Some Lisp tools for musicology

Research aids for an Electronic Corpus of Lute Music

David Lewis*, Christophe Rhodes†

Centre for Cognition, Computation and Culture
Goldsmiths College, University of London
New Cross, London, SE14 6NW

28th June 2006

Abstract

We present the tools that have been developed in the course of implementing a resource to assist in musicological investigation of lute music. We discuss the use we have made of current Lisp-based technologies in our applications, and the modifications and customizations we have needed to make. Our tools both have the current ability to service musicological queries, and have an evident path for further development to enhance their utility.

1 Introduction

There is a long history of association between musicology and computer technology. The term 'Music Information Retrieval' dates back at least as far as 1966 [1], and the earliest computer-based music encoding and analysis initiative, centred around the works of Josquin des Prez, was started by Lewis Lockwood and Arthur Mendel in Princeton at 1963 [2]. It is perhaps surprising, therefore, that the impact of computing on the discipline has been negligible. One reason for this is the complexity of some of the tasks required: music analysis is frequently very difficult or even impossible to specify algorithmically and artistic, interpretative or creative processes pose special challenges. This is not, however, the case for all musicological tasks, and probably the greatest impediment to routine, productive use of computers for musicology lies in a severe lack of encoded data.

Lisp's history predates even the use of computers in musicology; the vibrancy of the user community has ebbed and flowed over the last 45 years, but it is relatively clear that Lisp is emerging from the downturn of the AI Winter: over the last five years, many developments (including implementation maintenance, library development, and the continuation of Moore's law) have made it easier for users to run powerful software on their desktop computers.

The remainder of this paper describes a new application of Lisp to musicology: in section 2 we discuss previous applications of computer technology to musicological tasks, and introduce our own field of investigation; section 3 describes our present and future applications of Lisp to the field, and we conclude in section 4.

2 Computational Musicology

Given the long history of association between musicians and musicologists and computer technology, it may seem surprising that computational musicology should still suffer from a lack of data. There are

*d.lewis@gold.ac.uk
†c.rhodes@gold.ac.uk

many large resources of recorded music in the form of audio files, and websites such as the 'Classical Music Archives' [3] store a significant number of MIDI files, but here our problems start to become more apparent. Much existing musicology operates on some sort of structured, graphical musical notation, usually a score; automatically extracting the high-level constructs required from an mp3 or wav file remains a difficult problem. MIDI operates in a more suitable domain, specifying pitch and rhythm information directly, but is still lacking in many notational features vital for our purposes.

An important resource for musical scores is the MuseData collection based at the Center for Computer-Assisted Research in the Humanities at Stanford University[4]. This data-set contains in excess of 4,000 movements, most of which are by Bach, Handel, Telemann, Haydn, Corelli, Vivaldi, Mozart or Beethoven. Despite the size of the resource, there remain several methodological difficulties with using this collection as anything more than a test set for new algorithms.

Firstly, the scores presented do not in any way represent the original state or states of the music. For copyright reasons, the creators of the MuseData collection have been unable to use new critical editions. Lacking the time and resources to transcribe from contemporary sources either, they have resorted to relying on the best available nineteenth-century editions. These are, unfortunately, of limited scholarly value – many vital sources have come to light in the intervening period, and even in the more scrupulous editions of the period, editorial 'corrections' and 'updatings' to the music do occur. To make historically-informed observations based on these is thus problematic.

An additional issue is the selection of music. Although drawn predominantly from the early eighteenth century, works in the data-set range from the mid-seventeenth century to, in one case, the early 1890s. The selection of composers is largely limited to the traditional canon of 'geniuses', including no composers who are not Italian or German. Since a basic element of any attempt to draw historically valid conclusions from music must be a representative sample of works from the time, this practice of selecting pieces and composers considered to be of exceptional style or quality and omitting those whose posthumous reputation does not match their contemporary influence can only reduce the usefulness of the resource.

2.1 Lute Music

It is in this context that the Electronic Corpus of Lute Music (ECOLM) must be considered, since music for the lute provides a population of music sources that is coherent, chronologically bounded and of great historical and musical interest. For a period spanning from at least the end of the fifteenth century till the latter half of the eighteenth, the lute was one of the most popular instruments in Europe, with only keyboard instruments as substantial competitors, and there are estimated to be nearly 60,000 extant pieces for the instrument[5]. That the lute's rôle in music history is consistently underestimated is partly due to the miscellaneous nature of many of its sources, but perhaps mostly due to its notation which bears little resemblance to that used by most other classical music.

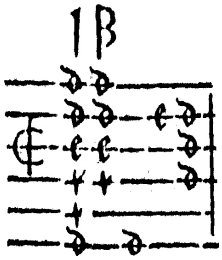
2.2 Tablature and *TabCode*

Lute music is written using one of several varieties of *tablature* notations, which give players physical information about the strings to play, where to put their fingers and how long to wait between one event and the next. The music progresses left to right, from the top system[1] of a page to the bottom. On the whole, the notation is sequential in nature, having few symbols that overlap from one chord to another and almost no way of indicating different rhythmic patterns occurring simultaneously (an important feature of modern staff notation).

This attribute of sequential ordering and discrete separation of events makes the encoding of tablature into a text string a more straightforward process than it is for most other musical notations, and several schemes have been devised for this. We use *TabCode*[6] for our encodings, in which discrete events are translated into whitespace-separated 'tabwords'[2]. Characters representing rhythm signs,

[1] A musical line.

[2] 'Whitespace' here includes line breaks.



```
M(C/)
Hd1d2e3f4f5d6
Ed1d2e3f4
d6
e2
d2d3d4
|
```

Figure 1: The first bar of a prelude attributed to the sixteenth-century composer Laurencini, 'PRæl. Laurenc.', in Jean-Baptiste Besard, *Thesaurus Harmonicus* (1603), f.12v, and its *TabCode* encoding. For a transcription to staff-notation, see Figure 4 (right).

where present, begin the tabword, followed by letter-number pairs for each symbol representing a fret and a course[3] (Figure 1 shows an example of a tablature and its transcription).

2.3 ECOLM and the identification of tasks

The Electronic Corpus of Lute Music was not funded solely in order to provide a resource to increase the availability and understanding of lute-related music, although this is clearly a central goal. It was explicit from the start that another objective was to explore computer tools for musicological and related work on large music corpora.

Several important tasks in the context of ECOLM were identified. Not all have involved Lisp in their solution, nor is it intended that they should; nevertheless we give a general overview of them, and then deal in detail with those that do. The tasks divide broadly into 4 categories: **music input**, which can be manual – *TabCode* was designed to be input using a simple text editor – or automated, and we have been involved in the development of OCR tools for lute tablatures[4]; **editing and annotation**, which might include corrections, editorial intervention, cross-references with other versions of the same piece, etc.; **presentation**, which primarily consists of web application development, but also includes transcription from tablature to score and the co-ordination of different visualisations of the music: perhaps a facsimile of the source with an edition for comparison, or the presentation of divergences between different sources; finally, we have **music processing**, including, with various degrees of automation, Music Information Retrieval and musical analysis at the level of the individual piece or of an entire corpus.

The bulk of our discussion below concerns the software developed to facilitate editing and entry. This is partly because this is the most intensely developed software to come out of the project and partly, as we shall demonstrate, because it forms the foundation on which further developments may be built.

3 Applications

3.1 Editing and entry

The project to write an editor for *TabCode* arose from dissatisfaction with the previous system of data entry by text editor with subsequent rendering and editing using an approximately WYSIWYG graphical interface, Tim Crawford's TabProcessor[5]. Since these two processes were necessarily distinct – the TabProcessor had no text view and limited content-editing functionality – the procedure could be

[3] A string, or a pair of strings tuned to similar pitches and struck together.

[4] Since these were built with Python and C++ (using the Gamera[7] toolkit) we do not discuss them further.

[5] A Macintosh program written in C for Mac OS 7.6 and not updated since c. 1997. This could read and write *TabCode*, although its primary data-entry method was by mouse and keyboard.

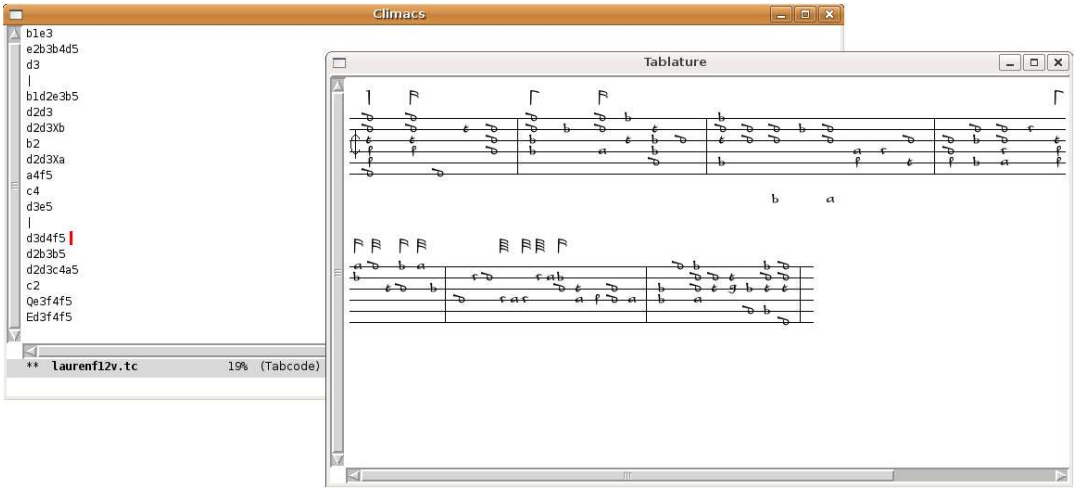


Figure 2: A screenshot of the editor component of our suite of tools. The left-hand (partially obscured) window is the text editor component, and contains some *TabCode*, which is rendered in the right-hand (front) window. Note that the first bar in the rendered output corresponds to Figure 1.

cumbersome. What was required was a system that retained the direct text editing – not only because this method is quicker for most entry tasks, but also because the text contains more information than a graphical representation can – but which still provided instant visual and audible feedback.

The application that we describe here (see Figure 2 for a screenshot) is written with Climacs, an emacs-like editor with its user interface implemented using the Common Lisp Interface Manager (CLIM) [8, 9, 10]. Although Climacs includes a syntax analysis module with various advanced parser frameworks [11], the comparative simplicity of *TabCode* and the nature of our requirements (such as a correct parse tree of the whole buffer after each edit, for use in display) led us to implement a combined lexer and parser hooking in to Climacs's buffer update protocol.

3.1.1 Parsing

Since the parser is required to operate after every editing action, it will frequently encounter half-completed tabwords that formally give rise to parse errors. On such an error, the parser preserves the partial parse leading up to the error, advances to the nearest lexically following whitespace and resets the analyser's state. Syntactically invalid code is coloured in red to alert the user without direct interruption.

The parser generates a sequence of `tabword` objects (corresponding to the 'tabwords' described earlier) from the text in the editor buffer, incorporating previously generated sequences if it can prove, based on markers showing the extent of the text region 'damaged' by user interaction, that the parse for that sequence is unchanged.

The sequence of tabwords is then parsed for 'system breaks' and 'page breaks'. Since we attempt to encode the source as it appears on the page, the points at which the music spans systems or pages are recorded in the *TabCode*. This provides very useful higher levels of structure with which to operate, as we shall see.

3.1.2 Incremental redisplay

Especially on slower machines, redrawing a long piece can have a significant effect on application speed. Climacs's `incremental-redisplay` functionality checks the identity of objects to be drawn against that of previously drawn objects in its output records. If the object has already been drawn, it will not be

redrawn. By providing the `incremental-redisplay` with a method for drawing whole systems, we can use this functionality to redraw only those that have changed since the last drawing operation. Within this system-drawing method, is a method that is also called with `incremental-redisplay` for drawing tabwords. Thus the amount of redrawing can be minimized quite precisely.

There is a trade-off to be had, however. The `incremental-redisplay` functionality also checks to see if a redisplay needs to update previously drawn objects if they overlap an output record which has moved or been erased. This involves testing all moved or erased output records for overlap with all other output records, taking (as currently implemented) $O(n^3)$ time for cases where the output records are irregularly sized and not aligned to baselines. Our solution for this has been to provide our own method, specialised on the tablature display window, that compares the bounding-rectangle of the moved and erased records with all other records, resulting in a more manageable $O(n)$. See Appendix A for details.

3.1.3 Displaying fonts

The CLIM specifications are somewhat elliptical on the subject of font abstractions. CLIM protocols for requesting specific fonts exist, such as `make-device-font-text-style`; however, there is unfortunately no specification for how this request is serviced, nor how the programmer should identify a given font. While a standard X11 font (along with all the resources required to display text and calculate text bounding boxes) is uniquely identified by its resource string[6], using a TrueType font on an X11 display requires a pathname to the font file, not simply a string; using a Postscript font for generating printable output requires two pathnames: one for a file containing information about individual glyphs, and one for a file containing the font's metrics (properties of individual and combinations of glyphs such as kerning, advance width and baselines).

We have implemented `device-font-name` structures to deal with this problem: rather than assuming that a string names a device font, we provide a function per CLIM backend (X11, freetype and Postscript) for making an opaque object which contains enough information to identify all the resources needed for using a given font. This is not necessary for the standard text fonts, which McCLIM (the implementation of CLIM that we use) maps to the Bitstream Vera fonts itself in its X11/freetype backend in a manner transparent to the programmer; however, it enables us to render tablature using fonts which have been designed after historical sources for the graphical elements (see the right-hand portion of Figure 2).

Additionally, the editor application refers to these fonts in a manner which allows it to function irrespective of its filesystem location; we use the `sb-ext:*core-pathname*` extension from Steel Bank Common Lisp (SBCL) to search for the font files relative to the delivered application as well as in system-standard locations.

3.1.4 Playback

The editor was developed for deployment, initially at least, on Apple Macintosh computers running OS X, so MIDI playback was developed in the first instance by using Apple's C API for CoreMIDI. This is capable of giving instant audio feedback, playing either extracts of the piece in the buffer or simply the chord on which the cursor is located. A Linux implementation of this functionality, using external processes, is straightforward.

3.2 Future work

3.2.1 Text criticism and source studies

TabCode as it stands is capable of encoding one view of one text of a piece of music. If we wish to be involved as editors or if we wish to compare the presentation of the same piece in a number of sources, we need a way of enriching the code to represent these aspects of *text criticism*.

[6] e.g. `"-adobe-courier-bold-r-normal--10-100-75-75-m-60-iso10646-1"`.

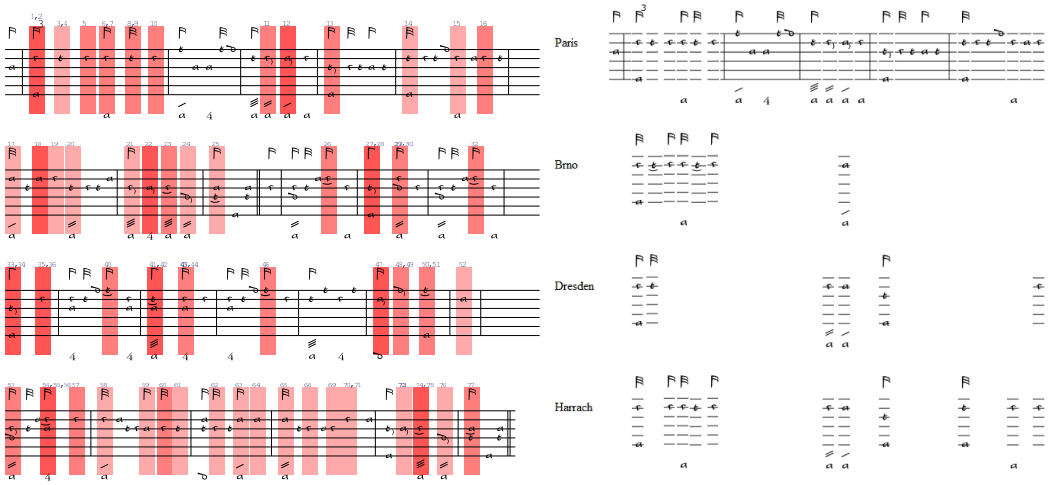


Figure 3: Two modes of (web) presentation for tablature generated using our system – designed for and described in [14] – give views of a Bourrée by Weiss that occurs, with differences, in four different sources. In the example on the left, points of discrepancy are coloured based on the number of sources that disagree with the presented version, with darker rectangles indicating more controversial readings. In the right-hand example, all points of disagreement are given in parallel with a reference text, this gives more detail, but takes more space (the music presented is that of the first system of the left-hand example). In both cases, the reference text is selected by the user.

Preliminary work in association with Frans Wiering has resulted in the initial development of TabXML, an XML implementation of TabCode incorporating a more general text-critical mark-up for time-sequence ordered music[7], conforming (largely) to Text Encoding Initiative (TEI[12]) guidelines[13]. These developments pose a significant challenge for the tablature editor's way of operating, since the textual editing of XML is not only time-consuming, but it also provides more opportunities for local editing to have dramatic global effects.

Currently, our work with *TabXML* has limited the role of the tablature editor to generating graphics from *TabCode* arising from XSLT processes (see below). The editor's future utility must be partly judged by its ability to support some degree of enriched *TabCode*, whether that enrichment is provided by XML or some other means.

3.2.2 Web delivery

Presenting the information from the corpus to a distributed user base requires web site generation. Currently, a PHP web application takes data from the MySQL database and elsewhere and presents it to the user. In the case of tablature, a socket is opened to a waiting Lisp process, and the relevant *TabCode* is piped through. The web application then receives the required result, either a PNG (generated using McCLIM's postscript backend and ghostscript tools) or a MIDI file.

The data may then be presented in a manner appropriate to the current context. Figure 3 gives two examples of different modes of presentation for this, illustrating ways of viewing deviations between sources, as extracted from *TabXML* using XSLT.

For historical and practical reasons, ECOLM's database-handling, text-critical processing and tablature rendering these have been developed separately, and using different technologies. Clearly, there is a strong argument for integrating these. Not only would this enable more efficient caching, simpler maintenance and more flexibility of presentation, but it would also reduce the barrier between the data in *TabCode* and that in the rest of the database.

[7] The issue of music encoded one part at a time has not yet been fully explored.

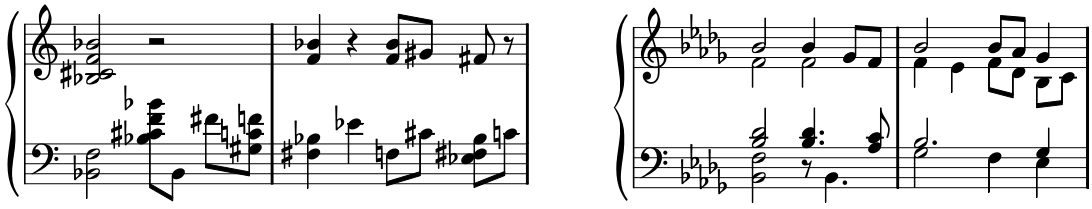


Figure 4: An extract as rendered in Sibelius automatically from MIDI (left) and with some manual intervention (right). The source is the first two bars of the prelude used in Figure 1. Important elements of staff notation, such as note duration, pitch name and key are absent from lute tablature, so naïve transcription (as on the left) is of limited use.

As the sophistication of the *TabCode* processing increases, the facility for turning our software to searches and to musical analysis increases. Some preliminary work has already been presented (in [15], with a more general description in [16]), but this made no direct use of the ECOLM database. Since this database contains information about dates, people and places, where they are available, for all the music in the corpus (and much that is not), computer-based music information retrieval (MIR), with correlations drawn from musical and extra-musical data becomes an exciting possibility to explore.

3.3 Transcription to staff notation

Lute tablature is a notation that can be read fluently by a very small proportion of those who might be interested in the musical content of our corpus. We have already seen that basic MIDI output is possible, and this might make it seem that staff output would be a straightforward development from this.

Figure 4 (left) shows the result of importing one of the MIDI files we generate from *TabCode* into Sibelius – one of the most popular pieces of music engraving software on the UK market. This is wholly inadequate as staff-notated music, as the structure of the music is not evident from the staff notation. The problem arises from the fact that tablature notation specifies neither pitch names nor durations for notes, two elements that are essential elements of staff notation; Sibelius is being forced to make decisions about these elements, and defaults to a set of internal rules for deducing these.

Pitch naming and spelling is a music-theoretic construct and, as such, can be deduced from a piece in which only absolute pitch is given. Some considerable recent success has been achieved using David Meredith's *ps13* family of algorithms [17] for this. The algorithm is known to outperform others in more mainstream musical realms, but appears also to be effective even in music from earlier periods. Analysing duration, and consequently voicing information, remains a challenge for the future.

Once transcriptions of sufficient quality are possible, we hope to use Robert Strandh's ongoing CLIM-based score-editor project, GSharp, to enable the direct visualisation of passages of tablature from within the editing environment and its presentation in modern score notation over the web.

4 Conclusions

We have presented a set of tools that we have developed to assist musicological investigation of a historically important repertoire. The majority of these tools have been developed using Lisp, and their development has itself spurred improvements in the libraries that they are built on. There remains interesting work to be done in the future, both in refining and improving the existing tools and in their application to new musicological questions; we believe that the quality of current Lisp software is sufficient to enable us to perform this work with good efficiency.

All software discussed in this paper is available under Free Software terms: SBCL, CLX, McCLIM and Climacs from their respective project homes; the *TabCode* parser and its integration into Climacs

(as well as the modifications to the Gamera optical document recognition framework) from the ECOLM project home at `http://www.ecolm.org`.

5 Acknowledgements

We thank Alastair Craft for his assistance in producing figures for this paper. D.L. and C.R. are supported by AHRC grant B/RE/AN9717/APN15559 and EPSRC grant GR/S84750/01 respectively.

References

[1] M. Kassler, "Toward Music Information Retrieval," *Perspectives of New Music*, vol. 4, pp. 59–67, Summer 1966.

[2] J. Morehen and I. Bent, "Computer Applications in Musicology," *The Musical Times*, vol. cxx, pp. 563–6, 1979.

[3] Classical Archives, LLC, "Classical Music Archives." `http://www.classicalarchives.com`, 1994–.

[4] W. Hewlett *et al.*, "MuseData: An Electronic Library of Classical Music Scores." `http://www.ccarh.org`, 1984–. From Center for Computer-Assisted Research in the Humanities.

[5] A. J. Ness and C. A. Kolczynski, "Sources of lute music," in *The New Grove Dictionary of Music and Musicians* (S. Sadie and J. Tyrrell, eds.), vol. 23, pp. 39–63, London: Macmillan, 2001.

[6] T. Crawford, "Applications Involving Tablatures: *TabCode* for Lute Repertories," *Computing in Musicology*, vol. 7, pp. 57–59, 1991.

[7] M. Droettboom, K. MacMillan, and I. Fujinaga, "The Gamera framework for building custom recognition systems," in *Symposium on Document Image Understanding Technologies*, pp. 275–86, 2003.

[8] S. McKay, "CLIM: The Common Lisp Interface Manager," *Communications of the ACM*, vol. 34, no. 9, pp. 58–59, 1991.

[9] R. Strandh and T. Moore, "A Free Implementation of CLIM," in *International Lisp Conference*, (San Francisco), Franz Inc., 2002.

[10] R. Rao, W. M. York, and D. Doughty, "A guided tour of the Common Lisp interface manager," *ACM SIGPLAN Lisp Pointers*, vol. 4, no. 1, pp. 17–37, 1990. Updated by Clemens Fruhwirth, 2006.

[11] C. Rhodes, R. Strandh, and B. Mastenbrook, "Syntax Analysis in the Climacs Text Editor," in *International Lisp Conference*, (Stanford), 2005.

[12] The Text Encoding Initiative Consortium, "TEI: Yesterday's Information Tomorrow." `http://www.tei-c.org`, 2005.

[13] F. Wiering, "Creating an XML vocabulary for encoding lute music," in *Proceedings of the XVIth International Conference of the Association for History and Computing, Amsterdam, 14-17 September 2005*, 2005.

[14] F. Wiering, T. Crawford, and D. Lewis, "Digital Critical Editions of Music: a multidimensional model," in *ICT Methods Netword Expert Seminar: Modern Methods for Musicology* (T. Crawford and M. Deegan, eds.), (London), in press, 2006.

[15] M. Gale and D. Lewis, ""La battaglia": a computer-assisted approach to an extended musical family." Presented at the 51st Annual Conference of the Renaissance Society of America, April 2005.

[16] D. Lewis, T. Crawford, G. Wiggins, and M. Gale, "Abstracting Musical Queries: Towards a musicologist's workbench," in *Computer Music Modelling and Retrieval: Third International Symposium, Pisa, Italy, 2005* (R. Kronland-Martinet, T. Voinier, and S. Ystad, eds.), Lecture Notes in Computer Science, Springer, forthcoming, 2006.

[17] D. Meredith and G. A. Wiggins, "Comparing pitch spelling algorithms," in *Proceedings of the Sixth International Conference on Music Information Retrieval (ISMIR), 2005* (J. D. Reiss and G. A. Wiggins, eds.), pp. 280–287, 2005.

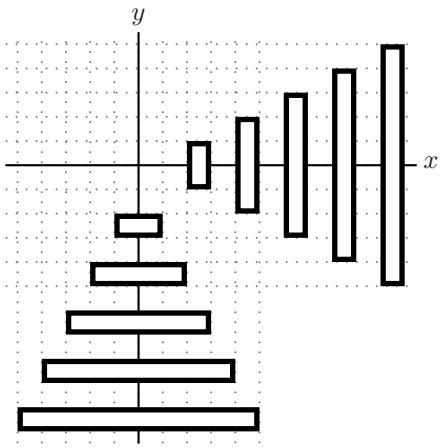


Figure 5: A rectangle set. This arrangement of rectangles illustrates the worst case for incremental redisplay: for $x - y$ bands, and $2k$ rectangles, there are more than k bands having an average of k intervals in their interval sums, and by the figure's symmetry, there is no advantage to $y - x$ bands. Thus the space requirements for the data structure are $O(k^2)$.

A Incremental Redisplay

A.1 Interval Sums

An interval is a tuple (x, u), representing the co-ordinate range $[x, u]$. An interval sum I is a sequence of co-ordinates $(x_0\, u_0 \ldots x_i\, u_i \ldots)$, representing the sum of the intervals $\oplus_i(x_i, u_i)$; for instance, the range defined by $\{x : 0 \leq x \leq 2 \vee 5 \leq x \leq 7\}$ is represented by the sequence `(0 2 5 7)`.

As implemented in McCLIM 0.9.2, every binary interval sum operation (union, intersection, difference) has the same complexity[8], and is described in algorithm 1. In that description, t_a (t_b) is a boolean representing whether we are considering a point inside or outside interval sum A (B); c_0 if non-null is the start point of an interval in the result interval sum.

Denoting the number of intervals in an interval sum I by $|I|$, there are $2(|A| + |B|)$ entries in the lists representing the interval sums, each of which must be processed in algorithm 1. Therefore, the complexity of interval sum operations as implemented in McCLIM is $O(|A| + |B|)$.

A.2 Bands and Rectangle-Sets

A band β is a tuple (y_j, I_j). A rectangle set is a list of bands, representing the region S such that $(x, y) \in S$ if and only if $x \in I_j$ for $y \in [y_j, y_j + 1]$. For example, the region of the plane consisting of two unit squares, centred at $(0.5, 0.5)$ and $(2.5, 3.5)$, would be represented as the list `((0 (0 1)) (1 nil) (3 (2 3)) (4 nil))`; the union of this region and another unit square centred at $(4.5, 3.5)$ would be represented as the list `((0 (0 1)) (1 nil) (3 (2 3 4 5)) (4 nil))`

Denote the coordinate of a band β as $y(\beta)$, and the interval sum as $I(\beta)$. Then, eliding some details of canonization and edge cases, a rectangle-set operation is carried out using algorithm 2. There are $O(|R|)$ bands in each rectangle set (see figure 5 for an example construction demonstrating this), each of which contains $O(|R|)$ intervals in its interval sum. So the complexity of a rectangle set operation is $O((|R_1| + |R_2|)^2)$.

[8] it would probably be trivial to optimize common cases of intersection and difference.

Algorithm 1 Interval sum operation **isum-op**

$t_a \leftarrow 0$, $t_b \leftarrow 0$, $c_0 \leftarrow$ nil
while A, B not both empty **do**
  if first(A) = first(B) **then**
    $c_t \leftarrow$ first(A)
    $t_A \leftarrow \overline{t_A}, t_B \leftarrow \overline{t_B}$
    pop from A, pop from B
  else
    $c_t \leftarrow \min_I(\text{first}(I))$
    $C \leftarrow \text{argmin}_I(\text{first}(I))$
    $t_C \leftarrow \overline{t_C}$
    pop from C.
  end if
  if boole$(\text{op}, t_a, t_b) \wedge c_0$ is not nil **then**
    accumulate the interval (c_0, c_t)
    $c_0 \leftarrow$ nil
  else if boole$(\text{op}, t_a, t_b) \wedge c_0$ is nil **then**
    $c_0 \leftarrow c_t$
  end if
end while
return accumulated interval sum

Algorithm 2 Rectangle Set Operation

$a \leftarrow$ nil, $b \leftarrow$ nil, $z_0 \leftarrow$ nil
while A, B not both empty **do**
  $I \leftarrow$ **isum-op**(op, a, b)
  accumulate (z_0, I)
  $z_1 \leftarrow \min_C(y(\text{first}(C)))$
  if $z_1 = y(\text{first}(A))$ **then**
    pop from A
  end if
  if $z_1 = y(\text{first}(B))$ **then**
    pop from B
  end if
  $a \leftarrow I(\text{first}(A))$, $b \leftarrow I(\text{first}(B))$
end while

A.3 Incremental Redisplay algorithm

The critical loop in incremental redisplay is building the region over which to replay the output history. This is done by looping over output-records which need to be redrawn as part of the redisplay due to some other record which used to overlap with it having been moved (`move-overlapping` in Algorithm 3) or erased (`erase-overlapping`), performing a region-union rectangle set operation at each stage. Since each rectangle set operation is $O(n^2)$, the overall complexity in building the final data structure is proportional to $\sum_n^N n^2$, or $O(N^3)$. Note also that after building this data structure which is $O(N^2)$ in space (see figure 5), there is another step, replaying the output history in that region. This involves K region-intersection computations with this data structure, which has a time complexity of $O(KN^2)$, comparable with the time spent building the structure in the first place.

It should be noted that the current implementation does not perform poorly for all applications. In particular, for ordinary text or other non-intersecting small data (such as traditional GUI widgets), the $O(N^3)$ case is not typically reached. However, with our complex tablature layout (with multiple overlapping graphical elements: see Figure 2), it was necessary to customize incremental redisplay to achieve acceptable performance.

A.4 Customization

We customize the `incremental-redisplay` generic function for display on the window displaying the rendered tablature: this method implements the generation of a conservative (that is, larger than strictly necessary) region for replaying the output history.

The use of `clim-internals` prefixes in Algorithm 3 is not problematic, as the functions called could easily be replaced by portable code; however, the `tabcode-window-stream` class is at present a subclass of a McCLIM-internal class, `clim-internals::window-stream`. We have not investigated how much work it would be to remove the dependency on this internal class.

Algorithm 3 Customized `incremental-redisplay` for the tablature display.

```
(defmethod incremental-redisplay
    ((stream tabcode-window-stream) position
     erases moves draws erase-overlapping move-overlapping)
  (declare (ignore position))
  (let ((history (stream-output-history stream)))
    (with-output-recording-options (stream :record nil :draw t)
      (loop for (nil br) in erases
            do (clim-internals::erase-rectangle stream br))
      (loop for (nil old-bounding) in moves
            do (clim-internals::erase-rectangle stream old-bounding))
      (loop for (nil br) in erase-overlapping
            do (clim-internals::erase-rectangle stream br))
      (loop for (nil old-bounding) in move-overlapping
            do (clim-internals::erase-rectangle stream old-bounding)))
    (loop for (r) in moves do (replay r stream))
    (loop for (r) in draws do (replay r stream))
    (let ((res +nowhere+))
      (loop for (r) in erase-overlapping
            do (setf res (bounding-rectangle (region-union res r))))
      (loop for (r) in move-overlapping
            do (setf res (bounding-rectangle (region-union res r))))
      (replay history stream res))))
```