

ANALYTICAL AND COMPOSITIONAL APPLICATIONS OF A NETWORK-BASED SCALE MODEL IN MUSIC21

Christopher Ariza

Michael Scott Cuthbert

Music and Theater Arts Section
Massachusetts Institute of Technology

ABSTRACT

Like many artifacts of music notation and theory, musical scales are easy to represent in software for simple cases, but rapidly become very difficult for more complex cases (e.g., melodic minor scale, Indian ragas, or microtonal scales). The BoundIntervalNetwork and Scale objects of the open-source `music21` toolkit provide new and powerful tools for abstracting and manipulating scales as used in actual compositions. Using a novel application of a constrained node-and-edge graph model, with intervals on edges and probability weightings on nodes, `music21` interval networks aid analysts in searching and annotating otherwise difficult-to-find moments in musical pieces (regardless of representational encoding) and can help composers in writing new pieces that conform to complex, asymmetrical, non-octave-bound, and even non-deterministic scalar models. This paper introduces the low-level BoundIntervalNetwork and high-level Scale objects, and, through numerous examples both in Python code and musical notation, demonstrates their usage and potential.

1. INTRODUCTION

Musical scales have more than theoretical or pedagogical value: they offer important resources for composers and tools for analysts. Even those scales that are more often found in theory texts than in actual pieces of music add to our knowledge of musical structures. Software tools for modeling, analyzing, and creating music need good representations of scales. At first, a scale might seem like a concept that could be modeled simply, such as with an ordered list of pitches. Yet, like many musical structures deployed in software for musical analysis, representation, and composition, the simplest concepts are insufficient for flexible implementations. For example, common scale-types, such as the melodic minor, have different pitches when ascending versus descending; some scale-types, such as those derived from Indian musical traditions, may have both non-linear contours and repeated notes in their distinctive ascending and descending forms; scales that do not repeat at the octave pose particular difficulties in encoding their upper and lower boundaries.

A list of pitches, represented symbolically or otherwise, will thus not accommodate the true diversity of scale forms, but this problem has not prevented such encodings from being the norm. There are numerous examples of such shallow encodings. For example, WolframTones [12], a web-based music generator employing cellular automata, claims to offer “all standard named scales, in all major standard musical traditions.”

Yet, these scales are explicitly specified as subsets of twelve semitones (pitch classes) and are encoded as twelve-bit binary strings. While the system offers over 300 named scale forms, the melodic minor (the most common form of minor in all common-practice Western music) is not available; the encodings of numerous Indian-derived scales called *raga*, *mela*, or *that*, all are fixed within one octave and do not distinguish between ascending or descending forms; and, of course, microtonal and non-octave repeating scales are not represented. This binary representation is illustrative of a common shortcoming in describing and encoding musical scales. In Gareth Loys’s extensive *Musimathics*, for example, scales are defined simply as “an ordered set of pitches, together with a formula for specifying their frequencies” [5].

Scala, an extensive system written in Ada for exploring and deploying (as tuning tables for a wide variety of software packages) non-equal-tempered and microtonal scales, tunings, and temperaments, recognizes over 1,200 musical modes and features an archive of more than 3,900 scales. Modes, however, are defined as an ordered list of ascending scale degrees [7]. The Scala scale file format similarly represents scales as a single, fixed list of cent or ratio offsets above an implied tonic [8]. Such a representation does not allow for unique ascending or descending pathways, nor defined spelling of quarter-tone inflections. Though it is a broadly supported format and permits great specification of tuning, Scala’s representation is, in some ways, as structurally limited as WolframTones’s binary arrays.

A list of pitches can, however, be generalized into a list of interval spacings, describing not just one pitch formation but all pitch formations. Interval spacings can then be generalized to use sophisticated interval objects, providing proper enharmonic spellings under transposition and microtonal spacings. Finally, a list of interval objects can be generalized as a directed network, where nodes are undefined pitch placeholders and edges are intervals. Such a model permits multiple pathways in ascent or descent, pathway branching, and weighted or probabilistically-selected pathways. This new model of the scale, based on an object called the BoundIntervalNetwork, is the focus of this paper.

With this model, a scale, either as an abstract interval network (such as a melodic-minor scale) or as a concrete pitch collection (such as the G melodic-minor scale from D4 to D6), can be created and deployed. Pitches can be obtained from the scale by specifying a degree, degrees can be obtained from a pitch, and arbitrary pitch collections can be used to derive one or more new scales that fit this collection. Scales can be transposed, can be walked with variable step sizes, and can

be used to create new concrete scales based on any scale degree specification. No other software system has offered such a comprehensive model of scale formation.

This powerful model and interface, implemented in the BoundIntervalNetwork and associated Scale objects, fully models diatonic scales and modes, harmonic and melodic minor scales, chromatic, pentatonic, octatonic, and whole tone scales, non-octave repeating pitch scales such as the sieves of Iannis Xenakis [13], microtonal scales such as Harry Partch’s 43-tone scale [9], and scales derived from Indian raga that define both contour and ascending and descending formations. Further, any scale defined in the Scala scale file format, and a scale within the Scala archive of over 3,900 scales, can be quickly deployed.

This new model is implemented in Python as part of the music21 toolkit, an open-source, cross-platform framework written in Python for computer-aided musicology [2]. (See mit.edu/music21 for downloads and documentation.). This paper will describe the object design and interface, and demonstrate applications in automated analysis and algorithmic composition.

2. OBJECT HIERARCHIES AND COMPOSITIONS

To inform the discussion that follows, we first present the basic definitions of the objects, their hierarchies, and their compositions. The BoundIntervalNetwork is the core object for all scale processing. A BoundIntervalNetwork is composed of three or more Node objects and two or more Edge objects. As a low-level object, BoundIntervalNetwork objects do not need to be used by users who only wish to work with higher-level scale objects.

The Scale is a base class providing common resources to both AbstractScale and ConcreteScale objects. AbstractScales are composed of one (or possibly more) BoundIntervalNetworks and expose the BoundIntervalNetwork interface to ConcreteScales. AbstractScales represent a type of scale independent of specific pitches, such as all major scales (as opposed to, say, the D-major scale); they are responsible for defining the meaning of various scale degrees, configuring the BoundIntervalNetwork on instantiation, and passing scale-type parameters to the BoundIntervalNetwork. ConcreteScales, on the other hand, represent a type of scale with a specific pitch collection; they are responsible for assigning a pitch value to a scale degree, and, by passing this assignment through the AbstractScale and BoundIntervalNetwork, providing realized scale pitches.

Figure 1 illustrates the composition of these objects. Closed diamonds are object compositions and suggest life-time object responsibility.

The ConcreteScale and its subclasses provide the main public interface for working with Scales. Each ConcreteScale contains one AbstractScale instance. For convenience, the same AbstractScale class may be used

by multiple ConcreteScale classes (this relationship is not represented in Figure 1). For example, an instance of the AbstractDiatonicScale is used inside of MajorScale, MinorScale, LydianScale, and related diatonic scales. In some cases, a unique AbstractScale class is required for a single ConcreteScale, such as the AbstractMelodicMinorScale used in the concrete MelodicMinorScale class.

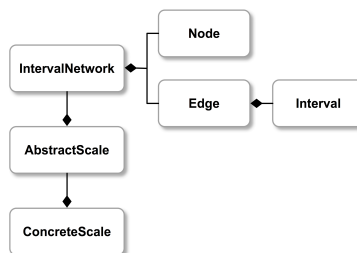


Figure 1. Object composition of Scale-related classes.

Figure 2 summarizes the object inheritance hierarchy of Scale subclasses, and provides a small sampling of ConcreteScale and AbstractScale subclasses. Open arrows point to parent classes inherited by subclasses.

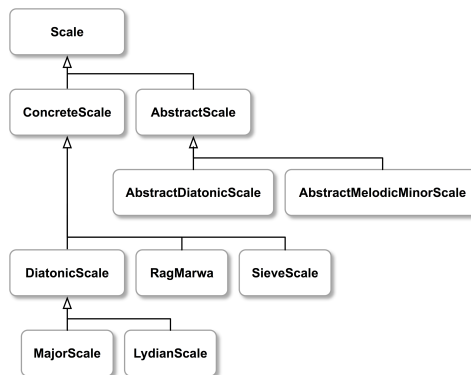


Figure 2. Object inheritance and example subclasses of ConcreteScale and AbstractScale.

3. THE BOUNDINTERVALNETWORK OBJECT

The BoundIntervalNetwork is a limited or bound form of a directed graph. Specifically, the class models a directed graph where each Edge encapsulates a music21 Interval object (defining both the size and direction of a potential pitch transposition with correct enharmonic spelling) and is associated with a pathway direction (ascending, descending, or bi-directional); each Node represents an unrealized pitch, is defined with an integer degree value, and has a numerical weight representing the probability of this path being taken (as used for non-deterministic scales). Each graph also has two nodes defined as termini (low and high) that function as wrapping, cyclical boundaries.

While a free-form interval network might permit any possible connection, the bound network requires (1) at least one ascending and one descending pathway con-

necting both termini and (2) that it must be possible for each node to be part of a pathway that connects both termini. Future implementation of the `IntervalNetwork` base class may support free-form interval networks in pitch class space, such as those described in Johnson [3], alternative formulations found in Lewin [4], or pitch *Tonnetz* such as those demonstrated by Leonhard Euler as early as 1739.

In this model, each `Node` represents a pitch but does not contain a specific `Pitch` object. When a `Pitch` object is associated with a `Node`, new pitches in the scale are realized by navigating the pathway in either ascending or descending directions (though not necessary ascending or descending pitch space values, as in the case of scales that are considered ascending, yet curve back on themselves along the way, such as some *ragas* or scales described in Slonimsky [10]). As each `Edge` defines an `Interval`, the next `Node` in the pathway is realized by transposing the source pitch. This transposition may be up or down in pitch space and is dependent on the supplied `Interval` definition. When realizing an ascending pathway, unaltered `Edge Intervals` are used for transposition; when realizing a descending pathway, `Edge Intervals` are used for transposition in reverse (e.g., an ascending minor second in ascent, a descending minor second in descent).

`BoundIntervalNetworks` are cyclical, though their realized pitch space collections may be infinite. When a pathway arrives at a terminus, the realized pitch is treated as a pitch in the alternate terminus, and the pathway continues. For example, if a pitch is realized for the high terminus, the next edge is found by treating the just-realized pitch as a pitch for the low terminus. In this way the termini nodes are essentially the same, wrapping around in the graph. The realized pitch sequence, however, need not wrap, and often forms an infinite sequence of pitch space values. While a theoretical graph might easily represent the two termini as one node, this musically-informed design uses two distinct nodes to better represent conventional models of scales.

The most relevant functionality of `BoundIntervalNetworks` is exposed in the interface of the `ConcreteScale`, which will be discussed later. Understanding a few attributes and methods of the `BoundIntervalNetwork`, however, will clarify the design. Each `BoundIntervalNetwork` has two stored `Boolean` values that declare its basic structural characteristics: the `deterministic` attribute declares whether every ascending and descending pathway for a given `Pitch-Node` assignment will always be the same; the `octaveRepeating` attribute declares whether all realized pitches repeat for each octave. The `realize()` method, given a single `Pitch`, a `Node` to which that `Pitch` is assigned to, a pathway direction, and a `Pitch` range, navigates a pathway through the network and returns the resultant `Pitches`, paired with references to the specific `Nodes` used in realization. For deterministic `BoundIntervalNetworks`, pitch segments obtained from `realize()` are cached and reused when possible.

While interval spacings drive the formation of scales, it is useful to apply interval transformations to resultant `Pitch Nodes` after `Edge-derived` transpositions. An `alteredDegrees` dictionary accommodates this functionality. It is a data structure, stored in `AbstractScales`, that is passed to `BoundIntervalNetworks` whenever pitches need to be realized. This approach permits two-levels of design: (1) the `BoundIntervalNetwork` structure, based on pathways of intervals, and (2) the `AbstractScale alteredDegrees` dictionary, based on transforming realized `Nodes`. A complex `AbstractScale` might, for example, adjust the `alteredDegrees` dictionary for stochastic variation or contextual pitch adjustments. An application of the `alteredDegrees` dictionary is demonstrated below as the `HarmonicMinorScale`.

The following figures illustrate reduced, hypothetical `BoundIntervalNetwork` graph structures, and describe archetypical formations. In all cases, `Nodes` labeled *a* and *b* represent the low and high termini respectively. `Nodes` at parallel vertical positions share the same degree value. In all cases, the number of nodes and edges can be increased to accommodate more intervals. While graph structures define scales as conjunct movements, any skip or disjunct motion is possible.

Figure 3 illustrates the most common types of deterministic `IntervalNetworks`. These structures may or may not be octave repeating. Figure 3a is a simple bi-directional structure, the type of scale most software models. Any number of `Nodes` may intervene between the termini. For any defined degree, there is only one `Node` available. Figure 3b has independent ascending and descending pathways. Between the termini there are two nodes for each degree. Requests for a pitch based on a degree are resolved depending on a provided pathway direction. Figure 3b could represent scale degrees five to eight of the melodic minor scale. Figure 3c has a bidirectional segment and an independent pathway segment. An expanded structure, similar to this one, is used to represent the entire melodic minor scale as presented below. Figures 3b and 3c represent scale archetypes not available in existing software implementations.

Not all scales and scale-like objects produce the same `Pitch` sequence with each realization. Figure 4 illustrates some possibilities for these non-deterministic `BoundIntervalNetworks`. Figure 4a, for example, has all bi-directional `Edges`, but branches in both ascent and descent for `Node y`. The determination of which `Node` is realized is determined by weighted random selection of the possible destination `Nodes`. Depending on the weights, for instance, the `Pitch` in `Node x` might ascend to `Node y` 40% of the time, and directly to `Node z` 60% of the time. Figure 4b defines a structure with two ascending and two descending pathways. Again, the choice of `Node`, when more than one is available for a given pathway direction, is based on weighted random selection. When requesting a pitch from a degree that is associated with more than one `Node`, weighted random selection is again used. For example, if `Nodes q, r, and s` are all associated with degree 2, and a user requests an ascending pitch at this degree, one of `Node q` or *r* (the

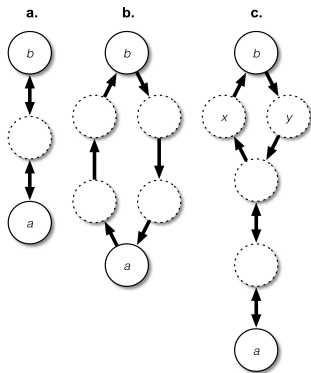


Figure 3. Deterministic BoundIntervalNetwork structures.

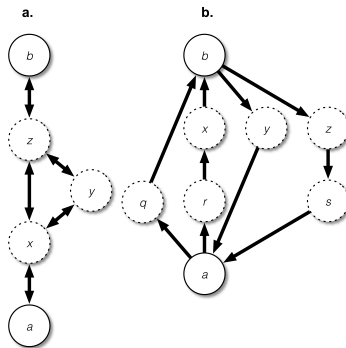


Figure 4. Non-deterministic BoundIntervalNetwork structures.

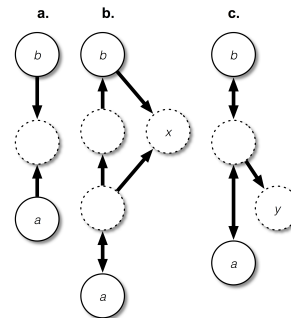


Figure 5. Directed graphs that cannot be modeled as BoundIntervalNetworks.

only nodes on an ascending pathway) will be selected according to their respective weights. The modeling and deployment of such scale formations is unique to the Scale model presented in this paper.

As stated previously, the BoundIntervalNetwork is a directed graph with some restrictions. Example 5 shows these restrictions through three examples of networks that, while potentially modeling interesting pitch collections, are illegal. Figure 5a has no complete pathway between the lowest and highest termini. Figure 5b has an ascending pathway between the termini, but no complete descending pathway; further, Node *x* is not part of a complete pathway. Though Figure 5c has both a complete ascending and descending pathway, its Node *y* is not part of a complete pathway.

4. ABSTRACTSCALES, CONCRETE-SCALES, AND THEIR SUBCLASSES

Music21 possesses two main types of scale classes: (1) AbstractScales expose interval networks independent of specific pitch positions, while (2) ConcreteScales localize a scale with specific pitches for each node or scale degree. While the main role of an AbstractScale is to configure and expose the interface of the BoundIntervalNetwork, AbstractScales also provide the additional functionality of a `tonicDegree` attribute, which defines the tonic position as being some numerical degree. The term tonic here is used loosely, and may refer to a tonic on scale degree 1 in the tonal sense, or may refer to a *finalis*, such as scale degree 4 in chant's plagal Dorian mode. AbstractScales can also export any scale in the Scala file format with the `write('scala')` or `show('scala')` methods and arguments.

The main public interface for working with scales are the ConcreteScale and its subclasses. A ConcreteScale, on instantiation, is given a music21 Pitch object as a tonic. Additionally, some ConcreteScale subclasses may take other parameters to specialize their mode or formation. To localize the pitch collection, the `tonicDegree` attribute in the AbstractScale is combined with a stored tonic Pitch in the ConcreteScale. The ConcreteS-

cale provides a wide range of functionality: access to the stored AbstractScale via the `abstract` property, a `transpose()` method, a method to get a list of Pitches for any specified pitch range (`getPitches()`) or as a Chord (`getChord()`), methods for obtaining one or more Pitches from one or more degree values (`pitchFromDegree()`, `pitchesFromScaleDegrees()`), a method to get a degree from a pitch (`getScaleDegreeFromPitch()`), a method to get the next pitch in the scale given another Pitch and a pathway direction (`next()`), various methods for comparing this realized scale to other scales (`match()`, `findMissing()`), and methods to create a new ConcreteScale of this class based on arbitrary pitch collections or a pitch assigned to new scale degree (`derive()`, `deriveByDegree()`). Examples of these methods are given below.

4.1. Modeling all Diatonic Scales and Modes

All diatonic scales and modes (e.g., major, natural minor, and historical and theoretical diatonic modes) are made available through pairing of an AbstractDiatonicScale instance and a ConcreteScale subclass. In addition to the `tonicDegree` attribute, the AbstractDiatonicScale class defines a `dominantDegree` attribute. The suggestion of a dominant is again used loosely, and may refer to something more like a modal reciting tone. Concrete diatonic scales are given additional features by inheriting a common ConcreteScale subclass: DiatonicScale. This class provides methods for all modes such as `getLeadingTone()`, `getParallelMinor()`, `getParallelMajor()`, `getRelativeMinor()`, and `getRelativeMajor()`.

The Python example in Figure 6 demonstrates basic functionality of the MajorScale. The same functionality is available with all DiatonicScale subclasses.

```
# providing a tonic
gScale = scale.MajorScale('g4')
esScale = scale.MajorScale('e-3') # E-flat
# comparing Concrete and Abstract Scales
assert (gScale == esScale) == False
assert (gScale.abstract == esScale.abstract) == True
# getPitches() with and without arguments
```

```

assert str(gScale.getPitches()) ==
    '[G4, A4, B4, C5, D5, E5, F#5, G5]'
assert str(esScale.getPitches('c2', 'd3')) ==
    '[C2, D2, E-2, F2, G2, A-2, B-2, C3, D3]'
# additional functionality as a Chord
assert gScale.getChord().forteClass == '7-35'
# given a degree, get a pitch, and vice versa
assert str(gScale.pitchFromDegree(5)) == 'D5'
assert str(esScale.pitchesFromScaleDegrees(
    [7,2], 'e-6', 'e-9')) == '[F6, D7, F7, D8, F8,
    D9]'
assert esScale.getScaleDegreeFromPitch('d') ==
    7
# get the next pitch given step directions
match = [pitch.Pitch('g2')]
for dir in [1, 1, 1, -2, 4, -1, 1, 1, 1]:
    # get next pitch based on the last
    match.append(gScale.next(match[-1], dir))
assert str(match), '[G2, A2, B2, C3, A2, E3,
    D3, E3, F#3, G3]'
# derive new scales based on pitches or degree
print gScale.derive(['c4', 'g4', 'b8', 'f2'])
<music21.scale.MajorScale C major>
print gScale.deriveByDegree(7, 'C#')
<music21.scale.MajorScale D major>
    
```

Figure 6. Creating, using, and deriving MajorScale instances.

Modal scale definitions permit access to both relative major and minor scales, as well as access to appropriate *finalis* and reciting-tone values. These attributes are demonstrated in Figure 7.

```

ph = scale.PhyrgianScale('g4')
assert str(ph.getPitches('F2', 'G3')) ==
    '[F2, G2, A-2, B-2, C3, D3, E-3, F3, G3]'
assert str(ph.getRelativeMajor()) ==
    '<music21.scale.MajorScale E- major>'
assert str(ph.getTonic()),
    str(ph.getDominant()) == ('G4', 'D5')
hd = scale.HypodorianScale('a6')
assert str(hd.getDominant()) == 'C7'
    
```

Figure 7. Useful methods of the PhrygianScale and Hypodorian modal scales.

The diatonic scales offer numerous resources for analyzing melodic material found in common-practice functional harmony. Applications include labeling pitch collections by scale degrees and illustrating parallel and simultaneous key interpretations based on scale segments. The following example, Figure 8, employs the `music21.analysis.search.findConsecutiveScale()` routine to find and label consecutive, mono-directional scale segments, consisting of at least five degrees, for two major scales (G, D), in an excerpt from the first violin of W. A. Mozart's String Quartet No. 1. This illustrates extended linear scalar passages and multiple key interpretations. Such a routine could easily be extended to systematically find all such passages in an entire work or corpus, permitting generalizations about melodic writing and key usage by part, work, period, or composer.

```

scGMajor = scale.MajorScale('g4')
scDMajor = scale.MajorScale('d4')
s = corpus.parseWork(
    'mozart/k80/movement1').measures(21,25)
for part in s.parts:
    for sc in [scGMajor, scDMajor]:
        groups = analysis.search.
    
```

```

        findConsecutiveScale(part.flat, sc,
        degreesRequired=5,
        comparisonAttribute='name')
        for group in groups:
            for n in group['stream'].notes:
                n.addLyric('%s^%s' %
                    (sc.getTonic().name,
                    sc.getScaleDegreeFromPitch(
                        n.pitch)))
s['violin i'].show('musicxml')
    
```




Figure 8. Searching and labeling consecutive directional major scale passages.

4.2. Altered Minor Scales

As already suggested, the harmonic and melodic minor scales pose a challenge to software modeling. As implemented here, both have concrete subclasses that are derived from `DiatonicScale`; each, however, has a custom `AbstractScale` subclass.

The harmonic minor scale is best modeled as a natural minor scale with an altered (raised) seventh degree. This approach permits shifting a `Node` rather than altering the spacing of two `Edges`, and provides proper enharmonic spelling. The `AbstractHarmonicMinorScale` class defines a private `alteredDegrees` dictionary. This dictionary defines an augmented unison transposition for scale degree seven. For each call on `BoundIntervalNetwork`'s `realize()` method, the `AbstractHarmonicMinorScale` passes a reference to this dictionary and, when necessary, raises the seventh scale degree.

Figure 9 demonstrates basic functionality of the `HarmonicMinorScale`, as well notation of a simple melodic line with disjunct melodic motion based on a numerical scale degree specification.

```

sc1 = scale.HarmonicMinorScale('a3')
assert str(sc1.getPitches()) ==
    '[A3, B3, C4, D4, E4, F4, G#4, A4]'
assert str(sc1.getTonic()),
    str(sc1.getDominant()) == ('A3', 'E4')
s = stream.Stream()
for d in [1, 3, 2, 1, 6, 5, 8, 7, 8]:
    s.append(note.Note(sc1.pitchFromDegree(
        d, equateTermini=False), type='eighth'))
s.show()
    
```



Figure 9. Employing the `HarmonicMinorScale`.

While the harmonic minor scale can be modeled with a single-pathway network, the melodic minor re-

quires, for the last two degrees, independent pathways in ascent and descent. The necessary structure is suggested in part by Figure 3c. While it is possible to represent such a structure with alteredDegrees, a true branching of the network is employed here.

Figure 10 demonstrates the MelodicMinorScale in ascent and descent, and creates a melody through iterative calls to the next() method, randomly moving up or down scale steps.

```

cmin = scale.MelodicMinorScale('c4')
assert str(cmin.getPitches(
    direction='ascending')) == '[C4, D4, E-4,
    F4, G4, A4, B4, C5]'
assert str(cmin.getPitches('c3', 'c5',
    direction='descending')) == '[C5, B-4, A-4,
    G4, F4, E-4, D4, C4, B-3, A-3, G3, F3, E-3,
    D3, C3]'
s = stream.Stream()
p = None
for i in range(16):
    dir = random.choice([-1, 1])
    for j in range(2):
        p = cmin.next(p, dir)
        s.append(note.Note(p, type='16th'))
s.show()

```



Figure 10. Randomly walking the C melodic minor scale.

Analytical applications of the melodic minor scale are significant. Because the direction of the scale defines the pitches used, non-connected (or non-networked) scale models will not be able to properly isolate melodic minor passages. Figure 11 finds melodic minor scale passages of four consecutive degrees in two keys (D and G minor) in *Contrapunctus III*, from J. S. Bach’s *Die Kunst der Fuge*.

```

scDMelodicMinor = scale.MelodicMinorScale('d4')
scGMelodicMinor = scale.MelodicMinorScale('g4')
part = corpus.parseWork('bwv1080/03')
part.parts[0].measures(46,51)
for sc in [scDMelodicMinor, scGMelodicMinor]:
    groups = analysis.search.findConsecutiveScale(part.flat, sc,
    degreesRequired=4,
    comparisonAttribute='name')
    for group in groups:
        for n in group['stream'].notes:
            n.addLyric('%s^%s' % (
            sc.getTonic().name.lower(),
            sc.getScaleDegreeFromPitch(n.pitch,
            group['direction'])))
part.show()

```



Figure 11. Searching and labeling consecutive directional melodic minor scale passages.

4.3. Chromatic Scales

Common symmetrical and chromatic scales, such as the whole tone scale, octatonic scale, or Olivier Messiaen’s modes of limited transposition [6], are easily represented with the BoundIntervalNetwork model.

The octatonic scale provides an example. The OctatonicScale class takes two initialization arguments: a value for the tonic, and a value for the mode, where the term mode here refers to the different rotations of the scale. The octatonic scale has two modes: one with a minor second as the first interval, and the other with a major second as the first interval. These modes can be selected with arguments given as numerical or string representations: “m2” or 1, and “M2” or 2, respectively.

Figure 12 demonstrates returning pitches from the two OctatonicScale formations.

```

sc1 = scale.OctatonicScale('e3', 'm2')
assert str(sc1.getPitches()) ==
'[E3, F3, G3, A-3, B-3, C-4, D-4, D4, E4]'
sc2 = scale.OctatonicScale('e3', 'M2')
assert str(sc2.getPitches()) ==
'[E3, F#3, G3, A3, B-3, C4, D-4, E-4, F-4]'

```

Figure 12. Employing the OctatonicScale.

4.4. Xenakis Sieves as BoundIntervalNetworks

Iannis Xenakis’s sieve (Xenakis 1990) is a compact notation for generating complex, microtonal, and periodic interval sequences and spacings. The application of sieves to pitch scales is particularly fruitful in that octave repetition is neither required nor assumed: sieve-based pitch scales can create infinite interval sequences simply from repeated patterns of intervals.

A sieve can be implemented as a bi-directional BoundIntervalNetwork, where each interval is a spacing between active sieve points. Terminology, notation, and implementation are taken from Ariza [1]. The Sieve-Scale permits any sieve string representation to be used as a creation argument. The period of the sieve is then used to find the termini, and degree values are automatically assigned within this range.

For example, a sieve in the form of the union of a three half-step periodicity and a four half step periodicity (3@0|4@0, or the union of a fully diminished seventh chord and an augmented triad) has a period of 12

half steps and is octave repeating. A sieve in the form of the union of three half-steps and seven half steps (3@0|7@0, or the union of a fully diminished seventh chord and a sequence of perfect fifths) has a period of 21 half steps and is not octave repeating. These scales, and their resultant pitches, are presented in Figure 13, along with a random permutation of pitches from a sieve spread over four octaves.

```

scl = scale.SieveScale('c4', '3@0|4@0')
assert str(scl.getPitches()) ==
    '[C4, E-4, F-4, G-4, A-4, A4, C5]'
sc2 = scale.SieveScale('c4', '5@0|7@0')
assert str(sc2.getPitches()) == '[C4, F4, G4,
B-4, D5, E-5, A-5, A5, C#6, E6, F#6, B6]'
s = stream.Stream()
pCollection = sc2.getPitches('c3', 'c7')
random.shuffle(pCollection)
for p in pCollection:
    s.append(note.Note(p, type='16th'))
s.show()

```



Figure 13. Applications of the SieveScale.

4.5. Raga-Derived Scales

Raga, or the melodic material of Hindustani and Carnatic musical traditions, is a complex conceptual framework extending well-bound traditional Western concepts of scale [11]. A comprehensive software model of all the aspects of Raga would include seasonal and temporal associations, emotional associations (*bhava* and *rasa*), common melodic fragments and motives, microtonal inflections, and numerous other attributes. However, raga are in some cases used like scales, though they are scales that commonly involve contour: an ascending or descending pathway may define both upward and downward intervals.

As a proof of modeling aptitude, raga-derived scales can be encoded as BoundIntervalNetworks. Two scales are presented, Asawari and Marwa. While not designed to be authoritative, these models are evidence of the power and flexibility of this new scale model.

Rag Asawari is unlike any previously-discussed scale in that its ascent has five pitches while its descent has seven. This means that some degrees are not available in the ascending form. This is modeled as a directional network similar to the model shown in Figure 3b.

Rag Marwa is unlike any previously-discussed scale in that ascent and descent each have contour and repeat the same pitch level twice. This means that a request for a scale degree, given only a Pitch, produces two possible results that are resolved by weighted random selection. Depending on usage, this scale may be non-deterministic.

Note that, as stated above, such scales cannot be completely encoded in the widely used Scala scale format, although Scala file-format output of a realized

pathway is available. The Python example in Figure 14 illustrates applications of these two scales.

```

ragA = scale.RagAsawari('g3')
assert str(ragA.getPitches(
    direction='ascending')) ==
    '[G3, A3, C4, D4, E-4, G4]'
assert str(ragA.getPitches(
    direction='descending')) ==
    '[G4, F4, E-4, D4, C4, B-3, A3, G3]'
ragM = scale.RagMarwa('g3')
assert str(ragM.getPitches(
    direction='ascending')) ==
    '[G3, A-3, B3, C#4, E4, F#4, E4, G4, A-4]'
assert str(ragM.getPitches(
    direction='descending')) ==
    '[A-4, G4, A-4, F#4, E4, C#4, B3, A-3, G3]'
p1 = None
s = stream.Stream()
for dir in ([1]*10) + ([-1]*8) + ([1]*4) +
    ([-1]*3) + ([1]*4):
    p1 = ragA.next(p1, dir)
    s.append(note.Note(p1, quarterLength=.25))
s.show()

p1 = None
s = stream.Stream()
for dir in ([1]*10) + ([-1]*8) + ([1]*4) +
    ([-1]*3) + ([1]*4):
    p1 = ragM.next(p1, dir)
    s.append(note.Note(p1, quarterLength=.25))
s.show()

```

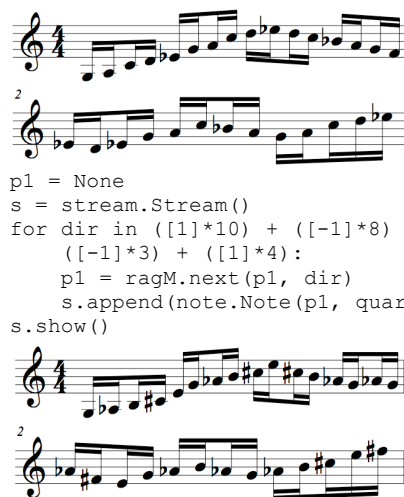


Figure 14. Examples of two raga-derived scale classes, RagAsawari and RagMarwa.

4.6. Microtonal Scales from Scala Scale Files

Scala scale files encode a vast variety of microtonal and non-standard scales, tunings, and temperaments. The ScalaScale ConcreteScale class, given a tonic and a Scala scale (denoted as any file name within the complete Scala scale archive bundled with music21, as a file path to a .scl file, or as a complete string representation of such a file), will create a bi-directional BoundIntervalNetwork representation of the desired scale, with complete microtonal specification and the same features as other ConcreteScale subclasses. In Figure 15, a single octave of two different microtonal slendro scales are created as two music21 Parts attached to a common Score, illustrating in parallel their distinctive microtonal tunings.

```

s = stream.Stream()
s.append(meter.TimeSignature('6/4'))
sc1 = scale.ScalaScale('c2', 'slendro_ang2')
sc2 = scale.ScalaScale('c2', 'slendroc5.scl')

```

```

p1 = stream.Part()
p1.append([note.Note(p, lyric=p.microtone) for
p in scl.pitches])
p2 = stream.Part()
p2.append([note.Note(p, lyric=p.microtone) for
p in sc2.pitches])
s.insert(0, p1); s.insert(0, p2)
s.show()

```

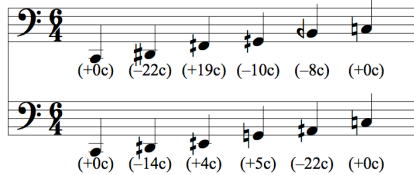


Figure 15. Comparing microtonal tuning of two slendero scales from the Scala scale archive.

4.7. Non-Deterministic Scales

While there are many analytical applications for deterministic scales, non-deterministic scales are particularly well-suited for use in compositional applications. Custom BoundIntervalNetwork objects can be designed to describe a wide range of scale structures. An example of such a scale is provided by the WeightedHexatonicBlues class. This scale models the combination of what are sometimes called a “minor pentatonic” scale with a “hexatonic blues” scale. In this network design, the raised fourth scale degree is placed on an alternative pathway branch, similar to the model shown in Figure 4a. Thus, depending on weighted random selection, an ascending or descending C-tonic WeightedHexatonicBlues pathway may move from F to G, or may alternatively move from F to F-sharp to G.

The Python example in Figure 16 illustrates generating a melodic passage with this scale.

```

whb = scale.WeightedHexatonicBlues('c3')
p = 'c3'
s = stream.Stream()
for n in range(32):
    p = whb.next(p, random.choice([-1, 1]))
    n = note.Note(p,
quarterLength=random.choice([.5, .25, .25]))
s.append(n)

```

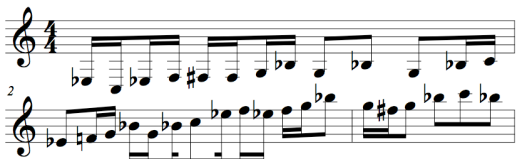


Figure 16. An example of a generating a melody with the non-deterministic WeightedHexatonicBlues.

5. FUTURE WORK

With these new object models, as well as support for Scala scale files and integrated access to the Scala scale archive, thousands of scales are already available. Our goal is for any scale to be fully represented, all while maintaining a powerful and uniform interface. Addi-

tional analytical and searching routines, as well as the ability to dynamically replace scales or temperaments of existing works, will provide even further resources for analysis and composition.

The present implementation of the BoundIntervalNetwork asserts that there is always a single high and low terminus. This limitation may be removed with the development of the IntervalNetwork object, a parent class of BoundIntervalNetwork that has no restrictions on network structure, and that could model musical transformations (including chordal transformations) through time.

6. ACKNOWLEDGEMENTS

Development of the BoundIntervalNetwork and the music21 toolkit is conducted as part of a multi-year research project funded by the Seaver Institute. Thanks also to Manuel Op de Coul for permission to distribute the Scala scale archive with music21.

7. REFERENCES

- [1] Ariza, C. 2005. “The Xenakis Sieve as Object: A New Model and a Complete Implementation.” *Computer Music Journal* 29(2): pp. 40-60.
- [2] Cuthbert, M. S., and C. Ariza. 2010. “music21: A Toolkit for Computer-Aided Musicology and Symbolic Music Data.” *Proceedings of the International Society on Music Information Retrieval*: pp. 637–42.
- [3] Johnson, J. 1997. *Graph Theoretical Models of Abstract Musical Transformation: An Introduction and Compendium for Composers and Theorists*. Santa Barbara, California: Greenwood Press.
- [4] Lewin, D. 1987. *Generalized Musical Intervals and Transformations*. New Haven: Yale University Press.
- [5] Loy, G. 2007. *Musimathics: The Mathematical Foundations of Music, Volume 2*. Cambridge: MIT Press.
- [6] Messiaen, O. 1944. *The technique of my musical language*. Paris: Alphonse Leduc.
- [7] Op de Cou, Manuel. *List of musical modes*. Available online at www.huygens-fokker.org/docs/modename.html
- [8] Op de Cou, Manuel. *Scala scale file format*. Available online at www.huygens-fokker.org/scala/scl_format.html
- [9] Partch, H. 1949. *Genesis Of A Music: An Account Of A Creative Work, Its Roots, And Its Fulfillment*. Madison: University of Wisconsin Press.
- [10] Slonimsky, N. 1947. *Thesaurus of Scales and Melodic Patterns*, New York: Scribners.
- [11] Wade, B. C. 1994. *Music in India: The Classical Traditions*. New Delhi: Manohar.
- [12] Wolfram Research. 2005. “About WolframTones.” Available online at <http://tones.wolfram.com/about/>.
- [13] Xenakis, I. 1990. “Sieves.” *Perspectives of New Music* 28(1): pp. 58-78.