A Musical Score for Csound with Abjad

Gianni Della Vittoria,

Liceo Musicale “Canova” di Forlì

giannidellavittoria.audio@gmail.com

Abstract. This paper presents the advantages of using a traditional musical score to make music with Csound. After illustrating some alternative approaches, examines Abjad, a Python library for printing music through Lilypond, and explains the technique for linking Abjad to Csound. Since in order to compose a musical score of synthesizers it is necessary to manage complex envelopes, particular attention is paid to how to represent the envelope profiles of the various parameters on a score and how to interpret them in Csound. As the system is open to several possibilities, the choice is proposed that seems simpler and allows the user to see the envelope as a musical element to be set on a staff, providing a better overview of the whole composition.

Keywords: Score, Abjad, Python, Algorithmic composition, Lilypond

1 Introduction

When writing music in Csound, the issue of realizing the score, as it is well known, can be solved in a vastness of ways. There is a variety of approaches ranging from direct compilation of the score to the use of third-party software. Each of them has its own peculiarities that can best fit the sometimes opposite preferences of those preparing to compose music with Csound.

One of the necessities that arise during the creation of an average complex score is to organize musical events at multiple levels, and not as a simple succession of instances. Today this is certainly possible with the sole use of Csound, without having to resort to external tools. One typical arrangement, in this sense, is to write the score to instantiate instruments which in turn call other instruments according to a specific algorithmic plan. The score, thus, hosts not so much the "notes", but the musical "phrase". By extending the principle, the orchestra can organize calls at multiple levels, leaving the score the task to coordinate the highest level.

There are also a whole series of external programs that deal with the problem of managing music organization levels in very interesting ways, such as Blue by Steven Yi, a rich environment where it is possible to view series of events arranged on a temporal plan with the chance of nested levels, or athenaCL by Christopher Ariza, which instead uses a command line approach to determine series of events according to well-defined parameter masks with many categories of envelopes.

It is useful to be able to represent the complexity of musical thought in some way, so as to be able to observe its unraveling over time. In this article I am going to present an-

other way to organize and visualize the Csound score, that is, through a traditional musical score adapted to Csound's needs.

2 Preparing a Musical Score for Csound

Instantiating Csound with a musical score has been done in various ways. One of them is via MIDI: you create a score with score editing software like MuseScore, extract the MIDI version and import it into Csound. This mode has various appreciable aspects, such as the ease with which Csound can be played by connecting each instrument of the musical score to the desired sound. However, it is not so easy to integrate a mechanism for transferring the parameter envelopes. How to describe them in a musical score? And solved this, how to be able to transfer them in real time to Csound, given that Csound should know in advance where the envelope will end up for at least the duration of the entire note?

Another way is by using visual programming software such as Open Music or PWGL. Open Music, for example, has several libraries specifically built to communicate with Csound, but, alongside the wealth of algorithms for processing various musical parameters, there are constraints such as the difficulty of managing dynamic markings or other musical symbols.

2.1 Lilypond and Abjad

To get a traditional musical score full of details, Lilypond is certainly an excellent choice. This software requires ASCII text notation, which is then compiled into documents such as PDF, PostScript, SVG of acknowledged quality. While remaining faithful to its textual nature, Lilypond has seen the contribution of various external GUI programs to facilitate the introduction of musical content (Frescobaldi, Denemo, Canorus, ...). Furthermore, there are many programming languages that use Lilypond to visualize algorithmic music. Among them Abjad stands out, an extensive Python library that allows the user to work with various Lilypond elements implemented in a sophisticated class structure. The advantage of Abjad lies in the fact that the algorithmic composition becomes simpler than if you were to create a pure Lilypond text file, being practically every element manageable with simple class instances.

Given the richness and the ability to produce very complex graphics, Abjad and Lilypond are particularly suitable for the composition of contemporary music; hence the idea of using these tools in conjunction with Csound.

2.2 From Abjad to Csound

Various ways can be used to connect Abjad to Csound. Here the python ctcsound module is taken into account: it allows you to compile csound through a variable containing the .orc and .sco text.

To show a short example, let's consider a simple orchestra with this beginning of Csound score

```
  sco_text = ['''
  f1 0 8192 10 1 .1 .01 .02 .03 0 0 .01 0 .02 .01 0 .02
  ;      amp midi attack decay pan
  i1 0 1 0.5 60   0.01   0.1   0.5
  ''']
```

After creating the musical score in Abjad, in which each instr has a dedicated staff like in a traditional orchestral score, it is necessary to extract the onset, duration and pitch information relying on the abjad parser, so that it selects the events iteratively, avoiding rests. The iteration must take place over logical_ties, so as to consider tied notes as individual units.

```
for logical_tie in
abj.iterate(score).logical_ties(pitched=True):
  offset = abj.inspect(logical_tie).timespan().start_offset
  offset_seconds = 60*offset/(metronome_mark.reference_du-
ration * metronome_mark.units_per_minute)
  dur = abj.inspect(logical_tie).duration()
  dur_seconds = 60 * dur /
(metronome_mark.reference_duration *
metronome_mark.units_per_minute)
scoLine = ['\ni1 ', str(offset_seconds.__float__()),
str(dur_seconds.__float__()), '.5']
scoLine.append(str(60 + abj.NumberedPitch(logical_tie[0])))
```

The onset times (p2) are taken from the Abjad inspector through the timespan().start_offset method, which returns the value in musical figures of duration. The following line makes the conversion in seconds, taking into account the metronome. The same procedure is applied to the duration, while the frequency is drawn from from the NumberedPitch class of Abjad, which is 0 for middle C, 1 for C#, 2 for D and so on. By adding 60 they can be easily intercepted by Csound through the cpsmidinn opcode.

Finally, the other p-fields are added, which obviously could be freely processed in python or derived from the abjad score. After calling csound through ctcsound, what you get is real-time listening and visualization of the score.

2.3 Microtonal tuning

The representation of micro-intonation on a musical score always raises questions that force us to take sides. The choice of the best notation system depends on the compositional needs and can be quite different from one piece to another.

Fortunately, the flexibility of Abjad provides the freedom of choice you prefer. A fairly general approach could be the quarter-tone notation, which is easily readable, with deviations expressed in cents by a small number before the note. This number would therefore range from +25 to -25 and can be float for fractions of a cent.

Once paired with the Note class, all the python code has to do is parse and convert this value to a decimal midi note: Csound will take care of the rest (the cpsmidinn opcode is able to correctly evaluate decimals).

3 Graphical representation of envelopes

Just as in an instrumental musical score it is important to show every detail relating to the execution of each instrument, to put together a score of synthesizers it is useful to define the dynamic developments of the various parameters by displaying the envelopes of the most significant parameters. Defining them only in Csound would require a constant reuse of envelopes with the same number of p-fields, and in any case the fact remains that they cannot be displayed. Not even the Open Music maquette comes to the punctual clarity that only a musical score can allow. Defining the envelopes in Abjad, on the other hand, allows them to be diversified for each event and to show them all in a clear overview.

Also for this there can be various approaches. For example, we could use traditional dynamic markings, such as pp, mf, crescendo, associating them with certain amplitude patterns. Alongside the undoubted advantage of easy readability, however, there is the downside of a reduced number of nuances, compared to the possibilities of a synthesizer. However it would not be bad to employ them in conjunction with other systems.

Another way would be to imitate the envelope profiles of the most popular synthesizer graphics, technically feasible thanks to the powerful PostScript language that Lilypond introduced. Here, however, the problem would consist in deciding how to get to the Csound transcription, which can be solved in various ways, but perhaps a bit cumbersome. For example, PostScript uses elegant cubic Bezier curves that could be converted to Csound, but Csound for now only has the quadratic Beziers (GEN "quadbezier") and these do not have the same flexibility as cubic ones.

3.1 Envelopes on musical staves

Perhaps the fastest method to represent complex and articulated envelopes is to use normal additional staves where for each parameter the profiles will be drawn through pitched notes. Handling notes and durations is the most natural thing in Abjad and this would allow for easy algorithmic manipulations. Moreover, it can be interesting to have an approach of proportional durations definable with rhythmic figures, where more musical parameters of the same instrument can be easily compared and played with subtle synchronizations.

Here we will give an example with a single instrument (FM1) and a single parameter (the IndFM1 modulation index) for the sake of clarity. Let's start with the orchestra loaded in ctcsound.

```
orc_text = '''

  sr     = 48000
  ksmps  = 8
  nchnls = 2
```

```
  0dbfs  = 1

  instr FM
   kamp = .3
   kcps = cpsmidinn(p5)
   kcar = 1
   kmod = .6
   kndx table p4,100; read from Abjad staff "IndFM"
   asig foscili kamp, kcps, kcar, kmod, kndx, 1
   outs asig, asig
  endin

  instr lin
   kval linseg p5,p6,p7,p8,p9,p10,p11,p12,p13,p14,p15,p16,
p17,p18,p19,p20,p21,p22,p23,p24,p25
   tablew kval, p4, 100
  endin
  '''
```

We want to infer the vertices of the envelope for a linseg to handle the modulation index of the FM from the notes on a specific staff. We establish that the range goes from 0 (middle C) to 20 (C 2 octaves above). Since we do not know *a priori* how many vertices the parameter IndFM1 will have for each FM1 instance, we have created an instr "lin" with a very long p-fielded linseg. If less p-fields are used, Csound actually warns, but does not protest and this allows us the flexibility to create very different profiles in the score. This is the starting score.

```
  sco_text = [''']
  f 1 0 16384 10 1 ;sine
  f 100 0 2048 -2 0 ;only for envelopes
  ;How score should look like
  ;            channel midinote
  ;i "FM" 0 5  0       69
  ;            ch linseg_values(no fixed number of pfields)
  ;i "lin" 0 5 0  1 2.5 20 2.5 1
  ''']
```

As you can see from the commented part, each "FM" is activated simultaneously with its "lin" instr and linked by a unique channel so that while "lin" writes, "FM" reads the k data. For each new instance of "FM" the reading channel is updated, provided by ftable 100, so that there is never any interference between any overlapping instances.

The conversion from musical score to Csound happens as follows

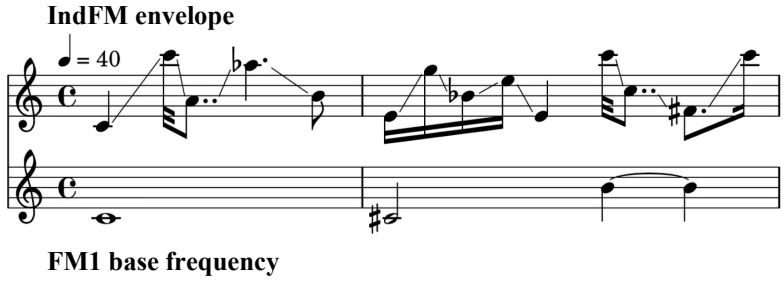
```
channel = 0
instrument_list = ['FM1']
for instrument in instrument_list:
  for logical_tie in abj.iterate(score[instrument]).logi-
cal_ties(pitched=True):
```

```
    offset = abj.inspect(logical_tie).timespan().start_off-
set
    offset_seconds = 60*offset/
(metronome_mark.reference_duration *
metronome_mark.units_per_minute)
    dur = abj.inspect(logical_tie).duration()
    dur_seconds = 60 * dur / (metronome_mark.reference_du-
ration * metronome_mark.units_per_minute)
    scoLine = ['\n','i
"FM"',str(offset_seconds.__float__()),
str(dur_seconds.__float__()), str(channel)]
    sco_text.extend(scoLine)
    sco_text.append(str(60 +
abj.NumberedPitch(logical_tie[0])))
    #Envelope for IndFM
    sco_text.extend(['\n','i "lin"',
str(offset_seconds.__float__()),
str(dur_seconds.__float__()), str(channel)])
    for segment in
abj.iterate(score['IndFM1']).logical_ties(pitched=True):
      segmentStart =
abj.inspect(segment[0]).timespan().start_offset
      if offset <= segmentStart < (offset+dur):
        value = abj.NumberedPitch(segment[0]).__float__() /
24 # scaled 0 -> 1 between c' c'''
        segmentDur = abj.inspect(segment).duration()
        segmentDur_seconds = 60 * segmentDur /
(metronome_mark.reference_duration *
metronome_mark.units_per_minute)
        scaledValue = value * 20  # scale relative to the
particular parameter
        sco_text.extend([str(scaledValue),
str(segmentDur_seconds.__float__())])
    sco_text.append(str(scaledValue))# last value repeated
  channel += 1
sco_text = ' '.join(sco_text)
```

As can be seen, the conversions are quite similar to those shown above, but this time the iteration that inspects the notes-vertices of the IndFM parameter is filtered by the time window corresponding to the duration of the relative FM1 note. Finally, the rescaling operated on 24 semitones is spread over a range from 0 to 20.

Notes and final image.

```
IndFM1staff = abj.Staff("c'4 c'''32 a'8.. a''4. b'8 e'16
g'' b' e'' e'4 c'''32 c''8.. fs'8. c'''16 ", name='IndFM1')
FM1staff = abj.Staff("c'1 cs'2 b'4~ b'4 ", name='FM1')
```



References

1. Lazzarini, V. et al.: Csound: A Sound and Music Computing System. Springer (2016)
2. Lazzarini, V.: Computer Music Instruments. Springer (2017)
3. Baca L., Oberholtzer J. W., Trevino J., Adan V.: "Abjad: An Open-Source Software System for Formalized Score Control" in Proceedings of the International Conference of Technologies for Music Notation and Representation, 2015
4. Oberholtzer J. W.: A Computational Model of Music Composition. Doctoral dissertation, Harvard University, 2015
5. Trevino, J. R.: Compositional and Analytic Applications of Automated Music Notation via Object-oriented Programming. Doctoral dissertation, University of California, 2013