more noticeable with fixed-crossover reproduction. With ran-

dom crossover, the population becomes fairly homogenous sounding. With fixed crossover, different recognizable sounds rise to prominence. Consider the examples real-pot-amp-c1-0.wav, real-pot-amp-c1-4.wav, real-pot-amp-c1-5.wav, and real-pot-amp-c1-8.wav that evolved in the pot.wav.wav environment. real-pot-amp-c1-0.wav converges to a point in which the first half of the individuals in the population is almost silent, and the second half is voice and cymbal. real-pot-amp-c1-4.wav converges to a point in which there is sound throughout; all percussion, but the first half is a couple of wood instruments being hit, and the second half is a more metallic percussive sound. real-pot-amp-c1-5.wav converges to a point in which the first half of the individuals is a quiet hum, and the second half is voice combined with a single percussive hit. real-pot-amp-c1-8.wav converges to a rattling, scraping sound. real-gliss-amp-c1-1.wav converges to ascending vocal pitches with a high metallic percussive sound, while real-gliss-amp-c1-5.wavconverges to two vocal sounds: the first is a high glissando up and the second is a lower sustained pitch. real-walking-amp-c1-3.wav and real-walking-amp-c1-9.wav are interesting because they converge to have different sounds for their first halves (a single metallic percussive sound vs. two wooden percussive sounds) but the same sound for their second halves (a metallic percussive sound mixed with voice).

5.1.7 Mutation by Exponentiation

Noise

The populations of noise evolved with the exponentiation mutation (noise-cricket-pow-c0-0.wav noise-pot-pow-c0-0.wav noise-walking-pow-c0-0.wav noise-gliss-pow-c0-0.wav noise-cricket-pow-c1-2.wav noise-pot-pow-c1-0.wav noise-gliss-pow-c1-0.wav noise-walking-pow-c1-0.wav) sound substantially like other populations of evolved noise. They evolve towards a slightly filtered sound.

Sinusoids

In all cases (sin-cricket-pow-c0-4.wav sin-cricket-pow-c0-8.wav sin-gliss-pow-c0-1.wav sin-gliss-pow-c0-4.wav sin-pot-pow-c0-0.wav sin-walking-pow-c0-0.wav sin-cricket-pow-c1-1.wav sin-walking-pow-c1-0.wav sin-gliss-pow-c1-0.wav sin-pot-pow-c1-0.wav), sinusoids evolved with the exponentiation mutation have the same large-scale features as are exhibited by evolution in general (§5.1.4). What's noteworthy is that the characteristic sound of the mutation can be heard in places. Unlike other evolutionary runs with sinusoids, which retain their crisp, sinusoid character, these runs are slightly distorted at points. This is particularly evident during the middle ten generations of most runs, before the population has converged to a point where fewer mutated genes are present.

These examples also present several instances of differing outcomes from the same variables, with different random seeds. sin-cricket-pow-c0-4.wav and sin-cricket-pow-c0-8.wav evolve towards a high-pitched local minima with shortened waveforms, but they converge to different places. sin-cricket-pow-c0-4.wav converges to a chirping pitch followed by a low pitch, with a scraping sound. I believe the scraping sound to have resulted from mutation, since it doesn't sound at all sinusoidal. sin-cricket-pow-c0-8.wavconverges to something that sounds rather like a pitchy, high pitched toilet plunger being used. sin-gliss-pow-c0-1.wav and sin-gliss-pow-c0-4.wav both converge to ascending pitches. This is noteworthy, since both evolved in the gliss.wav environment. However, sin-gliss-pow-c0-4.wavhas more discrete pitches than sin-gliss-pow-c0-1.wav.

Real-World Sounds

The real-world sounds evolved with the exponentiation mutation (real-cricket-pow-c0-6.wav, real-gliss-pow-c0-0.wav, real-pot-pow-c0-2.wav, real-pot-pow-c0-8.wav, real-walking-pow-c0-3.wav, real-cricket-pow-c1-0.wav, real-cricket-pow-c1-2.wav, real-gliss-pow-c1-0.wav, real-pot-pow-c1-2.wav, real-walking-pow-c1-6.wav, and real-pot-pow-c1-7.wav) also exhibit the characteristic sounds of the mutation.

Distortions are evident later in each run, much lie in the sinusoid runs, that are not evident in examples without the exponentiation mutation. Examples real-pot-pow-c0-2.wav and real-pot-pow-c0-8.wav, evolved in the pot.wav environment, and real-cricket-pow-c1-0.wav and real-cricket-pow-c1-2.wav, evolved in the cricket.wav environment, are more examples of different seeds resulting in different outcomes.

5.1.8 Mutation by Reversal

Noise

The examples of noise evolved with the reversal mutation (noise-cricket-rev-c0-6.wav, noise-gliss-rev-c0-0.wav, noise-pot-rev-c0-0.wav, noise-walking-rev-c0-0.wav, noise-cricket-rev-c1-4.wav, noise-gliss-rev-c1-4.wav, noise-pot-rev-c1-0.wav, and noise-walking-rev-c1-0.wav) also sound substantially like other populations of evolved noise. It is noteworthy that, despite the inability of the mutation to change amplitude, the population has still evolved amplitude variation. After some amount of evolution, the population also sounds rather filtered. This is probably for the same reasons as it sounds filtered without any mutation at all (§5.1.4).

Sinusoids

The populations of sinusoids evolved with the reversal mutation and random crossover points (sin-cricket-rev-c0-1.wav, sin-walking-rev-c0-0.wav, sin-gliss-rev-c0-5.wav, and sin-pot-rev-c0-0.wav) share many of the same features as those evolved without mutation (§5.1.4). However, the populations of sinusoids evolved with the reversal mutation and fixed crossover point (sin-cricket-rev-c1-8.wav, sin-walking-rev-c1-0.wav, sin-gliss-rev-c1-0.wav, and sin-pot-rev-c1-0.wav) are markedly different. The populations converge to points with quite a lot of amplitude variation. Although the reversal mutation doesn't allow the algorithm to explicitly change amplitude, it does allow phase to be changed, which could contribute to this affect.

Real-World Sounds

The real-world sounds provide the best examples of the characteristic sound of the reversal mutation. Because they don't converge as quickly, the random=crossover examples (real-cricket-rev-c0-0.wav, real-gliss-rev-c0-9.wav, real-pot-rev-c0-0.wav, and real-walking-rev-c0-8.wav) are thicker and more complex, making it more difficult to hear this characteristic sound. The fixed crossover examples (real-cricket-rev-c1-7.wav, real-gliss-rev-c1-9.wav, real-pot-rev-c1-8.wav, and real-walking-rev-c1-3.wav) demonstrate the sound more clearly.

Despite the relative lack of clarity, the characteristic sound of the reversal mutation is still evident in the random-crossover examples. real-cricket-rev-c0-0.wav exhibits a reversed percussive sound a few generations in, and it remains prominent throughout evolution. real-gliss-rev-c0-9.wav has a reversed vocal sound, that is also introduced early and lasts throughout the individual; in this case, it is edited through crossover and is shorter by the end of the example. All four of the random-crossover examples exhibit much shorter mutations throughout. These have the characteristic sound of the reversal mutation, but are much harder to perceive as the same sounds throughout. This either because they leave the population quickly so we don't grow to recognize them, or because they are so short that we can't recognize the source.[6]

The fixed-crossover examples tend to converge to a few similar members, so the characteristic sounds of the mutation tends to be easier to hear throughout the sound sample. In real-cricket-rev-c1-7.wav, a backwards vocal sound emerges fairly early. It seems to fade in, growing in loudness over several repetitions. This suggests that the mutation that produced the sound happened fairly early on, and that the mutated genes gained prominence in the population. Example real-gliss-rev-c1-9.wav has a reversed percussive sound that appears early in evolution. That particular mutation is eliminated from the population, but another reversed

[6] I find it much easier to follow a gene-segment through evolution if it is from a recognizable source, than if I can't assign a type to it.

percussive sound appears later in evolution and persists to the end. In addition, there are some reversed vocal sounds that do not *sound* reversed. Sustained sounds rarely sound reversed when they are reversed, but if they have slight amplitude variation throughout, reversal often manifests as repeated attacks of the sound. Example real-walking-rev-c1-3.wav doesn't exhibit any obvious reversals during the first 10 generations. By the middle 10 generations, a metallic percussive sound has been reversed. By the last 10 generations, multiple reversals have occurred, and the multiple-attack effect manifests. It is hard to hear much of the mutation early in real-pot-rev-c1-8.wav, but by the end there are some clearly reversed sounds, as well as the multiple-attack effect.

5.1.9 Mutation by Swapping

Noise

The noise examples evolved with the swapping mutation and random crossover point (noise-cricket-swap-c0-0.wav, noise-gliss-swap-c0-0.wav, noise-pot-swap-c0-0.wav, and noise-walking-swap-c0-0.wav) are not markedly different from the examples with no mutation at all (§5.1.4). However, the fixed-crossover point examples (noise-cricket-swap-c1-3.wav, noise-cricket-swap-c1-8.wav, noise-gliss-swap-c1-0.wav, noise-pot-swap-c1-0.wav, and noise-walking-swap-c1-0.wav) demonstrate the mutation more clearly. Unlike their mutation-less counterparts, they do not evolve loud beginnings and quiet endings. Because they have the ability to move portions of waveform from one point in time to another, their amplitudes are far more varied.

For instance, noise-cricket-swap-c1-3.wav and noise-cricket-swap-c1-8.wav vary only by seed. Example noise-cricket-swap-c1-3.wav is loud with decreasing amplitude for its first half, and soft with increasing amplitude for it's second half. Example noise-cricket-swap-c1-8.wav begins similarly, but it's second half is fairly constant amplitude throughout, except an extremely quiet waveform segment has been edited in towards the end. Example noise-gliss-swap-c1-0.wav has a much

more jagged amplitude envelope than the other examples.

Sinusoids

The random-crossover sinusoids evolved with the swapping mutation (sin-cricket-swap-c0-4.wav, sin-gliss-swap-c0-0.wav, sin-pot-swap-c0-0.wav, and sin-walking-swap-c0-0.wav) are quite different from the sinusoids evolved without mutation (§5.1.4). Swapping increases the amount of edits made to individuals. Just as random-crossover runs sound more fragmented than fixed-crossover runs, runs with the swapping mutation are more fragmented than runs without the swapping mutation. Runs converge to washes of sinusoids of different pitches spliced together, instead of converging to simpler sequences of pitches.

The fixed-crossover sinusoids evolved with the swapping mutation (sin-cricket-swap-c1-3.wav, sin-gliss-swap-c1-0.wav, sin-pot-swap-c1-0.wav, and sin-walking-swap-c1-5.wav) provide a clearer picture of the swapping mutation's behavior, since the population converges to the point where most of what we can hear are copies of the same individuals. What we end up with has wide amplitude variation and many different pitches. Although amplitude variation can't be introduced by the mutation itself, silence can creep in at the ends of waveforms. Once we can copy any part of the waveform and splice it anywhere else, silence can be introduced anywhere in the waveform. Despite the sound of editing, there is very little pitch variation in the waveforms at the end of evolution. Sometimes the waveform is a combination of a single pitch and silence (sin-walking-swap-c1-5.wav), sometimes the waveform is composed of a couple of pitches and silence (sin-gliss-swap-c1-0.wav, sin-cricket-swap-c1-3.wav), and sometimes there are multiple pitches and silences in parallel (sin-pot-swap-c1-0.wav). In this last case, the population hasn't converged many versions of the same individuals; there are two or more waveforms represented in the population.

Real-World Sounds

Real-world sounds with the swapping mutation exhibit many of the same features as sinusoids. One crucial difference is that it's easier to tell where the swapped components of a sound came from if we know the original samples well enough. The random-crossover examples (real-cricket-rev-c0-0.wav, real-walking-rev-c0-1.wav, real-gliss-rev-c0-0.wav, and real-pot-rev-c0-0.wav) don't converge enough for us to tell what edits happened to make the waveforms in the population. However, we can discern the characteristic sound of the mutation. In each of these examples, the population evolves towards a texture of dense cuts and splices. This mutation also lets the population evolve a varied amplitude envelope, which shares features with the environmental amplitude envelope. For instance, by the end of 300 generations of evolution, real-walking-rev-c0-1.wav has acquired a few peaks but is relatively soft throughout, much like the walking.wav environment it evolves in. Whereas real-gliss-rev-c0-0.wav has a rather jagged envelope, but it tends towards a higher amplitude envelope than the other examples, which corresponds to the gliss.wav environment. The pitch profiles also develop correspondences with the environments. The walking.wav environment has low background noise and real-walking-rev-c0-1.wav evolves to a noisy texture, clearly from percussive instruments; the vocal sounds have been weeded out of the population.

The fixed crossover examples (real-cricket-rev-c1-1.wav, real-walking-rev-c1-3.wav, real-gliss-rev-c1-0.wav, and real-pot-rev-c1-0.wav) give us a clearer picture of what happens to individuals. In real-cricket-rev-c1-1.wav, one of the most prominent sounds early in evolution is a vocal [i] sound. In the middle section, we can hear that a small snipped has been swapped out of the [i] sound, giving us two articulations of the vowel. By the end of evolution, I can hear at least 5 small pieces of this sound spread throughout the waveform. In real-gliss-rev-c1-0.wav, by the middle 10 generations, we hear some skipping that results from a part of the sound being swapped with something unobtrusive. By the end, we hear that

most of the intelligible sounds have been swapped away, and the resulting sound is rather choppy. Example real-pot-rev-c1-0.wav also evolves towards fragmented waveforms, although by the end of 300 generations of evolution, its constituent sounds are recognizable. By the middle ten generations, real-walking-rev-c1-3.wav has converged to the point where most of its individuals begin with a struck cymbal mixed with voice and end with another cymbal followed by two percussive sounds that sound as though they have already been edited by the mutation. By the last 10 generations of evolution, the sonic material that had been only in the second half of the individuals seems to dominate the entire individual.

5.1.10 Mutation by Duplication

Noise

The examples of noise evolved with the duplication mutation do not behave substantially differently from previous noise examples. Both the examples evolved with a random crossover point (noise-cricket-odup-c0-7.wav, noise-gliss-odup-c0-0.wav, noise-pot-odup-c0-0.wav, and noise-walking-odup-c0-0.wav) and the examples evolved with a fixed-crossover point (noise-cricket-odup-c1-6.wav, noise-gliss-odup-c1-2.wav, noise-pot-odup-c1-0.wav, and noise-walking-odup-c1-0.wav) grow somewhat pitchier over the course of evolution. We might expect their amplitude envelopes to behave like the amplitude envelope of noise evolved with the swap and reverse mutations, since the three mutations involve editing. But, it does not. It behaves more like the amplitude envelopes evolving with the amplify and exponentiate mutations. This might be because duplication can only stretch the point of the mutation; it cannot move material from one place to the other.

Sinusoids

The random-crossover examples (sin-cricket-odup-c0-3.wav, sin-gliss-odup-c0-0.wav, sin-pot-odup-c0-1.wav, and sin-walking-odup-c0-0.wav) and fixed

crossover examples (sin-cricket-odup-c1-0.wav, sin-pot-odup-c1-0.wav, sin-gliss-odup-c1-0.wav, and sin-walking-odup-c1-0.wav) of sinusoids evolving with the overwriting duplication mutation begin to exhibit the characteristic sound of the mutation: an echoing quality. Although it does not audibly affect the waveform, another characteristic of the mutation is that it sometimes introduces substantial DC offset by duplicating a segment of waveform that happens to have low enough frequency to be skewed in either positive or negative. This is seen in sin-pot-odup-c0-1.wav, sin-walking-odup-c0-0.wav, and sin-gliss-odup-c1-0.wav.

Real-World Sounds

The effect of the overwriting duplication is clearest with real world sounds, both with random crossover points (real-cricket-odup-c0-2.wav, real-cricket-odup-c0-4.wav, real-gliss-odup-c0-0.wav, real-pot-odup-c0-3.wav, and real-walking-odup-c0-9.wav) and with fixed crossover points (real-gliss-odup-c1-3.wav, real-cricket-odup-c1-0.wav, real-pot-odup-c1-3.wav, real-pot-odup-c1-5.wav, real-walking-odup-c1-3.wav, and real-pot-odup-c1-2.wav). In all cases, the sounds gain more rhythmic repetitions as evolution progresses. As in previous cases, fixed crossover runs converge more quickly than random-crossover runs. However, the characteristic sound of the mutation is more noticeable, and mutations near the crossover point can smear sounds more readily to the other side. The end result is that it is more difficult to discern the first and second halves of the individuals.

We hear both real-pot-odup-c1-2.wav and real-pot-odup-c1-3.wav, which differ only in their random seed, converge to fairly quiet percussive sounds, with real-pot-odup-c1-2.wav quieter than real-pot-odup-c1-3.wav. Example real-pot-odup-c1-5.wav, which also shares the same variables but has a different seed, converges to a louder vocal sound. real-pot-odup-c1-2.wav exhibits a single duplication fairly early in evolution. The mutation has been applied to a fairly long segment of waveform, and manifests as something a percussive tremolo. Midway through evolution, we hear a similar tremolo in the voice, but it is quickly drowned out

by percussive sounds. These percussive sounds exhibit mutation applied to much shorter segment of waveform and with many more duplications. The sounds take on a buzzing quality. By the end of evolution, most of the mutations have died out; the metallic percussive sound the population converges to has a bit of a rattle in its tail, but otherwise just sounds much like an unmodified sample. real-pot-odup-c1-3.wav has a similar trajectory to real-pot-odup-c1-2.wav, but the details are different. Vocal sounds are more prominent than percussive sounds in the beginning, although they both end with a percussive sound with slight mutation. Recall that since there is no amplification from mutation, the only source of amplitude variation is weighted fitness. Examples real-pot-odup-c1-2.wav and real-pot-odup-c1-3.wav are quite similar, but it is clear that neither of them converged to a very fit population. They ended up in similar local minima, because fitter sounds were lost to chance. Example real-pot-odup-c1-5.wav converges to a louder sound: a vocal sound much like those that are evident in the middle generations of real-pot-odup-c1-2.wav and real-pot-odup-c1-3.wav. It is interesting to note that real-pot-odup-c1-2.wav and real-pot-odup-c1-3.wav converge to an amplitude envelope that is similar to the environment, but that real-pot-odup-c1-5.wavhas a more similar spectrum.

Example real-cricket-odup-c1-0.wavconverges to a point that is fairly similar in spectrum, length, and amplitude to its environment. In the middle ten generations, one can clearly hear the competition between two sounds: a repetitive buzz followed by a voice and a voice with repetition applied to a fairly long segment of waveform. Both sounds are present in the middle ten generations, but one is louder, and then the other. In the end, the buzz followed by a voice wins. In the final ten generations, this the only audible sound present in the population. It has a buzzier quality, suggesting that it has mutated somewhat since its earlier instantiation.

Example real-gliss-odup-c1-3.wav still has the characteristic sound of the mutation: various tremolos and buzzes, depending on the length of the affected

segment. However, unlike evolution in the pot.wav environment, the evolution in the gliss.wav environment tended to select for heavily mutated samples. Although in the middle ten generations, there is a prominent vocal glissando, by the end of evolution, the population has converged to a very buzzing sound, whose original samples can hardly be identified. It is tempting to say they were originally percussive, but the mutation has enough impact on the timbre that the last part of the sound could just as easily be heavily transformed vocal sounds.

Example real-walking-odup-c1-3.wav converges to a point at which the original sounds are overwhelmed by the buzzing character of the mutation. Given the starting sounds available to the algorithm, this is a fair approximation of the background noise in the environment. The amplitude is also similar, with two loud peaks at the beginning, and a shorter peak midway through the sound.

The random-crossoer examples generally sound more denatured than the fixed-crossover examples. Example real-cricket-odup-c0-4.wav evolves to a point in which the sound is an unrecognizable gurgle. This suggests many edits in all prominent members of a diverse population. Example real-cricket-odup-c0-2.wav, which has evolved in the same environment with a different seed, converges a great deal more. The characteristic repetitive sound of the mutation is present throughout, but the resultant sound is pitched.

Example real-gliss-odup-c0-0.wav converges to a fairly noisy sound. Like real-cricket-odup-c0-4.wav, it has a gurgling quality, suggesting that the population has not converged. The amplitude envelope is the loud beginning and quieter endings that have characterized many of the amplitude envelopes in previous examples. As I suggest above, this is amplitude characteristic seems to characterize a collection of very common local minima.

Example real-pot-odup-c0-3.wav converges to a loud beginning and soft ending. However, it doesn't have the same shape as the envelopes that are characteristic of the local minima above. Since the environment is a struck pot, this envelope would have been selected for by evolutionary pressures. The population

converges to a point in which it is primarily composed of a repeated vocal sound.

The real-walking-odup-c0-9.wav example converges to very quiet, crackling noise with a peak at the beginning. Again, this is fairly similar, both in timbre and envelope, to its environment.

5.2 Creative Works

I've produced several tape pieces using my genetic algorithm framework. *Run5* and *run12* are two pieces that present different runs of the same underlying algorithm on the same population. *Run16* uses the same algorithm as *run5* and *run12 Snapshots* is a CD-length tape loop composed from multiple runs of a different instantiation of the algorithm. An untitled real-time installation uses a substantially different algorithm; in order to function in real time, it manages time and presents sounds in a substantially different manner than the tape pieces.

5.2.1 Run5 & Run12

Run5 and *run12* are two pieces evolved from the same population with the same initial conditions. Run5 is 3'54 and is for stereo (two channels of sound). Run12 is 10' and is for quad (four channels of sound). Both evolve the same population of sounds gathered from around my apartment: toys, playground equipment, pots, pans, stemware, bottles, coins, pebbles, etc. *Run5*[Mag03c] was presented in concert on November 6, 2003 at UCSD and on April 13, 2006 at Mills College. *Run12* was presented in concert on November 10, 2003 at UCSD and included on the *Sound check one*[Mag03b] CD.

These were the first real pieces I made using genetic algorithms. I developed the framework as I saw compositional need. I started with an otherwise conventional genetic algorithm (described in [Mag03a]). I wanted the results to be rhythmically variable, so I replaced fixed-length critters using discrete generations with variable-length critters using overlapping generations (pg. 38 ff.). Based on

my experiments in §5.1, I knew that all of the mutations I had developed would be sonically interesting. I gave every mutation the same probability except exponentiate, which I made less probable because in large doses it can alter the sounds more drastically than the others.

I did not want these compositions to have a linear trajectory. I wanted something with an organic form. So, I set out to develop a world that changed in time. I let the environment interpolate between sounds, occasionally shifting to an entirely different sound (pg. 61 ff.). I wanted the world to map onto speakers, so I let each speakers have its own environmental conditions (pg. 60 ff.). These environmental conditions were the fitness functions and variables tied to each environment. I liked the sounds of each mutation over different chunk sizes.[7] I wanted the potential for them to manifest in different ways in the same piece. A hodgepodge of all possibilities at once would not be as musically compelling. I thought it would be better to present periods in which mutations had one musical character followed by mutations with another character. Mutation probabilities and chunk sizes were given bounds, not fixed values. Each mutation had its own randomly assigned chunk size and probability. New values are selected each time the changing-world algorithm triggers a cataclysmic change.

I wanted the speakers to interact—anything less would just be separate pieces that happened to play at the same time out of different speakers—so I developed a system that allowed sounds to migrate between speakers (pg. 60 ff.). Another relic of the original algorithm was the method for selecting parents (pg. 58 ff.). Each time an individual dies, another individual replaces it in the population. The replacement individual is the offspring of two parents chosen from the population based on fitness. This means that sounds might play without reproducing even once. When I composed these pieces, I had not considered the possibility of having the replaced individual always be a parent.

The output representation of the population was based on fitness. Indi-

[7]Chunk size here is in terms of number of zero crossings, as described in Chapter III, rather than time-based chunks, as described in chapterIV.

viduals' amplitudes were weighted by their fitness, then the entire population was recorded into a sound file with a channel for each location (pg. 38 ff.). I wanted the potential for musical surprises. With this representation, unfit sounds can lurk unheard in the population and appear when the environment changes. For this potential to be tapped, the unfit sounds can't be allowed to die out. This meant it was extremely important to maintain biodiversity. So, I used a biodiversity modifier, that would make sounds less fit each time they reproduced, giving less fit sounds better odds of reproducing (pg. 44).

Each of these took several hours to calculate. To get *run5*, I ran the algorithm repeatedly with the same starting conditions then sat down a few days later with the results and picked the one I liked best. All of the runs were pleasant to listen to and shared certain qualities, but *run5* were selected for presentation because it was particularly engaging. After the concert, most people told me that they liked it, but that they wished it were longer. *Run12* was generated in the same way, but with more generations of evolution.

Both *run5* and *run12* evolved in four-location environments. I chose my two favorite locations for *run5*, and presented all locations in *run12*. This decision was both practical and aesthetic. The concert on which *run5* was first presented had two speakers; the concert for *run12* had four speakers. I could have just evolved two locations for run5, but I liked the idea of presenting a partial view of the world. Evidence of the unpresented locations creeps into the presented locations through migration, but the entire process doesn't confront the listener.

Both pieces sound repetative and rhythmic on the surface. Closer attention reveals a thick texture of aperiodic sounds that somehow give the illusion of periodicity. This compound rhythm changes over the course of the piece. The shape repeats with rhythmic deviations.

Because multiple copies of fit sounds appear in the population, at some parts of the pieces there are hints of effects created by digital delay. These effects are subtle and intermittent.

There's a thrilling moment almost two minutes into *run5* where a periodic whistling sound—I think it's the rubbed rim of a piece of stemware—blooms out of the background textures, plays a few times at different volumes, then disappears. The sound appears and disappears in five seconds. Since run5 only presents two of four co-evolved channels, one might think this is the result of migration from an unheard channel. However, the same phenomena occurs for different sounds in several places in run12. Migration changes a sound's environment, and drastically changing the environment is necessary to produce the drastic change in fitness that foregrounds previously unheard sounds.

In addition to these blooming moments that pop up in both pieces, sounds constantly fade in and out over long periods of time in both pieces. For instance, there is a sound of a rattling lid on a glass container that takes 30 seconds to appear in run5 that takes another 30 seconds to disappear. It returns over three minutes into the piece and stays around in various guises until the end of the piece. Most of the sounds that make up the periodic texture come and go over long periods of time. The listener can focus on any one sound and hear it gradually change in time and recognize it when it comes back later in the piece. This results from gradually changing environments.

5.2.2 Run16

Run16 was a piece developed in collaboration with saxophone improviser Tracy McMullen for a performance on January 31, 2004, as part of the *Powering Up/Powering Down* festival at UCSD. Ideally, we would have wanted to use an interactive algorithm. However, when we developed this piece, the algorithm could not run in real-time. But, we knew that a real-time algorithm changing only through evolution wouldn't ever catch up with the playing. It would always evolve towards the sound world she was creating, and gain features from it if she stayed there long enough. But, she would generally be a moving target. This meant that we could simulate interaction using the same underlying algorithm as I used for

run5 and *run12*.

We recorded several sessions of Tracy improvising on her own. I edited two types of material from these sessions: discrete sounds and larger phrases. Since Tracy's playing in the performance would be a moving target, the collection of large phrases could be used as a pool of potential fitness functions. My thought was that evolving toward a succession of Tracy-like phrases would be close enough to evolving towards Tracy's actual playing.

The initial population was comprised of discrete sounds. Since longer sounds are produced either from the duplication mutation or from the random crossover points in sexual reproduction, evolutionary pressures should cause the discrete sounds in the initial population to gradually build up phrase-like structures.

I produced several runs of the algorithm with samples of Tracy's playing. I chose my favorite run for the performance. We rehearsed with other runs so Tracy would be improvising with a previously unheard tape. We rehearsed with other runs to simulate a live run of the algorithm in anticipation of eventually developing a real-time version.

5.2.3 *Snapshots*

Snapshots is a tape piece intended for installation. It was commissioned by Adrienne Jenik for her *Specflic* installation at UCSD on October 28, 2005. It was also presented in the Artpool gallery in Budapest on April 10–12, 2006, as part of the *EvoWorkshops 2006* conference. An excerpt of it is included in the *Process Revealed* DVD, which was released as part of the *4th European Workshop on Evolutionary Music and Art*.

Snapshots is roughly the length of a CD and is intended for looping . A CD-lengthed loop was a practical decision. I wanted it to be as long as possible and still fit on a CD; I could have made it a more regular time — an even hour, for instance — but I didn't want its looping to predictably align with the clock.

Rather than evolve the same set of sounds once, *Snapshots* presents multiple runs of the same algorithm on subsets of the same set of sounds. The sounds are all sampled from technological artifacts: bleeps, bloops, rings, fans, machinery, buttons, etc. The variables are the same for each run, allowing initial conditions to send the algorithm in different evolutionary directions.

In previous versions of the algorithm (§5.2.1–5.2.2), I had mutations available for editing and changing amplitude. The DUPLICATE mutation can make pitches where there were none, but it can't change pitch. I wanted the algorithm to have the potential to change pitch. So, I added the CHANGE LENGTH mutation (§3.1.4).The introduction of the CHANGE LENGTH mutation is perceptible: there are places in the piece where pitches can be heard to drift as evolution progresses.

I wanted to explore ways of introducing biodiversity with something more integral to the act of reproduction than the biodiversity modifier (pg. 44). I changed the reproduction model so that each individual reproduced at least once: an individual would be one of the parents of the individual that would replace it. The other parent would be chosen based on fitness (pg. 58 ff.).

The algorithm was run many times and I chose some of the runs to splice together to make the piece. As with *run5*, I ran the algorithms with four discrete locations, then chose two channels for each run I presented. I modified the algorithm from the one used for *run5* and *run12* so I could run the algorithm for a particular time period rather than for a set number of generations. The algorithm ran until it reached the designated end point, then continued to run without reproduction until each member of the population had finished playing. I edited the runs together: some of them I faded out early; some of them played their entire length; some areas overlap quite a lot; others have no overlap.

Beyond the surface quality resulting from the sounds chosen for the initial population, the *run5* and *run12* pieces and *Snapshots* share sonic qualities. *Snapshots* is made from multiple evolutionary runs, each of which shares the same

formal structure as *run5* and *run12*. There is a repetative, rhythmic quality. The rhythms constantly change. Each repetition is different from the last. Sometimes the changes are slight; the piece evolves gradually for a time. Sometimes another member of the population has risen to prominence. Sometimes it stays for a moment then disappears; sometimes it stays for a while, and another period of gradual evolution begins.

5.2.4 Real-Time Installation

I am currently developing a real-time version of the algorithm as part of a collaborative installation with Sean Griffin, Miya Masaoka, and others.The installation examines mythos and pseudoscience that have grown up around Neanderthals since their discovery in . My component of the installation evolves samples of Neanderthal speech and song in the context of a modern Homo Sapien gallery. The samples are from poems and songs written by Sean Griffin in Mousterian, a conjectural Neanderthal language. The piece is designed to be installed in a space ringed by speakers. The ring world that the sounds inhabit is mapped onto the ring of speakers (see §3.2.2).

The out-of-time versions of the algorithm were programmed in ansi c and were free to be fairly ideological. The only constraint on the algorithm was my patience. Since I could always run the algorithm when I went to bed or left the house, this wasn't much of a constraint. The real-time version is programmed in Pure Data with some custom objects written in c. The constraints are considerable and I have had to sacrifice ideology.

Pure Data calculates data a block at a time. If the computer is able to calculate everything it's asked to calculate within the time a block takes to play, the output is seamless. If the computer is asked to calculate too much, it will delay output of the block, causing a popping sound. To some extent, I can get around this with larger block sizes, but this introduces latency. The evolutionary process is relatively forgiving of latency, but I don't want to rely on this too heavily in a

version of the algorithm intended for real-time.

In an ideal world, all critters would be identical agents who interact with their idiosyncratic representations of the world. Over their lifespan, they would act in the world, then they would mate and produce offspring to take their place in the population when they reached the end of their lifespan. Unfortunately, reproduction is computationally expensive. Individuals have to find a mate and reproduce, then their offspring has to undergo mutation. These operations are easiest to implement in Pure Data as instantaneous processes: they have to happen in a single block. Computational constraints already require me to use much smaller populations than I used in out-of-time versions of the algorithm. I have had to place constraints on the length of an individual because reproduction of a single, long individual, can cause glitches. I can't risk multiple critters reproducing at the same time. So, I have developed a real-time algorithm that takes reproductive agency from individuals in the population and uses a single, central, reproduction module.

Only one critter at a time reproduces. Critters can't reproduce unless they've been played at least once. When they play, they are added to a queue if they aren't already in it. As soon as one individual finishes reproducing, the next individual in the queue starts reproducing. The individual looks for a mate within some mating distance specified by the individual's data gene. Once the pool of potential mates is determined, mate choice is random based on fitness.

Rather than play once with volume weighted by fitness, sounds play periodically over their (much longer) lifespans. Fit sounds play more frequently than unfit sounds. All mutations described in §3.1.4 are available to the algorithm.

The rhythmic nature of evolution manifests in the real-time algorithm, but it changes more slowly than the tape version. The repetitions are regular over the course of an individual. Instead of hearing a gradually changing aggregate of all sounds, rhythms of individuals can be heard as such. This perception changes somewhat as the population converges. Because individuals look for mates nearby,

there is a tendency for inbreeding. As nearby individuals converge, groups sound more like channels from the tape pieces.

The fitness function uses the output of the fiddle object from Pure Data. The fitness is the correlation between the individual's pitch and the environment's pitch plus the correlation between the individual's amplitude envelope and the environment's amplitude envelope. The environment is the environment of the room in which the piece is installed. It is fed to the algorithm by mics placed around the room and mapped onto the evolutionary space (see §3.2.2). It is important to note that since speakers are playing into the room and mics are sampling from the room, the sounds of the population playing through the speakers are part of the environment the sounds themselves evolve in. This acts as a sort of social evolutionary force.

I've experimented with the installation in several environments and have found it to be quite responsive. It is more responsive to the amplitude component of fitness than to pitch. When the environment is dense, the population grows dense. When the environment is sparse, the population grows sparse. Since there is a feedback between the population and the environment, there is a point where there will be balance—the density of the population stays the same without appreciable sounds in the environment besides the sound from the speakers. Turning down the speaker volume will cause the population to grow sparse.

The tape pieces are intended for much shorter periods of presentation than the real-time installation. As such, the installation evolves much further—it pushes the boundaries of the algorithm and brings to light constraints on its usage. First, the algorithm cannot create; it can only modify. Second, extreme periods of evolution eventually degrade the signal.

The algorithm can only evolve sounds that are descended from starting sounds. Eventually, biodiversity is lost. If there is even one unfit individual in the population that is carrying genes not present in the fit individuals, those sounds have the potential to enter the rest of the population if the environment changes

such that that individual becomes fit. Once that individual dies, its sounds are lost forever. If the environment changes, the population can only adjust to it using sounds remaining in the gene pool.

The above claim about sounds in the population following the density of the environment only holds to a point. Loud sounds get shorter and softer relatively easily. If the environment is silent for too long, it loses its ability to spring back and become dense. If there are any long individuals in the environment, they will parent many offspring when the population becomes dense again and the population will spring back quickly. Offspring can range in length from arbitrarily short to the length of both parents. In practice, most offspring fall in the middle. But, the population only needs one arbitrarily short individual to become incredibly fit to bring down density if the environment calls for short, sparse sounds. Since the length of the offspring can, at best, be the combined lengths of both parents, it takes much more time to evolve long individuals once they've all become very short.

Loss of biodiversity affects timbre even more. In one experimental session, I ran the algorithm in a room with several friends who were studying, watching TV, and talking. The population of sounds were speech based, and they maintained a speechy timbre. At some point, people stopped talking and started approaching the mics and making noises in a deliberate attempt to affect the sounds produced by the algorithm. Mostly people clapped, made clicking noises with their tongues, and PTHBed. After 5–10 minutes, the population responded to the impulses by becoming gritty. Consonants were relatively fit, but vowels weren't. The vowels sounds were lost. People decided they didn't like the sound and tried singing long tones and humming. There were not any harmonic sounds left in the population to benefit from the new environment. Since there were no vowels left to benefit from the new environment, the only way to get pitched sounds would be to rebuild them from the consonants through mutation. This is possible, assuming the environment stays the same. Since the odds of mutation are low, it would take a very long

time to build pitchy sounds solely through mutation. And the sounds that were build this way would still have some timbral print of the consonants they were constructed from.

After very long periods of evolution—on the order of days—all sounds degrade. Initially, edits are far enough apart to leave identifiable ssegments of sound between them. If edits can happen at any zero crossing, eventually most zero crossings in the waveform will have been the site of an edit at some point over the course of evolution. Intelligibility, or recognizable timbres and snippets from the original sounds, will be lost. This process is even quicker—on the order of hours—if edits are allowed to happen anywhere in the waveform and not restricted to zero-crossings. The resultant sounds are staticy growls.

This means there are practical constraints on the installation: it can't run forever with impunity. It can run in a gallery setting where it is turned off at night and started up fresh every morning. If it's going to be left running for long periods of time, there will need to be some algorithmic mechanism to counter degredation. I will probably add a counter that keeps track of how many generations have occurred for a particular agent. After a certain number of generations, instead of reproducing, the original sound from the initial population, or some other new sound, can be introduced to the population and the population can continue to evolve.

5.3 Future Directions

The evolutionary framework described here has a demonstrated potential to produce a myriad of aesthetic sonic artifacts. Here are some directions I'd like to take this work in, as the future permits.

- One of my first concerns is to find a way to modify the algorithm to restore reproductive agency to individual sounds. The most obvious way to do this is to replace the control objects in Pure Data that implement reproduction and

mutation instantaneously with signal objects that implement reproduction in time. This should distribute the computations over the life of an individual so that it will no longer matter if multiple individuals reproduce at the same time.

- Explore the use of sonic features instead of waveforms as fitness functions.
- Explore implicit fitness functions. I would like to explore a system that uses purely implicit fitness functions. In my review of the literature (§2.3), I describe works by Brooks and Ross [BR96], Dahlstedt and Nordahl [DN01], and Berry and Dahlstedt [BD03] that create sound works using simulated worlds. All of them use synthesized sounds, but I think it would be interesting to develop simulated worlds along these lines that use the genetic basis developed in §3.1.1. They might result in pieces with different formal behaviors than those whose form comes from evolving in an algorithmically generated world (as in §3.2.2).
- It would be interesting to make an online installation that harvests sounds for evolution—either uploaded from users or from public domain or creative commons websites. They could be initial population or the fitness function.
- Deriving form from process is an important part of using this evolutionary framework, but *Snapshots* demonstrates that larger forms that incorporate evolutionary segments can be aesthetically rewarding. I would like to explore the use of this framework to produce tape pieces whose formal structure is different from the one dictated by the evolutionary process. For instance, I'd like to make a piece that evolves backwards: reverse all of the initial samples, run the algorithm as normal, then reverse the result. I've done some experiments with this. It is especially interesting with speech. The result is intelligible words that gradually are constructed from fragmented babble. It might be interesting to combine forwards and backwards evolution. It might be nice to just use a short snippet of evolved sounds in the body of another

tape piece.

- The metaphor of evolution in particular environments suggests site-specific pieces. I'd like to do a series of site-specific pieces that use sounds recorded at Geocache sites. At each site, I would record sounds and use them as my initial population. I would evolve a 3-5 minute piece using an algorithm similar to the one used in *Snapshots* (§5.2.3). The piece would be burned to a CD then left as a hitchhiker in the cache where it was recorded. People who find the CD would be invited to burn themselves a copy, but to leave the original CD in another cache.

- It would be nice to develop a system based on my work with Tracy McMullen, that would let the performer play with their own samples and have those samples respond to what they are doing in the performance. Given the speed at which the algorithm responds to the environment, I think a slightly faster version of the algorithm could be adapted for performance. It could be seeded with prerecorded samples or take samples from the performance. The later could be done either manually or algorithmically.

- I've experimented with an algorithm in which a sub-population is created from automatically sampled sounds; each newly sampled sound replaces the oldest sound in the sub-population so that all samples are relatively recent. The rest of the population is, in effect, the working population. Only the working population mates and plays back; but individuals in the working population only select mates, based on fitness, from the auto-recorded sub-population. This means recent sounds are always introduced into what is played back, while memories of past sounds remain in the algorithm's output. This could be interesting for either an installation or live performance.

Appendix

The tables below provide details regarding the parameters that vary between examples discussed in §5.1. The following program variables are held constant for all examples discussed in §5.1. All waveforms in the populations are normalized before processing. All examples run for 300 generations. The biodiversity modifier is 0.95 for each example.

file	cross		Mutation				pop	Soundfiles		Seed		
name	over	amp	dup	pow	rev	swap	size	env	pop			
noise-cricket-all-c01	0	0.05	0.05	0.05	0.05	0.05	101	noise	cricket	43293096	10559325	45606104
noise-cricket-all-c03	0	0.05	0.05	0.05	0.05	0.05	101	noise	cricket	77270357	3779144	22818268
noise-cricket-all-c10	1	0.05	0.05	0.05	0.05	0.05	101	noise	cricket	47485088	17197482	91477056
noise-cricket-all-c16	1	0.05	0.05	0.05	0.05	0.05	101	noise	cricket	23628766	15591008	38836465
noise-cricket-amp-c00	0	0.05	0	0	0	0	101	noise	cricket	49386355	22845755	74711416
noise-cricket-amp-c10	1	0.05	0	0	0	0	101	noise	cricket	4625383	84881360	59954716
noise-cricket-none-c00	0	0	0	0	0	0	101	noise	cricket	71738919	69674637	54824857
noise-cricket-none-c07	0	0	0	0	0	0	101	noise	cricket	84320446	620113	42257326
noise-cricket-none-c08	0	0	0	0	0	0	101	noise	cricket	71359546	25667508	89221312
noise-cricket-none-c13	1	0	0	0	0	0	101	noise	cricket	91129370	23002981	56226085
noise-cricket-odup-c07	0	0	0.05	0	0	0	101	noise	cricket	19472774	98901610	95526510
noise-cricket-odup-c16	1	0	0.05	0	0	0	101	noise	cricket	87582801	31814948	13158686
noise-cricket-odup-c17	1	0	0.05	0	0	0	101	noise	cricket	19891356	25649526	36211225
noise-cricket-pow-c00	0	0	0	0.05	0	0	101	noise	cricket	50762939	85852661	63519255
noise-cricket-pow-c04	0	0	0	0.05	0	0	101	noise	cricket	30992785	78097985	91242474
noise-cricket-pow-c07	0	0	0	0.05	0	0	101	noise	cricket	83313470	78628301	53989057
noise-cricket-pow-c08	0	0	0	0.05	0	0	101	noise	cricket	46100701	99273180	41072954
noise-cricket-pow-c09	0	0	0	0.05	0	0	101	noise	cricket	18717013	50958386	27037799
noise-cricket-pow-c12	1	0	0	0.05	0	0	101	noise	cricket	16488520	99904024	64621298
noise-cricket-pow-c14	1	0	0	0.05	0	0	101	noise	cricket	45133852	33538879	31552687
noise-cricket-rev-c06	0	0	0	0	0.05	0	101	noise	cricket	18454221	69587125	75262492
noise-cricket-rev-c14	1	0	0	0	0.05	0	101	noise	cricket	15953253	26460847	3012364
noise-cricket-rev-c18	1	0	0	0	0.05	0	101	noise	cricket	56005799	3690625	44481226
noise-cricket-swap-c00	0	0	0	0	0	0.05	101	noise	cricket	65518802	65164492	49962044
noise-cricket-swap-c13	1	0	0	0	0	0.05	101	noise	cricket	51993196	94276797	53651555

file	cross	Mutation					pop	Soundfiles		Seed		
name	over	amp	dup	pow	rev	swap	size	env	pop			
noise-cricket-swap-c18	1	0	0	0	0	0.05	101	noise	cricket	22642690	64586789	89713314
noise-gliss-all-c01	0	0.05	0.05	0.05	0.05	0.05	101	noise	gliss	61382360	41193507	60620692
noise-gliss-all-c10	1	0.05	0.05	0.05	0.05	0.05	101	noise	gliss	11427312	23702462	12491402
noise-gliss-amp-c00	0	0.05	0	0	0	0	101	noise	gliss	76731938	40255073	44374543
noise-gliss-amp-c10	1	0.05	0	0	0	0	101	noise	gliss	31550451	77271107	31461800
noise-gliss-none-c04	0	0	0	0	0	0	101	noise	gliss	12251361	20903206	87774187
noise-gliss-none-c16	1	0	0	0	0	0	101	noise	gliss	66203269	32541285	40686266
noise-gliss-odup-c00	0	0	0.05	0	0	0	101	noise	gliss	97641715	57010512	82655167
noise-gliss-odup-c12	1	0	0.05	0	0	0	101	noise	gliss	7756083	82918150	83650568
noise-gliss-odup-c19	1	0	0.05	0	0	0	101	noise	gliss	72680740	15367449	85705366
noise-gliss-pow-c00	0	0	0	0.05	0	0	101	noise	gliss	19031540	18715171	74114079
noise-gliss-pow-c10	1	0	0	0.05	0	0	101	noise	gliss	97331242	25878436	53564024
noise-gliss-pow-c13	1	0	0	0.05	0	0	101	noise	gliss	19950760	26679383	21008591
noise-gliss-pow-c15	1	0	0	0.05	0	0	101	noise	gliss	19918026	98330042	99493517
noise-gliss-rev-c00	0	0	0	0	0.05	0	101	noise	gliss	87869994	66908975	75491595
noise-gliss-rev-c14	1	0	0	0	0.05	0	101	noise	gliss	29045333	99807910	45285204
noise-gliss-swap-c00	0	0	0	0	0	0.05	101	noise	gliss	54280246	56376228	56201959
noise-gliss-swap-c10	1	0	0	0	0	0.05	101	noise	gliss	68035371	9900798	32252918
noise-pot-all-c00	0	0.05	0.05	0.05	0.05	0.05	101	noise	pot	49024800	84120001	58108859
noise-pot-all-c10	1	0.05	0.05	0.05	0.05	0.05	101	noise	pot	31121095	94218104	58161805
noise-pot-none-c00	0	0	0	0	0	0	101	noise	pot	71121059	33237943	24722904
noise-pot-none-c11	1	0	0	0	0	0	101	noise	pot	30274354	51419946	42419207
noise-pot-odup-c00	0	0	0.05	0	0	0	101	noise	pot	79574076	47042282	6065490
noise-pot-odup-c10	1	0	0.05	0	0	0	101	noise	pot	26483811	17266054	98116256
noise-pot-pow-c10	1	0	0	0.05	0	0	101	noise	pot	83159976	36088896	9764803

file	cross	Mutation					pop	Soundfiles		Seed		
name	**over**	**amp**	**dup**	**pow**	**rev**	**swap**	**size**	**env**	**pop**			
noise-pot-rev-c00	0	0	0	0	0.05	0	101	noise	pot	6397633	34910911	56672863
noise-pot-rev-c10	1	0	0	0	0.05	0	101	noise	pot	58105680	22915314	87073564
noise-pot-swap-c00	0	0	0	0	0	0.05	101	noise	pot	11214660	84181974	84291095
noise-pot-swap-c10	1	0	0	0	0	0.05	101	noise	pot	31074996	84753645	5927890
noise-walking-all-c00	0	0.05	0.05	0.05	0.05	0.05	101	noise	walking	54401472	89946732	22923309
noise-walking-all-c10	1	0.05	0.05	0.05	0.05	0.05	101	noise	walking	72325788	59588175	68814583
noise-walking-amp-c00	0	0.05	0	0	0	0	101	noise	walking	53332603	89524350	60223164
noise-walking-amp-c10	1	0.05	0	0	0	0	101	noise	walking	18113444	80689356	34328130
noise-walking-none-c00	0	0	0	0	0	0	101	noise	walking	77283421	90136607	36209639
noise-walking-none-c10	1	0	0	0	0	0	101	noise	walking	97358539	12574095	94961395
noise-walking-odup-c00	0	0	0.05	0	0	0	101	noise	walking	60560508	28424793	69441185
noise-walking-odup-c10	1	0	0.05	0	0	0	101	noise	walking	29624682	25892326	18514231
noise-walking-pow-c00	0	0	0	0.05	0	0	101	noise	walking	85963445	34953474	57601378
noise-walking-pow-c10	1	0	0	0.05	0	0	101	noise	walking	93283050	29630768	54315102
noise-walking-rev-c00	0	0	0	0	0.05	0	101	noise	walking	82228417	26369845	75136380
noise-walking-rev-c10	1	0	0	0	0.05	0	101	noise	walking	21043872	84422286	41059276
noise-walking-swap-c00	0	0	0	0	0	0.05	101	noise	walking	76518630	37719536	22221236
noise-walking-swap-c10	1	0	0	0	0	0.05	101	noise	walking	79198705	26559457	48931889
real-cricket-all-c02	0	0.05	0.05	0.05	0.05	0.05	50	real	cricket	25284153	13956867	31313516
real-cricket-all-c10	1	0.05	0.05	0.05	0.05	0.05	50	real	cricket	11406186	73564934	52160867
real-cricket-amp-c04	0	0.05	0	0	0	0	50	real	cricket	24884671	48815142	7658037
real-cricket-amp-c10	1	0.05	0	0	0	0	50	real	cricket	74417840	93972592	29939287
real-cricket-none-c00	0	0	0	0	0	0	50	real	cricket	19396821	978750	76171884
real-cricket-none-c14	1	0	0	0	0	0	50	real	cricket	67904858	43014484	44265108
real-cricket-odup-c02	0	0	0.05	0	0	0	50	real	cricket	66935513	64709756	63503727

file	cross			Mutation			pop	Soundfiles			Seed	
name	over	amp	dup	pow	rev	swap	size	env	pop			
real-cricket-odup-c04	0	0	0.05	0	0	0	50	real	cricket	44273979	55944076	75216045
real-cricket-pow-c06	0	0	0	0.05	0	0	50	real	cricket	15885196	10822857	43761893
real-cricket-pow-c10	1	0	0	0.05	0	0	50	real	cricket	44987473	24863621	97076892
real-cricket-pow-c12	1	0	0	0.05	0	0	50	real	cricket	55096973	59608067	51434661
real-cricket-rev-c00	0	0	0	0	0.05	0	50	real	cricket	90884683	95322563	14971033
real-cricket-rev-c17	1	0	0	0	0.05	0	50	real	cricket	98909284	78224380	33253245
real-cricket-swap-c00	0	0	0	0	0	0.05	50	real	cricket	14918809	32273978	3903746
real-cricket-swap-c11	1	0	0	0	0	0.05	50	real	cricket	69942356	62734142	51062593
real-cricket-swap-c16	1	0	0	0	0	0.05	50	real	cricket	88828378	72312459	60956315
real-gliss-all-c00	0	0.05	0.05	0.05	0.05	0.05	50	real	gliss	95877086	93764220	99461538
real-gliss-all-c08	0	0.05	0.05	0.05	0.05	0.05	50	real	gliss	38922279	95964780	98564959
real-gliss-all-c10	1	0.05	0.05	0.05	0.05	0.05	50	real	gliss	29530426	5766019	27123498
real-gliss-amp-c00	0	0.05	0	0	0	0	50	real	gliss	34609756	93450835	79206299
real-gliss-amp-c11	1	0.05	0	0	0	0	50	real	gliss	89082170	48408776	6304831
real-gliss-amp-c15	1	0.05	0	0	0	0	50	real	gliss	17549090	2515250	60728080
real-gliss-none-c00	0	0	0	0	0	0	50	real	gliss	49049962	52118866	6538006
real-gliss-none-c10	1	0	0	0	0	0	50	real	gliss	92168685	34659160	92610418
real-gliss-odup-c00	0	0	0.05	0	0	0	50	real	gliss	11081047	51528234	15041383
real-gliss-odup-c13	1	0	0.05	0	0	0	50	real	gliss	70373759	29894429	79723163
real-gliss-pow-c00	0	0	0	0.05	0	0	50	real	gliss	4708434	57324749	51977049
real-gliss-pow-c10	1	0	0	0.05	0	0	50	real	gliss	55245209	97494801	80242143
real-gliss-rev-c09	0	0	0	0	0.05	0	50	real	gliss	29165525	45241819	66435784
real-gliss-rev-c19	1	0	0	0	0.05	0	50	real	gliss	59455391	60287171	58269714
real-gliss-swap-c00	0	0	0	0	0	0.05	50	real	gliss	13561415	1469869	49184141
real-gliss-swap-c10	1	0	0	0	0	0.05	50	real	gliss	66191813	8507885	58828956

file	cross	Mutation					pop	Soundfiles		Seed		
name	**over**	**amp**	**dup**	**pow**	**rev**	**swap**	**size**	**env**	**pop**			
real-pot-all-c00	0	0.05	0.05	0.05	0.05	0.05	50	real	pot	1206950	76319405	34381853
real-pot-all-c15	1	0.05	0.05	0.05	0.05	0.05	50	real	pot	85632679	34885989	35886502
real-pot-amp-c02	0	0.05	0	0	0	0	50	real	pot	2821216	49824850	28656974
real-pot-amp-c10	1	0.05	0	0	0	0	50	real	pot	81658056	97234734	57520350
real-pot-amp-c14	1	0.05	0	0	0	0	50	real	pot	99940653	18624887	13776333
real-pot-amp-c15	1	0.05	0	0	0	0	50	real	pot	52386275	37258624	4752186
real-pot-amp-c18	1	0.05	0	0	0	0	50	real	pot	45991544	30418848	10866515
real-pot-none-c02	0	0	0	0	0	0	50	real	pot	77736421	27992257	46723042
real-pot-none-c06	0	0	0	0	0	0	50	real	pot	74831174	8349127	83508906
real-pot-none-c12	1	0	0	0	0	0	50	real	pot	83816081	93832681	33667893
real-pot-odup-c03	0	0	0.05	0	0	0	50	real	pot	52364499	82437072	79633590
real-pot-odup-c12	1	0	0.05	0	0	0	50	real	pot	42856008	77382983	14786327
real-pot-odup-c13	1	0	0.05	0	0	0	50	real	pot	24529728	95781248	28086229
real-pot-odup-c15	1	0	0.05	0	0	0	50	real	pot	90438475	2325105	65980352
real-pot-odup-c17	1	0	0.05	0	0	0	50	real	pot	66707537	31861391	8928898
real-pot-pow-c02	0	0	0	0.05	0	0	50	real	pot	4866492	27585732	87897332
real-pot-pow-c08	0	0	0	0.05	0	0	50	real	pot	67391006	82484345	45170991
real-pot-pow-c12	1	0	0	0.05	0	0	50	real	pot	39012029	9475121	46410583
real-pot-pow-c17	1	0	0	0.05	0	0	50	real	pot	76875837	18092320	90140420
real-pot-rev-c00	0	0	0	0	0.05	0	50	real	pot	8560486	78845026	78505238
real-pot-rev-c10	1	0	0	0	0.05	0	50	real	pot	84079250	22092470	92190568
real-pot-rev-c18	1	0	0	0	0.05	0	50	real	pot	82359313	81599705	26713840
real-pot-swap-c00	0	0	0	0	0	0.05	50	real	pot	87683506	62472265	22477811
real-pot-swap-c10	1	0	0	0	0	0.05	50	real	pot	31385018	91016669	48086858
real-walking-all-c03	0	0.05	0.05	0.05	0.05	0.05	50	real	walking	81620090	55678259	43213408

file	cross	Mutation					pop	Soundfiles		Seed		
name	over	amp	dup	pow	rev	swap	size	env	pop			
real-walking-all-c12	1	0.05	0.05	0.05	0.05	0.05	50	real	walking	95868733	53497673	98268979
real-walking-all-c13	1	0.05	0.05	0.05	0.05	0.05	50	real	walking	94985393	46855106	89257279
real-walking-amp-c05	0	0.05	0	0	0	0	50	real	walking	46470305	89045927	19768258
real-walking-amp-c13	1	0.05	0	0	0	0	50	real	walking	56810202	54638494	88361678
real-walking-amp-c19	1	0.05	0	0	0	0	50	real	walking	85323445	53884602	482975
real-walking-none-c03	0	0	0	0	0	0	50	real	walking	41970244	32715030	2641051
real-walking-none-c11	1	0	0	0	0	0	50	real	walking	94051600	46858333	29717490
real-walking-none-c18	1	0	0	0	0	0	50	real	walking	77456518	59618054	84635423
real-walking-odup-c09	0	0	0.05	0	0	0	50	real	walking	7842666	36312596	53561788
real-walking-odup-c13	1	0	0.05	0	0	0	50	real	walking	25505288	53654085	47634726
real-walking-pow-c03	0	0	0	0.05	0	0	50	real	walking	12027725	80916550	63141824
real-walking-pow-c16	1	0	0	0.05	0	0	50	real	walking	85951546	97702841	10897582
real-walking-rev-c08	0	0	0	0	0.05	0	50	real	walking	90641287	36528672	93498366
real-walking-rev-c13	1	0	0	0	0.05	0	50	real	walking	20882875	70655511	72136262
real-walking-swap-c01	0	0	0	0	0	0.05	50	real	walking	38945743	92003710	57641190
real-walking-swap-c13	1	0	0	0	0	0.05	50	real	walking	72076545	74629505	53728498
sin-cricket-all-c09	0	0.05	0.05	0.05	0.05	0.05	121	sin	cricket	10290787	37078298	14464031
sin-cricket-all-c10	1	0.05	0.05	0.05	0.05	0.05	121	sin	cricket	22021096	61617730	93057612
sin-cricket-all-c16	1	0.05	0.05	0.05	0.05	0.05	121	sin	cricket	95573998	62113887	81319306
sin-cricket-amp-c01	0	0.05	0	0	0	0	121	sin	cricket	84173340	69717348	53195315
sin-cricket-amp-c12	1	0.05	0	0	0	0	121	sin	cricket	31052576	8947155	32073837
sin-cricket-amp-c18	1	0.05	0	0	0	0	121	sin	cricket	9583036	7018222	42356259
sin-cricket-none-c03	0	0	0	0	0	0	121	sin	cricket	72497053	26886426	1827308
sin-cricket-none-c10	1	0	0	0	0	0	121	sin	cricket	85441714	13234719	34992552
sin-cricket-odup-c03	0	0	0.05	0	0	0	121	sin	cricket	68604889	3314835	86036712

file	cross		Mutation				pop	Soundfiles			Seed	
name	over	amp	dup	pow	rev	swap	size	env	pop			
sin-cricket-odup-c10	1	0	0.05	0	0	0	121	sin	cricket	84033338	92412737	51223641
sin-cricket-odup-c14	1	0	0.05	0	0	0	121	sin	cricket	37045275	32995013	77869013
sin-cricket-odup-c18	1	0	0.05	0	0	0	121	sin	cricket	18109375	66535803	87517083
sin-cricket-pow-c04	0	0	0	0.05	0	0	121	sin	cricket	10671095	68415736	59256078
sin-cricket-pow-c08	0	0	0	0.05	0	0	121	sin	cricket	35667402	85005920	33400719
sin-cricket-pow-c11	1	0	0	0.05	0	0	121	sin	cricket	17595390	25228129	81412574
sin-cricket-rev-c01	0	0	0	0	0.05	0	121	sin	cricket	1710033	4764036	34332879
sin-cricket-rev-c18	1	0	0	0	0.05	0	121	sin	cricket	40428435	96071581	87606012
sin-cricket-swap-c04	0	0	0	0	0	0.05	121	sin	cricket	72047325	8239887	84021148
sin-cricket-swap-c13	1	0	0	0	0	0.05	121	sin	cricket	50395151	37418699	94566575
sin-gliss-all-c08	0	0.05	0.05	0.05	0.05	0.05	121	sin	gliss	62332919	37742164	47631375
sin-gliss-all-c18	1	0.05	0.05	0.05	0.05	0.05	121	sin	gliss	42651542	67585543	91138182
sin-gliss-amp-c00	0	0.05	0	0	0	0	121	sin	gliss	34311317	12868891	23646358
sin-gliss-amp-c10	1	0.05	0	0	0	0	121	sin	gliss	4540798	23353663	74576982
sin-gliss-none-c03	0	0	0	0	0	0	121	sin	gliss	8539392	26826133	43458395
sin-gliss-none-c14	1	0	0	0	0	0	121	sin	gliss	84605147	97165579	96334820
sin-gliss-odup-c00	0	0	0.05	0	0	0	121	sin	gliss	31009067	71168811	60374689
sin-gliss-odup-c10	1	0	0.05	0	0	0	121	sin	gliss	35910392	55255715	55161513
sin-gliss-pow-c01	0	0	0	0.05	0	0	121	sin	gliss	61232786	76696010	34677990
sin-gliss-pow-c04	0	0	0	0.05	0	0	121	sin	gliss	87031563	7070204	99096499
sin-gliss-pow-c10	1	0	0	0.05	0	0	121	sin	gliss	37538453	32199979	43170531
sin-gliss-rev-c05	0	0	0	0	0.05	0	121	sin	gliss	43297847	88607918	50303411
sin-gliss-rev-c10	1	0	0	0	0.05	0	121	sin	gliss	27940058	405358	55499139
sin-gliss-swap-c00	0	0	0	0	0	0.05	121	sin	gliss	36752663	33868454	4325051
sin-gliss-swap-c10	1	0	0	0	0	0.05	121	sin	gliss	68846867	95438892	40191789

file	cross	Mutation					pop	Soundfiles		Seed		
name	**over**	**amp**	**dup**	**pow**	**rev**	**swap**	**size**	**env**	**pop**			
sin-pot-all-c01	0	0.05	0.05	0.05	0.05	0.05	121	sin	pot	14946318	85554586	24762415
sin-pot-all-c12	1	0.05	0.05	0.05	0.05	0.05	121	sin	pot	39958683	27408269	10839040
sin-pot-amp-c07	0	0.05	0	0	0	0	121	sin	pot	5273416	52397497	97883436
sin-pot-amp-c12	1	0.05	0	0	0	0	121	sin	pot	25857557	59442030	96374595
sin-pot-none-c00	0	0	0	0	0	0	121	sin	pot	5853758	3926270	37462255
sin-pot-none-c12	1	0	0	0	0	0	121	sin	pot	49859643	39148298	76837801
sin-pot-odup-c01	0	0	0.05	0	0	0	121	sin	pot	48879475	24158109	56857881
sin-pot-odup-c10	1	0	0.05	0	0	0	121	sin	pot	76030106	3850298	22991648
sin-pot-pow-c00	0	0	0	0.05	0	0	121	sin	pot	87206989	12140878	48097045
sin-pot-pow-c10	1	0	0	0.05	0	0	121	sin	pot	10046245	78232029	95912541
sin-pot-rev-c00	0	0	0	0	0.05	0	121	sin	pot	37408108	15759161	8532662
sin-pot-rev-c10	1	0	0	0	0.05	0	121	sin	pot	46017991	81349736	49963971
sin-pot-swap-c00	0	0	0	0	0	0.05	121	sin	pot	44763443	15753728	33118369
sin-pot-swap-c10	1	0	0	0	0	0.05	121	sin	pot	33360297	70318962	83885325
sin-walking-all-c00	0	0.05	0.05	0.05	0.05	0.05	121	sin	walking	60722572	6468833	2211629
sin-walking-all-c10	1	0.05	0.05	0.05	0.05	0.05	121	sin	walking	60146545	57064958	297429
sin-walking-amp-c00	0	0.05	0	0	0	0	121	sin	walking	63825899	61132210	3022893
sin-walking-amp-c10	1	0.05	0	0	0	0	121	sin	walking	60693691	93945241	46137233
sin-walking-none-c00	0	0	0	0	0	0	121	sin	walking	33967361	26721567	7393620
sin-walking-none-c16	1	0	0	0	0	0	121	sin	walking	45065889	23231067	95044177
sin-walking-odup-c00	0	0	0.05	0	0	0	121	sin	walking	21886248	69409744	40998086
sin-walking-odup-c10	1	0	0.05	0	0	0	121	sin	walking	36181162	50215026	40033807
sin-walking-pow-c00	0	0	0	0.05	0	0	121	sin	walking	16553345	80700622	64059692
sin-walking-pow-c10	1	0	0	0.05	0	0	121	sin	walking	65837098	75481384	45262870
sin-walking-rev-c00	0	0	0	0	0.05	0	121	sin	walking	23849623	19350275	8074900

file	cross	Mutation					pop	Soundfiles		Seed		
name	**over**	**amp**	**dup**	**pow**	**rev**	**swap**	**size**	**env**	**pop**			
sin-walking-rev-c10	1	0	0	0	0.05	0	121	sin	walking	72902741	41567337	10818256
sin-walking-swap-c00	0	0	0	0	0	0.05	121	sin	walking	42489583	53342590	31217547
sin-walking-swap-c15	1	0	0	0	0	0.05	121	sin	walking	61157717	47291385	22003178

Bibliography

[BD03] Rodney Berry and Palle Dahlstedt. Artificial life: Why should musicians bother? *Contemporary Music Review*, 22(3):57–67, 2003.

[BE95] John A. Biles and William G. Eign. Genjam populi: Training an iga via audience-mediated performance. *Proceedings of the International Computer Music Conference*, pages 347–348, 1995.

[Bey99] Peter Beyls. Evolutionary strategies for spontaneous man-machine interaction. *Proceedings of the International Computer Music Conference*, pages 171–174, 1999.

[Bil94] John A. Biles. Genjam: A genetic algorithm for generating jazz solos. *Proceedings of the International Computer Music Conference*, pages 131–137, 1994.

[Bil98] John A. Biles. Interactive genjam: Integrating real-time performance with a genetic algorithm. *Proceedings of the International Computer Music Conference*, pages 232–235, 1998.

[Bil03] John A. Biles. Genjam in perspective: A taxonomy for ga music and art systems. *Leonardo*, 36(1):43–45, 2003.

[Bou00] Richard Boulanger, editor. *The Csound book: perspectives in software synthesis, sound design, signal processing, and programming.* MIT Press, Cambridge, Mass., 2000.

[Bow90] Peter Bowcott. High level control of granular synthesis using the concepts of inheritance and social interaction. *Proceedings of the International Computer Music Conference*, pages 50–52, 1990.

[BR96] Stephen Brooks and Brian J. Ross. Automated composition from computer models of biological behavior. *Leonardo Music Journal*, 6:27–31, 1996.

[Bro04] Andrew R. Brown. An aesthetic comparison of rule-based and genetic algorithms for generating melodies. *Organised Sound*, 9(2):191–197, 2004.

[BV97] A. Burton and T. Vladimirova. Genetic algorithm utilising neural network fitness evaluation for musical composition. In *ICANNGA 97 Abstracts*, 1997.

[BV99] Anthony R. Burton and Tanya Vladimirova. Generation of musical sequences with genetic technique. *Computer Music Journal*, 23(4):59–73, Winter 1999.

[CYH96] San-kuen Chan, Jennifer Yuen, and Andrew Horner. Discrete summation synthesis and hybrid sampling-wavetable synthesis of acoustic instruments with genetic algorithms. *Proceedings of the International Computer Music Conference*, pages 49–51, 1996.

[Dah01] Palle Dahlstedt. Creating and exploring huge parameter spaces: Interactive evolution as a tool for sound composition. *Proceedings of the International Computer Music Conference*, pages 235–242, 2001.

[Deg97] Bruno Degazio. The evolution of musical organisms. *Leonardo Music Journal*, 7:27–33, 1997.

[DN01] Palle Dahlstedt and Mats G. Nordahl. Living melodies: Coevolution of sonic communication. *Leonardo*, 34(3):243–248, 2001.

[Fed03] Francine Federman. The nextpitch learningg classifier system: Representation, information theory and performance. *Leonardo*, 36(1):47–50, 2003.

[FF99] Angela Fraser and Ichiro Fujinaga. Toward real-time recognition of acoustic musical instruments. *Proceedings of the International Computer Music Conference*, pages 175–177, 1999.

[Fuj96] Ichiro Fujinaga. Exemplar-based learning in adaptive optical music recognition system. *Proceedings of the International Computer Music Conference*, pages 55–56, 1996.

[FV94] Ichiro Fujinaga and Jason Vantomme. Genetic algorithms as a method for granular synthesis regulation. *Proceedings of the International Computer Music Conference*, pages 138–141, 1994.

[Gar00] Ricardo A. Garcia. Towards the automatic generation of sound synthesis techniques: Preparatory steps. In *Audio Engineering Society 109th Convention*, Los Angeles, September 22-25 2000.

[Gar01] Guillermo Garcia. A genetic search technique for polyphonic pitch detection. *Proceedings of the International Computer Music Conference*, pages 435–438, 2001.

[GB91] P. M. Gibson and J. A. Byrne. Neurogen: Musical composition using genetic algorithms and cooperating neural networks. In *Proceedings of the Second International Conference on Artificial Neural Networks*, pages 309–313, London, 1991. IEE.

[HA95] Andrew Horner and Lydia Ayers. Harmonization of musical progressions with genetic algorithms. *Proceedings of the International Computer Music Conference*, pages 458–484, 1995.

[HA96] Andrew Horner and Lydia Ayers. Common tone adaptive tuning using genetic algorithms. *Journal of the Acoustic Society of America*, 100(1):630–640, July 1996.

[HB96] Andrew Horner and James Beauchamp. Piecewise-linear approximatin of additive synthesis envelopes: A comparison of varous methods. *Computer Music Journal*, 20(2):72–95, Summer 1996.

[HBH92] Andrew Horner, James Beauchamp, and Lippold Haken. Wavetable and fm matching synthesis of musical instrument tones. *Proceedings of the International Computer Music Conference*, pages 18–21, 1992.

[HBH93] Andrew Horner, James Beauchamp, and Lippold Haken. Machine tongues xvi: Genetic algorithms and their application to fm matching synthesis. *Computer Music Journal*, 17(4):17–29, Winter 1993.

[HBP93] Andrew Horner, James Beachamp, and Norman Packard. Timbre breeding. *Proceedings of the International Computer Music Conference*, pages 396–398, 1993.

[HC95] Andrew Horner and Ngai-Man Cheung. Genetic algorithm optimization of additive synthesis envelope breakpoints and group synthesis parameters. *Proceedings of the International Computer Music Conference*, pages 215–221, 1995.

[HG91] Andrew Horner and David E. Goldberg. Genetic algorithms and computer-assisted composition. *Proceedings of the International Computer Music Conference*, pages 479–482, 1991.

[Hol92] John H. Holland. *Adaptation in natural and artificial systems : an introductory analysis with applications to biology, control, and artificial intelligence*. MIT Press, Cambridge, Mass., 1992.

[Hor94] Damon Horowitz. Generating rhythms with genetic algorithms. *Proceedings of the International Computer Music Conference*, pages 142–143, 1994.

[Hor95a] Andrew Horner. Envelope matching with genetic algorithms. *Journal of New Music Research*, 24:318–341, 1995.

[Hor95b] Andrew Horner. Wavetable matching synthesis of dynamic instruments with genetic algorithms. *Journal of the Audio Engineering Society*, 43(11):916–931, 1995.

[Hor96] Andrew Horner. Double-modulator fm matching of instrument tones. *Computer Music Journal*, 20(2):57–71, Summer 1996.

[Hor03] Andrew Horner. Auto-programmable fm and wavetable synthesizers. *Contemporary Music Review*, 22(3):21–29, 2003.

[HR96] Dominik Hörnel and Thomas Ragg. Learning musical structure and style by recognition, prediction and evolution. *Proceedings of the International Computer Music Conference*, pages 59–62, 1996.

[Jac95] Bruce L. Jacob. Composition with genetic algorithms. *Proceedings of the International Computer Music Conference*, pages 452–455, 1995.

[Joh99] C. G. Johnson. Exploring the sound-space of synthesis algorithms using interactive genetic algorithms. In G. A. Wiggins, editor, *Proceedings of the AISB Workshop on Articial Intelligence and Musical Creativity*, Edinburgh, 1999.

[Joh03] Colin G. Johnson. Exploring sound-space with interactive genetic algorithms. *Leonardo*, 36(1):51–54, 2003.

[JP98] Brad Johanson and Riccardo Poli. GP-music: An interactive genetic programming system for music generation with automated fitness raters. In John R. Koza, Wolfgang Banzhaf, Kumar Chellapilla, Kalyanmoy Deb, Marco Dorigo, David B. Fogel, Max H. Garzon, David E. Goldberg, Hitoshi Iba, and Rick Riolo, editors, *Genetic Programming 1998: Proceedings of the Third Annual Conference*, pages 181–186, University of Wisconsin, Madison, Wisconsin, USA, 22-25 1998. Morgan Kaufmann.

[Koz92] J.R. Koza. *Genetic Programming; On the Programming of Computers by Means of Natural Selection*. The MIT Press, 1992.

[LK94] Pauli Laine and Mika Kuuskankare. Genetic algorithms in musical style oriented generation. *International Conference on Evolutionary Computation*, pages 858–862, 1994.

[Mag03a] Cristyn Magnus. Evolving waveforms with genetic algorithms. Master's thesis, University of California, San Diego, 2003. http://crca.ucsd.edu/∼cmagnus/research.html.

[Mag03b] Cristyn Magnus. run12. for quad tape, 2003. performed November 10, 2003; stereo mix on UCSD Music Department CD 2005.

[Mag03c] Cristyn Magnus. run5. for stereo tape, 2003. performed November 6, 2003.

[McI94] Ryan A. McIntyre. Bach in a box: The evolution of four part baroque harmony using the genetic algorithm. *International Conference on Evolutionary Computation*, pages 852–857, 1994.

[MH03] James Mandelis and Phil Husbands. Musical interaction with artificial life forms: Sound synthesis and performance mappings. *Contemporary Music Review*, 22(3):69–77, 2003.

[MKT03] Eduardo Reck Miranda, Simon Kirby, and Peter M. Todd. On computational models of the evolution of music: From the origins of musical taste to the emergence of grammars. *Contemporary Music Review*, 22(3):91–111, 2003.

[MMZG99] Jonatas Manzolli, Artemis Moroni, Fernando Von Zuben, and Ricardo Gudwin. An evolutionary approach to algorithmic composition. *Organised Sound*, 4(2):121–125, 1999.

[MMZG00] Artemis Moroni, Jônatas Manzolli, Fernando Von Zuben, and Ricardo Gudwin. Vox populi: An interactive evolutionary system for algorithmic music composition. *Leonardo Music Journal*, 10:49–54, 2000.

[NW03] Paul Nemirovsky and Richard Watson. Genetic improvisation model. In *EvoWorkshops*, pages 547–558, 2003.

[PDBB97] John Polito, Jason M. Daida, and Tommaso F. Bersano-Begey. Musica ex machina: Composing 16th-century counterpoint with genetic programming and symbiosis. In Peter J. Angeline, Robert G. Reynolds, John R. McDonnell, and Russ Eberhart, editors, *Evolutionary Programming VI: Proceedings of the Sixth Annual Conference on Evolutionary Programming*, Indianapolis, Indiana, USA, 1997. Springer-Verlag.

[PSA+03] Alejandro Pazos, Antonino Santos, Bernardino Arcay, Julián Dorado, Juan Romero, and Jose Rodriguez. An application framework for building evolutionary computer systems in music. *Leonardo*, 36(1):61–64, 2003.

[Ral95] David Ralley. Genetic algorithms as a tool for melodic development. *Proceedings of the International Computer Music Conference*, pages 501–502, 1995.

[RV02] Janne Riionheimo and Vesa Välimäki. Parameter estimation of a plucked string synthesis model with genetic algorithm. *Proceedings of the International Computer Music Conference*, pages 283–286, 2002.

[SA94] Lee Spector and Adam Alpern. Criticism, culture, and the automatic generation of artworks. In *Proceedings of the Twelfth National Conference on Artificial Intelligence*, volume 1, pages 3–8, 1994.

[THG+93] Tapio Takala, James Hahn, Larry Gritz, Joe Geigel, and Jong Won Lee. Using physically-based models and genetic algorithms for functional composition of sound signals, synchronized to animated motion. *Proceedings of the International Computer Music Conference*, pages 180–183, 1993.

[Thy96] Kurt Thywissen. Genotator: An environment for investigating the application of genetic algorithms in computer assisted composition. *Proceedings of the International Computer Music Conference*, pages 274–277, 1996.

[Thy97] Kurt Thywissen. Evolutionary based algorithmic composition: A demonstration of recent developments in genotator. *Proceedings of the International Computer Music Conference*, pages 368–371, 1997.

[Thy99] Kurt Thywissen. Genotator: an environment for exploring the application of evolutionary techniques in computer-assisted composition. *Organised Sound*, 4(2):127–133, 1999.

[TL91] P. M. Todd and E. D. Loy, editors. *Music and Connectionism*. MIT Press, Cambridge, MA, 1991.

[TW99] Peter M. Todd and Gregory M. Werner. Frankensteinian methods for evolutionary music composition. In N Griffith and P. Todd, editors, *Musical Networks: Parallel Distributed Perception and Performance*, pages 313–339. MIT Press, 1999.

[VV93] Jarkko Vuori and Vesa Välimäki. Parameter estimatin of non-linear physical models by simulated evolution—application to the flute model. *Proceedings of the International Computer Music Conference*, pages 402–404, 1993.

[Was99] Rodney Waschka. Avoiding the fitness "bottleneck": Using genetic algorithms to compose orchestral music. *Proceedings of the International Computer Music Conference*, pages 201–203, 1999.

[WPPAT98] G. Wiggins, G. Papadopoulos, S. Phon-Amnuaisuk, and A. Tuson. Evolutionary methods for musical composition. In *Proceedings of the CASYS98 Workshop on Anticipation, Music and Cognition*, 1998.