An Object-Oriented Model of the Xenakis Sieve for Algorithmic Pitch, Rhythm, and Parameter Generation

Christopher Ariza
Graduate School of Arts and Sciences, New York University
ariza@flexatone.net

Abstract
This paper presents an object-oriented model and Python implementation of the Xenakis sieve. In developing a new model of the sieve, Xenakis’s original sieve theory and functional implementation are analyzed and shown to be insufficient. The model presented here offers a practical notation, use of all logical operators, production of sieve segments and transpositions, and easy integration within higher-level systems. This model is deployed within athenaCL, a cross-platform, open-source, interactive command-line environment for algorithmic composition in Csound and MIDI. Within athenaCL, dedicated sieve objects for generating pitch-space sets, rhythms, and parameter values are demonstrated.

1. Introduction

Iannis Xenakis, in at least five documents published between 1967 and 1990, developed an elegant, efficient, and powerful system for generating complex sequences called sieves. Xenakis demonstrated the use of sieves for the generation of pitch scales, rhythms, and a variety of musical parameters. Though sieves are best calculated with the aid of a computer, there is only one software implementation available. This functional, C-based implementation is insufficient for broad use.

This paper introduces a new object-oriented model and Python implementation of the Xenakis sieve. This model offers a practical notation, use of all logical operators, production of sieve segments and transpositions, and easy integration within higher-level systems. This model is deployed within athenaCL, a cross-platform, open-source, interactive command-line environment for algorithmic composition in Csound and MIDI. Within athenaCL, dedicated sieve objects for generating pitch-space sets, rhythms, and parameter values are demonstrated.

2. Definition of a Sieve

Xenakis’s sieve theory changed over the course of his writings. Procedures, notation, and nomenclature were expanded and varied as the theory developed, yet often with inconsistent and unexplained usage. To avoid the complexity of Xenakis’s original presentation, a sieve will be defined with the new model presented here. New terms and concepts are introduced to replace Xenakis’s sometimes inconsistent or incomplete usage.

A sieve consists of one or more residual classes combined by logical operators. A residual class consists of two integer values, a modulus (M) and a shift (I). The modulus can be any positive integer greater than 0; the shift, for a given modulus M, can be any integer between 0 and M-1. A modulus and shift will be notated M@I, read modulus M at shift I. A shift I greater than or equal to M is replaced by the common residue, or I % M. (All computational examples are given in the Python programming language. The “%” is the modulus operator. For example: 13 % 12 == 1)

The residual class defines an infinite number sequence. For a sequence generated by M@I, each value in the sequence mod M is equal to I.

For example, the residual class 3@0 defines an infinite sequence consisting of all integers \(x\) where \(x \mod 3 == 0\). The resulting sieve sequence is \{…, -6, -3, 0, 3, 6, …\}. A residual class with the same modulus and a shift of 1, notated 3@1, produces a sequence where, for each value \(x\), \(x \mod 3 == 1\), or \{…, -5, -2, 1, 4, 7, …\}. For any modulus M, there are M unique shifts (the values 0 to M-1).

Logical operators are used to combine residual classes into complex sieve sequences. The logical operators are union (“or”), intersection (“and”), and complementation (“not”). The logical operators will be notated “|” for union, “&” for intersection, and “-” for complementation.

For example, the sieve 3@0 | 4@0 produces the union of two residual classes, or the sieve sequence \{…, 0, 3, 4, 6, 8, 9, 12, …\}. The intersection of the same two residual classes, notated 3@0 & 4@0, produces the sieve sequence \{…, 0, 12, 24, …\}.

Unlike union and intersection, complementation operates on a single residual class. A residual class under complementation, -M@I, is equivalent to the union of all residual classes of the modulus \{I from 0 to M-1\} excluding the complemented residual class. For example, -3@0 is equal to the sieve 3@1 | 3@2, or the sieve sequence \{…, -7, -5, -4, -2, -1, 1, 2, 4, 5, 7, …\}. Likewise, -5@2 is equivalent to the sieve 5@0 | 5@1 | 5@3 | 5@4. Any residual class
under complementation can be expressed as a union of non-complemented residual classes. The sieve sequence of a complemented residual class, -M@I, is the sequence of all integers not in M@I.

A sieve filters the set of all integers to produce a sieve sequence. A sieve segment, or a finite contiguous section of a sieve sequence, can be extracted for practical deployment. Rather than filtering the set of all integers, a sieve segment filters a finite range of integers. The sieve 3@2, filtering integers from 20 to 38, produces the sieve segment {20, 23, 26, 29, 32, 35, 38}. The set of integers a sieve filters will be called “z.”

A sieve can be transposed by any integer. A sieve transposition is created by adding the transposition value to the shift of each residual class in the sieve. For example, the sieve 5@2 & 2@0 produces the sieve sequence {..., 2, 12, 22, 32, 42, 52, ...}. If this sieve is transposed by a value of 4, the sieve 5@1 & 2@0 results (modulus reduction of 5@6 & 2@4), producing the sieve sequence {..., 6, 16, 26, 36, 46, 56, ...}. A transposition value will be called “n.”

The integers of a sieve sequence or segment have no prescribed value: each integer step can be mapped to any value. (Xenakis calls this unit interval the “elementary displacement,” or ELD.) Thus what a sieve creates is a sequence of points on a line, or a sequence of proportions between these points. This sequence can be applied to any parameter and can be treated as either an ordered or unordered collection. This sequence has a period and an internal symmetry, though such features may be complex and unrecognizable.

Xenakis proves two essential theorems of sieves. First, any finite number sequence can be expressed as a sieve consisting only of residual classes under union. Second, any two (non-complemented) residual classes under intersection can be reduced to a single residual class. It follows that any number of residual classes, if intersected, can be reduced to a single residual class.

3. Xenakis’s Writings on the Sieve

Xenakis’s sieve theory changes over the course of his writings. These changes demonstrate a refinement of Xenakis’s thought, in part a result of his transition from manual sieve calculation on graph paper to computer-aided sieve calculation.

Sieves were first introduced in “Vers un métamusique” (Xenakis 1967), published in La Nef. The first English translation was published in Tempo in 1970, and was included in the 1971 English edition of Formalized Music as chapter VII, “Towards a Metamusic.”

Xenakis returned to discussion of the sieve in “Towards a Philosophy of Music” (1968), first published in French in Revue d’Esthétique, and latter translated and included in the 1971 edition Formalized Music as chapter VIII.

Sieves are discussed in “Concerning Time, Space and Music” (Xenakis 1988), published first in French. In 1989 an English translation of excerpts from this paper appeared in Perspectives of New Music as “Concerning Time.” The complete text was ultimately included as chapter X in the 1991 edition of Formalized Music.

The article “Sieves” (Xenakis 1990) was published in Perspectives of New Music. The 1991 edition of Formalized Music includes nearly the same article as chapter XI. “Sieves” (Xenakis 1990) includes the only published software implementation, written in the C language. Chapter XI in Formalized Music, titled “Sieves: A User’s Guide,” includes nearly the same software implementation.

3.1 The Evolution of Sieve Theory

Xenakis offers two sieve models. The first model is described in his earliest writings (Xenakis 1967, 1968, 1988). The second model is found in his last treatment of the topic and its accompanying software implementation (Xenakis 1990). The writings of the first model express a computationally incomplete system. This is, in part, a result of Xenakis calculating sieve sequences by hand on graph paper. The development of Xenakis’s computer implementation seems to have led to a more complete specification of the model. This second model, however, fails to reconcile certain aspects of the original, and Xenakis gives no discussion concerning this difference. The models differ in the allowed logical operators, the levels of residual class nesting, and the supporting nomenclature and theoretical proofs.

Xenakis’s first model was based on the manual calculation of sieves. Xenakis (1990, 234) provides an image of a hand-written diagram on graph paper labeled “Nomos alpha Sieves.” Each column of the graph is clearly labeled “ELD = 1/4 tone.” Parallel, horizontal lines illustrate a sieve as successions of ticks. Notations label each line as a single sieve operation, such as “∩ (5,2).” Each horizontal line represents one step towards the realization of the sieve sequence. When introducing the sieve Xenakis describes this technique: he sets the ELD to both a “semitone or a millimeter,” and describes notating a sieve on graph paper (Xenakis 1990, 269).

All three logical operators are employed in the first model. Xenakis (1967, 1968, 1988) describes sieve union, intersection, and complementation. In “Sieves” (Xenakis 1990) however, both in prose and in code, there is no mention or example of complementation. Xenakis says nothing about this omission.

Unlimited nesting of residual class intersections, unions, and complementations is employed in the first model. For example, the following sieve demonstrates unordered, three-level groupings:

\((-M@I & M@I & (-M@I | M@I)) | M@I & M@I\)

The second model, by exclusive use of ordered two-level groupings, is simpler. The inner level consists only of
intersections, the outer level consists only of unions, and complementation is not used. This is demonstrated in Xenakis (1990, 271), in both the definition of the theory and the software implementation. The following sieve demonstrates this ordered, two-level grouping:

\[(M@I \& M@I \& M@I) | M@I | (M@I \& M@I)\]

A transition between the two models can be seen in Xenakis’s choice of examples. Xenakis (1967, 197) gives examples of the logical formula of a sieve with three levels of nesting, intermingling all operators. Xenakis (1968) exclusively uses two-level groupings. Intersections and unions are mixed, and complementation is used in intersection. Xenakis (1988) also uses two-level groupings, but here groupings are organized such that the inner level consists only of intersections, while the outer level consists only of unions. Examples of complementation are still given. By Xenakis (1990) two-level groupings are exclusively organized as in Xenakis (1988), but complementation is completely excluded.

These changes reflect a refinement of sieve theory. Xenakis organized groupings and reduced the use of logical operators because it became apparent through software modeling that these features were superfluous and inelegant. Two proofs, introduced only in Xenakis (1990), support this claim and define the difference between the first and the second models. These proofs provide a computationally complete model.

Xenakis (1990, 275) provides an algorithm for the derivation of a logical sieve from a sequence. This proof demonstrates that any finite number sequence can be represented as a logical sieve created exclusively from unions of residual classes. Thus intersection, complementation, and multiple levels of embedding are not required to construct a sieve of any complexity. For example, with this algorithm the sequence \{0, 3, 6, 9, 12, 13\} is shown to be a segment of the sieve \{3@1 \& 7@6\}; the sequence \{2, 5, 14, 23, 41\} is shown to be a segment of the sieve \{21@2 \& 18@5 \& 27@14\}.

Union, it is shown, is the only necessary logical operator: intersection and complementation are superfluous. In Xenakis’s second model, complementation is removed, yet intersection is retained. This peculiarity is explained by an algorithm, introduced in the same text, that allows any two residual classes, upon intersection, to be reduced to a single residual class (1990, 271). Since any two intersected residual classes can be reduced to a single residual class, it follows that any number of intersected residual classes can be reduced to a single residual class. For example, \{3@2 \& 8@0\} is equivalent to 24@8; \{3@2 \& 4@7 \& 6@11 \& 8@7\} is equivalent to 24@23.

If intersection can reduce any number of residual classes to a single residual class, a two-level grouping of residual classes, where the inner group consists exclusively of intersections, can be resolved to a single group of unions. Intersection is used not out of necessity, but as a notational convenience. For example, the following sieve contains four groups of intersections joined by union:

\[(3@2 \& 4@7 \& 6@11 \& 8@7) | (6@9 \& 15@18) | (13@5 \& 8@6 \& 4@2) | (6@9 \& 15@19)\]

Applying Xenakis’s algorithm, this complex sieve is reduced to a single group of unions:

\[24@23 | 30@3 | 104@70 | 570@285\]

If a complemented residual class is included in a group of intersections, the group cannot be reduced to a single residual class. As shown above, a complemented residual class is equivalent to a group of unions. Thus if a group of intersections contains a complemented residual class, it contains both intersections and unions, and cannot be further reduced.

### 3.2 The Evolution of Sieve Notation

As Xenakis’s sieve model evolved, his notation changed. As supported above, these changes are a direct result of working with the computer, where logical symbols such as \(\cup\) and \(\cap\) are not easily available at the keyboard. The notation presented in Xenakis’s software implementation (1990) offers an alternative, but requires further refinement. This paper presents a new notation with greater string parsing flexibility, visual clarity, and ease of use.

A sieve notation must include a means of specifying a residual class as a modulus and a shift, and symbols for the logical operators union, intersection, and complementation. Symbols indicating residual class grouping may optionally be included.

Xenakis’s early writings (1967, 1968, 1988) employ a complex notation. Shifts are represented as subscripts of the modulus. Traditional logical symbols are used: a \(\cup\) for union, a \(\cap\) for intersection, and an over-score line for complementation. (For typographic simplicity, over-score is replaced by underline below.) Transposition is represented by the variable \(n\), and is included with every subscript as “\(n+\text{shift}\)” or “\(n\)” (when shift is equal to zero). Embedding and grouping of residual classes is notated with parenthesis. For example:

\[(3_{n+2} \cap 4_{n}) \cup (3_{n+1} \cap 4_{n+1}) \cup (3_{n+2} \cap 4_{n+2}) \cup (3_{n} \cap 4_{n+3})\]

This notation is used in all early writings. For the two proofs presented in “Sieves” (Xenakis 1990), an algebraic notation is introduced. Rather then using subscripts, residual classes are represented as a number pair, given either with integers or variables (where \(M\) represents modulus and \(I\) represents shift). Logical operators are notated as before. Transposition, as the variable \(n\) or otherwise, is no longer notated. Intersection groups are notated with curly braces.
and joined with a union operator. The following example demonstrates this notation:

\[
\{(3,2) \cap (4,7) \cap (6,11) \cap (8,7)\} \cup \{(6,9) \cap (15,18)\} \cup \\
\{(15,5) \cap (8,6) \cap (4,2)\} \cup \{(6,9) \cap (15,19)\}
\]

The C-language implementation accompanying Xenakis (1990) uses an ASCII symbol representation. The use of ASCII ensures the availability of every character at a computer keyboard. Modulus and shift are represented as a number pair. Intersection is represented as “*,” union is represented as “+,” and complementation is not specified. Transposition is not noted. A sieve is exclusively represented as a union group, each member consisting of any number of intersections. Intersection groups are notated with square brackets. For example:

\[
[(3,2) * (4,7)] + [(6,9) * (15,18)]
\]

Xenakis’s notation, though effective, can be further refined. Like the notation presented in Xenakis (1990), the new notation presented here is also an ASCII representation. A sieve is notated by a “logical string.” Residual classes are given as a modulus and a shift separated by an “@” symbol. In the case that shift is equal to zero, shift is not notated. Logical operators are notated using the pipe (“|”) for union, the ampersand (“&”) for intersection, and the negative sign (“-”) for complementation. All three logical operators are permitted, but complemented residual classes cannot be combined under intersection. Groupings follow Xenakis’s two-level order, but no symbols are used for demarcation. A simple rule of precedence is established: all intersections are reduced before unions. A sieve consisting of a single residual class is always notated with a single union operator. The example given above, in this notation, would be:

\[
3@2 & 4@7 \ | \ 6@9 \ & \ 15@18
\]

This notation has many advantages. It is more compact and requires fewer characters than Xenakis’s notations. The use of bitwise logical symbols (pipe and ampersand) are transparent in meaning to those familiar with programming languages. The replacement of grouping symbols with a precedence rule simplifies the notation. Finally, the sieve contains no commas or (optionally) no spaces, allowing for easy parsing and isolation when passed as an argument or included with complex data.

5. Object-Oriented Implementation

Due to these design flaws, the functional sieve implementation (Xenakis 1990) is not ported. Rather, an object-oriented Python implementation is developed based on the new model and notation presented above. This implementation is portable, modular, and offers easy creation and deployment of sieve segments and transpositions. The file “sieve.py,” part of the athenaCL (Ariza 2004) library libATH, implements this model as the Residual, Sequence, and Sieve objects. Algorithms for the intersection of two residual classes and the calculation of the logical formula from a sequence are derived from proofs in Xenakis (1990).

5.1 The Residual Class

The Residual object is a representation of the residual class. A Residual contains data attributes for modulus (m), shift (shift), complement (neg), and integer range (z). Modulus and shift are integer values as defined above. Complementation is denoted as a boolean value (neg) and, unlike other logical operators, is stored with the Residual object. Each Residual class contains a reference to a finite integer list (z) from which sieve segments are filtered.
Object initialization requires a modulus \((m)\) value. Optional arguments can be provided for shift, neg, and \(z\). If no \(z\) is given, a default range is provided.

The method “\(n\)” of a Residual instance outputs a transposed sieve segment. This method has two optional arguments: an integer value for transposition \((n)\), by default zero, and a list of integers \((z)\), by default the \(z\) set at initialization. The method “\(\text{setZ}\)” can be used to set the \(z\) attribute of a Residual object. The method “\(\text{intersection}\)” returns a new Residual instance that is the reduced intersection of two residual classes. The “\(\text{repr}\)” method provides various string representations of the logical sieve, including Xenakis-style notations.

The “\(\text{intersection}\)” method employs the logical proof given in Xenakis (1990, 271-2). As already mentioned, Xenakis’s algorithm does not accommodate the intersection of a complemented residual class. Nonetheless, in this model a single Residual instance can be complemented (by setting the “\(\text{neg}\)” boolean) and produce correct sieve segments alone or under union.

Operator overloading is employed to provide greater flexibility. The object’s \(\text{__call__}\) method is mapped to the “\(\text{n}\)” method; the \(\text{__str__}\) method is mapped to the “\(\text{repr}\)” method, and the \(\text{__and__}\) method is mapped to the \(\text{intersection}\) method. In order to compare two objects, the equal and not-equal operator methods \(\text{(\_eq\_ and \_ne\_)}\) are defined by comparing \(m\) and \(shift\) attributes.

(In Python, all objects can overload operators simply by defining specially named methods. The \(\text{__call__}\) method is called when an object \(x\) is evaluated as \(x()\). The \(\text{__str__}\) method is called whenever a string representation of an object is needed; \(\text{print x}\). The \(\text{__and__}\) method is called with the operator “\&”; the \(\text{__eq__}\) and \(\text{__ne__}\) methods are called with “==” and “!=” operators respectively.)

The following UML class diagram summarizes the public attributes and methods of this object.

<table>
<thead>
<tr>
<th>Residual</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>m</td>
<td>shift</td>
</tr>
<tr>
<td></td>
<td>neg</td>
</tr>
<tr>
<td></td>
<td>(z)</td>
</tr>
<tr>
<td>(\text{<strong>call</strong>}(,,,))</td>
<td>(\text{n})</td>
</tr>
<tr>
<td>(\text{<strong>and</strong>}(,,,))</td>
<td>(\text{intersection})</td>
</tr>
<tr>
<td>(\text{<strong>str</strong>}(,,,))</td>
<td>(\text{repr})</td>
</tr>
<tr>
<td>(\text{<strong>eq</strong>})</td>
<td>(\text{<strong>ne</strong>})</td>
</tr>
</tbody>
</table>

**Figure 1.** Residual object class diagram.

The following Python interactive session demonstrates the Residual object:

```python
>>> from athenaCL.libATH import sieve
>>> a = sieve.Residual(3,2)
>>> print a
3@2
```

5.2 The Sequence Class

The Sequence object is employed exclusively within a Sieve instance. The Sequence object takes a list of any integers and returns a list of Residual instances. These Residual objects, when combined by union, will reproduce the initial sequence. Alternatively, a Sequence object can produce the logical string necessary to construct a Sieve object.

Sequence objects are initialized with a list of integers stored as the “\(\text{match}\)” attribute. A “\(\text{z}\)” list is constructed by taking the range of integers from the minimum to the maximum of the match list. Each value in the match list is then passed to the private \(\text{“\_find\_”}\) method.

The \(\text{“\_find\_”}\) method is an implementation of the prose algorithm provided in Xenakis (1990, 274) for the calculation of a sieve from a sequence. The single argument required by the \(\text{“\_find\_”}\) method (a value from the “\(\text{match}\)” list), is treated as a shift value. A Residual object is created with a modulus of 2, the provided shift, and the \(\text{z}\) list calculated at initialization. A sieve segment is created by calling the Residual instance. If this sieve segment is a subset of “\(\text{match}\),” an identical Residual object has not already been found, the object is appended to the list “\(\text{residuals}\).” Otherwise, \(m\) is incremented and the process is repeated until a subset sieve segment is found.

The only public methods of Sequence are provided through the \(\text{__call__}\) and \(\text{__str__}\) methods. The \(\text{__call__}\) method returns the list of Residual objects, suitable for creating a Sieve. The \(\text{__str__}\) method returns the logical string representation of the sieve, also suitable (though less efficient) for creating a Sieve.

The following Python interactive session demonstrates the Sequence object:

```python
>>> from athenaCL.libATH import sieve
```

Proceedings ICMC 2004
>>> a = sieve.Sequence((2,4,6,8))
>>> print a
|2
>>> b = sieve.Sequence((0,3,6,9,12,13))
>>> print b
3|7@6
>>> c = sieve.Sequence((2,5,14,23,41))
>>> print c
21@2|18@5|27@14

5.3 The Sieve Class

The Sieve class combines Residual objects, the Sequence object, and an advanced string parser to produce a Sieve object with broad utility.

The Sieve object takes either a string or a list upon initialization. If the argument is a string, private methods are used to parse the string into Residual instances. (Alternative sieve notations, such as Xenakis-style number pairs, are also accepted.) For each intersection group, each Residual object is instantiated and intersected (via the “&” operator) until one Residual object remains. This Residual object is appended to the “union” list, which represents the group of all unions. Alternatively, if the initialization argument is a list of integers, the Sequence object is used to construct a list of Residual instances, and this list is assigned to the “union” list.

The public methods of the Sieve object are similar to those found in the Residual class. The “n” method is mapped to the __call__ method and accepts optional arguments for n and z. The __str__ method returns the logical string.

When the “n” method is called, each Residual instance in the union list calls its own “n” method, returning a sieve segment for the given n and z values. These segments, as lists, are then unified and returned to the user as a complete sieve segment.

The following UML class diagram summarizes the public attribute and methods of this object.

![Sieve class diagram]

Figure 2. Sieve object class diagram.

The following Python interactive session demonstrates the Sieve object:

```python
>>> from athenaCL.libATH import sieve
>>> a = sieve.Sieve('3@1|4')
>>> print a
3@1|4
>>> a()
[0, 1, 4, 7, 8, 10, 12, 13, 16, 19, 20, 22, 24, 25, 28, 31, 32, 34, 36]
>>> a(2, range(-10,10))
[-10, -9, -6, -3, -2, 0, 2, 3, 6, 9]
>>> b = sieve.Sieve((0, 1, 4, 7, 8, 10, 12, 13, 16, 19))
>>> print b
4|3@1
```

6. Integration in athenaCL

The objects provided in sieve.py can be used in isolation or in cooperation with other software. Within athenaCL high-level, practical object interfaces provide access to sieve functionality. These interfaces provide tools for the algorithmic composition of pitches, rhythms, and general parameter values.

6.1 Sieve Pitch Generation

The athenaCL system features specialized objects for dealing with pitch, multisets (pitch space sets allowing redundancy), and paths (ordered multisets). When using athenaCL for composition, the path defines pitch organization. Pitch groups, entered either via user-interface or command line, can be provided in many representations. In addition to logical sieve strings, pitch-space sets, pitch names, and Forte names are all accepted.

The sieve segment is constructed within a chromatic pitch range. When entering a sieve, the logical string can be followed by two arguments for lower and upper bounding pitches.

In the following athenaCL interactive session the user creates a path named “a” (with the PathInstance New command “pin”) from three sieves. Note, in the creation of path “b,” the same path is constructed from a single space-separated command-line argument. User input is given in bold.

```
:: pin
please name this PathInstance: a
enter a pitch set, sieve, or set-class: 3|4,c4,c5
SC 6-28 as (C4,D#4,E4,F#4,G#4,A4)?
(y, n, or cancel): y
add another set? (y, n, or cancel): y
enter a pitch set, sieve, or set-class: 5@1|7@3|10@8|15@12|20@16|25@20|30@24|35@29
add another set? (y, n, or cancel): y
enter a pitch set, sieve, or set-class: SC 8-11B as (C#3,E3,G#3,D4,A#4,C5,F5,A#5)?
(y, n, or cancel): y
add another set? (y, n, or cancel): y
```

```python
>>> c = sieve.Sieve('5@4 | 3@2 | 7@3')
>>> print c
5@4|3@2|7@3
>>> c(0, range(0,36))
[2, 3, 4, 5, 8, 9, 10, 11, 14, 17, 19, 20, 23, 24, 26, 29, 31, 32, 34, 35]
>>> c(7, range(0,36))
[0, 1, 3, 6, 9, 10, 11, 12, 15, 16, 17, 18, 21, 24, 26, 27, 30, 31, 33, 36]
>>> d = sieve.Sieve((2, 3, 4, 5, 8, 9, 10, 11, 14, 17, 19, 20, 23, 24))
>>> print d
3@2|7@3|5@4
```

Proceedings ICMC 2004
Path “a,” realized below as a succession of simultaneities, demonstrates the ease with which sieves can be used to create complex pitch structures.

Figure 3. Sieve-generated path as simultaneities.

Sieves are integrated throughout athenaCL such that wherever a pitch group is required a sieve may be submitted. The command SCv, for SetClass View, provides useful analysis information about a set, including normal form and subset vectors. (The SCv command is made available on-line through a web-browser as part of athenaCL netTools.) The following example analyzes a complex sieve as a set-class.

```
:: scv
tie
event (4,1,1)
```

Xenakis (1990, 269) demonstrates using a sieve to generate a rhythm by setting the ELD to a duration. The athenaCL “pulseSieve” object provides similar functionality. When specifying a RhythmObject in athenaCL, the user must provide a list of comma-separated arguments. The arguments for pulseSieve are as follows: object name, sieve logical string, length of sieve to use (creating a z from 0 to length), and an ELD pulse duration (specified as a Pulse object).

The following example demonstrates assigning a pulseSieve object to the rhythm parameter of texture “a.” The sieve given here, “4|3,” will generate a three-over-four hemiola. A z from 0 to 12 designates a sieve segment of one period. The ELD, or duration of each pulse, will be a triplet-eighth note, given as (3,1,1).

```
:: ti:
tie
```

The following rhythm will be looped over the duration of the texture. The sieve provides a compact and precise representation of the resulting sieve segment: (0, 3, 4, 6, 8, 9). The resulting rhythm will be constructed by placing articulations at each point on the sieve segment, and rests on all other points within the z list.

```
:: tiv
```

Textures in athenaCL, or algorithmic music layers, feature dedicated objects to control musical parameters. The texture object manages the deployment and interaction of parameter objects. Rhythms are generated using specialized parameter objects. Pulse objects, the rhythm primitive in athenaCL, are notated in terms of a divisor, a multiplier, and an accent. The divisor is a fraction of a beat; the multiplier scales the fractional division, and the accent can code for notes (value of 1) or rests (a value of 0). Thus, at quarter-note = 60 BPM, a pulse object of (4,3,1) would produce a dotted eighth note duration equal to .25 seconds. A pulse object of (4,3,1) would produce a dotted eighth note duration equal to .375 seconds.

Proceedings ICMC 2004
6.3 Sieve Parameter Generation

With the exclusion of rhythm and pitch, all other note event attributes in athenaCL may be controlled by any of a wide library of general parameter objects. Xenakis describes the utility of controlling many attributes of a note event with a sieve. The athenaCL parameter object “valueSieve” facilitates flexible generation of values with a scalable ELD. The arguments for valueSieve are as follows: object name, sieve logical string, length, scalar, shift, and control. Sieve logical strings and length work as in pulseSieve. The resulting sieve is scaled between the unit interval. The values of this sieve are then multiplied by the scalar and added to shift. Values from the resulting list are chosen in a variety of fashions, such as “orderedCyclic” or “randomChoice.”

The following example demonstrates assigning the amplitude of texture “a” to a complex sieve. The resulting values are depicted in the TiV display.

```
:: tie
edit TI a
which parameter? (i,t,b,r,p,f,o,a,n,x,e): a
current (a)mplitude: cyclicGen, linearDown, 60.00, 70.00, 4.00
new value: vs, 984 & 684 | 783, 40, 50, 20, oc
TI a: parameter (a)mplitude updated.
```

```
p{auto} t(a) :: tiv
TI: a, TM: LineGroove, TC: (0), TT: TwelveEqual
PitchMode: pitchSpaceSet, PolyMode: set, MidiPgm: pianol
  o/+: +
(i)nstrument 10 (tamHats)
(t)ime range 000.00--015.00
(b)pam constant, 30
(r)hythm pulseSieve, 4|3, 12, (3,1,+) (0,3,4,6,8,9)
(p)ath auto
  (8,4,5), (5,11,0), (4,3,1,0) 5.00(a), 5.00(a), 5.00(a)
local (f)ield constant, 0
local (o)ctave constant, 8
(a)mplitude valueSieve, 18@4|7@3, 40, 50.00, 20.00, orderedCyclic
  (20.0,21.43,30.0,40.0, 47.14,50.0,60.0,70.0)
pa(n)ing constant, .5
au(x) pfields p7 constant, .5
  p8 constant, 800
t(e)xture e0 parallelMotionList, [], 0
e1 nonRedundantSwitch, on
```

The sieve has produced eight values within the z defined by the length argument. These eight values are normalized, scaled, and shifted, producing floating-point amplitude values in the range of 20 to 70: {20.0, 21.43, 30.0, 40.0, 47.14, 50.0, 60.0, 70.0}. For each note event, unless behavior is changed by the TextureModule, a value is drawn from this list in order.

7. Future Work

With the sieve Xenakis sought to “add to our arsenal sharper tools,” tools of “trenchant axiomatics and formalization” (1967, 194). The sieve model presented above shares the elegance and power of Xenakis’s original theory, while adding a complete, modular, and efficient implementation. With this new model rigorous exploration of the sieve is possible, including applications in algorithmic composition, algorithmic synthesis, or even computer graphics. Within athenaCL, such research might include the systematic investigation of the relationship between set-class and sieve, the use of sieves to control large-scale textural or melodic shapes, and the development of parameter objects that employ multiple sieves.

References


