music21: A Toolkit for Computer-Aided Musicology and Symbolic Music Data

Michael Scott Cuthbert **Christopher Ariza**
Music and Theater Arts
Massachusetts Institute of Technology
{cuthbert, ariza}@mit.edu

ABSTRACT

Music21 is an object-oriented toolkit for analyzing, searching, and transforming music in symbolic (score-based) forms. The modular approach of the project allows musicians and researchers to write simple scripts rapidly and reuse them in other projects. The toolkit aims to provide powerful software tools integrated with sophisticated musical knowledge to both musicians with little programming experience (especially musicologists) and to programmers with only modest music theory skills.

This paper introduces the music21 system, demonstrating how to use it and the types of problems it is well-suited toward advancing. We include numerous examples of its power and flexibility, including demonstrations of graphing data and generating annotated musical scores.

1. INTRODUCTION: WHY MUSIC21?

Computers have transformed so many aspects of musicology—from writing and editing papers, to studying manuscripts with digital files, to creating databases of composers' letters, to typesetting editions—that it is incredible that most analytical tasks that music historians perform remain largely untouched by technology. The study of the rich troves of musical data in scores, sketches, intabulations, lead-sheets, and other sources of symbolic music data is still done almost exclusively by hand. Even when databases and spreadsheets are employed, they are usually created for a single purpose. Such specialized approaches cannot easily be reused.

Computer scientists often assume that, compared to working with scanned images of scores or sound files, manipulating symbolic data should be a cinch. Most of the information from a score can easily be converted to text-based or numeric formats that general-purpose statistical or information-retrieval tools can manipulate. In practice the complexities of music notation and theory result in these tools rarely being sufficient.

For instance, a researcher might want to compare how closely the music of two composers adheres to a particular scale (say, the major scale). What begins as a straightforward statistical problem requiring little musical knowledge—simply encode which notes are in the scale of the piece's key and which are not—can quickly grow beyond the capabilities of general statistics packages. Suppose that after some initial work, our researcher decides that notes on stronger beats should be weighed more heavily than those on weaker beats. Now she must either add the information about beats by hand to each note or write a new algorithm that labels the beats. Beat labeling is another task that initially seems easy but rapidly becomes extremely troublesome for several reasons. Are grace-notes accented or unaccented? Only a musically-trained ear that also knows the norms of an era can tell. Incompletely-filled measures, such as pickup measures and mid-bar repeats, present problems for algorithms. As the researcher's corpus expands, the time spent on meta-research expands with it. What began as a straightforward project becomes a set of tedious separate labors: transforming data from multiple formats into one, moving transposing instruments into sounding pitch, editorial accidentals in early music, or finding ways of visualizing troublesome moments for debugging.

Researchers in other fields can call upon general-purpose toolkits to deal with time-consuming yet largely solved problems. For instance, a scientist working with a large body of text has easy access to open-source libraries for removing punctuation, converting among text-encoding formats, correcting spelling, identifying parts of speech, sentence diagramming, automatic translation, and of course rendering text in a variety of media. Libraries and programs to help with the musical equivalents of each of these tasks do exist, but few exchange data with each other in standardized formats. Even fewer are designed in modern, high-level programming languages. As a result of these difficulties, computational solutions to musicological problems are rarely employed even when they would save time, expand the scope of projects, or quickly find important exceptions to overly broad pronouncements.

The music21 project (http://web.mit.edu/music21) expands the audience for computational musicology by creating a new toolkit built from the ground up with intuitive simplicity and object-oriented design throughout. (The "21" in the title comes from the designation for MIT's classes in Music, Course 21M.) The advantages of object-oriented design have led to its wide adoption in many realms of software engineering. These design principles have been employed in music synthesis and generation systems over the past 25 years [2, 9, 10] but have not been thoroughly integrated into packages for the analysis of music as symbolic data. Humdrum, the most widely adopted software package [6], its contemporary ports [7, 11], and publications using these packages show the great promise of computational approaches to music theory and musicology. Yet Humdrum can be difficult to use: both programmers and non-programmers alike may find its reliance on a chain of shell-scripts, rather than object-oriented libraries, limiting and not intuitive.

Nicholas Cook has called upon programmers to create for musicologists "a modular approach involving

an unlimited number of individual software tools" [3]. A framework built with intuitive, reusable, and expandable objects satisfies Cook's call without sacrificing power for more complex tasks.

As a new, open-source, cross-platform toolkit written in Python, `music21` provides such a modular approach, melding object-oriented music representation and analysis with a concise and simple programming interface. Simplicity of use, ease of expansion, and access to existing data are critical to the design of `music21`. The toolkit imports Humdrum/Kern, MusicXML [4], and user-defined formats (with MIDI and MuseData forthcoming). Because it is written in Python, `music21` can tap into many other libraries, integrating internet resources (such as geomapping with Google Maps), visualization software, and sophisticated database searches with musical analysis.

This brief paper gives an overview of the `music21` toolkit. Through examples of musicological applications that the system enables, the main distinguishing features are illustrated: simplicity of use and expansion.

2. SCRIPTING AND OBJECTS

`Music21` is built in Python, a well-established programming language packaged with Macintosh and Unix computers and freely downloadable for Windows users. The toolkit adds a set of related libraries, providing sophisticated musical knowledge to Python. As shown in Figure 1, after adding the system with "`from music21 import *`", straightforward tasks such as displaying or playing a short melody, getting a twelve-tone matrix, or converting from Humdrum's Kern format to MusicXML can each be accomplished with a single line of code.

Display a simple melody in musical notation:

```
tinyNotation.TinyNotationStream(
      "c4 d8 f g16 a g f#", "3/4").show()
```

Print the twelve-tone matrix for a tone row (in this case the opening of Schoenberg's Fourth String Quartet):

```
print(serial.rowToMatrix(
          [2,1,9,10,5,3,4,0,8,7,6,11]) )
```

or since most of the 2nd-Viennese school rows are already available as objects, you could instead type:

```
print(serial.RowSchoenbergOp37().matrix() )
```

*Convert a file from Humdrum's **kern data format to MusicXML for editing in Finale or Sibelius:*

```
parse('/users/documents/composition.krn').
    write('xml')
```

Figure 1. Three simple examples of one-line `music21` scripts.

Though single-line tasks are simpler to accomplish in `music21` than in existing packages, the full power of the new toolkit comes from bringing together and extending high-level objects. The framework includes many objects, including Pitches, Chords, Durations, TimeSignatures, Intervals, Instruments, and standard Ornaments. Through method calls, objects can perform their own analyses and transformations. For instance, Chords can find their own roots, create closed-position versions of themselves, compute their Forte prime forms, and so on. Researchers can extend objects for their own needs, such as altering the pitch of open Violin strings to study *scordatura,* specializing (subclassing) the Note class into MensuralNote for studying Renaissance Music, or grouping Measures into Hypermeters. The object-oriented design of `music21` simplifies writing these extensions.

3. STREAMS: POWERFUL, NESTABLE, CONTAINERS OF TIMED ELEMENTS

At the core of `music21` is a novel grouping of musical information into Streams: nestable containers that allow researchers to quickly find simultaneous events, follow a voice, or represent instruments playing in different tempi and meters. Elements within Streams are accessed with methods such as `getElementById()`, an approach similarly to the Document Object Model (DOM) of retrieving elements from within XML and HTML documents. Like nearly every `music21` object, Streams can immediately be visually displayed in Lilypond or with programs that import MusicXML (such as Finale and Sibelius). Through the Stream model, a program can find notes or chords satisfying criteria that change from section to section of a piece, such as all notes that are the seventh-degree of the current key (as identified either manually or with an included key-detection algorithm) and then retrieve information such as the last-defined clef, dynamic, or metrical accent level at that point.

Many tools to visualize, process, and annotate Streams come with the `music21` framework. These tools include graphing modules, analytical tools, and convenience routines for metrical analysis [8], phrase extraction, and identification of non-harmonic tones. Figure 2 demonstrates the use of metrical analysis, derived from nested hierarchical subdivisions of the time signature [1], to annotate two Bach chorales in different meters.

```
from music21.analysis import metrical

# load a Bach Chorale from the music21 corpus of supplied pieces
bwv30_6 = corpus.parseWork('bach/bwv30.6.xml')

# get just the bass part using DOM-like method calls
bass = bwv30_6.getElementById('Bass')

# get measures 1 through 10
excerpt = bass.getMeasureRange(1,10)

# apply a Lerdahl/Jackendoff-style metrical analysis to the piece.
metrical.labelBeatDepth(excerpt)

# display measure 0 (pickup) to measure 6 in the default viewer
# (here Finale Reader 2009)
excerpt.show()
```

```
# perform the same process on a different chorale in 3/4 time
bwv11_6 = corpus.parseWork('bach/bwv11.6.xml')
alto = bwv11_6.getElementById('Alto')
excerpt = alto.getMeasureRange(13,20)
metrical.labelBeatDepth(excerpt)
excerpt.show()
```

Figure 2. Analytical tools, such as this metrical accent labeler, are included with `music21` and work with most Streams (including Scores and Parts). Here, excerpts of two Bach chorales, each in a different meter, are labeled with dots corresponding to their metric strengths.

4. FURTHER FEATURES

In addition to providing sophisticated resources in a modern programming language, the `music21` package takes advantage of some of the best contemporary approaches to software distribution, licensing, development, and documentation. These approaches assure both the longevity of the system across multiple platforms as well as the ability of the system to grow and incorporate the work of contributors.

4.1 An Integrated and Virtual Corpus of Music for Researchers

`Music21` comes with a corpus package, a large collection of freely-distributable music for analysis and testing, including a complete collection of the Bach Chorales, numerous Beethoven String Quartets, and examples of Renaissance polyphony. The virtual corpus extends the corpus package even further. Similar to a collection of URL bookmarks to music resources, additional repertories, available online, can be automatically downloaded when first requested and then made available to the researcher for future use. The corpus includes both Kern and MusicXML files. Future system expansions will not only grow the tools for analysis, but also the breadth and depth of the corpus of works.

4.2 Permissive License and Full Documentation

`Music21` is a toolkit: a collection of tools that work together in a wide range of contexts. The promise of a toolkit is only achieved if users can expand and integrate software components in their own work. Thus, `music21` is released under the Lesser General Public License (LGPL), allowing its use within both free and commercial software. So that implementation and documentation stay synchronized, the toolkit also features high-quality, indexed, and searchable documentation of all modules and classes, automatically created from the source code and test routines. The `music21` site (http://web.mit.edu/music21) hosts up-to-date information, documentation and release links. Code browsing, feature requests, and bug reports are housed at Google Code.

5. EXAMPLES

Better than an explanation of high-level features, a few specific examples illustrate the toolkit's promise. These examples are chosen for both their novel and practical utility.

5.1 Finding Chords within Melodic Fragments

The script in Figure 3 searches the entire second violin part of a MusicXML score of Beethoven's *Große Fuge*, op. 133, to find measures that melodically express dominant seventh chords in consecutive notes. It then displays the chord in closed position, the surrounding measure, and the Forte prime form. (Running the same query across barlines would add just a few lines of code).

```
op133 = corpus.parseWork(
            'beethoven/opus133.xml')
violin2 = op133.getElementById('2nd Violin')

#  an empty container for later display
display = stream.Stream()

for thisMeasure in violin2.measures:

  # get a list of consecutive notes, skipping unisons, octaves,
  # and rests (and putting nothing in their places)
  notes = thisMeasure.findConsecutiveNotes(
  skipUnisons = True, skipOctaves = True,
  skipRests = True, noNone = True)

  pitches = stream.Stream(notes).pitches
```

```
   for i in range(len(pitches) - 3):
      # makes every set of 4 notes into a whole-note chord
      testChord = chord.Chord(pitches[i:i+4])
      testChord.duration.type = "whole"

      if testChord.isDominantSeventh():
         # A dominant-seventh chord was found in this measure.

         # We label the chord with the measure number
         # and the first note of the measure with the Forte Prime form
         testChord.lyric = "m. " + str(
                   thisMeasure.measureNumber)
         primeForm = chord.Chord(
             thisMeasure.pitches).primeFormString
         firstNote = thisMeasure.notes[0]
         firstNote.lyric = primeForm

         # Then we append the chord in closed position followed
         # by the measure containing the chord.

         chordMeasure = stream.Measure()
         chordMeasure.append(
            testChord.closedPosition())
         display.append(chordMeasure)
         display.append(thisMeasure)
display.show()
```

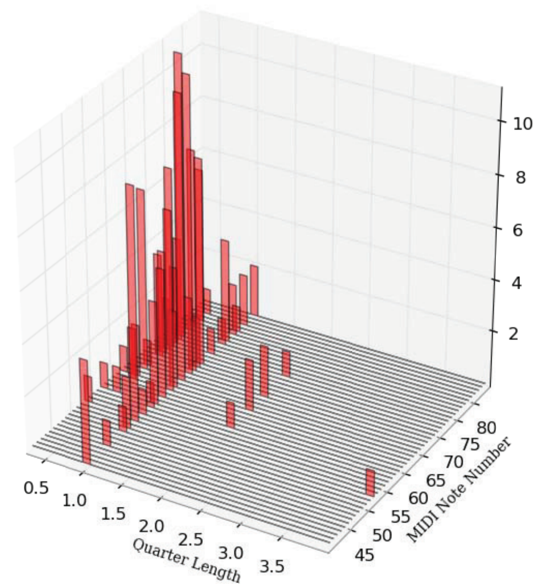
Figure 3. The results of a search for chords expressed melodically.

5.2 Distributions of Notes by Pitch and Duration

Figure 4 demonstrates the ability of `music21` graphs to help visualize trends that are otherwise difficult to discern. These graphs plot three features: pitch, duration of notes, and how frequently these pitches and durations are used. From two small excerpts of pieces in 3/4 by Mozart (a minuet, in red) and by Chopin (a mazurka, in blue), it can be seen that pitches in the Mozart example follow a type of bell-curve distribution, with few high notes, few low notes, and many notes toward the middle of the registral space. Chopin's usage jumps throughout the piano. The differences in pitch usage suggest that this line of inquiry is worth pursuing further, but no connection between duration and pitch appears. `Music21`'s easy-to-use graphing methods help researchers find the best visualization tool for their data, easily switching among diverse formats.

```
from music21.musicxml import testFiles as xml
from music21.humdrum import testFiles as kern

# display 3D graphs of count, pitch, and duration
mozartStream = music21.parse(
    xml.mozartTrioK581Excerpt)
notes = mozartStream.flat.stripTies()
g = graph.Plot3DBarsPitchSpaceQuarterLength(
    notes, colors=['r'])
g.process()
```




```
# perform the same process on a different work
chopinStream = music21.parse(kern.mazurka6)
notes = chopinStream.flat.stripTies()
g = graph.Plot3DBarsPitchSpaceQuarterLength(
    notes, colors=['b'])
g.process()
```

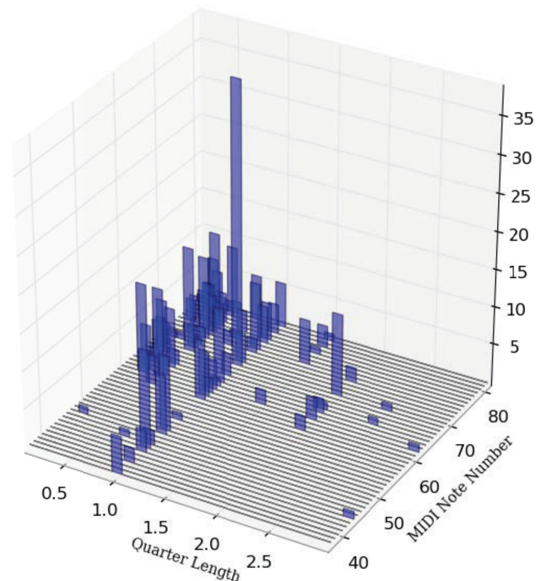


Figure 4. Differences in pitch distribution between Mozart and Chopin.

The Mozart and Chopin examples, while showing distinctive individual usage, show little correlation between pitch and duration. Many other pieces do. An extreme example is Messiaen's "Mode de valeurs et d'intensités" from *Quatre études de rythme*, perhaps the first work of total serialism. A perfect correlation between pitch and duration, as found in the middle voice (isolated for clarity), is plotted in Figure 5. An aspect of

the composition that is difficult to observe in the score but easy to see in this graph is the cubic shape ($-x^3$) made through the choice of pitches and rhythms. This shape is not at all explained by the serial method of the piece. Also easily seen is that, although Messiaen uses lower notes less often, there is not a perfect correlation between pitch and frequency of use (e.g., 21 B-flats vs. 22 A-flats).

```
messiaen = converter.parse(
               'd:/valeurs_part2.xml')
notes = messiaen.flat.stripTies()
g = graph.PlotScatterWeightedPitch\
    SpaceQuarterLength(notes, xLog = False,
    title='Messiaen, Mode de Valeurs,
    middle voice')
g.process()
```

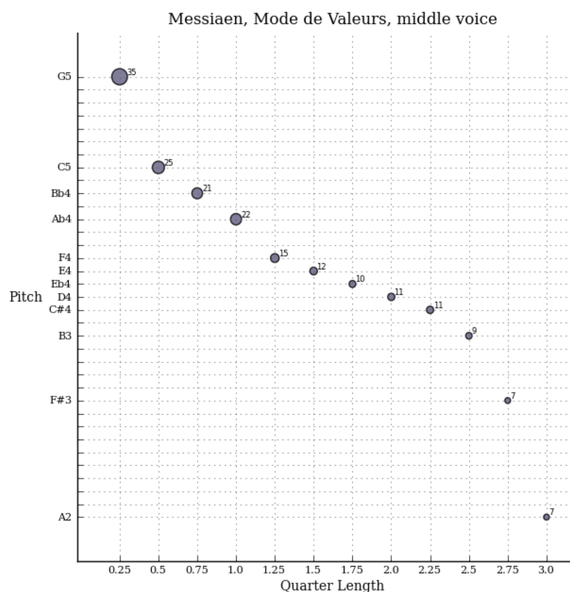


Figure 5. A graph of pitch to duration relationships in Messiaen, "Mode de valeurs," showing the correlation between the two note attributes.

5.3 Pitch Class Density Over Time

In Figure 6, pitch class usage, over the duration of the composition in the cello part of a MusicXML score of Beethoven's *Große Fuge*, is graphed. Even though the temporal resolution of this graph is coarse, it is clear that the part gets less chromatic towards the end of the work. (We have manually highlighted the tonic and dominant in this example.)

```
beethovenScore = corpus.parseWork('opus133.xml')
celloPart = \
      beethovenScore.getElementById('Cello')

# given a "flat" view of the stream, with nested information
# removed and all information at the same hierarchical level,
# combine tied notes into single notes with summed durations
notes = celloPart.flat.stripTies()

g = graph.PlotScatterPitchClassOffset(notes,
    title='Beethoven Opus 133, Cello')
g.process()
```

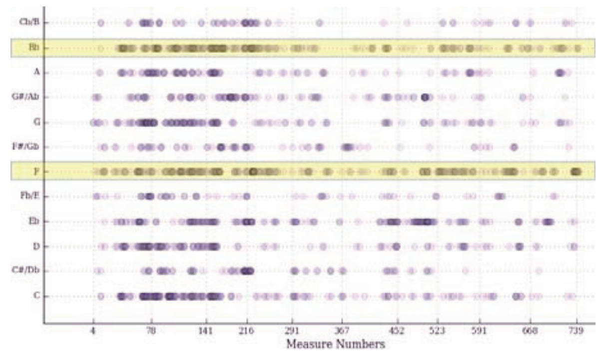


Figure 6. A graph of pitch class usage over time in Beethoven's *Große Fuge*.

5.4 Testing Nicolaus de Capua's *Regulae* of *Musica Ficta*

This example shows a sophisticated, musicological application of `music21`. Among his other writings, the early-fifteenth century music theorist Nicolaus of Capua gave a set of *regulae*, or rules, for creating *musica ficta* [5]. *Musica ficta*, simply put, was a way of avoiding tritones and other undesirable intervals and create more conclusive cadences through the use of unwritten accidentals that performers would know to sing. Unlike the rules of most other theorists of his time, Nicolaus's four rules rely solely on the melodic intervals of one voice. Herlinger's study of Nicolaus's rules suggested that they could be extremely successful at eliminating harmonic problems while at the same time being easy enough for any musician to master. However, as is conventional in musicology, this study was performed by hand on a few excerpts of music by a single composer, Antonio Zachara da Teramo. Using `music21` we have been able to automatically apply Nicolaus's rules to a much wider set of encoded music, the complete incipits and cadences of all Trecento ballate (about 10,000 measures worth of music) and then automatically evaluate the quality of harmonic changes implied by these rules. Figure 7 shows an excerpt of the code for a single rule, that a descending major second ("M-2") immediately followed by an ascending major second ("M2") should be transformed into two half-steps by raising the middle note:

```
# n1, n2, and n3 are three consecutive notes
# i1 is the interval between n1 and n2
# i2 is the interval between n2 and n3

i1 = generateInterval(n1,n2)
i2 = generateInterval(n2,n3)

# we test if the two intervals are the ones fitting the rule
if i1.directedName == "M-2" and \
   i2.directedName == "M2":

  # since the intervals match , we add an editorial accidental
  n2.editorial.ficta = \
         Accidental("sharp")
```

```
# we also color the affected notes so that if we display the music
# the notes stick out.  Different colors indicate different rules
n1.editorial.color = "blue"
n2.editorial.color = "forestGreen"
n3.editorial.color = "blue"
```

Figure 7. Applying ficta accidentals with `music21`.

The results of applying one or all the rules to an individual cadence or piece can be seen immediately. Figure 8 shows the rules applied to one piece where they create two "closest-approaches" to perfect consonances (major sixth to octave and minor third to unison). These are the outcomes one expects from a good set of *regulae* for *musica ficta*.

```
# get a particular worksheet of an Excel spreadsheet
ballataObj = cadencebook.BallataSheet()
# create an object for row 267
pieceObj = ballataObj.makeWork(267)
# run the four rules (as described above)
applyCapua(pieceObj)
# display the first cadence of the piece (second snippet) by
# running it through Lilypond and generating a PNG file
pieceObj.snippets[1].lily.showPNG()
```

Figure 8. Music21 code for automatically adding *musica ficta* to Francesco (Landini), *De[h], pon' quest'amor,* first cadence.

In other pieces, Nicolaus's rules have an injurious effect, as Figure 9 shows. With the toolkit, we were able to run the rules on the entire database of Trecento ballatas and determine that Nicolaus's rules cannot be used indiscriminately. Far too many cases appeared where the proposed *ficta* hurt the harmony. One of the main advantages of the `music21` framework is making such observations on large collections of musical data possible.

Figure 9. Francesco, *D'amor mi biasmo,* incipit after automatically applying *ficta* accidentals.

6. FUTURE WORK

The first alpha releases of `music21` introduce fundamental objects and containers and, as shown above, offer powerful tools for symbolic music processing.

The next stage of development will add native support for additional input and output formats, including MIDI. Further, libraries of additional processing, analysis, visualization routines, as well as new and expanded object models (such as non-Western scales), will be added to the system. We are presently focusing on creating simple solutions for common-practice music theory tasks via short `music21` scripts, and within a year hope to be able to solve almost every common music theory problem encountered by first-year conservatory students.

7. ACKNOWLEDGEMENTS

The authors thank the Seaver Institute for their generous funding of `music21`. Additional thanks are also extended to three anonymous reviewers for their helpful comments.

8. REFERENCES

[1] Ariza, C. and M. Cuthbert. 2010. "Modeling Beats, Accents, Beams, and Time Signatures Hierarchically with music21 Meter Objects." In *Proceedings of the International Computer Music Conference.* San Francisco: International Computer Music Association.

[2] Buxton, W. and W. Reeves, R. Baecker, L. Mezei. 1978. "The Use of Hierarchy and Instance in a Data Structure for Computer Music." *Computer Music Journal* 2 (4): 10-20.

[3] Cook, N. 2004. "Computational and Comparative Musicology." In *Empirical Musicology: Aims, Methods, Prospects*. N. Cook and E. Clarke, eds. New York: Oxford University Press. 103-126.

[4] Good, M. 2001. "An Internet-Friendly Format for Sheet Music." In *Proceedings of XML 2001*.

[5] Herlinger, J. 2004. "Nicolaus de Capua, Antonio Zacara da Teramo, and *musica ficta."* In *Antonio Zacara da Teramo e il suo tempo.* F. Zimei, ed. Lucca: LIM. 67–89.

[6] Huron, D. 1997. "Humdrum and Kern: Selective Feature Encoding." In *Beyond MIDI: the Handbook of Musical Codes*. E. Selfridge-Field, ed. Cambridge: MIT Press. 375-401.

[7] Knopke, I. 2008. "The PerlHumdrum and PerlLilypond Toolkits for Symbolic Music Information Retrieval." *ISMIR 2008* 147-152.

[8] Lerdahl, F. and R. Jackendoff. 1983. *A Generative Theory of Tonal Music*. Cambridge: MIT Press.

[9] Pope, S. T. 1987. "A Smalltalk-80-based Music Toolkit." In *Proceedings of the International Computer Music Conference*. San Francisco: International Computer Music Association. 166-173.

[10] Pope, S. T. 1989. "Machine Tongues XI: Object-Oriented Software Design." *Computer Music Journal* 13 (2): 9-22.

[11] Sapp, C. S. 2008. "Museinfo: Musical Information Programming in C++." Internet: http://museinfo.sapp.org.