Optical music interpretation

Michael Droettboom, Ichiro Fujinaga, and Karl MacMillan

Digital Knowledge Center,
Milton S. Eisenhower Library,
Johns Hopkins University,
Baltimore, MD 21218
{mdboom,ich,karlmac}@peabody.jhu.edu

Abstract. A system to convert digitized sheet music into a symbolic music representation is presented. A pragmatic approach is used that conceptualizes this primarily two-dimensional structural recognition problem as a one-dimensional one. The transparency of the implementation owes a great deal to its implementation in a dynamic, object-oriented language. This system is a part of a locally developed end-to-end solution for the conversion of digitized sheet music into symbolic form.

1 Introduction

For online databases of music notation, captured images of scores are insufficient to perform musically meaningful searches (Droettboom et al. 2001) and analyses on the musical content itself (Huron 1999). Such operations require a logical representation of the musical content of the score. To date, creating those logical representations has been very expensive. 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 conversion by automatically producing the musical content directly from a digitized image of the printed score.

2 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[1] (Milton S. Eisenhower Library, Johns Hopkins University) (Choudhury et al. 2001). The Levy Collection consists of over 29,000 pieces of popular American music. Phase One of the digitization project involved optically scanning the music in the collection and cataloging them with metadata such as author, title, and date. Currently, Phase Two of the project involves using OMR to derive the musical information from the score images. The OMR system being developed for this purpose must be flexible and extensible enough to deal with the diversity of the collection.

[1] http://levysheetmusic.mse.jhu.edu

3 Overview

For the purposes of this discussion, the problem of optical music recognition is divided into two subproblems: a) the classification of the symbols on the page, and b) the interpretation of the musical semantics of those symbols. The first subproblem has been thoroughly explored and implemented by Fujinaga (1996) in the Adaptive Optical Music Recognition (AOMR) system. The second subproblem, Optical Music Interpretation (OMI), builds on this work and is the subject of this paper, discussed in greater detail in Droettboom (2002).[2]

The AOMR system proceeds through a number of steps. First, the staff lines are removed from the input image file to separate the individual symbols that overlap them. Lyrics are also removed using various heuristics. Commonly occurring symbols, such as stems and noteheads, are then identified and removed using simple filtering techniques. The remaining musical symbols are 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. The recognition itself is exemplar-based and built around the k-nearest-neighbor (k-NN) algorithm (Cover and Hart 1967). The accuracy of the k-NN database can be improved offline by adjusting the weights of different feature dimensions using a genetic algorithm (GA) (Holland 1975).

Recently, the AOMR part of the system has been extended into a more general and powerful system currently under active development: *Gamera* (MacMillan et al. 2002).

4 Background

In general, OMI involves identifying the relationships between symbols by examining their identities and relative positions, and is therefore a structural pattern recognition problem. From this information, the semantics of the score (e.g. the pitches and durations of notes) can be derived.

A number of approaches to OMI use two-dimensional graph grammars as the central problem-solving mechanism (Fahmy and Blostein 1993; Couasnon and Camillerapp 1994; Baumann 1995). Fahmy and Blostein use a novel approach, called graph-rewriting, whereby complex syntactic patterns are replaced with simpler ones until the desired level of detail is distilled. Graph grammar systems may not be the best fit for the present problem, however, since notated music, though two-dimensional on the page, is essentially a one-dimensional stream. 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 into a one-dimensional list before performing any interpretation. Once sorted, all necessary operations for interpretation can be performed on the objects quite conveniently. Any errors in the ordering of symbols, often cited as a major difficulty in OMI, in fact tend

[2] All of the software discussed here is open source and licensed under the GNU General Public License, and runs on Microsoft Windows, Apple MacOS X, and Linux.

to be quite localized and simple to resolve. Therefore, graph grammars are not used as part of the present implementation.

Another approach to OMI is present in the underlying data structure of a music notation research application, *Nutator* (Diener 1989). Its T-TREES (temporal trees) are object-oriented 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. 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 staff underneath it and the clef on top of that staff. This paradigm of communication between glyphs is used heavily throughout the present system. The advantage of this approach is that glyphs can be edited throughout the process at run-time and the results of those changes can be determined very easily.

5 Procedure

In general, the OMI system proceeds in a linear, procedural fashion, applying heuristic rules, and is therefore not a learning system. However, some amount of feedback-based improvement is provided by consistency-checking. In general, due to the diversity and age of our collection, *ad hoc* rules for music notation are used, which are not necessarily those laid out in music notation texts (e.g. Gerou and Lusk 1996).

The OMI system moves through the following steps: input and clean-up, sorting, reference assignment, metric correction, and output. Optionally, the system itself can be tested using an interactive self-debugging system. Each phase of execution is discussed below.

5.1 Input and clean-up

The output from AOMR used by OMI is an eXtensible Markup Language (XML) description of the glyphs identified on the page. Each `<glyph>` entry contains a classification, a bounding box, and a list of features. Object instances are created from the input based on a class name, therefore new classes of glyphs can be easily added to the system.

Glyphs that were separated by poor printing or improper binary thresholding are then joined together using heuristic rules.

5.2 Sorting

Since the glyphs are output from AOMR in an arbitrary order, the sorting phase must put them into a useful order for interpretation: that in which they would be read by a musician. This ordering makes many of the algorithms both easier to write and maintain as well as more efficient.

Contextual information, such as clefs and time signatures, must carry over from one page to the next. The easiest way to deal with this problem is to treat multi-page scores as one very long page. Each page is input in sequence and the bounding boxes are adjusted so that each page is placed physically below the previous one. In this way, multi-page scores are not a special case: they can be interpreted exactly as if they were printed on a single page. (Overflow)

In common music notation, events are read from left to right on each staff. Therefore, before the glyphs can be put into this order, they must first *belong* to a staff. Each glyph will have a reference to exactly one staff. Staff assignment is determined by starting with glyphs that physically overlap a staff and then moving outward to include other related glyphs. Once glyphs have been assigned to staves, those staves need to be grouped into systems (a set of staves performed simultaneously), and then each staff in each system is assigned to a part (a set of notes played by a single performer, or set of performers).

Lastly, in the sorting phase, glyphs are put into musical order. Glyphs are sorted first by part, then voice (see Section 5.3), 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 sorted order has a number of useful properties. Most inter-related glyphs, such as noteheads and stems, appear very close together in the list. Finding relationships between these objects requires only a very localized search. `staff` glyphs serve to mark system breaks and `part` glyphs mark the end of the entire piece for each part. Lastly, this ordering is identical to that used in many musical description languages, including GUIDO (Hoos and Hamel 1997), Mudela (Nienhuys and Nieuwenhuizen 1998) and MIDI (MIDI 1986), and therefore output files can be created with a simple linear traversal of the list.

5.3 Reference assignment

The purpose of this phase is to build the contextual relationships between glyphs to fully obtain their musical meaning. For instance, to fully specify the musical meaning of a notehead, it must be related to a staff, stem, beam, clef, key signature, and accidentals (Figure 1). This is the core of OMI.

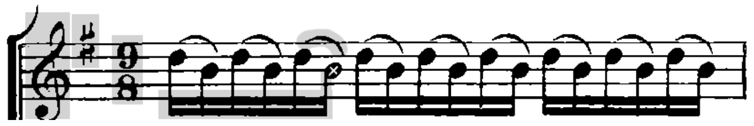

Fig. 1. References to other glyphs (shaded in grey) are required to fully determine the meaning of a notehead (marked by ×).

Reference assignment proceeds through a number of categories of symbols: pitch, duration, voice, chord, articulation, and text. Most of these processes are performed using a single linear scan through the sorted glyphs, much like a Turing machine.

Class hierarchy All glyph classes are members of an object-oriented class hierarchy based on functionality. In this style, most abstract subclasses can be named by adjectives describing their capabilities. For instance, all symbols that can have their duration augmented by dots are subclasses of `DOTTABLE`. This allows new classes of glyphs to be added to the system simply by combining the functionalities of existing classes. It also means that reference-assignment algorithms using these classes can be as abstract as possible. This general design would be much more difficult to implement in more static languages, such as C++, where type modification at run-time is not possible. All of the reference assignment operations described below make extensive use of this class hierarchy.

Pitch OMI has a three-tiered hierarchy of pitch: staff line (which requires a reference to a staff), white pitch (which requires a reference to a clef) and absolute pitch (which requires references to key signatures and accidentals). Each level adds more detail and requires more information (i.e. references to more classes of glyphs) in order to be fully specified. These three different levels are used so that the functionality can be shared between glyphs that use all three, such as notes, and those that only use a subset, such as accidentals.

Determining the correct staff line location of notes on the staff is relatively easy, since most scores have relatively parallel staff lines, pitch can be determined by a simple distance calculation from the center of the staff. However, one of the most difficult problems in determining pitch is dealing with notes outside the staff. Such notes, which require the use of short "ledger" lines, are often placed very inaccurately in hand-engraved scores (Figure 2). The most reliable method to determine the pitches of these notes is to count the number of ledger lines between the notehead and the staff, as well as determining whether a ledger line runs through the middle of the notehead.

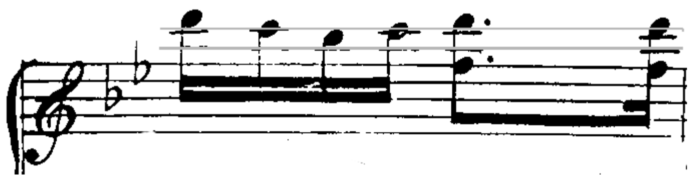
Fig. 2. An example of poorly aligned ledger lines. The grey lines are perfectly horizontal and were added for emphasis.

Duration Durations are manipulated throughout the system as rational (fractional) numbers. Operations upon `Rational` objects preserve the full precision (e.g. triplet eighth notes are represented as exactly $\frac{1}{3}$).

Assigning stems to noteheads, the single most important step in determining the duration of a note, is a difficult problem since stems are visually identical to barlines, although they serve a very different purpose. Height alone is not enough information to distinguish between the two, since many stems may be taller than the staff height, particularly if they are part of a chord. Instead, vertical lines are dealt with by a process of elimination.

1. Any vertical lines that touch noteheads are assumed to be stems.
2. Any remaining vertical lines taller than the height of a staff are assumed to be barlines.
3. The remaining vertical lines are likely to be vertical parts of other symbols that have become broken, such as sharps or naturals.

If the guesses made about stem/barline identity turn out to be wrong, they can often be corrected later in the metric correction stage (Section 5.4).

The direction of the stem is determined based on the horizontal location of the stem. If the stem is on the right-hand side, the stem direction is assumed to be up. If the stem is on the left-hand side, the stem direction is down. Stem direction can not be determined based on the vertical position of the stem because the notehead may be part of a chord, in which case the notehead intersects the stem somewhere in the middle. This method must be superseded by a more complex approach for chords containing second (stepwise) intervals, since some of the noteheads are forced to the other side of the stem.

Voices Multi-voicing, where multiple parts are written on the same staff, often occurs in choral music or compressed orchestral scores to conserve space. Just as in multi-page scores, the approach here is to massage the data into a form where it no longer is a special case. Therefore, each voice is split into a separate logical part (Figure 3). Note that some glyphs exist in all logical parts (such as clefs and time signatures) whereas others are split (notes). Determining whether to split a measure into multiple parts is determined automatically.

5.4 Metric correction

Physical deterioration of the input score can cause errors at the recognition (AOMR) stage. Missing or erroneous glyphs cause voices to have the wrong number of beats per measure. These errors are quite serious, since they accumulate over time, and parts become increasingly out of synchronization. Fortunately, many of these errors can be corrected by exploiting a common feature of typeset music: notes that occur at the same time are aligned vertically within each system (set of staves) of music. Unfortunately, some poorly typeset scores do not exhibit this feature. In that case, metric correction fails consistently, and is automatically bypassed.

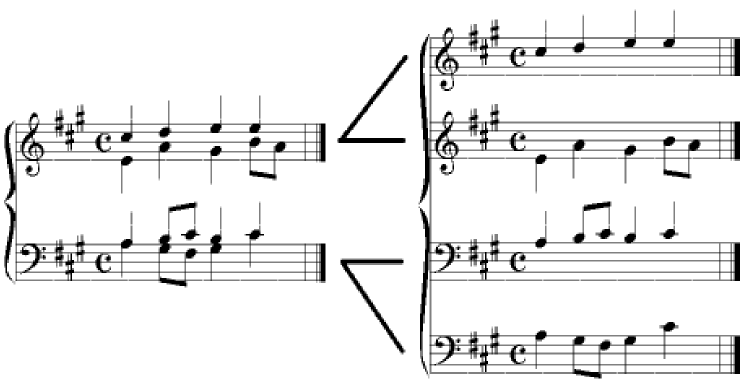

Fig. 3. Splitting multi-voiced scores.

The score is examined, one measure at a time, across all parts simultaneously. A number of approaches are then applied to that measure to correct the durations of notes and rests and barline placement. The primary goal is to ensure that the length of the measure across all parts is the same before moving to the next measure, and to make any corrections in the most intelligent way possible.

At present, there are seven approaches to metric correction that are attempted. For each, a particular change is made, and then the consistency check is performed again. If the change does not improve the measure, the change is undone and the next approach is tried. a) Measures containing only a single rest are adjusted to the length of the entire measure. b) Whole rests and half rests, which are visually identical, are traded and checked for consistency. c) Specks of ink or dust on the page can be confused for augmentation dots. Therefore, augmentation dots are ignored. d) Stems that are too far from a notehead may be interpreted as a barline. These barlines are reexamined as if they were stems. e) Barlines can be missed entirely, and new ones are inserted based on the locations of barlines in other parts. f) Flags and beams can be misread. In this case, the duration of notes is estimated by examining their horizontal position in relation to notes in other parts (Figure 4). g) As a worst case scenario, empty durational space is added to the end of the measure so that all parts have the same duration. This does not usually produce an elegant solution, but it still prevents the errors of one measure to accumulate across an entire piece.

Metric correction works best in scores with many parts, because there is a large amount of information on which to base the corrections. It is also in multi-part scores where metric correction is most crucial. However, many of the algorithms can improve the accuracy of single-part scores as well.

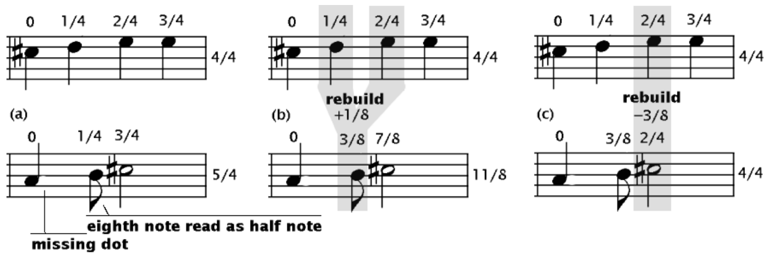

Fig. 4. Adjusting the durations of notes based on the durations in other parts.

5.5 Output

Unfortunately, there is no single accepted standard for symbolic musical representation (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. For example, GUIDO files are organized part by part, whereas Type 0 MIDI files interleave the parts together by absolute time (a temporal stream). To handle this, 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 and MIDI is implemented, but other musical representation languages such as Lilypond Mudela are planned.

5.6 Interactive self-debugger

The ability to interact with the data of a running program, using a scripting language such as Python, greatly reduces the length of the develop-test cycle. However, manipulating graphical data, such as that in OMI, is quite cumbersome using console-based tools. For example, selecting two-dimensional 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. The interactive self-debugger proved to be an invaluable tool when developing the OMI application. While extra development effort was expended to create it, those hours were easily made up by the ease with which it allows the programmer to examine the state of the data structures.

6 Conclusion

The system presented here represents a number of pragmatic solutions to the problem, providing a useful tool that is effective on a broad range of scores. In the near future, it will allow for the creation large online databases of symbolic musical data: a valuable resource for both musicologists and music-lovers alike.

Acknowledgements

The second phase of the Levy Project is funded through the NSF's DLI-2 initiative (Award #9817430), an IMLS National Leadership Grant, and support from the Levy Family.

References

Baumann, S.: A simplified attributed graph grammar for high-level music recognition. International Conference on Document Analysis and Recognition. (1995) 1080–1083

Choudhury, G. S., DiLauro, T., Droettboom, M., Fujinaga, I., MacMillan, K.: Strike up the score: Deriving searchable and playable digital formats from sheet music. D-Lib Magazine. **7(2)** (2001)

Couasnon, B., and Camillerapp, J.: Using grammars to segment and recognize music scores. International Association for Pattern Recognition Workshop on Document Analysis Systems. (1994) 15–27

Cover, T., Hart, P.: Nearest neighbor pattern classification. IEEE Transactions on Information Theory. **13(1)** (1967) 21–27

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

Droettboom, M., Patton, M., Warner, J. W., Fujinaga, I., MacMillan, K., DiLauro, T., Choudhury, G. S.: Expressive and efficient retrieval of symbolic musical data. International Symposium on Music Information Retrieval. (2001) 163–172

Droettboom, M.: Selected Research in Computer Music. Master's thesis. (2002) The Peabody Institute of the Johns Hopkins University.

Fahmy, H., D. Blostein.: A graph grammar programming style for recognition of music notation. Machine Vision and Applications. **6(2)** (1993) 83–99

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

Gerou, T., L. Lusk.: Essential Dictionary of Music Notation. (1996) Alfred, Los Angeles.

Holland, J. H.: Adaptation in Natural and Artificial Systems. (1975) University of Michigan Press, Ann Arbor.

Hoos, H. H., Hamel, K.: GUIDO Music Notation Version 1.0: Specification Part I, Basic GUIDO. (1997) Technical Report TI 20/97, Technische Universität Darmstadt.

Huron, D.: Music Research Using Humdrum: A User's Guide. (1999) Center for Computer Assisted Research in the Humanities, Menlo Park, CA.

Huron, D., Selfridge-Field, E.: Research notes (the J. S. Bach Brandenburg Concertos). (1994) Computer software.

MacMillan, K., Droettboom, M., Fujinaga, I.: Gamera: A Python-based toolkit for structured document recognition. Tenth International Python Conference. (2002) (In press)

MIDI Manufacturers Association Inc.: The Complete MIDI 1.0 specification. (1986)

Musitek.: MIDISCAN. Computer Program (Microsoft Windows).

Neuratron. Photoscore. Computer Program (Microsoft Windows, Apple MacOS).

Nienhuys, H., Nieuwenhuizen. J.: LilyPond User Documentation (Containing Mudela Language Description). (1998)

Van Rossum, G., Drake, F. L.: Python Tutorial. (2000) iUniverse, Campbell, CA.

Selfridge-Field, E.: The MuseData universe: A system of musical information. Computing in Musicology **9** (1993) 11–30

Selfridge-Field, E. Beyond MIDI: The Handbook of Musical Codes. (1997) MIT Press, Cambridge, MA.