Selected Research in Computer Music

Michael Droettboom

Submitted in partial fulfillment of the requirements for the degree of
Master of Music in Computer Music Research
at The Peabody Conservatory of Music,
The Peabody Institute of the Johns Hopkins University

Baltimore, Maryland, United States of America
April, 2002

Peabody Conservatory of Music

Johns Hopkins University

Statement of Acceptance

Master of Music in Computer Music (Research)

Be it known that the attached research thesis submitted by Michael Droettboom has been accepted in partial fulfillment of the requirements for the degree of Master of Music in Computer Music (Research).

Computer Music Faculty

Date

Computer Music Faculty

Date

Abstract

This thesis describes three interrelated projects that cut across the author's interests in musical information representation and retrieval, programming language theory, machine learning and human/computer interaction.

I. Optical music recognition. This first part introduces an optical music interpretation (OMI) system that derives musical information from the symbols on sheet music.

The first chapter is an introduction to OMI's parent field of optical music recognition (OMR), and to the present implementation as created for the Levy project.

It is important that OMI has a representation standard in which to create its output. Therefore, the second chapter is a somewhat tangential but necessary study of computer-based musical representation languages, with particular emphasis on GUIDO and Mudela.

The third and core chapter describes the processes involved in the present optical music interpretation system. While there are some details related to its implementation in the Python programming language, most of the material involves issues surrounding music notation rather than computer programming.

The fourth chapter demonstrates how the logical musical data generated by the OMI system can be used as part of a musical search engine.

II. Tempo extraction. The second part presents a system to automatically obtain the tempo and rubato curves from recorded performances, by aligning them to strict-tempo MIDI renderings of the same piece of music. The usefulness of such a system in the context of current musicological research is explored.

III. Realtime digital signal processing programming environment. Lastly, a portable and flexible system for realtime digital signal processing (DSP) is presented. This system is both easy-to-use and powerful, in large part because it takes advantage of existing mature technologies. This framework provides the foundation for easier experimentation in new directions in audio and video processing, including physical modeling and motion tracking.

Contents

List of Figures **vii**

List of Tables **ix**

I Optical music interpretation **1**

1 Introduction **2**
1.1 The Lester S. Levy Collection of Sheet Music . . . 3
1.2 Overview . . . 4
1.2.1 Adaptive optical music recognition . . . 4
1.2.2 Optical music interpretation . . . 5
1.3 Overview . . . 7

2 Musical representation languages **8**
2.1 Resources . . . 9
2.2 Background . . . 9
2.3 Basic syntax . . . 10
2.4 Extension framework . . . 12
2.5 Human issues . . . 13
2.5.1 Brevity vs. clarity . . . 13
2.5.2 Representational adequacy and context-dependence . . . 13
2.6 Implementation issues . . . 14
2.6.1 Parsing . . . 14
2.7 Logical abstraction . . . 15
2.7.1 Logical abstraction in text typesetting . . . 15
2.7.2 Logical abstraction in music typesetting . . . 15
2.7.3 Bibliographic information . . . 16
2.8 Software tools . . . 16
2.8.1 GUIDO . . . 16
2.8.2 Mudela . . . 18
2.9 Conclusion . . . 19

3 Implementation of an optical music interpretation system 20
3.1 Input . . . 21
3.1.1 XML glyph format . . . 21
3.1.2 Glyph list . . . 23
3.2 Assembly . . . 23
3.3 Sorting . . . 24
3.3.1 Handling multi-page scores . . . 24
3.3.2 Assigning glyphs to staves . . . 25
3.3.3 Grouping staves into systems . . . 27
3.3.4 Grouping staves into parts . . . 27
3.3.5 Temporal sorting . . . 27
3.4 Reference assignment . . . 28
3.4.1 Class hierarchy . . . 29
3.4.2 Pitch . . . 29
3.4.3 Duration . . . 35
3.4.4 Voices . . . 39
3.4.5 Chords . . . 40
3.4.6 Articulation . . . 41
3.4.7 Text . . . 42
3.5 Metric correction . . . 43
3.5.1 Overview . . . 43
3.5.2 Classification . . . 43
3.5.3 Algorithms for metric correction . . . 44
3.6 Output . . . 50
3.6.1 File formats . . . 50
3.6.2 Pluggable back-ends . . . 50
3.6.3 Mixin classes . . . 51
3.6.4 Demonstration of output . . . 52
3.7 Interactive self-debugger . . . 53
3.7.1 Overview . . . 55
3.7.2 Pages . . . 55
3.7.3 Reflections on the interactive self-debugger . . . 59
3.8 Conclusion . . . 59

4 Symbolic music information retrieval 60
4.1 Introduction . . . 60
4.2 Other search engines . . . 60
4.2.1 Themefinder . . . 61
4.2.2 MELDEX . . . 62
4.3 Capabilities . . . 62
4.3.1 Extensibility . . . 62
4.3.2 Meeting diverse user requirements . . . 63
4.4 The core search engine . . . 63
4.4.1 Inverted lists . . . 64
4.5 The musical search engine . . . 65

4.5.1 Secondary indices . . . 66
4.5.2 Partitions . . . 69
4.5.3 Regular expressions . . . 72
4.6 Conclusion . . . 72

References **73**

II Tempo extraction **78**

5 RUBATO: A system for determining tempo fluctuation in recorded music **79**
5.1 Introduction . . . 79
5.2 A brief history of musical time . . . 80
5.2.1 Current research . . . 81
5.2.2 Tempo extraction . . . 82
5.3 The RUBATO program . . . 84
5.3.1 Smoothing the audio signal . . . 84
5.3.2 Dynamic programming algorithm . . . 85
5.3.3 Extracting tempo curves . . . 87
5.3.4 More examples . . . 91
5.4 Conclusion . . . 93

References **94**

III Realtime digital signal processing environment **96**

6 RED: Realtime Environment for Digital Signal Processing **97**
6.1 Introduction . . . 97
6.2 Architecture . . . 98
6.2.1 Overview . . . 98
6.2.2 Goals . . . 98
6.2.3 General . . . 99
6.2.4 Modularized . . . 99
6.2.5 Realtime, low-latency performance . . . 102
6.2.6 Built from existing tools . . . 103
6.2.7 Portable . . . 106
6.3 Usage examples . . . 106
6.3.1 Unit generator objects . . . 106
6.3.2 Connecting signals . . . 108
6.4 Conclusion . . . 111

References **112**

List of Figures

1.1 Differences in music typsetting. . . . 3
1.2 Differences in musical semantics. . . . 3

2.1 *GUIDO NoteServer* . . . 17
2.2 *Denemo* music editor. . . . 19

3.1 Some example glyphs from the XML output produced by AOMR. . . . 22
3.2 Broken lines that are rejoined by the assembly phase. . . . 23
3.3 Handling multi-page scores. . . . 25
3.4 Assigning glyphs to staves. . . . 26
3.5 References required to fully determine a notehead's meaning. . . . 28
3.6 The pitch hierarchy. . . . 30
3.7 Different clefs. . . . 31
3.8 Poorly aligned ledger lines. . . . 32
3.9 The finite-state automaton (FSA) used to locate and build key signatures. . 34
3.10 The durational heirarchy. . . . 35
3.11 Chords containing seconds. . . . 37
3.12 Augmentation dots. . . . 38
3.13 A notehead with two stems is separated into two notes. . . . 38
3.14 Splitting multi-voiced staves. . . . 40
3.15 Chords. . . . 41
3.16 Articulations. . . . 42
3.17 Whole rests depend on the time signature. . . . 45
3.18 Erroneous dot removal algorithm . . . 46
3.19 Barline to stem algorithm. . . . 47
3.20 Splice algorithm . . . 48
3.21 Readjust algorithm with an exact match. . . . 49
3.22 Readjust algorithm with estimation. . . . 50
3.23 The original image (for output demonstration). . . . 52
3.24 The PostScript output (of output demonstration). . . . 53
3.25 The GUIDO output (of output demonstration). . . . 54
3.26 The output from GUIDO *NoteServer* (of output demonstration). . . . 54
3.27 Attribute page in the interactive debugger. . . . 56

3.28 Classes page in the interactive debugger. 56
3.29 Glyph info page in the interactive debugger. 57
3.30 List browser page in the interactive debugger. 58

4.1 The importance of rhythmic specificity. 61
4.2 Workflow diagram of the musical search engine. 64
4.3 Musical example used to demonstrate the search engine. 66
4.4 Fully specified symbolic representation of the example in Figure 4.3. 68
4.5 The example measure of music showing moment numbers. 71

5.1 Dr. Arthur B. Lintgen 80
5.2 Audio waveform of a MIDI rendering. 85
5.3 Audio waveform of a performance by Glenn Gould. 85
5.4 The gross contours of two recordings. 86
5.5 Match path of the Bach fugue. 88
5.6 Tempo curve of Gould's 1963 recording of the Bach fugue. 90
5.7 Tempo curves of 1955 and 1982 Gould recordings. 91
5.8 Tempo curves of recordings of Fauré's *Requiem*. 92

List of Tables

3.1 Metric correction algorithms 44
3.2 An example of a core and mixin class. 51

5.1 Results of the dynamic programming algorithm. 87

6.1 Compatibility matrix of the various third party tools used to build RED. . 107

Part I

Optical music interpretation

Chapter 1

Introduction

In recent years, the availability of large online databases of text have created entirely new methods of scholarly research. Those same collections are beginning to add multimedia content; unfortunately, the art and science of content-based retrieval of non-textual data is significantly behind that of text. In the case of music notation, captured images of scores are insufficient to perform musically meaningful searches and analyses on the musical content itself. For instance, a casual user may wish to find a work containing a particular melody, or a musicologist may wish to perform a statistical analysis on a particular body of work. Such operations require a logical representation of the musical meaning 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 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 interpreting the musical content directly from a digitized image of the printed score.

There has already been a moderate amount of academic research in OMR, most notably by Bainbridge (1997), Fujinaga (1996) and Ng (1992). Since the Western music notation system is over 350 years old, and has evolved significantly during that time (Read 1969; Gerou and Lusk 1996), 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 1.1) and semantic (Figure 1.2) levels. Inflexibility to such differences is the primary drawback of commercial OMR products, such as MIDISCAN (Musitek 2000) and 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

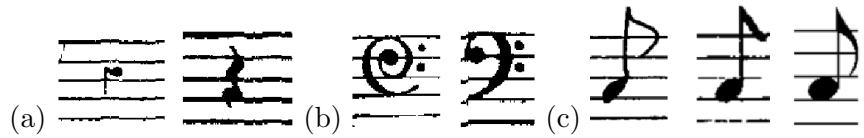


Figure 1.1: Differences in music typesetting: (a) two different typeset quarter rests; (b) two different typeset bass clefs; (c) handwritten, hand-engraved and digitally typeset eighth notes.

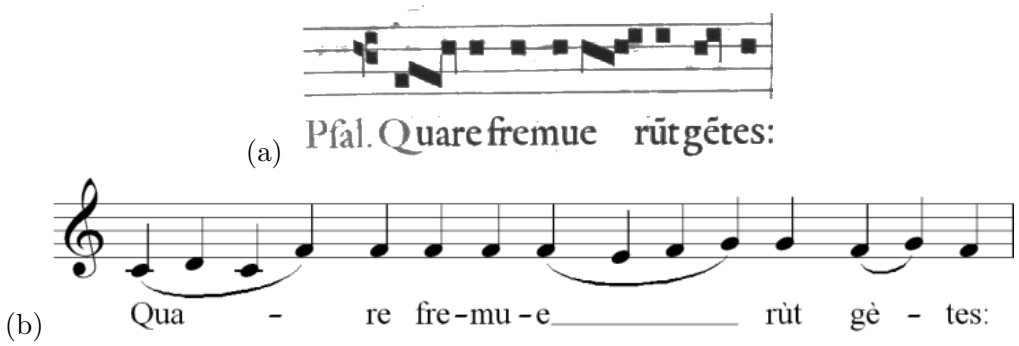


Figure 1.2: Differences in musical semantics. An excerpt from an anonymous plain chant composition from the *Typographica Medicea* in (a) it original square note neums, and (b) the equivalent in modern notation.

historical sheet music collections, are likely to have a difficult 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) (Choudhury et al. 2000). 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 involved optically scanning 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 information 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.

1.2 Overview

For the purposes of this discussion, 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 Fujinaga as the Adaptive Optical Music Recognition (AOMR) system, summarized below. The second subproblem builds on this work and is the subject of Part I of this thesis.

1.2.1 Adaptive optical music recognition

The AOMR system offers five important advantages over similar commercial offerings.

- **Automatability.** Multiple scores can be interpreted in sequence automatically, an essential feature for large musical collections.
- **Portability.** The software is written in portable C and C++ and therefore runs on many platforms.[1]
- **Extensibility.** AOMR can *learn* to recognize different music symbols, a serious issue considering the diversity of common music notation.
- **Freedom.** The software is open-source and licensed under the GPL (Free Software Foundation 1991).

The AOMR process proceeds through a number of steps. First, using vertical run-length coding and projection analysis, the staff lines are removed from the input image file. 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.

[1] AOMR has been ported to GNU/Linux on x86 and PPC, Sun Solaris, sgi IRIX, NeXTSTEP, Apple Macintosh OS-X, and Microsoft Windows 95/98/NT/2000 all using the GNU `gcc` compiler.

The exemplar-based classification model is based on the idea that objects can be categorized by their similarity to stored examples. In AOMR, this model is implemented using 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 neighbors is then assigned to the unclassified sample. Besides its simplicity and intuitiveness, the classifier can be easily modified. By continually adding new samples as it encounters them into the database, it becomes an adaptive learning 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 1993). 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 et al. 1997). Note that the genetic algorithm can be run off-line without slowing the speed of the recognition process.

Recently, the AOMR system is being phased out in favor of a more general and powerful system, *Gamera*, for general-purpose document recognition (MacMillan et al. 2001).

1.2.2 Optical music interpretation

In general, the Optical Music Interpretation (OMI) phase involves identifying the relationships between symbols by examining their relative positions. From this information, the semantics of the score (e.g. the pitches and durations of notes) can be derived. Lastly, many errors during the AOMR stage can be corrected by examining the consistency of symbols on the page.

Background

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). Graph grammars parse the relationships between symbols on the page by matching them to a set of syntactic patterns. 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 derived. 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, cited as a major difficulty in OMI, in fact tend to be quite localized and simple to resolve. Therefore, graph grammars are not a part of 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 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. 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 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 at run-time and the results of those changes can be determined very efficiently.

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. To meet these goals, the Python programming language (Van Rossum and Drake 2000) was chosen. Python is an object-oriented, dynamic interpreted language. In addition, the source code is publicly available. It proved to be an effective tool for meeting the design requirements:

- **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 and therefore used by other scripting systems.

- **Portability.** Since all of the input and output formats of OMI are in eXtensible Markup Language (XML) or ASCII text, the OMI system is portable to any platform with 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 each symbol can be refined interactively without the need to recompile or even re-execute the application.
- **Freedom.** OMI is open-source, as is the Python language.

Logical interpretation vs. performance

The OMI stage does not attempt to apply any performance knowledge to the music, but instead aims to create an accurate representation of everything on the page. It is possible, however, to feed the output of OMI to a performance software that would create a more life-like rendition of the music.

1.3 Overview

This part of the thesis proceeds through the beginning, middle and future of the development of an optical music interpretation system. Before beginning development, it was important to choose the right output format. Therefore, in chapter 2, research into musical representation languages is presented. Following that is the description of the actual task of designing and developing an optical music interpreter. Lastly, future directions using the generated data as the basis for an on-line, musically intelligent search engine are presented.

Chapter 2

Musical representation languages

> "If musical information were well understood and fixed, music representation would be a much simpler problem. In reality, we do not know all there is to know and the game is constantly changing. For both of these reasons, it is important that music representations allow extensions to support new concepts and structures." —Roger B. Dannenberg (1993, p. 22)

Clearly, the optical music interpretation (OMI) system needs a way to format and store its output. Unfortunately, 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 has been argued that common music notation (CMN) (Read 1969) is one of the most complex notations in existence (Byrd 1994). Representing the graphically complex two-dimensional printed score as a one-dimensional string of characters is such a difficult problem that, in nearly forty years of research, a ubiquitous standard has not emerged. Therefore, compromise is almost inevitable when selecting a musical representation language. Add to that the practical issues of application and development support, and the compromises can be even greater.

When the OMI project began, an in-depth comparison of the two most promising candidates, GUIDO (Hoos and Hamel 1997) and MUsical DEscription LAnguage (Mudela) (Nienhuys and Nieuwenhuizen 1998) was made to assess their suitability to the long-term archiving of scanned sheet music.

2.1 Resources

The book *Beyond MIDI* (Selfridge-Field 1997) is an exhaustive survey of musical representation languages, (though it predates both GUIDO and Mudela). It also includes chapters regarding issues (Selfridge-Field 1997b) and guidelines (Halperin 1997) for musical codes. Dannenberg (1993) provides an fairly exhaustive overview regarding music representation. As well, there are at least two good indices of musical codes available on the internet (Castan 2000, Mounce 2000).

2.2 Background

GUIDO

Holger H. Hoos originally developed GUIDO out of the necessity for a human-readable music description language for his music analysis package *SALIERI* (Hoos et al. 1998). In partnership with Keith A. Hamel, and others, GUIDO was extended to support other important common music notation features (Hoos and Hamel 1997).

GUIDO is currently defined in two documents: The Basic GUIDO Specification (Hoos and Hamel 1997) and The Advanced GUIDO Specification (Hoos, correspondence 2000). The forthcoming Extended GUIDO Specification is expected to cover unconventional music notation (e.g. microtones, absolute timing).

Mudela

Mudela is more closely tied to the practical considerations of music typesetting. It originated as the input language to the TEX-based "music compiler" *LilyPond*, and is now also the official language of the *GNU Music Project*. Its original form borrowed heavily from the Tilia representation used by the *Lime* music editor (Cottle and Haken 1997; Haken and Blostein 1993).

Mudela does not have a formal specification, only a preliminary user's guide (Nienhuys and Nieuwinhuizen 1998).

2.3 Basic syntax

Detailed information about the GUIDO and Mudela languages are freely available from their respective authors. This section, therefore, is designed not to introduce the reader to the languages, but rather to provide enough detail for comparison.

Both GUIDO and Mudela are encoded as text and are therefore human-readable and portable. While GUIDO syntax is borrowed from many sources, Mudela syntax is inspired mainly by TEX (Knuth 1984).

GUIDO

The primary advantage of GUIDO's syntax is its simplicity. It defines only three categories of atoms: notes and rests, ordering constructs, and tags.

Notes and rests. These simple atoms are used only for the most basic elements of music notation: notes and rests. They are made up of five parts, all of which except the first are optional:

`c`	`#`	`-1`	`*3`	`/4`
pitch-name	*accidental*	*octave*	*numerator*	*denominator*

Examples of note and rest atoms:

`d1*3/4`	*d*, octave 1, dotted half note
`d1/2.`	*d*, octave 1, dotted half note
`c#-1/8`	$c^\sharp$, octave -1, eighth note
`h/6`	*b* (German *h*), triplet quarter note
`_*2`	rest (_), half note
`c2&&/2`	$c^{\flat\flat}$, octave 2, half note
`cis/4`	$c^\sharp$ (German *c-is*), quarter note
`fa1##`	$f^\times$ (*fa* in fixed-*do* with a double-sharp), octave 1
`sol&0`	$g^\flat$ (*sol* in fixed-*do* with a flat)

Ordering constructs. The ordering of atoms places objects sequentially (`[ ... ]`) or simultaneously (`{ ... }`). A minimal GUIDO file must be enclosed in square brackets to indicate that the notes appear in sequential order.

```
[ c1/4 d e f g a b c2/2 ] % C major scale
{ c e g } % C major triad
```

Tags. All other elements and properties that modify notes and rests are indicated using tag syntax. Since tags begin with an escape-character (the backslash '\'), it is always possible to distinguish tags from normal note elements. Tags have an *id* (name), zero or more named arguments (in hairpin brackets '< >') and, optionally, an associated group of notes (in parentheses '( )').

```
[ \repeatBegin c4 d e c \repeatEnd ] % Frere Jacques
```

```
\clef<"treble">     % Insert a treble clef
[ \beam(c8 d e c) ] % Double-time Frere Jacques with beams
```

Mudela

While Mudela syntax is more complex, it should be natural to those familiar with TEX or LATEX(Lamport and Bibby 1994). Groupings are formed with braces '{ }', and escape commands are preceded with a backslash '\'. Mudela makes extensive use of bracketing: braces '{ }' for sequencing, hairpin brackets '< >' for simultaneity, parentheses '( )' for slurs and square brackets '[ ]' for beams. Note that braces are used in two contexts: both for sequencing and more general grouping of elements.

Anything between white space can be interpreted as an atom. Packages (similar to modules, libraries or plugins) can be loaded to interpret atoms in new ways. This flexibility makes the language quite convenient for authors, but very convoluted for programmers who want to support the language. The language definition, therefore, is a moving target: never fully defined without actual source code, and somewhat analogous to a natural language without a dictionary or grammar.

There are four standard ways in which atoms are interpreted: normal mode, note mode, chord mode and lyric mode.

Normal mode. A mode in which atoms have meaning only as strings. The exact purpose of this mode is unclear.

Note mode. Each atom represents a pitch name. The keyword used for pitch names can be changed by loading different language packages. As in GUIDO, redundant information is omitted by passing along octave and duration values from one note to the next.

```
d2. cs8 \times 1/6 {hf8} r2 c'ff2 cs4
```

Chord mode. Chords can be built automatically using notation similar to jazz chord notation. The root of the major chord is followed by any additional, ommitted or modified scale degrees. If this notation is insufficient, notes can be combined into chords individually using parentheses '`( )`'.

```
c1 c:3- c:9-.5+.7+ c:7\^\ 5.3 c/e c:7/e
```

Lyric mode. Lyric mode views each atom as a syllable of lyrics followed by a durational value.

```
\lyrics Oh4 say, can you see2 by8.\ the16
```

2.4 Extension framework

Both GUIDO and Mudela have well-defined ways to add new features to the language.

GUIDO

GUIDO has an eXtensible Markup Language (XML) inspired approach to extensions: new features can be added by using new tags, but the underlying lexical syntax cannot be changed (Bray et al. 2001). This allows legacy or domain-specific programs to ignore unrecognized or unimportant tags and utilize the remaining information in a GUIDO file. Since the tag syntax is standardized, it is even possible, in many instances, for an application to manipulate recognized elements while maintaining their relationship to unrecognized tags.

For example, suppose a GUIDO author invented the tag `\editorial` to handle editor's notes.

```
[ c8 f \editorial<"Handel wrote D-sharp">(ef) f ]
```

Now suppose the author wanted to run the file through a program, *transpose*, that is unaware of `\editorial` tags. It might transpose the file down a semi-tone as follows:

```
[ b7 e8 \editorial<"Handel wrote D-sharp">(d) e ]
```

Note that the original editorial comment remains intact, even though *transpose* is unaware of its meaning.

Mudela

Like TEX, Mudela allows the inclusion of macros, or pieces of executable code, to extend the language. Unlike GUIDO's tags, macros can add arbitrary tokens to the language, or even redefine existing ones. In common use are the language packages that change the name of pitches and keywords. For example, if the Mudela author loads the `dutch` package, $c^\sharp$ is represented by `cis`, whereas with the `english` package it is `cs`. More complex concepts such as percussion staves are also implemented as macro packages. Packages are written in the Mudela language itself, or embedded version of the Scheme programming language (Abelson et al. 1998), meaning that any extension derived from existing elements can be used by any program that supports Mudela extensions.

2.5 Human issues

This section deals with GUIDO and Mudela's suitability for use directly by human-users, who would both write and read code.

2.5.1 Brevity vs. clarity

Mudela's syntactic flexibility is exploited to make the language concise and often isomorphic (atoms look like the symbols they represent) or mnemonic (atoms have a memorable association to their meaning). For example, slurs are '`( )`' and accents are '`->`'. While this makes the language very convenient for authors, it can be opaque for inexperienced readers.

The meaning of GUIDO keywords is more self-evident, at least to English speakers: If one knows the English term for a musical element, one already knows the associated GUIDO keyword. This breaks down, however, when keywords are abbreviated. While some abbreviations, such as `\bm`, have equivalent full forms (`\beam`), others do not, e.g. `\stacc` (staccato) and `\oct` (octave). By including both full and shortened versions in the language definition, the problem of forgetting the correct abbreviations could be eliminated.

2.5.2 Representational adequacy and context-dependence

Both GUIDO and Mudela are based on the concept of *representational adequacy*. This means that the bare minimum of information is required to represent scores. For example,

since notes of a given duration tend to be followed by notes of the same duration, when a duration value is not supplied it defaults to the value of the previous note.

Representational adequacy is a great convenience for authors who are entering scores manually. It can also, in many cases, improve the readability of the text representations because the reader does not have to wade through redundant information. However, it ties the particular representation of phrases to their context (location within the file.) Consider the following string of GUIDO notes:

```
d e f g
```

The appearance of these notes after either `c4/8` or `c2/4` results in an entirely different interpretation. This makes copy-and-paste score writing precarious. In Mudela, context dependence is an even bigger problem because atom interpretation is also dependent on the current syntax mode.

2.6 Implementation issues

The human considerations and machine considerations are often at odds. This section examines the issues in writing software that supports these languages.

For example, the context-dependence problems created by the concept of representational adequacy, a point of potential trouble for human users, is not a problem for computer programs. The score can be normalized (fully specified) on input and distilled (redundancies removed) on output. The core parser and data structures, therefore, do not need to be concerned with representational adequacy.

2.6.1 Parsing

GUIDO's syntactic simplicity makes implementing programs that read it (parsers) less complex and more robust. Only a handful of atom types need to be recognized. This simplicity also allows elegant extensibility of the language (Section 2.4) and the handling of undefined tags.

While macros add a lot of power to Mudela files, particularly for logical abstraction (Section 2.7), it makes implementation of the language much more difficult. All packages used by the score must be available on the host machine. It would be difficult to make a

reverse-engineered Mudela parser since the exact semantics of Mudela's macro language is undefined outside of the *LilyPond* source code.

2.7 Logical abstraction

2.7.1 Logical abstraction in text typesetting

Logical abstraction is commonplace in text typesetting. An author may specify the logical structure of a document, such as chapter or section headings, without specifying any visual attributes. This aids in two things:

1. **Flexible presentation.** The document can easily adapt to different formats. For example, fonts may change if the document is presented in journals, in books, on large computer displays or small palmtop devices.

2. **Expressive searching.** If the different parts of the text are marked by logical meaning, automated searches through a large number of documents can be more accurate. For instance, you may want to search for an author's name only in the *author* field of a number of documents.

Examples of this approach to text formatting appear in the LaTeX macro package for TeX (Lamport and Bibby 1994), and style templates in Microsoft Word (Rubin 1999).

2.7.2 Logical abstraction in music typesetting

Logical abstraction of common music notation is a thornier issue. Byrd's Ph. D. thesis argues for the "non-feasibility of fully-automatic high-quality music notation" (1984, 1). This implies that one cannot leave all issues of visual formatting in the hands of the computer, as is often done in text formatting. For long-term archiving of a large number of musical scores, however, it would be advantageous to store both the logical information and the visual formatting together.

In most cases, GUIDO and Mudela already specify music logically, using visual overrides only when needed. For example, notes are given as pitch names, with facilities to move notes if the automatic placement is not adequate. In music, the need for having both logical and visual information is much more acute than with text, since it is much harder to automate the conversion from a logical representation to a visual one.

2.7.3 Bibliographic information

It would also be desirable to store all bibliographic information related to the score in the file itself. Mudela has a header group in which one can provide information such as title, opus number, composer, and editor. An extension of this system is used by the Mutopia Project (Nienhuys 2001), a database of scores in Mudela format, to automatically generate web-based catalogs.

The GUIDO specification does not have such a system. It only defines tags for author and title. However, it would be trivial for one to define more bibliographic tags and publish their definitions along side the standard GUIDO specifications. These tags could even be based on an existing bibliographic file standard such as BIBTEX (Patashinik 1988) or Dublin Core Metadata (Weibel et al. 1998) to provide interchange with existing publishing and cataloguing systems.

2.8 Software tools

The presence of existing tools can add value to a language, since features that already exist do not need to be re-written in-house.

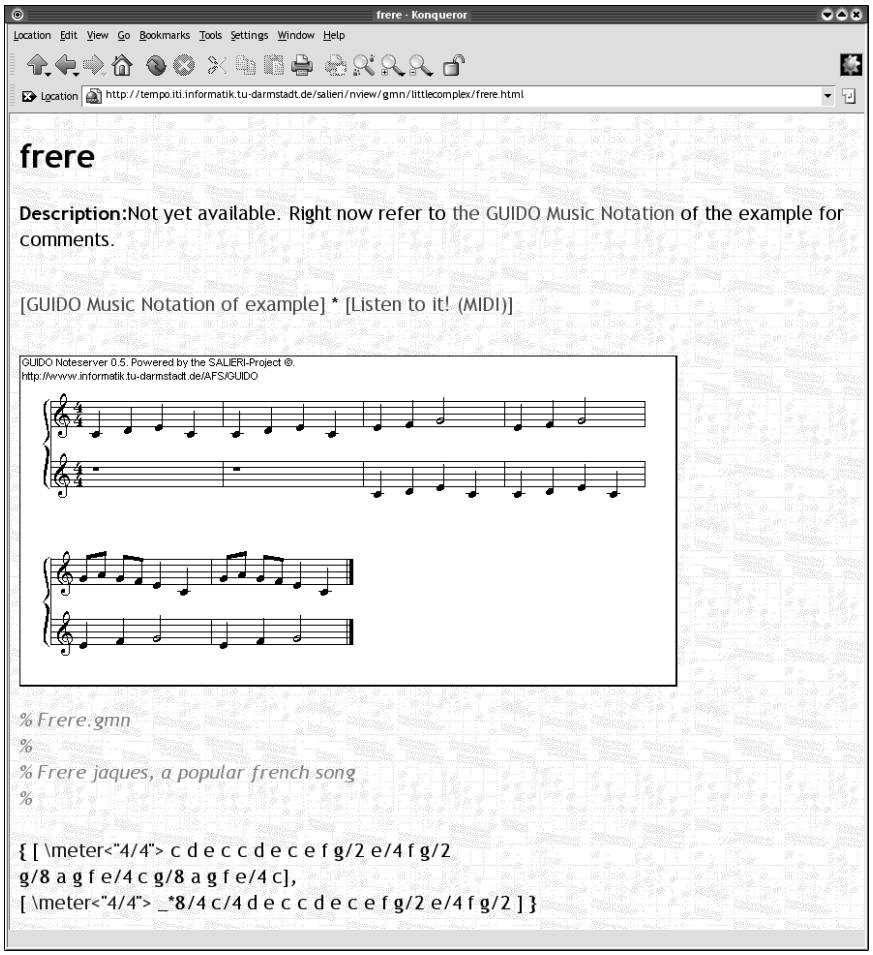
2.8.1 GUIDO

Applications

GUIDO NoteServer (Figure 2.1) (Renz and Hoos 1998) displays GUIDO as common music notation inside a web-browser. The user enters GUIDO into a web-based form, and an image of that music in common music notation is returned. At the time of this writing, it supports most of the Basic GUIDO specification. It can be used with any modern web-browser.

SALIERI is an algorithmic composition and analysis program that uses GUIDO as its internal data representation.

NoteAbility is a commercial notation program developed by Keith Hamel (1998). *NoteAbility Pro* runs on OpenStep and Mac OS-X and allows fully specified Advanced GUIDO to be imported and exported. The scaled-down *NoteAbility Lite* runs on Microsoft Windows 95/98/ME and Mac OS 8/9 and can export Advanced GUIDO. Importing is limited to Basic GUIDO.



Corresponding GUIDO input:

```
{ [ \meter<"4/4"> c d e c c d e c e f g/2 e/4 f g/2
    g/8 a g f e/4 c g/8 a g f e/4 c],
  [ \meter<"4/4"> _*8/4 c/4 d e c c d e c e f g/2 e/4 f g/2 ]
}
```

Figure 2.1: Output from *GUIDO NoteServer* displayed in the Konqueror web browser.

Command-line converters between GUIDO and MIDI (*gmn2midi* and *midi2gmn*) also exist.

Development Tools

The *GUIDO Parser Kit* is available for free download. The kit has already been used to implement *NoteServer*, *NoteViewer*, and *NoteAbility*'s GUIDO import/export abilities. Almost all of the work of normalizing (i.e. fully-specifying) a file, such as the automatic inference of `bar` tags from the `meter` tag, is implemented as GUIDO-to-GUIDO transformations in the *GUIDO Parser Kit*, greatly simplifying the task of reading GUIDO files.

2.8.2 Mudela

Applications

Mudela is the official language of the ambitious *GNU Music Project,* which aims to provide a complete suite of open-source music applications including notation typesetting, sequencing, and optical music recognition.

The only *GNU Music Project* application that currently exists in usable form is the music typesetter *LilyPond*. It is a command-line application that converts Mudela input to TEX output which can then be printed or viewed on-screen using standard TEX tools. It currently runs on UNIX and Microsoft Windows, though the Windows version requires large and complex applications that are not commonly installed (i.e. TEX and Python). Unlike the other TEX-based music typesetting packages, *MusiXTEX* (Icking 1997) and *OpusTEX*, which are implemented entirely as TEX macros, *LilyPond* is implemented in a combination of C++ (Stroustrup 2000), Scheme (Abelson et al. 1998), Python (Van Rossum and Drake 2000), TEX (Knuth 1984) and the Mudela language itself.

There is also an interactive graphical editor for Mudela, *Denemo* (Figure 2.2), which is still in the very initial stages of development. Though graphical, the music editing is entirely QWERTY keyboard-based. The keyboard input deliberately bears some abstract resemblance to Mudela.

Development Tools

LilyPond's source code is documented such that it could be used as the basis for parsing Mudela. According to the LilyPond internals document (Nienhuys et al. 2000) there are

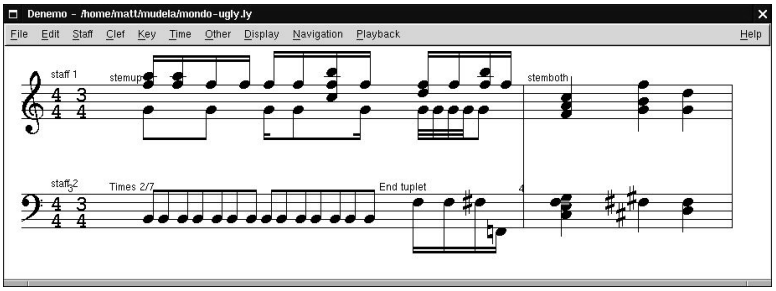


Figure 2.2: *Denemo* music editor.

plans for the Mudela parser to be exposed as a library, meaning that third-party developers could use it in their own projects.

2.9 Conclusion

For an end user typing in the representation directly, Mudela's concise syntax can be learned easily. GUIDO's more verbose syntax can be cumbersome at times. The human issues of entering the representation manually, however, was not a strong consideration for the present system.

The availability of software tools is also an important factor. Presently, the freely available GUIDO tools are not fully functioning, while *LilyPond* is stable and quickly approaching the level of professional-quality output. The commercial GUIDO tools have not been evaluated, but regardless of their design or usefulness, it may be problematic to embark on an open-source project whose only possible interchange is with a commercial product.

From the point-of-view of an implementor, GUIDO is a much more elegant and practical language than Mudela. It is clearly defined and its design ensures language stability even as new extensions are added.

Any decision in this area would be a compromise, but in practice, we have chosen GUIDO as the primary output language of the system, with a strong impetus to keep the system as flexible as possible so that new output formats can be added at a later date.

Chapter 3

Implementation of an optical music interpretation system

This chapter describes the algorithms necessary to perform effective optical music interpretation (OMI).

The execution of the OMI system involves the following steps:

1. **Input.** The bounding boxes, describing the location of each symbol, are input from AOMR and converted into OMI's internal data structure (Section 3.1).

2. **Assembly.** Glyphs, (the combination of a symbol identity and its location on the page), that are broken due to faint or damaged printing are reassembled (Section 3.2).

3. **Sorting.** Each glyph is assigned to a staff and put into temporal order. This new ordering makes many of the reference assignment algorithms more transparent and efficient (Section 3.3).

4. **Reference assignment.** References between glyphs are determined and assigned. This is the core of the OMI process (Section 3.4).

5. **Metric correction.** Rhythmic errors in the optical music recognition (OMR) stage are corrected by examining the metrical context and vertical alignment of glyphs (Section 3.5).

6. **Output.** A representation of the score is output (Section 3.6).

7. **Interactive debugging.** Optionally, the results of the OMI system can be examined in an interactive debugger. The debugger provides the programmer/user with full access to the internals of the system (Section 3.7).

Each phase of execution will be discussed in order in the following chapter.

3.1 Input

3.1.1 XML glyph format

The output from the Adaptive Optical Music Recognition (AOMR) system (Fujinaga 1996) is an eXtensible Markup Language (XML) description (Marsh 2001) of the glyphs identified on the page (Figure 3.1). Each `<glyph>` entry contains:

- **Identifier** defining its type (e.g. `id="['notehead.filled']"`)
- **Bounding box** describing its location relative to the page (e.g. `bbox="(1008, 632, 15, 19)"`)

Other elements output by AOMR but currently unused by OMI include:

- **Classification state** describing whether the glyph was classified by a general classifier, a heuristic classifier or an end user (e.g. `classification_state="3"`)
- **Features** describing characteristics of the glyph, such as moments, number of holes and aspect ratio (e.g. `<aspect_ratio-00 value="0.789473712444"/>`)

Though the current implementation of OMI does not use this extended information, it is hoped that in the future it may help to resolve some ambiguities in scores with recognition errors.

OMI uses tools included with the Python language (Van Rossum and Drake 2000) to read the input file.[1] As each glyph is input, a Python object is created based on its symbol identification. For instance, if a symbol was identified as a `notehead.filled` by AOMR, a new instance of the `notehead_filled` class will be created[2]. The advantage of this approach is that new classes of symbols can be added to the system simply by writing a new Python class. Therefore, there is no need to explicitly register the new class in a prototype database (Gamma et al. 1995) in order for the XML parser to create new glyph instances.

[1] Specifically, these are the bindings to the Expat non-validating XML parser library (Cooper 1999).
[2] The periods (`.`) are converted to underscores (_) since periods can not be used in Python identifiers.

```
<glyph bbox="(1008, 632, 15, 19)"
       classification-state="3"
       id="['notehead.filled']"
       source="Untitled">
  <features>
    <area-00 value="285.0"/>
    <aspect_ratio-00 value="0.789473712444"/>
    <moments-00 value="0.312280714512"/>
    <moments-01 value="0.26795938611"/>
    <moments-02 value="0.0720662251115"/>
    <moments-03 value="0.0931687057018"/>
    <moments-04 value="0.0197625178844"/>
    <moments-05 value="0.0101371835917"/>
    <moments-06 value="-0.0069728018716"/>
    <moments-07 value="-0.00531475618482"/>
    <moments-08 value="-0.0228708032519"/>
    <nholes-00 value="0.0"/>
    <nholes-01 value="0.0526315793395"/>
    <volume-00 value="0.687719285488"/>
    <width-00 value="15.0"/>
    <height-00 value="19.0"/>
    <has_notehead-00 value="0.0"/>
  </features>
</glyph>
<glyph bbox="(1636, 1436, 5, 17)"
       classification-state="3"
       id="['verticalline']"
       source="Untitled">
  <features>
    <area-00 value="85.0"/>
    <aspect_ratio-00 value="0.29411765933"/>
    <moments-00 value="0.36235293746"/>
    <moments-01 value="0.357785463333"/>
    <moments-02 value="0.0231125578284"/>
    <moments-03 value="0.342480778694"/>
    <moments-04 value="0.0264799520373"/>
    <moments-05 value="0.000463378295535"/>
    <moments-06 value="-0.0139860631898"/>
    <moments-07 value="-0.00381278875284"/>
    <moments-08 value="0.586884319782"/>
    <nholes-00 value="0.0"/>
    <nholes-01 value="0.0"/>
    <volume-00 value="0.776470601559"/>
    <width-00 value="5.0"/>
    <height-00 value="17.0"/>
    <has_notehead-00 value="0.0"/>
  </features>
</glyph>
```

Figure 3.1: Some example glyphs from the XML output produced by AOMR.

3.1.2 Glyph list

Other approaches to OMI have used temporal trees (Diener 1989) or labeled graph-rewriting systems (Fahmy and Blostein 1993) as the internal data structure (Section 1.2.2). This system, however, uses only a simple list, sorted in temporal order (Section 3.3). There is also an index using a Python dictionary (hash table). It maps each class name to a list of instances of that class. This allows algorithms working on certain classes of glyphs to find them in the list without traversing the list in its entirety. While these data structures are quite simplistic, they have proven to be an effective and convenient basis for optical music interpretation.

3.2 Assembly

The purpose of the assembly phase is to rejoin glyphs that were separated by poor or eroded printing, or improper binary thresholding (conversion from greyscale to binary) (Figure 3.2).

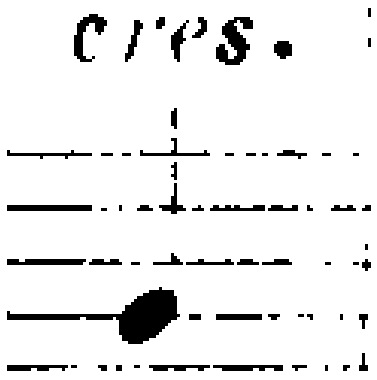


Figure 3.2: This score segment has broken lines that are rejoined by the assembly phase. It has been magnified from its original size.

The algorithm for assembling vertical lines is as follows:

1. For each glyph identified as being part of a vertical line, find the closest glyph that is also identified as a vertical line.

2. If the distance between the two vertical lines is within a certain threshold, join them together.

3. Repeat all steps until no more matches are found.

3.3 Sorting

The purpose of the sorting phase is to put the glyphs into the order that they are read by a musician. This ordering makes many of the algorithms that work upon that data easier to write and maintain and, at the same time, more efficient.

Sorting the glyphs involves the following steps:

1. **Adjusting for multi-page scores** (Section 3.3.1)

2. **Assigning glyphs to staves** (Section 3.3.2)

3. **Grouping staves into systems** (Section 3.3.3)

4. **Grouping staves into parts** (Section 3.3.4)

5. **Temporal sorting** (Section 3.3.5)

3.3.1 Handling multi-page scores

Most scores are made up of multiple pages. One of the difficulties when interpreting multi-page scores is that contextual information, such as clefs and time signatures, must carry over from one page to the next. It turns out that 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 (Figure 3.3). The original page number is saved with each glyph, so that at the output stage any positional information can be readjusted to be relative to its original source page. This way, multi-page scores are not a special case—they can be interpreted exactly as if they were printed on a single page.

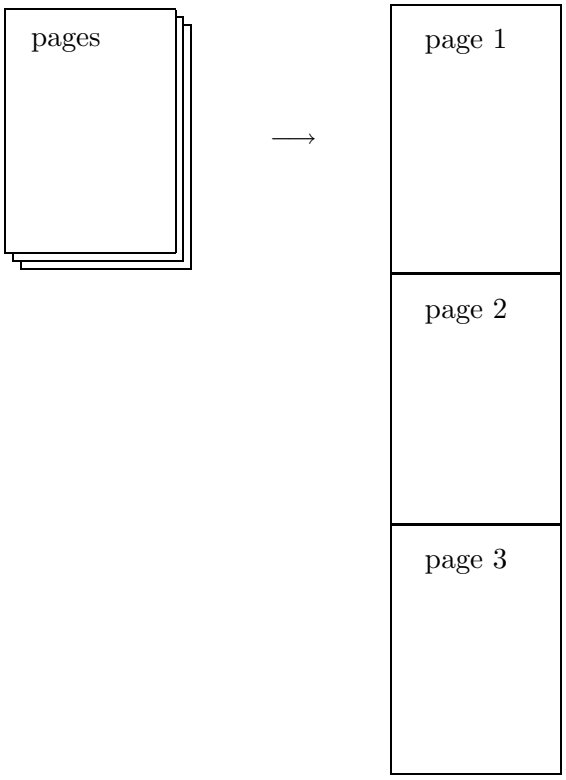


Figure 3.3: Multiple page scores are handled by turning the individual pages into one large page.

3.3.2 Assigning glyphs to staves

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. This reference is determined in three steps:

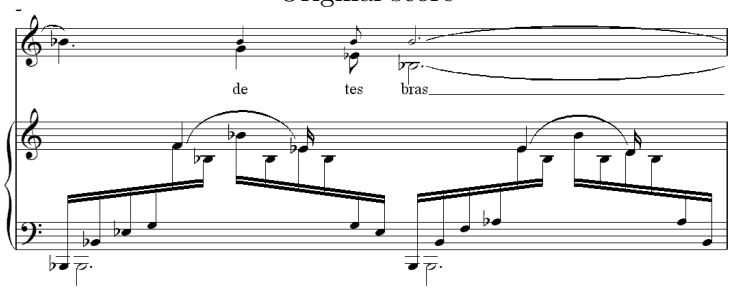
1. Each glyph that overlaps a staff is assigned to that staff. If the glyph overlaps multiple staves, copies are created and each copy is assigned to one of those staves.

2. Each remaining glyph that overlaps one or more assigned glyph(s) uses its staff assignment as its own. If it overlaps multiple glyphs that overlap multiple staves, it is copied, and each copy is assigned to a different staff.

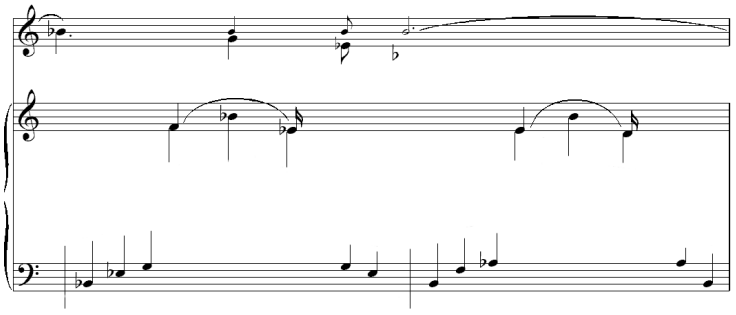
3. Each remaining unassigned glyph is assigned to the closest staff by vertical distance.

This process is applied to a real musical example in Figure 3.4.

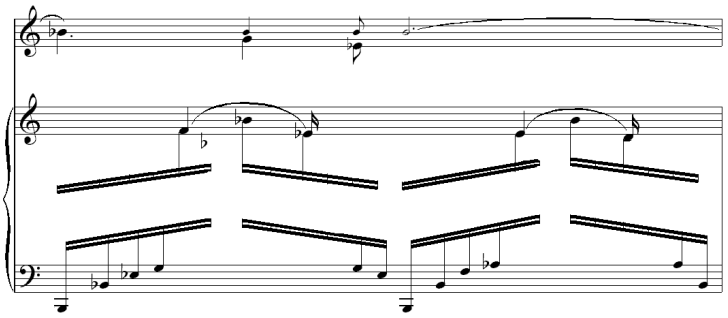
Original score



Glyphs assigned after staff assignment step 1



Glyphs assigned after staff assignment step 2



Glyphs assigned after staff assignment step 3

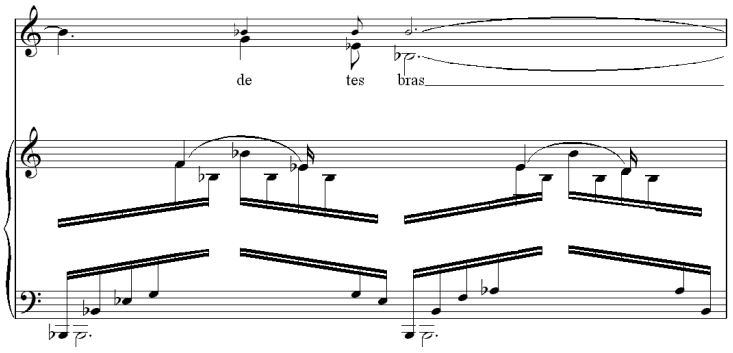


Figure 3.4: The three steps involved in assigning glyphs to staves. Note that the beams are copied and assigned to both the left- and right-hand piano parts. (Duparc, H. *Chanson Triste*.)

3.3.3 Grouping staves into systems

Once glyphs have been assigned to staves, those staves need to be grouped into systems. Each system is a set of staves that are performed in parallel. The grouping is determined by examining how they connect: any staves that share the same glyphs are grouped together. For example, a bracket or brace glyph on the left hand side of the score serves to group staves together into a system. Any cross-staff beaming, such as in keyboard scores, will also group staves together (Figure 3.4).

3.3.4 Grouping staves into parts

Once staves are grouped into systems, each staff is assigned to a *part*. For our purposes, a part is analogous to the set of staves across all systems that is played by a particular instrument, (with the exception of keyboard and harp parts, which have left-hand, right-hand and sometimes pedal parts on separate staves.) In MIDI terminology, a part is analogous to a channel.

Presently, the part assignment algorithm is quite simple: Parts are numbered by starting at the bottom of each system and going upwards. For example, the last staff in each system will make up one part; the next staff up makes up the next part, and so on. This approach works quite well on most keyboard-vocal music, where the singer's staff is sometimes omitted but the keyboard staves are always at the bottom. It breaks down, however, with more complex scores, such as full orchestral scores, when staves are removed or combined within systems to preserve space. Properly interpreting such scores would require reliable optical character recognition (OCR) of the staff names in the margins.

3.3.5 Temporal sorting

The temporal sort function puts the glyphs in musical order. Glyphs are sorted first by part, then voice (see Section 3.4.4), 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 some interesting properties that can be taken advantage of:

- Most inter-related glyphs, such as `NOTEHEAD`s and `STEM`s, 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.
- `part` glyphs mark the end of the entire piece for each part.
- This ordering is identical to that used in most musical description languages, including GUIDO, Mudela and MIDI (MIDI 1986), and therefore output files can be created with a simple linear traversal of the list.

The sorting itself is performed using Python's built-in quicksort implementation using a custom sorting predicate.

3.4 Reference assignment

The purpose of this phase is to build the contextual relationships between glyphs to fully obtain their musical meaning. This is where the bulk of the OMI processing is performed. 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 3.5).

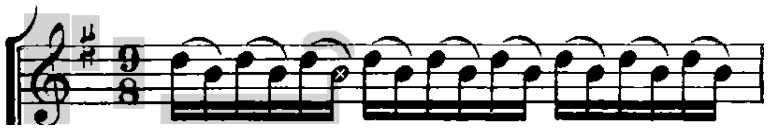


Figure 3.5: References to other glyphs (shaded in grey) are required to fully determine the meaning of a notehead (marked by ×).

Reference assignment proceeds in a number of subphases:

- **Pitch.** Creates the references needed to determine the pitches of pitched glyphs (Section 3.4.2)
- **Duration.** Creates the references needed to determine the durations of durational glyphs (Section 3.4.3)
- **Voice.** Splits multi-voiced parts into separate voices based on stem direction (Section 3.4.4)
- **Chord.** Builds chords within each voice (Section 3.4.5)
- **Articulation.** Assigns articulations to glyphs (Section 3.4.6)
- **Text.** Assigns lyrics to notes and constructs and identifies titles and other textual information (Section 3.4.7)

3.4.1 Class hierarchy

All glyph classes are members of an object-oriented class hierarchy[3] in the style promoted in the Eiffel programming language (Meyer 1997) and used throughout the Java Foundation Classes (Sun Microsystems 1998). In this style, most abstract subclasses can be described 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 present system simply by combining the functionalities of existing classes. This style of programming is easy and natural in Python. 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 run-time type inspection and type modification are possible but much less convenient. By local convention, abstract classes are in `UPPERCASE` and concrete classes are in `lowercase`. All of the reference assignment operations described below make extensive use of this class hierarchy.

3.4.2 Pitch

Classes for pitch

OMI has a three-tiered hierarchy of pitch (Figure 3.6). This is an extension of the more standard binomial pitch representation (Brinkman 1990). Each level adds more detail and requires more information (i.e. references to more classes of glyphs) in order to be fully specified. The 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.

On staff line. The most abstract of the levels of pitch information, `ON_STAFF_LINE`, is the subclass of all glyphs whose vertical placement on the staff is meaningful (e.g. notes, rests, and accidentals.) Perhaps confusingly, `ON_STAFF_LINE` is the parent class of glyphs on both staff lines and staff spaces. Even numerals are subclasses of `ON_STAFF_LINE` so that they can be combined to form time signatures. In order to determine the staff line placement, the glyph must have a reference to a staff.

[3] Meyer (1997) provides a good introduction to the object-oriented programming concepts presented here.

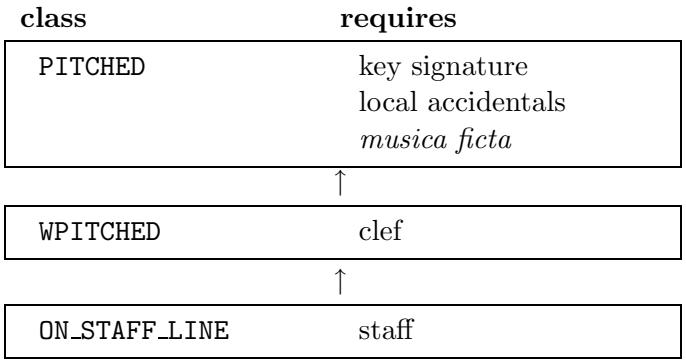


Figure 3.6: The pitch hierarchy. Each layer adds more detail, but requires references to more types of glyphs.

White pitched. The next level, `WPITCHED`, is the subclass of all glyphs that have a white-key pitch. White key pitch corresponds to a note letter. For example, c and $c^\sharp$ have the same white pitch, but a different twelve-tone pitch. White pitched glyphs include notes and accidentals. In order to determine white pitch, the glyph must have a reference to a clef.

Pitched. The outer-most level, `PITCHED`, is the subclass of all glyphs that have a fully specified pitch. The pitch is determined by the two other layers, (i.e. the vertical placement on the staff and its white pitch) as well as the key signature and any accidentals or *musica ficta* accidentals that may be attached. If no accidentals are present, the pitch is assumed to be the natural (♮) version of its white pitch.

Clefs. Clefs change the way in which the placement of notes on the staff are converted into pitch. The treble and bass clefs perform this conversion in a direct way. However, the vertical placement of the c-clef is meaningful to determining the pitches of following notes. It is therefore a subclass of `ON_STAFF_LINE` (Figure 3.7).

Accidentals. Accidentals include sharps, double-sharps, flats, double-flats, and naturals. They can appear in at least three contexts:

1. As part of a key signature
2. As local accidentals: they affect all notes of the same white pitch to the right of them in the same measure

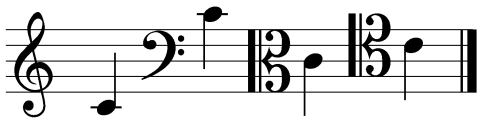


Figure 3.7: Different clefs (treble, bass, alto and tenor), and the placement of middle-*c*. Note that the alto and tenor clefs, collectively known as *c*-clefs, are visually identical, but their meaning is determined by their placement on the staff.

3. As *musica ficta* accidentals: they appear above the staff and affect the note directly below them

To prevent accidentals from being interpreted in multiple contexts, each context is examined in order and those found to be meaningful in that context are marked so they are not used in proceeding contexts.

Algorithms for pitch

There are a number of algorithms that create the references necessary to determine pitch:

- **Ledger line counting and adjustment.** Count ledger lines above and below notes outside of the staff
- **Clef assignment.** Assigns clefs to white pitched glyphs
- **Key signature building and assignment.** Builds accidentals into key signatures and assigns them to pitched glyphs
- **Local accidental assignment.** Assigns accidentals to pitched glyphs using an *accidental comb*
- ***Musica ficta.*** Assigns *musica ficta* accidentals, found above the staff, to pitched notes

The more interesting ones, ledger line counting, key signatures, and accidentals, are discussed below.

Ledger line counting and assignment. Determining the correct staff line location of notes on the staff is relatively easy, since most (though not all) scores have relatively parallel staff lines. In fact, the staff line of the note can be determined by a simple distance

calculation from the center of the staff. However, notes outside of the staff, which require the use of ledger lines (short horizontal lines), are often placed very inaccurately in hand-engraved scores (Figure 3.8).

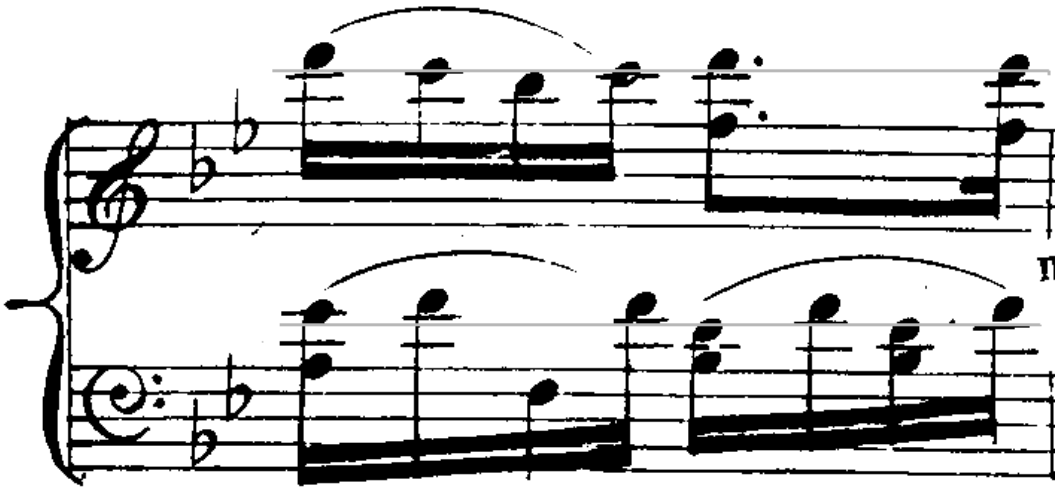


Figure 3.8: An example of poorly aligned ledger lines. The grey lines are parallel to the staff lines and were added for emphasis.

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. While this approach works most of the time, sometimes the ledger line glyphs are missed by the AOMR system. In this case, it is possible to get wildly inaccurate results. Therefore, a check is performed: if the pitch by distance from the center of the staff is very different from the pitch by ledger line counting, the average of the two is taken. While this approach is somewhat arbitrary, it has proven to be very effective in practice.

Key signature. Every `PITCHED` glyph needs a reference to the key signature most recently preceding it. This referencing is done part by part, so that transposing scores, where each part might have different key signatures, are handled correctly. (The actual transposition of parts is a performance issue and not a notational one, and therefore it is not a part of OMI.)

The difficulty of this algorithm arises in distinguishing between key signature accidentals and local accidentals, which are symbolically identical. In general, the rules that define a key signature are:

- Key signatures can only begin directly following a clef or a barline.
- The individual accidentals in the key signature must follow a rigidly defined order:
  - *f, c, g, d, a, e, b* for sharps
  - *b, e, a, d, g, c, f* for flats

The key signature-finding algorithm itself is implemented as a finite-state automaton (FSA) (Kelley 1995) with four states (Figure 3.9):[4]

- **Finding clefs or barlines.** This state looks for legal places for a key signature to begin (i.e. a clef or barline). Once arriving at a legal starting place, one can start building a new key signature by moving to the next state.
- **Finding the start of key signatures.** In this state, the algorithm is looking for a legal first accidental to a key signature (i.e. $b^\flat$ or $f^\sharp$). If one is found, the algorithm proceeds to find more accidentals of the same kind. If anything else is encountered, the state gives up and returns control to the first state.
- **Finding more flats.** This state looks for more flats in the required sequence. As `flat` glyphs are correctly identified to be part of a key signature, their class is changed to `flat_in_key_sig`. This marks them so they will not be used as local accidentals later on by the `local_accidental` algorithm. If an accidental that does not belong to the key signature or a durational glyph is encountered, the building of the key signature stops and the algorithm returns to the first state.
- **Finding more sharps.** This state is analogous to the **finding more flats** state.

There are some cases in which there is ambiguity around the last accidental in a key signature. For example, a piece in F major (key signature containing only a $b^\flat$) with a starting note of $e^\flat$ might erroneously be interpreted as a piece in the key of $B^\flat$ major (two flats). This problem could be dealt with by examining other parts or systems, but that can be dangerous when there are transposing parts or key signature changes throughout the piece.

[4]For an introduction to finite state automata (FSA), see Kelley (1995). The circles represent states. Different inputs in a stream cause transitions between states (indicated by the arrows). An arrow marked with a * represents "all other inputs."

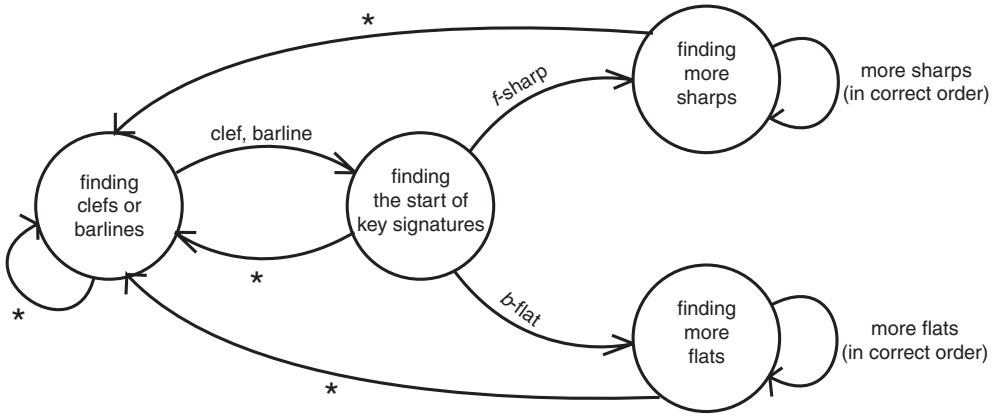


Figure 3.9: The finite-state automaton (FSA) used to locate and build key signatures.

Local accidentals. The purpose of this algorithm is to assign local accidentals (i.e. accidentals not in key signatures) to notes.

First, the correct white pitch of all accidentals must be determined. As shown by the ledger line counting problem, glyphs outside of the staff cannot determine their pitch based on distance from the center of the staff. Therefore, noteheads have to make use of a ledger line-counting algorithm. However, accidentals do not have ledger lines, and therefore their pitch must be determined by their proximity to noteheads to the right. For each accidental outside of the staff, the staff line location is inherited from the closest notehead to the right of the accidental.

To assign accidentals to pitched glyphs, a linear pass is made through the list with an *accidental comb*. The *comb* is a list of seven elements, one for each possible white pitch (*c*-*g*). When an accidental is encountered, it is placed in the comb at its corresponding white pitch. When a pitched glyph is encountered, it is referenced to the accidental at the corresponding white pitch in the comb.

Certain glyphs, such as barlines, system breaks and part breaks clear the effect of accidentals (since accidentals do not carry from measure to measure.) For convenience, they are all subclasses of an abstract class, `CLEARS_ACCS`. When these glyphs are encountered, the comb is emptied, with the exception of accidentals that are being held over by ties.

3.4.3 Duration

Classes for duration

Like pitch, the concept of duration is implemented in multiple layers (Figure 3.10).

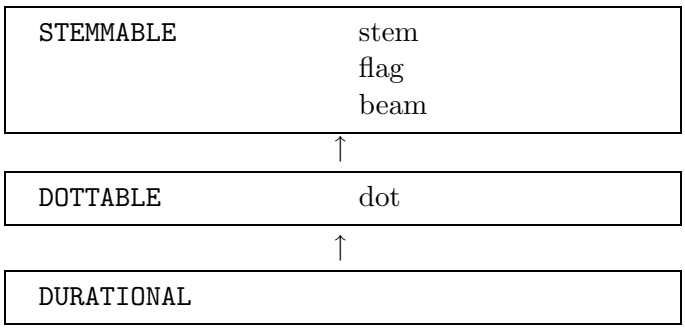


Figure 3.10: The durational hierarchy. Each layer adds more detail, but requires references to more types of glyphs.

Durational. The most generic of these layers, `DURATIONAL`, is the subclass of all glyphs that have duration. The duration itself is stored as a rational (fractional) number. Rational numbers are implemented as a class containing a numerator and denominator. Operations upon `Rational` objects preserve the full fractional precision. If this were not the case, rounding error might be introduced when durations that do not divide evenly in a decimal or binary representation, such as triplets, are used. For example, three quarter-triplets (represented as $\frac{1}{6}$) should always be equal in length to two quarters (represented as $\frac{1}{4}$).

$$3\left(\tfrac{1}{6}\right) \equiv 2\left(\tfrac{1}{4}\right)$$

Dottable. The next layer, `DOTTABLE`, is the subclass of all glyphs that can have their duration lengthened by augmentation dots. Each augmentation dot increases the duration of the durational by 50% (Section 3.4.3).

Stem. `STEM` is the subclass of all stems. Stems help to determine the duration of a notehead based on the shape of the notehead itself and any beams or flags attached to the stem.

Since stems are symbolically identical to barlines, they can be confused by OMI. 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 vertical lines remaining that are taller than the height of one 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. They are not currently dealt with.

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 3.5).

Stemmable. `STEMMABLE` is the subclass of all glyphs that can have a stem attached. This includes quarter and half note heads.

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 has problems with chords containing second (stepwise) intervals, since some of the noteheads are forced to the other side of the stem (Figure 3.11). Presumably, the stem direction could be determined by selecting a single notehead and passing its stem direction attribute to all other noteheads in the chord. However, selecting that notehead is not straightforward. As shown in Figure 3.11, this would not be simply a matter of selecting the top or bottom notehead. According to Read (1969), the *outside* notehead, beyond which the stem does not vertically extend, should never be on the "wrong" side of the stem. Therefore, both the top and bottom noteheads are examined. If the stem goes beyond the notehead, well outside of the chord, that notehead is the *inside* notehead, and the opposite notehead, by definition, is the *outside* notehead. The *outside* notehead is used to determine stem direction by the normal method and its stem direction is inherited by all other noteheads in the chord.

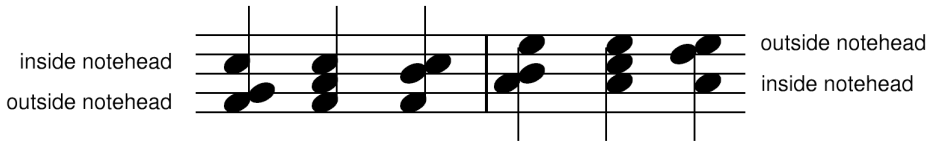


Figure 3.11: Chords containing seconds pose a problem for determining stem direction since noteheads are pushed to the other side of the stem.

This approach breaks down with some older scores where half note heads are on the wrong side of the stem. This is rarely a problem, however, since stem direction is only meaningful to OMI in multi-voiced scores (Section 3.4.4).

Flag. The `FLAG` class is the subclass of all flags. Each flag attached to a stem divides its duration by half.

Beam. `BEAM` is the subclass of all beams. Each beam attached to a stem divides its duration by half.

`BEAM` is also the subclass of all glyphs that look like beams, such as `multiplerest`. That way, any `multiplerest`s that are in fact `beam`s will be found by the beam algorithm and attached to stems. Any `multiplerest`s that are not attached to stems, will retain their meaning as multiple-measure rests.

Algorithms for duration

There are a number of algorithms related to duration:

- **Augmentation dots.** Assigns dots to durationals
- **Stems.** Assigns stems to noteheads
- **Beams.** Assigns beams to stems
- **Flags.** Assigns flags to stems
- **Time signature construction.** Builds a time signature out of two numerals
- **Time signature assignment.** Assigns time signatures to barlines
- **Barline construction.** Converts two consecutive barlines into a double barline

The more interesting ones are described below:

Augmentation dots. Assigning dots to DOTTABLEs requires some flexibility. In properly typeset music, dots never appear directly on a staff line, but instead move to the nearest possible space (Figure 3.12). Therefore, dots are matched to DOTTABLEs by their staff line value within a range of ± 1 of the DOTTABLE's own staff line value.

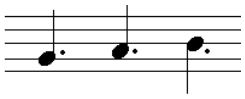


Figure 3.12: How placement on the staff and stem direction can affect the placement of the corresponding augmentation dot.

When a dot is assigned to a DOTTABLE, its class is changed to augmentation_dot so that it is not used later as a *staccato* articulation (Section 3.4.6).

Stems. Stems are assigned to STEMMABLEs by intersection. The intersection is loosened by a constant equal to the average stem width, because many stems do not intersect perfectly with noteheads.

The case where a STEMMABLE has two stems (Figure 3.13) should be logically two separate notes: one belonging to the upper voice and one belonging to the lower voice. To handle this, the STEMMABLE (notehead) is copied. One copy gets a reference to the up stem, the other to the down stem.

Figure 3.13: A notehead with two stems is separated into two notes.

Time signature construction. Time signatures come out of the AOMR system in two forms: either as one glyph containing both numerals, (e.g. classes _4_4 and _6_8), or as two separate numerals. The first case is handled directly. For example, class _4_4 ($\frac{4}{4}$ time signature) is a subclass of TIME_SIG. In the second case however, one must examine the positions of numerals and combine them.

In common music notation, time signatures are always made up of a numerator and a denominator: the numerator digits lie between the first and third staff lines, and the denominator digits lie between the third and fifth staff lines.

This algorithm examines all pairs of numerals. First, numerals that are horizontally close together are grouped. This takes care of cases where either the numerator or the denominator is a multi-digit number (e.g. $\frac{4}{16}$). Then, any numerals that fall between the correct staff lines and are aligned are grouped into a special class, `constructed_time_sig`. A `constructed_time_sig` can then function as a regular complete time signature.

3.4.4 Voices

This subphase deals with multi-voice scores, where multiple voices appear on one staff. Multi-voicing is often seen in choral music or compressed orchestral scores.

Voiced class

`VOICED` is the subclass of all glyphs that exist only in one voice of a multi-voiced staff. (This should not be confused with the terms *voiced* and *unvoiced* in reference to notes vs. rests.) This includes all notes, and also rests that are not centered vertically on the staff. All glyphs that are not subclasses of `VOICED` apply to or exist in both voices of a multi-voiced staff. This includes things like barlines and accidentals which are not specific to a given voice.

Voices algorithm

In multi-voiced scores, each part is divided into two voices. Each voice can then be treated thereafter as a separate part, even though visually it shares a staff with another voice. This arrangement effectively abstracts out the concept of voices to the part level and makes any further processing much simpler because multi-voiced and single-voiced staves do not need to be handled differently (Figure 3.14).

To split the part into voices, all `VOICED` glyphs are assigned to one of the voices. For notes this is determined by stem direction, and for rests this is determined by vertical placement. Non-`VOICED` glyphs are copied and placed in both voices. In this way, `BARLINE`s and other global glyphs will appear in both voices and the voices can be completely independent of each other.

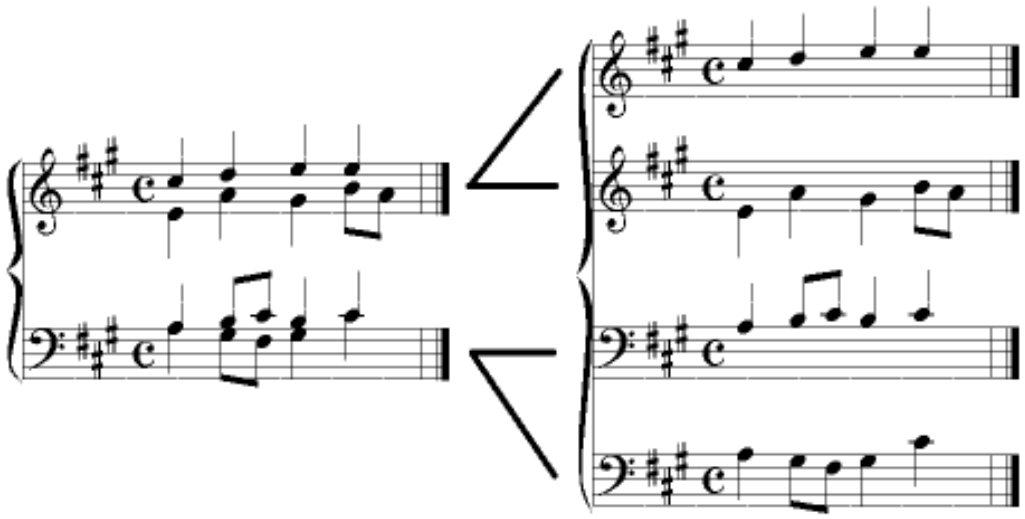


Figure 3.14: Multi-voiced staves are split into two separate voices.

Determining whether to split a given part is determined automatically. The user does not need to tell the system that a score is multi-voiced. Each measure is examined individually. If it contains two glyphs at the same vertical position that have different stem directions, that entire measure will be split. Otherwise, everything in the measure is assigned to the first voice, regardless of stem direction. This leaves the second voice with an empty measure. The **readjust** algorithm in the metric correction phase will fill the second part with empty duration so that both the measures in both voices will have the same duration (see Section 3.5.3).

One shortcoming of this approach is that the system does not handle the separation of parts into more than two voices, as this would require examining more than just stem direction. Any possible solution would not be trivial. Fortunately, three or more voices on a staff is quite rare in the Levy collection.

3.4.5 Chords

Chords are defined as groups of notes occurring simultaneously in the same voice. Unlike voices, the notes are logically one unit, typically played by one performer (Figure 3.15).

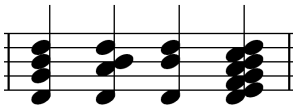


Figure 3.15: Some example chords.

Classes for chords

Chordables. `CHORDABLE` is the subclass of all glyphs that can be combined to create chords. This includes all noteheads.

Chords. A `CHORD` instance contains a list of `CHORDABLE` objects that make up a chord (i.e. occur simultaneously in the same voice).

Chords algorithm

The algorithm goes through all `CHORDABLE`s, finding groups of them that are aligned vertically or intersecting one another. For each group, a new `CHORD` object is created containing a list of its respective `CHORDABLE`s. The `CHORDABLES` are then be removed from the main glyph list.

The overall duration of the chord is set to be equal to the duration of the majority of noteheads in the chord. This corrects some errors where some noteheads are not properly connected to their stem.

3.4.6 Articulation

An articulation, for the purposes of OMI, is defined as any glyph directly above or below another glyph that adds additional properties to that note (Figure 3.16). This definition is somewhat broader than the traditional definition, since it includes things such as hairpin *crescendi*.

Classes for articulations

Articulations. `ARTICULATION` is the subclass of all articulations. This includes glyphs such as accents, *staccato* dots, *fermati*, and dynamic markings.

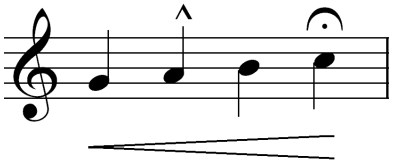


Figure 3.16: Some example articulations. Note that the *fermata* and accent apply only to one note, whereas the hairpin *crescendo* applies to a range of notes.

Range articulations. `ARTIC_RANGE` inherits from `ARTICULATION` and is the subclass of all articulations that cover a range of glyphs. This includes hairpins and slurs.

Articulatables. `ARTICABLE` is the subclass of all glyphs that can have an articulation assigned to them. This includes noteheads and rests.

Articulations algorithm

The algorithm steps through each `ARTICULATION` and finds all `ARTICABLES` that are underneath it in the same staff. Each of this `ARTICABLES` is given a reference to the `ARTICULATION`.

3.4.7 Text

The text subphase is currently unimplemented, as there has not been adequate experimentation with a reliable optical character recognition (OCR) system. Recent experiments using the AOMR system to perform OCR have shown some promise.

In common music notation, text appears in a number of different contexts. For example, tempo markings, dynamic expressions, performance instructions, fingerings, and lyrics. In most published music, each of these types of text are usually set in a different typeface and in a different position on the staff. The exact typeface and placement varies between publishers and between different types of scores from the same publisher. Determining the correct context for blocks of text may prove to be a difficult problem.

3.5 Metric correction

3.5.1 Overview

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 can become quite serious, since they are accumulated over time, and parts will become more and more 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 (Figure 5)[5].

The score is examined, one measure at a time, across all parts simultaneously. Each part in the measure is classified into correct, short and long based on its metric duration and its width. The score is assumed to not be multi-metric (i.e. has the same time signature across all parts). Based on this information, different algorithms are performed on the measure to correct durations of notes and rests and barline placement. The primary goal of the correction algorithms is to ensure that the length of the measure across all parts is the same before moving to the next measure, and to make those corrections in the most intelligent way possible.

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, as will be shown, many of the algorithms can be applied to improve the accuracy of single-part scores as well.

3.5.2 Classification

Each measure is read in by reading all parts in parallel until the next barline is encountered. Next, the parts are classified. Classification helps the metric corrector to decide which algorithms are applicable. The parts are classified by two separate criteria:

- **Duration.** The sum of all of the contributing durations in the measure in a given part
- **Width.** The spatial width of the measure in a given part on the page

Therefore, duration relates to time, and width relates to space. Within each criteria, the parts are classified into three categories: correct, short, and long.

[5]Some items in the Levy Collection are improperly typeset and do not have this property. In this case, metric correction is likely to generate errors, and is therefore automatically turned off.

- Correct duration is defined to be equal to the time signature. If a time signature is not present, the duration of the majority of parts is taken as the correct length. If no majority exists, the part with the longest duration is assumed to be correct. There is always at least one part with a correct duration that can be used as a yardstick for all other parts. Among other things, this provides robust handling of pickup measures.
- Correct width is defined to be the width of any part with a good duration. Parts that have an incorrect duration will have their width determined relative to a part with a correct duration.

3.5.3 Algorithms for metric correction

Each metric correction algorithm is tried in order, moving on to the next measure when all parts have the same duration. Therefore, if all parts have the same duration to begin with, no metric correction algorithms are tried. Table 3.1 lists the metric correction algorithms and their applicability to different types of incorrect measures.

Algorithm	**Duration**		**Width**	
	short	long	short	long
Measure of rest. Fixes measures containing only a single rest.	•	•		
Whole rest/half rest conversion. Exchanges misread whole/half rests.	•	•		
Erroneous dot removal. Removes noise or dust that was misinterpreted as augmentation dots.		•		
Barline to stem. Converts barlines to stems and attempts to assign them to noteheads.	•		•	
Splice. Makes up for missed barlines by cutting long measures into shorter ones.				•
Readjust. Changes durations of individual notes so that they line up metrically with other glyphs that line up vertically.	•	•		
Extend. Adds dummy rests to the end of measures so that the duration of each part is the same.	•	•	•	•

Table 3.1: The metric correction algorithms and their applicability.

The algorithms are described in detail below.

Measure of rest

It is quite common for typesetters to print a single whole rest to indicate an empty measure regardless of the time signature (Figure 3.17).

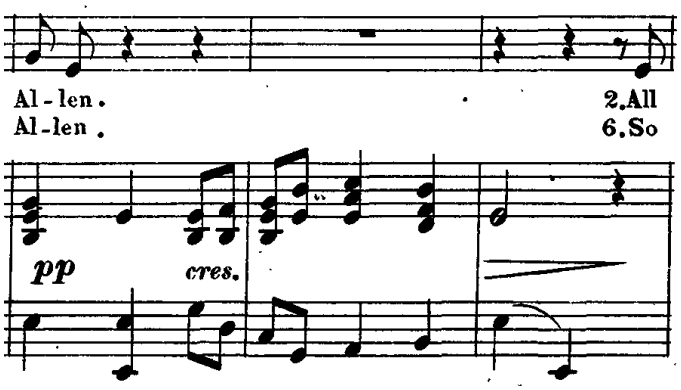


Figure 3.17: The duration of whole rests is determined by the time signature.

The **measure of rest** algorithm deals with measures containing only a single half or whole rest. This algorithm replaces the rest with a rest equal to the length of the time signature.

This algorithm does not require a metrically good measure to be available and therefore works on single-part scores.

Whole rest/half rest conversion

Since whole rests and half rests are visually identical, and their vertical placement on the staff can often not be determined accurately, AOMR cannot distinguish them. This algorithm will replace one with the other, but only if it makes sense to do so. The rest is replaced and then the measure is verified by ensuring that the glyphs following the rest are at the same metric position as glyphs vertically aligned to them in other parts. If this does not prove to be correct, the change is undone.

If the rest is the last glyph in the measure, the change is much easier. We only have to know that the overall duration of the measure is under or over by a half note to know that the change is legal.

This algorithm does not require a metrically good measure to be available and therefore works on single-part scores.

Erroneous dot removal

Damaged or degraded scores often have noise that AOMR interprets as dots. This can cause some durationals to be augmented that shouldn't be.

This algorithm removes the `DURATIONAL`'s reference to the dot and then verifies the change.

This algorithm may seem to be, and in many cases is, redundant with the **readjust** algorithm. However there are cases when the **erroneous dot removal** algorithm results in a correct solution when **readjust** fails (Figure 3.18). Also, this algorithm does not require a metrically good measure to be available and therefore, unlike the **readjust** algorithm, works on single-part scores.

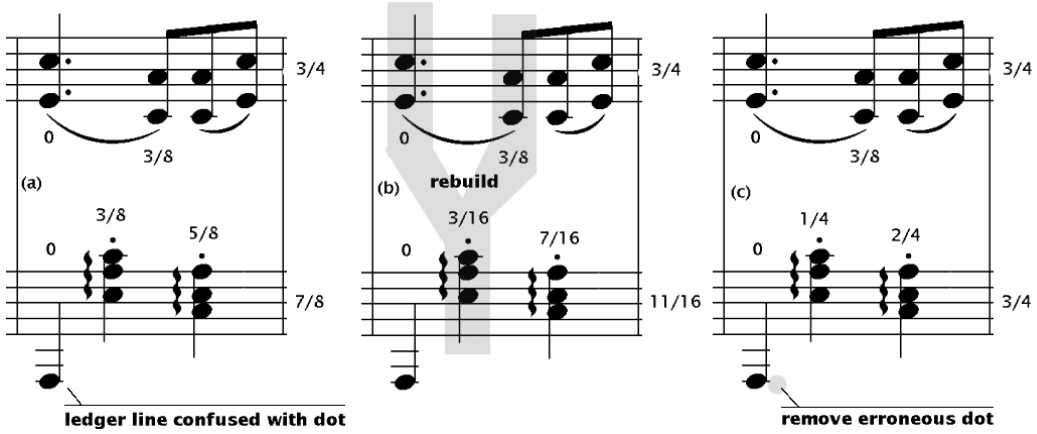


Figure 3.18: This illustrates the necessity for the **erroneous dot removal** algorithm. (a) The original as read in from OMR. The ledger line on the first note in the left hand part was misread as an augmentation dot. (b) The result of the `rebuild` algorithm. Note that the second chord in the bass part is wrongfully placed at $\frac{3}{16}$. (c) The result of the **erroneous dot removal** algorithm. The second chord in the left hand part is rightfully placed at $\frac{1}{4}$.

Barline to stem

Since barlines and stems are both vertical lines, they are considered to be the same glyph by AOMR. While most of this can be resolved by fairly straightforward methods (Section 3.4.3), some can still be incorrectly identified by the metric correction phase. If a stem in the middle of a measure is thought to be a barline, the width of the measure will be too narrow. Vice versa, if a barline is assumed to be a stem, the measure will be too long. This algorithm takes care of the former case. The **splice** algorithm takes care of the latter.

If the measure is spatially narrow, the algorithm searches for a notehead very close to the barline. If it finds one, it is likely that the barline is in fact a stem. The class of the barline is changed to `stem` and the result is verified (Figure 3.19).

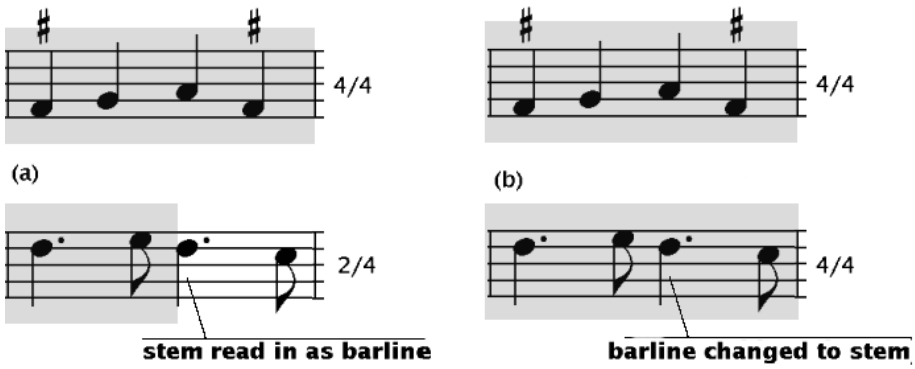


Figure 3.19: The shading represents the areas perceived to be part of the measure. (a) The original interpretation, in which a stem is read as a barline. The **barline to stem** algorithm converts the barline to a stem and results in (b).

Splice

Sometimes barlines are miscategorized as stems, or are missed by AOMR altogether. This causes the measure being read in to be spatially wider than necessary. The **splice** algorithm has two ways of solving the problem (Figure 3.20):

- The first method copies a barline from a metrically good part into the part that seems to be missing a barline. This change is verified.

- The second method inserts a new barline at the moment where a barline would make a complete and correct measure. Since this does not require a metrically good measure to be available, it also works on single-part scores.

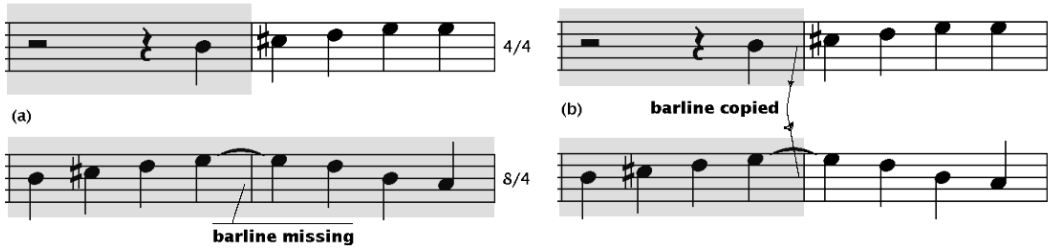


Figure 3.20: Splice algorithm. The shading represents the area perceived to be part of the measure. (a) is the original input with the missing barline. (b) is the result of the **splice** algorithm copying the barline from the top part to the bottom part.

Readjust

The **readjust** algorithm is the most versatile and solves the greatest range of problems. It examines parts with an incorrect length and attempts to correct the durations of individual notes based on their alignment with notes in other parts.

It takes advantage of musical typographic convention: notes that occur at the same metric instance are vertically aligned across all parts in a system. (This is not always the case, but it occurs often enough that it should work in most situations.) If vertically aligned notes are not at the same metric moment, there has likely been an error in the AOMR and corrections should be made. Note that this algorithm does not determine durations from the relative horizontal spacings of notes. The relative horizontal spacing of notes is often designed more for visual than metric considerations (Read 1969), and therefore is unreliable.

Readjust works by comparing notes in the incorrect part to notes in all other parts known to be correct. If a note is found that is visually aligned but not at the same metric position, the duration of the note preceding it is changed in order to move the misplaced note into the same metric position as those visually aligned to it (Figure 3.21). If there is no preceding note, empty space is inserted.

Often a note will be metrically misplaced but there are no notes in any correct parts that are visually aligned to it that can be used as guideposts. In such a case, the correct

metric placement is determined by measuring the distance between itself and the closest good notes to either side of it and estimating the metric position (Figure 3.22(b)).

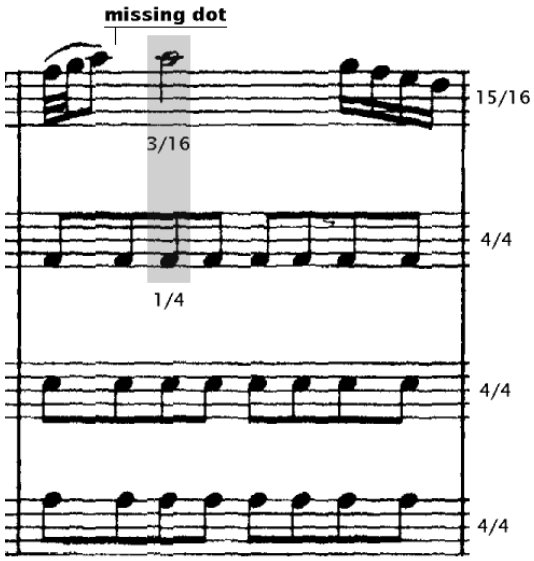


Figure 3.21: The **readjust** algorithm corrects for missing dots. The length of the eighth note in the first part is lengthened so the half note now occurs on the correct beat $\left(\frac{1}{4}\right)$.

Extend

The **extend** algorithm is intended to be a last resort case. All other, smarter, algorithms have already been tried and have been unable to bring all parts to a correct measure length. **Extend** simply ensures that all parts have the *same* measure length, even if this is incorrect and not equal to the time signature. This prevents the errors in this measure from propagating into future measures. It also does not ensure that vertically aligned notes will be metrically aligned. Though this is not an ideal solution, it ensures that the worst case possible are isolated incorrect measures, as opposed to an entirely misaligned score.

First, the longest duration across all parts is found. Empty rests are added to the end of all metrically shorter parts to bring them up to the duration of the longest.

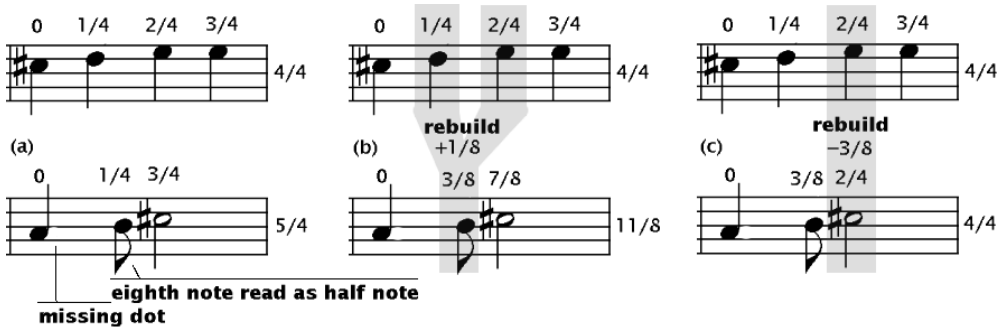


Figure 3.22: This figure shows two applications of the **readjust** algorithm in the same measure. (a) The original interpretation. In (b), the eighth note in the left hand part is moved between two quarter notes in the right hand part. In (c) the eighth note is shortened to move the half note into the correct metrical position.

3.6 Output

3.6.1 File formats

As illustrated in Chapter 2, there is no clear standard for symbolic musical representation. 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 (Hoos and Hamel 1997) is implemented, but other musical representation languages such as Lilypond Mudela (Nienhuys and Nieuwenhuizen 1998) are planned. Standard MIDI file-format (MIDI Manufacturers Association Inc. 1986) is currently supported through a third party tool that converts GUIDO to MIDI (Martin and Hoos 1997).

3.6.2 Pluggable back-ends

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. After re-ordering, the output function of each glyph is called. The output functions

are implemented as mixin classes (Bracha and Cook 2000) in the pluggable back-end and merged into the core glyph classes.

3.6.3 Mixin classes

Mixin classes are incomplete class definitions that add members to a core class. Exploiting the dynamic nature of Python, the mixin-merging process is performed at run-time. The matching of the core class to the mixin class is determined by their names (in Python, their `__name__` member). For a concrete example, consider the class definitions in Table 3.2. 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 effectivly adds 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.

Core class	**Mixin class**

`class CLEF(CLEARS_ACCS, BASE):`
`  middle_line = B`
`  octave = 1`
`  key_sig = None`
`  fixed_name = "treble"`
`  def __repr__(self):`
`  # etc...`
`  def get_wpitch(self, staff_line):`
`  # etc...`
`  def get_octave(self, staff_line):`
`  # etc...`
	`class CLEF(GUIDO):`
	`  def bas_guido_clef(self):`
	`  # etc...`
	`  def bas_guido(self):`
	`  # etc...`

Table 3.2: An example of a core and mixin class.

The primary advantage to this approach is extensibility. When new classes are added to the core heirarchy, they do not need to be updated in all output extension modules, they will simply be ignored by the mixin application function.

3.6.4 Demonstration of output

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.

Original image

The original image (Figure 3.23) was scanned at 300 DPI, 8-bit grayscale. Note that there is a fair amount of noise due to age of the score.

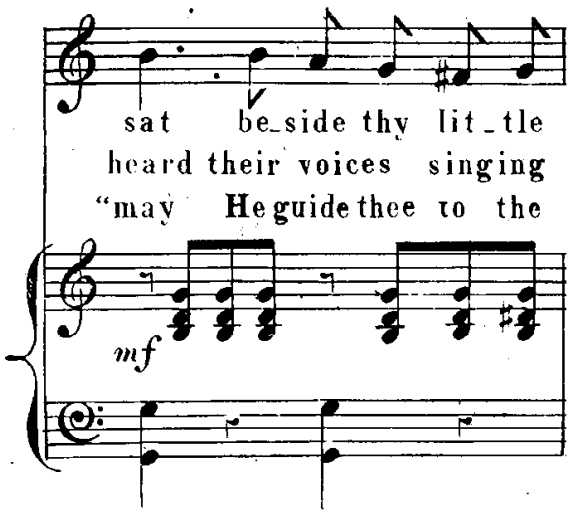


Figure 3.23: The original image.

PostScript output

The PostScript output (Figure 3.24) 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 XML-based glyph list that forms the bridge between AOMR and OMI. It was used during the development of AOMR to debug the primitive recognition system.

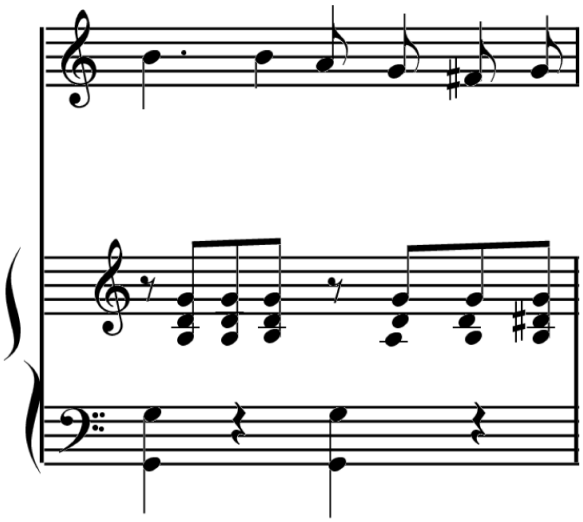


Figure 3.24: The PostScript output.

GUIDO output

Figure 3.25 is the logical interpretation of the score in GUIDO format. Note that the format is human readable and fairly intuitive. Since this is a musical representation, the physical layout of the original score is not stored.

Re-rendered notation

Figure 3.26 shows output of OMI re-rendered using the GUIDO *NoteServer*. The exact positions of the notes are determined solely from the logical representation of the score and thus are different from the original.

3.7 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 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

```
% 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
  ]
}
```

Figure 3.25: The GUIDO output.

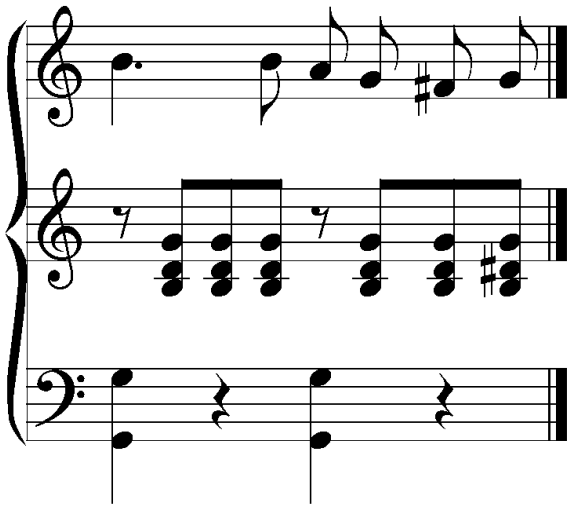


Figure 3.26: The output from GUIDO *NoteServer*.

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.

3.7.1 Overview

The overall debugging system is divided between an image viewer and the OMI debugging interface. Besides providing the basic functionality of scaling and displaying the image, the viewer can colorize or draw rectangles on arbitrary parts of the image. Secondly, a simple GUI allows 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 colors itself in the viewier. In addition, the `__repr__` function (normally used in Python to print an object to standard out) 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.

3.7.2 Pages

The interactive self-debugger uses a notebook interface to divide the functionality into different categories. These pages are described below.

Attribute page

Each button on the attribute page (Figure 3.27) 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.) Coloring is an efficient way for the developer to debug an algorithm and ensure that it is producing the correct results.

Class browser page

The class browser page (Figure 3.28) displays a list of all classes in the glyph class hierarchy. Clicking on an entry highlights all glyphs of that class in the viewer. This helps the debugger ensure that glyphs were properly identified during the AOMR stage and were properly disambiguated by OMI.

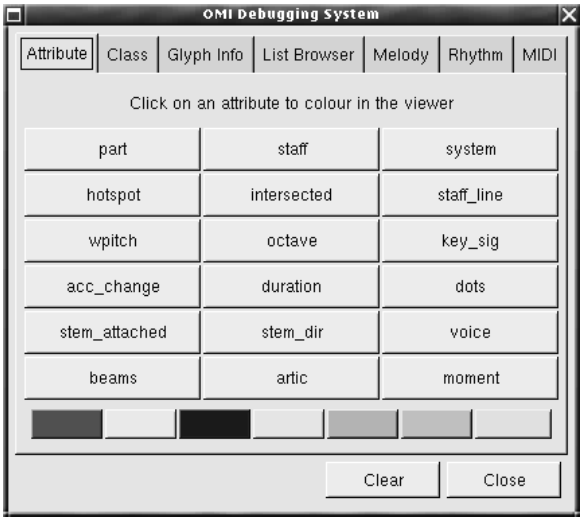


Figure 3.27: The attribute page of the interactive debugger.

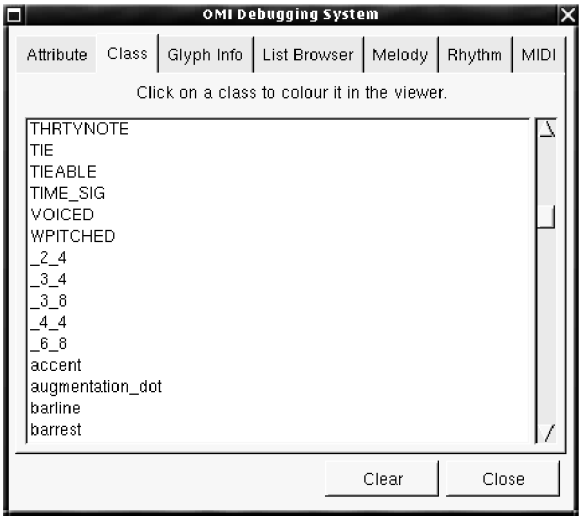


Figure 3.28: The classes page of the interactive debugger.

Glyph info page

Clicking on a glyph in the viewer displays all of its data members on the glyph info page (Figure 3.29). The text displayed on the glyph info page is taken directly from the output of the glyph's `__repr__` function. These details can help debug why certain algorithms were failing for certain glyphs.

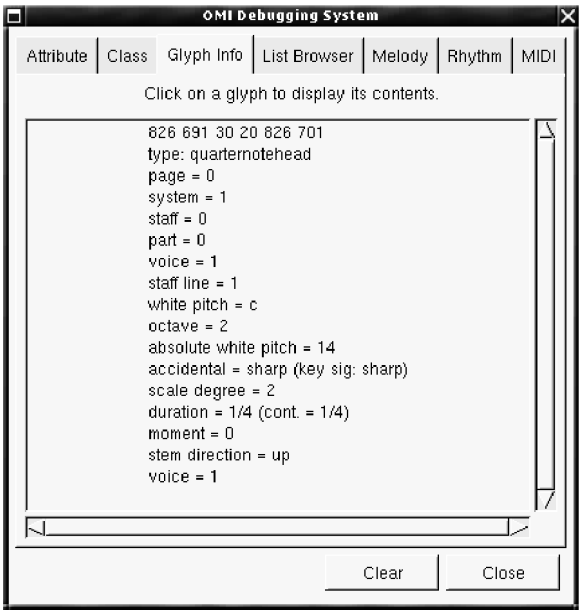


Figure 3.29: The glyph info page of the interactive debugger.

List browser page

The list browser page (Figure 3.30) displays a list of all glyphs in the score in their temporal order. This page helps to debug the sorting algorithms (Section 3.3), 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.

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

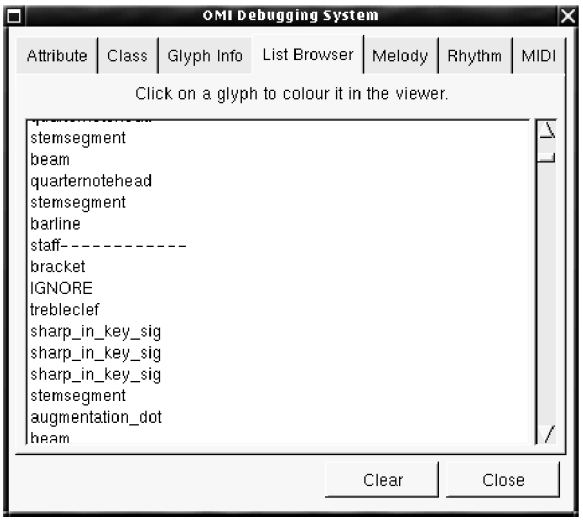


Figure 3.30: The list browser page in the interactive debugger.

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.

Melody page

The melody page provides a prototype of a musical search engine. The user may type in a search query and the results are highlighted in the image viewer. Melodic searching is a large topic that has also been the subject of research in the Levy project (Chapter 4).

Rhythm page

The rhythm page provides a basic rhythmic search engine. The user may search for a rhythm by tapping it on the mouse. All instances of that rhythm on the score are highlighted in the image viewer. Since the root of the rhythm can not be determined by mouse clicking alone, all equivalent rhythms are highlighted. For instance, two eighth notes followed by a quarter note is equivalent to two quarter notes followed by a half note.

MIDI page

The MIDI page automatically converts the GUIDO output to MIDI using the third-party *gmn2midi* program and plays it over the computer's sound hardware.

3.7.3 Reflections on the interactive self-debugger

The interactive self-debugger has proven to be an invaluable tool for 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 allowed the programmer to examine the state of the data structures. The combination of Python with one of the convenient GUI development tools, in this case *Python-Gnome* (Henstridge 2000), makes developing such in-house tools relatively easy work, and certainly should be a recommended development practice.

3.8 Conclusion

The present system handles many of the inherent difficulties of optical music interpretation in an elegant and simple way. This elegance is in no small part due to its implementation in Python, which facilitated achieving the three main design criteria: automatability, portability, and extensibility. Due to its solid foundation in a flexible object-oriented language, any future changes should remain relatively simple to implement, keeping development time to a minimum. Ultimately, the hope is that other large sheet music digitization projects will use this system because it presents a flexible and extensible alternative to closed systems.

Chapter 4

Symbolic music information retrieval

4.1 Introduction

This chapter describes a system for music searching that is expressive enough to perform both simple and sophisticated searches that meet a broad range of user needs. It is also efficient enough to search through a large corpus in a reasonable amount of time. The music search system was created by extending an existing advanced natural language search engine with simple filters and user-interface elements.

First, the search engine will be related other musical search engines already available for use on the web. Then, the capabilities of the non-music-specific core of the search engine will be described, followed by the extensions necessary to adapt it to music.

4.2 Other search engines

None of the available musical search engines we evaluated met the needs of the diverse user base of the collection, or could handle the large quantity of data in the complete Levy collection. In particular, we evaluated two projects in detail: Themefinder (Kornstädt 1998) and MELDEX (McNab et al. 1997).

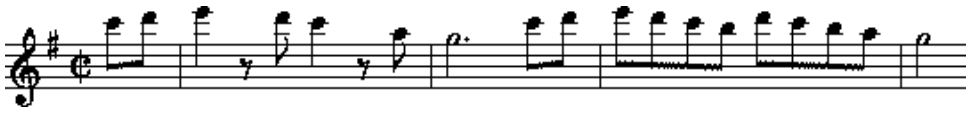
4.2.1 Themefinder

Themefinder's goal is to retrieve works by their important themes. These themes are manually determined ahead of time and placed in an incipit database.

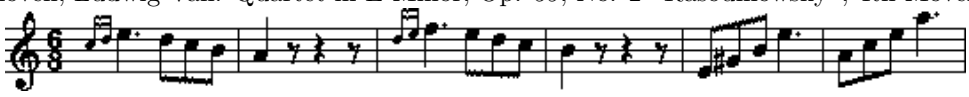
One can query the database using five different kinds of search queries: pitch, interval, scale degree, gross contour, and refined contour. These five categories served as the inspiration for a subset of the basic query types in the present system. The user can query within an arbitrary subset of these categories and then intersect the results. However, Themefinder does not allow the user to combine these query types within a single query in arbitrary ways. For instance, a user may know the beginning of a melodic phrase, while the ending is more uncertain. Therefore, the user may want to specify exact intervals at the beginning and use gross contours or wild-cards at the end. Unfortunately, in Themefinder, the user must have the same level of certainty about all of the notes in the query. This is perhaps not consistent with how one remembers melodies (McNab et al. 2000).

In addition, Themefinder does not have a notion of rhythmic searching. While its invariance to rhythm can be an asset, it can also be cumbersome when it provides too many irrelevant matches. Figure 4.1 shows the results of a query where one result is more relevant than the other. Such queries may return fewer false matches if they could include rhythmic information.

The searches themselves are executed in Themefinder using a brute-force method. The entire database is linearly searched for the given search query string. While this is acceptable for the 18,000 incipits in Themefinder's largest database, it may not scale well for searching across a full-text database such as the Levy collection.



Beethoven, Ludwig Van. Quartet in E Minor, Op. 59, No. 2 "Rasoumowsky", 4th Movement.



Beethoven, Ludwig Van. Sonata No. 4, in A Minor, Op. 23, Violin and Pianoforte, 1st Movement.

Figure 4.1: These two incipits start with the identical set of pitches, [*c d e d*], but with different rhythmic content. With better rhythmic specificity, irrelevant results could be eliminated. (`http://www.themefinder.org/`)

4.2.2 MELDEX

The simple text-based query strings in Themefinder are easy to learn and use by those with moderate musical training. MELDEX, however, has a more natural interface for non-musicians. The user sings a melody using a syllable with a strong attack such as "tah." The pitches of the melody are determined using pitch-tracking, and the rhythm is quantized. The results are used as the search query. The query is approximately matched to melodies in the database using a fast-matching algorithm related to dynamic programming. While this approach is highly effective for non-musicians and simple queries, it is limiting to those wanting more fine-grained control.

4.3 Capabilities

The present musical search engine supports both melodic and rhythmic searches. Search queries can also include the notion of simultanaeity. That is, events can be constrained to occur at the same time as other events. The search engine, as described here, is limited to standard-practice Western music, though modifications could be made to support other musical traditions.

4.3.1 Extensibility

Other types of musical searching beyond these core capabilities require additional layers of musical knowledge to be built on top of the search engine. The general design of the search engine encourages such extensibility. Any analytical data that can be derived from the score data can be generated offline (ahead of time) and later used as search criteria. This data can be generated by new custom tools or existing analysis tools such as the Humdrum toolkit (Huron 1999).

For example, the search engine could be extended to support harmonic searches with respect to harmonic function. Western tonal harmonic theory is ambiguous, making it difficult to objectively interpret and label harmonies. This is a largely unsolved problem that is not the subject of this research. However, assuming an acceptable solution to these issues could be found, labeling of harmonic function could be implemented as an input filter.

Also, the core search engine does not include any notion of melodic similarity. This is an open problem strongly tied to subjective matters of human perception (Hewlett and

Selfridge-Field 1998). It is possible for a specialized front-end to include notions of melodic similarity by generating specialized search queries. The search query language of the core search engine is expressive enough that these advanced features could be added without modifying the core itself.

4.3.2 Meeting diverse user requirements

The user of this musical search engine can be anyone who wants to access the collection in a musical way. Of course, the needs of different users are greatly varied. A non-musician may want to hum into a microphone to retrieve a particular melody. A copyright lawyer may want to track the origins of a particular melody, even melodies that are merely similar. A musicologist may want to determine the frequency of particular melodic or rhythmic events. To meet these diverse needs, it is necessary to provide different interfaces for different users. The set of interfaces is arbitrary and can be extended as new types of users are identified. It may include graphical applications, web-based forms and applets, or text-based query languages. Audio interfaces, with pitch- and rhythm-tracking may also be included. The purpose of these interfaces is to translate a set of user-friendly commands or interactions into a query string accepted by the search engine. The details of that query can be hidden from the end-user and therefore can be arbitrarily complex.

At present, attention has been focused on the core search engine itself. In the second phase of the search engine project, the user interfaces will be developed in collaboration with a usability specialist.

4.4 The core search engine

The core search engine in this system was originally developed for text-based retrieval of scores based on their metadata and full-text lyrics. Its overall design was inspired by recent developments in the field of natural-language searching (DiLauro et al. 2001). These features allow the user to perform search queries using the embedded context in natural languages, such as parts of speech, rhyming scheme, and scansion. While not originally intended for musical searching, it was soon discovered that the core was very well suited for searching through symbolic musical data.

The core itself did not need to be modified to support music searching. Instead, specialized filters and front-ends were added to adapt it to the music domain. In the ingestion

stage, the data is filtered to store it in the appropriate indices and partitions (see Section 4.5). When searching, special user interfaces handle the details of generating search query strings and filtering and displaying the resulting data. Figure 4.2 shows how the individual parts of the system fit together to ingest and query the data.

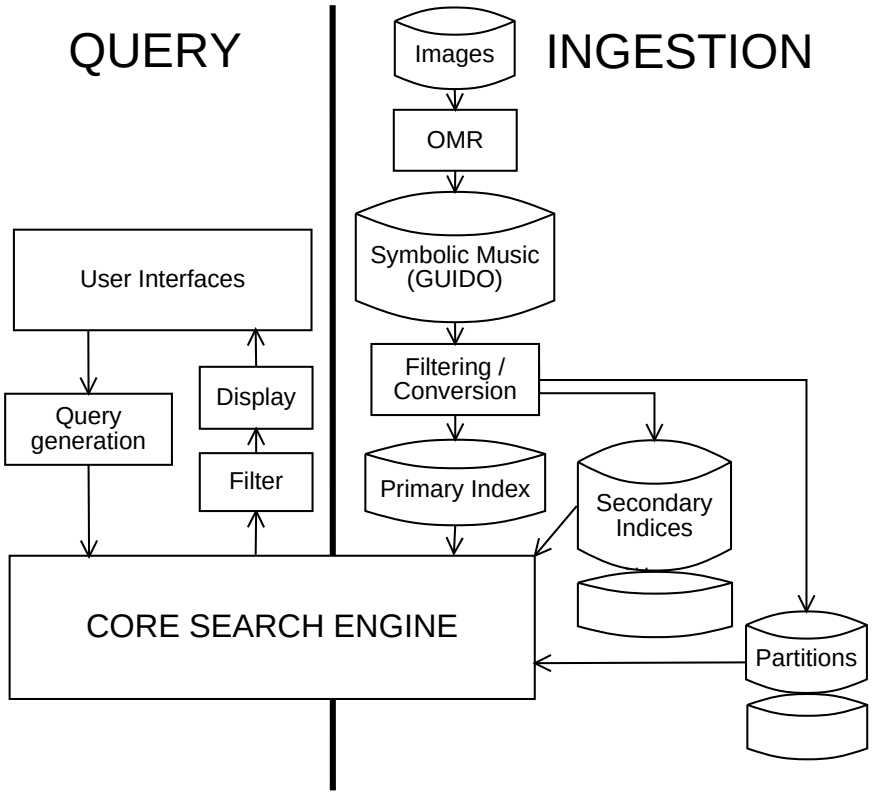


Figure 4.2: Workflow diagram of the musical search engine.

4.4.1 Inverted lists

Many search engines, including this one, are built on the concept of an inverted list. For a complete discussion of inverted list search engines, see Witten et al. (1999).

Sequential data, such as English prose or melodic data, is stored on disk as a sequence of atoms. In the case of English, the atom is the word and the sequence is simply the ordered words as they appear in sentences and paragraphs. Take for example the following sentence:

```
To be , or not to be , that is the question .
```

Note that both words and punctuation are treated as indivisible atoms. To search for a particular atom in this string, a computer program would need to examine all thirteen atoms and compare it with a query atom. To increase searching efficiency, an inverted list search engine would store this string internally as:

, $\longrightarrow$ {3, 8}
. $\longrightarrow$ {13}
be $\longrightarrow$ {2, 7}
is $\longrightarrow$ {10}
not $\longrightarrow$ {5}
or $\longrightarrow$ {4}
question $\longrightarrow$ {12}
that $\longrightarrow$ {9}
the $\longrightarrow$ {11}
to $\longrightarrow$ {1, 6}

Here, each atom in the string is stored with a list of numbers indicating the atom's ordinal location within the string. The set of words in the index is called the vocabulary of the index. To search for a particular atom using the index, the program needs only to find that word in the vocabulary and it can easily obtain a list of indices (or pointers) to where that atom is located within the string. Since the vocabulary can be sorted, the lookup can be made faster using hashing or a binary search.

Inverted lists perform extremely well when the size of the vocabulary is small relative to the size of the corpus. In the case of English, of course, the vocabulary is much smaller relative to the size of all the works written in that language. This property also allows us to improve the efficiency of musical searching as we will see below.

4.5 The musical search engine

The musical search capabilities are supported by three main features of the core search engine:

1. **Secondary indices** allow the amount of specificity to vary with each token.

2. **Partitions** allow search queries to be performed upon specific discontiguous parts of the corpus.

3. **Regular expressions** allow advanced pattern matching.

4.5.1 Secondary indices

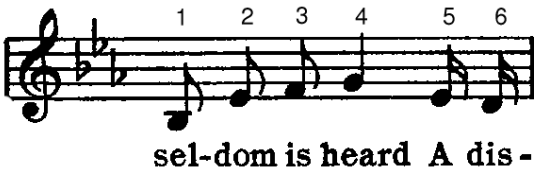


Figure 4.3: A measure of music, from the Levy collection, used as an example throughout this section. (Guion, D. W., arr. 1930. "Home on the range." New York: G. Schirmer.)

In the case of music, the searchable atom is not the word, but the musical event. Events include anything that occurs in the score, such as notes, rests, clefs, and barlines. Each of these events, of course, can have many properties associated with it. For instance, the note $b\flat$ at the beginning of the fragment in Figure 4.3 has the following properties:

- **Pitch name:** b
- **Accidental:** $\flat$
- **Octave:** -1 (first octave below middle-c)
- **Twelve-tone pitch:** 10 (10^{th} semitone above c)
- **Base-40 pitch:** 37 (see Hewlett 1992)
- **Duration:** eighth note
- **Interval to next note:** perfect 4^{th}
- **Contour (direction) to next note:** up
- **Lyric syllable:** "sel–"
- **Metric position:** Beat 1 in a $\frac{6}{8}$ measure

All of these properties are self-evident, with the exception of base-40 pitch, which is a numeric pitch representation where the intervals are invariant under transposition while maintaining their enharmonic spelling (Hewlett 1992). Note also, we use GUIDO-style octave numbers, where the octave containing middle-c is zero, as opposed to ISO-standard octave numbers.

The concept of secondary indices allows the individual properties of each atom to be indexed independently of any other properties. This allows search queries to have arbitrary levels of specificity in each event. The set of properties can be extended to include any kinds of data that can be extracted from the musical source. For example, if the harmonic function of chords could be determined unambiguously, a secondary index containing chord names in Roman numeral notation could be added. In this design, secondary indices are used to handle the properties of events that change from note to note. Continuous properties of events, that are carried from one event to the next, such as clefs, time signatures, and key signatures, are handled using partitions, explained below (see Section 4.5.2).

Ingestion of secondary indices

During the ingestion phase, the source GUIDO data is first converted to an interim format where all of each event's properties are fully specified. For example, the GUIDO representation of Figure 4.3 is as follows:

```
[ \clef<"treble"> \key<-3> \time<6/8>
    \lyric<"sel-"> b&-1*/8.
    \lyric<"dom"> e&*0
    \lyric<"is"> f
    \lyric<"heard"> g/4
    \lyric<"A"> e&/16
    \lyric<"dis-"> d
]
```

Each event is then extended so it is fully specified. In this format, each note event is a tuple of properties:

> *pitch-name, accidental, octave, twelve-tone-pitch, base-40-pitch, duration, interval, contour, scale-degree, lyric-syllable, metric-position*

Figure 4.4 shows the example in fully-specified symbolic representation.

Each one of these fields is used to index the database in a particular secondary corpus. For example, if the notes in the example were labeled 1 through 6, the data in the secondary indices may look something like:

- **Pitch name**

  `a` $\longrightarrow \emptyset$

```
[ \clef<"treble"> \key<-3> \time<6/8>
    b, &, -1, 10, -3,  1/8, P4, /, so, "sel-",      0
    e, &,  0,  3, 14,  1/8, M2, /, do, "dom",     1/8
    f, n,  0,  5, 20,  1/8, M2, /, re, "is",      1/4
    g, n,  0,  7, 25,  1/4, M3, \, mi, "heard",   3/8
    e, &,  0,  3, 15, 1/16, m2, \, do, "A",       5/8
    d, n,  0,  2,  9, 1/16, M2, \, ti, "dis-",  11/16
]
```

Figure 4.4: Fully specified symbolic representation of the example in Figure 4.3.

`b` $\longrightarrow \{1\}$

`c` $\longrightarrow \emptyset$

`d` $\longrightarrow \{6\}$

`e` $\longrightarrow \{2, 5\}$

`f` $\longrightarrow \{3\}$

`g` $\longrightarrow \{4\}$

- **Accidentals**

  `n` ($\natural$) $\longrightarrow \{3, 4, 6\}$

  `&` ($\flat$) $\longrightarrow \{1, 2, 5\}$

- **Octave**

  `-1` (octave below middle-c) $\longrightarrow \{1\}$

  `0` (octave above middle-c) $\longrightarrow \{2, 3, 4, 5, 6\}$

- **Duration**

  `1/4` (quarter note) $\longrightarrow \{4\}$

  `1/8` (eighth note) $\longrightarrow \{1, 2, 3\}$

  `1/16` (sixteenth note) $\longrightarrow \{5, 6\}$

Searching using secondary indices

The search query itself is simply a series of events. Each event can be indicated as specifically or as generally as the end user (as represented by a user interface) desires. For example, the following query would match any melodic fragment that begins on a $b^\flat$ eighth note, has a sequence of 3 ascending notes, ending on a g:

```
b,&,1/8 / / / g
```

To execute a search query using secondary indices, the search engine looks up each parameter in their corresponding secondary indices, and retrieves tokens in the secondary index. These tokens are then looked up in the primary index, returning a list of positions. These lists are intersected to find the common elements. This list of locations is then filtered to include only those events that are sequenced according to the search query.

Supported user interfaces

This design supports a broad range of user interfaces. A text-based user interface may allow a user to be very specific in the query, and then incrementally remove layers of specificity until the desired match is retrieved. An audio-based user interface could be more or less specific depending on the pitch-tracker's confidence in each event.

Efficiency of secondary indices

One of the efficiency problems with this approach is that the vocabularies of the individual secondary indices tend to be quite small, and thus the index lists for each atom are very large. For instance, the "pitch name" secondary index has only seven atoms in its vocabulary (a - g). "Accidentals" is even smaller: {$\flat\flat$, $\flat$, $\natural$, $\sharp$, $\times$}. Therefore, a search for a $b^\flat$ must intersect two very large lists: the list of all b's and the list of all flats. However, the search engine can combine these secondary indices in any desired combination off-line. For example, given the "pitch name" and "accidental" indices, the search engine can automatically generate a hybrid index in which the vocabulary is all possible combinations of pitch names and accidentals. The secondary indices can be automatically combined in all possible combinations, to an arbitrary order.

4.5.2 Partitions

Partitioning can be used to restrict a search query to a particular part of the corpus. Each partition is a description of how to divide the corpus into discontiguous, non-overlapping regions. More specifically, each partition is a file containing a list of regions. Each region within a partition is named and has a list of its start and stop positions.

In this music search engine, the metadata is used to partition the corpus into regions. For example, all works by a given composer would make up a discontiguous region in the

"composer" partition. Partitions exist for all types of metadata in the collection, including date, publisher, geographical location, etc.

In addition, we have extended partitioning to include musical elements derived directly from the GUIDO data. Regions are generated from key signatures, clefs, time signatures, measures, movements, repeats, etc. This allows for searching for a particular melody in a particular key and clef, for example.

Ingestion of partition data

When a new work is added to the corpus, the data is partitioned automatically. First, the metadata regions, such as title, composer, and date, are set to include the entire piece. As the piece is scanned, continuous musical elements, such as clef, key signature, and time signature, are regioned on-the-fly. Therefore, when the ingestion filter sees a "treble clef" token, all further events are added to the "treble clef" region until another clef token is encountered. Lastly, events are added to the moment regions on an event-by-event basis.

For the example in Figure 4.3, again assuming the notes are numbered 1 through 6, the partitions may look something like:

- **Title partition**

  "Home on the range" $\longrightarrow$ [1, 6]
- **Clef partition**

  Treble clef $\longrightarrow$ [1, 6]
- **Time signature partition**

  $\frac{6}{8} \longrightarrow$ [1, 6]

Searching using partitions

Extending the example in Section 4.5.1, the user may wish to limit the search to the key signature of $E^{\flat}$-*major*:

```
( b,&,1/8 / / / g ) @ key:"E& major"
```

Here the non-partitioned search query is performed as described above, and then the results are intersected with the results of the partition lookup. Since in this case, the entire range of notes [1, 6] is in the key signature of $E^{\flat}$-*major*, the query will retrieve the example in Figure 4.3.

Searching with simultaneity using partitions

Scores are also partitioned at the most atomic level by "moments." A moment is defined as a point in time when any event begins or ends in any part. Moments almost always contain multiple events, and events can belong to multiple moments (e.g. when a half note is carried over two quarter notes in another part). Each moment within a score is given a unique numeric identifier, and all events active at a given point are included in a moment region. In this way, one can search for simultaneous polyphonic events very efficiently.

To explain this further, Figure 4.5 shows the example measure with its assigned moment numbers. Each event is assigned to one or more moments so that it can be determined which, if any, of the events are active at the same time. These moment numbers are used to create regions. For example, the dotted half note in the left hand of the piano part would be assigned to all seven moment regions.

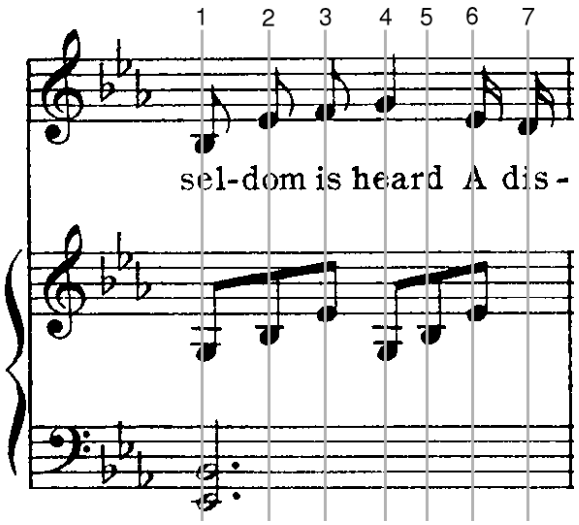


Figure 4.5: The example measure of music showing moment numbers.

To perform searches involving simultanaeity, the query for each part is performed separately, and then the results are intersected based on their moments. Only the query results that occur at the same time (existing in the same moment regions) will be presented to the user.

4.5.3 Regular expressions

The core search engine supports a full complement of POSIX-compliant regular expressions. Regular expressions, a large topic beyond the scope of this paper, are primarily used for pattern-matching within a search string (Friedl 1997). The Humdrum toolkit (Huron 1999) has proven the usefulness of regular expressions for musicological research.

Many users find regular expressions difficult and cumbersome ways to express searches. The intent is that most of these details will be hidden from the user by appropriate interfaces. For example, regular expressions would be very useful for an interface that allowed searching by melodic similarity. What is important to our present research is that regular expressions are supported in the core search engine, leaving such possibilities open.

4.6 Conclusion

Based on existing advanced natural-language search techniques, an expressive and efficient musical search engine has been developed. Its special capabilities include: secondary indices for gradiated specificity, partitions for selective scope and simultanaeity, and regular expressions for expressive pattern matching. This allows users with different search needs to access the database in powerful and efficient ways.

References

Abelson, H., R. K. Dybvig, C. T. Haynes, G. J. Rozas, N. I. Adams IV, D. P. Friedman, E. Kohlbecker, G. L. Steele Jr., D. H. Bartley, R. Halstead, D. Oxley, G. J. Sussman, G. Brooks, C. Hanson, K.M. Pitman and M. Wand. 1998. Revised report on the algorithmic language Scheme. *Higher-order and symbolic computation* 11(1): 7–105.

Aha, D. W. 1997. Lazy learning. *Artificial 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 Proceedings, *International Conference on Document Analysis and Recognition* 1080–3.

Bracha, G., and W. Cook. 1990. Mixin-based inheritance. In Proceedings, 8^{th} *Conference on Object-Oriented Programming, Systems, Languages, and Applications / European Conference on Object-Oriented Programming (OOPSLA/ECOOP)*, ACM SIGPLAN Notices 25(10): 303–11.

Bray, T., M. Paoli, C. M. Sperberg-McQueen, E. Maler. 2000. Extensible Markup Language (XML) 1.0 (Second Edition). W3C. `http://www.w3.org/TR/`

Brinkman, A. R. 1990. *Pascal programming for music research.* University of Chicago Press.

Byrd, D. 1984. Music notation by computer. Ph. D. thesis, Indiana University.

Byrd, D. 1994. Music notation software and intelligence. *Computer Music Journal* 18(1): 17–20.

Castan, G. 2000. Common music notation and computers. World wide web. `http://www.s-line.de/homepages/gerd_castan/compmus/index_e.html`

Choudhury, G. S., C. Requardt, I. Fujinaga, T. DiLauro, E. W. Brown, J. W. Warner, and B. Harrington. 2000. Digital workflow management: The Lester S. Levy digitized collection of sheet music. *First Monday* 5(6).

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

Cooper, C. 1999. Using Expat. *XML magazine* 9/99.

Cormen, T. H., C. E. Leiserson and R. L. Rivest. 1997. *Introduction to algorithms.* Cambridge, MA: MIT Press.

Cost, S., and S. Salzberg. 1992. A weighted nearest neighbor algorithm for learning with symbolic features. *Machine Learning* 10: 57–78.

Cottle D., and L. Haken. 1997. The *LIME Tilia* representation. In E. Selfridge-Field, ed., *Beyond MIDI: The handbook of musical codes.* Cambridge, MA: MIT Press.

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

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

Dannenberg, R. 1993. Music representation issues, techniques and systems. *Computer Music Journal* 17(3): 20–30.

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

DiLauro, T., G. S. Choudhury, M. Patton, J. W. Warner, and E. W. Brown. 2001. Automated name authority control and enhanced searching in the Levy collection. *D-Lib Magazine* 7(4).

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

Free Software Foundation, Inc. 1991. GNU general public license. `http://www.gnu.org/licenses/gpl.html`

Friedl, J. E. F. 1997. *Mastering regular expressions.* Sebastopol, CA: O'Reilly.

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.* New York: Addison-Wesley.

Gerou, T., and L. Lusk. 1996. *Essential dictionary of music notation.* Los Angeles: Alfred.

Haken, L. and D. Blostein. 1993. The *Tilia* music representation: Extensibility, abstraction, and notation contexts for the *LIME* music editor. *Computer Music Journal* 17(3): 43–58.

Halperin, D. 1997. Afterword: Guidelines for new codes. In E. Selfridge-Field, ed., *Beyond MIDI: The handbook of musical codes.* Cambridge, MA: MIT Press.

Hamel, K. 1998. NoteAbility: A comprehensive music notation editor. In Proceedings, *International Computer Music Conference* 506–9.

Henstridge, J. 2000. Python-Gnome. Computer Program (UNIX). `http://www.theoenlab.uml.edu/pygtools/`.

Hewlett, W. B. 1992. A base-40 number-line representation of musical pitch notation. *Musikometrika* 4: 1–14.

Hewlett, W. B., and E. Selfridge-Field. 1998. *Melodic similarity: Concepts, procedures and applications.* Cambridge, MA: MIT Press.

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 Universitat Darmstadt. `http://www.informatik.tu-darmstadt.de/AFS/GUIDO/docu/spec1.htm`.

Hoos, H. H., J. Kilian, K. Renz, and T. Helbich. 1998. The SALIERI project. In Proceedings, *International Computer Music Conference* 385–92.

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

Huron, D. 1999. *Music research using Humdrum: A user's guide.* Menlo Park, CA: Center for Computer Assisted Research in the Humanities.

Icking, W. 1997. MuTEX, MusicTEX, and MusiXTEX. In E. Selfridge-Field, *Beyond MIDI: The handbook of musical codes.* Cambridge MA: MIT Press.

Kelley, D. 1995. *Automata and formal languages: An introduction.* Englewood Cliffs, NJ: Prentice Hall.

Kornstädt, A. 1998. Themefinder: A web-based melodic search tool. *Computing in Musicology* 11: 231–6.

Knuth, D. 1984. *The TEXbook.* Reading, MA: Addison Wesley.

Lamport, L., and D. Bibby. 1994. *LATEX: A document preparation system: User's guide and reference manual.* Reading, MA: Addison Wesley.

Lutz, M. 1996. *Programming Python.* Sebastopol, California: O'Reilly.

McNab, R. J., L. A. Smith, D. Bainbridge, and I. H. Witten. 1997. The New Zealand Digital Library MELody inDEX. *D-Lib Magazine*: 3(5).

MacMillan, K., M. Droettboom and I. Fujinaga. 2001. Gamera: A structured document recognition application development environment. In Proceedings, *International Symposium on Music Information Retrieval* 15–6.

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 Inc. 2000. Pixel/FX! 2000. Computer Program (UNIX). `http://www.mentalix.com/`.

Meyer, B. 1997. *Object-oriented software construction.* 2nd ed. New York: Prentice-Hall.

MIDI Manufacturers Association Inc. 1986. *The complete MIDI 1.0 specification.* `http://www.midi.org`.

Mounce, S. 2000. Music encoding standards. `http://www.student.brad.ac.uk/srmounce/encoding.html`

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., and J. Nieuwenhuizen. 1998. *LilyPond user documentation (containing Mudela language description).* GNU Project. `http://www.gnu.org/`.

Nienhuys, H., J. Nieuwenhuizen, and A. Mariano. 2000. LilyPond internals. `http://www.cs.uu.nl/people/hanwen/lilypond/`

Nienhuys, H. 2001. *The Mutopia Project.* `http://www.mutopiaproject.org`.

Patashnik, O. 1998. *BIBTEX ing.* Included with BIBTEX computer software.

Read, G. 1969. *Music notation: A manual of modern practice.* New York: Taplinger.

Renz, K., and H. H. Hoos. 1998. A web-based approach to music notation using GUIDO. In Proceedings, *International Computer Music Conference* 455–8.

Van Rossum, G., and F. L. Drake. 2000. *Python tutorial.* Campbell, CA: iUniverse.

Rubin, C. 1999. *Running Microsoft Word 2000.* Redmond, WA: Microsoft Press.

Schulenberg, J. 2000. *gocr.* Computer Program (Microsoft Windows, UNIX).

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

Selfridge-Field, E. 1997. *Beyond MIDI: The handbook of musical codes.* Cambridge, MA: MIT Press.

Selfridge-Field, E. 1997b. Beyond codes: Issues in musical representation. In E. Selfridge-Field, ed., *Beyond MIDI: The Handbook of Musical Codes.* Cambridge, MA: MIT Press.

Smith, L. A., and R. J. McNab. 1996. Melody transcription for interactive applications. Technical report, University of Waikoto, Hamilton, NZ. 96(32).

Smith, L. A., R. J. McNab, and I. H. Witten. 1998. Sequence-based melodic comparision: A dynamic-programming approach. *Melodic Similarity: Concepts, Procedures and Applications*, edited by W. B. Hewlett and E. Selfridge-Field. *Computing in Musicology*: 11.

Stroustrup, B. 1997. *The C++ programming language (Special edition).* Reading, MA: Addison Wesley.

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

Weibel, S., J. Kunze, C. Lagoze, and M. Wolf. 1998. Dublin core metadata for resource discovery. IETF #2413. *The Internet society* September, 1998.

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

Witten, I., A. Moffat, and T. Bell. 1999. *Managing gigabytes.* 2nd ed. San Francisco: Morgan Kaufmann.

Part II

Tempo extraction

Chapter 5

RUBATO: A system for determining tempo fluctuation in recorded music

5.1 Introduction

In a 1981 episode of *That's Incredible*, Dr. Arthur B. Lintgen demonstrated his unusual gift for identifying recordings of classical music simply by examining the groove patterns on phonograph records (Figure 5.1). While his accuracy within the standard classical repertoire is reported to be close to 100%, he has difficulty distinguishing between two performances of the same work (Holland 1981). This is not due to a lack of skill, but to the nature of a phonograph's grooves. To the naked eye, the grooves give an overview of the recorded sound, (i.e. a particular work of music) while the fine-grained details (i.e. the performance differences) are hidden.

Lintgen's ability inspired Jonathan Foote to design the ARTHUR system for automatic identification of recorded music (Foote 2000). ARTHUR simulates the visual appearance of a phonograph by smoothing a digital audio signal to such an extent that only its rough contours remain. The resulting "fingerprint" can be matched against pre-labeled prototypes to identify the composition. The matching itself is performed using the dynamic programming algorithm (DPA) (Section 5.3.2) that can handle variations in tempo and amplitude.

Figure 5.1: Dr. Arthur B. Lintgen was able to identify recordings of classical music simply by examining the groove patterns on phonograph records (Holland 1981).

The DPA determines both how closely two recordings match, and, more importantly, the moments along their duration where the music coincides.

The RUBATO (Rubber-banded Automatic Tempo Obtaining) system presented here builds on Foote's work. While ARTHUR was designed specifically for identification of musical works, RUBATO extracts a tempo and rubato curve for time-based performance analysis. Since the system is still in active development, this is a report of a work in progress, outlining the system's successes and shortcomings. Potential research applications of the technology are also discussed.

5.2 A brief history of musical time

Tempo has an unusual place in Western music. Unlike pitch or rhythm, tempo is usually only suggested by the composer.

> "Tempo indication is not a creation, but an afterthought related to performance. Naturally an inherently fast piece must be played fast, a slow one slow—but to just what extent is a decision for players." —Rorem (1983)

Fluctuation in tempo, or *rubato*, is expected and encouraged, whether or not it is indicated in the score. In fact, music without any rubato is perceived as mechanical and unnatural (Todd 1989). The tempo and rubato of a performance are, therefore, usually a committee

decision between the composer, the performer, and the too oft-forgotten force of common practice and experience. It is perhaps because tempo is not strictly documented in the score, that it has historically received relatively little attention from musicologists (Bowen 1996).

With the advent of recordings, musicologists now have access to a resource for analysis that contains tempo information in a very concrete form. However, the researcher must exercise caution when using recordings as documents of authentic performance practice. No less a composer than Igor Stravinsky noted:

> "The disadvantages [of recordings], are that one performance represents only one set of circumstances, and that mistakes and misunderstandings are cemented in tradition quickly and canonically as truths."
>
> —Stravinsky (1971) as quoted in Buxbaum (1988)

In addition, the creation of, listening to, and distribution of recordings has a direct effect on performance practice, and therefore, cyclically on recordings themselves (Katz 1999). With these pitfalls in mind, recordings can still be a valuable resource. Unlike a live performance, recordings allow for repeated identical listenings and the precise gathering of data for analysis and comparison (Bowen 1996).

5.2.1 Current research

Much of the research that involves extracting tempi from recordings (see Bowen, 1999) falls into three broad categories. Some papers attempt to provide practical guidelines to performers. Others use comparisons of recordings to highlight external social forces and historical trends in performance practice. The last group attempts to distill the human processes involved in the selection of tempo and rubato in order to mimic it by automatic means.

As an example of performance recommendations, Buxbaum (1988) compares the scores, recordings, and comments of Stravinsky. Unfortunately, the only conclusion made is that all of these factors must be taken into account. The "correct" tempi of Stravinsky likely do not exist.

Bowen (1996) and Katz (1999) use the tempo data from recordings to discover trends in performance practice. One of the most interesting observations is that "conductors from the first half of this century use more tempo fluctuation in more diverse ways than conductors

from the second half of the century" (Bowen 1996). This is perhaps an example of what Katz calls a "phonograph effect": frequent tempo fluctuation does not wear well with the repeatibility of recordings, and thus has been less common in recent years.

The last category of research aims to build a workable model of tempo and rubato practice. Todd (1989) is an early attempt to use extracted performance data and perceptual studies to develop an algorithm for tempo and rubato. Feldman et al. (1992) show that Todd's approach was partially unsuccessful because it assumed that tempo change was a linear phenomenon. They go on to demonstrate that a natural-sounding rubato curve is at least a third-degree polynomial. Another factor in the generation of natural-sounding timing is the relationship between tempo and rubato. As global tempo increases, what happens to the rubato microstructure? Desain and Honing (1994a) and Repp (1994) have participated in a long debate on this matter. Both groups agree that a relationship between tempo and rubato exists and that it is very complex. There are many factors involved, including rhythmic density and pitch. Early studies that analyzed recordings by themselves were not conclusive, since the performer's comfort level at different tempi would affect the amount of rubato they were able to play. Later studies that involved perceptual testing of the relative "naturalness" of different amounts of rubato were more successful (Repp 1995). Both groups of researchers have yet to find a practical model of the relationship. One of the obstacles to their research is the relatively data-poor environment. It is very expensive to analyse many recordings *en masse* using manual data-acquisition methods.

5.2.2 Tempo extraction

Manual methods

> "The most time-consuming step in the process of collecting information about performance is to measure the duration between each beat, convert this number into a tempo measurement, and then store this series of numbers in a database."
>
> —Bowen (1993)

Epstein (1985) and Katz (1999) describe using a stopwatch to time the length of each beat or measure. The time of each event is repeatedly resampled, and the average is taken to reduce the amount of human error. Bowen (1996) used custom computer software which recorded taps on the keyboard while the music was played. While more convenient than the stop watch method, it is prone to the same kinds of human error. Feldman et al. (1992) have a more accurate, if more time-consuming approach:

> "The beats were measured by first transferring the recordings to 7.5 ips [inches per second] open-reel tape. The attack points were then marked on the tape using a conventional editing technique, in which the tape is run back and forth over the playback head by hand at very slow speed until the attack point can be identified exactly. The distances between the attack points were measured to the nearest millimeter, resulting in a final resolution of about 5 msec."
>
> —Feldman et al. (1992)

This 5 millisecond resolution is well below the threshold of perceptible simultaneity (Levitin and Mathews 2001).

The RUBATO system represents a major improvement over these manual tempo extraction techniques. While it currently does not achieve the same resolution (Section 5.3.1), it is much less prone to manual error, and more importantly, much less time-consuming. On a typical notebook computer[1], for example, a tempo curve for a ten-minute piece can be obtained in five seconds. With the large quantity of recordings already available in digital form on compact discs and on the Internet, large data sets can be generated very quickly.

Automatic methods

RUBATO's technique is also markedly different from existing computer-based tempo-tracking systems. Most systems either work directly from MIDI events or use beat-induction or pitch-tracking techniques.

MIDI (Musical Instrument Digital Interface) events are simply instructions generated by an electronic instrument, typically a keyboard (MIDI 1986). For example, a MIDI instruction may represent the event: "press this key down at beat 4, using 50% pressure (or velocity), and hold for 2 beats." The data from MIDI events is therefore very specific and accurate, and has proved useful for a number of tempo experiments (Desain and Honing 1994a). However, it is limited in scope since a musician must perform directly on a MIDI instrument. Until automatic transcription is perfected, it is impossible to generate MIDI events from an existing recording. Also, MIDI is only sufficiently representative for modeling keyboard performances. The expressive range of other instruments and the voice are not adequately represented by the small set of MIDI events.

Beat-induction has many variations and is by far the most common form of tempo-tracking. In general, it locates the strong beats of each measure by finding peaks in am-

[1] Intel Pentium III 700 MHz with 128MB RAM, running Linux 2.4.

plitude (Desain and Honing 1994b). Unfortunately, the music being tracked "must have a regular pulse, corresponding to a regular time signature" (Driesse 1991). This requirement, of course, makes it difficult to use in the study of rubato.

Pitch-tracking matches a recording against a list of expected pitches (Smith and McNab 1996). Unfortunately, such an approach is dependent on the key of the performance, and is often undermined when a performer adds or misses notes. Pitch-tracking is also not very robust at handling complex multi-pitched textures.

In contrast, RUBATO's main strength lies in the fact that it is not searching for beats. It is merely matching one recording to another. It is more successful than the other approaches because it already has an approximate idea of what to look for. However, RUBATO's tempo matching can not be performed in real-time. It needs to scan the recordings in their entirety before it can begin to match them (Section 5.3.2). It is therefore unsuited to applications such as score-following or automatic accompaniment (Allen and Dannenberg 1990).

5.3 The RUBATO program

The following is an overview of the important algorithms in the program. Throughout, two recordings of Fugue No. 1 from J. S. Bach's *Well-Tempered Clavier*, BWV 846, are used as an example. One recording is performed by Glenn Gould (1963). The other is a strict MIDI rendering created directly from the score using the step-recording and quantization functions found in most MIDI sequencers. The reason for using a strict MIDI rendering is explained in Section 5.3.3.

RUBATO accepts digital audio data as input, such as that on a digital audio compact disc. The analysis data is output as both numeric values and graph plots.

5.3.1 Smoothing the audio signal

CD-quality digital audio signals are recorded at 44,100 samples per second. A visual representation of such waveforms is shown in Figures 5.2 and 5.3. Note that the two waveforms have some superficial resemblance in their overall contours, despite being radically different performances.

This raw digital data contains a lot of detail. Of course, when comparing two recordings of the same work by different performers, much of that detail should be ignored. Therefore, the data is reduced by taking the mean of the absolute values within some relatively long

window of time. The optimal window size depends on the overall textural complexity of the music. The goal is that the window should be as small as possible, but not so small that it includes too many timbral and performance details. Experiments show that window sizes of roughly one second produce consistently good results. The drawback of having such a long window, however, is that the resolution of the matching information can only be accurate within that window size. Using overlapping windows can increase the resolution, but overlapping more than two or three times re-introduces too much detail. Figure 5.3 shows how the overlapping windows line up with the original digital waveform.

Figure 5.2: Audio waveform of the first ten seconds of the strict MIDI rendering of Fugue No. 1 from J. S. Bach's *Well-Tempered Clavier*, BWV 846.

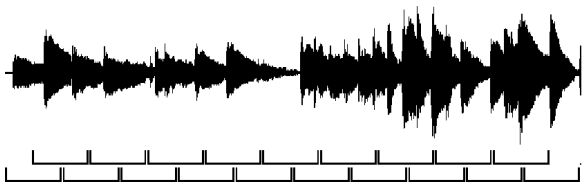


Figure 5.3: Audio waveform of the first ten seconds of Gould's 1963 recording of the fugue. The brackets at the bottom show the overlapping windows in which the means are gathered.

Once a mean has been extracted from each window, these values are smoothed to increase the continuity of the individual values. Figure 5.4 shows the derived means with their corresponding fitted curves. It is these curves, or the gross contours, of the recording that are used as input to the dynamic programming algorithm.

5.3.2 Dynamic programming algorithm

The dynamic programming algorithm (DPA) is a popular general-purpose algorithm in computer science, first developed by R. Bellman (Bellman 1957; Cormen et al. 1997). In

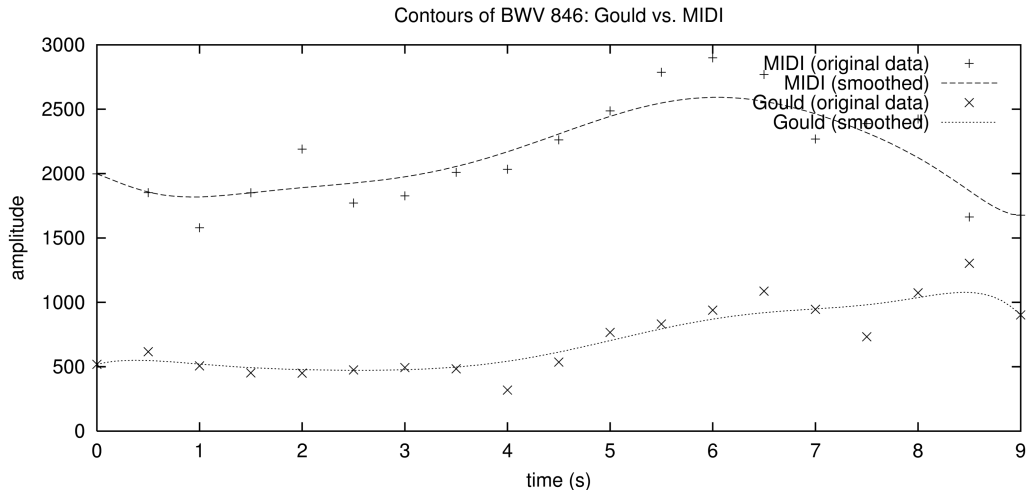


Figure 5.4: The gross contours of two recordings.

general, it is used to match strings of data that differ only in small variations of insertion, deletion, or repetition. It has been used in many applications that must match two streams of data with similar characteristics, such as natural language translation (Gale et al. 1992) and melodic similarity (Smith et al. 1998).

In the case of the RUBATO system, the DPA is used to find all the moments where the two recordings match. More specifically, it must find a contiguous path through all the possible matching points such that the sum of the differences at each point along that path is minimized. In order to do this, a difference table is created. It is filled with the absolute difference between the slopes of the two recordings at every possible combination. For example, Table 5.1 is the difference table for the Bach fugue (showing only the first ten windows, for simplicity). The slopes of the MIDI rendering are on the x-axis, and the slopes of the Gould recording are on the y-axis. The rest of the table is filled with the absolute differences between the slopes at each point. Given recordings X and Y, there are three choices at each cell of the table:

- **move right:** progress only in recording X.
- **move down:** progress only in recording Y.
- **move diagonally southeast:** progress in both recordings.

		MIDI (X)								
		-60.4	6.41	-5.89	5.33	6.11	6.02	-6.44	5.05	-6.61
	-2.63	• 3.41	9.04	3.26	7.96	8.74	8.65	3.81	7.68	3.98
	-4.04	• 2.00	10.40	1.85	9.37	10.20	10.10	2.40	9.09	2.57
G	3.78	9.82	• 2.63	9.67	1.55	2.33	2.24	10.20	1.27	10.40
o	-5.17	0.87	11.60	• 0.72	10.50	11.30	11.20	1.27	10.20	1.44
u	6.11	12.20	0.30	12.00	• 0.78	0.00	0.09	12.6	1.06	12.7
l	5.15	11.20	1.26	11.00	• 0.18	0.96	0.87	11.6	0.10	11.80
d	2.08	8.12	4.33	7.97	3.25	• 4.03	3.94	8.52	2.97	8.69
(Y)	4.86	10.90	1.55	10.80	0.47	1.25	• 1.16	11.30	0.19	11.50
	-5.15	0.89	11.60	0.74	10.50	11.30	11.20	• 1.29	• 10.20	• 1.46

Table 5.1: The difference table searched by the dynamic programming algorithm. The strict MIDI rendering is on the x-axis and the Gould recording is on the y-axis. The bullets (•) indicate the path of best match between the two recordings.

The move with the smallest *global* cost is chosen at each stage. The cost of each move is defined as the difference between the two slope values at a given point. The DPA does not select a move based only on the minimum absolute difference in the next element, since each move affects all subsequent moves. Instead, at each step of the path, the algorithm recursively determines the global cost of each move. This is somewhat analogous to a game of chess. The expert player considers all possible outcomes of each move up to many steps ahead, not just the moves in the immediate future. In this way, the DPA can find the optimal match among all possible matches between the recordings. The optimal match path in the fugue example is indicated by the bullets (•) in Table 5.1. The match path for the entire piece is plotted in Figure 5.5.

5.3.3 Extracting tempo curves

The concept of the tempo curve is controversial. Desain and Honing (1991), prolific researchers in the field of tempo-tracking and analysis, believe the tempo curve

> "... is a dangerous notion, despite its widespread use and comfortable description, because it gives the users the false impression that a continuous concept of temporal flow has an independent existence—a musical or a psychological reality, and that time can be perceived independent of the events carrying it."
>
> —Desain and Honing (1991)

It is necessary, therefore, to be aware that the tempo curve is merely an abstraction. More specifically, a tempo curve is, for better or worse, independent of the structural details

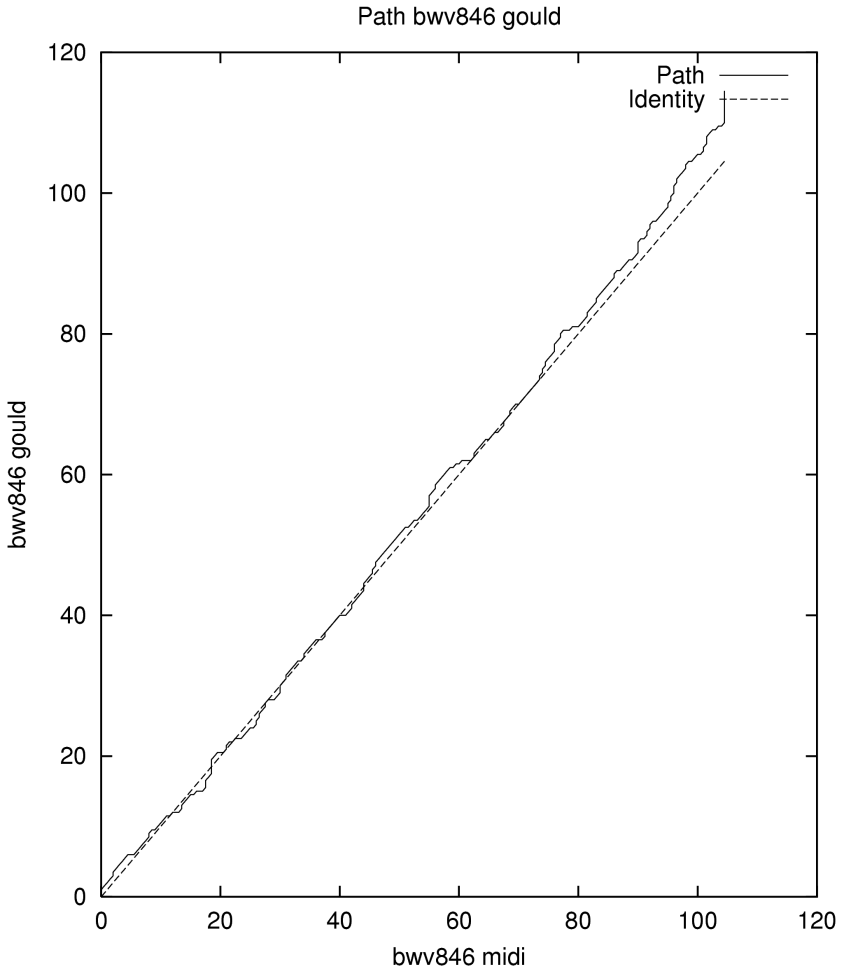


Figure 5.5: The match-path of the fugue. The strict MIDI rendering is on the x-axis and the Gould recording is on the y-axis. The line represents where the two recordings match.

to which musical tempo is inherently linked. These structural details include chord asynchronicity, ornamentation, *appogiatura*, and phrase length. Despite the abstract nature of tempo curves, Mazzola and Zahorka (1994) prove the utility of tempo curves for analysis and re-creation of performances, especially by computers.

The tempo curve that RUBATO extracts is inherently relative; it is based on where two recordings match, and it is impossible to distinguish between a *ritardando* in one recording and an *accelerando* in another. If the tempo of one of the recordings is known, however, we have a fixed point of reference for determining the absolute tempo values. One of the easiest ways to generate a recording with a strict known tempo is to use a MIDI sequencer/synthesis system. Since RUBATO is invariant to detail, the poor realism of most MIDI synthesizers does not affect the accuracy of the result. For example, choral textures are usually produced very inaccurately by MIDI sampling synthesizers, often using only a single vowel tone for all notes. However, this timbral difference is unimportant to the RUBATO system. In fact, it has worked successfully on works where the MIDI rendering has very little timbral resemblance to the original (Section 5.3.4). Creating the MIDI performance turns out to be the most time-consuming step in the process. However, some strict-tempo MIDI renderings of classical music are freely available on the world wide web (Nienhuys 2001). Forthcoming optical music recognition technology promises to automate the creation of MIDI performances directly from printed scores (Part I).

The tempo curve is derived from the match-path created by the DPA. It is, essentially, the slopes (the first derivative) of the match-path. Since the tempo of the rendered MIDI file is known, the result can be scaled to a concrete unit, such as beats per minute. The resulting graph is identical in definition to what Bowen (1993) calls a "Tempo Map." The x-axis displays the measure, while the y-axis displays the tempo for each measure in beats per minute (bpm).

The tempo curve of the Gould fugue recording is shown in Figure 5.6. The sharp peaks at either end are caused by the fact that continuous tempo is impossible to determine at the extreme beginning and end of the recording. Refinements to the extrapolation process should improve this.

At present, the resolution of the tempo curve is bounded by the window size. This makes it easy to derive coarse tempo curves, but finer-grained *rubati* generally get smoothed out. This makes it unsuitable for research into what Desain and Honing (1991) call "micro-tempo", or what Repp (1994) calls "expressive microstructure" (i.e. timing deviation on a

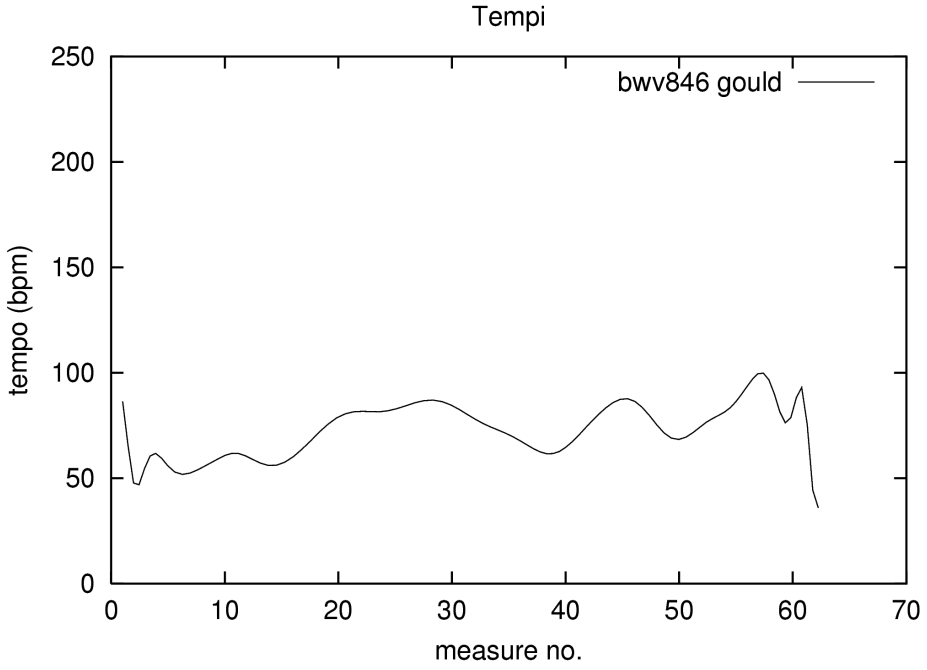


Figure 5.6: The tempo curve of Gould's 1963 recording of the Bach fugue.

note-to-note basis). Possible ways to increase the resolution include performing fine-grained dynamic programming within each window, or adding a second stage of beat-induction.

5.3.4 More examples

Two Gould *Goldberg*s

The RUBATO system is not limited to the comparison of only one recording and one MIDI rendering. In fact, any number of recordings can be cross-compared. As an example, Figure 5.7 shows the tempo curves from Gould's 1955 and 1982 recordings of J. S. Bach's *Goldberg Variations* BWV 988 (first sixteen measures). You can see that the 1955 recording is faster than the 1982 since the average bpm is higher, but it also has more tempo variation. The earlier version also has a prominent *ritardando* around measure 14.

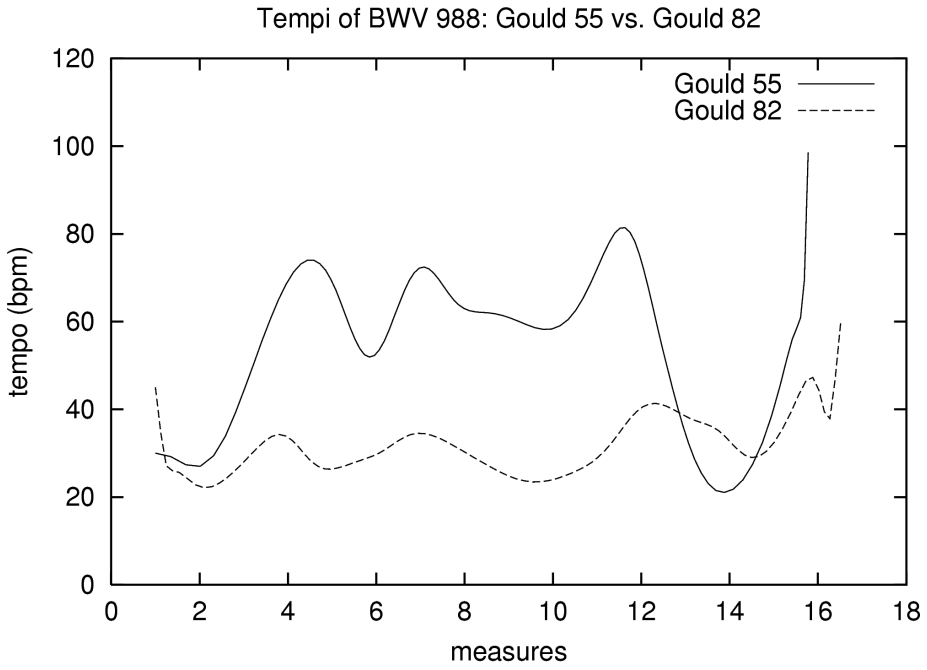


Figure 5.7: Tempo curves of Gould's 1955 and 1982 recordings of J. S. Bach's *Goldberg Variations* BWV 988 (first sixteen measures).

Complex orchestral timbres

The "In Paradisum" movement of G. Fauré's *Requiem* was chosen as a challenge for the system. Three recordings spanning three decades directed by Cluytens (1963), Yamada (1974), and Shaw (1987) were compared. There were two expected difficulties with the piece. First, it contains strings and chorus, which are reproduced very unrealistically on most MIDI synthesisers. Second, it has a very smooth gross amplitude contour, which gives the DPA fewer pronounced guide-posts to work with. However, the matching and tempo extraction of this work was very successful, proving the robustness of RUBATO's approach (Figure 5.8).

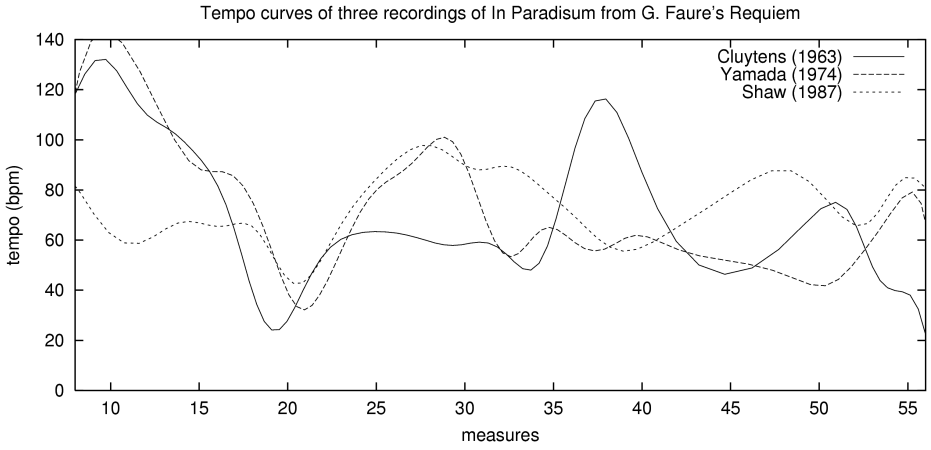


Figure 5.8: Tempo curves of three recordings of the "In Paradisum" movement of G. Fauré's *Requiem*: Cluytens (1963), Yamada (1974) and Shaw (1987).

Interpreting the results, one can see that all three recordings follow roughly the same tempo curve. These similarities are due to the marked *ritardandi* in the score. But it is interesting to note how the Cluytens has a divergent swelling of tempo around measure 38. Though it is impossible to draw conclusions from such a small sample, it is at least consistent with Bowen's hypothesis (1996) that tempo variation has decreased in recent years.

5.4 Conclusion

Research into tempo is important for the understanding and historical documentation of performance practice. However, current methods for extracting tempo data directly from recordings are cumbersome. They either include too much human error or are overly time-consuming. The RUBATO system aims to solve this problem by providing an efficient tool with which a researcher can generate large amounts of data. While its resolution is currently inadequate for many tempo-based research projects, further refinements and possible hybridization with other tempo-tracking techniques show promise.

References

Allen, P. E., and R. Dannenberg 1990. Tracking musical beats in real-time. In Proceedings, *International Computer Music Conference* 140–3.

Bellman, R. 1957. *Dynamic Programming.* Princeton, NJ: Princeton University Press.

Bowen, J. 1993. A computer-aided study of conducting. *Computing in Musicology* 9: 93–103.

Bowen, J. 1996. Tempo, duration, and flexibility: Techniques in the analysis of performance. *Journal of Musicological Research* 16: 111–56.

Bowen, J. 1999. A bibliography of performance analysis. `http://www.georgetown.edu/departments/AMT/music/bowen/perfbiblio.html`

Buxbaum, E. 1988. Stravinsky, Tempo and *Le Sacre*. *Performance Practice Review*. 1: 66–88.

Cluytens, A. 1963. *G. Fauré: Requiem.* Compact Disc. EMI 5 66946.

Cormen, T., C. E. Leiserson, and R. L. Rivest. 1997. *Introduction to Algorithms.* Cambridge, MA: MIT Press.

Desain, P., and H. Honing. 1991. Tempo curves considered harmful: A critical review of the representation of timing in computer music. In Proceedings, *International Computer Music Conference* 143–9.

Desain, P. and H. Honing. 1994a. Does expressive timing in music performance scale proportionally with tempo? *Psychological Research*. 564: 285–92.

Desain, P. and H. Honing. 1994b. Foot-tapping: a brief introduction to beat induction. In Proceedings, *International Computer Music Conference*, 78–101.

Driesse, A. 1991. Real-time tempo-tracking using rules to analyse rhythmic qualities. In *International Computer Music Conference*, 578–81.

Epstein, D. 1985. Tempo relations: A cross-cultural study. *Music Theory Spectrum* 7: 34–71.

Feldman, J., D. Epstein, and W. Richards. 1992. Force dynamics of tempo change in music. *Music Perception* 102: 185–204.

Foote, J. 2000. ARTHUR: Retrieving Orchestral Music by Long-Term Structure. In Proceedings, *International Symposium on Music Information Retrieval*.

Gale, W., K. Church, and D. Yarowsky. 1992. Using bilingual materials to develop word sense disambiguation methods. In Proceedings, *International Conference on Theoretical and Methodological Issues in Machine Translation* 101–12.

Gould, G. 1955. *J. S. Bach: The Goldberg Variations.* Compact Disc. CBS, MYK 38479.

Gould, G. 1963. *J. S. Bach: The Well-Tempered Clavier.* Compact Disc. Sony Classical, SM2K 52 600.

Gould, G. 1983. *J. S. Bach: The Goldberg Variations.* Compact Disc. CBS Masterworks, MK 37779.

Holland, B. 1981. A man who sees what others hear. New York Times, Nov. 19, 1981.

Katz, M. 1999. *The phonograph effect: The influence of recording on listener, performer, composer.* Ph.D. diss., University of Michigan.

Levitin, D. J., and Mathews, M. V. 200. The perception of cross-modal simultaneity. Technical Report. McGill University.

Mazzola, G. and O. Zahorka. 1994. Tempo curves revisited: Hierarchies of performance fields. *Computer Music Journal*, 181: 40–52.

MIDI Manufacturers' Association. 1986. *MIDI Specification, Version 1.0.* La Habra, CA: MIDI Manufacturers' Association.

Nienhuys, H. 2001. *The Mutopia Project.* `http://www.mutopiaproject.org`.

Repp, B. H. 1994. Relational invariance of expressive microstructure across global tempo changes in music performance: An exploratory study. *Psychological Research* 564: 269–84.

Repp, B. H. 1995. Quantitative effects of global tempo on expressive timing in music performance: Some perceptual evidence. *Music Perception* 13: 39–58.

Rorem, N. 1983. *The later diaries of Ned Rorem 1961–1972.* San Francisco, CA: North Point Press.

Shaw, R. 1987. *G. Fauré: Requiem.* Compact Disc. Telarc, CD-80135.

Smith, L. A., R. J. McNab, and I. H. Witten 1998. Sequence-based melodic comparision: A dynamic-programming approach. In *Melodic Similarity: Concepts, Procedures and Applications*, edited by W. B. Hewlett and E. Selfridge-Field. *Computing in Musicology*: 11.

Smith, L. A., and R. J. McNab. 1996. Melody transcription for interactive applications. Technical report, University of Waikoto, Hamilton, NZ. 96(32).

Todd, N. 1989. A computational model of rubato. *Contemporary Music Review* 3: 69–88.

Yamada, K. 1974. *G. Fauré: Requiem.* Compact Disc. CBS Masterworks, MK 44738.

Part III

Realtime digital signal processing environment

Chapter 6

RED: Realtime Environment for Digital Signal Processing

6.1 Introduction

RED is a language-based programming environment for realtime digital signal processing. The system is general enough that the signals being processed may be any form of data, including audio, MIDI and video signals. The algorithms upon these diverse forms of data intercommunicate using a flexible message passing system. This allows the user of the environment to create custom domain-specific applications that include audio filtering and synthesis, and video processing and tracking.

To use RED, one writes programs that build "patches" (signal processing graphs) connecting unit generators (UGs) together. UGs are software modules that filter or produce digital signals. Max V. Mathews originally introduced them in the Music III language developed at Bell Telephone Laboratories in 1960 (Roads 1999). This same conceptual model is used in a number of popular signal processing environments, including Max/MSP (Cycling '74 2001) and PureData (Puckette 2001). However, since RED is a programming language environment, it more closely resembles systems such as SuperCollider (McCartney 1996) and Nyquist (Dannenberg 1997).

While these other systems are more complete and mature, RED stands out for its flexibility and extensibility. It contains a complete, modern programming language, not a domain-specific subset. This means it can be easily be part of real-world applications. Also,

virtually every subsystem in RED is modular, including the signal processing schedulers, and UGs. This makes it easy for these subsystems to be adapted or replaced for particular needs.

RED, as a research system, is primarily concerned with creating a powerful and extensible infrastructure for signal processing. Therefore, this chapter discusses global architecture issues and does not go into detail about specific unit generators or synthesis algorithms.

6.2 Architecture

6.2.1 Overview

To use RED, Python programs are written that build "patches" or signal-processing graphs.[1] (See Section 6.3 for examples of these programs.) Once created, the graphs are executed (signals are processed) in realtime. For purposes of efficiency, these graphs are first sorted into a processing list using a temporal sort (Cormen et al. 1997). The processing list contains all of the UGs in the correct order needed to process the graph, as well as the assignment of buffers to the inlets and outlets of the UGs. Processing the signal, then, involves a straightforward traversal through the processing list, and does not require parsing the graph to process each block. (This is very similar to the signal processing technique used in PureData.) However, unlike PureData, RED can have multiple processing lists for different kinds of signals (e.g. audio or video). A different processing list is created for each kind of signal and these lists are assigned to signal-specific schedulers to process the data. The processing of each kind of data occurs in a separate thread. The scheduler threads can then communicate to one another through an efficient message-passing system (Section 6.2.5). This framework provides the basis for building complex, realtime, digital signal processing graphs.

6.2.2 Goals

This overall architecture has the following important features:

- **General.** RED is not tied to particular types of digital signals, but instead can process different types of data, including audio and video (Section 6.2.3).

[1] Here, the word "graph" is used not in the visual sense, but in the conceptual sense as in mathematical graph theory.

- **Modularized.** Virtually every subsystem of RED is modularized. In addition, UGs are easy to develop by C++ programmers (Section 6.2.4).
- **Realtime, low-latency performance.** RED is efficient, while still being flexible at runtime, and the audio latency is as low as possible on the respective platforms (Section 6.2.5).
- **Built from existing tools.** In an attempt to avoid re-inventing the wheel, RED is built from a number of existing, mature, open source tools (Section 6.2.6).
- **Portable.** By using tools that are already cross-platform, and avoiding the use of advanced language features, RED runs on a variety of platforms (Section 6.2.7).

6.2.3 General

RED is designed to handle multiple types of digital signals, including audio and video. There is a simple restriction, however, that allows for this generality while keeping the implementation simple and manageable: all signal processing is performed on buffers (arrays) of floating-point values,[2] and, in most cases, normalized to the range $[-1, 1]$. In the case of audio, this is an array of samples, and in video it is a two-dimensional array of either grey-scale pixel values or red-green-blue (RGB) triplets. Since all signals are processed as floating-point arrays, the only difference between the signal buffers of different types is their length. This means that the same UGs can be used to process different kinds of signals.

Authors of UGs can specify whether a UG is applicable only to a particular signal type or to all types of signals. UGs that work on all signals can be written and compiled in only one form and still process multiple types of signals. For example, a UG for addition can be used to mix audio *or* video signals. Other UGs that are signal-specific, especially video UGs that need a notion of (x, y) coordinates, may be declared to work only with particular types of signals. The graph-sorting algorithm examines these declarations and the connections to other UGs to determine which scheduler and which size of buffer to assign to each UG.

6.2.4 Modularized

RED has a highly modular design. Almost every subsystem, excepting the Python language and the message-passing system, can be loaded and interchanged at runtime. Here, the term "module" refers to Python "extension modules": separately compiled code

[2] These floating-point values can be either 32-bit or 64-bit floats depending on the most efficient method on the host processor.

objects that are loaded at runtime.[3] Most UGs, except the most basic, are grouped into modules, such that only the categories of UGs that are needed in a given application need to be loaded at runtime. Schedulers, which direct the processing of data are also modular.

Writing new UGs is accomplished through extensions to the UG classes developed for the Realtime Audio Template Library (RATL) (MacMillan et al. 2001; MacMillan 2002). These classes allow the UG programmer to focus on the development of the core algorithm, and expend less effort fitting it into the overall infrastructure of a DSP system. RATL's design also ensures efficiency by using advanced C++ features such as templates and inlining (Stroustrup 1997). As an example, the code for a very simple oscillator (`Osc`) UG is presented below. The only functions that need to be written are:

- A constructor (`Osc()`) to initialize variables.
- A member function (method) so the user can set the frequency (`freq`).
- A special `tick` member function that processes a single sample. This function is inlined into a function in the base class that processes an entire block of samples. This block processing function can be overridden if it is advantageous to write the algorithm for an entire block of samples.

Finally, a small amount of code is needed to create a Python extension module for this UG, and wrap the UG itself and any new member functions it may have. Additional convenience macros are available to provide more information to the infrastructure, such as overriding the default names of the inlets and outlets or specifying which kind of signals the UG is specialized for.

```
/*
 * Oscillator class
 *
 * A simple implementation that uses the standard C library's
 * sin() function.
 *
 * Inherits from UgenProducer (a UG that produces a signal)
 * The template argument <Osc> allows functions in this
 * subclass to be inlined in functions in the base class.
 */
```

[3] Shared objects (`.so`) under UNIX, and dynamically loaded libraries (`.dll`) on Microsoft Windows.

```
class Osc : public UgenProducer<Osc> {
public:
  // Declare that this UG works on audio signals
  SIGNAL("audio");

  // Constructor: chain to the base class and setup defaults
  Osc() : UgenProducer() {
    m_phase = 0;
    freq(440);
  }

  // freq: set the frequency of the oscillator
  // This is a new member function not in the base class
  inline void freq(sample freq) {
    m_freq = freq;
    m_step = (freq * TWOPI) / m_sr;

    // Send out a message that the frequency has changed
    send_message('freq', freq);
  }

  // tick: where each sample is processed
  // This is called from a member in the base class that
  // processes an entire buffer of samples.
  inline sample tick(sample f, sample amp) {
    m_phase += m_step;
    if (m_phase > TWOPI)
      m_phase -= TWOPI;
    return sin(m_phase);
  }

private:
  sample m_phase, m_step;
```

```
  sample m_freq;
};

/*
 * WRAPPING CODE
 *
 * This code exposes the UG to Python
 */
// Create a new module
PLUGIN_INIT(osc, "osc")
// Add this ugen
  ugen_builder<Osc> osc_obj(osc, "_Osc");
// Define any member functions not in the base class
  osc_obj.def(&Osc::freq, "freq");
}
```

6.2.5 Realtime, low-latency performance

Being a dynamic scripting language, Python is not suited for realtime processing. However, by writing the realtime portions (UGs and schedulers) in C++, other parts of the system that only need to be "adequately fast" (graph sorting, graph building) can be written in Python and reap the benefits of its dynamic data structures, lack of compilation step and automatic memory management.

Schedulers

The schedulers are separate threads that perform the actual signal processing. Since the scheduler loop has no direct interaction with Python, it is not affected by Python's inefficiencies. A scheduler must be provided for each kind of signal. At present, schedulers exist for audio, MIDI and video data, but more could be added as desired. The schedulers each run in their own thread of execution, so that the processing within the thread can be "driven" directly off of the input/output hardware. For instance, audio processing would occur every time enough samples had been received from the audio input device (i.e. sound card). It is possible to run two schedulers of the same type simultaneously (e.g. for two sound

cards on the same machine). Schedulers run autonomously, and the operating system is ultimately in charge of managing their "parallel" execution. However, the real power comes from being able to communicate between the different scheduler threads and between the scheduler threads and Python.

Message passing

These communications are performed using a specialized message-passing system based loosely on the Smalltalk programming language (Ingalls 1981). Messages are sent from unit generators to other unit generators, usually when a particular event occurs, such as a MIDI keypress. They consist of two parts: a message name and associated data. Messages can be sent from a UG by calling its `send_message` member function. When a message is received by a UG, a member function with the same name as the message name is called, with the associated data as an argument. Therefore, new ways of receiving messages can be created by simply adding new member functions to a UG. (For more details on how messages can be connected, see Section 6.3.2.) Messages can be sent and received with equal flexibility in both Python and C++. The dynamic function calling, when a message is received, is made more efficient than a typical dynamic Python function call through a combination of various caching techniques for polymorphic calls presented by Driesen (1999).

A message is sent by calling the `send_message` member function of a UG. This places the message name and data in a queue data structure (first-in, first-out (FIFO)). Items in the queue are cleared as often as possible by the central Python thread, which has a lower priority than the realtime scheduler threads. This ensures that sending a message will not interrupt a realtime thread. In practice, the latency of this approach is quite acceptable.[4]

6.2.6 Built from existing tools

RED is built from a number of existing, mature, and open-source tools that allowed it to be rapidly developed into a more advanced system than the time and resource constraints might have otherwise allowed. All of the tools are open source and compatible with the GNU General Public License (Free Software Foundation 1991). Therefore, RED is also an open-source project, and users are free to modify and extend the system. The existing tools that make up the system are described below.

[4]Experiments show that floating-point messages can be passed through up to approximately 70 objects in under 1ms on a PC running Linux 2.4 with a 500MHz Intel Celeron processor.

Core language: Python

At the core of the system is the object-oriented, interpreted language, Python (Van Rossum and Drake 2000). It was chosen for its ease-of-use, portability (Section 6.2.7) and extensibility (Section 6.2.6). Since Python has some of the common features of scripting languages (Salus 1998), namely that it can be run interactively (where each line of code is executed as it is typed), and running programs (scripts) requires no compilation, it is an ideal language for experimentation and rapid development. Python also features automatic memory management, both reference counting and garbage collection, and therefore users do not need to concern themselves with the freeing of memory when objects are destroyed: this is handled automatically by the language implementation. This automatic memory management also made the development of RED as a whole considerably easier. In addition to these language features, Python also includes a rich set of libraries, including tools for graphical user interfaces (GUI), networking, and advanced input/output (I/O). Therefore, it is easy to include the digital signal processing features of RED inside of a larger, full-featured application.

Using Python as part of a DSP environment is not a new idea, though the approach used in RED is. There are at least three other systems related to both Python and real-time audio DSP: Boodler (Plotkin 2001), the Python bindings to RTCMix (Topper 2000; Gibson 2001), and Serpent (Dannenberg 2001).

Boodler Boodler, while providing only a very basic sample playback system, includes some interesting ways to algorithmically build score and performance instructions in Python. However, it is inflexible at the signal processing level, since the individual processes are not modularized and exposed to Python.

RTCmix Recently, RTCmix added the ability to write scores in Python and Perl, in addition to the traditional Cmix language, minc. Unlike RED or Boodler, RTCmix itself runs as an entirely separate process, not a Python module, and is therefore not strictly dependent upon Python to function. RTCmix and RED work on different synthesis paradigms, however. Whereas the basic unit in RED is the unit generator, RTCmix deals with the higher-level concept of instruments.

Serpent Serpent is a new language design and implementation "inspired" by Python for use in CMU's Aura project for pervasive computing. Serpent is not strictly a synthesis system: it is more focused on realtime message-passing. The primary advantages of Serpent are described as real-time garbage collection, sub-millisecond message-passing and the ability to have multiple virtual machines. Unfortunately, those advantages gained from implementing a new language from scratch perhaps do not justify the considerable effort involved. For the purposes of real-time DSP, advanced garbage collection is not strictly necessary, as long as the garbage collection is "adequately fast" and does not interrupt the real-time processing (Section 6.2.5). Sub-millisecond message-passing can be achieved, even on modest hardware, using a specialized efficient message-passing module loaded into the standard Python virtual machine. Lastly, the ability to have multiple virtual machines comes for free with the standard Python implementation. The incremental improvements in Serpent are interesting, but there are perhaps more advantages to building upon existing standard tools.

Wrapping mechanism: Boost Python library

Boost is a collection of advanced low-level libraries for C++ (Abrahams et al. 2002a). Within that collection, the Boost Python library (`boost::python`) (Abrahams et al. 2002b) allows C++ libraries to be easily accessed from Python. This approach is considerably easier than using the standard Python/C API normally used to interface C or C++ with Python. This wrapping mechanism is what makes it possible for the core of the system to be written in Python, while the realtime critical portions are written in much more efficient C++ (Section 6.2.5).

Access to audio and MIDI hardware: PortMusic

PortMusic is a collection of programming libraries for audio (PortAudio) and MIDI (PortMIDI).

PortAudio (Burk et al. 2001) allows the programmer to develop applications with realtime full-duplex audio processing that run on multiple platforms. Programming with PortAudio is also generally simpler than programming directly for the native audio interfaces of most operating systems. Furthermore, it usually offers the best latency performance possible on all supported platforms (MacMillan et al. 2001b).

A related library still under development, PortMIDI, will provide cross-platform access to MIDI devices.[5]

Access to video capture and display hardware: wxLIVID and SDL

At present, there are no complete cross-platform libraries for accessing video hardware. However, wxLIVID (Surveyor 2001), currently under development, aims to provide easy access to video capture hardware under Linux and Microsoft Windows.[6]

To display video output, the cross-platform Simple DirectMedia Layer (SDL) (Wen 2001) library is used. Designed primarily for game development, SDL provides, among other things, a very fast method to display video.

6.2.7 Portable

Since all of the tools used in the making of RED are cross-platform, RED is also runs on many common desktop platforms. Table 6.1 shows the different platforms supported by these tools.

In addition to the use of cross-platform tools, a restricted subset of the C++ language is used that is supported by the major compilers on these platforms.[7]

6.3 Usage examples

This section will give a brief overview of how an end user/programmer may use RED, and gives some simple examples. A basic knowledge of Python is assumed. Since Python can be run interactively, RED programs can be typed in one line at a time and the results are heard immediately, or the entire script can be saved in a file to be run automatically.

6.3.1 Unit generator objects

As stated before, to use RED, one writes programs that build "patches" by connecting unit generators (UGs) together. UG instances are created, as all object instances in Python,

[5] Currently, partial MIDI support for Linux and Microsoft Windows is provided with code borrowed from Synthesis Toolkit (STK) (Cook 1999). It is hoped that some of this work can be contributed back to the PortMIDI project.

[6] Currently, basic access to the Video4Linux API (Red Hat Labs 2001) has been implemented.

[7] GNU compiler collection (gcc) for UNIX and MacOS-X, Microsoft Visual C++ on Microsoft Windows, and Metrowerks CodeWarrior on MacOS 8.x and 9.x

	UNIX	Microsoft Windows	Apple MacOS	
	FreeBSD, Linux & Solaris	95, 98, ME, NT, 2000, XP	8.x, 9.x	OS-X
Python	•	•	•	•
Boost Python	•(a)	•(b)(c)	•(c)	•(a)
PortAudio	•(d)	•(e)(f), ∘(g)	•(h), ∘(g)	∘(i)
PortMIDI		∘(e)	∘	
wxLIVID	∘(j)	∘(k)		
SDL	•(l)	•(f)	•(m)	•(m)

- • Completed and ready for widespread use
- ∘ Planned or in alpha stage of development
- (a) GNU compiler collection (gcc)
- (b) Microsoft Visual C++
- (c) Metrowerks CodeWarrior
- (d) Open Sound System (OSS)
- (e) Windows Multimedia Extensions (MME)
- (f) Microsoft DirectX
- (g) Steinberg Audio Streaming Input and Output (ASIO)
- (h) Apple SoundManager
- (i) Apple CoreAudio API
- (j) Video4Linux API
- (k) Video for Windows VIDCAP API
- (l) X-Windows with Direct Graphics Architecture (DGA) acceleration
- (m) Apple Carbon and DrawSprockets APIs

Table 6.1: Compatibility matrix of the various third party tools used to build RED.

by calling the UG's constructor and assigning the result to a variable. For instance, one can create a new oscillator (`Osc`) object with a frequency of 440Hz by typing:

```
osc = Osc(freq = 440)
```

6.3.2 Connecting signals

However, creating a UG on its own will not allow it to be heard or seen. It must first be "connected" to an audio or video output object. For each unit generator object, the signal inlets and outlets are accessed by providing an inlet name inside square brackets '`[ ]`'. For instance, to get the outlet of the oscillator object we can use the expression:

```
osc['out']
```

These inlets and outlets can be connected to the inlets and outlets of other objects using the overloaded left-shift '`<<`' and right-shift '`>>`' operators. The connection is made in the direction of the "arrows." For example, to create an oscillator object, and then connect its outlet to the a stereo digital audio convertor (DAC) output generator:

```
dac = Dac()
osc = Osc(freq = 440)
# Connect the outlet of the osc to an inlet of the Dac
osc['out'] >> dac['left']
# Connect the output of the oscillator to the other channel
dac['right'] << osc['out']
```

Multiple outlets can be connected to the same inlet, and the signals are implicitly mixed (added together). For example, two oscillators can be sent to the same output:

```
dac = Dac()
osc1 = Osc(freq = 440)
osc2 = Osc(freq = 220)
osc1['out'] >> dac['left']
# Implicitly mixes signals
osc2['out'] >> dac['left']
```

When objects are deleted, their connections are deleted as well, so it is perfectly valid, using the above example, to later destroy one of the UGs, and it will no longer produce a signal:

```
del osc2
```

Since we have a full-featured programming language at our disposal, it is also easy to create a large number of oscillators and connect them to the output. In this example, the outlet of the oscillator can be connected to all inlets of the `Dac` object using the special member `inlets`.

```
dac = Dac()
for i in range(50):
    osc = Osc(freq = 100 * i)
    # Shorthand, equivalent to
    #  osc['out'] >> dac['left']
    #  osc['out'] >> dac['right']
    osc['out'] >> dac.inlets
```

As another simple signal-connecting example, we can create a ring modulation patch by creating two oscillators and connecting them to a `Multiply` object. In this case, we're using the convenience feature that the oscillator outputs can be passed in to the constructor of the `Multiply` object.

```
osc1 = Osc(freq = 440)
osc2 = Osc(freq = 100)
# Create a new Multiply object, and connect its two inlets
# to the outputs of osc and osc2
mult = Multiply(osc1['out'], osc2['out'])
dac = Dac(mult['out'], mult['out'])
```

Connecting messages

Messages are sent from unit generators to other unit generators, usually when a particular event occurs, such as a MIDI keypress. They consist of two parts: a message name and associated data. Messages can be sent from an object by calling its `send_message` member function. When a message is received by a unit generator, a member function with the same name as the message name is called, with the associated data as an argument. Therefore, new ways of receiving messages can be created by simply adding new member functions to a unit generator.

Messages connections are accessed from the object by specifying a message name in parentheses '`( )`' and connected, in the same way as signals, using the '`<<`' and '`>>`' oper-

ators. For example, to connect the `x` of the `Mouse` object (which reports the x position of the mouse pointer) to the frequency of an oscillator:

```
mouse('x') >> osc('freq')
```

To connect all messages emitted by one unit generator to another, simply do not specify a message name:

```
osc >> osc2
```

Often it is useful to modify the value of a message as it is passed to another unit generator. For this reason, function adaptors are provided. The `FA` (function adaptor) object constructor takes as its arguments a function, and one or more emitting messages. For example, to scale the output of the mouse object to be in a more audible frequency range:

```
def scale_freq(x):
  return 500 + x * 0.9


osc('freq') << FA(scale_freq, mouse('x'))
```

Of course, it is often more convenient to use Python's `lambda` notation and avoid the function `def`inition altogether:

```
osc('freq') << FA(lambda x: 500 + x * 0.9, mouse('x'))
```

Function adaptors can also be used to combine the results of two messages into one. Say for example, frequency was to be a function of both mouse position and MIDI note:

```
osc('freq') << FA(lambda x, y: y + x * 0.01,
                  mouse('x'),
                  midi_in('freq'))
```

In this case, anytime either the mouse position or MIDI note changes, a message will be sent to the oscillator that is a function of the two. This behavior can be changed, using the `only_on` keyword argument, so that, for instance, a message is only sent when the mouse moves, not when the MIDI note changes:

```
osc('freq') << FA(lambda x, y: y + x * 0.01,
                  mouse('x'),
                  midi_in('freq'),
                  only_on=0)
```

6.4 Conclusion

RED offers flexibility, extensibility and portability. It provides the foundation for both experimentation with signal processing algorithms and the construction of large end-user applications. Given the power of graphical user interface (GUI) programming in Python, it should provide a useful basis for large GUI applications that contain signal processing, such as a Max-like environment or video production tool. Future developments will add more UGs to the system and further refine the performance of the core infrastructure. Due to its modular nature, it should be easy for multiple geographically disperse developers to improve it cooperatively.

References

Abrahams, D., D. Adler, B. Dawes, J. Maddock, and J. Maurer. 2002a. Boost libraries for C++. Computer software. `http://www.boost.org`

Abrahams, D., U. Koethe, R. W. Grosse-Kunstleve, et al. 2002b. Boost Python library. Computer software. `http://www.boost.org/libs/python/`

Burk, P., R. Bencina, D. Gibbs, D. Repetto, S. Letz, P. Suurmond et al. 2001. PortAudio: Portable audio library. Computer software. `www.portaudio.com`

Cook, P. R. and G. Scavone. 1999. The synthesis toolkit (STK), version 2.1. In Proceedings *International Computer Music Conference.*

Cormen, T., C. E. Leiserson, and R. L. Rivest. 1997. *Introduction to Algorithms.* Cambridge, MA: MIT Press.

Cycling '74. 2001. Max/MSP. Computer Software. `http://www.cycling74.com`

Dannenberg, R. 1997. The Implementation of Nyquist, a Sound Synthesis Language. *Computer Music Journal* 21(3): 71–82.

Dannenberg, R. et al. 2001. An introduction to Serpent. `http://www-2.cs.cmu.edu/afs/cs.cmu.edu/project/music/web/aura/serpent.htm`

Driesen, K. 1999. Software and hardware techniques for efficient polymorphic calls. Ph. D. diss., University of California, Santa Barbara.

Free Software Foundation. 1991. GNU General Public Licence. `http://www.gnu.org/licenses/gpl.html`

Gibson, J. 2001. Python bindings for RTCmix. Computer Software. `http://www.music.columbia.edu/cmix/`

Ingalls, D. H. H. 1981. Design principles behind Smalltalk. *BYTE Magazine*, August 1981.

MacMillan, K., M. Droettboom, and I. Fujinaga. 2001a. A system to port unit generators between audio DSP systems. In *International Computer Music Conference.* In press.

MacMillan, K., M. Droettboom, and I. Fujinaga. 2001b. Audio latency measurements of desktop operating systems. In *International Computer Music Conference.* In press.

MacMillan, K. 2002. Master's Thesis. Peabody Institute of the Johns Hopkins University.

McCartney, J. 1996. SuperCollider: A new real-time sound synthesis language. In *International Computer Music Conference* 257–8.

Plotkin, A. 2001. Boodler: a programmable soundscape tool. Computer software. `http://www.eblong.com/zarf/boodler/`

Puckette, M. S. 2001. Pure Data. Computer software. `http://www.pure-data.org`

Red Hat Labs. 2001. Video4Linux API Version 2.2. Computer software. `http://roadrunner.swansea.linux.org.uk/v4l.shtml`

Roads, C. 1999. *The computer music tutorial.* Cambridge, MA: MIT Press.

Van Rossum, G., and F. L. Drake. 2000. *Python tutorial.* Campbell, CA: iUniverse.

Salus, P. H., ed. 1998. *Handbook of programming languages, Volume III: Little languages and tools.* Indianapolis: MacMillan Technical Publications.

Surveyor Corporation. 2001. wxLIVID: A multi-platform, open-source approach to video capture, processing and display. Computer software. `http://www.surveyorcorp.com/wxLivid/`

Stroustrup, B. 1997. The C++ programming language. 3rd ed. Reading, MA: Addison-Wesley.

Topper, D. 2000. RTCmix for Linux: Part 1. *Linux Journal* 178.

Wen, H. 2001. SDL: The DirectX alternative. O'Reilly Network. 09/21/2001.

Acknowledgements

I wish to acknowledge the assistance of Dr. Ichiro Fujinaga at the Peabody Institute of the Johns Hopkins University.

The work in optical music recognition was conducted as part of the Lester S. Levy Collection of Sheet Music Digitization Project of the Special Collections and Digital Knowledge Center branches of the Milton S. Eisenhower Library at Johns Hopkins University. Funding was provided in part by the National Science Foundation, the Institute for Museum and Library Services, the International Symposium on Music Information Retrieval and the Levy family.

I wish to thank G. Sayeed Choudhury, Tim DiLauro, Holger H. Hoos, Karl MacMillan, Mark Patton, Kai Renz and James Warner for their assistance with this research.

Vita

Michael Droettboom received an Honours Bachelor of Arts degree in Computer Science, with a minor in Music, from York University in 2000. He is currently employed by the Digital Knowledge Center of the Milton S. Eisenhower Library developing software for the automatic recognition of cultural heritage documents, including sheet music. He is also an award-winning classical vocalist and composer.