Christopher Ariza
New York University
Graduate School of Arts and Science,
Music Department
24 Waverly Place, Room 268
New York, New York 10003 USA
ariza@flexatone.net

The Xenakis Sieve as Object: A New Model and a Complete Implementation

In numerous publications from 1965 to 1992, composer, architect, and theorist Iannis Xenakis (1922–2001) developed an elegant and powerful system for creating integer-sequence generators called *sieves.* Xenakis used sieves (*cribles*) for the generation of pitch scales and rhythm sequences in many compositions, and he suggested their application to a variety of additional musical parameters. Though sieves are best calculated with the aid of a computer, no complete implementation has been widely distributed. Xenakis's published code is incomplete and insufficient for broad use.

This article demonstrates a new object-oriented model and Python implementation of the Xenakis sieve. This model introduces a bi-faceted representation of the sieve, expands Xenakis's use of logic operators, employs a practical notation, produces sieve segments and transpositions, and easily integrates within higher-level systems. This modular implementation is deployed within athenaCL, a cross-platform, open-source, interactive command-line environment for algorithmic composition using Csound and MIDI. High-level, practical interfaces have been developed to provide athenaCL users with sieve-based tools for the algorithmic generation of pitches, rhythms, and general parameter values.

Definition of the Sieve

Xenakis's theory changed over the course of his many writings on the sieve. Procedures, notation, and nomenclature varied as the theory developed, yet often with inconsistent and unexplained usage. To avoid the complexity of Xenakis's presentation, a sieve will be defined with a new model and notation. New terms and concepts are introduced to replace Xenakis's sometimes-inconsistent usage. Alternative theoretical treatments of the sieve have been proposed by others (Riotte 1979; Squibbs 1996; Gibson 2001; Jones 2001; Andreatta 2003), though none have integrated a complete software model.

A sieve is a formula consisting of one or more residual classes combined by logic 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 or equal to 0; the shift, for a modulus *M* greater than 0, can be any integer from 0 to *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 in this article are given in the Python programming language, where the "%" is the modulus operator. For example: 13 % 12 == 1.)

The residual class defines an infinite number sequence. Given a sequence generated by *M@I,* each value in the sequence modulus *M* is equal to *I.* For example, the residual class 3@0 defines an infinite sequence consisting of all integers *x* where *x* % 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* % 3 == 1, or [. . . , –5, –2, 1, 4, 7, . . .]. For any modulus *M,* there exist *M* unique shifts (the values 0 to *M*–1). For each shift of *M,* a sequence of equally spaced integers is produced, where the difference between any adjacent integers is always *M.* A residual class produces a periodic sequence with a period equal to *M.*

Although modulus by zero, and thus zero-division, is typically undefined for real numbers (and an error for computers), the residual class 0@0 is permitted, defining the empty sieve sequence: []. The residual class 1@0, the complement of 0@0, defines the infinite integer sequence $\mathbb{Z}$.

Logic operators are used to combine residual classes. Four operators are permitted: union ("or"), intersection ("and"), symmetric difference ("xor"), and complementation ("not"). Union, intersection, and symmetric difference are binary operators; complementation is a unary operator. The logic operators will be notated "|" for union, "&" for inter-

Computer Music Journal, 29:2, pp. 40–60, Summer 2005

section, "^" for symmetric difference, 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 residual classes, notated 3@0 & 4@0, produces the sieve sequence [. . . , 0, 12, 24, . . .]. The symmetric difference, or the values in each residual class and not in both residual classes, notated 3@0 ^ 4@0, produces the sieve sequence [. . . , –3, 3, 4, 6, 8, 9, 15, . . .].

Unlike union, intersection, and symmetric difference, unary complementation operates on a single residual class or a group of residual classes. Binary complementation is not permitted. The sieve sequence of a complemented residual class, *–M@I,* is the sequence of all integers not in *M@I.* 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.

Operator precedence, from most- to least-binding, is in the following order: unary complementation, intersection, symmetric difference, union. By using braces ("{" and "}") as delimiters, a sieve can employ unlimited nesting. A delimited collection of residual classes is always evaluated before residual classes on the same hierarchical level. For example: 7@1 | 3@2 & 4@3 & –9@1 is evaluated 7@1 | {3@2 & 4@3 & {–9@1}}. A delimited group can be complemented.

There exist two types of sieves: simple and complex. A simple sieve uses at most a two-level, ordered grouping of residual classes, in which the inner level uses intersection, the outer level uses union, and complemented residual classes are never intersected. A maximally simple sieve is made of any number of single residual classes combined only by union. A complex sieve uses residual classes with any combination of logic operators at any hierarchical level, with hierarchical levels of unlimited depth.

A sieve filters the set of all integers to produce an infinite sieve sequence. As all residual classes are periodic, all sieves are periodic. The period of a sieve is equal to the lowest common multiple (LCM) of all residual class moduli. In the case of some sieves, the period is easily apparent. The sieve 3@2 | 4@1, for example, has a period of 12. In the case of other sieves, however, the period can be so large as to be practically unrecognizable.

A sieve segment, or a finite contiguous section of a sieve sequence, can be extracted for practical deployment. Rather than filtering the all-integer set, a sieve segment filters a finite range of integers. The set of integers filtered will be called *z* and is specified *z* = [*a*, . . . , *b*], where *a* is the minimum, and *b* is the maximum. The sieve 3@2 | 4@1 with *z* = [–37, . . . , –25], produces the sieve segment [–37, –35, –34, –31, –28, –27, –25].

The integers of a sieve segment can be thought of as points; the remaining integers found in *z* but not in the sieve segment can be thought of as slots. An integer representation of a sieve segment provides only the points, ignoring the slots. The internal slots between integers can be deduced; external slots beyond the minimum and maximum of the sieve segment, however, cannot be determined without knowledge of *z.* If external slots are not represented, for example, the segment may appear symmetrical when, in terms of its *z*-relative position, it is not.

There are four practical representations, or formats, of a sieve segment. Three are slot-discarding: integer, width, and unit segments; one is slot-retaining: binary segments. Integer segments discard external slots and may distort *z*-relative position. Width segments measure the distance from one point to the next, counting the point itself and the intervening slots. Width segments, like integer segments, discard external slots and may distort *z*-relative position. Unit segments map *z* to the unit interval and translate segment points into real numbers between 0 and 1. Because unit segments do not treat points as integers, both internal and external slots are discarded, and *z,* normally discrete, becomes a continuous range. A unit segment contains no information on the number of slots between points, yet accurate proportional spacing, including *z*-relative position, is retained. Binary segments retain all slots, both internal and external, and thus provide the must complete representation. Figure 1 summarizes the features of sieve segment representations.

Figure 1. Sieve segment formats for sieve 3@2 | 4@1, where z = [7, . . . , 16].

Format	Sieve segment	*z*	Internal slots	External slots	*z*-relative position
integer	[8, 9, 11, 13, 14]	discrete	retain	discard	discard
width	[1, 2, 2, 1]	discrete	retain	discard	discard
unit	[0.11, 0.22, 0.44, 0.66, 0.77]	continuous	discard	discard	retain
binary	[0, 1, 1, 0, 1, 0, 1, 1, 0, 0]	discrete	retain	retain	retain

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*.

When *z* is discrete (integer, width, and binary segments), the integer step can be mapped to any value. The integer step will be called the elementary displacement, or ELD. When *z* is continuous (unit segments), points can function as floating-point scalars of any value. Thus, what a sieve creates is a sequence of points on a line, a sequence of proportions between these points, or a distribution of points within a range. This sequence can be treated as an ordered or unordered collection.

Simple and complex sieves can undergo compression. There are two forms of compression: by intersection and by segment. Compression always results in the production of a maximally simple sieve. A maximally simple sieve cannot be further compressed. A sieve that has not been compressed is an expanded sieve.

A simple sieve can be compressed by intersection. Compression by intersection is a process of combining all residual classes within inner intersection groups into a single residual class. A maximally simple sieve, a collection of single residual classes combined by union, results. Compression by intersection is non-lossy: sieve segments produced with any *z* will be identical for both compressed and expanded sieves.

A complex sieve can be compressed only by segment. Compression by segment requires generating a sieve segment from a complex sieve, then resampling the values within this set to produce a maximally simple sieve. Compression by segment can be lossy. The compressed and expanded sieves will generate identical sieve segments only for the *z* provided during compression. Segments generated with the compressed sieve beyond this *z* may deviate from segments generated with the expanded sieve. Increasing the size of *z,* and thus the size of the segment sampled, improves the quality of compression. Compression by segment with a *z*-length equal to the sieve period, or even multiple periods, may not result in a compressed sieve that, when compared to the expanded sieve, produces identical segments for all *z*.

Sieve compression is defined by two theorems provided by Xenakis. First, 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. This is compression by intersection. Second, any finite integer set can be expressed as a maximally simple sieve. This is compression by segment.

The Xenakis sieve is a set theory of infinite periodicities. This sieve is a unique structure. Certainly, the concept of selecting a collection of numbers by removing elements from a set is common in mathematics (Hawkins 1958). Few mathematical sieves, however, have the sole goal of creating geometrically and aesthetically pleasing structures for sonic deployment. As Xenakis states, "the image of a line with points on it, which is close to the musician and to the tradition of music, is very useful" (Xenakis 1996, p. 147). Such an image is provided by the sieve.

History and Foundations of the Sieve

In 1963, Aaron Copland invited Xenakis to teach at the Berkshire Music Center at Tanglewood. Xenakis's notes and sketches from that summer contain his first experiments with sieves (Barthel-Calvet 2001). As a result of a Ford Foundation grant, Xenakis lived in Berlin from the fall of 1963 to the spring of 1964. During this time, he developed sieve theory further (Barthel-Calvet 2001). Xenakis's theory of sieves can be seen in the context of his interest in number sequences, his use of logic operators with screens and groups, and his desire to develop "outside-time" musical structures.

The sieve produces numerical sequences. Xenakis's interest in the musical deployment of numerical sequences has been frequently demonstrated. An example can be seen in his prominent use of the Fibonacci series in his early compositions *Zyia* (1952) and *Sacrific* (1953; see Solomos 2002, pp. 26–29). Xenakis may have had a similar interest in the series of prime numbers. The sieve of Eratosthenes of Cyrene (c. 276–c. 194 BCE), a well-known method of generating prime numbers (Horsely 1772), may have inspired Xenakis's sieve (Flint 1989).

The sieve of Eratosthenes can be explained through the following steps: (1) list a range of ordered integers starting from 1; (2) let *M* be 2; (3) then select *M:* this value is prime; (4) eliminate all values that are multiples of *M* (i.e., $M \times 2$, $M \times 3$, $M \times 4$, . . .); (5) set *M* to the next available (i.e., not eliminated) value; (6) if integers remain, return to step 3. After *M* exhausts all available values, only primes will remain.

This process has features in common with the Xenakis sieve. The collected multiples of *M* are similar to the periodic integers defined by a residual class; combining these prime-number multiples is similar to the union of multiple residual classes. Multiples are used here, however, to generate slots, not points, and they exclude their first multiple (the prime).

The sieve combines structures with logic operators. Xenakis's writings collected in the 1963 edition of *Musiques Formelles* demonstrate similar combinations with different materials. Chapter 2, "Markovian Stochastic Music," introduces the concept of screens (1963; 1990, p. 51). A screen represents a moment in time with a two-dimensional plane of frequency and intensity; this plane can be populated with events, or what Xenakis calls grains. Xenakis employs the logic operators union, intersection, and complementation to "envisage in all its generality the manner of combining and juxtaposing screens" (1963; 1992, p. 57). *Analogique A* (1958) and *Analogique B* (1959) are offered by Xenakis as demonstrations of this technique (1963; 1992, pp. 98–108). The sieve might be seen as a one-dimensional screen. In chapter 3, "Symbolic Music," logic operators are further used by Xenakis to combine groups of musical materials. The pitch groups of *Herma* (1960–1961), for solo piano, are offered by Xenakis as a demonstration of this technique (Xenakis 1963; 1992, p. 175).

The sieve is a generator of outside-time structures. To Xenakis, a structure that is outside-time is unique regardless of order or temporal deployment. This contrasts with an "in-time" structure, which only retains identity in an ordered or temporal deployment (1992, p. 207). Using the standard transformations of a twelve-tone row as an example, a retrograde transformation is an "in-time" operation, whereas inversion, in producing new intervals, is an "outside-time" operation: "of the four forms of the series, only the inversion of the intervals is related to an outside-time structure" (Xenakis 1992, p. 193). Xenakis, taking an historic view, often lamented the decline of outside-time structures to in-time structures in Western music. He offered the sieve as an antidote.

Xenakis wrote frequently about sieves. There are at least five unique discussions of the sieve; each is found in multiple translations and editions. (Throughout this article, each discussion will be referenced by the date of the earliest document, followed, when necessary, by the date of the actual citation.)

Sieves were first introduced in "La voie de la recherche et de la question" (Xenakis 1965), published in *Preuve* and later reprinted in *Kéleütha* (1994). Xenakis further demonstrated the sieve in "Vers une philosophie de la Musique," published in French in *Gravesaner Blätter* (1966), *Revue d'Esthétique* (1968), and as chapter 6 in *Musique Architecture* (Xenakis 1976). This text, detailing the application of sieves in *Nomos Alpha* (1965–1966),

was later translated to English by John and Amber Challifour and included in the 1971 edition of *Formalized Music* as chapter VIII. A detailed presentation of sieves occurs in the 1967 article "Vers une métamusique," published first in *La Nef,* and subsequently republished as chapter five of *Musique Architecture* (Xenakis 1976). The first English translation by G. W. Hopkins was published in *Tempo* in 1970 and was included in the 1971 English edition of *Formalized Music* as chapter VII, "Towards a Metamusic." Sieves are also discussed in "Redécouvrir le Temps" (Xenakis 1988). In 1989, excerpts from this article, translated into English by Roberta Brown, appeared in *Perspectives of New Music* as "Concerning Time." The complete text, titled "Concerning Time, Space and Music," was included as chapter X in the 1992 edition of *Formalized Music.*

The article "Sieves" (Xenakis 1990), translated into English by John Rahn, was first published in *Perspectives of New Music.* Nearly the same article is included as chapter XI in the 1992 edition of *Formalized Music.* "Sieves" (Xenakis 1990) includes the only published software implementation of the sieve, written in the C language. Chapter XI of *Formalized Music,* titled "Sieves: A User's Guide," includes nearly the same software implementation.

Xenakis frequently mentioned composing with sieve structures. Xenakis describes sieve-based pitch structures in *Akrata* (1964–1965; Xenakis 1966), *Nomos Alpha* (1965–1966; Xenakis 1966), *Jonchaies* (1977; Varga 1996, p. 164), *Pléïades* (1978; Varga 1996, p. 180), and *Aïs* (1979; Varga 1996, p. 165). Xenakis describes sieve-based time structures in *Psappha* (1975; Emmerson 1976, p. 24) and *Komboï* (1981; Varga 1996, p. 71). In addition to Xenakis's explicit statements, analysts have found sieve structures in numerous compositions including *Eonta* (1963; Barthel-Calvet 2000), *Anaktoria* (1969; Solomos 1996 p. 93), *Persephassa* (1969; Harley 2004, p. 64), *Pléïdes* (1978; Harley 2004, p. 121) *Pour la Paix* (1981; Solomos 1996, p. 93), *Embelllie* (1981; Solomos 1996, p. 93), *Mists* (1981; Squibbs 1996, 2002), *Neküia* (1981; Gibson 2001), *Serment-Orkos* (1981; Solomos 1996, p. 93), *À R. (Hommage à Maurice Ravel)* (1987; Squibbs 1996, p. 61), *Tetora* (1990; Jones 2001), and *Paille in the Wind* (1992; Solomos 1996, p. 93).

Two Models of the Sieve

Xenakis offers two sieve models. The first model, the complex sieve, is described in his earliest writings (Xenakis 1965, 1966, 1967, 1988). The second model, the simple sieve, is found only in his last treatment of the topic and its accompanying software implementation (Xenakis 1990). Interestingly, this second model fails to incorporate aspects of the original, and Xenakis provides no explanation for this difference. The models differ in their allowed logic operators and their levels of residual class nesting.

Xenakis's first model was based on the manual calculation of sieves. Xenakis (1966; 1992, p. 234) provides an image of a hand-written diagram on graph paper labeled "Nomos alpha Sieves." Each column of the graph is treated as an integer unit and is clearly labeled "ELD = 1/4 tone." Numerous parallel, horizontal lines are used to calculate the sieve. Each line illustrates a residual class segment (with points marked as vertical tick-marks) and is labeled with a logic operation such as "∩ (5,2)." By applying the logic operator of each line to the previous line, the final sieve segment is realized. Xenakis mentions this technique elsewhere: when introducing the sieve in a later document, he describes setting the ELD to both a "semitone or a millimeter," and notating a sieve on graph paper (Xenakis 1990; 1992, p. 269). This tedious process has been duplicated by analysts of Xenakis's music (Vriend 1981, pp. 55–57; Harley 2004, p. 43).

Intersection, union, and unary complementation operators are employed in the first model (Xenakis 1965, 1966, 1967, 1988). In the second model (Xenakis 1990), however, both in prose and in code, there is no mention or example of complementation. Xenakis says nothing about this omission.

In the first model, logic operators are permitted in any combination, within any hierarchical level. The following complex sieve is an example:

$$\{-M@I \,\&\, M@I \,\&\, \{-M@I \mid M@I\}\} \mid M@I \,\&\, M@I$$

The second model exclusively uses two-level ordered groups. The inner level consists only of intersections, the outer level consists only of unions, and

complementation is not used. The following simple sieve is an example:

M@I & *M@I* & *M@I* | *M@I* | *M@I* & *M@I*

It can be suggested that Xenakis ordered groupings and reduced the use of logic operators because it became apparent that these features were not formally necessary to produce all segments. Further, software implementation may have encouraged the definition of a simpler, easily computable model. Two proofs, introduced with the second model (Xenakis 1990), support this claim.

Xenakis (1990; 1992, p. 275) provides an algorithm for sieve compression by segment: the derivation of a sieve from an arbitrary integer set. As mentioned earlier, this proof demonstrates that any finite set of integers can be generated by a maximally simple sieve: a sieve created exclusively from the union of residual classes. Thus, intersection, complementation, and multiple levels of nesting are not required to construct a sieve of any complexity and may be deemed superfluous. Union is the only necessary logic operator.

In Xenakis's second model, complementation is removed, yet intersection is retained. This peculiarity is explained by Xenakis's algorithm for compression by intersection. This proof demonstrates that any number of residual classes, upon intersection, can be reduced to a single residual class (1990; 1992, p. 271).

Intersection, in the second model, is used not out of necessity, but as a notational convenience. For example, the following simple sieve (Xenakis 1992, p. 274) 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 compression by intersection, this sieve is reduced to a maximally simple sieve. The last pair of residual classes, having no points in common, reduces to the null residual class 0@0:

24@23 | 30@3 | 104@70 | 0@0

The new model presented here combines both Xenakis's first and second models. Complex and simple sieves are incorporated into a single, bifaceted object. All forms of hierarchical nesting and all common logic operators, including symmetric difference (after Amiot et al. 1986), are permitted. This model reformulates Xenakis's reduction algorithms into the concept of compression, a method of moving from a complex to a maximally simple sieve. Though these features are not formally necessary to produce all sets, the expressive power of the sieve formula is expanded by their use. This notational convenience is, in fact, an essential feature of the model.

The Notation of the Sieve

The notation of a sieve is important. To a user, the logic formula is the primary interface, and its notation directly affects the utility of the model. A sieve notation must include a means of specifying a residual class as a modulus and a shift, symbols for the logic operators, and symbols to delimit residual class groups.

Xenakis uses one notation, with slight variation, for complex sieves (1965, 1966, 1967, 1988). Shifts are represented as subscripts of the modulus. (In some instances, this notation is reversed, with the modulus represented as a subscript of the shift; see Xenakis 1965.) Traditional logic symbols are used: $\cup$ for union, $\cap$ for intersection, and an over-score line for complementation. (Xenakis 1963 uses the same logic symbols for intersection and union; complementation, as a binary operator, is notated with a "–".) Transposition is represented by the variable n and is included with every subscript as "n+shift" (or "n" when shift is equal to zero). Groups are notated with parentheses. The following example (Xenakis 1992, p. 197) demonstrates this notation:

$$(\bar{3}_{n+2} \cap 4_{n}) \cup (\bar{3}_{n+1} \cap 4_{n+1}) \cup (3_{n+2} \cap 4_{n+2}) \cup (\bar{3}_{n} \cap 4_{n+3})$$

Xenakis uses two notations for simple sieves. The first is introduced in the two proofs presented in "Sieves" (Xenakis 1990). Residual classes are represented as number pairs delimited by parentheses and are given with either integers or variables (where *M* represents modulus, and *I* represents shift). Logic operators are notated as before. Transposition, as the variable *n* or otherwise, is no longer notated. Groups are notated with braces. The fol-

lowing example (Xenakis 1992, p. 274) 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 second notation for simple sieves, part of the C-language implementation accompanying Xenakis (1990), uses an American Standard Code for Information Interchange (ASCII) symbol representation. The use of ASCII ensures the availability of every character on a computer keyboard. Modulus and shift are represented as a number pair. Intersection is represented by the symbol "*," and union is represented with the symbol "+." Incidentally, Xenakis had used these symbols earlier in "Symbolic Music" (1963) and "La voie de la recherche et de la question" (1965). Complementation is not specified, and transposition is not notated. Groups are notated with square brackets, for example:

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

The new notation presented here is a "logic string." Similar to the notation presented in Xenakis (1990), ASCII characters are used to represent a sieve formula. To avoid using commas or parentheses, a residual class is given as a modulus and a shift separated by the "@" symbol. This symbol is chosen primarily for its infrequent use in other notations. (Representing a residual class with two figures separated by a symbol has a precedent in Xenakis 1966, where shift and modulus—in that order—are separated by the character "*M*".) Logic operators are notated using the pipe ("|") for union, the ampersand ("&") for intersection, the circumflex ("^") for symmetric difference, and the dash ("–") for unary complementation. All four logic operators are permitted, and complementation can be applied to a single residual class or a group. A group is notated with braces.

This notation has many advantages. It is more compact and requires fewer characters than Xenakis's notations. The use of bitwise logic operator symbols (pipe, ampersand, and circumflex) are transparent in meaning to those familiar with programming languages such as C, Java, or Python. The use of a single set of characters for grouping (braces) reduces visual complexity. Finally, the sieve contains only ASCII characters, no commas, and (optionally) no spaces, allowing for easy parsing and isolation when passed as an argument or included with complex data. This notation, for example, could easily be sent as an Open Sound Control (OSC) string; if stored in HyperText Markup Language (HTML) or eXtensible Markup Language (XML), one character used in this notation, the ampersand ("&"), must be converted to a character entity ("&").

Implementations of the Sieve

A software model of the sieve must treat the logic formula as the primary object, not one of the formula's possible segments. The logic formula should be retained as a reusable object such that it can be represented as conceived, multiple transpositions and segments can be extracted, and the formula itself can be logically combined. Simply combining fixed sets with logic operators reduces the functionality of a sieve to simply an input notation: the information contained in the design of the formula is lost only to one of its many output potentials. Of the handful of implementations available to date, none has treated the logic formula of the sieve as a reusable object, allowing the input, maintenance, representation, and processing of a complex sieve from a single argument.

Xenakis (1990) provides the first published software implementation of a sieve. This software consists of two main procedures: the generation of a sieve segment from a logic formula, and the generation of a logic formula from an arbitrary set of integers. Xenakis, in the 1992 edition of *Formalized Music,* credits the C code to Gérard Marino, stating that it is a port of Xenakis's original BASIC code. For the version published in *Perspectives of New Music,* no authorship is given. Functionally, the two versions are nearly identical. The version in *Formalized Music* (1992), however, contains numerous typographic omissions. Ronald Squibbs has listed these errors and provides corrected code (1996, pp. 292–303).

Unfortunately, the implementation Xenakis offered in 1990 cannot process the many complex

sieves he had demonstrated since 1965. This software, permitting only the calculation of simple sieves, has other shortcomings. The code lacks modular design, limiting both flexibility and expansion. The program is intended exclusively for user rather than programmatic interaction; a general purpose interface, necessary for use in larger systems, is not provided. A complete sieve cannot be supplied as a single argument: when entering a sieve, for example, the user must first declare the number of unions, then declare the number of residual classes in each intersection, then enter a modulus and shift one at a time. Further, there is no implementation of transposition and the user cannot freely designate *z*.

In 1980, prior to the publication of Xenakis's code, Sever Tipei implemented a sieve model in FORTRAN for use with his computer-assisted composition program MP1 (Tipei 1975). In addition to using sieves for parameter generation, Tipei produced weighted sieve-segments. If each residual class is assigned a weight, a sieve can be constructed that produces segments encoding combined weights. These techniques and others were used in his compositions and described in numerous articles (Tipei 1981, 1987, 1989).

Researchers at the Institut de Recherche et Coordination Acoustique/Musique (IRCAM) employed the sieve in a number of publications. Malherbe et al. (1985) demonstrate a novel application of sieve structures to model spectral peaks found in analyzed sound files. Amiot et al. (1986) develop upon this model and provide a more complete description. Here, sieves are employed primarily for filtering metric patterns. Logic operators are expanded to include symmetric difference, composition, and exponentiation. This sieve was implemented by Gérard Assayag as the "Langage de Cribles" (LC) in the now-obsolete Le Lisp language. Neither details of the implementation nor examples of its use are available. Assayag's Lisp-based ScoreBoard application (1993) may offer a related model.

In 1990, Marcel Mesnage programmed a text-based system in Macintosh Common Lisp called "Partitions d'Ensembles de Classes de Résidus" (PCR; Riotte 1992; Mesnage and Riotte 1993). This system, based on a model by André Riotte, implements a variety of sieve structures using union, intersection, and complementation, and produces sieve segments in integer, binary, and width formats. The system's sieve notation, however, significantly deviates from a logic-formula representation.

A more sophisticated sieve model, also written in Macintosh Common Lisp, was included in Mesnage's "Criblograph," a graphical user interface environment for music composition developed from 1989 until 1997, and then merged into Mesnage's "Mélusine" system. The sieve implementation in these systems offered all common logic operators, the creation and modification of reusable sieve segments, and a unique graphical sieve segment editor. Some of these tools have been subsequently incorporated into Mesnage's "Morphoscope" (1993). Despite the broad functionality of these systems, the sieve is not given a practical notation, and a complex sieve cannot be provided as a single argument.

Within Paul Berg's AC Toolbox (available online at www.koncon.nl/ACToolbox), a Lisp-based software system for algorithmic composition running on MacOS X are four sieve-related objects: `Sieve`, `Sieve-Union`, `Sieve-Filter`, and `Interpret-Sieve`. The `Sieve` object here can be better thought of as a residual class: the user provides a modulus, a shift (as start value) and a *z* (as minimum and maximum). Evaluation returns an integer segment. The object `Sieve-Union` is a union operator that combines any number of lists (generated by `Sieve` or otherwise) and returns a new list. `Sieve-Filter` employs a list (generated by `Sieve` or otherwise) to select values from another list. `Interpret-Sieve` is designed for the production of rhythms: a list of values, interpreted as either a binary or a width sieve segment, is used to process a list of rhythm durations into note (positive) or rest (negative) values. The user provides a list of durations, the sieve segment, and a control variable. Although providing useful tools for the deployment of combined residual class segments, this model does not allow the input or maintenance of a complex sieve as a complete logic formula.

Jones (2001), following the limits and notation of Xenakis (1990), models only simple sieves (called RCsets) as residual classes combined under union. Jones is primarily interested in deriving a maximally simple sieve from an arbitrary set for the pur-

pose of analysis. A new method of compression by segment is proposed. He rejects Xenakis's algorithm for only accepting residual classes with perfect correspondence to the source set. Because Xenakis often varied the realization of a sieve in its musical deployment, Jones proposes an algorithm that provides a "statistically optimal" (2001, p. 225) match to the source set, weighted toward fewer residual classes with lower moduli, even if this requires producing sieve segments that do not match the source. Candidate residual classes, for example, must match at least three points within the source set. Although this model suggests an interesting method of lossy sieve compression, the deviation Jones allows between source set and resulting sieve is so high as to suggest a transformation beyond mere compression. In his analysis of *Tetora,* Jones offers simple sieves to replace source sets that, in one case, cover as few as 10 out of 37 original points. Jones (2001) includes an implementation, programmed in BASIC, providing an interface similar to Xenakis (1990).

Some, perhaps owing to the absence of a complete software implementation, have calculated sieves with a numeric table (Squibbs 1996, pp. 304–306; Gibson 2001). If a sieve uses only two moduli, a table can be constructed with each modulus assigned to an axis. Each axis has rows or columns for every shift in the modulus (0 to *M*–1), thus representing every residual class of each modulus. Values in the table are filled by wrapping a *z* (from zero to one less than the product of the moduli) diagonally through the table. The resulting values, for any two residual classes, provide the intersection point found within a single-period sieve segment. Though Squibbs has demonstrated the use of multiple tables to handle sieves employing more than two moduli, this technique is very limited.

The predecessor of the sieve model presented here (Ariza 2004) introduced the logic string notation and modeled the logic formula as a reusable object. Sieve objects, however, were limited to simple sieves.

Object-Oriented Implementation

An object-oriented Python implementation has been developed based on the new model and notation presented above. This implementation is portable, modular, and offers the easy creation and deployment of sieve segments and transpositions. The model combines Xenakis's two models into a bi-faceted object: a sieve can contain both complex and simple representations simultaneously. The file `sieve.py`, part of the athenaCL library `libATH`, implements this model as the `Residual` and `Sieve` objects.

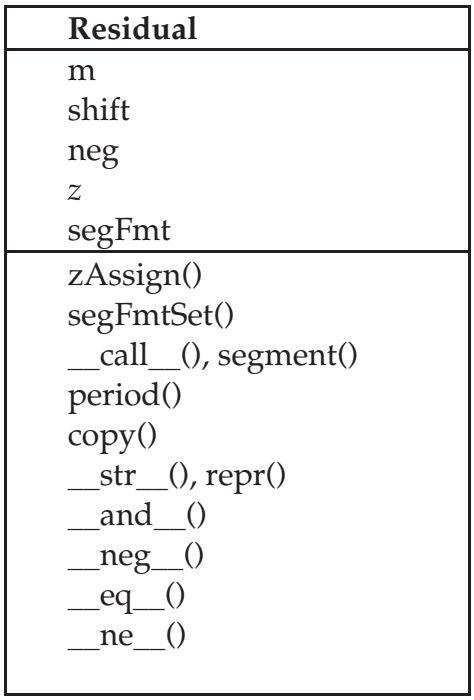
The Residual Object

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. As complementation of a single residual object is a unary operation, complementation is a Boolean value (`neg`) stored as an attribute of the `Residual object`. Each `Residual` instance contains a reference to a finite list of contiguous integers (`z`) from which sieve segments are filtered. The attribute `segFmt` determines segment output format, where format strings for integer, binary, unit, and width are notated `int`, `bin`, `unit`, and `wid`, respectively. The default `segFmt` is `int`. 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 `segment()` method of a `Residual` instance returns a transposed sieve segment in one of four formats. This method has three optional arguments: an integer value for transposition (`n`), by default zero; a list of integers (`z`), by default the `z` set at initialization; and a format string. Arguments passed to the segment method do not change attributes of the object: they are used only in the calculation of the desired segment. If changes to these attributes are desired, the method `zAssign()` can be used to set `z`, and the method `segFmtSet()` can be used to set `segFmt`. The method `period()` provides the period of the residual, which in all cases is equal to the modulus. The `repr()` method provides a string representations of the residual class, and the `copy()` method returns a new object with identical `m`, `shift`, `neg`, and `z` attributes.

Additional functionality is provided through op-

Figure 2. Residual object class diagram.



erator overloading. The object's `__call__()` method is mapped to the `segment()` method, and the `__str__()` method is mapped to the `repr()` method. (In Python, all objects can overload operators by defining specially named methods. The `__call__()` method is called when an object `x` is evaluated as `x()`. The `__str__()` method is called whenever a string representation of an object is needed, for example `print x`.)

The `__and__()` method, called with the binary `&` operator, applies compression by intersection to two `Residual` operands and returns the `Residual` of the reduced intersection. The `z` of this new object is formed from the union of each operand's `z`. Compression by intersection is implemented after the algorithm presented in Xenakis (1990). If intersection of a complemented `residual` class is attempted, an error is raised. The `__or__()` method, evoked with the | operator, is not implemented: the union of two residual classes can only be represented by two residual classes. The `__xor__()` method, evoked with the ^ operator, is likewise not implemented.

The `__neg__()` method, called with the unary – operator, changes the `neg` attribute to its Boolean opposite. To compare two `Residual` objects, the equal and not-equal operator methods, `__eq__()` and `__ne__()`, are defined by comparing `m`, `shift`, and `neg` attributes.

A Unified Modeling Language (UML) class diagram of the `Residual`, summarizing the public attributes and methods of this object, is provided in Figure 2. Figure 3 demonstrates the `Residual` object within a Python interactive session. (The Python prompt >>> precedes user input. Comments are preceded by #.)

The Sieve Object

The `Sieve` object is a bi-faceted representation of the Xenakis sieve. One representation is the expanded state, which can be any sieve from simple to complex. The other representation is the same sieve compressed; the compressed state is always a maximally simple sieve. Each state is an independent sieve.

The logic string provided at initialization becomes the expanded sieve. The type of this sieve (complex or simple) is stored as the `expType` attribute and is used to determine the form of compression. Each state (expanded and compressed) has unique attributes for logic string, residual classes, and period. Each state generates independent segments. The `z` attribute is shared between both states. The object has an attribute for current state, set at initialization to `expanded`. To change states, the `compress()` or `expand()` methods are called.

The `Sieve` shares aspects of the `Residual` object interface. The `zAssign()` and `segFmtSet()` methods set `z` and `segment` formats, respectively. The `segment()` method, with the addition of an argument for state, returns a sieve segment for an optional shift, `z`, and format argument. This method, using the current state, is mapped to the `__call__()` method. Segment formats are the same as for `Residual` objects: `integer`, `binary`, `unit`, and `width`. The `period()` method returns the period of the current state, or the state designated with an optional argument. The period of each state is calculated by finding the lowest common multiple of all residual periods. The `repr()` method, also mapped to the `__str__()` method, returns a string representation

Figure 3. Python session demonstrating the Residual object.

```
>>> from athenaCL.libATH import sieve
>>> # the built-in Python function "range" can be used to create a z
>>> z = range(0,25) # creates a list of contiguous integers from 0 to 24
>>> a = sieve.Residual(3,2,0,z) # a new Residual object
>>> print a # return a string representation
3@2
>>> a() # calling the Residual returns a sieve segment
[2, 5, 8, 11, 14, 17, 20, 23]
>>> a(2) # the first argument is a transposition
[1, 4, 7, 10, 13, 16, 19, 22]
>>> a(2, range(-20,-10)) # the second argument is z
[-20, -17, -14, -11]
>>> a(0, range(0,13), 'bin') # the third argument is segment format
[0, 0, 1, 0, 0, 1, 0, 0, 1, 0, 0, 1, 0]
>>> a(0, range(0,13), 'unit') # a unit segment
[0.1666666666666666, 0.41666666666666669, 0.66666666666666663,
0.91666666666666663]
>>> a(0, range(0,13), 'wid') # a width segment
[3, 3, 3]
>>> a.period() # period, modulus, and width are equal
3
>>> b = sieve.Residual(8,0,1) # a complemented Residual, with a default z
>>> print b
-8@0
>>> b = -b # unary complementation
>>> print b
8@0
>>> b() # calculating a segment with default transposition, z, and format
[0, 8, 16, 24, 32, 40, 48, 56, 64, 72, 80, 88, 96]
>>> c = a & b # the intersection of two Residuals
>>> print c
24@8
>>> c()
[8, 32, 56, 80]
>>> c == a # Residuals tested for equality
0
>>> -a != a # Residuals tested for inequality
1
```

of the logic formula; the `copy()` method returns a new `Sieve` instance with identical attributes.

Many steps are performed at initialization. The `Sieve` object is instantiated with either a logic string or a list of integers and an optional argument for `z`. The sieve given at initialization is set as the expanded sieve and is parsed.

Parsing consists of translating the formula of the sieve into a tree string and a collection of Residual objects stored in a dictionary (`resLib`). A tree string is the logic formula of the sieve with residual class notations replaced by unique string keys. A tree string is created for each state (`expTree` and `cmpTree`). Each residual class notated in the formula is identified and replaced by a key, in the form of "<R*n*>," where *n* is an integer index. For each residual class declared in the logic string, a `Residual` object is instantiated in the `resLib` dictionary.

A `Sieve` produces a segment by combining residual class segments with the specified logic operators. The combination of segments is facilitated by Python's built-in `Set` object, distributed with Python 2.3 and found in the module `sets.py`. The `Set` object offers standard set procedures with operator overloading of the same symbols used in the logic string notation; nested structures, furthermore, are evaluated with the desired precedence.

If a segment is requested from the `Sieve`, each key

in the appropriate tree string is replaced by the string necessary to instantiate a Set; this Set contains the integer segment from the corresponding Residual object. If a residual class is complemented, the segment returned already reflects this complementation. After all keys are replaced, the entire string is evaluated, causing the instantiation and evaluation of all Set objects. Evaluation processes Set objects with the logic operators specified in the tree string. A single Set results and is returned by the Sieve as a list. If a string representation is requested from the Sieve, a similar operation is performed: all keys in the tree string are replaced with the string representation of the corresponding Residual object.

Sieve objects forbid binary complementation, but allow unary complementation of Residuals and groups. Python Set objects, however, employ only binary complementation. This difference is handled in two ways: (1) A single Residual class, under complementation, internalizes its complemented state. A Set is then instantiated from an already-complemented Residual segment; the resulting Set thus does not require complementation. (2) In the case that a group of Residual objects is complemented outside of a delimiter, the complementation operator, at evaluation, is preceded by a Set object corresponding to a segment of 1@1 for the current z. Because binary complementation is evaluated before intersection, symmetric difference, and union, this effectively converts unary negation into binary negation at the time of Set evaluation.

During initialization, and after the expanded sieve is parsed, compression is performed. Two methods of compression are available: by intersection and by segment. The expanded sieve type (expType) determines which compression is performed. If a sieve is complex, compression by segment must be performed. If a sieve is simple, compression by intersection is performed.

If compression by intersection is mandated by expType, the expTree is divided into intersection groups. The keys for each group of Residual objects are collected, and the corresponding objects are intersected. A new tree string (cmpTree) is constructed, joining by union keys for each of the resulting Residual objects. Each Residual is stored in resLib.

If compression by segment is mandated by expType, a segment is created at the current z and is processed. Compression by segment cannot be performed if, owing to the logic formula or the size of z, an empty segment is retuned. Compression returns a list of Residual objects that, when combined under union, will create a sieve that returns the source segment for the provided z. The Residual objects are stored in resLib, and a new tree string (cmpTree) is created.

A variation of Xenakis's algorithm (1990; 1992, p. 274) for compression by segment is implemented as follows: (1) Two copies of the source integer set are stored as src and match lists. (2) A z list, if not provided, is constructed by taking the range of integers from the minimum to the maximum of the match list. (3) A value in the match list is treated as a shift value. (4) A Residual object is created with this shift, a modulus of 1, and z. (5) A sieve segment is created by calling the Residual instance. (6a) If this sieve segment is a subset of src, the object is appended to the list residuals and the shift value, as well as any points on the segment also found in the match list, are removed from match. (6b) Otherwise, the shift is retained, the modulus is incremented, and the process is repeated until a subset sieve segment is found. (7) Remaining values in match are each treated as a shift (repeating from step 3), until match is empty. (Note that, because the segment is always compared to src and not match, found Residual objects may cover redundant points; this addresses a shortcoming of Xenakis's algorithm mentioned in Jones 2001, p. 233.)

If an integer in a source set is treated as a residual class shift, a modulus can always be found that, with this shift, produces a segment that is both a subset of the source and matches at least the shift. This segment will always be found before the modulus is incremented past the number of points in z. At an extreme, a residual class can be created for each point in the source set, each residual, for the current z, contributing only one point to a union. It follows that any arbitrary set can be represented as a maximally simple sieve.

Compression occurs at initialization with the provided or default z. After initialization, compression can be re-performed if a new z is provided. This new

z will be set as the current z and will be used to create the segment sampled for compression. Changing the z value, in the case of compression by segment, can result in different compressed representations.

If, rather than a logic string, a list of integers is entered at initialization, compression by segment is used to create the necessary Residual objects; these objects are stored in resLib, and a tree string (expTree) is created. The expanded sieve, in this case, will always be a maximally simple sieve: further compression is not possible. The compressed sieve representation of the object will be identical to the expanded sieve.

Sieve objects themselves, through operator overloading, can be combined with logic operators to produce new Sieve objects. The methods __and__(), __or__(), __xor__(), and __neg__() are defined to overload operators &, |, ^, and –. A new sieve is created by instantiating an object with a new logic string and the union of each operand's z. This new logic string encodes the operation on each operand's expanded sieve, producing a new object that models the operation.

A UML class diagram, summarizing the public attributes and methods of this object, is provided in Figure 4.

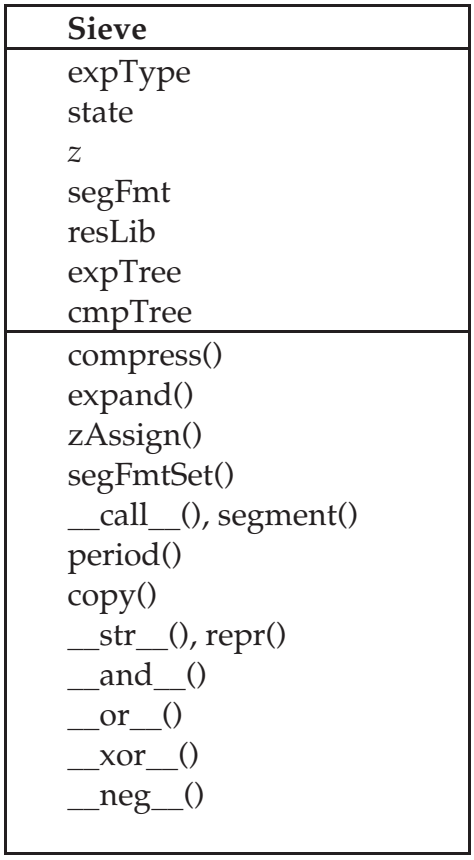


Figure 4. Sieve object class diagram.

The Python session provided in Figure 5 demonstrates, for a complex sieve, the internal representation of tree strings for both the expanded and compressed states. Combining Sieve objects with logic operators is also demonstrated. Practical examples of the Sieve object are provided next.

Demonstrating the Sieve

The Sieve object can realize all of Xenakis's sieves, both complex and simple. One of Xenakis's earliest examples is a complex sieve for the generation of a major scale. This elegant formula is, however, incompatible with both his second model and its software implementation (Xenakis 1990). This sieve provides an excellent example of the utility of a bi-faceted representation of the sieve.

A Sieve object can be created from Xenakis's formula for the major scale. This expanded sieve has a period of 12, and it produces a segment corresponding to the major scale for all z. Using the conversion function psToNoteName() from the athenaCL libATH module pitchTools, each integer can be converted to a pitch name to verify the accuracy of the scale over four octaves. (The Python map function applies a function to each value in a list and returns a new list.) The compressed sieve, after compression by segment, is a maximally simple sieve. It also has a period of 12 and produces the major scale over four octaves.

Figure 6 hides a potential confusion. Because a complex sieve is supplied at initialization, compression must occur by segment. As no z is supplied, a default z of 0 to 99 is provided. For some smaller z ranges, however, compression by segment will result in a different compressed sieve.

If a Sieve object is created from an integer set of a one-octave major scale (where z is automatically determined by the minimum and maximum of the

Figure 5. Python session demonstrating internal representation of a logic formula and the combination of Sieve objects with logic operators.

```
>>> from athenaCL.libATH import sieve
>>> a = sieve.Sieve('7@0|{-5@2&-4@3}') # create a complex sieve
>>> a.expType # the expanded type is automatically identified
'complex'
>>> a.expTree # the expanded tree string
'<R0>|{<R1>&<R2>}'
>>> a.period() # the period of the expanded state
140
>>> a.compress() # changing the current state of the Sieve
>>> print a # a maximally simple sieve
7@0|10@0|10@4|10@6|10@8|16@13|20@1|20@5|20@9|20@13
>>> a.cmpTree # the corresponding compressed tree string
'<R3>|<R4>|<R5>|<R6>|<R7>|<R8>|<R9>|<R10>|<R11>|<R12>'
>>> a.period() # the period of the compressed sieve
560
>>> a(0, range(0,20), 'bin') # return a binary segment
[1, 1, 0, 0, 1, 1, 1, 1, 1, 1, 1, 0, 0, 1, 1, 0, 1, 0, 1, 0]
>>> b = -a # Sieve object complementation returns a new object
>>> print b
-{7@0|{-5@2&-4@3}}
>>> b(0, range(0,20), 'bin')
[0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 1, 0, 1, 0, 1]
>>> c = sieve.Sieve('9@7')
>>> print c
9@7
>>> d = c | b # Sieve object union returns a new object
>>> print d
{-{7@0|{-5@2&-4@3}}}|{9@7}
>>> d(0, range(0,20), 'bin') # return a binary segment
[0, 0, 1, 1, 0, 0, 0, 1, 0, 0, 0, 1, 1, 0, 0, 1, 1, 1, 0, 1]
>>> d(0, range(0,20), 'wid') # return a width segment
[1, 4, 4, 1, 3, 1, 1, 2]
>>> e = sieve.Sieve('5@2|5@3')
>>> f = d & e # Sieve object intersection returns a new object
>>> print f
{5@2|5@3}&{{-{7@0|{-5@2&-4@3}}}|{9@7}}
>>> f(0, range(0,20), 'bin') # the expected binary segment
[0, 0, 1, 1, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0]
```

segment), a logic formula is found. This formula, however, is only valid for the points covered by the `z` of one octave. Beyond this, pitches deviate from the major scale. Figure 7 creates four `Sieve` objects. Sieve "a" is initiated with a one octave set of the major scale; the resulting sieve is shown to have a period of 210 and deviate from the major scale at values beyond the `z` set at initialization. Sieve "b" extends the source set to two octaves. A different sieve is found with the same period. Again, `z` values beyond the `z` set at initialization result in a scale that deviates from the desired major scale. This process is repeated with a three-octave set. Given a four-octave set, the desired maximally simple sieve is finally found, having the necessary period of 12 and producing correct segments for all `z`. Xenakis, aware of this aspect of compression by segment, states that "one should take into account as many points as possible in order to secure a more precise logical expression" (1990, p. 275). Gibson (2001), likewise using segments of the major scale, has also demonstrated this constraint.

Jan Vriend, in his examination of *Nomos Alpha,* demonstrates the complex system Xenakis employed for creating a succession of sieves, all based on a common formulation (1981, p. 55). The sieve is provided as an algebraic logic formula (Xenakis 1992, p. 230):

Figure 6. Python session demonstrating Xenakis's sieve for the major scale and its compressed form.

```
>>> from athenaCL.libATH import sieve, pitchTools
>>> # Xenakis's logical formula for the major scale
>>> a = sieve.Sieve('{-3@2&4}|{-3@1&4@1}|{3@2&4@2}|{-3@0&4@3}')
>>> print a
{-3@2&4@0}|{-3@1&4@1}|{3@2&4@2}|{-3@0&4@3}
>>> a.period() # the period of the major scale is 12
12
>>> a(0, range(0,13)) # one octave segment as pitch class
[0, 2, 4, 5, 7, 9, 11, 12]
>>> # four octave segment as note names
>>> map(pitchTools.psToNoteName, a(0, range(0,49)))
['C4', 'D4', 'E4', 'F4', 'G4', 'A4', 'B4', 'C5', 'D5', 'E5', 'F5', 'G5', 'A5',
'B5', 'C6', 'D6', 'E6', 'F6', 'G6', 'A6', 'B6', 'C7', 'D7', 'E7', 'F7', 'G7',
'A7', 'B7', 'C8']
>>> a.compress() # toggle the current sieve state
>>> print a # display the compressed sieve
6@5|12@0|12@2|12@4|12@7|12@9
>>> a.period() # return the compressed period
12
>>> # four octave segment from the compressed sieve
>>> map(pitchTools.psToNoteName, a(0, range(0,49)))
['C4', 'D4', 'E4', 'F4', 'G4', 'A4', 'B4', 'C5', 'D5', 'E5', 'F5', 'G5', 'A5',
'B5', 'C6', 'D6', 'E6', 'F6', 'G6', 'A6', 'B6', 'C7', 'D7', 'E7', 'F7', 'G7',
'A7', 'B7', 'C8']
```

$L\,(M,N) = \{–\{N@i \mid N@j \mid N@k \mid N@l\}\ \&\ M@p\} \mid \{–\{M@q \mid M@r\}\ \&\ N@s\} \mid \{N@t \mid N@u \mid N@v\}$

By algorithmically generating variables for modulus and shift, this structure is used to produce six unique sieves and their resulting segments. A sieve, using this formulation and employed in *Nomos Alpha,* is given below (Xenakis 1966; 1992, p. 230):

$\{–\{13@3 \mid 13@5 \mid 13@7 \mid 13@9\}\ \&\ 11@2\} \mid \{–\{11@4 \mid 11@8\}\ \&\ 13@9\} \mid \{13@0 \mid 13@1 \mid 13@6\}$

Vriend (1981, p. 57), after demonstrating the manual calculation of a segment from this sieve, provides the resulting sieve segment mapped onto a quarter-tone ELD pitch scale. This scale is notated in Figure 8. Figure 9, using the `Sieve` object to model Xenakis's sieve, confirms Vriend's realization of the segment.

Squibbs (1996, p. 61) provides the logic formula for a sieve from Xenakis's *À R. (Hommage à Maurice Ravel):*

$\{8@0\ \&\ \{11@0 \mid 11@4 \mid 11@5 \mid 11@6 \mid 11@10\}\} \mid$

$\{8@1\ \&\ \{11@2 \mid 11@3 \mid 11@6 \mid 11@7 \mid 11@9\}\} \mid$

$\{8@2\ \&\ \{11@0 \mid 11@1 \mid 11@2 \mid 11@3 \mid 11@5 \mid 11@10\}\} \mid$

$\{8@3\ \&\ \{11@1 \mid 11@2 \mid 11@3 \mid 11@4 \mid 11@10\}\} \mid$

$\{8@4\ \&\ \{11@0 \mid 11@4 \mid 11@8\}\} \mid$

$\{8@5\ \&\ \{11@0 \mid 11@2 \mid 11@3 \mid 11@7 \mid 11@9 \mid 11@10\}\} \mid$

$\{8@6\ \&\ \{11@1 \mid 11@3 \mid 11@5 \mid 11@7 \mid 11@8 \mid 11@9\}\} \mid$

$\{8@7\ \&\ \{11@1 \mid 11@3 \mid 11@6 \mid 11@7 \mid 11@8 \mid 11@10\}\}$

Two segments from this sieve are given, at transpositions 0 and 10. These are provided in Squibbs's notation (1996, p. 61–62):

$K = \{0, 2, 3, 4, 7, 9, 10, 13, 14, 16, 17, 21, 23, 25, 29, 30, 32, 34, 35, 38, 39, 43, 44, 47, 48, 52, 53, 57, 58, 59, 62, 63, 66, 67, 69, 72, 73, 77, 78, 82, 86, 87\}$

$T_{10}(K) (\text{mod } 88) = \{0, 4, 8, 9, 10, 12, 13, 14, 17, 19, 20, 23, 24, 26, 27, 31, 33, 35, 39, 40, 42, 44, 45, 48, 49, 53, 54, 57, 58, 62, 63, 67, 68, 69, 72, 73, 76, 77, 79, 82, 83, 87\}$

Figure 9 demonstrates that the creation of a Sieve object with this logic formula accurately produces the desired segments.

Ellen Rennie Flint, in her analysis of *Psappha,* demonstrates Xenakis's use of sieves to generate rhythms. Flint (1989, p. 228) provides the logic for-

Figure 7. Python session demonstrating compression by segment of a major scale.

```
>>> from athenaCL.libATH import sieve, pitchTools
>>> # one octave of the major scale, entered as a list
>>> a = sieve.Sieve([0,2,4,5,7,9,11])
>>> print a
5@2|5@4|6@5|7@0
>>> # pitches deviate from the major scale, excluding E5
>>> map(pitchTools.psToNoteName, a(0, range(0,25)))
['C4', 'D4', 'E4', 'F4', 'G4', 'A4', 'B4', 'C5', 'D5', 'F5', 'G5', 'A5', 'A#5',
'B5', 'C6']
>>> a.period() # expected period of 12 not found
210
>>> # two octaves of the major scale, entered as a list
>>> b = sieve.Sieve([0,2,4,5,7,9,11,12,14,16,17,19,21,23])
>>> print b
5@4|6@5|7@0|7@2|7@5
>>> # pitches deviate from the major scale at F#6
>>> map(pitchTools.psToNoteName, b(0, range(0,37)))
['C4', 'D4', 'E4', 'F4', 'G4', 'A4', 'B4', 'C5', 'D5', 'E5', 'F5', 'G5', 'A5',
'B5', 'C6', 'D6', 'E6', 'F6', 'F#6', 'A6', 'A#6', 'B6']
>>> b.period() # expected period of 12 not found
210
>>> # three octaves of the major scale, entered as a list
>>> c = sieve.Sieve([0,2,4,5,7,9,11,12,14,16,17,19,21,23,24,26,28,29,31,33,35])
>>> print c
6@5|7@0|7@5|10@9|12@0|12@2|12@4|12@7
>>> # pitches deviate from the major scale at D#7
>>> map(pitchTools.psToNoteName, c(0, range(0,48)))
['C4', 'D4', 'E4', 'F4', 'G4', 'A4', 'B4', 'C5', 'D5', 'E5', 'F5', 'G5', 'A5',
'B5', 'C6', 'D6', 'E6', 'F6', 'G6', 'A6', 'B6', 'C7', 'D7', 'D#7', 'E7', 'F7',
'F#7', 'G7', 'B7']
>>> c.period() # expected period of 12 not found
420
>>> # four octaves of the major scale, entered as a list
>>> d =
sieve.Sieve([0,2,4,5,7,9,11,12,14,16,17,19,21,23,24,26,28,29,31,33,35,36,38,40,
41,43,45,47])
>>> print d
6@5|12@0|12@2|12@4|12@7|12@9
>>> d.period() # expected period of 12 found
12
```

mula of the sieve used to create the rhythmic articulations of the first 39 time units of the composition:

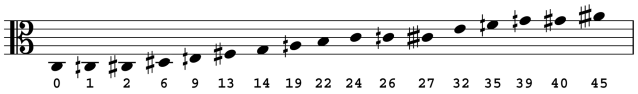
{{8@0 | 8@1 | 8@7} & {5@1 | 5@3}} | {{8@0 | 8@1 | 8@2} & 5@0} |

{8@3 & {5@0 | 5@1 | 5@2 | 5@3 | 5@4}} | {8@4 & {5@0 | 5@1 | 5@2 | 5@3 | 5@4}} |

{{8@5 | 8@6} & {5@2 | 5@3 | 5@4}} | {8@1&5@2} | {8@6&5@1}

The resulting sieve segment, as attack points, can be transcribed from the score; the second voice of group B provides a clear statement of this sieve (Harley 2004, p. 96). These points, notated and expressed as an integer segment, are provided in Figure 10.

Figure 8. Sieve generated scale from Nomos Alpha, *with ELD equal to a quarter tone (Vriend 1981).*



Using the `Sieve` object, this sieve can be recreated from its logic formula. In Figure 9 integer and binary segments, over a `z` from 0 to 38, are created.

Figure 9. Python session demonstrating Xenakis's sieves from Nomos Alpha, À R. (Hommage à Maurice Ravel) *and* Psappha.

```
>>> from athenaCL.libATH import sieve
>>> # sieve from Xenakis's Nomos Alpha
>>> x = sieve.Sieve('{-{13@3|13@5|13@7|13@9}&11@2}|{-
{11@4|11@8}&13@9}|{13@0|13@1|13@6}')
>>> print x
{-{13@3|13@5|13@7|13@9}&11@2}|{-{11@4|11@8}&13@9}|{13@0|13@1|13@6}
>>> x() # segment notated in Vriend (1981)
[0, 1, 2, 6, 9, 13, 14, 19, 22, 24, 26, 27, 32, 35, 39, 40, 45, 52, 53, 58, 61,
65, 66, 71, 78, 79, 84, 87, 90, 91, 92, 97]

>>> # sieve from Xenakis's À R. (Hommage à Maurice Ravel)
>>> a =
sieve.Sieve('{8@0&{11@0|11@4|11@5|11@6|11@10}}|{8@1&{11@2|11@3|11@6|11@7|11@9}}
|{8@2&{11@0|11@1|11@2|11@3|11@5|11@10}}|{8@3&{11@1|11@2|11@3|11@4|11@10}}|{8@4&
{11@0|11@4|11@8}}|{8@5&{11@0|11@2|11@3|11@7|11@9|11@10}}|{8@6&{11@1|11@3|11@5|1
1@7|11@8|11@9}}|{8@7&{11@1|11@3|11@6|11@7|11@8|11@10}}')
>>> a.period() # as modulus values of only 8 and 11 are used, the period is 88
88
>>> a(0, range(0,88)) # segments that correspond with Squibbs (1996)
[0, 2, 3, 4, 7, 9, 10, 13, 14, 16, 17, 21, 23, 25, 29, 30, 32, 34, 35, 38, 39,
43, 44, 47, 48, 52, 53, 57, 58, 59, 62, 63, 66, 67, 69, 72, 73, 77, 78, 82, 86,
87]
>>> a(10, range(0,88)) # transposition at 10
[0, 4, 8, 9, 10, 12, 13, 14, 17, 19, 20, 23, 24, 26, 27, 31, 33, 35, 39, 40,
42, 44, 45, 48, 49, 53, 54, 57, 58, 62, 63, 67, 68, 69, 72, 73, 76, 77, 79, 82,
83, 87]

>>> # sieve from Xenakis's Psappha (Flint 1989)
>>> x =
sieve.Sieve('{{8@0|8@1|8@7}&{5@1|5@3}}|{{8@0|8@1|8@2}&5@0}|{8@3&{5@0|5@1|5@2|5@
3|5@4}}|{8@4&{5@0|5@1|5@2|5@3|5@4}}|{{8@5|8@6}&{5@2|5@3|5@4}}|{8@1&5@2}|{8@6&5@
1}')
>>> segment corresponds to opening rhythm in group B, voice 2
>>> x(0, range(0,39))
[0, 1, 3, 4, 6, 8, 10, 11, 12, 13, 14, 16, 17, 19, 20, 22, 23, 25, 27, 28, 29,
31, 33, 35, 36, 37, 38]
>>> x.segFmtSet('bin') # change the segment format
>>> x(0, range(0,39))
[1, 1, 0, 1, 1, 0, 1, 0, 1, 0, 1, 1, 1, 1, 1, 0, 1, 1, 0, 1, 1, 0, 1, 1, 0, 1,
0, 1, 1, 1, 0, 1, 0, 1, 0, 1, 1, 1, 1]
```

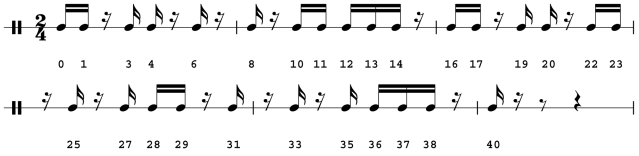
The binary segment provides a clear representation of a sieve as a sequence of points (notes) and slots (rests).

Xenakis discussed many methods of transforming sieve structures. One method, demonstrated above with the sieve system from *Nomos Alpha,* includes the algorithmic generation of sieve moduli and shifts; other methods employ permutations of shift values or logic operators, transpositions, or ELD transformations. Xenakis called these operations *métabolae.* Although space does not permit demonstration here, métabolae of any complexity can be programmed in Python by using dynamically generated `Sieve` objects.

Integration in athenaCL

The objects provided in `sieve.py` can be used in isolation or in cooperation with other software. Within athenaCL (Ariza 2002, 2003, 2004), high-level, practical object interfaces have been developed to provide access to sieve functionality. These interfaces provide sieve-based tools for the algorithmic com-

Figure 10. Psappha*, opening passage, group B, voice 2. Complex sieve with a period of 40.*



position of pitches, rhythms, and general parameter values. Complete demonstration is beyond the scope of this article; basic functionality of a selection of these tools, however, will be summarized.

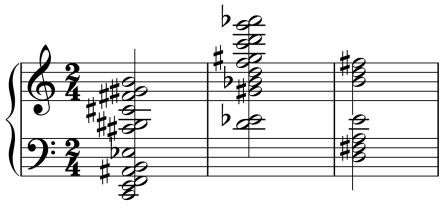
Sieve Pitch Generation

The athenaCL system features specialized objects for pitch structures, including `Pitch`, `Multiset` (collections of `Pitch` objects), and `Path` (ordered `Multiset` objects). When using athenaCL for composition, a `Path` defines reusable pitch collections. The actual use of a `Path` is dependent on interpretation by a `Texture`, to be described below. The `Multiset` objects of a `Path` can be provided by the user in many representations, including pitch-class (e.g., 0, 5, 6), pitch names (e.g., C2, F6, F#4), and Forte (1973) set-class names (e.g., 3–5B). The `Sieve` object enables users to enter a `Multiset` as a sieve.

When using a sieve in athenaCL to create a `Path`, the sieve segment is constructed upon a chromatic pitch range. When entering a sieve, the logic string can be followed by two comma-delimited arguments for lower- and upper-bounding pitches. These pitches are notated with pitch names, where middle C is denoted by C4. Rather than defining `z` with integers, `z` is derived from the range given by the bounding pitches. An optional third argument is the pitch of origin, or the location where the zero of the sieve is placed within the pitch scale. This value, if treated as an integer, is the same as transposition. If no argument is provided, the origin is automatically set to the lower-bounding pitch.

Commands in athenaCL can be executed with single command-line arguments or through a series of interactive, text-based prompts. The following command-line argument, using the `PathInstance New` command (`PIn`), creates a new `Path` named "a" from three simple sieves:

Figure 11. Three sieve-generated simultaneities: (1) 5@1 | 7@1,C2,C5,C4; (2) 6@2 | 7@3,C4,C7; and (3) 10@4 | 7@4,C3,C6,C4.



```
:: PIn a 5@1|7@1,C2,C5,C4 6@2|7@3,C4,C7 10@4|7@4,C3,C6,C4
```

Each sieve has a cycle of fifths (a residual class with modulus 7), and all share a common origin (C4).

This `Path`, notated in Figure 11 as a succession of un-transposed, sustained simultaneities, demonstrates the ease with which sieves can be used to create diverse pitch structures. A `Texture` in athenaCL can employ these pitch structures in a variety of fashions: as a scale from which melodies are created, as a collection from which subset chords are derived, or as a pointillistic cloud of pitches.

Sieve Rhythm Generation

A `Texture` in athenaCL is a model of an algorithmic music layer employing two levels of algorithmic design. The lower level of algorithmic design is controlled by `ParameterObject` objects, objects that provide values for every parameter of an event. The number of parameters in a `Texture`, as well as their function and interaction, is determined by the parent type and instrument model of the `Texture`. The upper level of algorithmic design is controlled by the `TextureModule`, the parent class of each `Texture` instance. The `TextureModule` manages the deployment and interaction of lower-level `ParameterObject` objects, as well as offering linear or non-linear event processing. `TextureModule` objects can generate either monophonic or polyphonic musical parts, and they are capable of algorithmic procedures not possible by parameter generation alone, such as algorithmically generated ornamentation (Ariza 2003).

A `ParameterObject` is entered by the user as a list of arguments. Arguments can be strings, numbers, lists, or arguments for nested `ParameterObject` objects. The first argument is always the name of the `ParameterObject`.

Rhythms are generated using specialized ParameterObject objects. Pulse objects, the rhythm primitive in athenaCL, are measures of duration relative to a beat. These objects are notated with three values: a divisor, a multiplier, and an accent. The divisor designates a fraction of the beat; the multiplier scales this fractional division, and the accent can code for notes (value of 1) or rests (a value of 0). Thus, at a quarter-note beat equal to 120 BPM, a Pulse object of (2,1,1) would be an eighth-note duration equal to 0.25 seconds. A Pulse object of (4,3,1) would produce a dotted eighth-note duration equal to 0.375 seconds.

Two sieve-based rhythm ParameterObject objects, pulseSieve and rhythmSieve, are deployed within athenaCL. Xenakis (1990; 1992, p. 269) demonstrates converting a binary sieve segment to a rhythm by setting the ELD to a duration, segment points to notes, and segment slots to rests. The athenaCL pulseSieve object extends this application. The arguments for pulseSieve are as follows: name; sieve logic string; length; pulse; a selection string randomChoice, randomWalk, randomPermutate, orderedCyclic, or orderedOscillate; and an articulation string attack or sustain. The length value creates a z from 0 to length–1; pulse defines the durational ELD as a Pulse object; the selection string determines the method of reading values from the sieve segment; and articulation selects whether the rhythm is created from a binary segment (an attack articulation, in which intervening slots become rests) or from a width segment (a sustain articulation, in which intervening slots are sustained).

The pulseSieve object can be thought of as filtering a z of equal-valued pulses. For more rhythmic variety, the ParameterObject called rhythmSieve allows z to be supplied by any non-rest pulse list. This pulse list is then filtered by the sieve: points on the sieve segment remain notes, and slots on the sieve segment become rests. The pulse list, or z, can be dynamically generated by any other rhythm ParameterObject. A ParameterObject for generating rhythmic variants with genetic algorithms (Ariza 2002), for instance, could be used to generate the z filtered by the sieve. The arguments for the rhythmSieve ParameterObject are as follows: name, sieve logic string, length, selection string, and rhythm ParameterObject. The rhythm ParameterObject is given as a list of arguments for the desired rhythm generator.

All sieves, when interpreted as rhythms, can be thought of as composite polyrhythms. As each Residual object produces a periodic stream of pulses, so the sieve is a periodic stream of composite pulses. The sieve, used as such, provides a compact and precise notation for composite polyrhythms. In athenaCL, for example, numerous independent Texture objects, each with related sieves, could be used to create dense polyrhythmic structures.

Sieve Parameter Generation

With the exclusion of rhythm, all event values in athenaCL can be controlled by generator ParameterObject objects. Event values include tempo, amplitude, panning, microtonal pitch transposition, and, depending on instrument model, values for synthesis parameters. Xenakis describes the utility of controlling many attributes of a note event with a sieve: "[S]ieve theory is very general and consequently is applicable to any other sound characteristic that may be provided with a totally ordered structure, such as intensity, instants, density, degrees of order, speed, etc." (1966; 1992, p. 199).

The athenaCL ParameterObject called valueSieve facilitates flexible generation of masked, floating point values applicable to a wide range of parameters. The arguments for valueSieve are as follows: name, sieve logic string, length, minimum, maximum, and selection string. Sieve logic strings, length values, and selection strings function as in the pulseSieve and rhythmSieve objects. However, where pulseSieve and rhythmSieve use binary or width sieve segments, valueSieve uses a unit segment. Segment points, as floating-point values within the unit interval, are read from the unit segment (as dictated by the selection string) and then scaled within dynamic minimum and maximum values. Varying the maximum and minimum with nested ParameterObject objects provides dy-

namic scaling of the sieve segment's proportions. The sieve is thus bound by what G. M. Koenig called a tendency mask (1970). Although with such a mask absolute values of the sieve are not maintained, the proportional distribution of values between points remains intact. Within athenaCL, such a generator could be used to control tempo values, panning positions, amplitudes, or microtonal transpositions; in combination with a Csound instrument, sieve-derived values could be used for selecting filter cutoff frequencies or start times of an audio file.

Conclusion

With the sieve, Xenakis sought to "add to our arsenal sharper tools"—tools of "trenchant axiomatics and formalization" (1967; 1992, p. 194). The sieve model presented here shares the elegance and power of Xenakis's original theory while providing an open-source, portable, modular, and complete implementation. With this new model, rigorous exploration of the sieve is possible, including applications in algorithmic composition, algorithmic synthesis, music analysis, and music theory. In 1996, Xenakis stated "the basic problem for the originator of computer music is how to distribute points on a line" (1996, p. 150); the sieve is one solution. The Python module sieve.py is distributed under the General Public License (GPL) as part of athenaCL, and it can be downloaded online at www.athenacl.org.

Acknowledgments

I particularly thank Elizabeth Hoffman, who, over the course of this project, provided advice and guidance. I am grateful to Paul Berg, Sean H. Carson, Robert Rowe, and the Editors and anonymous reviewers for providing valuable comments and suggestions on earlier versions of this article. Thanks also to the following people for discussing and sharing their research: Emmanuel Amiot, Gerard Assayag, Anne-Sylvie Barthel-Calvet, Ellen Flint, Benoît Gibson, James Harley, Evan Jones, Marcel Mesnage, André Riotte, and Sever Tipei.

References

Amiot, E., et al. 1986. "Duration Structure Generation and Recognition in Musical Writing." *Proceedings of the International Computer Music Conference.* San Francisco, California: International Computer Music Association, pp. 185–192.

Andreatta, M. 2003. "Méthodes algébriques en musique et musicologie du XXe siècle: aspects théoriques, analytiques et compositionnels." Dissertation. École des Hautes Etudes en Sciences Sociales.

Ariza, C. 2002. "Prokaryotic Groove: Rhythmic Cycles as Real-Value Encoded Genetic Algorithms." *Proceedings of the International Computer Music Conference.* San Francisco, California: International Computer Music Association, pp. 561–567.

Ariza, C. 2003. "Ornament as Data Structure: An Algorithmic Model based on Micro-Rhythms of Csángó Laments and Funeral Music." *Proceedings of the International Computer Music Conference.* San Francisco, California: International Computer Music Association, pp. 187–193.

Ariza, C. 2004. "An Object-Oriented Model of the Xenakis Sieve for Algorithmic Pitch, Rhythm, and Parameter Generation." *Proceedings of the International Computer Music Conference.* San Francisco, California: International Computer Music Association, pp. 63–70.

Assayag, Gérard. 1993. "Cao: vers la partition potentielle." *Les Cahiers de l'Ircam: La composition assistée par ordinateur* 1(3). Available at mediatheque.ircam.fr/articles/textes/Assayag93b/.

Barthel-Calvet, A. S. 2000. "Le Rythme dans l'Oeuvre et la Pensée de Iannis Xenakis," Dissertation. L'Ecole des Hautes Etudes en Sciences Sociales.

Barthel-Calvet, A. S. 2001. "Chronologie." In F. B. Mache, ed. *Portrait(s) de Iannis Xenakis.* Paris: Bibliotheque nationale de France, pp. 133–142.

Emmerson, S. 1976. "Xenakis Talks to Simon Emmerson." *Music and Musicians* 24:24–26.

Flint, E. R. 1989. "An Investigation of Real Time as Evidenced by the Structural and Formal Multiplicities in Iannis Xenakis' Psappha." Dissertation. Graduate School of the University of Maryland.

Forte, A. 1973. *The Structure of Atonal Music.* New Haven, Connecticut: Yale University Press.

Gibson, B. 2001. "Théorie des cribles." In M. Solomos, ed. *Presences of / Présences de Iannis Xenakis.* Paris: Centre de Documentation de la Musique Contemporaine, pp. 85–92.

Harley, J. 2004. *Xenakis: His Life in Music.* Oxford: Routledge.
Hawkins, D. 1958. "Mathematical Sieves." *Scientific American* 199:105–112.
Horsley, S. 1772. "The Sieve of Eratosthenes. Being an Account of His Method of Finding All the Prime Numbers." *Philosophical Transactions (1683–1775)* 62:327–347.
Jones, E. 2001. "Residue-Class Sets in the Music of Iannis Xenakis: An Analytical Algorithm and a General Intervallic Expression." *Perspectives of New Music* 39(2):229–261.
Koenig, G. M. 1970. "Project Two—A Programme for Musical Composition." *Electronic Music Reports* 3. Utrecht: Institute of Sonology.
Malherbe, C., G. Assayag, and M. Castellengo. 1985. "Functional Integration of Complex Instrumental Sounds in Musical Writing." *Proceedings of the International Computer Music Conference* San Francisco, California: International Computer Music Association: pp. 185–192.
Mesnage, M. 1993. "Mophoscope, a Computer System for Music Analysis." *Interface* 22(2):119–131.
Mesnage, M., and A. Riotte. 1993. "Modélisation informatique de partitions, analyse et composition assistées." *Les Cahiers de l'Ircam: La composition assistée par ordinateur* 1(3). Available at mediatheque.ircam.fr/articles/textes/Mesnage93a/.
Riotte, A. 1979. *Formalisation de structures musicales.* Paris: Université de Paris 8, Département d'Informatique.
Riotte, A. 1992. "Formalisation des échelles de hauteurs en analyse et en composition." Actes du Colloque "Musique et Assistance Informatique." Marseille: Musique et Informatique de Marseille.
Solomos, M. 1996. *Iannis Xenakis (Echos du XXe siècle).* Mercuès: P.O. Editions.
Solomos, M. 2002. "Xenakis' Early Works: From 'Bartokian Project' to 'Abstraction'." *Contemporary Music Review* 21(2–3):21–34.
Squibbs, R. 1996. "An Analytical Approach to the Music of Iannis Xenakis: Studies of Recent Works." Dissertation. Yale University.
Squibbs, R. 2002. "Some Observations on Pitch, Texture, and Form in Xenakis' *Mists.*" *Contemporary Music Review* 21(2–3):91–108.
Tipei, S. 1975. "MP1—A Computer Program for Music Composition." *Proceedings of the Second Music Computation Conference.* Urbana: University of Illinois at Urbana-Champaign, pp. 68–82.
Tipei, S. 1981. "Solving Specific Compositional Problems with MP1." *Proceedings of the International Computer Music Conference.* San Francisco, California: Computer Music Association, pp. 68–82.
Tipei, S. 1987. "Maiden Voyages: A Score Produced with MP1." *Computer Music Journal* 11(2):49–64.
Tipei, S. 1989. "The Computer: A Composer's Collaborator." *Leonardo Journal* 22(2):189–195.
Varga, B. A. 1996. *Conversations with Iannis Xenakis.* London: Faber and Faber.
Vriend, J. 1981. "*Nomos Alpha* for Violoncello Solo (Xenakis 1966): Analysis and Comments." *Interface* 10:15–82.
Xenakis, I. 1963. *Musiques Formelles.* Paris: Editions Richard-Masse.
Xenakis, I. 1965. "La voie de la recherche et de la question." *Preuve* 177:33–36.
Xenakis, I. 1966. "Vers une philosophie de la Musique." *Gravesaner Blätter* 29:39–52.
Xenakis, I. 1967. "Vers une métamusique." *La Nef* 29:117–140.
Xenakis, I. 1968. "Vers une philosophie de la musique." *Revue d'Esthétique* 21(2–4):173–210.
Xenakis, I. 1970. "Towards a Metamusic." *Tempo* 93:2–19.
Xenakis, I. 1971. *Formalized Music.* Bloomington: Indiana University Press.
Xenakis, I. 1976. *Musique, Architecture.* Paris: Casterman.
Xenakis, I. 1988. "Sur le Temps." In *Redécouvrir le Temps.* Bruxelles: Editions de l'Université de Bruxelles 193–200.
Xenakis, I. 1989. "Concerning Time." *Perspectives of New Music* 27(1):84–92.
Xenakis, I. 1990. "Sieves." *Perspectives of New Music* 28(1):58–78.
Xenakis, I. 1992. *Formalized Music.* Hillsdale, New York: Pendragon Press.
Xenakis, I. 1994. *Kéleütha.* Paris: L'Arche.
Xenakis, I. 1996. "Determinacy and Indeterminacy." *Organised Sound* 1(3):143–155.