

## **Prokaryotic Groove: Rhythmic Cycles as Real-Value Encoded Genetic Algorithms**

Christopher Ariza, NYU GSAS, New York, New York

Since their creation in the 1960s, Genetic Algorithms (GAs) have been employed to solve a staggering diversity of problems. This research, initiated by computer-scientists, engineers, and biologists, was first given a theoretical framework in John Holland's *Adaptation in Natural and Artificial Systems* (1975). The Holland-GA is designed to enable a population of "chromosomes" to mate under the constraints of "natural selection" and form new offspring (more "chromosomes") under genetic-modeled operators (crossover, mutation). Through successive populations, chromosomes "evolve" toward individuals better suited to the given selection criteria. Traditional GAs are well suited for problems that require searching through large numbers of possible solutions (Mitchell 4). GAs, in these cases, provide effective use of parallelism, in which numerous possibilities are simultaneously explored in an efficient fashion. GAs have been successfully implemented in a wide variety of tasks, including computational protein engineering, automatic programming, and the modeling of economic and ecological systems.

More recently GAs have been shown useful in contexts where the problem to be solved is not rigorously defined, but rather an aesthetic goal. This is the point of departure for this study. Here GAs are developed to create varied successions of rhythmic loops. Rather than solve a difficult problem, these GAs solve a simple problem: match a predefined rhythm. The goal is not the solution, but the trajectory of successive populations moving toward the solution. In this way, each time the GA is run a collection of rhythmic variants is produced that tend toward the fit rhythm. This suggests that the actual process of a GA, under the appropriate musical representation, has musically desirable attributes.

This study attempts to capture the musicality of the GA in rhythmic systems. Using Python, a portable, object-orientated GA generator titled "GARthm.py" is developed. This module is integrated within the cross-platform open-source athenaCL composition system, allowing the deployment of the resulting rhythms in algorithmically created scores and rendered with Csound.

### **Genetics and GAs**

This paper will assume familiarity with the elementary vocabulary of biological genetics. Though sharing vocabulary, GAs are abstractions of biological and ecological systems. The parallel is approximate: many of the subtleties of evolution in real organisms are lost in the GA. Accordingly, terms borrowed from genetics do not have the same rigor when applied to GAs, though they nevertheless indicate parallel functionality. The differences between GAs and their natural models is most significant in the distinction between phenotype and genotype.

Most GAs maintain no distinction between phenotype (data expressed) and genotype (the data coded). In most GAs, phenotype is genotype. This is more like the behavior of simple prokaryotic organisms like bacteria than complex eukaryotes like humans. In bacteria, there is a single copy of the genetic material (a chromosome), wrapped in a loop. There are no dominant or recessive genes (or alleles, the term given to gene-variants) that code for physically manifested traits. In most cases, what is in the chromosome (genotype) is what is physically manifested in the bacterium (phenotype). In humans, on the other hand, there are two complete versions of the genetic material (two copies of each of the 23 chromosomes), wrapped in tight strands in the cell-nucleus. A given gene may have a dominant allele on one copy of a chromosome and a recessive allele on the other copy of the chromosome. What is in the chromosomes (genotype) is not completely expressed in the physical manifestation (phenotype); in such a case, the recessive allele will not get expressed, though it has equal opportunity of being passed on to successive generations.

### **The Design of the Chromosome**

This study uses many-character or real-valued encodings of chromosomes rather than the more common binary encodings. There is no fixed rule for when a chromosome should be encoded as a bit-string or as a real-value. Though fixed-length, fixed-order binary encodings are the most common and best-explained by the existing theoretical literature, "binary encodings are unnatural and unwieldy for many

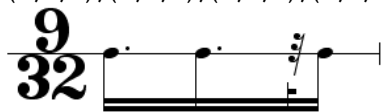
problems..., and they are prone to rather arbitrary orderings (Mitchell 157). As Melanie Mitchell states, “for many applications, it is most natural to use an alphabet of many characters or real numbers to form chromosomes” (157).

Using real-value encodings allows the development of a GA based on fitness in relation to a target-chromosome. That is, instead of searching the fitness-landscape for an unknown solution, the GA is given the solution from the beginning. The fitness of each individual is measured in comparison to the fit chromosome, the “goal” provided at initialization. As should be apparent, this is a trivial problem in comparison to typical GA applications, where the “answer” is not (and sometimes cannot be) known in advance. Rather, here the answer is not of interest; what is of interest is the path that each population takes toward the known-destination. The starting point is a population of chromosomes randomly filled with alleles from the allele alphabet of the fit-chromosome.

In GARthm.py each allele is a data-structure that defines a duration and a note/rest-state value. That is to say, each rhythm-allele is defined by a tuple, a Python data structure that holds information about the rhythm-unit, called here a rhythm-tuple. This information is given as a list of three integers: (Divisor, Multiplier, Note/Rest-State). The duration of all rhythm-tuples in a chromosome are measured in reference to a beat-time, or the rate of pulse measured in seconds. Thus a pulse of 120 BPM would have a beat-time of .5 seconds. The duration of the rhythm-tuple is calculated in two-steps: first the beat-duration is divided by the divisor, then multiplied by the multiplier. The note/rest-state value determines whether the measured duration is a note (value of 1) or a rest (value of 0). For example, a sixteenth-note rest would thus be expressed as (4,1,0). If the beat-time is .5 seconds, the duration is .5 divided by 4, then multiplied by 1, or .125 seconds. Since the last value in the rhythm-tuple is a 0, the duration is a rest. A dotted sixteenth-note, on the other hand, can be expressed as (8,3,1). Given the same beat time of .5 seconds, this duration is divided by 8, then multiplied by 3, yielding 0.1875 seconds. The duration of the beat itself (a quarter note) is represented as (1,1,1).

A chromosome in this model is an ordered succession of rhythmic-tuples. All the rhythms in a single chromosome are defined in relation to a single beat-time. Though reliance on a fixed, referential beat may seem a shortcoming, it is both musically intuitive and capable of expressing nearly any duration. In Figure 1 a chromosome is illustrated as a list of rhythmic-tuples and as a notated rhythm.

**Figure 1.**  
 (8, 3, 1), (8, 3, 1), (8, 1, 0), (4, 1, 1)



The same rhythm can be presented at a different metric level by scaling all the divisors. For instance, if each divisor in Figure 1 is quartered (played four-times as slow) the result is the same rhythm in dotted-quarters, one-quarter the original speed. This is notated in Figure 2 .

**Figure 2.**  
 (2, 3, 1), (2, 3, 1), (2, 1, 0), (1, 1, 1)



Non-even and non-rational divisions of the beat can be used alongside traditional beat-divisions, allowing a wide variety of rhythms. The following chromosome, notated in Figure 3, demonstrates the variety of rhythms that can be represented as rhythm-tuples:

**Figure 3.**

(4,1,1), (4,1,0), (8,1,1), (8,1,1), (4,1,1),  
(6,1,1), (12,1,0), (12,1,1), (12,1,0), (12,1,1),  
(5,1,0), (5,1,1), (5,1,0), (15,2,1), (15,2,1), (15,2,0)



Rhythm-tuple alleles present a data-structure for designating note and rest durations. They also allow gradual, musical modification of rhythms. Any rhythm-tuple can, through successive modifications, transform into any other rhythm-tuple. For example, a quarter note might evolve into a triplet-eighth rest after three simple transformations: (1,1,1), (2,1,1), (3,1,1), (3,1,0). Rhythm-tuples allow a high degree of redundancy: (6,2,1) is precisely the same duration as (3,1,1). This redundancy allows for a number of different expressions of the same duration, promoting fluid pathways from one duration to another. These transformations and redundancies form the basis of mutation-algorithms in GARthm.py.

### Populations, Selection, and Fitness

A population is a collection of individuals, or a collection of individual chromosomes. The population size in a GA is the number of individuals in the population, typically from twenty to a few hundred. In this model, the population size is twenty (a value notated as  $n=20$ ). To initiate the system, the population is created with  $n$  chromosomes, each chromosome based on the fit-rhythm given by the user. The fit-rhythm is the goal of the population. The chromosomes in the initial population all have the same length as the fit-chromosome, and length is fixed. Alleles on each chromosome are randomly chosen from alleles available in the fit-chromosome. This population is the first generation, and is designated by its iteration number (notated as  $i=0$ ).

Next, each member of the population is analyzed with a fitness function. A fitness function tests the individual chromosome in a manner specific to the problem at hand. Where a chromosome is a candidate solution to a problem, a fitness function provides a quantitative measure of how well the problem is solved. In this model, the fitness function is a quantitative evaluation of the relation of the chromosome to the fit-rhythm. The fitness function will be explained in more detail below.

To produce the next population ( $i=1$ ) members of the population at  $i=0$  are selected to reproduce. Based on the values calculated for fitness above, more-fit individuals are proportionally more likely to be selected to reproduce, though for each selection any member of the population is a candidate. This procedure is called fitness-proportionate selection, and can be carried out with a variety of algorithms. Here, the method termed “roulette wheel sampling” is used. Each individual in the population is given a “slice” of an imaginary wheel proportional to the individual’s fitness. The wheel is “spun” and, to continue the metaphor, whichever wedge the ball stops on is the “winner” and is selected. After two individuals have been selected, they are then mated to produce two new offspring to fill population  $i=1$ . This process is repeated until the size of population  $i=1$  is equal to  $n$  (the size of population  $i=0$ ).

The mating of individuals selected from the population goes through two steps: crossover and mutation. These are often referred to as the simple genetic operators. Crossover is the process of mixing parent chromosomes to produce new child chromosomes. To do single-point crossover, a locus is chosen at random. Each parent’s chromosome is “cut” at this locus and two new chromosomes are created by combining a sub-section from each parent. To use binary chromosomes as an example, parent-chromosomes such as 00110101 and 11111111 might undergo crossover after the second locus. The two resultant chromosomes would be 11110101 and 00111111. When child-chromosomes are produced without crossover, they are genetic clones of the parent chromosomes. The crossover rate determines the percentage of matings that undergo crossover. A crossover rate of .7 designates that 70% of matings use crossover (notated as  $p_c=.7$ ), whereas the remaining 30% of matings simply produce genetic clones.

Following crossover, child chromosomes may undergo mutation. Mutation takes a child chromosome and alters it at a specific locus. This study employs musically meaningful mutations, to be discussed in detail below. The frequency of mutation is determined by the mutation rate (notated as

$p_m=.001$ ). A chromosome's probability of mutation is independent of whether or not crossover has been performed.

After a complete population has been created through selecting and mating parents, new fitness values are calculated and the process is repeated. There is no mating between chromosomes in different generations: each successive iteration completely replaces the former. Through a process called elitism a small number of the most fit chromosomes of a population can be allowed to carry over into the next population unchanged, without crossover or mutation.

#### The Design of the Fitness Function:

The importance of the fitness function in determining the ability of a GA to solve a problem cannot be underestimated. The fitness function can be adjusted in many ways to fine-tune the movement of populations. This model measures the relation to the fit-chromosome with five independent time-distance measures. These time-distance measures describe the qualitative difference between the chromosome and the fit-chromosome in terms of seconds, where a time-distance value of zero expresses a perfect match (no time-distance between the compared chromosomes). This fitness scale is inversely proportionate: the most fit chromosome has a fitness value of zero. There is no maximum fitness value. The five time-distances compare (1) the duration of the sum of note-durations, (2) the duration of the sum of rest-durations, (3) the total duration of the chromosome, including notes and rests, (4) the duration of not-matched alleles, (5) the duration of not-matched values. These measures, all in units of seconds, are weighted to heighten selection preferences, and then summed to produce a final value. The following table provides a prose definition of the fitness algorithm used in GARthm.py.

**Figure 4.** Fitness Function

#### Inverse Proximity in Time Distance

- (a) Calculate the note-duration, rest-duration, and total duration of the chromosome in seconds. Calculate note-duration, rest-duration, and total duration of the fit-chromosome in seconds. Find the absolute value of the difference between each of these values, named `noteDistance`, `restDistance`, and `durDistance`.
- (b) Compare each allele in the chromosome to the allele in the corresponding position of the fit-chromosome. If duration and note/rest-state values are identical, add 1 to `matchAlleleScore`. If duration values are identical, despite value of note/rest-state, add 1 to `matchValueScore`.
- (c) `matchAlleleScore` is subtracted from the bit length to produce the `noMatchAlleleScore`. `matchValueScore` is subtracted from the bit length to produce the `noMatchValueScore`.
- (d) `noMatchAlleleScore` and `noMatchValueScore` are scaled by the average duration of each allele of the fit-chromosome. `noMatchAlleleScore` becomes `noMatchAlleleDistance`, the duration in seconds of alleles that do not match. `noMatchValueScore` becomes `noMatchValueDistance`, the duration in seconds of values that do not match.
- (e) Sum values after weighting as follows: `noteDistance*1.50`, `restDistance*1.50`, `durDistance*2.33`, `noMatchAlleleDistance*1.00`, and `noMatchValueDistance*0.66`.

This model of rhythmic fitness evaluation can be weighted to select for different results. For instance, the weightings presented above allow the mixture of rest and note positions (low `noMatchValueDistance` and low `noMatchAlleleDistance` weightings) while preserving the total duration (high `durDistance` weighting). Reversing these values would result in the selection of chromosomes with better matching segments but wide-ranging total durations.

#### The Design of the Genetic Operators

Single point crossover, as used in basic bit-string GAs, has a functional limitation in that it cannot combine all possible schemas, or building-block segments, of a chromosome. Single point crossover is damaging to schemas with long defining lengths (a "positional bias"), and disproportionately favors endpoints (segments exchanged between parents always contain endpoints) (Mitchell 172). To alleviate this problem, this model uses two-point crossover, dividing each chromosome into three segments and exchanging the parts. As Mitchell states, two-point crossover reduces positional bias, the endpoint effect, and "is less likely to disrupt schemas with large defining lengths and can combine more schemas than single-point crossover" (172). There are still schemas that two-point crossover cannot combine.

Mutation as an instrument of variation has, in traditional GA research, taken a background role. Generally, “crossover is the major instrument of variation and innovation in GAs, with mutation insuring the population against permanent fixation at any particular locus” (173). There are few empirical comparisons to support either argument. Recent thought, Mitchell informs us, has shown a growing appreciation for the role of mutation (174), with some studies even demonstrating verifiable superiority. It is important, however, to keep in mind that “it is not a choice between crossover and mutation but rather the balance among crossover, mutation, and selection that is all important” (174).

Creative applications of GAs have often emphasized mutation. Robert Rowe, in his survey of sub-symbolic processes applied to music, discusses GenJam, software developed by John Biles that uses genetic algorithms to improvise solo jazz lines in real-time. GenJam uses four types of mutation, termed by the author “musically meaningful mutations”: reverse (retrograde), rotate, inverse (melodic inversion), and transpose (Rowe 253). This idea of musically meaningful mutations is precisely what is used in GARthm.py to operate on rhythmic-tuples in an efficient and musically-rational way.

GARthm.py has five mutation types. Four of these act on a single, randomly-selected locus at a time. These mutation operators are outlined in Figure 5 and consist of Ratio Equivalence ((3,1,0) becomes (6,2,0)), Divisor-Mutate ((3,1,0) becomes (4,1,0)), Multiplier-Mutate ((3,1,0) becomes (3,2,0)), and Flip Note/Rest-State ((3,1,0) becomes (3,1,1)). The fifth mutation operator replaces a segment of a chromosome with the segment’s positional retrograde (Inversion). When a chromosome has been selected to be mutated, one of the five mutation types is chosen based on an equal probability. The following table provides prose definitions of the mutation algorithms used in GARthm.py.

**Figure 5.** Mutation Types

**Ratio Equivalence**

Select a random locus. Multiply or divide both the divisor and multiplier of the rhythm-tuple by either 2 or 3. Division or multiplication is chosen at random; if undividable, multiplication is used.

**Divisor-Mutate**

Select a random locus. If the divisor of the rhythm-tuple is equal to 1, add 1 to the divisor. If the divisor is greater than 1, either add or subtract 1.

**Multiplier-Mutate**

Select a random locus. If the multiplier of the rhythm-tuple is equal to 1, add 1 to the multiplier. If the multiplier is greater than 1, either add or subtract 1.

**Flip Note/Rest-State**

Select a random locus. If the rhythm-tuple is a note, make it a rest. If the rhythm-tuple is a rest, make it a note.

**Inversion**

Randomly select two loci form the chromosome. Reverse the segment and replace the original with the retrograde segment.

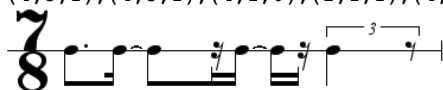
Elitism affects the process of selection by automatically selecting a percentage of the most-fit in the population to pass on as clones, un-mated and un-mutated, to the next generation. This greatly increases the stability of the GA, particularly in cases where the fit-chromosome is either long or consists of many elements. Because of the power of elitism, it can be used when needed to a control an otherwise unstable GA system. As GARthm.py allows for different allele lengths (depending on the fit-chromosome), elitism is available to counter-act the large number of iterations needed for populations to evolve fit-chromosomes with large allele lengths.

**Observing Each Generation in GARthm.py**

Creating a GARthm.py genome object in Python requires six arguments. The first is *n*, the population size, here given a value of 20. The second is the fit-chromosome, notated in Figure 6.

**Figure 6.**

(4, 3, 1), (4, 3, 1), (4, 1, 0), (2, 1, 1), (4, 1, 0), (3, 2, 1), (3, 1, 0)



The third argument is the tempo in BPM (120), the fourth the crossover rate (.8), the fifth mutation rate (.18), and the sixth the elitism percentage (.16). As can be seen in Figure 7, data about each iteration is presented on a single row of text. Text output for twenty-five iterations of GARthm.py are illustrated. Generally, the “best” chromosome of a given population is of most interest. This is the chromosome displayed following “bst” for each iteration. In the case where the same “bst” chromosome has been found twice in a row, the repeated chromosome is not displayed.

**Figure 7.** Python output of GARthm.py

```
>>> a = GARthm.genome(20,((4,3,1),(4,3,1),(4,1,0),(2,1,1),(4,1,0),(3,2,1),(3,1,0)), 120, .8, .18, .16)
>>> a.gen(25)
bst[(2,1,1),(4,3,1),(4,1,0),(4,3,1),(2,1,1),(4,1,0),(3,1,0)]          i: 1  avgFit:756.58  bstFit:457.40  bstDur:400.00
bst[(3,2,1),(2,1,0),(3,2,1),(2,1,1),(4,1,0),(3,2,1),(3,1,0)]          i: 2  avgFit:719.68  bstFit:405.32  bstDur:430.00
bst -                                                                    i: 3  avgFit:661.97  bstFit:405.32  bstDur:430.00
bst -                                                                    i: 4  avgFit:651.27  bstFit:405.32  bstDur:430.00
bst -                                                                    i: 5  avgFit:724.03  bstFit:405.32  bstDur:430.00
bst[(3,2,1),(3,1,0),(3,2,1),(2,1,1),(4,1,0),(3,2,1),(3,1,0)]          i: 6  avgFit:638.69  bstFit:361.04  bstDur:410.00
bst -                                                                    i: 7  avgFit:543.03  bstFit:361.04  bstDur:410.00
bst[(2,1,1),(4,3,1),(4,1,0),(2,1,1),(4,1,0),(3,2,1),(3,1,0)]          i: 8  avgFit:492.02  bstFit:207.72  bstDur:390.00
bst[(3,2,1),(4,3,1),(4,1,0),(2,1,1),(4,1,0),(3,2,1),(3,1,0)]          i: 9  avgFit:527.34  bstFit:135.88  bstDur:410.00
bst -                                                                    i: 10 avgFit:407.23  bstFit:135.88  bstDur:410.00
bst -                                                                    i: 11 avgFit:380.75  bstFit:135.88  bstDur:410.00
bst -                                                                    i: 12 avgFit:386.61  bstFit:135.88  bstDur:410.00
bst -                                                                    i: 13 avgFit:322.37  bstFit:135.88  bstDur:410.00
bst -                                                                    i: 14 avgFit:269.45  bstFit:135.88  bstDur:410.00
bst -                                                                    i: 15 avgFit:223.09  bstFit:135.88  bstDur:410.00
bst -                                                                    i: 16 avgFit:239.08  bstFit:135.88  bstDur:410.00
bst -                                                                    i: 17 avgFit:292.48  bstFit:135.88  bstDur:410.00
bst -                                                                    i: 18 avgFit:307.37  bstFit:135.88  bstDur:410.00
bst -                                                                    i: 19 avgFit:291.24  bstFit:135.88  bstDur:410.00
bst -                                                                    i: 20 avgFit:243.52  bstFit:135.88  bstDur:410.00
-----
bst -                                                                    FitV:((4,3,1),(4,3,1),(4,1,0),(2,1,1),(4,1,0),(3,2,1),(3,1,0))
bst -                                                                    Pn:20  Pc:0.800  Pm:0.180  fitDur:420.00
bst -                                                                    i: 21 avgFit:268.34  bstFit:135.88  bstDur:410.00
bst -                                                                    i: 22 avgFit:250.30  bstFit:135.88  bstDur:410.00
bst -                                                                    i: 23 avgFit:189.80  bstFit:135.88  bstDur:410.00
bst -                                                                    i: 24 avgFit:203.18  bstFit:135.88  bstDur:410.00
bst[(4,3,1),(4,3,1),(4,1,0),(2,1,1),(8,2,0),(3,2,1),(3,1,0)]          i: 25 avgFit:172.81  bstFit:0.00    bstDur:420.00

unique BestFit BitVectors
[(4,3,1),(4,3,1),(4,1,0),(2,1,1),(8,2,0),(3,2,1),(3,1,0)]          fitness = 0.0000
[(3,2,1),(4,3,1),(4,1,0),(2,1,1),(4,1,0),(3,2,1),(3,1,0)]          fitness = 135.8800
[(2,1,1),(4,3,1),(4,1,0),(2,1,1),(4,1,0),(3,2,1),(3,1,0)]          fitness = 207.7200
[(3,2,1),(3,1,0),(3,2,1),(2,1,1),(4,1,0),(3,2,1),(3,1,0)]          fitness = 361.0400
[(3,2,1),(2,1,0),(3,2,1),(2,1,1),(4,1,0),(3,2,1),(3,1,0)]          fitness = 405.3200
[(2,1,1),(4,3,1),(4,1,0),(4,3,1),(2,1,1),(4,1,0),(3,1,0)]          fitness = 457.4000

true unique BestFit BitVectors
[(4,3,1),(4,3,1),(4,1,0),(2,1,1),(8,2,0),(3,2,1),(3,1,0)]          fitness = 0.0000
[(3,2,1),(4,3,1),(4,1,0),(2,1,1),(4,1,0),(3,2,1),(3,1,0)]          fitness = 135.8800
[(2,1,1),(4,3,1),(4,1,0),(2,1,1),(4,1,0),(3,2,1),(3,1,0)]          fitness = 207.7200
[(3,2,1),(3,1,0),(3,2,1),(2,1,1),(4,1,0),(3,2,1),(3,1,0)]          fitness = 361.0400
[(3,2,1),(2,1,0),(3,2,1),(2,1,1),(4,1,0),(3,2,1),(3,1,0)]          fitness = 405.3200
[(2,1,1),(4,3,1),(4,1,0),(4,3,1),(2,1,1),(4,1,0),(3,1,0)]          fitness = 457.4000
```

The direction toward the fit-chromosome is captured in successive populations, with the fit-chromosome found by iteration 25. At this iteration the average fitness is 172.81, demonstrating the diversity still present in the remainder of the population. As more iterations pass the population may become completely homogenized, at which point the average fitness (avgFit) is equal to zero. Depending on the values set for mutation, crossover, and elitism the population may never find the fit-chromosome; likewise, homogenization may arrive but, after successive populations, drift away from the fit-chromosome.

Below each run of GARthm.py are two lists that summarize the findings of the GA. The first is a list of unique best-fit vectors, or chromosomes with unique allele representations of all best-fit vectors. Because two chromosomes can be rhythmically identical and have different allele representations, the second list removes these redundancies, labeled the “true unique” best-fit vectors.

### GARthm.py in athenaCL

The genetic algorithm GARthm.py is designed such that it can be used as an object from within the command-line program athenaCL. This software functions as both a pitch (class) set utility and as an algorithmic front-end to Csound. Algorithmic composition in Csound is done by creating TextureInstances, algorithmic music layers. Each TextureInstance has a number of configurable parameters, including beat-time (entered as BPM) and rhythm. Rhythm parameters in athenaCL textures are defined by an object type followed by a list of arguments. There are a number of different rhythm-objects available, including rhythmic generators like ‘binaryAccent’ and ‘loop.’

In order to give athenaCL access to GARthm.py, a rhythm-object named “gaRhythm” is made available, encapsulating the functionality of GARthm.py. With this design, each texture that has a gaRhythm instance has a complete, functioning GA, including all its particular settings and the current population’s chromosomes. Figure 8 shows the display of an athenaCL texture with a “gaRhythm” object. (The command “Tiv” displays the parameters of a TextureInstance in athenaCL):

**Figure 8.** gaRhythm in athenaCL TextureInstance View display

```
[PI(chord)TI(drumC)] :: tiv
  TI: drumC, TM: LineGroove, TC: (0)
  PitchMode: pitchSpaceSet, PolyMode: set, TT: 24ToneUpper
  o/+ : +
(i)nstrument      10 (tamHats)
(t)ime range      000.00--015.00
(b)eat            120.00 BPM (0.50 s/beat)
(r)hythm          'gaRhythm', ((4,3,+),(4,3,+),(4,1,o),(2,1,+),(4,1,o),(3,2,+),(3,1,o)), 0.8, 0.18, 0.16
                  (4,3,+) (3,1,o) (4,1,o) (2,1,+) (4,3,+) (3,2,+) (3,1,o)
                  (4,3,+) (3,1,o) (4,1,o) (2,1,+) (4,3,+) (3,2,+) (4,1,o)
                  (3,1,o) (4,3,+) (4,1,o) (2,1,+) (2,1,+) (4,3,+) (3,1,o)
                  (4,3,+) (2,1,o) (4,1,o) (2,1,+) (4,3,+) (3,2,+) (4,1,o)
                  (4,3,+) (3,1,o) (4,1,o) (2,1,+) (2,1,+) (3,2,+) (4,1,o)
(p)ath:          chord
                  (0,1,3,6,8)
local (f)ield     'basketGen', 'orderedCyclic', [0]
local (o)ct       'basketGen', 'randomChoice', [6,6,7]
(a)mplitude       'cyclicGen', 'linearUp', 77.00, 79.00, 0.20
pa(n)ing         'cyclicGen', 'linearDownUp', 0.30, 0.60, 0.03
au(x)illary pfields
  p7              'cyclicGen', 'linearUp', 0.10, 0.90, 0.10
  p8              'cyclicGen', 'linearUp', 1200.00, 1300.00, 20.00
t(e)xture        [], 0
[PI(chord)TI(drumC)] ::
```

The gaRhythm object takes four arguments in athenaCL. The first is a fit-chromosome. In this example the chromosome can be represented as a bar of seven-eight time. See Figure 6 for a transcription of this rhythm. The second argument is a crossover rate (0.8), the third is the mutation rate (.18), and the final value is the percent of elitism (.16). In addition to the values 0 and 1 for the note/rest-status of a rhythm-tuple, athenaCL displays and reads “+” as note, “o” as rest. The beat-time used to calculate the actual durations in this example are given under the heading “beat,” equal to 120 BPM (.5 seconds per beat).

In Figure 8 five “true-unique” chromosomes have been found and are listed below the rhythm-object’s arguments. After removing redundancies, these are the chromosomes with the best fitness of each population run. The gaRhythm object, by default, runs fifty iterations of the GA. For the duration of this particular musical texture (given as time range) rhythms will be read from this list in order. In Figure 8 a small number of chromosomes are found due to the high percentage of elitism. Reducing this elitism value allows the user to obtain a larger number of rhythmic variants, which in many circumstances may be desirable. Figure 9, with a significantly reduced percent of elitism, illustrates this change:

**Figure 9.** gaRhythm in athenaCL TextureInstance View display

```
[PI(chord)TI(drumC)] :: tiv
  TI: drumC, TM: LineGroove, TC: (0)
  PitchMode: pitchSpaceSet, PolyMode: set, TT: 24ToneUpper
  o/+ : +
(i)nstrument      10 (tamHats)
(t)ime range      000.00--015.00
(b)eat            120.00 BPM (0.50 s/beat)
(r)hythm          'gaRhythm', ((4,3,+),(4,3,+),(4,1,o),(2,1,+),(4,1,o),(3,2,+),(3,1,o)), 0.8, 0.18, 0.02
                  (3,1,o) (4,3,+) (4,1,o) (4,2,+) (3,2,+) (3,2,+) (3,1,o)
                  (3,1,o) (4,3,+) (4,1,o) (2,1,+) (4,1,o) (3,2,+) (2,1,+)
                  (3,1,o) (4,3,+) (3,2,+) (4,2,+) (4,1,o) (3,2,+) (4,1,o)
                  (3,1,o) (4,3,+) (4,1,o) (2,1,+) (3,2,+) (3,2,+) (4,1,o)
                  (6,2,o) (4,3,+) (4,1,o) (4,2,+) (2,1,+) (3,2,+) (2,1,+)
                  (3,1,o) (4,3,+) (3,2,+) (2,1,+) (4,1,o) (3,2,+) (4,1,+)
                  (4,1,o) (4,3,+) (4,1,o) (3,1,o) (4,3,+) (3,2,+) (2,1,+)
                  (4,3,+) (4,3,+) (4,1,o) (3,1,o) (3,2,+) (2,1,+) (4,1,o)
                  (3,2,+) (4,3,+) (4,1,o) (3,1,o) (3,2,+) (2,1,+) (3,1,o)
                  (3,2,+) (4,3,+) (4,1,o) (3,1,o) (3,1,o) (9,3,o) (3,2,+) (2,1,+)
                  (6,2,o) (4,3,+) (3,2,+) (3,1,o) (4,1,o) (3,2,+) (2,1,+)
                  (4,1,o) (4,3,+) (4,1,o) (3,2,+) (2,1,+) (4,2,+) (3,1,o)
                  (3,1,o) (4,3,+) (4,1,o) (2,1,+) (3,2,+) (4,2,+) (2,1,+)
                  (3,1,o) (4,3,+) (4,1,o) (3,2,+) (4,3,+) (4,2,+) (12,3,o)
                  (3,2,+) (4,3,+) (4,1,o) (3,1,o) (3,2,+) (4,1,o) (2,1,+)
                  (3,2,+) (4,3,+) (4,1,o) (3,1,o) (3,2,+) (4,2,+) (4,1,o)
                  (4,1,o) (4,3,+) (4,1,o) (3,2,+) (4,3,+) (4,2,+) (4,1,o)
                  (3,1,o) (4,3,+) (4,1,o) (9,6,+) (4,1,o) (3,2,+) (3,2,+) (12,3,o)
                  (3,1,o) (4,3,+) (4,1,o) (3,2,+) (2,1,+) (2,1,+) (12,3,o)
                  (3,1,o) (4,3,+) (4,1,o) (4,1,o) (3,2,+) (4,2,+) (2,1,+)
                  (3,1,o) (4,3,+) (4,1,o) (3,1,o) (3,2,+) (4,2,+) (2,1,+)
                  (3,1,o) (4,3,+) (3,2,+) (4,1,o) (4,3,+) (2,1,+) (12,3,o)
(p)ath:          chord
```

```

(0,1,3,6,8)
local (f)ield      'basketGen', 'orderedCyclic', [0]
local (o)ct        'basketGen', 'randomChoice', [6,6,7]
(a)mplitude      'cyclicGen', 'linearUp', 77.00, 79.00, 0.20
pa(n)ing          'cyclicGen', 'linearUpDown', 0.30, 0.60, 0.03
au(x)illary pfields
  p7              'cyclicGen', 'linearUp', 0.10, 0.90, 0.10
  p8              'cyclicGen', 'linearUp', 1200.00, 1300.00, 20.00
t(e)xture         [], 0

[PI(chord)TI(drumC)] ::

```

Notice that in Figure 9 the fit-rhythm is not found. In most chromosomes the first and last tuple do not match the fit-chromosome. Nevertheless, there is a consistency to these rhythms, expressed in over-all duration, in distribution of rests and notes, and in segments that match the fit rhythm, for instance the numerous appearances of (4,3,1) followed by (4,1,0) in allele positions two and three, and (4,1,0) followed by (3,2,1) in allele positions five and six.

Using athenaCL, users can easily test, audition, and modify the results of the GA to best suit desired aesthetic goals, and incorporate the results in larger musical works. As an algorithmic front end to Csound, athenaCL can be used to create and render Csound scores employing the gaRhythm object, allowing rapid aural feedback.

### Future Work

GArthm.py, and its implementation in athenaCL as the gaRhythm object, are atypical, creative approaches to GAs. Rather than develop a GA to solve a problem efficiently, this approach captures the teleology of a GA as the motion towards a known solution. GArthm.py successfully creates non-deterministic, musically meaningful rhythmic variations within well defined boundaries.

There is room for much expansion of this project. One persistent problem is that chromosomes with large allele alphabets, even with high elitism percentages and two-point crossover, still fail to evolve toward the fit-vector in a reasonable numbers of generations. Further refinement of the fitness function and its weightings could lead to improvements.

Integrated into athenaCL, there are many possible expansions of the GArthm.py object. A higher-level rhythm-object could be developed, built of ordered, independent “gaRhythm” objects. A rhythmic segment in this poly-GA, each constructed from a different fit-chromosome, would consist of independent populations. This would allow variation at the level of segment while maintaining ordered, large-scale unity. Such a rhythm-object would alleviate the problem presented by long chromosomes with large allele alphabets by allowing multiple GAs to deal with numerous short chromosome segments rather than one large chromosome.

### Works Cited

Holland, John: *Adaptation in Natural and Artificial Systems* (Cambridge: MIT Press, 1975)  
 Mitchell, Melanie: *An Introduction to Genetic Algorithms* (Cambridge: MIT Press, 1996)  
 Rowe, Robert: *Machine Musicianship* (Cambridge: MIT Press, 2001)

### Software Resources

athenaCL <http://flexatone.com/athena>  
 GArthm.py <http://flexatone.com/ga/GArthm.py>  
 Python <http://www.python.org>