AUTONOMOUS MUSIC VIA ARTIFICIAL EVOLUTION

Peter Velikonja

A DISSERTATION

PRESENTED TO THE FACULTY

OF PRINCETON UNIVERSITY

IN CANDIDACY FOR THE DEGREE

OF DOCTOR OF PHILOSOPHY

RECOMMENDED FOR ACCEPTANCE

BY THE DEPARTMENT OF

MUSIC

January 2004

© Copyright by Peter Velikonja, 2003. All rights reserved.

Contents

Abstract	iv
Part 1	1
Part 2	28
Part 3	42
Part 3	47
Appendix A	53
Appendix B	64

Abstract

The potential of a computer to compose original music using a quasi-autonomous method is explored. Using a genetic-programming technique, an artificial life environment controls an additive-synthesis audio engine. Emphasis is placed on the musical quality of results rather than on a particular research goal. The problem of human versus evolutionary time is discussed, as is the degree of control a composer may exercise over a nominally autonomous process.

We know from experience that computers need careful guidance to create even the simplest musical sounds; but the potential of computers has inspired many composers to use them as tools or even as active partners. Computers excel at repetition and numeric calculation – which is not surprising, as computer programs consist fundamentally of variable assignments and loops. It is not a simple matter to construct music from these building blocks. The formal constructs of programming languages do not translate naturally into musical syntax, and obtaining aural complexity from a computer is always a challenge.

Yet composers persist with this compelling notion: we know computers can work tirelessly on intricate problems, so we can imagine them creating a new kind of music – perhaps one not fully ruled by human logic. We may not understand the results; nonetheless we are curious.

The paper is in four parts, which are progressively *less* technical. Part 1 explains why artificial evolution should be a useful technique for automatic music composition. It explains why *frames* of digital audio are generated rather than musical notes or phrases; an overview of genetic programming is followed by test examples. Part 2 applies the technique to create musical sounds. Part 3 introduces methods a composer might follow to obtain musical variety. Part 4 describes two musical compositions created by the author, *Suite for Proteins*

and *Passacaglia Polymer C3*. An Appendix describes the C++ implementation. Audio and code examples are included on two CDs.

Part 1

Introduction

A screwdriver is a versatile tool. Used optimally to turn screws, it might also be used to shuck oysters, open a beer bottle, or, in the hands of an imaginative two year old child, it might be applied to a working electrical outlet. It provides a user with both flexibility and control – although perhaps in tasks not directly related to screw turning it leans harder toward flexibility rather than control. The two qualities are usually reciprocal, where greater control, for example, means less flexibility.

A computer program is a tool a composer can use to create music. As might be predicted, a program capable of creating music of great variety will be difficult to control, and a program that is easy to control can be expected to produce music of limited variety (and presumably quality). In practice, the composer at one extreme will accept arbitrary output from a program and call it music, while at the other extreme a composer might define tightly focused rules to obtain specific results. In this paper we examine a computer program that aspires to become a screwdriver: it leans decidedly toward the flexibility position, where a computer can exercise its inspiring processing power and enjoy some autonomy; guidance is needed, and a composer provides it. Yet through ceding much control we can hope to gain variety and to hear some pleasant surprises.

The goal, expressed in this program, is to produce a wide range of sounds when provided with only the most general guidelines. The sound should change over time; this rules out the use of any specific composition method. Something much more general is needed: a way to establish guidelines without specifying an *implementation*. To address that goal, let us consider genes within the context of evolution.

Richard Dawkins [1976] extends the Theory of Evolution proposed by Darwin to focus on the gene rather than the individual organism. Genes are carried by individuals, wherein their configuration determines an individual's *fitness* in an environment. Highly fit individuals are more likely to produce offspring than those with low fitness, allowing *genes* to persist across generations – the individual is a disposable carrying machine in this scenario. A process of *natural selection* favors genes that confer fitness over successive generations.

For an autonomous music system, this synopsis of evolution theory suggests the possibility of *evolving* a computer program from a set of guidelines. The program would, like an organism, carry genes needed to survive; but, like Dawkins, we do not care about individual programs -- we care about the genes. Through natural selection, successive generations of computer programs would attempt to meet musical guidelines, presumably with an increasing degree of fitness. Specific implementation of compositional logic would be expressed in the arrangement of genes; the composer's challenge is to provide useful guidelines.

A *genetic programming* technique described by Koza [1992] can be used to develop a population of computer programs. In each *generation* the population is evaluated for fitness, and the less fit programs are replaced by the offspring of the more fit. Over successive generations, a group of highly fit programs eventually emerge. Genetic programming, like evolution in the natural world, is a hugely inefficient and wasteful problem solving method. While it may not provide a single perfect solution to a problem, it can be relied upon to generate many *imperfect* solutions. These programs can be highly unstable -- as unused genes may suddenly be engaged by a change in the environment. Unacceptable in an engineering application, these faults might become assets when solving other kinds of problems.

An autonomous music program appears to be that kind of problem. Imagine a population of artificial organisms, each producing a unique sound in an attempt to survive. As the environment changes, the genetic makeup of the population shifts. The aggregate audio output

is in constant flux. A composer can be a gardener: either breeding exotics, letting the weeds take over, or finding beauty in combinations.

Scope

An artificial sound-emitting organism lives or dies based on periodic *evaluation*. Here a composer decides if an organism makes a "good" sound or not, defining guidelines through which a population of organisms is controlled. Organisms that make "good" sounds survive; the others do not.

But what is evaluated -- notes, phrases, or complete compositions? In most of the existing work on evolutionary music a note or short phrase is created by an individual organism; this material is evaluated by a human or by a software listener or *critic*. The preliminary advantage of using a human listener (who might be trusted to know what "good" is) is somewhat undone by lack of processing capability, as a human does not have enough time to audition and evaluate millions (or billions) of notes or phrases – a scale of operation required for meaningful evolution to take place. Consider for example the most humble species of the natural world, which has presumably undergone millions of years of comparable "evaluation." **GenJam**, by John A. Biles [1994], can succeed in relying on human evaluation because a formal musical structure (trading 32-bar solos in a jazz combo context) is fixed from the outset. Another effort, by Jeffrey Putnam, proposes to distribute a human-evaluation effort across the internet (http://www.t0.or.at/msguide/ai/genart.htm) by playing phrases for remote listeners and collecting their votes. This method could work if millions of listeners were willing to participate [the above link was inactive as of September 2003].

Machine evaluation is several orders of magnitude faster but unfortunately machines require some instruction on what "good" is, and a composer is left with the task of trying to define fitness criteria using computer code. In **VoxPopuli** Artemis Moroni and her co-

authors [2000] use fixed rules to reward “choirs” of organisms for exhibiting desired features of melody and consonance. In **Living Melodies**, Palle Dahlstedt and Mats Nordahl [2001] assign an organism a list of favorite notes. When another organism plays one of these favored notes the original organism is rewarded; we might call this a “game” method of evaluation. A paper by Peter M. Todd and Gregory M. Werner [1999] describes how a group of “female” organisms listen to the mating songs of “male” organisms and choose their mates based on a desired proportion of familiarity and innovation in the calls. This method of potentially great power benefits from the work done by biologists who study the natural world. Their paper also contains a coherent survey of recent developments in evolutionary music composition, and proposes the intriguing notion of co-evolving music critics in parallel with music-producing populations as a means of sustaining diversity in a musical output – a notion handled somewhat differently in this paper as will be seen in Part 3.

In all the work mentioned above evaluation takes place at the note or phrase level; the programs generate (and evaluate) musical *constructions*, a vocabulary made of notes, used to describe musical sounds. A more general and low-level approach is desired in this paper, so the musical sounds are generated (and evaluated) at the level of the audio synthesis *frame* or *grain*. It is hard to write computer code capable of distinguishing “good” pitch sequences from “bad” ones -- a problem regularly mentioned in the literature. Even if it can be done, it requires organisms to have musical knowledge – at the minimum they must own the concept of a “note” – which must be hard-wired in. This imposes from the outset a *discrete* (as in piano-keyboard-derived) musical model to the detriment of a *continuous* (as in singing) musical model. In a frame-based approach, where organisms do not require *any* musical knowledge, a continuous musical model can be supported, *and* it may be possible to write a very general, yet effective, evaluation method to obtain musical behavior. Ichiro Fujinara describes the potential of using genetic algorithms (a precursor of the genetic programming technique used in this paper) to control grain-based synthesis [1994], but his paper is too brief to offer any specific thoughts about evaluation.

A frame-based method also supports the desired evaluation intensity level. If a population of 1000 organisms is evaluated once per frame at a rate of 1000 frames per second the result is one million evaluations per second of audio output. Since thousands of individuals are born and die each second, this evaluation intensity allows a composer to audition results in an effective manner, as major evolutionary epochs may be heard in timely succession. Alternatively, if the population reaches evolutionary stasis the composer will hear it quickly. The composer listens at a very high level, where a small decision can dramatically influence the musical result. With this approach it may be possible to learn something about the most basic elements of musical construction.

Introduction to Genetic Programming

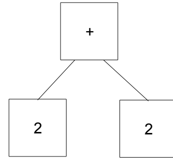
Genetic programming simulates a rudimentary evolution process. A population of programs is created using an arbitrary (i.e. pseudo-random) method. The programs are evaluated; weaker programs are discarded and replaced by the offspring of stronger-performing programs. The process is repeated until a significant proportion of the population performs well, the focus is on developing the gene pool rather than on creating individual high performers (known as *freaks*). The goal is to develop a population of highly fit programs.

The programs are tree structures made of *functions* and *terminals* -- their configuration within a tree can be thought of as genes. Functions have one or more inputs and a single output. Terminals have no inputs; they are outputs by definition. The input to a function may be another function, as these examples show.

In C:

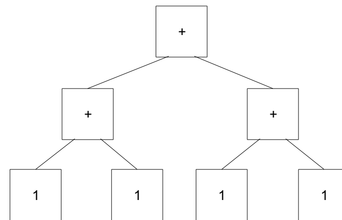
```
plus(2, 2);
```

As a diagram:



When evaluated, the result is 4. The next example also evaluates to 4:

```
plus(
     plus(1, 1),
     plus(1, 1));
```



The example shows how a function output may be an input to another function.

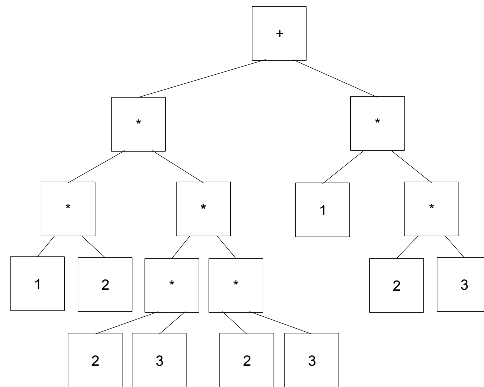
Rudimentary programs like these are assembled arbitrarily from a set of functions and terminals. Consider the following function and terminal sets:

F = { plus, minus, multiply }
T = { 1, 2, 3 }

A large number of programs can be generated from these sets. Let us create two to demonstrate how new *child* programs are created by mating two *parent* programs. Note here that we consider only how new programs are created through mating, not what the programs might do.

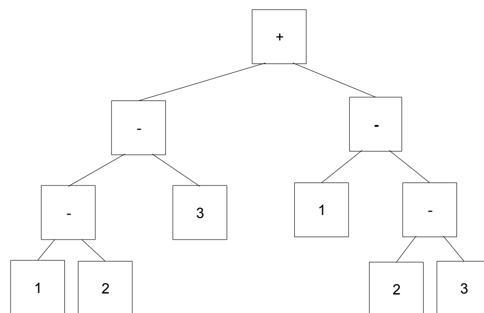
Parent A:

```
plus(
     multiply(
          multiply (1, 2)),
          multiply (
               multiply (2, 3),
               multiply (2, 3))),
     multiply (1,
          multiply (2, 3))));
```



Parent B:

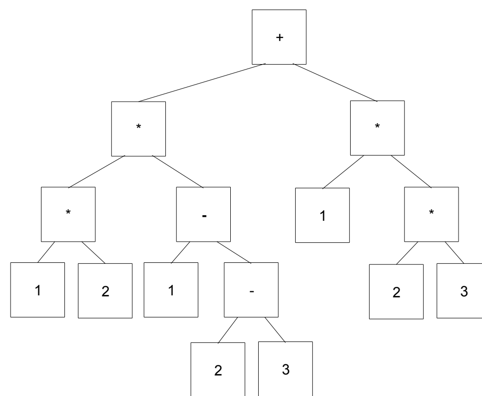
```
plus(
     minus(
          minus(1, 2)),
          3),
     minus(1,
          minus(2, 3))));
```



A *crossover* point is chosen at random in each parent program (shaded in the diagram, **bold** in the code). To mate these two programs we make a copy of each one, then swap subtrees (in the copies) at the crossover points. These are *children*, which in this example look like:

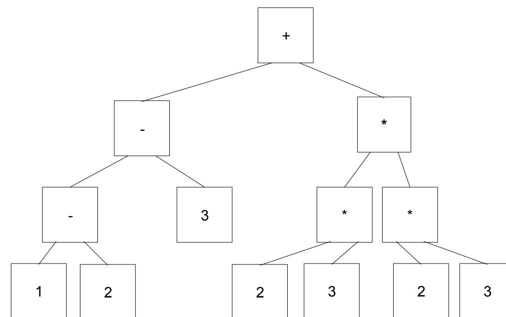
<u>Child 1</u>:

```
plus(
     multiply (
          multiply (1, 2)), minus(1,minus(2, 3)),
     multiply (1, multiply (2, 3))));
```



<u>Child 2</u>:

```
plus(
     minus(
          minus(1, 2)),3),
     multiply (
          multiply (2, 3), multiply (2, 3));
```



The crossover points are shown in the children here so the reader may more easily compare them with the parents.

As noted above, after the programs are generated, they are evaluated; weaker programs are discarded and are replaced with the offspring of the stronger programs. We have shown how these offspring are created through mating, now we can look at how programs are evaluated. We choose a problem with a known solution and attempt to arrive at that same solution using artificial evolution.

A Genetic Programming Example

The equation for determining the length of the hypotenuse of a right triangle is:

```
hypotenuse = sqroot(length1 * length1) + (length2 *
length2))
```

To evolve the equation, we first define a minimum set of functions and terminals:

F = { sqroot, plus, multiply }
T = { len1, len2 }

Next we define an evaluation function. This function should determine the fitness of a program to solve a defined problem. The problem we wish to solve is:

> "Given any two numbers greater than 1, return the sum of their squares."

Our evaluation function should generate numbers to represent two triangle lengths, pass them to the program being evaluated (called calculateHypotenuse() below) and calculate the deviation, if any, from the known correct result. We do this a number of times and accumulate the deviation, otherwise we will develop programs that

only work for specific pairs of numbers. The evaluation function should look like this pseudo-code:

```
evaluate()
{
	loop 100 times {
		/*
		 * choose two triangle leg lengths at random
		 */
		len1 = random()
		len2 = random()

		/*
		 * calculate and store the correct result
		 */
		correctResult = sqroot((len1 * len1) + (len2 * len2))

		/*
		 * pass the program being evaluated the two legs
		 * and get its result
		 */
		result = calculateHypotenuse(len1, len2)

		/*
		 * square the difference between the answer given
		 * and the correct answer and accumulate the error
		 */
		error += (result - correctResult) * (result - correctResult)
	}
}
```

After each program has been evaluated by this function we can sort the programs in order of their magnitude of error. A program with an accumulated error of zero is desired. After the programs have been sorted we can destroy the lower-performing half and repopulate it.

In a test, we create 1000 calculateHypotenuse() programs, evaluate them, sort, discard 500, replace those 500 with new offspring and

evaluate all the programs again. We repeat until the correct solution emerges and is dominant in the population; this takes 18 repetitions (generations). At an earlier stage, ten generations, we see the population still struggling with the problem. The accumulated error of the top ten programs at that point is:

[1] 2.174935
[2] 2.519883
[3] 2.673281
[4] 3.462669
[5] 20.260468
[6] 20.260468
[7] 3120.386016
[8] 3120.386016
[9] 3120.386016
[10] 3120.386016

The best performing calculateHypotenuse() of generation 10 is:

```
sqroot(
  plus(
    plus(
      mult(len1, len1),
      sqroot(len1)),
    plus(
      mult(len2, len2),
      sqroot(
        plus(
          mult(
          sqroot(
            sqroot(
  sqroot(len1))), len1), len2)))));
```

If we try it out with len1 = 3 and len2 = 4 the result is 5.427705 (5.0 is correct), and we can see the program has the basic elements needed to provide consistently correct results. It squares both `len1` (line 4) and `len2` (line 7) and adds them together (line 2) in some fashion; it performs a `sqroot()` (line 1) on the outgoing result. It contains extraneous code but with a little breeding this sort of thing will do.

After 18 generations we see a few programs perform correctly and we stop the evolution process. The top ten programs have these accumulated errors:

[1] 0.000000
[2] 0.000000
[3] 0.000000
[4] 0.000000
[5] 0.000000
[6] 0.000000
[7] 0.026456
[8] 0.091000
[9] 0.091000
[10] 0.318309

The first six programs perform perfectly (each has an accumulated error of zero). Each expresses the optimal solution and is optimally efficient – there is no extraneous code; the sum of all the functions and terminals used is 8; we say the tree has 8 *nodes*.

```
sqroot(
  plus(
    mult(len2, len2),
    mult(len1, len1)));
```

The seventh program returns 5.081338 when given triangle sides of 3 and 4 and looks like this:

```
sqroot(
  plus(
    mult(
      sqroot(plus(
                  mult(len2, len2),
                  sqroot(
      sqroot(plus(
                  len1, mult(len2,
                  sqroot(
                  sqroot(sqroot(
                  plus(len1, len1)))))))))), len2),
    mult(len1, len1)));
```

The eighth program returns 5.116752 when given triangle sides of 3 and 4 and looks like this:

```
sqroot(
  plus(
    mult(
      sqroot(
                plus(
                  mult(len2, len2),
                  sqroot(
                    plus(len2,
          sqroot(len2))))), len2),
    mult(len1, len1)));
```

In these two programs we see the correct solution buried amid junk code. While not perfect, these incorrect examples still manage to produce near-correct results. The remaining 992 programs in the population drift ever further from the optimal solution; the majority are completely useless. We can continue the evolution process until more programs exhibit perfect behavior, but in this example we are interested in obtaining a single solution to the problem; once it emerges there is no need to refine the population further.

Two issues remain to consider. The first is mutation. Mutation is an important part of implementing a genetic programming system. Early-stage populations may achieve moderate success using only a subset of needed functions and terminals, and may exclude the very ones needed to succeed fully. Mutation brings forward previously unused or hidden genes. In this implementation mutation is done during mating, where one parent from a selected pair is ignored and replaced by a freshly created individual.

Second, we performed the above example with a minimum set of functions and terminals. We knew the desired solution at the outset and were able to define sets containing only the needed items. But for other problems we do not expect to know the solution beforehand and we will have to include items in the function and terminal sets that are likely to be extraneous. If we *do* include extra functions or terminals the evolution process will take longer and in some cases may never

produce a solution. For example, we can repeat the test above and add one extra function, `divide()`, to the function set:

F = { sqroot, plus, multiply, divide }
T = { len1, len2 }

In this test the optimal solution takes 25 generations to emerge rather than 18. After twenty generations the best program is:

```
sqroot(
  plus(
    mult(
      plus(
        sqroot(
            divide(
                  divide(
                    divide(len2, len1),
                    plus(len2, len1)),
                plus(
                  sqroot(
                    divide(len1, len2)),
                  plus(len1,
                    plus(len2, len2))))), len2),
               len2),
                  mult(len1, len1)));
```

Considering the amount of junk code it carries, this program performs well. When passed 3 and 4 it returns 5.050425 and its overall error accumulation is 0.009708. The extraneous function in the set causes extra junk code to accumulate -- and the amount of junk increases exponentially as more functions and terminals are added to the set. As illustration, adding a second extraneous function, `minus()`, to the example set causes so much *bloat* that the computer runs out of resources after 430 generations without finding a solution. The best program at that point has 1333 nodes, which is wildly inefficient. In later examples we penalize bloated programs to discourage their crippling influence.

Making Music with Continuous Evolution

The method described above is called *generational evolution*; it is used when seeking a *single* (and best) solution to a problem. To make music we can expect to want *multiple* solutions – both the "correct" and "almost correct." These programs will need a context to make music in -- a virtual petri dish we can think of as an *environment*. Within it a *continuous evolution* process may be supported (see Banzhaf et al. [1998] for a survey of genetic programming techniques), where instead of evolving in lockstep sequence, programs are created, live, reproduce and die. We can think of the programs as *individuals* that exist in *time* -- an *artificial life* system.

To persist in time an individual needs state, which is its health or *score*. Healthy individuals (those with high scores) will reproduce; a score of zero will cause an individual to die. While the individual is alive it may make a sound. To make sounds, each individual will have an audio *frequency* at which it may produce a sine wave. The sine waves of the entire population will be combined. This method, additive synthesis, is a well known way to produce a rich composite. It is described by a number of authors on computer music, for example Dodge and Jerse [1985], and is considered a computationally expensive but high-quality synthesis method capable of realizing any sound.

We create a population of 1024 individuals (a number that current hardware can handle without exhausting memory or CPU limits). Each individual is an instantiation of a C++ class with these members:

an evaluation tree
a health score
a frequency

Before going further, let us consider what we have at hand. We wrapped the concept of an *individual* around a computer program. This individual is born, makes sound, mates and dies. We created a *population* of individuals which we might think of as an additive synthesis instrument with a large number of oscillators. We will

evaluate the population many times per second; we can think of a single evaluation of the entire population as a synthesis *frame*; the number of frames per second is the *frame rate*. If an individual plays a tone for one second at a frame rate of 100, it means the individual has survived 100 evaluations.

As an initial test, we ask the individuals to move from an initial (random) frequency to match a target frequency (1000 Hz.). This problem is the audio equivalent of the cart-centering problem described by Koza [1992]. We should hear, in a successful test, a transition from broadband noise to a unison at the target frequency.

We define a set of functions and terminals, not all of which will be needed in the optimal solution

F = { mult, divide, plus, minus }
T = { zero, one, two, three, four, five, six, seven, eight, nine, ten,
currentFrequency, targetFrequency }

The optimal tree is:

```
minus(targetFrequency, currentFrequency);
```

In other words, the program should subtract its current frequency from the target frequency. The difference will be added to the individual's current frequency. We see that of the four functions in our set, only `minus()` is actually needed in the optimal solution. Similarly, only `currentFrequency` and `targetFrequency` are needed from the terminal set. We don't need the other terminals or functions, but let's pretend we don't know the solution to the problem; we'll go ahead and cultivate a solution.

The population is evaluated in a loop; an evaluation of the entire population was called a frame above because of the audio synthesis perspective; from the point of view of the population, however, it is called a *day*. Every day costs an individual a fixed amount. In the natural world (plants for example) the amount is measured in *calories*; to survive, an organism must replace expended calories. We mimic

this with the score member of the C++ class. An individual begins life with a score of 100 points (1000 is the maximum), and every day it loses 2 points; it loses more as it ages (we want old age to kill even the successful individuals to prevent them from playing the same sound indefinitely) – so a completely unfit individual will die in less than 50 days. But an individual can be rewarded based on the proximity of its current frequency to the target frequency -- up to 4 points per day. We expect a successful individual to move its frequency close to the target frequency (where the reward will be greater than its daily cost) before running out of resources.

We can try both generational and continuous evolution to see how the results differ. Starting with a generational method, we create a population of individuals. For each generation we run the population for 50 days, destroy the weaker half and repopulate. It takes about 15 generations to gain an audible convergence around 1000 Hz. [sound example 1.0]. The example demonstrates why a generational method is useful for developing a gene pool but not for creating music. Each generation begins with the creation of 512 new individuals, most of which are unfit and generate noise. In the sound example, the frame rate is 50, so we hear each new generation at one-second intervals. It is instructive to listen closely to this example, noting the succession of generations.

It may also help to walk through this example in a detailed manner. First, let us examine the problem we wish to solve:

> "Move from current frequency to a desired (target) frequency."

The evaluation function for the problem looks like this pseudo-code:

```
evaluate()
{
	currentFrequency += myTree()
	difference = targetFrequency – currentFrequency
	if(difference == 0) reward = maximumReward
	else reward = maximumReward
				* (1.0 / (difference * difference))

	score += reward
}
```

In this function we add the result of the individual's tree to its current frequency, which could be any number. The difference between the current frequency and the target frequency is stored. If the difference is zero, the individual gains the maximum reward, otherwise it receives the maximum reward times the reciprocal of the difference squared; if the difference is large the reward will be vary small, but close misses will receive reasonable rewards.

In example 1.0 we first hear individuals attempt an incrementation strategy (heard as *glissando*). In other words, the tree of most individuals looks like this:

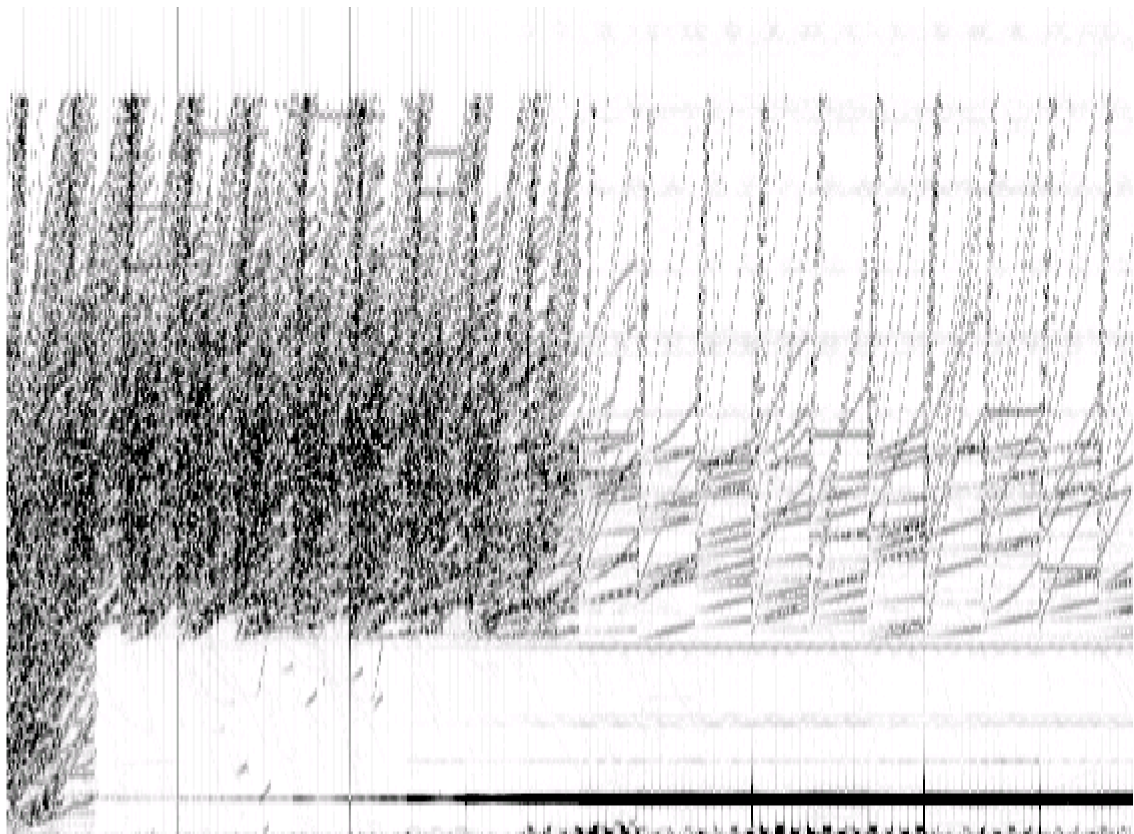
```
myTree()
{
	return 1;
}
```

At 50 frames per second, incrementing its frequency each frame, an individual with this tree will produce a glissando -- as long as it lives. The problem for this individual is that if it lives long enough to reach the target frequency where it will get a big reward, it will then fly past the target back into the land of poor rewards and die there of starvation.

By the third generation we can hear the incrementation strategy has mostly died out, and we hear the introduction of a subtraction strategy that will eventually dominate. The tree will look like this:

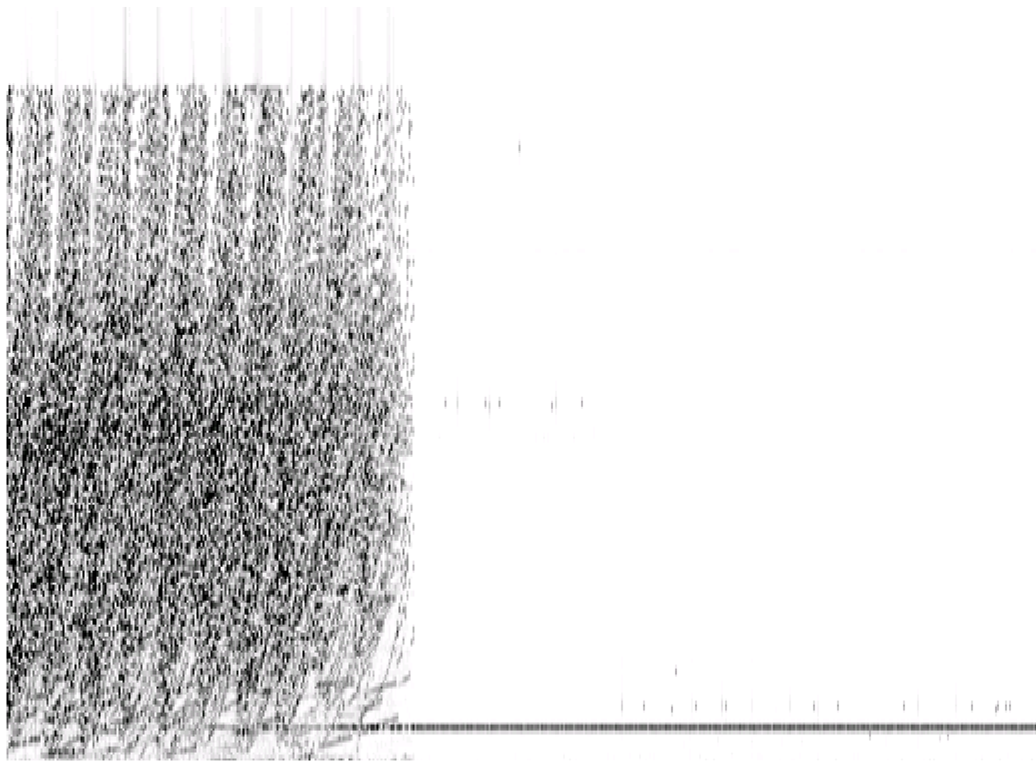
```
myTree()
{
     return targetFrequency - currentFrequency;
}
```

Using this strategy the individual zooms directly to the target frequency, even if the target frequency is moving. Over successive generations we hear the steady growth of this strategy until it is audibly dominant; the glissando gene persists, but in a dimished capacity.



Spectrogram of sound example 1.0, which shows segmentation from generational evolution. The thick black horizontal line low in the display is 1000 Hz.

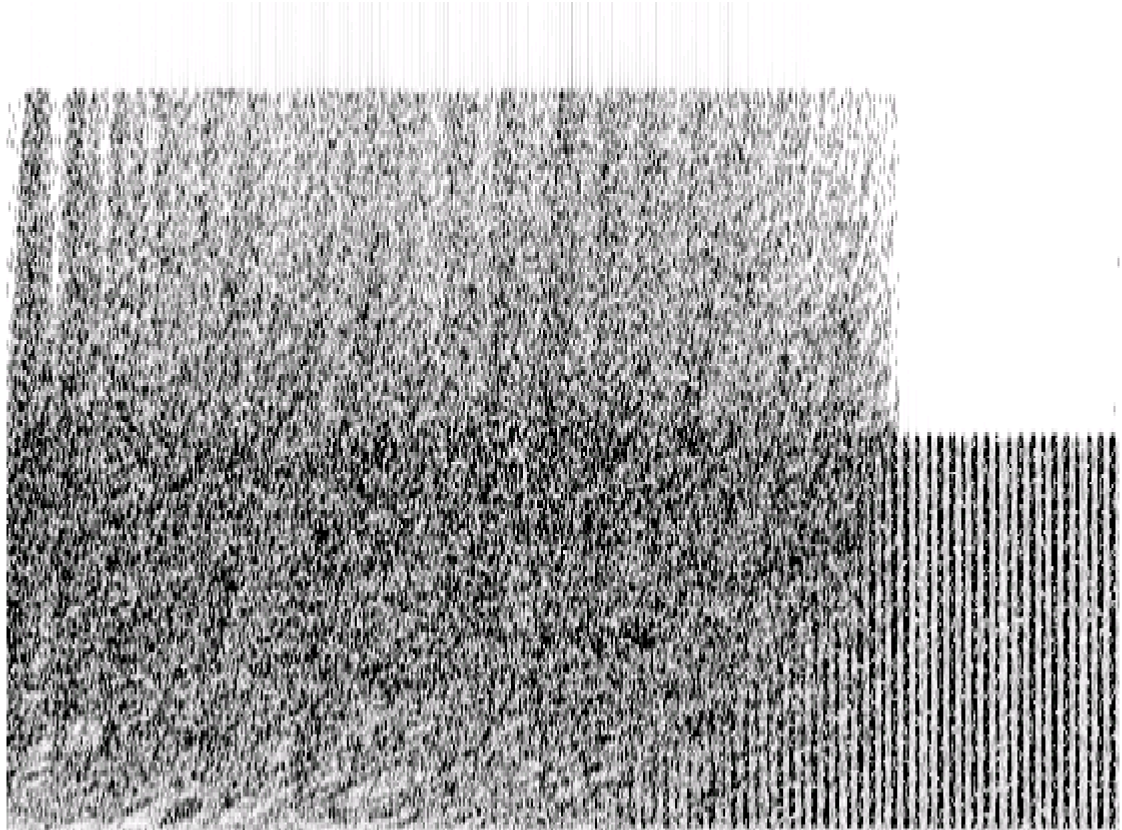
Next we try continuous evolution [sound example 1.1]. Here, individuals are immediately replaced as they die; there are no formal generations. In the sound example we again hear the glissando strategy being succeeded by the subtraction strategy; we also hear, at the outset, individuals dying and being replaced in large groups (as in the previous example). The effect diminishes as the coincidence of dying/replacement diminishes. As in the previous example, some errant individuals attempt to play arbitrary frequencies even after the general population has converged around the target frequency. We begin to see how music might emerge from the system.



Spectrogram of sound example 1.1, which begins with strong vertical segments that soften, then disappear. When continuous evolution populations identify a high-fitness gene they converge upon it rapidly.

By simulating evolution we can develop a population of computer programs to solve a simple problem. We have been able to converge to a single frequency; can we converge to a moving frequency? Sound example 1.2 shows that we can.

We next learn to match amplitude; if a population can generally match frequency and amplitude it will be capable of creating a wide variety of sounds. In sound example 1.3 the same gene is selected for as in the frequency tests, but the result is applied to amplitude. We hear the population converge to a loud-soft-loud-soft alternation. Since frequency is not selected for, individuals produce arbitrary frequencies.



Spectrogram of sound example 1.3, showing the emergence of broad-band pulses.

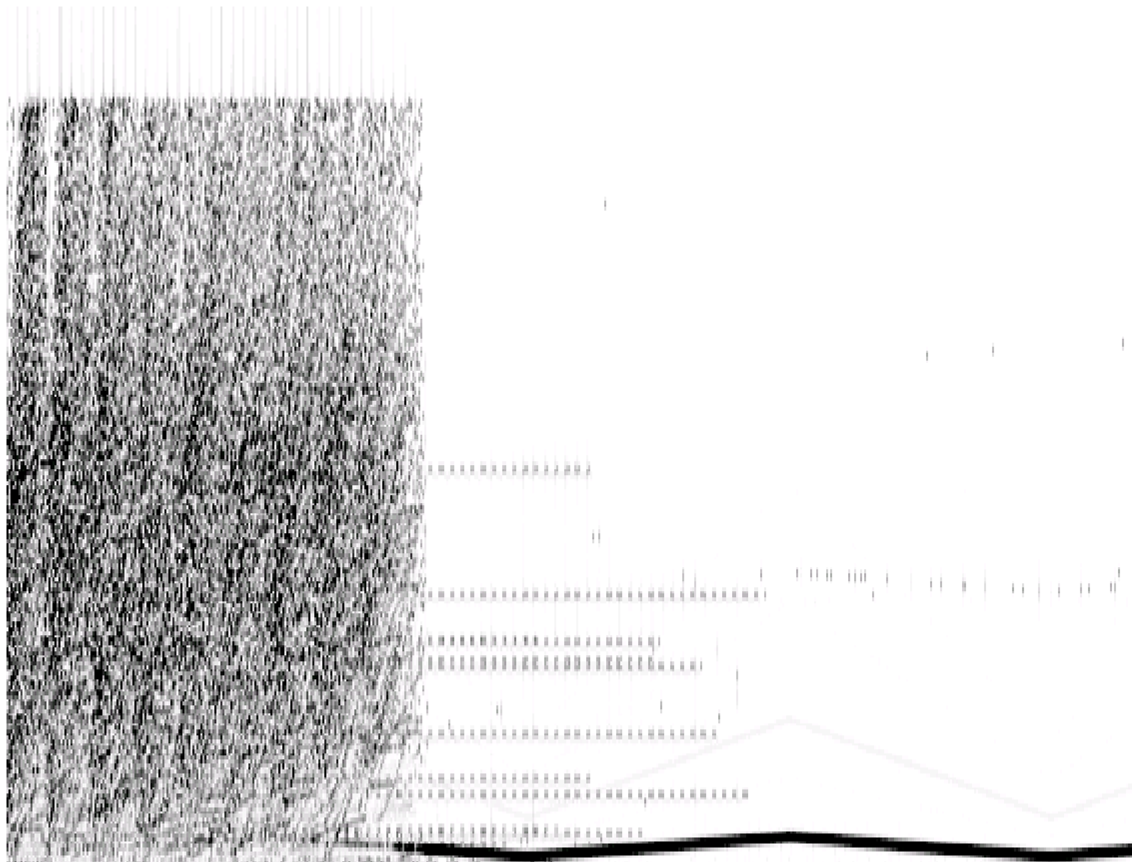
Finally, we select for frequency and amplitude together. Since a tree returns a single value we will evaluate each tree twice, setting a flag beforehand to indicate whether we are evaluating for frequency or amplitude. The code we hope to evolve looks like this:

```
state = listen_for_frequency;
frequency += myTree(state);

state = listen_for_amplitude;
amplitude += myTree (state);
```

Where `myTree()` is defined as:

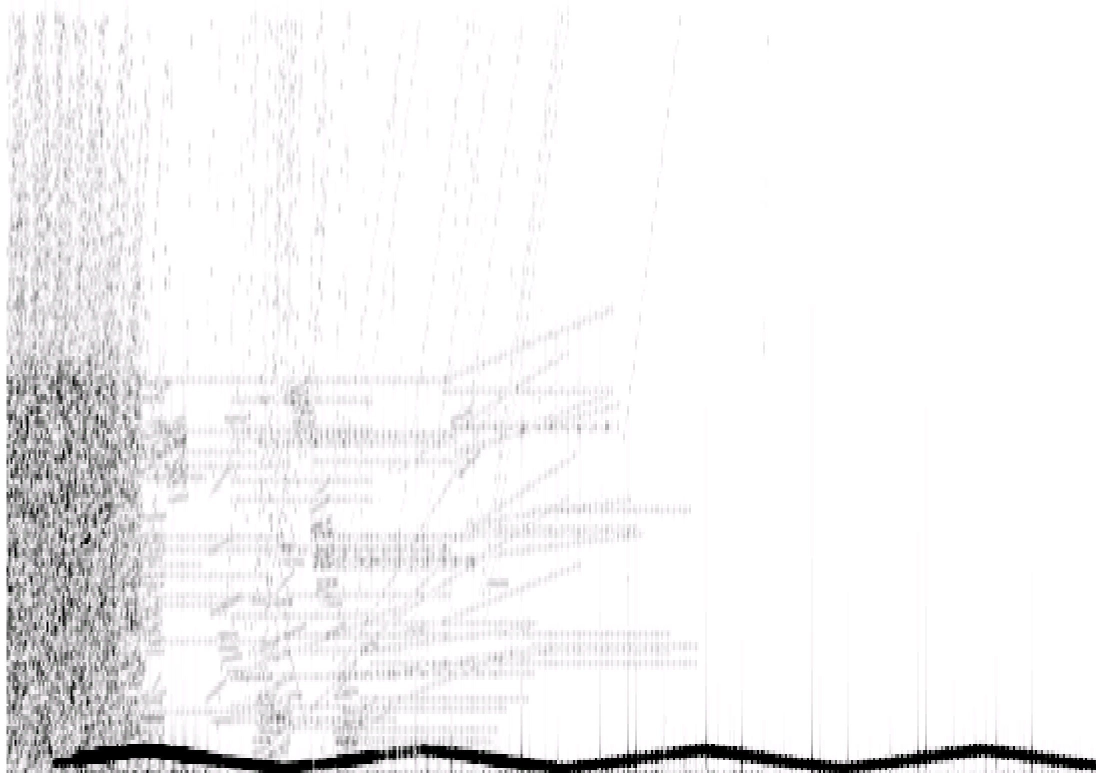
```
myTree(state) {

     if(state == listen_for_frequency)
          return ((target_frequency) - (frequency));
     else if(state == listen_for_amplitude)
          return ((target_amplitude) - (amplitude));

}
```

It takes a fair amount of computing to evolve the above code because a successful evaluation tree needs to contain genes for both pitch and amplitude. An individual will not succeed unless both genes are present. If the evaluation function does not reward the desired genes equally, one will tend to dominate. We hear this in sound example 1.4, where amplitude and frequency genes coexist for a while before frequency eventually wins.



Spectrogram of sound example 1.4; we see frequency and amplitude coexist until the frequency genes come to predominate.

Finally, once the reward function is tuned properly we hear the desired result in sound example 1.5.



Spectrogram of sound example 1.5; showing desired convergence of both frequency and amplitude.

Summary

Evolution can be simulated on a computer, and a computer program that generates music within such a simulation might operate indefinitely and, to some degree, autonomously. Attracted by the idea of creating self-sustaining populations of sound-emitting creatures, we performed some tests to explore the possibility of defining general guidelines and obtaining musical results.

Creating these guidelines is a challenge. Happily, experiments that go awry may produce entertaining results. These initial results hint at the musical potential of artificial evolution, but they were difficult to

obtain. Evolution is a slow and inefficient process. There may not be time (in the human time scale) to obtain desired results. This program will need a helping hand if any worthwhile music is to be derived from it.

References

Banzhof, Wolfgang; Nordin, Peter; Keller, Robert E.; Francone, Frank D., Genetic Programming ~ An Introduction. On the Automatic Evolution of Computer Programs and Its Applications. San Francisco: Morgan Kaufmann Publishers, Inc., 1998.

Dahlstedt, Palle and Nordahl, Mats G., Living Melodies: Coevolution of Sonic Communication. Leonardo Music Journal, 34:3, 2001. See also: http://www.design.chalmers.se/projects/art_and_interactivity/living-melodies

Dawkins, Richard, The Selfish Gene. Oxford: Oxford University Press, 1976.

Dodge, Charles and Jerse, Thomas A., Computer Music. New York: Schirmer Books, 1985.

Biles, John A., GenJam: A genetic algorithm for generating jazz solos. In *Proceedings of the 1994 International Computer Music Conference*. San Francisco: International Computer Music Association. See also: http://www.it.rit.edu/~jab/GenJam.html

Fujinaga, Ichiro, Genetic Algorithms as a Method for Granular Synthesis Regulation. In *Proceedings of the 1994 International Computer Music Conference*. San Francisco: International Computer Music Association. See also: http://www.music.mcgill.ca/~ich/research/ICMC94_paper.html

Griffith, Gregory N. and Todd, Peter M. Frankenseinian Methods for Evolutionary Music Composition. In Griffith, N. and Todd, P. M. [editors], Musical Networks: Parallel Distributed Perception and Performance. Cambridge: MIT Press, 1999.

Koza, John R., Genetic Programming: on the programming of computers by means of natural selection. Cambridge: MIT Press, 1992.

A. Moroni, J. Manzolli, F. Von Zuben, and R. Gudwin. Vox populi: An interactive evolutionary system for algorithmic music composition. Leonardo Music Journal, 10:pp. 49–54, 2000. See also: http://www.ici.org.br/invencao/papers/Manzolli.htm

Part 2

Introduction

In Part 1 the sounds we heard were only tests, unlikely to be wanted in a musical composition. In this part we attempt to create more musical sounds. To do that, we take a less technical approach while engaging in some musical exploration; we exercise musical judgment in an attempt to create attractive sounds.

Getting Started

Some assumptions may help provide focus. The goal is to create sounds suitable for use in a musical composition using continuous evolution, so a primary assumption is that we are not detached observers of phenomena but breeders of ephemeral flora. We have our musical preferences and we expect to exercise them.

We know from experience that interesting sounds are non-periodic (let us permit "interesting" to mean "interesting to the author"); producing them (in the context of signal processing) usually involves nonlinear feedback. In the context of this paper, feedback means that organisms are somehow connected to each other; we find a biological parallel in the notion of *swarms* where organisms are capable of simple interaction.

But caution is warranted. Computer programs are not "aware" and the simulation of awareness on computers has a rocky history. Dreyfus [1993] cites many examples to debunk claims made by Artificial Intelligence researchers, and his arguments are convincing. We should look, therefore, to the most simple interactions, hoping that interesting results will emerge. Interesting, or *complex*, behavior derived from many simple interactions is known as *emergent behavior*. Cellular Automata, artificial swarms and other systems belong in this category; a good survey is found in Stan Franklin's book *Artificial Minds* [1995]. Although not directly related to the

present work, it is worthwhile to note Eduardo Miranda's book, *Composing Music with Computers* [2001], which surveys a number of ways artificial intelligence has been applied to music composition. Miranda's program, CAMUS, composes music using two parallel cellular automata to control pitch and MIDI instrument number.

A cellular automaton is normally thought of as a square within a two-dimensional matrix. Each such "cell" has a *state*, which is on or off; it has eight adjacent neighbors just like itself. The cell queries these neighbors to learn their state, then applies a rule to determine its own state based on the results of its queries. The interaction consists solely of queries regarding the states of neighbors.

This is not difficult to imagine. But in this paper we have been generating populations of programs, not cells with binary states. How does a computer program respond to another *program*? In this implementation, an individual exists within a virtual machine as an instantiation of a C++ class. It lives in a static array; each "day" it is evaluated and its score is adjusted. When a program "starves" (because its score is 0) it is destroyed; its replacement is a product of mating by two more successful programs. Individuals are not independent computer processes but objects in an array, and they can be assigned references (pointers) to others. The simplest way to create relations between individuals is to assign each individual a pointer to another individual.

Heretofore knowing nothing beyond its own routine of daily costs and gains, an individual now has a link to another individual, from whom it might gain information. Since individuals are stored in a linear array, it is easy to link each individual to its neighbor. The last individual in the array can point to the first, completing a circle. Going around the circle and rewarding "good" frequency relationships should contribute to producing a desired aggregate sound.

Amplitude is correlated with health; a healthy individual is loud, a waning one diminishes in volume. Tones will fade in and out as individuals grow and die. This is a helpful indicator, albeit a musical limitation.

As noted earlier each individual is evaluated repeatedly at a specified frame rate; this evaluation is called a *day*. A cost is subtracted from the health of an individual each day and a reward is calculated. Individuals that gain a reward larger than the daily cost will live for a long time. Cost and reward (and punishment) are calculated and applied in an *evaluation function*.

Consider now an evaluation function designed to produce a musical sound. The function expresses a desire that:

1. An individual's frequency should not be the same as that of its linked neighbor.
2. An individual's frequency should remain stable – *glissandi* are discouraged.
3. An individual's frequency should be *close to* (but not exactly) an integer multiple or division of its linked neighbor's frequency.

The first two requirements are safeguards to protect us from dead-ends: if an organism's frequency is the same as its linked neighbor's frequency we run the risk of developing a population of individuals all producing the same frequency; if *glissandi* are permitted freely we cannot hope to build stable tones (we remove this penalty later to hear its effect).

The third desire proposes that if the frequency of each individual is related by integer proportion (but not *exact* integers, in order to obtain a less perfect result) to the frequency of its linked neighbor, a rich harmonic composite will be the outcome. We do not know what will emerge, as the function only defines an expressed interest in tones built on harmonic proportions, but we await eagerly the spontaneously beautiful sounds that are sure to spill forth this our first experiment.

Two more musical decisions are defined:

1) At birth each organism is assigned an initial frequency of 440 Hz., and

2) An organism may not make sound until it is 40 “days” old (to reduce noise in the output signal).

The decisions are made; we listen to sound example 2.0, where a 440 Hz. tone quickly turns into a timbral refraction exercise. Can we use this for musical purposes? Yes, we like it.

Let’s play with this instrument for a while. We notice it produces a different result each time it runs, demonstrated nicely by sound example 2.1(a and b) the output of a single program run twice in quick succession. Similar in substance, the details are different. If this were a program for computing income taxes that would be bad; for music composition purposes it is good.

A modest change in settings can have a significant effect. These are some of the settings:

Daily cost calculation

The daily cost for an individual is 1.0 points, to which is added a cost for program complexity (the number of nodes in a tree) and a cost for age. Complexity is penalized in order to reduce program bloat – the tendency of these programs to grow very large without actually doing anything (similar to junk DNA). The penalty is mild and it is applied only to keep the population from overwhelming the memory and computational resources of a computer. Age is penalized in order to refresh the population with new organisms and new sounds. The calculation used is:

```
Cost = 1.0 + (nodes_in_tree * COMPLEXITY_COST)
           + (age_in_days * AGE_COST)
```

By changing the values for the constants COMPLEXITY_COST and AGE_COST we determine an individual’s daily cost.

Daily reward calculation

A flattened exponential curve is used to reward desired frequencies. The width of the flattened portion can be adjusted, which influences the lifecycle of each organism. An organism that lives longer produces a longer tone. The width is adjusted with a multiplier constant WIDTH.

Maximum and minimum frequency

A sampling rate of 44100 samples per second provides high audio fidelity. The highest frequency supported is about 22,000 Hz., the lowest is 0. But different results emerge when other values are used for minimum and maximum frequency.

Initialization frequency

New organisms are assigned an initial frequency, which, we will hear, influences the composite result significantly.

Minimum age to produce sound

A new organism may be totally unfit, in which case it will die soon after it is created. In most instances, these unfit organisms produce noise. To minimize the overall noise level, a minimum age before which the organism may not produce sound may be specified. If it survives to that age it is presumably not entirely unfit and should be heard. The natural world has many examples of producing numerous offspring that do not survive infancy, or that develop functionality only in adolescence.

Intervals

A list of frequency proportions (intervals) is defined. The author is primarily interested in creating quasi-harmonic spectra; accordingly, integer proportions are specified, as in 2:1, 3:1, 4:1 and so forth, as well as their reciprocals 1:2, 1:3, 1:4. The number of intervals in the

series should have a decided effect on the output. A value of 10 would create this list of ten intervals and ten reciprocals:

1:1, 1:2, 1:3, 1:4, 1:5, 1:6, 1:7, 1:8, 1:9, 1:10,
(1:1), 2:1, 3:1, 4:1, 5:1, 6:1, 7:1, 8:1, 9:1, 10:1

A Series of Musical Sounds

The initial example (sound example 2.0) was created with these settings:

Settings for Example 2.0	
COMPLEXITY_COST	0.0001
AGE_COST	0.001
WIDTH	50
Maximum frequency	19500
Minimum frequency	15
Initialization frequency	440.0
Minimum age to produce sound	40
Intervals	10

Sound example 2.1(a and b) had these settings:

Settings for Example 2.1	
COMPLEXITY_COST	0.0001
AGE_COST	0.001
WIDTH	32
Maximum frequency	12000
Minimum frequency	12
Initialization frequency	440.0
Minimum age to produce sound	40
Intervals	4

The chief difference between these first three examples is in their frequency distributions. When the maximum frequency is allowed to approach the limit supported by the sampling rate, the population eventually moves to a very high frequency range that can be uncomfortable to listen to for extended periods.

Let's take the settings from example 2.0 and change them one at a time.

Sound example 2.2 re-uses the settings from example 2.0; the result is similar, yet its details are different. From a musical point of view this is most welcome, as it supports the goal (from Part 1) of providing general guidelines and obtaining varied results. Removing COMPLEXITY_COST should have only minor audible effect but it will certainly slow down the computation. The average number of nodes (the sum of functions and terminals in an evaluation tree) in example 2.2 reached 90 by its conclusion.

Settings for Example 2.3	
COMPLEXITY_COST	0.0
AGE_COST	0.001
WIDTH	50
Maximum frequency	19500
Minimum frequency	15
Initialization frequency	440.0
Minimum age to produce sound	40
Intervals	10

In sound example 2.3 COMPLEXITY_COST is set to 0; it sounds the same as example 2.2. The number of nodes reached a high of 115; but it is not clear what, if anything, the extra complexity contributed, although 2.3 has a lower frequency distribution (which the author prefers). Further runs attained complexity levels of 77 and 90. For the moment it appears that COMPLEXITY_COST has little influence on audible output in these short examples. It should gain in importance as a braking mechanism on longer runs.

Consider now a cost for age. As indicated above, even a beautiful sound will lose its attraction if it goes on indefinitely. But we can try two tests: one with no penalty for age, one with a high penalty. In example 2.4 the penalty is removed. We should hear less change in the overall sound, as fit individuals will now be able to produce sound indefinitely.

Settings for Example 2.4	
COMPLEXITY_COST	0.0001
AGE_COST	0.0
WIDTH	50
Maximum frequency	19500
Minimum frequency	15

Initialization frequency	440.0
Minimum age to produce sound	40
Intervals	10

Indeed, sound example 2.4 is smooth, as component frequencies do not fade away to be replaced by others. Note that the example reaches a steady state midway through which would likely continue indefinitely with only very small changes.

In example 2.5 the penalty for age is increased. Here individuals should die in adolescence without having produced a stable tone of any length. Dying quickly, they will be replaced by new individuals that will also never get a chance to move away from their initial frequency. Accordingly we should hear quiet volatility never effectively departing from 440 Hz. The result should be quiet because the individuals never reach full maturity (in these exercises amplitude is still related to score). If the penalty is too high the result will be silence, as the individuals will all die before they are old enough to make a sound.

Settings for Example 2.5	
COMPLEXITY_COST	0.0001
AGE_COST	0.01
WIDTH	50
Maximum frequency	19500
Minimum frequency	15
Initialization frequency	440.0
Minimum age to produce sound	40
Intervals	10

Sound example 2.5 is attractive, but the population never evolves.

The WIDTH setting has a similar effect, since it affects the basic cost/gain premise of each individual. If the individuals succeed too easily they never die and are never replaced. If success is too difficult to attain the population never evolves. If WIDTH is low, the individual must perform very good frequency matching, as being only close yields insufficient reward. If high, the individual does not need

to match as well. After some experimentation, a minimum WIDTH value is found that will allow the population to evolve.

Settings for Example 2.6	
COMPLEXITY_COST	0.0001
AGE_COST	0.001
WIDTH	22
Maximum frequency	19500
Minimum frequency	15
Initialization frequency	440.0
Minimum age to produce sound	40
Intervals	10

In sound example 2.6 a minimum value (22) is explored to see how tuning might affected. The intervals heard may have greater purity. Note that a narrower WIDTH makes it harder for individuals to succeed, so the result is relatively quiet. After listening, it is not clear if the pitch relationships are purer, but it may be manifested in the high-frequency “shimmering” heard. In any case, the tougher survival requirements means that individuals that may have previously masked this quality are no longer present.

Setting the maximum frequency lower than the highest value permitted will certainly make a difference. A previous example (example 2.1) was heard with maximum frequency set to 12000. Let’s try 8000.

Settings for Example 2.7	
COMPLEXITY_COST	0.0001
AGE_COST	0.001
WIDTH	50
Maximum frequency	8000
Minimum frequency	15
Initialization frequency	440.0
Minimum age to produce sound	40
Intervals	10

The result is good; many details that were previously obscured in an area beyond our practical range of hearing are now audible.

If the original maximum frequency is restored and the minimum frequency is raised we hear sound example 2.8.

Settings for Example 2.8	
COMPLEXITY_COST	0.0001
AGE_COST	0.001
WIDTH	50
Maximum frequency	21500
Minimum frequency	1000
Initialization frequency	440.0
Minimum age to produce sound	40
Intervals	10

As expected, the frequency distribution is high, much of it beyond practical hearing range.

We next experiment with the initialization frequency. Sound example 2.9 demonstrates the result of setting this to a random value between the minimum and maximum.

Settings for Example 2.9	
COMPLEXITY_COST	0.0001
AGE_COST	0.001
WIDTH	50
Maximum frequency	21500
Minimum frequency	15
Initialization frequency	random
Minimum age to produce sound	40
Intervals	10

The example serves largely to indicate the importance of initialization in a feedback chain.

Setting the minimum age to produce sound is now explored. In sound example 2.10 it is set low to learn if the output will indeed be noisy.

Settings for Example 2.10	
COMPLEXITY_COST	0.0001
AGE_COST	0.001
WIDTH	50
Maximum frequency	21500
Minimum frequency	15
Initialization frequency	440.0
Minimum age to produce sound	2
Intervals	10

It is noisier, but in a good way. The noise creates a "guitar-feedback" quality that gives it dimension and power.

The number of intervals is now adjusted. We heard the result of using fewer intervals in example 2.1, but other parameters were also changed in that example. Let us try setting Intervals to 5.

Settings for Example 2.11	
COMPLEXITY_COST	0.0001
AGE_COST	0.001
WIDTH	50
Maximum frequency	21500
Minimum frequency	15
Initialization frequency	440.0
Minimum age to produce sound	40
Intervals	5

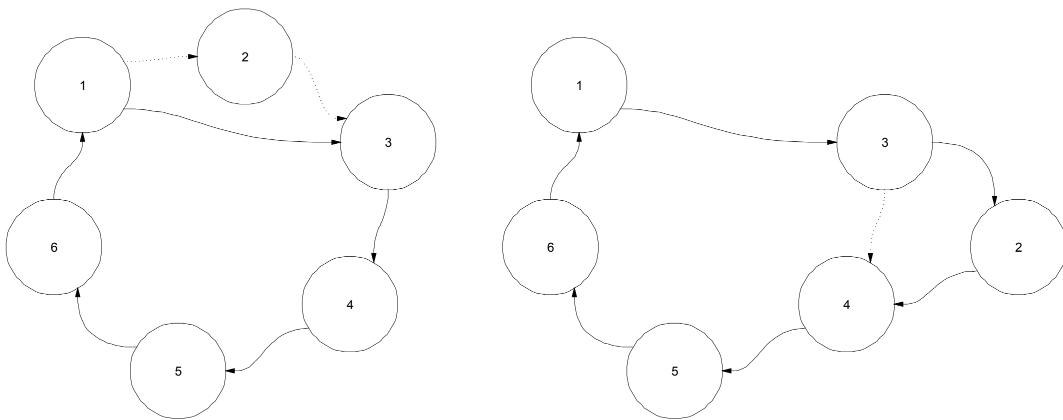
This was another good test. It will take a population longer to drift away from its initialization frequency when it uses a reduced set of intervals, but the result is less harmonically diffuse than previous examples, and we hope these settings would ultimately take the population somewhere harmonically interesting.

As promised earlier, the penalty for glissandi is now removed. Other than this change, the settings for sound example 2.12 are the same as for 2.0.

Settings for Example 2.12	
COMPLEXITY_COST	0.0001
AGE_COST	0.001
WIDTH	50
Maximum frequency	21500
Minimum frequency	15
Initialization frequency	440.0
Minimum age to produce sound	40
Intervals	10

The example demonstrates that the penalty is a good idea. The frequencies move about too much to create tones.

As the result of a happy accident, linked neighbors were reset at ten-second intervals, which produced the gong-like tones heard in sound example 2.13. This points the way to investigating relationships between individuals. Up to now, the population has been connected in a single unbroken chain. Individual number one always pointed to individual number two, for example. But other configurations are possible and are perhaps desirable. Here is new experiment: at death an individual is unlinked from the circle; when a new individual is created it can point to its parent.



In example 2.14 a dead individual (number 2) is removed from the circle and a new one inserted before its parent (number 4)

This makes a nice little difference in the output, as heard in sound example 2.14, but more importantly the *linked-list* technique provides a method for breaking up the single large chain into several smaller ones. This is attempted in sound example 2.15 with ten independent chains. We can hear from this example a more stratified frequency distribution as each independent chain evolves its own technique for survival. Pointing to the parent at birth also provides a handy way of setting the initial frequency of a new individual, since it can use its parent's frequency. According to rule, it will have to leave that frequency quickly, but this provides a method by which a 440 Hz. tonal center can be escaped more quickly (hear sound example 2.16). Note in this example that it takes over two minutes for the population to find successful survival strategies, as indicated by an increase in overall amplitude.

Finally, does leaving a population to evolve for a much longer period eventually produce sounds much different than those heard at the outset? In this implementation it does not. The population reaches a comfortable survival point and stays there, presumably because more sophisticated survival strategies are needed. A more dynamic environment is the probable next step toward obtaining more dynamic results. While sound example 2.17 is pleasant to listen to (over 36 minutes long) it is more or less the same from beginning to end. Part 3 looks at ways to increase variety.

Summary

The goal in Part 2 was to learn if sounds suitable for use in a musical composition could be obtained using the artificial evolution method explained in Part 1. They can, as shown by a number of attractive sound examples. Small changes in the general guidelines given to the system influence the results in ways we can often predict beforehand. A worthwhile composition could be made with the instrument developed here by varying its control settings over time.

References

Dreyfus, Hubert L., What Computers Still Can't Do, A Critique of Artificial Reason. Cambridge: MIT Press 1993 (originally What Computers Can't Do, 1972).

Franklin, Stan, Artificial Minds. Cambridge: MIT Press, 1995.

Miranda, Eduardo Reck, Composing Music with Computers. Oxford: Focal Press, 2001.

Part 3

Introduction

Artificial evolution can be used to create musical sounds. But musical sounds alone do not make interesting music. More human intervention is needed.

Interesting Music

Interesting means only "of moderate interest to the author," a fairly low threshold, as a mother loves even her ugly children. Evolution is a slow process where nothing happens for long periods. Genetic changes are a response to environmental pressure; a stable environment means a static gene pool. In a musical context, a stable environment is most likely to produce a *drone*. A beautiful drone is interesting to listen to, but a steady diet of it is musically incomplete. A changing environment, applying evolutionary pressure, should force greater variety to emerge.

But First, Drones

Consider a zero-sum strategy where a table of frequencies is borrowed from and repaid by individual organisms. The table holds a value between 0 and 1 for all audible frequencies. The environment maintains a *target frequency* which each individual attempts to find because a large reward is associated with doing so. An individual requests a frequency; the value for that frequency is raised slightly in the table, and the value for the target frequency is lowered. The environment returns the value associated with the requested frequency (between 0 and 1) which is the reward to the individual. If the requested frequency is the same as the target frequency or any integer multiple of it, the maximum reward is returned. As the target frequency is decremented continually, its value soon falls to zero and a new target frequency is calculated (the frequency in the table with

the highest current value). This zero-sum strategy limits the resources available to an individual; if a requested frequency is depleted the individual receives no reward. Those who learn to find the target frequency will survive.

An early success with this method is example 3.0a in which the strategy heard is a glissando – which rewards poorly but reliably. In the excerpt, we hear the strategy change; the sound becomes violent and monophonic. In another excerpt (example 3.0b) we hear the two strategies coexisting. The spectrogram, below, shows the strategy shift of example 3.0a (figure 3.0a).

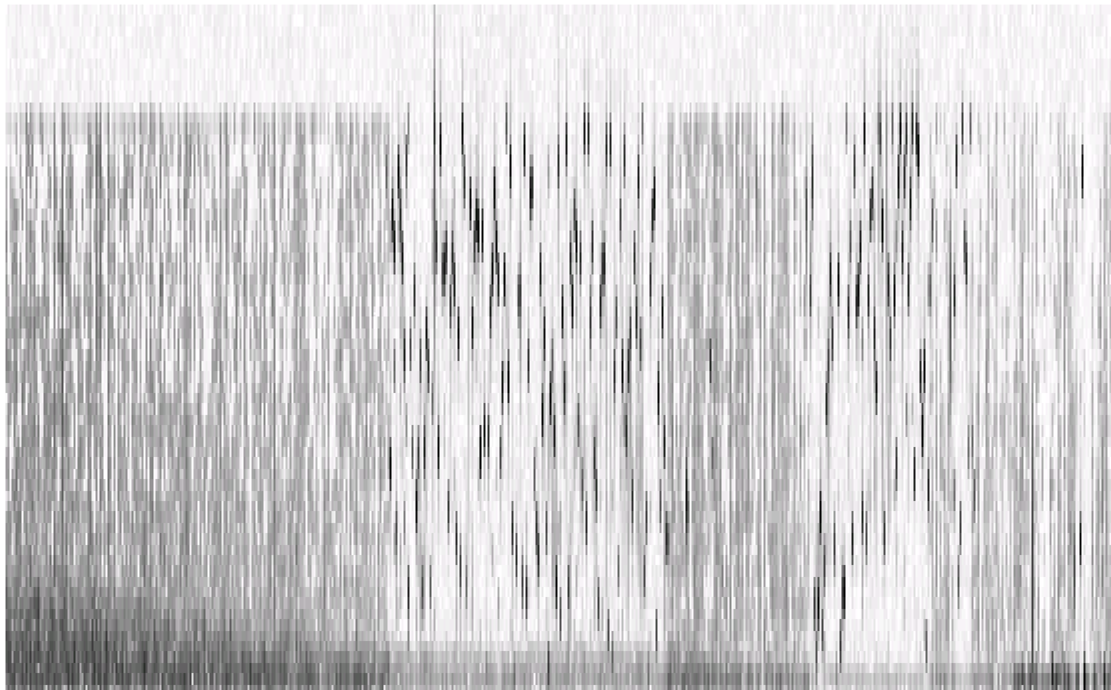


Figure 3.0a, spectrogram of (audio) example 3.0a

If the target frequency remains stable the result is a drone. We can reward individuals for playing overtones of the target frequency, and we can choose new target frequencies that are multiples (or factors) of the target. Example 3.1 demonstrates this; also, the *glissandi* heard in so many previous examples mostly die out when the population

suddenly discovers a better survival strategy, namely, they discover the existence of a function `getTargetFrequency()` which has been available from the outset but was ignored in favor of blind searching.

In the remaining examples, individuals are required to stick with a chosen frequency once they have begun playing it, as we have had our fill of glissandi. Any requests for integer multiples of the target frequency are rewarded highly, to encourage the emergence of an overtone series. Some beautiful drones emerge, as in example 3.2.

Some interesting sounds emerge after the initial values of the frequency table are adjusted. Drones are consistently created when the initialization values are very low; as the reward for finding the target frequency is much higher, frequencies unrelated to the target are not much heard. When the initial values are somewhat higher, individuals receive nominal rewards for exploring out-of-target areas. If new targets are simple winners (the frequency in the table with the highest value) we hear an almost instrumental result, a strange double-bass solo full of harmonics, *ponticello*, string crossings and *pizzicato*. Two examples are worth examining in some more detail.

In the first, example 3.3, observe that by rewarding frequencies that are integer multiples of the target both vertical and horizontal relationships (timbres and intervals) are created. This goes to the heart of the thesis proposed in this paper – that fundamental musical constructions will emerge from the simplest rules. We hear dominant-tonic relationships, for example, because the 3:2 proportion is rewarded by the evaluation function. Accordingly, individuals popularize frequencies (in the table) that are directly proportional to the target. Chosen by plurality, these frequencies become the next targets.

Next, in example 3.4, the amount of increment/decrement to the frequency table is smaller. Accordingly, the harmonic rhythm is slower and the notes stretch out longer. The overall effect is more atmospheric, with a less instrumental (physical) sound. At 2:43 a new

strategy appears that dispenses with "notes" and we hear a sustained wash until the end.

Composing with the Environment

The examples to this point have been created within a closed system; other than the initialization of the frequency table no composer has intervened to disrupt the environment – the frequency table is modified only by the population. Composer intervention should produce more dynamic musical results.

If the frequency table is initialized at 0.9;, then decremented slightly more than is incremented (i.e. decrement = 0.00011 and decrement = 0.0001), the frequency table will be entirely depleted over time. In example 3.5 we hear a good variety of frequencies and rhythms at the outset and the piece ends in a drone. This is music, created efficiently with a minimum of direction.

Next, frequency table values are shifted periodically up or down. In example 3.6 and 3.7 we shift values (up or down) by one every time a new target frequency is selected. In example 3.6 the shift is upward and an upward frequency sweep is head. In example 3.7 the shift is downward and a downward frequency shift is heard.

In example 3.8 a "swap-neighbors" method is applied to the table when a new target is selected. An intriguing rich sound is the result of this chronic perturbation.

These techniques give us what we need to compose a piece of music, heard in example 3.9. Here is the "score" of the piece:

> Initialize the frequency table to 0.9 for all frequencies.
> Set the first target frequency to 440.
> Use "swap-neighbors" to modify the frequency table.
> At 150 seconds, change to the "shift-down" method.
> At 280 seconds, set all values in the frequency table to 0.35, producing the double-bass effect.

At 400 seconds, set all values in the frequency table to 0.95.

The evaluation function for an individual is:

Select a frequency.
If the frequency selected is optimally rewarded, play it (do not evaluate further).
Play the frequency until death, procreate whenever possible.

Example 3.9 qualifies as interesting music (as defined above); the instructions (in the "score" above) are simple. The frequency table provides the spectral complexity, the individuals that manipulate it are simple.

Summary

The goal in this part was to graduate from "musical sounds" to "interesting music" which was accomplished with a modest set of general instructions called "composing with the environment."

Part 4

Introduction

In an attempt to obtain better musical variety, even more human intervention may be needed. Two compositions, *Suite for Proteins*, and *Passacaglia Polymer C3*, are described, and the dilemma of human involvement in a nominally autonomous process is discussed.

Variety

More variety can be obtained by modifying the evaluation function. In this implementation the evaluation function not only rewards an individual, it also schedules a frequency for audio synthesis; it can be responsible for determining the amplitude envelope of an individual's synthesized tone.

Until now, amplitude has been coupled to an individual's health; the tones fade in as the individual gains strength and fades out as it dies of old age – the correlation is good as a diagnostic, but it fails to provide variety. Amplitude can be decoupled from strength. In audio example 4.0 each individual is given a "play" switch which is turned on once it finds the target frequency (or a relative). The envelope onset is a linear ramp, the length of which is determined by the age of the individual; an older individual will have a longer onset. The onset peak amplitude is still coupled to the health of the individual. The remainder of the envelope is an exponential decay which is the same across the population. We hear in this example the beginnings of a slow pulse caused by group of healthy individuals who, after the decay portion of their envelope, are healthy enough to play again.

The evaluation function can be used to reward specific frequencies or frequency ranges, like a filter. An example will be discussed below in the *Suite for Proteins*.

To obtain more clearly rhythmic results, we can restrict onsets to specific times. In example 4.1 a 3/4 meter is accomplished by restricting the onsets of two-thirds of the population to beat 1; the remaining third may begin on beat 3. Dividing the population unequally provides a rough starting point for the strong and weak beats we hear, but as the individuals are not required to play on their designated beat, some shifts of emphasis are heard.

These three methods: assigning an envelope, filtering, and applying a rhythmic grid, are used to compose a *Suite for Proteins*.

Suite for Proteins

The suite has seven sections, each with a unique evaluation function. The function used in sections 1, 2, 3 and 6, focuses on applying an amplitude envelope. Section 4 follows a pre-composed harmonic progression. The functions in sections 5 and 7 apply a pulse.

In section 1 the individuals stay close to an A 440 initialization frequency. This is common when individuals are initialized to an audible frequency; an individual may be totally unfit for survival, but it may manage briefly to contribute its initialization frequency to the mix before dying. Its replacement (perhaps also totally unfit) may do the same. For this reason individuals are initialized to a subaudio frequency (normally 0) which is heavily penalized in the evaluation function. In this performance we hear the initialization seed an exploration of closely related frequencies.

In sections 2 and 3 the initialization frequency of individuals is set to 0, but the target frequency of the global frequency table is 440. Accordingly (at 44 seconds) we hear a grouping around 440 Hz. but this dissipates shortly as other frequency areas are visited. In the first 167 seconds of section 3 the population slowly depletes its frequency resources and we hear the individuals weaken as they search the frequency table. Sixty seconds from the end of section 3 the population discovers a new strategy, selecting a frequency just below the target frequency. We hear an overlapping sequence of downward

steps that create a glissando effect. In the last thirty seconds a better strategy is found along with an unexplored frequency area and the population settles into B major followed by C major.

While the first three sections contain relatively little “composer intervention” section 4 has the most, as we force the population to follow a pre-composed four-part harmonic sequence. High frequencies are strongly attenuated as they have a masking effect on the harmony. We wish to hear how a population “interprets” a musical score. The evaluation function rewards individuals that find the target frequency (or its multiples) as in other sections; but instead of being derived from the frequency table the target frequency is one of the notes of the four-part harmony (using a round-robin rotation). A pulse is also applied in this section, although it is not obvious; the population is divided into four groups each of which has a unique pulse. The pulse of all groups slows down over the duration of this section.

Individuals are again initialized to A 440 in section 5 and a uniform pulse is forced on the population; we hear a 6/4 meter. The quasi tonic-subdominant alternations heard in the latter part are accidental.

Section 6 uses an evaluation function is similar to the one used in sections 1, 2 and 3 above, but two differences will explain its somewhat different sound. First, very low frequencies (those below 100 Hz.) are not attenuated to the same degree as in the earlier sections. Next, an individual receives a bonus reward if it plays a frequency that is an integer relation of its neighbor’s frequency. This strategy is an attempt to create a resonant, organic sound. The same evaluation function is used for a much longer piece discussed below, *Passacaglia Polymer C3*.

Finally section 7 demonstrates a second pulse overlay, this time in a 2/4 meter. The basic pulse is interrupted every 37 seconds by a brief up-tempo riff, after which the normal pulse resumes.

Passacaglia Polymer C3

Neighbor frequency relations are given a vast amount of time to develop in *Passacaglia Polymer C3*, an hour-long work intended for reflection in a meditative context. The zero-sum frequency table and the composer interventions introduced in Part 3 are used. A polymer is a biological chain, commonly made of amino acids. While the piece is not a true Passacaglia, as a repeated bass is not heard, a set of interventions is repeated continuously and a sense of solid inevitability commonly heard in Passacaglia form is heard.

At a synthesis rate of 1000 frames per second, and a population size of 1024, evolution is given ample opportunity by the resulting 1,024,000 evaluations per second or 3,686,400,000 evaluations over the duration of the piece. If we calculate the length of a generation as the median lifetime of organisms in the population, 0.8 seconds, 4500 generations are represented in the piece. Applying the same calculation to the human time scale, using 30 years (the median human lifetime) as the interval between generations, we arrive at a figure of 135,000 years; notwithstanding the significant cultural and technical accomplishments of the species, it is normally considered that *homo sapiens* has not evolved in that interval. The comparison is intended to show the operational scope that may be needed to allow evolution to take place within a computer model such as this one; we may might observe that in all likelihood no significant evolution takes place within this musical composition.

Nonetheless, we do hear an extended audio realization of an artificial-life cosmos; we may also hear in this piece the beginnings of some primitive musical constructions: consonant harmonies, inchoate melodies. Recall that these organisms contain no internal musical knowledge; they have no awareness of notes, harmony, rhythm – not even of time. Knowing only the frequency of their immediate neighbor, however, allows individuals to participate in a consonant chain – a polymer – a simple form of cooperative behavior which enhances the fitness of any participating individual.

The Dilemma of Intervention

A short piece like *Suite for Proteins* represents a kind of music quite different than originally imagined in this paper. We learned earlier that evolution is a slow and largely static process. Left alone, its simulation on a computer produces music that changes little over time. An early hope was that complex organisms would emerge if given a long period in which to evolve; this assumed that a complex organism would produce interesting sounds. We learned that evolution (in this implementation) primarily favors efficiency; complex strategies emerged only rarely and died out immediately.

If the population is necessarily limited to simple strategies (both by the preferences of an evolutionary process and by the limits of computing power) it can still produce dynamic sounds if it exists within a changing environment. We did this in Part 3. But cataclysmic environmental shifts usually result in the death of most of the population and rapid adaptation to new circumstances, producing static music with occasional interruptions. We heard this in *Passacaglia Polymer C3*.

Now we encounter a dilemma. To create a more narrative style of music, one with small and regular changes as well as local contrasts, we should add more volatility to the environment. But the more we manage the environment, the less autonomy we allow the population. Taken to its extreme, managing the environment is equivalent to direct composition – where an evolutionary process is not needed. But an environment that is not given shape produces shapeless or static results.

The program described in this paper creates a different kind of music; we do not hear the narrative flow we might reasonably desire, and the music exists on a scale of time we shall not live long enough to fathom. We can understand the *idea* of the music but the music itself may be beyond us.

Summary

Rudimentary organisms can be forced to produce different kinds of sounds, but musical architecture is still crafted by humans. The *Suite for Proteins* is, for example, a thoroughly managed effort; like a flower arrangement, it may have originated in Nature, but it is not the result of a natural process. Artificial evolution nonetheless can be used to create music within a judiciously controlled environment.

Appendix A

Introduction

The C++ implementation used for this paper is described. A working code template (for windows and linux) is included.

The biggest problem for an artificial intelligence system like this one is evaluation. How does the author reward musical organisms that produce "good" results from millions of candidates – by listening to the output of individual organisms? There is not sufficient time in a human life to do so. A by-product of this paper is a serious reflection on scale: of human time, computer time and evolutionary time. The natural world, in computer terms, can devote a huge amount of memory, massively distributed computation power and incredible lengths of time to problem solving. Our computers, in comparison, are suitable only for the most simple tasks.

In this sense the code represents a collection of compromises – primarily the compromise of evaluating an artistic goal with numbers. We cannot write a function to reward "good" sounds and penalize the "bad." Instead we compromise, artfully, rewarding individuals for having desired frequency relationships with others. This fairly crude method sometimes makes sounds we can call music.

Nodes, Functions and Terminals

The "individuals" discussed in this paper are tree-shaped computer programs (*code-trees*) which have either been assembled arbitrarily or created by mating (note: we avoid using *random*. The word *arbitrary* better expresses the results of calling `rand()`). The trees are stored in an array and are evaluated within a virtual machine. The code follows the same top-down structure discussed throughout the paper:

> An environment contains a population of individuals and the resources the population uses.

A population is an array of individuals.
An individual is a code tree with associated variables such as a health score.
A code tree is an assemblage of *nodes*.
A node is a *function* or a *terminal*.
A function accepts one or more inputs and returns an output.
A terminal returns an output but takes no inputs.

The input to a function may be either another function or a terminal. For the sake of global compatibility all inputs and outputs are floating point numbers.

An example function:

```
float plus(float a, float b)
{
     return a + b;
}
```

An example terminal:

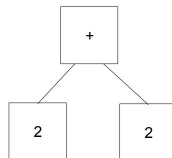
```
return (float)1.0;
```

The node class has a member, isFunction, which indicates whether a node is a function or a terminal. Each node has pointers to global (static) lists of functions and terminals (`funclist`, `termlist`) and pointers for individual functions and terminals. If the node is a function, `isFunction` will be set to 1, `funcp` will point to a function in `funclist` and `termp` will be set to NULL. If the node is a terminal, `funcp` will be NULL and `termp` will point to a terminal in `termlist`.

```
class node {
     public:
     Func funcp;
     Term termp;
     Func funclist;
     Term termlist;

     int inputCount;
     int depth;
     int isFunction;
     node ** input;
     node * parent;
};
```

A function node will have `inputCount` inputs – an array of nodes pointed to by `input`. When a node evaluates itself it first evaluates its input nodes, if any, then returns the result of its `funcp` member (a pointer to a function). For example, the expression:

plus(2, 2);



would occupy three nodes, as shown in the diagram. A function node would point to the address of a `plus()` function in `funclist`. This function requires two inputs, which in this case are terminals; in both terminal nodes, `termp` would point to the value 2 in `termlist`. When the top node is evaluated it first evaluates its two inputs. Since the inputs are terminals they each return 2. The top node then passes these results to its function address (`funcp`) and returns the result.

The code below shows how a tree is evaluated recursively. A maximum of three function inputs are handled by this code. When called, a node first determines whether it is a function or a terminal. If it is a terminal, it returns `termp->val;` otherwise it calls its one, two or three-input function address, passing it the specified number of

inputs. All trees eventually end with terminals; otherwise this evaluation would continue indefinitely.

```
float node::eval()
{
     float (*ff)(float);
     float (*fff)(float, float);
     float (*ffff)(float, float, float);


     if(isFunction) {
          switch(inputCount) {
          case 1:
               ff = (FF)funcp->func_address;
               return ff(input[0]->eval());
          case 2:
               fff = (FFF)funcp->func_address;
               return fff(
                    input[0]->eval(),
                    input[1]->eval());
          case 3:
               ffff = (FFFF)funcp->func_address;
               return ffff(
                    input[0]->eval(),
                    input[1]->eval(),
                    input[2]->eval());
          default:
               return 0.0f;
          }
     }
     return termp->val;
}
```

Recursion is also used when a tree is initially created, as seen in the code below. A function node needs inputs; the number of inputs is expressed in the function's *form* member (a text string). A form of "ff" means the function returns a float and takes one float as input. The string "fff" means the function takes two float inputs. For each input needed by a node, a new node is created at a depth one level greater than the current node. After some initialization (`setParent()`, `setTermList()`, `setFuncList()`) the input chooses

a function or terminal from a list. If a function is chosen, the input node continues the recursive growth process. As above, this growth must eventually end by choosing a terminal; accordingly, when the `depth` of a node reaches a predefined maximum it is forced to choose a terminal.

```
void node::grow()
{
     char * p;
     int count = 0;

     // Parse the form string to determine the number
     // of inputs for this node.

     for(p = funcp->form + 1, count = 0;
               *p != '\0';
               p++, count++) {

          input[count] = new node(depth + 1);
          input[count]->setParent(this);
          input[count]->setTermList(termlist);
          input[count]->setFuncList(funclist);
          input[count]->chooseFuncOrTerm();

          if(input[count]->getIsFunction()) {
               input[count]->grow();
          }
     }
}
```

The reason for this seemingly complex burying of functions and terminals within nodes (known as *encapsulation*) is to allow easy swapping between trees. Mating is done by swapping two node pointers; a complete tree is effectively managed by addressing its top node. For example:

```
node * head = new node;
head->grow();
head->eval();
```

This code creates a new (empty) head node, grows a complete tree from it, then evaluates the tree. A complete tree is a member of the ind ("individual") class, which also keeps track of the size of the tree in both nodes and bytes.

```
class ind {
     public:

     node * head;
     int nodeCount;
     int byteCount;
};
```

Two individuals mate by copying their respective trees, choosing a crossover node from each copy and swapping pointers as indicated in the code below.

```
void IndMate(ind * parent1,
               ind * parent2,
               ind ** child1,
               ind ** child2)
{
     *child1 = IndCopy(parent1);
     *child2 = IndCopy(parent2);

     IndCrossover(*child1, *child2);
}
```

The two children were empty pointers when they entered this function. They leave the function with new trees inherited from the two parents.

Solving a Simple Problem

We can now revisit the example from Part 1, where we evolved the equation for calculating the hypotenuse of a right triangle from two legs. The following is a synopsis of the procedure found in main.cpp of the attached code.

1) Initialize the lists of functions and terminals that will be used by calling `GenManInit()` ("genetic programming manager initialize"). In genman.cpp the functions `mult()`, `plus()` and `sqroot()` are defined and registered (using `FuncAdd()`).
2) Create a population of trees with
   ```
   for(i = 0; i < TREES; i++) tree[i] =
               GenManNewInd();
   ```
   where TREES is 1000.
3) Begin a loop where the population will be evaluated repeatedly. Exit the loop when four perfect results are found in the population.
4) To evaluate a tree, choose two random legs with lengths greater than 1.0; set those values, calculate the correct answer, then evaluate the tree with `GenManEval()`. Evaluate each tree 100 times and accumulate the error (the difference between the correct answer and the result from `GenManEval()`).
5) Sort the trees in the order of their accumulated error. Trees with the lowest amount of error go to the top.
6) Check if the exit condition is true (4 perfect results).
7) If not, replace the lower half of the population. Delete two trees, choose two parents from the surviving half of the population, check for a mutation condition (in this case, substitute a new tree for one of the parents) then replace the deleted trees with `IndMate()`. Increment the generation count and continue to iterate within the evaluation loop.

When the exit condition is met, the program will have evolved at least four correct (and probably identical) solutions. This code can serve as a starting template for solving other simple problems.

The complete code is in hypotenuse.tar, which compiles and runs on Windows (MSVC 6.0), Mac OS X and linux.

Continuous Evolution and Audio Synthesis

The above example uses generational evolution to solve a simple problem. In this paper we used continuous evolution to create musical sounds. Sound takes place over time, so we create a sound-making class capable of maintaining state over successive evaluations. In the code (for historical reasons related to onscreen rendering of populations) this is called a `Box`.

```
class Box {

     protected:

     Environ * environ;
     Soundob * soundob;
     float juice;
     float cost;
     float amp_gain;
     float freq_gain;
     unsigned long daycount;
     int state;
     int tag;

     double frequency;
     float listenToState;
     double amplitude;
     ind * dude;
};
```

The Box class persists over time – as long as `juice` is greater than zero. When a Box dies it sets `state` accordingly and the Environment will then replace it. A Box evaluates `dude` every audio synthesis frame and may contribute `frequency` and `amplitude` to `soundob`, the additive synthesis engine. A population of Boxes lives in an environment:

```
class Environ {

	protected:

	Box * population[MAXPOPULATION];
	Sinemix * sinemix;
	int popsize;
	int alive;
	int kills;
	int starves;
	int mates;
	int mutations;
	unsigned long daycount;
	unsigned long totalbytes;

};
```

which also owns the sinewave-mixing object `sinemix`, and other members used for accounting; it replaces dead Boxes by mating two healthy Boxes found in `population`.

`Sinemix` is a frame-based additive synthesis engine. For each audio frame (typically 12.5 milliseconds or 110 samples at 44100 samples per second) a Box may contribute its frequency and amplitude (for that frame) with `addFreq()`.

A synopsis of the main calling program is:

1) Initialize the lists of functions and terminals that will be used by calling `GenManInit()`. A long list of functions are registered, including functions inside a Box object – or in others. For example, a Box may be able to determine the frequency or amplitude of another Box object.
2) Create a single Environ object (`environment`).
3) Create a population of Boxes with
   ```
   for(i = 0;
   	i < MAXPOPULATION;
   	i++)
   		environment ->createNew();
   ```
   where MAXPOPULATION is 1000 or more.

4) Begin a loop where the population will be evaluated repeatedly (with `environment ->day()`). Exit the loop when the desired amount of audio has been generated.

Inside the evaluation loop, the population is evaluated each "day." The evaluation is found in `Environ::day()` which does the following:

1) Clear the output audio buffer with

```
sinemix->clearMix();
```

2) Evaluate each member of the population with

```
for(i = 0;
     i < MAXPOPULATION;
     i++)
          population[i]->day();
```

   Audio for a new output frame will accumulate in the output buffer.

3) Determine which members of the population are dead and delete them.
4) Replace deleted members by mating healthy members.
5) Flush the output audio buffer to file with

```
sinemix->writeMixbuf();
```

Finally, we look at the evaluation function for the Box class – which determines almost exclusively the kind of sounds the program will produce. In synopsis, `Box::day()` does this:

1) Compute the cost for this day (complexity plus age) and subtract from `juice`.
2) Evaluate `dude` and add the result to `frequency`.
3) Calculate a reward (if any) by comparing `frequency` to a desired target. In the attached code, an integer relationship to a

neighbor frequency (`buddy`) is highly rewarded. Add the reward to `juice`.

4) Set amplitude to correspond with health (`juice`).
5) If the individual has died (`juice < JUICE_DEAD`) set its state accordingly.
6) Penalize, heavily, frequencies outside the range of human hearing.
7) If the individual is alive (and has been for 40 days) and has an audible frequency, add its frequency and amplitude to the output audio buffer.
8) If the individual just died, add a final fadeout audio frame with `soundob->die()`.

The attached code created audio example 2.14 in this paper. It is simpler than code from later examples, making it a good starting template, but has the infrastructure in place for more advanced experiments. The code has compiled and run on NeXT, SGI, linux, Windows and Macintosh machines; it produces a wave-format output file (handling byte-swapping as needed). An output file containing all the code-trees in a population can be produced as desired; the program can later read in this file, which is handy when one wishes to pick up an evolution process from an intermediate point rather than starting over from the beginning.

The complete code is in continuous.tar, which compiles and runs on Windows (MSVC 6.0), Mac OS X and linux.

Appendix B

CD Supplements

Sound and code examples are provided on 2 CDs. CD 1 contains all the audio examples except for the composition *Passacaglia Polymer C3*. CD 1 also contains the code examples described in Appendix A. The audio examples on CD 1 are in mp3 format. The code examples are in uncompressed tar files.

Passacaglia Polymer C3 is in CD audio format and occupies the whole of CD 2.