Educational Tools for Csound

Gianni Della Vittoria,

Liceo Artistico e Musicale “A. Canova” di Forlì (Italy)
gianni.dellavittoria@liceocanovaforli.edu.it

Abstract. This paper presents some ideas to produce software applications aimed at simplifying the musical composition process with Csound for students and beginners. The tools developed for this purpose aspire to reduce the distraction that comes from having to carry out intermediate tasks that could be automated. They concern the editor's text expansion, automatic GUI and plotting, fast syntax for operations with arrays, python-style list comprehension, multichannel expansion for Csound opcodes. Although initially designed for the beginner, these tools may also prove useful for experienced users, freeing them from mechanical secondary tasks.

Keywords: workflow, text expansion, compositional tools, fast syntax, GUI, multichannel expansion, arrays, list comprehension

1 Introduction

Working with children in a musical learning experience in which the beginner's effort is aimed first of all at orienting himself in the face of an ambitious project such as composing a piece of music with a symbolic language like Csound, shows which are the main obstacles under a different light and suggests possible approaches.

From these observations arises the need to develop work tools capable of shortening the distance from the imagined goal, perhaps to discover that they could even be useful to the expert user. With the present paper I want to explore some ideas that can be the basis for designing software tools for helping Csound learners.

Basically everything starts from the Csound code, which remains at the core. Some of the facilities I propose aim at reducing the distracting power of secondary tasks that could be automated. I suggest a syntax that can decorate the original Csound code in order to hid what is not strictly necessary to the user, such as code specific to create graphical controllers or plot signals.

These ideas can take tangible form in a set of tools or directly in a unique workspace application, but some of them, like the array facilities, could be considered to be safely included in the Csound parser itself with no apparent conflicts with the current syntax.

2 Text expansion on more lines with commented name variables for opcodes and UDOs

Suppose we have built a complex audio process with about twenty control parameters and we have enclosed it in a UDO with the name of BigSynth.

How should text expansion take place in an educational editor, wishing to simplify as much as possible the association of the value with the right parameter? The practice to display the names of the parameters is usually adopted, which can then be replaced with numerical values by stepping through them with the tabulation. However, if you want to keep the name of the parameter visible, you have to transcribe this name into a variable and copy it above. It might as well be that the text expansion itself already proceeds with this operation, so as soon as we type “BigSynth” we get:

```
kAmp        = .2
kFundFreq   = 220
kPar3       = .95
kPar4       = 486.94
 ...
kPar20      = 1.02
aOut BigSynth kAmp,kFundFreq, kPar3, kPar4, … kPar20
```

Certainly clear, but these variables could conflict by homonymy with those of other UDOs or opcodes. Then perhaps it is preferable that the parameter name is simply suggested as a comment.

```
aOut = BigSynth(
  /*kAmp*/      .2,
  /*kFundFreq*/ 220,
  /*kPar3*/     .95,
  /*kPar4*/     486.94,
  …
  /*kPar20*/    1.02
)
```

A toggle command will then suffice to return all the values on a single line without comments, if the user should prefer a contracted form.

The parameter values should be pre-entered by default as much as possible, so that the user can immediately listen to the working UDO and intervene just on what he wants to change. It will be the user himself who defines these default values for subsequent work sessions, with even the possibility of having complex behaviors. For example, instead of kAmp = .2 it could be defined as:

```
/*kAmp*/    linenr:k(1,.02,3,.01),
```

3 Fast syntax for calling graphics controllers

Sometimes, while we are in the preparation phase of the sound material, we need graphic controls maybe just to quickly experiment with some possibilities, without the distraction of having to concentrate on creating a GUI.

A very simple way is to add a keyword next to the number you want to turn into an envelope. So if you want to display a slider for the amplitude and one for the frequency of an oscillator, just put the words *@sl* next to the respective numbers

```
poscil(.2@sl, 1000@sl, ifn)
```

where .2 and 1000 represent the maximum value of the ranges. The default minimum is .0001, but it can be changed. Furthermore, more precise indications can be given by adding elements: 2000 *@sl_log* for example to produce logarithmic interpolation.

3.1 Envelope Designer

If instead of placing the *@sl* symbol we had used *@env* next to the parameter value, we would have opened a graphical interface that enables us to design a sophisticated envelope in which each segment can have its own curve (linear, exponential, cosine, ...) and an independent curvature coefficient, as well as a proportionate lfo and a random component.

The number next to which the *@env* symbol is placed also represents the full scale value here. Each segment can then have absolute duration values or relative to (p3 - absolute durations). It should be possible to have or create even very complex envelope presets from which the user can easily choose.

Once he has perfected the desired envelope profile, the envelope will be created directly in Csound language, so the user can return the Csound code snippet obtained in the original file, perhaps by making the graphical design interface disappear.

3.2 GEN Table Builder

To draw a waveform with GEN 10 and 11, it would be useful to be able to view and edit the harmonics with a graphic approach. Just put the words *@gen* next to the number that represents the function and the graphic control interface will appear.

3.3 GUI presets

Counting on the fact that the data obtained from all the GUI controllers of the entire .csd file can be saved, it allows you to recall certain sound situations obtained perhaps after extensive experimentation, and to reuse the acquired sounds also in the future in the form of multiple variations.

The user will therefore no longer have to worry about making many versions of the same instr or UDO just to write down the most interesting values found, if he wants to rely on the presets system.

4 Plotter

The opportunity to view graphs is particularly useful for educational purposes, and must be immediately accessible by the user, without unnecessary distractions. In this way, thanks to the usual system of placing a symbol next to the variable to be investigated, it should be possible to monitor control (k) and audio (a) variables, as well as spectra and spectrograms.

It is important to be able to understand how certain parameters move in relation to the real-time sound performance, for which tools are provided that are able to align and compare audio variables with k variables.

Each variable labeled *@plot* will be displayed, and it could also be interesting to show overlays from different instances playing simultaneously.

5 Arrays

5.1 Array shortcuts syntax

To facilitate the use of arrays in a more dynamic form, in addition to the regular Csound opcodes, the use of a syntax is being tested where the array is directly declared with the data enclosed in square brackets, followed by the @ symbol:

```
[1, 2, 3]@
```

The @ symbol can actually be omitted, since the system detects the use of square brackets with contents other than what would be syntactically acceptable. If the user prefers to make it clear that it is a syntactic variation with respect to the pure Csound language, or this syntax should conflict with that of future versions of Csound, he can make the usual symbol explicit.

A series of elementary operations on one-dimensional arrays has been codified, again in order to make the use of these very useful tools more immediate. Here is an excerpt showing the equivalent syntax in the current Csound language (6.17).

Table 1. Arrays operations

New syntax	Equivalent Csound	Description
iar[] = [1, 2, 3]@	iar[] = fillarray(1, 2, 3)	1 2 3
iar[] = [1, 2, 3]	iar[] = fillarray(1, 2, 3)	1 2 3
kenvs[] = [k1, k2, k3]	kenvs[] = fillarray(k1, k2, k3)	k1 k2 k3
[1..10]	genarray(1, 10)	1 2 3 4 … 10
[0..1, .1]	genarray(0, 1, .1)	0 .1 .2 .3 … 1
[1, 2, 3] + [4, 5, 6]	fillarray(1, 2, 3) + fillarray(4, 5, 6)	5 7 9
[1, 2, 3] +@ [4, 5, 6]		1 2 3 4 5 6
[1, 2, 3] * 2	fillarray(1, 2, 3) * 2	2 4 6
[1, 2, 3] *@ 2		1 2 3 1 2 3

5.2 Multichannel expansion

As in SuperCollider, multichannel expansion consists in automatically transforming a normal opcode into an array with various instances of the same opcode. The mechanism works based on the use of an array as the input value of an opcode where only a scalar value would be allowed. Let's look at the following example.

```
1   iatc[] = 1/[1..10]             ;1,.5, .33, .25, …
2   kenv[] = linseg(0, iatc, 1, p3-iatc, 0)
3   kfreq = 100 + lfo(10, 1)
4   asound[] = poscil(kenv, kfreq * [1..10])
5   out sum(asound) / 10
```

In line 1 an array is created, then used as the attack time of a linseg envelope in line 2. Since the linseg opcode does not accept an array among its durations, the mechanism is activated that transforms the linseg into an array of various linsegs, each with an attack time taken from each value of the *iatc* array. Basically, it is as if line 2 was normally written like this:

```
kenv[] = fillarray(
    linseg(0, iatc[0], 1, p3-iatc[0], 0),
    linseg(0, iatc[1], 1, p3-iatc[1], 0),
    linseg(0, iatc[2], 1, p3-iatc[2], 0),
             ...
    linseg(0, iatc[9], 1, p3-iatc[9], 0)
)
```

In line 4 we have a similar case, but this time with 2 arrays: *kenv* and [1..10]. So also here *asound* will become an array of 10 audio channels, with frequencies equal to the first 10 harmonics of a vibrato fundamental of 100 Hz, and amplitudes with a triangular envelope with 10 different attacks. Normally written, it would have looked like this:

```
asound[] = fillarray(
    poscil(kenv[0], kfreq * 1),
    poscil(kenv[1], kfreq * 2),
    poscil(kenv[2], kfreq * 3),
           ...
```

```
      poscil(kenv[9], kfreq * 10)
  )
```

Finally, in line 5, the opcode sum, already in Csound, is responsible for adding up all the signals from the array in order to obtain the mix for the output. It is clear how much code saving is appreciable, without going to far from clarity.

If the lengths of the input arrays were different from each other, the shorter arrays would loop until the length of the larger one was exhausted.

5.3 Array with Python list comprehension

Another way to create arrays, even very complex ones, could be to resort to the syntax of Python list comprehensions. With the constraint of creating a list of numbers, which will then be automatically converted into an array of floats, you can use all the formulas provided by Python for list comprehensions, with various nested loops of *for* and *if* filters. I'm not here to explain the details, which are well documented online, but I'll just give a small example to appreciate the countless possibilities.

If we wanted to build a harmonic sound with 10 sinusoidal sounds and 4 detuning copies for each harmonic, we could do it in a single line using the list comprehension.

```
a1[] = poscil(
 kamp,
 kbaseFq * [i+j/100 for i in range(1,11) for j in range(5)]
)
```

The list comprehension [i+j/100 for i in range(1,11) for j in range(5)] in fact produces this list of coefficients:
[1.0, 1.01, 1.02, 1.03, 1.04, 2.0, 2.01, 2.02, 2.03, 2.04, 3.0, 3.01, 3.02, 3.03, 3.04, 4.0, 4.01, 4.02, 4.03, 4.04, 5.0, 5.01, 5.02, 5.03, 5.04, 6.0, 6.01, 6.02, 6.03, 6.04, 7.0, 7.01, 7.02, 7.03, 7.04, 8.0, 8.01, 8.02, 8.03, 8.04, 9.0, 9.01, 9.02, 9.03, 9.04, 10.0, 10.01, 10.02, 10.03, 10.04]

6 Organization of files, instr, UDOs, snippets and presets

The availability of a lot of material and many possibilities must not let our guard down on one of the central issues in the coordination of a complex project such as that of writing a piece with a symbolic language.

In a didactic environment, therefore, it is necessary to reflect on this theme. A requirement of a good student work platform is to be able to find what you need without too many distractions. Then you need an efficient search facility in the contents of files, but also a file archiving system that enriches information. Thus, next to the system of folders in which all the material created up to now is neatly arranged and which could be useful as an inspiration for future work, a tagging system that crosses all the files transversely according to different and more logical groupings seems more useful than folders.

It would also be a good practice to include a description in each file we save, which will then be shown in the search dialog box.

References

1. Lazzarini, V. et al.: Csound: A Sound and Music Computing System. Springer (2016)
2. Heintz J., Sigurðsson H. et al.: The Csound FLOSS Manual (2020)
3. Csound Github site, `http://csound.github.io`