Interpreting the semantics of music notation using an extensible and object-oriented system

Michael Droettboom, Ichiro Fujinaga
Peabody Conservatory of Music
Johns Hopkins University
Baltimore, MD

Abstract

This research builds on prior work in Adaptive Optical Music Recognition (AOMR) (Fujinaga 1996) to create a system for the interpretation of musical semantics. The existing AOMR system is highly extensible, allowing the end user to add new symbols and types of notations at run-time. It was therefore imperative that the new system have the ability to easily define the musical *semantics* of those new symbols. Python proved to be an effective tool with which to build an elegant and extensible system for the interpretation of musical semantics.

1 Introduction

In recent years, the availability of large online databases of text have changed the face of scholarly research. While those same collections are beginning to add multimedia content, content-based retrieval on such non-textual data is significantly behind that of text. In the case of music notation, captured images of scores are not sufficient to perform musically meaningful searches and analyses on the musical content itself. For instance, an end user may wish to find a work containing a particular melody, or a musicologist may wish to perform a statistical analysis of a particular body of work. Such operations require a logical representation of the musical meaning of the score. However, to date, creating those logical representations has been very expensive. Common methods of input include manually entering data in a machine-readable format (Huron and Selfridge-Field 1994) or hiring musicians to play scores on MIDI keyboards (Selfridge-Field 1993). Optical music recognition (OMR) technology promises to accelerate this process by automatically interpreting the musical content from the printed score.

Academic research in OMR is currently quite active, most notably by David Bainbridge (Bainbridge 1997), Nicholas Carter (Carter 1989), and Ichiro Fujinaga (Fujinaga 1996). Since modern Western music notation is over 350 years old, and has evolved significantly during that time (Read 1969), the most successful OMR systems are those that are easily adaptable to different types of input. Differences in music notation can occur both at the symbolic (Figure 1) and semantic (Figure 2) levels. Inflexibility to such differences is the primary drawback of commercial OMR products, such as MIDISCAN (Musitek 2000) and Neuratron Photoscore (Neuratron 2000). Musicians and musicologists who work with unusual notations, such as early or contemporary music, or physically damaged scores, such as those found in many historical sheet music collections, will have a hard time with a non-adaptive OMR system.

1.1 The Lester S. Levy Collection of Sheet Music

The present system is being developed as part of a larger project to digitize the Lester S. Levy Collection of Sheet Music (Milton S. Eisenhower Library, Johns Hopkins University).

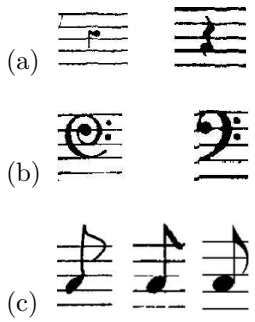


Figure 1: The appearance of individual musical symbols can vary quite dramatically. (a) two versions of typeset quarter rests; (b) two versions of typeset bass clefs; (c) handwritten, hand-engraved and digitally typeset eighth notes

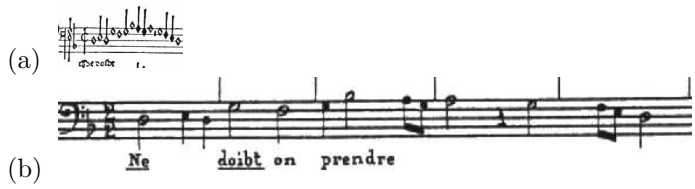


Figure 2: An excerpt from "No Doibt" by Loyset Compere in (a) its original white mensural notation and (b) the equivalent in contemporary typeset notation.

The Levy Collection consists of over 29,000 pieces of popular American music. While the Collection spans the years 1780 to 1960, its strength lies within its thorough documentation of nineteenth and early twentieth-century America.

Phase One of the digitization project, now nearing completion, involves capturing the images of the music in the collection and cataloging them with metadata such as author, title and date. The portions of the collection in the public domain are available to the general public at

`http://levysheetmusic.mse.jhu.edu`

Phase Two of the project involves using OMR to derive the musical semantics from the score images. The OMR system being developed for this purpose must be highly flexible and extensible to deal with the diversity of the collection.

2 Adaptive optical music recognition

For the purposes of our system, the problem of optical music recognition is divided into two subproblems: the classification of the symbols on the page and the interpretation of the musical semantics of those symbols. The first subproblem has been thoroughly explored and implemented by Ichiro Fujinaga as the Adaptive Optical Music Recognition (AOMR) system, summarized in this section. The second subproblem builds on this work and is the subject of the remainder of this paper.

The AOMR system offers five important advantages over similar commercial offerings. First, it can be run in batch processing mode, an essential feature for large musical collections. It is important to note that most commercial software is intended for the casual

user and does not scale for a large number of objects. Second, the software is written in C and therefore is portable across platforms[1]. Third, the software can "learn" to recognize different music symbols, an serious issue considering the diversity of the common music notation. Fourth, the software is open-sourced. Finally, this software can separate full-text lyrics that can be further processed using third-party optical character recognition (OCR) software. Preliminary attempts at using the existing OMR system for OCR also show some promise.

Using vertical run-length coding and projection analysis the staff lines are removed from the input image file. Lyrics are also removed using various heuristic rules. The music symbols are then segmented using connected-component analysis. A set of features, such as width, height, area, number of holes, and low-order central moments, is stored for each segmented graphic object and used as the basis for the adaptive recognition system based on examples.

The exemplar-based classification model is based on the idea that objects are categorized by their similarity to stored examples. The model can be implemented by the k-nearest-neighbor (k-NN) algorithm (Cover and Hart 1967), which is a classification scheme to determine the class of a given sample by its feature vector. Distances between feature vectors of an unclassified sample and previously classified samples are calculated. The class represented by the closest neighbor is then assigned to the unclassified sample. Besides its simplicity and intuitive appeal, the classifier can be easily modified, by continually adding new samples that it "encounters" into the database, to become an adaptive system (Aha 1997). In fact, "the nearest neighbor algorithm is one of the simplest learning methods known, and yet no other algorithm has been shown to outperform it consistently" (Cost and Salzberg 1992). Furthermore, the performance of the classifier can be dramatically increased by using weighted feature vectors. Finding a good set of weights, however, is extremely time-consuming, thus a genetic algorithm (Holland 1975) is used to find a solution (Wettschereck, Aha, and Mohri 1997). Note that the genetic algorithm can be run off-line without affecting the speed of the recognition process.

3 Optical music interpretation

In general, Optical Music Interpretation (OMI) involves identifying the connections between symbols and examining their relative positions. From this information, the semantics of the score (eg. the pitches and durations of notes) can be derived.

3.1 Background

A number of past approaches to OMI have used two-dimensional graph grammars as the central problem-solving mechanism (Fahmy and Blostein 1993) (Couasnon and Camillerapp 1994) (Baumann 1995). While (Fahmy and Blostein 1993) is relatively successful, it is unnecessarily complicated by the false assumption that relationships between musical symbols can occur in any arbitrary direction. The added complexity of this approach both decreases maintainability and increases algorithmic complexity. Common music notation, even when polyphonic, is essentially a one-dimensional stream that has a well-defined ordering in time: It is never the case that musical objects in the future will affect objects in the past. This property can be exploited by sorting all the objects by their temporal order into a one-dimensional list. In fact, in the present system, all objects are stored in a simple Python list during interpretation. Once sorted, all necessary operations for interpretation can be performed on the objects quite conveniently. Any errors in the ordering of symbols, cited as an major difficulty in OMI, in fact tend to be quite local and simple to resolve. Therefore, while one-dimensional grammars, such as those used in natural language processing, are

[1] AOMR has been ported to GNU/Linux on x86 and PPC, Sun Solaris, SGI IRIX, NeXTSTEP, Macintosh OS-X and Windows 95/98/NT/2000.

potentially useful, I assert that graph grammars are unnecessarily complex for the problem and therefore are not used in the present implementation.

Another approach to OMI is represented by the underlying data structure of a research-oriented music notation application, *Nutator* (Diener 1989). Its TTREES (temporal trees) are general data structures used to group objects in physical space and time. Each symbol in a score is composed of a type name, an (x, y) coordinate and a z ordering. Collectively, this object is referred to as a *glyph*. Glyphs exist in a two-and-a-half dimensional space and thus can be stacked on top of each other. This stacking implicitly defines relationships between glyphs. Glyphs in the foreground communicate with glyphs in the background in order to determine their semantics. For instance, a note would determine its pitch by communicating with the set of staff lines underneath it and the clef underneath and to the left. This paradigm of communication between glyphs is used heavily throughout the present system. The advantage of this approach is that glyphs can be edited at run-time and the semantic results of those changes can be determined very efficiently.

3.2 Design criteria

The goals of the present OMI implementation are consistent with those of the underlying AOMR system. The primary objectives are automatability (batch processing), portability and extensibility. Python was chosen as the implementation language partly because of the ease with which it can be used to meet all of these goals.

- **Automatability:** Python's simple scripting features make it easy to customize the workflow for different batch processing needs. In addition, OMI can be completely driven from the command line.
- **Portability:** Since all of the input and output formats of OMI are in clear text, the OMI system is portable to any platform for which there is a Python interpreter.
- **Extensibility:** Python's flexible object-oriented paradigm allows for the semantics of new symbols to be easily added to the system using inheritance. The exact definition of those symbols can be refined interactively without a separate compile step.

4 Python implementation issues

4.1 Overview

The Optical Music Interpreter (OMI) system is implemented entirely in Python 1.5.2. It is open-sourced under the GPL and available at:

`http://mambo.peabody.jhu.edu/omr/`

The overall execution of OMI proceeds linearly through the following phases:

1. **Input.** The bounding boxes are read in from AOMR. (Section 4.2).
2. **Staff assignment and temporal sorting.** Each glyph is assigned to a set of staff lines and put in temporal order. (Section 4.3).
3. **Reference assignment.** Glyphs are assigned references to other related glyphs. (Section 4.4).
4. **Metric correction.** Errors in the OMR stage are corrected by examining the metrical content and physical alignment of glyphs. (Section 4.5).
5. **Output.** The logical representation is output. Attributes of the individual glyphs are determined on-the-fly based on the references made in the reference assignment phase. (Section 4.6).

This section will discuss each phase in turn, highlighting the issues of interest to Python programmers in general. More detailed implementation documentation is available online.

4.2 Input

There are two kinds of output from Ichiro Fujinaga's AOMR used by OMI.

- The first is a list describing the musical symbols on the page. Each symbol entry contains a string defining its type, a rectangular bounding box relative to the page, and a hotspot coordinate.
- The second is a list of bounding boxes around the individual pieces of text in the image. The small images in these bounding boxes are extracted using the Python Imaging Library (PIL) and sent to a third-party Optical Character Recognition (OCR) system[2]. The results of the OCR are merged with the original text bounding boxes and sent back to OMI as glyphs.

These glyphs are interpreted by OMI, which then outputs a musical description. To avoid the necessity of writing a custom parser for these input files, the files themselves are formatted as a Python list. Reading all this in then takes only four lines of code:

```
fd = open(filename, 'r') # Open the file
input = fd.read()        # Read its contents into a string
fd.close()               # Close the file
data = eval(input)       # Convert the string to a list of
                         # elements by evaluating it as
                         # Python code
```

Each entry in the input list contains a string defining the type of symbol. These are converted into actual object instances by taking advantage of Python's ability to turn strings into code on the fly.

```
for element in data:
  # This is the name of the class
  name = element[0]
  # If the string is in fact the name of a class we
  # can create an instance...
  if name in glyph.__dict__.keys():
    create = "glyph." + name
  # ...otherwise create a default class instance
  else:
    create = "glyph.DEFAULT"
  # create the new instance using apply.
  # the new object is init. to the correct bounding box
  ng = apply(eval(create), (name,
              element[4],
              ((element[7]) - (PAGE_HEIGHT)) * -1,
              element[6],
              ((element[5]) - (PAGE_HEIGHT)) * -1,
              element[8],
              ((element[9] - (PAGE_HEIGHT)) * -1)))
  # add the new element to the list
  glyph_list.append(ng)
```

The advantage of this approach is that new classes of symbols can be added to the system merely by writing a new Python class in the appropriate module. There is no need

[2]To date, we have experimented with gocr (Schulenberg 2000) and Pixel/FX! 2000 (Mentalix 2000), but using AOMR itself for OCR has also shown some promise.

Figure 3: A dotted quarter note. The dot increases the duration of the quarter note by 50%.

to explicitly register the new class in a prototype database in order for the parser to handle it (Gamma, Helm, Johnson, and Vlissides 1995).

4.3 Staff assignment and temporal sorting

This phase is concerned with sorting the glyphs into a musically meaningful order. An example of this temporal ordering is shown in Figures 4 and 5.

Staves serve to put things in temporal order, in much the same way that a line of text is read from left to right. Therefore, before the glyphs can be sorted, they must be assigned to a particular staff. The glyphs are then sorted first by part, then voice, and then staff. Next, the glyphs are sorted in temporal order from left to right. Finally, glyphs that occur at the same vertical position are sorted top to bottom. This multi-level sorting is performed in one step using Python's built-in quick sort function using a custom ordinal function.

For efficient retrieval of information in the glyph list, it is indexed by class. It is then trivial to retreive all of the glyphs of a certain class from the score and ignore all others.

4.4 Reference assignment

The purpose of this phase is to build the relationships between glyphs necessary to fully derive their musical meaning. Most of the relationship algorithms are simple iterations over the glyph list (or indexed subsets of that list) and therefore run in linear time.

An interesting use of Python's object-oriented paradigm is the way in which run-time type inspection can used to keep the reference-building algorithms as abstract as possible. To support this, the glyph classes are all part of a complex multi-tiered hierarchy (Figure 6). The concrete classes correspond directly to the physical glyphs on the score retreived by AOMR. Then, using an object-oriented style popular in Eiffel (Meyer 1997) and the Java Foundation Classes (Sun Microsystems 1998), most of the abstract base classes are named using adjectives describing their ability (eg. `DOTTABLE`, `ARTICULATABLE`, `PITCHED`). Not only does this improve the readibility of the code, it also allows the algorithms to choose to operate on specific sets of glyphs based on their high-level abilities rather than their low-level identity.

For example, all classes that can have their duration augmented by a dot inherit from the `DOTTABLE` class (Figure 3). This includes both notes and rests. The algorithm that assigns dots to `DOTTABLE`s can simply use the expression

$$x\texttt{.isinstance(DOTTABLE)}$$

to determine if glyph x can be dotted. It does not need to know that in fact all notes and rests can be dotted, since this is already implied by the class heirarchy.

Such a paradigm is difficult to implement in more static languages such as ANSI C++ that do not have complete run-time type inspection.

4.5 Metric correction

Occasionally, noise in the input image can cause errors at the recognition stage. Missing or erroneous glyphs cause voices to have too few or too many beats per measure. Fortunately, many of these errors can be corrected by exploiting a common feature of typeset music:

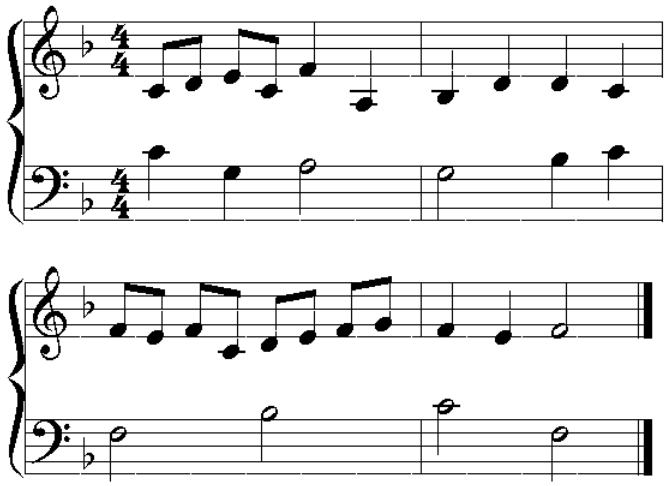


Figure 4: An example duet to demonstrate temporal ordering

PART 0
    STAFF 0 (implied system break)
        treble clef
        flat
        4/4
        c d e c f a
        barline
        b♭ d d c
        barline
    STAFF 2 (implied system break)
        treble clef
        flat
        f e f c d e f g
        barline
        f e f
        final barline
PART 1
    STAFF 1 (implied system break)
        bass clef
        etc.
    STAFF 3 (implied system break)
        etc.

Figure 5: A simplified listing of the example score in Figure 4 showing temporal ordering. Note that staff glyphs double as implied system breaks and the entire contents of the treble clef part appear before the bass clef part.

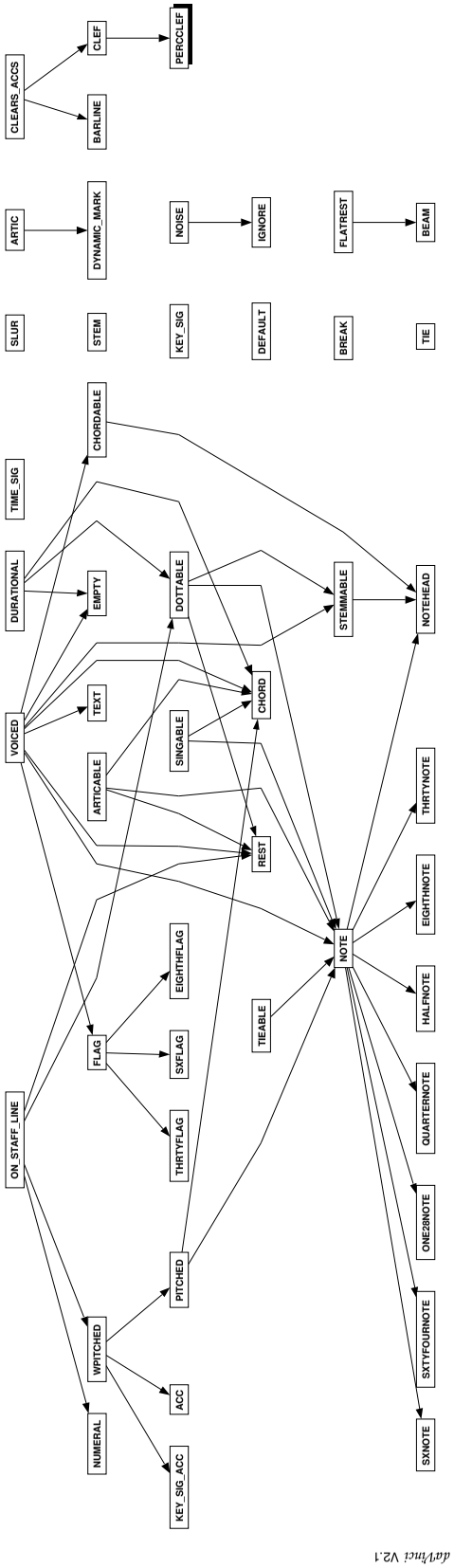


Figure 6: The main glyph class heirarchy. For simplicity, concrete classes and the top-level `BASE` class have been removed from the graph.

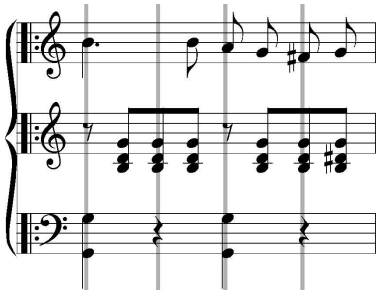


Figure 7: Each gray bar represents a quarter note beat. All notes occurring on that beat are vertically aligned.

To ease sight-reading, notes that occur at the same time are aligned vertically within each system (set of staves) of music[3] (Figure 7). OMI provides a number of algorithms to correct durational information based on this feature of music. They include:

1. **Bar rest:** Fixes measures with a single half or whole rest.
2. **Whole/half:** Converts whole rests and half rests and vice versa.
3. **Bad dot:** Removes erroneous augmentation dots.
4. **Barline to stem:** Converts stems that were misinterpreted as barlines back to stems.
5. **Splice:** Cuts long measures into shorter ones.
6. **Rebuild:** Changes durations of notes so that they correspond with vertically aligned notes in other parts.
7. **Extend:** Adds dummy space to the end of measures so that the total length across all parts is equal.

These algorithms are described in more detail online.

4.6 Output

One of the long-standing difficulties in the field of music information retrieval is the lack of a standard representation format for logical musical data (Selfridge-Field 1997). It is therefore necessary for the present system to support different output formats for different needs. Relying on external converters, as many word processors do, is not ideal since many musical representation formats have radically different ordinal structures and scope. Instead, OMI uses pluggable back-ends that map from OMI's internal data structure, a list of glyphs, to a given output file format. Presently, output to GUIDO (Hoos and Hamel 1997) is implemented, but other musical representation languages such as Lilypond Mudela (Nienhuys and Nieuwenhuizen 1998) are planned. MIDI (MIDI Manufacturers Association Inc. 1986) is currently supported through a third party tool that converts GUIDO to MIDI (Martin and Hoos 1997).

In general, output is generated in two phases. First, the pluggable back-end is given a chance to reorder the glyph list. This is useful since the ordering of objects differs across formats. For example, GUIDO files are organized part by part, whereas Type 0 MIDI files interleave the parts together by absolute time. After re-ordering, the output function of each glyph is called. The output functions are implemented in the pluggable back-end and

[3]Some older scores in the Levy Collection do not have this property. In this case, metric correction can be turned off.

Core class in `glyph.py`

```
class CLEF(CLEARS_ACCS, BASE):
  middle_line = B
  octave = 1
  key_sig = None
  fixed_guido_name = "treble"

  def __repr__(self):
    # etc...

  def get_wpitch(self, staff_line):
    # etc...

  def get_octave(self, staff_line):
    # etc...
```

Extension class in `guido.py`

```
class CLEF(GUIDO):
  def bas_guido_clef(self):
    # etc...

  def bas_guido(self):
    # etc...
```

Figure 8: The core functionality of the `CLEF` class is implemented in the class on the left, and the extensions that allow GUIDO output are implemented in the class on the right. The extension class' members are merged into the core class by class augmentation.

merged into the core glyph classes using a technique, *class augmentation*, that exploits the run-time flexibility of Python classes.

Class augmentation adds members to a core class from an extension class loaded at run-time. The matching of core class to extension class is determined by their names (i.e. `__name__` member). For a concrete example, consider the class definitions in Figure 8. The class augmentation procedure will add the GUIDO-specific functions in the extension class to the core class, since they both have the same name (`CLEF`). The merging itself is achieved by adding the extension class to the front of the tuple of base classes (i.e. `__bases__` member) of the core class. This serves to put the extension's members on the core class' search path.

The augmentation is performed on all the classes in a given module, so it is easy to extend large numbers of classes with one function call. The following function takes two modules as inputs and augments all of the classes in `core` for which there exists an extension class in `ext`.

```
def merge_classes(core, ext):
  # Dictionary to map the names of classes to classes
  klasses = {}

  # Fill the dictionary with all classes in core module
  for klass in vars(core).values():
    if type(klass) == types.ClassType:
      klasses[klass.__name__] = klass

  # Add extension classes to the __bases__ tuple of the
  # core classes whenever there is a match by name
  for klass in ext.values():
    if (type(klass) == types.ClassType and
        klass.__name__ in klasses.keys()):
      (klasses[klass.__name__].__bases__ =
       tuple([klass] + list(klasses[klass.__name__].__bases__)))
```

The primary advantage to this approach is its robust handling of extensibility: when new classes are added to the core heirarchy, they do not to be updated in all output extension modules.

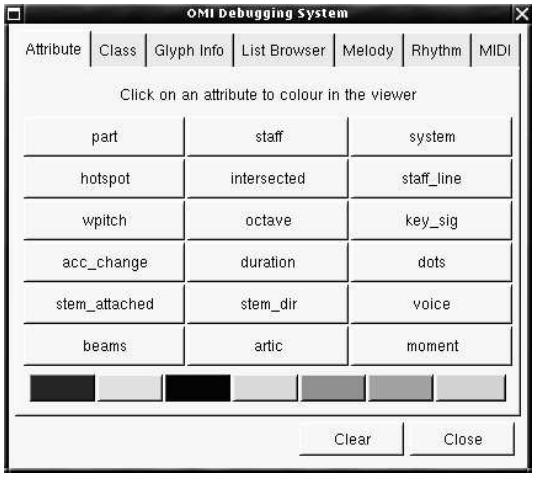
4.7 Graphical interactive self-debugger

Allowing the user to interact with the data of a running program is one of Python's greatest assets, and greatly reduces the length of the develop-test cycle (Lutz 1996). However, interacting with graphical data, such as that in OMI, is quite cumbersome using only text-based tools. For example, selecting (x, y) coordinates with a mouse is much easier than entering them numerically. For this reason, a graphical, interactive debugger was implemented that allows the programmer to examine the data structures of a running OMI session and execute arbitrary Python code upon it. This is analogous to running Python in interactive mode, except that it offers a graphical way of interacting with positional data.

The overall debugging system is divided between an image viewer and OMI debugging system itself. The source image is displayed using the custom **focus** image viewer, implemented in C++ using the Gtk−− toolkit. Besides providing the basic functionality of scaling and displaying the image, the viewer also accepts messages over a socket to colorize or draw rectangles on arbitrary parts of the image. On the Python side, a simple GUI implemented using the Python-Gtk+ bindings communicates with the viewer to allow the user to display or modify the logical data in different ways. To support the coloring of objects, each glyph has a `color` function that sends a message to the viewer regarding its position. In addition, the `__repr__` function of each glyph serves to both (a) return a text dump of all its pertinent data members in a human readable form and (b) call its `color` function so it will be highlighted in the viewer.

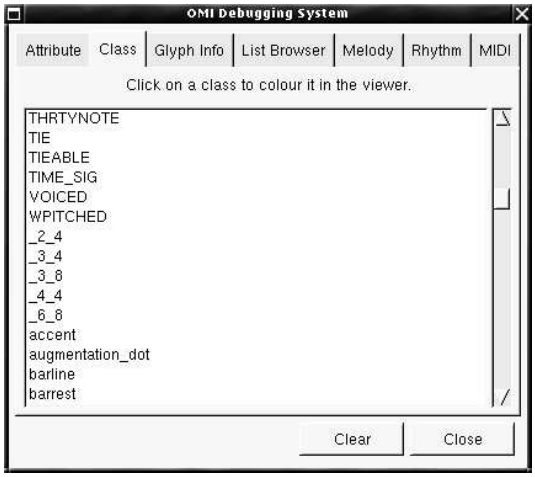
The GUI uses a notebook interface to divide the functionality into different pages. The pages pertaining to debugging are described below.

4.7.1 Attribute page



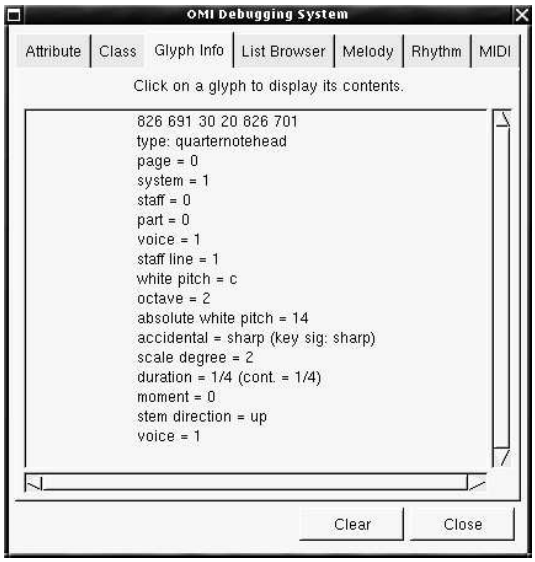
Each button on the attribute page colors the score based on different criteria. For example, the wpitch ("white pitch") button will color each notehead based on its note name (i.e. all a's will be red, all b's will be yellow etc.) While not particularly useful to an end user, coloring is an efficient way for the developer to debug an algorithm and ensure that it is producing the correct results.

4.7.2 Class browser page



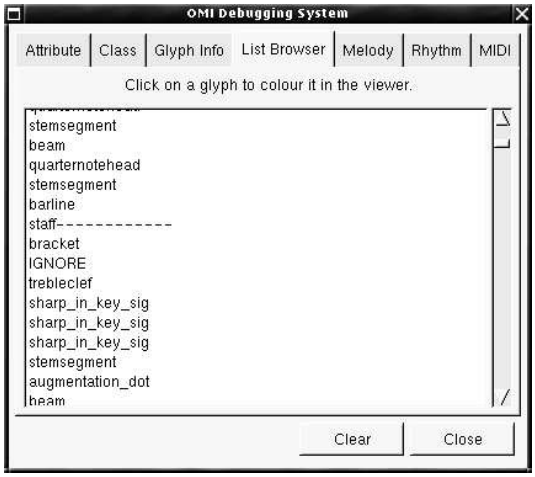
The class browser page displays a list of all classes in the glyph heirarchy (Figure 6). Clicking on an entry highlights all glyphs of that class in the viewer.

4.7.3 Glyph info page



Clicking on a glyph in the viewer displays all of its data members on the glyph info page. The text displayed on the glyph info page is taken directly from the output of the glyph's `__repr__` function.

4.7.4 List browser page



This page displays a list of all glyphs in the score in their temporal order. This page helps to debug the overall sorting algorithm, as well as any algorithms that rely on the relative position of glyphs within the glyph list. Clicking on an entry in the list highlights that glyph in the viewer.

4.7.5 Python console page

The Python page provides a console with an interactive Python session. Useful variables are defined in local scope, such as the glyph list, so that the developer can directly manipulate the data and see the results immediately. Printing out a glyph object (i.e. by typing the variable name and pressing Enter) displays its data members in the console and colorizes it in the viewer.

4.7.6 MIDI page

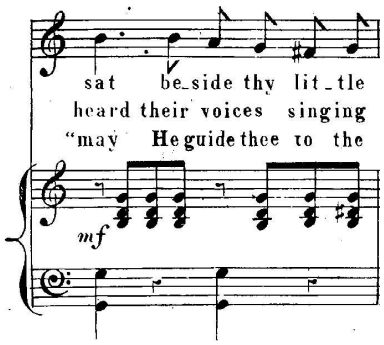
The MIDI page converts the GUIDO output of OMI to MIDI using the third-party tool **gmn2midi**. The result is then played using a user-defined MIDI file player.

5 Demonstration

This section demonstrates how a single measure from a score in the Levy Collection is converted into a number of different formats by the AOMR/OMI system.

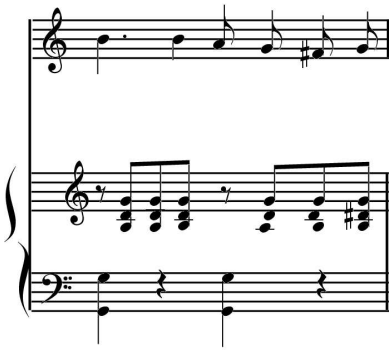
5.1 Original image

The original image was scanned at 300 DPI, 8-bit grayscale. Note that there is a fair amount of noise due to age of the score.



5.2 PostScript output

The PostScript output is an exact one-to-one copy of the recognized symbols on the page, recreated using PostScript primitives and the Adobe Sonata font. This phase is analogous to the glyph list that forms the bridge between AOMR and OMI.



5.3 GUIDO output

This is the logical interpretation of the score in GUIDO format (Hoos and Hamel 1997). Note that the format is human readable and fairly intuitive.

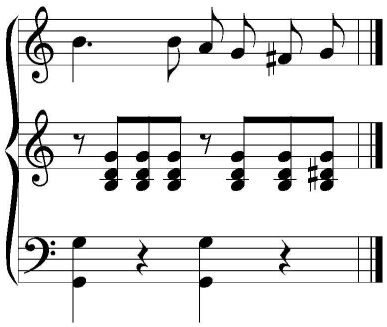
```
% GUIDO Music Notation format.
% Automatically generated from a scanned image.
{ [ \beamsOff \clef<"treble"> \key<0>
b1*1/4. b1*1/8 a1*1/8 g1*1/8 f#1*1/8 g1*1/8 |
],
[ \beamsOff \clef<"treble"> \key<0>
_*1/8 \beam( { b0*1/8 , d1*1/8 , g1*1/8 }
             { b0*1/8 , d1*1/8 , g1*1/8 }
             { b0*1/8 , d1*1/8 , g1*1/8 } )
_*1/8 \beam( { b0*1/8 , d1*1/8 , g1*1/8 }
             { b0*1/8 , d1*1/8 , g1*1/8 }
             { b0*1/8 , d#1*1/8 , g1*1/8 } ) |
],
[ \beamsOff \clef<"bass"> \key<0>
{ g-1*1/4 , g0*1/4 } _*1/4 { g-1*1/4 , g0*1/4 } _*1/4 |
] }
```

5.4 Re-rendered notation

The following is the output of OMI re-rendered using the GUIDO NoteServer available on-line at

http://tempo.iti.informatik.tu-darmstadt.de/salieri/nview/noteserver.html

The exact positions of the notes are determined solely from the logical representation of the score. Since GUIDO NoteServer aims to have some sense of musical intelligence, it added the implied double barlines at the end of the score, even though they were not specified in the GUIDO input file.



6 Conclusions

The present system handles with many of the inherent difficulties of optical music interpretation in an elegant and simple way. This elegance is in so small part due to its implementation in Python, which made easy work of the three main design criteria: automatability, portability and extensibility. However, as Phase Two of the Levy project gets under way, that project should provide a valuable in-house test bed to suggest improvements and refinements the system. Due to its solid foundation in a flexible object-oriented language, these changes should remain relatively simple to implement, keeping development time to a minimum. Ultimately, we hope other large sheet music digitization projects will use the system because it presents a flexible and extensible alternative to closed systems.

7 Acknowledgments

This work was conducted as part of the Lester S. Levy Collection of Sheet Music of the Digital Knowledge Center, Milton S. Eisenhower Library of the Johns Hopkins University. Funding was provided in part by the National Science Foundation, the Institute for Museum and Library Services, and the Levy family.

References

Aha, D. W. (1997). Lazy learning. *Aritificial Intelligence Review 11*(1), 7–10.

Bainbridge, D. (1997). *Extensible optical music recognition*. Ph. D. thesis, University of Canterbury.

Baumann, S. (1995). A simplified attributed graph grammar for high-level music recognition. In *International Conference on Document Analysis and Recognition*.

Carter, N. (1989). *Automatic Recognition of Printed Music in the Context of Electronic Publishing*. Ph. D. thesis, University of Surrey.

Cost, S. and S. Salzberg (1992). A weighted nearest neighbor algorithm for learning with symbolic features. *Machine Learning* (10).

Couasnon, B. and J. Camillerapp (1994). Using grammars to segment and recognize music scores. In *International Association for Pattern Recognition Workshop on Document Analysis Systems*.

Cover, T. and P. Hart (1967). Nearest neighbour pattern classification. *IEEE Transactions on Information Theory 13*(1), 21–7.

Diener, G. (1989). TTREES: A tool for the compositional environment. *Computer Music Journal 13*(2), 77–85.

Fahmy, H. and D. Blostein (1993). A graph grammar programming style for recognition of music notation. *Machine Vision and Applications*.

Fujinaga, I. (1996). *Adaptive Optical Music Recognition*. Ph. D. thesis, McGill University.

Gamma, E., R. Helm, R. Johnson, and J. Vlissides (1995). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley.

Holland, J. H. (1975). *Adaptation in natural and artificial systems*. Ann Arbor: University of Michigan Press.

Hoos, H. H. and K. Hamel (1997). Guido music notation version 1.0: Specification part i, basic guido. Technical Report TI 20/97, Technische Universität Darmstadt. http://www.informatik.tu-darmstadt.de/AFS/GUIDO/docu/spec1.htm.

Huron, D. and E. Selfridge-Field (1994). Research notes (the J. S. Bach Brandenburg Concertos). Software.

Lutz, M. (1996). *Programming Python*. O'Reilly.

Martin, L. and H. H. Hoos (1997). gmn2midi, version 1.0. Computer Program (Microsoft Windows, Apple Macintosh OS, IBM OS/2, UNIX). http://www.informatik.tu-darmstadt.de/AFS/GUIDO/.

Mentalix, I. (2000). Pixel/fx! 2000. Computer Program (UNIX). http://www.mentalix.com/.

Meyer, B. (1997). *Object-Oriented Software Construction* (Second Edition ed.). Prentice Hall.

MIDI Manufacturers Association Inc. (1986). *The Complete MIDI 1.0 Specification*. MIDI Manufacturers Association Inc. http://www.midi.org/.

Musitek (2000). MIDISCAN. Computer Program (Microsoft Windows). http://www.musitek.com/.

Neuratron (2000). Photoscore. Computer Program (Microsoft Windows, Apple Macintosh OS). http://www.neuratron.com/photoscore.htm.

Nienhuys, H.-W. and J. Nieuwenhuizen (1998). *Lilypond User Documentation (containing Mudela language description)*. GNU project. http://www.gnu.org/software/lilypond/.

Read, G. (1969). *Music Notation: A Manual of Modern Practice*. New York: Taplinger.

Schulenberg, J. (2000). gocr. Computer Program (Microsoft Windows, UNIX). http://sourceforge.net/projects/jocr/.

Selfridge-Field, E. (1993). The MuseData universe: A system of musical information. *Computing in Musicology 11*.

Selfridge-Field, E. (1997). Beyond codes: Issues in musical representation. In E. Selfridge-Field (Ed.), *Beyond MIDI: The Handbook of Musical Codes*. MIT Press.

Sun Microsystems (1998). *Java Foundation Classes*. http://java.sun.com/products/jfc/: Sun Microsystems.

Wettschereck, D., D. W. Aha, and T. Mohri (1997). A review and empirical evaluation of feature weighting methods for a class of lazy learning algorithms. *Artificial Intelligence Review 11*, 272–314.