Cuepatlahto and Lascaux: Two approaches to the formalized control of musical score

Víctor Adán[1] **, Trevor Bača**

[1] Columbia University Music Department
621 Dodge Hall
2960 Broadway, MC 1813
New York, NY 10027

vga2102@columbia.edu, trevorbaca@gmail.com

Abstract. Cuepatlahto *and* Lascaux *are two systems for the formalization and transcription of musical score. Developed independently, both systems implement a set of modules, classes and procedures written in the* Python *programming language as higher level wrappers around the* LilyPond *open source system for automated music engraving. In this article we discuss the background and motivations to our work, introduce the components and workflow of both systems, and provide examples of both systems operating on numeric "seed" input together with the musical score produced as output.*

1. Introduction

Cuepatlahto and *Lascaux*, the work of the first and second authors, respectively, address several problems in music composition with which we have each been confronted. This list of shared concerns includes:

1. The difficulty involved in transcribing larger scale and highly parameterized gestures and textures into traditional western notation.
2. The general inflexibility of commercial[1] music notation software packages.
3. The relative inability of objects on the printed page in conventional score to point to each other — or, indeed, to other objects or ideas *outside* the printed page — in ways rich enough to help capture, model and develop long-range, nonlocal relationships throughout our scores.

In picking the phrase *formalized score control* to capture our solutions to these problems, we hope to highlight our want to embody important elements of musical score *symbolically* such that the full power of modern programming languages and tools in mathematics can be brought to bear on all parts of the compositional process.[2] We developed *Cuepatlahto* and *Lascaux* independently over the course of several years, having each first tried out and then discarded a number of different technologies. We have reached remarkably similar conclusions as to the good fit of dynamic and interpreted languages in general — and of *Python* in particular — for the formalization part of this problem, and of the unique status of *LilyPond*[3] as possibly the only notation package — open source, commercial or otherwise — able to offer full control of all parts of the musical score by means of a publicly documented clear text input format able to be driven by a higher level language such as *Python*.

In the sections that follow, the first author describes his work on *Cuepatlahto* and the second author describes his work on *Lascaux*; we offer shared conclusions at the end.

[1] Primarily *Finale* and *Sibelius*.

[2] The term is also an acknowledgment of the important role played by Xenakis and his music.

[3] See [Nienhuys and Nieuwenhuizen, 2003].

2. Cuepatlahto

Cuepatlahto[4] is the convergence of several tools and solutions to problems in music composition that the first author has encounter over the years. It is a set of modules for the formalization and transcription of musical score written in the *Python* programming language.[5]

2.1. Background and motivations

How do we represent a musical idea? How do we take a vividly experienced sound, its motions, densities, displacements, timbres, *etc.*, and notate it on paper within the confines of traditional music notation? In my very early exercises in music composition I found myself in conflict between the act of creating music and that of analyzing its possibilities for representation. This was particularly true for timing information, which I found to be particularly important. I had to stop to think about the relative timings while trying to produce them "freely". At the time I found analyzing sound recordings to be a good solution to my problem. Looking at the spectrogram and observing the position of energy bursts along the gridded timeline allowed me to make very precise measurements and thus very precise transcriptions. This orthogonalization between the process of creation and that of notation became an important part of my musical thinking.

Rotaciones (for unaccompanied viola) from 1999 was conceived both as a sequence and a set of operations to act on the sequence. The piece was a study on what Julio Estrada called *topological variations*; see [Estrada, 2002]. Estrada's variations were real transformations of solid wires representing multiparametric sequences. At the time, it became clear to me that the full potential of this kind of thinking could only come through its formalization. I thus began writing a collection of functions in the *C* programming language for the formal study and transformation of musical materials.

2.2. Components of the system

From a general point of view, *Cuepatlahto* can be described as a high-level signal processing system with a couple of input and output interfaces. I will first describe the core of the system and then look at its input and output interfaces. As it stands today, *Cuepatlahto* comprises four main packages: `Digitizer`, `Lily`, `MusicObjects` and `NDarray`. The first two are input and output interfaces, respectively. `MusicObjects` is the main data structure / representation package, and `NDarray` implements analysis and processing functions that operate on NumPy's numeric arrays.[6]

Cuepatlahto implements two types of data representation:

1. A numerical representation in the form of n-dimensional arrays using NumPy.
2. A hierarchical tree-like structure that parallels that of a musical score.

In the first form, the musical data is represented as an array of numbers. The "music" can be anything from a single sequence representing pitch, durations or any other kind of sampled data ...

$$\mathbf{p} = [p_2, p_2, p_3, \ldots]$$

... to a multidimensional sequence of parallel parameters such as the position of a body in 3-space.

$$\mathbf{SPACE} = \begin{bmatrix} \mathbf{x} & = & [x_1, x_2, x_3, \ldots] \\ \mathbf{y} & = & [y_1, y_2, y_3, \ldots] \\ \mathbf{z} & = & [z_1, z_2, z_3, \ldots] \end{bmatrix}$$

[4] *Cuepatlahto* means *interpreter* in Nahuatl.

[5] `http://www.python.org`

[6] `http://numpy.scipy.org`

In *Cuepatlahto*, each dimension of a multidimensional array models some musical parameter defined *a priori*, for example pitch, loudness and distortion, or p, l, d, respectively.

$$\mathbf{M} = \begin{bmatrix} \mathbf{p} & = & [p_1, p_2, p_3, \ldots] \\ \mathbf{l} & = & [l_1, l_2, l_3, \ldots] \\ \mathbf{d} & = & [d_1, d_2, d_3, \ldots] \end{bmatrix}$$

This type of representation facilitates the parallel processing of large amounts of data. Arrays can be operated on as matrices using common linear algebra operations. For example, to make pitch and loudness a combination of each other, say $\mathbf{p}' = 0.75\mathbf{p} + 0.25\mathbf{l}$, and $\mathbf{l}' = 0.25\mathbf{p} + 0.75\mathbf{l}$, we can simply multiply our music matrix $\mathbf{M}$ with a transformation matrix $\mathbf{T}$ ...

$$\mathbf{T} = \begin{bmatrix} 0.75 & 0.25 & 0 \\ 0.25 & 0.75 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

... by typing `T*M` at the *Python* command prompt. There are a couple of problems with this representation though. First, one is forced to use numbers only in order to operate on the arrays. This makes it difficult to encode any kind of information that might better be represented as a *string*. A second problem is that all the dimensions in the array must have the same number of elements. If one dimension must grow or shrink, so must the others.

The second representation method is a hierarchy of nested musical objects akin to the structure of traditional music score. The music classes composing the hierarchy are: `Score`, `Section`, `Instrument`, `Staff`, `Voice`, `Tuplet`, `Note` and `Chord`. The baseclass for all these classes is the `MusicObject` class. All the other classes inherit from it. The `Note` and `Chord` classes are the only two possible ends (i.e. leaves) of the hierarchy and they inherit directly from `MusicObject`. The rest of the classes are

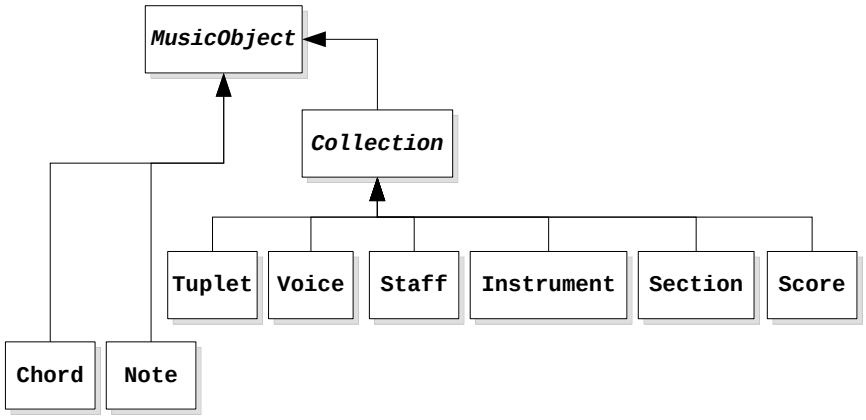


Figure 1: Class hierarcy of `MusicObject` structure.

`Collections` because each has a list containing other musical objects. `Collection` in turn inherits from `MusicObject`.

In contrast to the numeric arrays, each of these classes has a variety of attributes of different types. The richest class is the `Note` class, which has four mandatory attributes:

- duration (type *float*)
- value (*e.g.* pitch) (type *float*)
- continuous type (type *int*)
- head type (type *string*)

Optional attributes include articulation marks and any number of other tags, both of type *string*.

Here's an example of how a complete three-note score would be created in *Cuepatlahto*:

```
n1 = Note(1/2., 0, cont=0, head='diamond')
n2 = Note(1/4., 2, cont=1)
n3 = Note(1/4., 3, cont=0)
v = Voice([n1,n2,n3])
st = Staff([v], 'pitch')
i = Instrument([st],'violin')
s = Section([i])
score = Score([s], 'Title')
```

This type of data representation has some advantages over the first. It is easy to visualize and organize the musical data. Strings and numbers can live together, allowing for detailed descriptions. Also, in contrast to numeric array representation, each of these objects can have a different number of attributes and this number can grow without affecting the other nodes of the music tree.

2.3. Working with the system

Numbers to be processed in *Cuepatlahto* can be entered by hand using the computer keyboard. In addition, source data can be extracted from audio recordings through parameter trackers or read from a digitizing tablet in much the same way as with the UPIC machine; see [Henning, 1986].

Parameter trackers provide the data in its rawest form as numeric arrays of one or more dimensions. Currently, *Cuepatlahto* implements dedicated procedures to handle amplitude envelope extraction, onset detection and the measurement of spectral entropy. The graphic digitizer interface allows for a richer kind of input. Each point digitized can have as many attributes as the user requires and be of various data types. However, the interface is typically used much like the UPIC system, with time represented on the horizontal axis and some time-dependent parameter on the vertical. Used in this fashion, each point given as input has the following attributes:

1. Vertical position
2. Duration (the horizontal distance between two adjacent points)
3. Point type: a tag that associates a point to a class, *e.g.* round, triangle, square, diamond
4. Continuous type: a number defining whether the point is an inflection in a continuous function or not, and what kind (*e.g.* linear, spline, *etc.*)
5. Textual tag: a textual description of additional attributes of the point; defaults to empty

The digitizer interface can return either a numeric array or a `Voice` object, in which case one can immediately render the data to a musical score via the `lily` translator package.

By default, `MusicObjects` display as lists of $attribute : value$ pairs. For example, running `print` on the tiny score given in Section 2.2, yields:

```
Score: Title
 Section: section
  Instrument: violin
   Staff: pitch
    Voice: pitch num: 2
      arts:[] cont:0 dt:0.5  head:diamond type:normal val:0
      arts:[] cont:1 dt:0.25 head:round   type:normal val:2
      arts:[] cont:0 dt:0.25 head:round   type:normal val:3
```

However, one of the main purposes of *Cuepatlahto* is the generation of musical scores. Thus, the `lily` module provides the functionality to generate *LilyPond* input files from

the `MusicObject` data representation. Given a score `score`, a *LilyPond* file can be written to disk by calling the `write()` function from the `lily` module.

2.4. Application example

The following is a short example of a work session. There are four basic steps in the workflow of a session:

1. Input seed(s) or source material
2. Generalize and / or process seed to generate variations or to develop material
3. Create a piece by scaling and fitting instances and arranging them together
4. Generate output score

Let us define the seed $\mathbf{s} = [1.0, 1.0, 0.5, 0.5, 1.0, 1.0, 0.5, 0.5, 1.0, 0.5, 0.5]$. From this seed we will derive a collection of derivations $\mathbf{s}'_i$ by filtering $\mathbf{s}$ so: $\mathbf{s}'_i[n] = a_i\mathbf{s}[n-1] + b_i\mathbf{s}[n]$. In this particular example, we will use the trigonometric functions $\sin\theta$ and $\cos\theta$ as the a and b coefficients. Specifically, define the set of transformation vectors $\{\mathbf{r}_1, \mathbf{r}_2, \mathbf{r}_3, \ldots, \mathbf{r}_8\}$, such that $\mathbf{r}_i = [\cos\theta_i, \sin\theta_i]$ and $\theta_i = i2\pi/8$. To facilitate transforming the data, let us then create a two-dimensional array $\mathbf{S}$ of the consecutive pairs of numbers in $\mathbf{s}$.

$$\mathbf{S} = \begin{bmatrix} 1.0, & 1.0, & 0.5, & 0.5, & 1.0, & 1.0, & 0.5, & 0.5, & 1.0, & 0.5, & 0.5 \\ 1.0, & 0.5, & 0.5, & 1.0, & 1.0, & 0.5, & 0.5, & 1.0, & 0.5, & 0.5, & 1.0 \end{bmatrix}$$

This allows us to then obtain our variations by simply multiplying the matrix $\mathbf{S}$ by a transformation vector, so: $\mathbf{s}'_i = \mathbf{r}_i\mathbf{S}$. Applying this operation for each of the vectors $\mathbf{r}_i$ we obtain the following results:

$$\begin{aligned}
\mathbf{s}'_1 &= [1., 1., 0.5, 0.5, 1., 1., 0.5, 0.5, 1., 0.5, 0.5] \\
\mathbf{s}'_2 &= [0.727, 1.080, 0.727, 0.373, 0.727, 1.080, 0.727, 0.373, 1.080, 0.727, 0.373] \\
\mathbf{s}'_3 &= [0.454, 0.954, 0.954, 0.454, 0.454, 0.954, 0.954, 0.454, 0.954, 0.954, 0.454] \\
\mathbf{s}'_4 &= [0.341, 0.695, 1.048, 0.695, 0.341, 0.695, 1.048, 0.695, 0.695, 1.048, 0.695] \\
\mathbf{s}'_5 &= [0.454, 0.454, 0.954, 0.954, 0.454, 0.454, 0.954, 0.954, 0.454, 0.954, 0.954] \\
\mathbf{s}'_6 &= [0.727, 0.373, 0.727, 1.080, 0.727, 0.373, 0.727, 1.080, 0.373, 0.727, 1.080] \\
\mathbf{s}'_7 &= [1., 0.5, 0.5, 1., 1., 0.5, 0.5, 1., 0.5, 0.5, 1.] \\
\mathbf{s}'_8 &= [1.112, 0.759, 0.405, 0.759, 1.112, 0.759, 0.405, 0.759, 0.759, 0.405, 0.759]
\end{aligned}$$

Here is the *Python* code for this transformation:

```
s   = [1, 1, .5, .5, 1, 1, .5, .5, 1, .5, .5, 1]
s   = numpy.array(s)
S   = ss.ssrec(s, 1, 2)
S_i = ss.all2drot(S, 8)
s_i = S_i[:,:,0]
```

`numpy.array()` is an array creation function from the NumPy library. The `ss.ssrec(data, delay, dim)` function in the `NDarray` package creates an $(n \cdot dim)$-dimensional array from the n-dimensional data array given, and is used here to create the 2D pattern array. The `dim` parameter defines the number of dimensions in the output and `delay` is the displacement for each of the additional dimensions. `ss.all2drot(data, n)` is a function that returns a list of all $2\pi/n$ equidistant rotations of 2D data. Finally `s_i = S_i[:,:,0]` recovers only the first dimension of each of the 2D data sequences. The `s_i` variable is thus a list of our eight variation sequences. Now that we have our derived patterns $\mathbf{s}'_i$ we will make some music out of them. Let us use these sequences as rhythmic patterns and assign each to a different instrument. To do this we will create eight `Voices`, one per pattern, with each pattern term assigned a separate `Note`. In *Python* we can create both the `Notes` and the `Voices` in a single line like so:

```
voicelist = [Voice([Note(dt) for dt in pat]) for pat in s_i]
```

This will return a list of `Voices` (`voicelist`) that we can then assign to `Staffs`, `Instruments` and finally to a `Score`. Before creating the full score, however, we must process the `Voices` to conform to the limitations of standard western music notation. Observe that all note duration values in the western notation system must be of the form

$$\frac{1}{2^{(n)}} + \frac{1}{2^{(n+1)}} + \ldots + \frac{1}{2^{(n+m)}}, \text{ for } n, m \in \mathbb{Z}.^{7}$$

Most of the values in the sequences derived do not have this form. Thus, we must convert them before attempting to call the score renderer on them. First we quantize all the values using `Voice.quantizeTime(128)`. This function will make the smallest possible duration in our score a 128*th* note. Then we call `Voice.pow2dt()` to convert all durations to the form described above. If after quantization, `Voice.pow2dt()` finds that a `Note`'s duration does not conform, then the `Note` is split and new tied `Notes` are generated and inserted into the `Voice` calling this method.

In addition to these two preparatory methods, we will also call `Voice.scaleTime()` to scale the durations down by $1/8$ and make all durations less than or equal to an eighth note. The *Python* code to perform these three tasks is this:

```
for voice in voicelist:
    voice.scaleTime(1/8.)
    voice.quantizeTime(128.)
    voice.pow2dt()
```

Once the durations have been adjusted for score output we create a `Section` containing eight `Instruments`, each containing one `Staff`, and each `Staff` containing one `Voice`. Again we do this in *Python* in a single line using list comprehensions.

```
sec = Section([Instrument([Staff([voicelist[i]],name='0')],
            name=str(i)) for i in range(len(voicelist))])}
```

Finally we create a `Score` with this single section and pass it to the `write(score)` method in the `lily` module for rendering as a *LilyPond* score.

```
score = Score([sec], name='rhythm')
lily.write(score)
```

Figure 2 shows the rendered score. Note that each `Instrument` is rendered on a single-line staff. This is achieved by assigning the value `'1'` to the `name` parameter of each newly created `Staff`.

3. Lascaux

Lascaux[8] extends the *Python* programming language to an interactive composers' workbench. The project results from the second author's work formalizing different parameters of the musical score. In this section, the second author provides the background and motivation to *Lascaux* and introduces the components and basic workflow of the system; the section concludes with a working example.

[7] Note that this does not refer to the actual (absolute or relative) durations of notes, which may be altered through the use of tuplets or tempo changes in general. Rather, it refers to the duration traditionally represented with the graphical object that combines a notehead, zero or more dots, and zero or more flags: 𝅗𝅥.

[8] "It is in this way, under the pretext of saving the original, that the caves of Lascaux have been forbidden to visitors and an exact replica constructed 500 metres away, so that everyone can see them (you glance through a peephole at the real grotto and then visit the reconstituted whole). It is possible that the very memory of the original caves will fade in the mind of future generations, but from now on there is no longer any difference ..."; [Baudrillard, 1983].

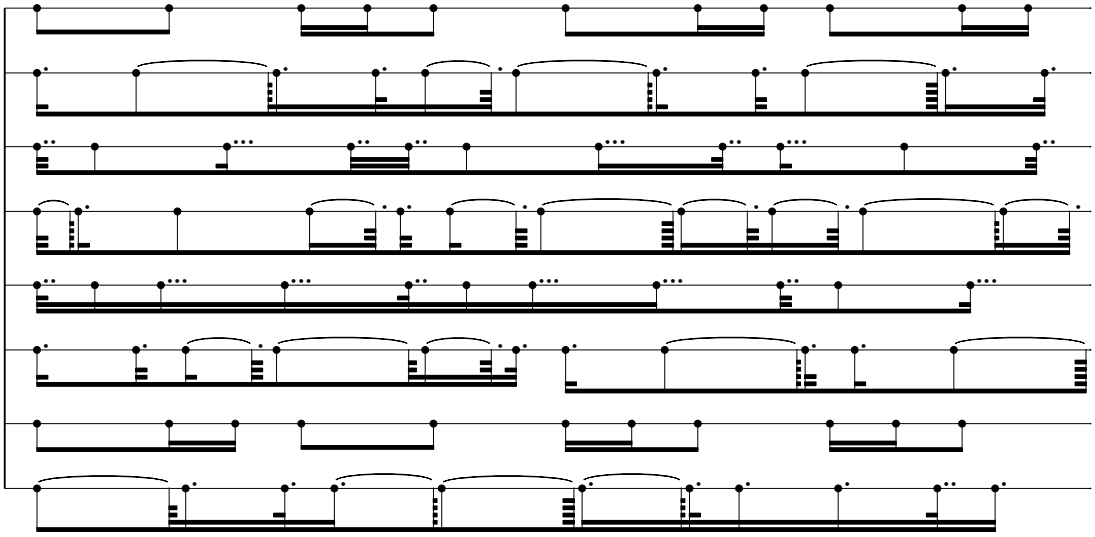


Figure 2: Score notation output from *Cuepatlahto* for the eight rhythmic patterns generated above; the first is the original seed pattern.

3.1. Background and motivation

During the composition of *ZEIT (14 players)* in 1997 – 98 I became interested in two different types of transform, the first[9] designed to partition and then registrally permute aggregate chords, and the second to collect the results of a series of successive order operations against some starting set of pitches or durations. I dubbed this second transform *helianthation* after the fanciful image of a bright collection of sunflowers turning to follow the course of the sun across the sky. The transform formalizes as follows.

Let $S = S_1, ..., S_n$ denote some *seed* S equal an ordered collection of sublists $S_1, ..., S_n$; and let each of the sublists $S_i \in S$ equal a possibly different number of elements $S_i = S_{i,1}, ..., S_{i,k}$, with each of the $S_{i,j} \in \mathbb{Z}$. Denote by $\mathfrak{h}_r^r(S)$ the order operation $\mathfrak{h}$ on seed S that first rotates each of the $S_i \in S$ one position to the right and then rotates each of the $S_{i,j}$ in each of the S_i one position to the right. And then denote by $\mathfrak{H}_r^r(S)$ the composite order operation $\mathfrak{H}_r^r(S) = S, \mathfrak{h}_r^r(S), \mathfrak{h}_r^r(\mathfrak{h}_r^r(S)), ...$ which nests and collects the unique applications of $\mathfrak{h}$ against S. Further, denote by ${}_f\mathfrak{H}_r^r(S)$ the flattened form of $\mathfrak{H}_r^r(S)$.

As an example, we take initial seed $S = [[4, 6], [2, 4, 6], [2, 8]]$ and see that the first application of $\mathfrak{h}$ against S gives $\mathfrak{h}_r^r(S) = [[8, 2], [6, 4], [6, 2, 4]]$, that the second application of $\mathfrak{h}$ gives $\mathfrak{h}_r^r(\mathfrak{h}_r^r(S)) = [[4, 6, 2], [2, 8], [4, 6]]$, and so on; after six such applications of $\mathfrak{h}$ against S we get back the initial seed S as a result. These successive applications of $\mathfrak{h}$ then give $\mathfrak{H}_r^r(S) = [[4, 6], [2, 4, 6], [2, 8]], [[8, 2], [6, 4], [6, 2, 4]], [[4, 6, 2], [2, 8], [4, 6]], ...]$; flattening $\mathfrak{H}_r^r(S)$ we get ${}_f\mathfrak{H}_r^r(S) = [4, 6, 2, 4, 6, 2, 8, 8, 2, 6, 4, 6, 2, 4, 4, 6, 2, 2, 8, 4, 6, 6, 4, 2, 4, 6, 8, 2, 2, 8, 4, 6, 6, 2, 4, 4, 6, 2, 8, 2, 6, 4]$.

I found the effect of this particular transform to be something like the constant recurrence of fixed elements shimmering in and out of relative order. The feasibility of working out fully helianthated examples by hand depends[10] crucially on the relative lengths of the sublists of S. So my strategy during *ZEIT* was to work out the different helianthations of both the pitch and rhythmic material with a one-off program written

[9] Later labelled *constellation*.

[10] In general, $\texttt{len}(\mathfrak{h}_r^r(S)) = LCM(\texttt{len}(S_1), ..., \texttt{len}(S_n), \texttt{len}(S))$ and $\texttt{len}({}_f\mathfrak{H}_r^r(S)) = \texttt{len}(\mathfrak{h}_r^r(S)) \times \sum_i \texttt{len}(S_i)$ for $S_i \in S$, at least where the $S_{i,j}$ are unique in S; symmetries between the $S_i \in S$ reduce the length of ${}_f\mathfrak{H}_r^r(S)$.

in *C*, designed to assist both score formalization and transcription. The code formalized ${}_f\mathfrak{H}^r_r(S)$ easily but gave mixed results with regards to transcription. The chosen transcription strategy was to write type 1 MIDI files to disk for import to *Finale*; see [The International MIDI Association, 1988]. The pitch material transcribed successfully while anything beyond the most basic rhythms imported into *Finale* only poorly.[11]

At the conclusion of *ZEIT*, I was struck by the compositional usefulness of ${}_f\mathfrak{H}^r_r(S)$ and related transforms, the relative ease of pitch transcription, the difficulty of rhythmic transcription, the poverty of MIDI as a medium for encoded notation generally, and the need for a different type of system to model higher level musical constructs. These conclusions were the motivators for my work on what later become *Lascaux*.[12]

3.2. Components of the system

Lascaux currently comprises 22 modules, 16 core classes, and open set of discrete transforms. Of these 16 classes, seven are *concrete typesetting classes* equipped with methods to generate exact *LilyPond* input for notes, rests, skips, tuplets, voices, staves and complete scores. These seven concrete typesetting classes are available to the user at runtime and inherit from a further five *abstract*[13] *typesetting classes*. The abstract typesetting classes are shielded from the user completely and not available at runtime. These 12 typesetting classes together participate in a *typesetting class hierarchy*, from which four remaining *utility*[14] *classes* stand apart.

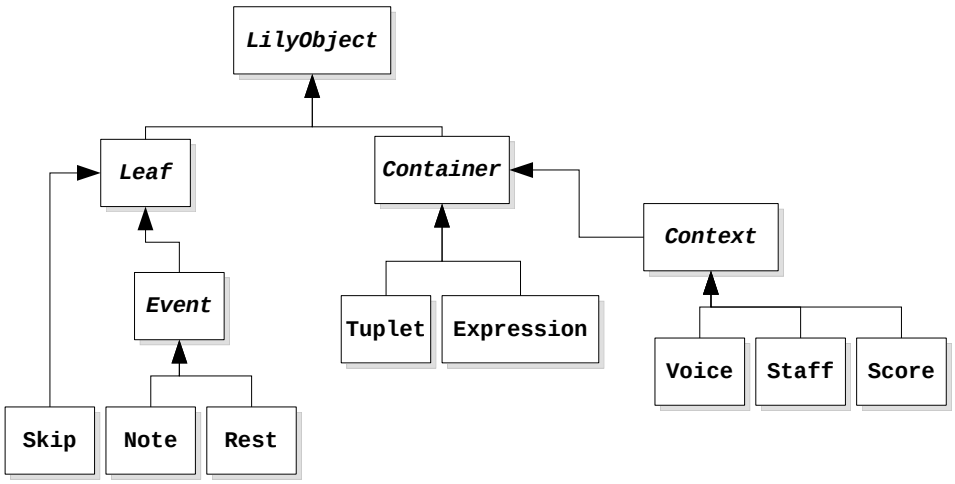


Figure 3: *Lascaux* typesetting hierarchy; arrows show inheritance.

3.3. Working with the system

Lascaux reads input from file and also accepts commands typed directly at the interactive interpreter. To work with the system interactively, type `lascaux` at the prompt.

[11] MIDI files model rhythm as a series of time deltas between consecutive events only, which the import filters to most of the commercial notation programs quantize to usually only binary divisions of some unit time. This makes nonbinary tuplet divisions problematic and makes nested tuplet divisions all but impossible. Beams, ties and the other aspects of rhythmic notation not directly associated with time deltas do not encode into MIDI files at all; see [Hewlett et al., 1997].

[12] Almost eight years separate this work on *ZEIT* from the implementation of *Lascaux* described in the next section. The time between first saw the replacement of *C* with *Mathematica* and then a move away from MIDI files in favor of the clear text `.pmx` input files to Leland Smith's SCORE. The first two verses of *POÈME RÉCURSIF (64 pieces of percussion)* in 2003 / 2005 were the product of that intervening implementation of the system.

[13] *Abstract* here means only that certain classes are, by convention, not meant for user instantiation and do not load into the global namespace on start-up. *Python* has no `abstract` modifier.

[14] `Rational`, `Duration`, `Pitch` and `Accidental`.

```
$ lascaux
LASCAUX 1.2 ...
Python 2.5 (r25:51918, Sep 19 2006, 08:49:13)
[GCC 4.0.1 (Apple Computer, Inc. build 5341)] on darwin
Type "help", "copyright" ... for more information.
>>>
```

Lascaux writes a greeting to screen and then invokes the *Python* interpreter. The global namespace contains *Lascaux* modules available to the user.

```
>>> dir()
['__builtins__', '__doc__', '__file__', '__name__',
'duration', 'expression', 'lily', 'music', 'notate',
'note', 'pitch', 'rest', 'score', 'skeleton', 'skip',
'staff', 'templates', 'transforms', 'tuplet', 'voice']
```

Concrete typesetting classes can be initialized one at a time by hand; here we specify a note with pitch 13 and duration equal to $3/16$ of a whole note; *Lascaux* pitches follow the convention in [Morris, 1987] with ..., $B\flat 3, C\flat 4, C\sharp 4$, ... equal to ..., $-1, 0, 1$, ...

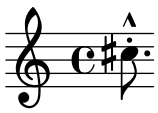
```
>>> n = note.Note(13, 3, 16,
                  right = [r'\marcato', r'\staccato'])
```

All classes override *Python*'s default `__repr__` method and publish approximate *LilyPond* input to the interpreter; note that *LilyPond* pitches follow the convention that pitches ..., $B\flat 3, C\flat 4, C\sharp 4$, ... equal to ..., `b`, `c'`, `cs'`, ...

```
>>> n
cs''8.
```

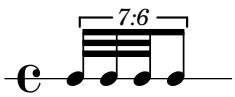
Exact *LilyPond* input is available at any time via the `lily` property string; all concrete typesetting classes implement this string. *Lascaux* makes available a large number of given and derived attributes regarding the duration of notes, rests, tuplets and other durated objects in the score, while making as few assumptions as possible about the graphic appearance of these objects, offering `left`, `right`, `before` and `after` attribute lists for the lexical positioning of raw *LilyPond* formatting directives against arbitrary system objects instead.

```
>>> print(n.lily)
cs''8. \marcato \staccato
>>> show(n)
```



The `show()` command writes *LilyPond* input for a complete score to file, calls *LilyPond*, and pushes the resulting `.pdf` to screen. Higher level procedures in the `music` module are available to create more objects at once; here we apply the ratio $1 : 1 : 1 : 4$ against a duration equal to $6/32$ of a whole note.

```
>>> t = music.divide([1, 1, 1, 4], 6, 32)
>>> t
(7:6, c'32, c'32, c'32, c'8)
>>> show(t)
```



Interactive sessions in *Lascaux* iterate through a straightforward two-part cycle — create some objects, `show()` the result, change the objects, `show()` the result again, *etc.*, building up ever larger collections of objects with ever more attributes and interrelations. *Lascaux* saves the `.pdf` of each bit of notation to disk, which makes it possible to review all intermediate products leading up to a complex expression, over the course of days or weeks of experimentation. Complete expressions usually then save to file for reimport to the interpreter later.

3.4. Application example

In the example below, from *LIDERCFENY (fl, vln, pf)* in 2007, we build up a rhythmic structure with seven strata, beginning with the *Lascaux* implementation of ${}_f\mathfrak{H}^r_r(S)$, defined above.

```
t     = helianthate([[4, 6], [2, 4, 6], [2, 8]])
s     = helianthate([[0, -1], [0, 0, -1], [0, 0, 1]])
```

`t` and `s` are now 42- and 48-element lists resulting from the helianthation of two different starting seeds. `increase(t, s)` cyclically adds elements in `s` to elements in `t`.

```
>>> t
[4, 6, 2, 4, 6, 2, 8, 8, 2, 6, 4, 6, 2, 4, 4, 6, 2, 2,
 8, 4, 6, 6, 4, 2, 4, 6, 8, 2, 2, 8, 4, 6, 6, 2, 4, 4,
 6, 2, 8, 2, 6, 4]
>>> s
[0, -1, 0, 0, -1, 0, 0, 1, 1, 0, 0, -1, 0, -1, 0, 0, 0,
 -1, 0, 0, 1, 0, 0, -1, -1, 0, 0, 0, -1, 0, 0, 1, 1, 0,
 0, 0, -1, -1, 0, 0, 0, -1, 0, 0, 1, 0, -1, 0]
>>> increase(t, s)
>>> t
[4, 5, 2, 4, 5, 2, 8, 9, 3, 6, 4, 5, 2, 3, 4, 6, 2, 1,
 8, 4, 7, 6, 4, 1, 3, 6, 8, 2, 1, 8, 4, 7, 7, 2, 4, 4,
 5, 1, 8, 2, 6, 3]
```

The elements in `t` now give the top-level durations of 42 consecutive measures in integer numbers of sixteenth notes — $2/8, 5/16, 1/8, ...$. The next series of assignments subdivide the top-level measure durations in `t` and determine the durations of spanning beam patterns.[15]

```
cut   = helianthate([[4, 8, 8], [8, 8], [4, 8, 8]])
j     = untie(intaglio(t, cut, t = 4))
g     = flatten(j)
h     = [2 * x for x in g]
beam  = [len(x) for x in resegment(j, [4, 8, 18], max = 3)]

cut   = helianthate([[0], [0, 0, 1]])
m1    = [divide(d, weight(d), 16) for d in plough(j, cut)]
```

We assign to `m1` the first of our seven rhythmic strata. Subsequent transformations and

[15]The functions `untie()`, `intaglio()`, `resegment()`, `plough()` and `stellate()` used through the body of this example are discrete and combinatorial transforms on integers, lists, or lists of lists, formalized much the way as the definition of ${}_f\mathfrak{H}^r_r(S)$ with which our description of *Lascaux* opened. Although a full definition of these and the related transforms in the system is beyond the scope of this paper, we summarize here that `untie()` transforms integers such as $5, 9, 10, ...$ to lists of integers such as $[4, 1], [8, 1], [8, 2], ...$, here used to strip ties from certain parts of the score. `intaglio()`, `resegment()` and `plough()` transform two-dimensional lists in different ways, according to the structural attributes of other two-dimensional lists, here used derive measure division, beaming and tupletting information from related but conflicting source material. `stellate()` accepts a number of different one- and two-dimensional lists as input and returns nested collections of tuplets as a result. This way of piling discrete transforms one on top of another came after a period of working to map the extensive set of physical proportions in a complex natural object — the magnified wing of a dragonfly — to rhythmic proportions in the score; the quantization involved in this work largely followed [Nauert, 1994].

assignments work out the remaining six rhythmic strata; each of the tightly related input seeds worked out here derive from many dozens of iterations at the *Lascaux* interpreter.

```
cut   = [[2, 3, -4], [2, -4], [2, 3, -3, 4]]
m2    = stellate(g, [[0]], cut, 16, beam)
stack = [[2, 4, 5, 5], [2, 4], [0, 2, 3]]

cut   = [[4, 5, 6, -7], [5, -7], [6, 7, 8, -8]]
m3    = stellate(g, stack, cut, 16, beam)

cut   = [[4, 6, 8], [4, -8], [4, 6, -6, 8]]
m4    = stellate(h, [[0]], cut, 32, beam)

stack = [[0, 2, 3], [2, 4], [2, 4, 5, 5]]
cut   = [[2, -2, 3, -4], [3, -4], [3, -4, 4, -6]]
m5    = stellate(h, stack, cut, 32, beam)

stack = [[2, 4, 5, 5], [2, 4], [0, 2, 3]]
cut   = [[4, 5, 6, -6], [5, -6], [6, 6, 8, -8]]
m6    = stellate(h, stack, cut, 32, beam)

stack = [[2, 4, 5], [2, 4, 4, 5], [0, 2, 3]]
cut   = [[2, 3, 4, -4], [2, -3], [4, 4, 6, -6]]
m7    = stellate(h, stack, cut, 32, beam)
```

Rendering the rhythmic strata `m1`, `m2`, `m3`, `m4`, `m5`, `m6`, `m7` as separate voices gives a 42-measure score in seven staves. The complete score comprises some 2055 notes and rests. In figure 4 we excerpt measures 36 – 38, and the first part of measure 39.

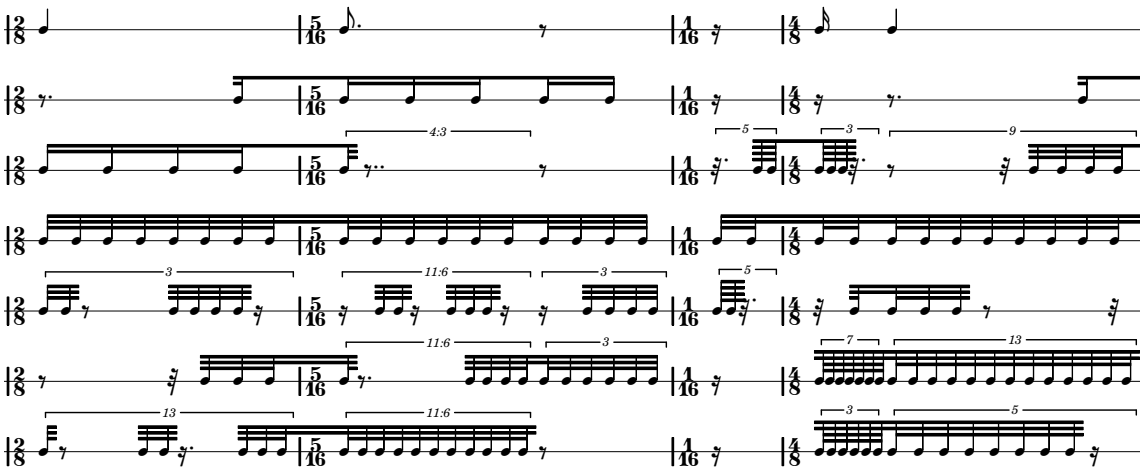


Figure 4: Output from *Lascaux*, excerpted from the 2055-note score encoded above.

4. Conclusions and future work

In the preceding sections we have presented our work on *Cuepatlahto* and *Lascaux*, systems we developed independently and continue to develop to formalize control of the musical score and help in the transcription of complex and composite musical material. Both systems implement higher level wrappers written in *Python* to drive the *LilyPond* open source system for automated music engraving; both systems implement a core class hierarchy together with modules and procedures for specialized work. Each system also capitalizes on different design strengths.

Cuepatlahto admits numeric input both by hand and from file as well as from the output of a digitizing tablet after the fashion of the UPIC machine of Xenakis; this specialized graphic input integrates *Cuepatlahto* tightly with the first author's work in geometric input and transforms as a driver of the compositional process. To translate graphic and geometric elements outside the score to concrete elements of notation within the score, *Cuepatlahto* implements strong support for the quantization of floating point or real numbers and the transformation of those values through the matrix operations of linear algebra.

Lascaux rules out quantization and real-valued modeling in favor of discrete and combinatorial transforms over the integers. This way of working follows the second author's rejection of overt mapping of visual or geometric information from the external world into the score in favor of the iterative and layered construction of complexes of information designed to rival our experience of the visual world.

Though neither system has a conventional graphic user interface, it seems unlikely that we will develop one for either system. We leave open the possible unification of both systems and also the eventual publication of one or both systems on the public Internet.

References

Baudrillard, J. (1983). The precession of simulacra. In *Simulations*, page 18. Semiotext[e], New York, NY.

Estrada, J. (2002). Focusing on freedom and movement in music: Methods of transcription inside a continuum of rhythm and sound. *Perspectives of New Music*, 40(1):70–91.

Henning, L. (1986). Xenakis and the UPIC. *Computer Music Journal*, 10(4):42–47.

Hewlett, W. B., Selfridge-Field, E., Cooper, D., Field, B. A., Kia-Chuan, N., and Sitter, P. (1997). MIDI. In Selfridge-Field, E., editor, *Beyond MIDI: The Handbook of Musical Codes*, pages 48–50. The MIT Press, Cambridge, Massachusetts.

Morris, R. D. (1987). *Composition with Pitch-Classes: A Theory of Compositional Design*, chapter 1, page 5. Yale University Press.

Nauert, P. (1994). A theory of complexity to constrain the approximation of arbitrary sequences of timepoints. *Perspectives of New Music*, 32(2):226–263.

Nienhuys, H.-W. and Nieuwenhuizen, J. (2003). Lilypond, a system for automated music engraving. In *Proceedings of the XIV Colloquium on Musical Informatics*.

The International MIDI Association (1988). Standard MIDI files 1.0. 5316 W. 57th St., Los Angeles, CA 90056.