Modeling a 'Classic' Hardware Sequencer in Csound: The Design and Use of the *sequ* Opcode

John ffitch[1] and Richard Boulanger[2]

[1] Alta Sounds
[2] Berklee College of Music
`jpff@codemist.co.uk rboulanger@berklee.edu`

Abstract. Over the years, there have been many instruments, designed and shared, that model the 'classic' analog step-sequencer. Some used the *table* opcodes, some did it with *Gens*, some employed score macros and score commands, and others simply copy-pasted lines in the note-list. More recently, impressive sequencer instruments are being built with arrays, and the *schedule*, *schedkwhen*, and *event* opcodes. These designs have ranged from the simple to the sublime and reveal many wonderful and inspiring approaches. All are worthy of study and imitation. Still, beginners always ask, "How can you do sequencing in Csound?" This question often leads into a deeper dive than they are ready for. Or they ask, "Does Csound have a sequencer opcode?" Until recently, the answer to that question was "no", but now the answer is "yes!". This paper will introduce the *sequ* opcode, discuss how it was designed, show how it works, and showcase some of the novel features, and the more esoteric possibilities, associated with its unique design.

Keywords: Csound, sequencer, *sequ*

1 The Inspiration

While collaborating on a new generative work that featured algorithmic permutations of short pitch and rhythm patterns using the existing opcodes it became clear to the authors that there might be a more efficient way of coding some of these desired behaviors.

```
; Starts with an 8 note riff (no pitch)
; - every 5 bars, two notes are randomly swapped
ib = 0.2 ; beat multiplier
gir[] fillarray 0,1,2,3,4,5,6,7,8
gil[] fillarray 0,ib,ib*1.5,ib*.5,ib*.5,ib*.5,ib*.5,ib*1.5,ib
   schedule 1,1,0
giCount = 0

instr 1
  iTime = 0
  iCount = 1
top:
    if iCount>8 goto ending
    schedule 10+gir[iCount], it, gil[gir[iCount]]
    iTie += gil[gir[iCount]]
    iCount += 1
  goto top
ending:
  giCount += 1
;----Mutator----;
if (giCount%5==0) then
    ir1 = rnd(7)
    ir1 = int(ir1)+1
    ir2 = rnd(7)
    ir2 = int(ir2)+1
  if (ir1 != ir2) then
    itmp = gir[ir1]
    gir[ir1] = gir[ir2]
    gir[ir2] = itmp
  endif
endif
  schedule 1, it, 0
endin

instr 11, 12, 13, 14, 15, 16, 17, 18
  kl   linseg 0, 0.01, p3-0.11, 0.1, 0
  a1   oscil 0.8, 440+p1
        out  a1*kl
endin
```

Discussions followed about the many ways to take the piece, and expand the model above to support these new directions, when it occurred to the authors that it might be useful, in this project, and in general, if an opcode that offered these permutational possibilities was written. As we listened to the 'riffs' we were generating, we were were instantly lead back to the modular electronic music if the 70's and 80's produced with 'classic' hardware sequencers and arpeggiators. We thought that if we added some of the variational and permutational possibilities of our "riff-generator" to a traditional sequencer that we could introduce both a familiar set of options, and some very new creative and sonic possibilities to Csound. With that goal *sequ* was born.

2 How *sequ* Works

The *sequ* opcode is an array-based step-sequencer. One version of the opcode supports one-dimensional arrays for rhythm, instrument, and data, and the other supports a two-dimensional array for data specification.

```
kres  sequ  irhythm[], iinstr[], idata[], kbpm, klen
            [, kmode] [, kstep] [, kreset] [, kverbose]

kres  sequ  irhythm[], iinstr[], idata[][], kbpm, klen
            [, kmode] [, kstep] [, kreset] [, kverbose]
```

The opcode has k-rate support for sequence speed – *kbpm*, in beats per minute, and length *klen*, and optional k-rate control of mode, step, and reset. Given the underlying hardware sequencer model, it is easy to see that the developers were emulating the traditional voltage-control of rate, length, step, and reset. In fact, many of the supported "modes" are found on classic analog hardware sequencers as well.

- 0 - forward loop
- n > 0 - forward loop with a mutation every n events
- -1 - backward loop
- -2 - back and forth
- -3 - random events
- -4 - play the entire sequence forward one time and stop
- -5 - play the entire sequence backward one time and stop
- -6 - shuffle the events
- -7 - reset to the initial state

3 Under the Hood

The basic design follows the original Csound language version. An internal array is initialised to the simple sequence [0, 1, 2, 3, ... len] and when it is time to start a new cycle this array is used as an indirection to the argument arrays of notes. When it is time to change the sequence, for example by swapping two notes, the change is made in the internal sequence. The individual notes are scheduled via the API function `csoundReadScore` with arguments specified in the 2-dimensional array . Most of the complexity of the code has to do with keeping track of where one is in the sequence for each mode, and their interactions.

```
// Mutate every mode events

if (mode > 0 && len>1 && p->cnt%mode == 0) {
  int r1, r2;
  do {
       r1 = rand()%len;
       r2 = rand()%len;
      } while (r1==r2);
  {
      int tm = p->seq[r1];
      p->seq[r1] = p->seq[r2];
      p->seq[r2] = tm;
```

```
        if (*p->verbos)
              printf("swap %d and %d\n", r1, r2);
      }
    }
```

4 Some Unique Features

Imagine setting up the opcode with a traditional 8-step sequence of rhythms and pitches, and on step 3 triggering a generative meta-instrument producing ebbing random clouds of minced audio and on step 7 triggering low random kicks and thuds from another instrument. Or, how about one master *sequ* instr "seqControl", launching *sequ* instr "Seq1" on step 1, and *sequ* instr "seq2" on step 2, instr "seq3" on step 3, and instr "seq4" on step 4; all with the same pitches and rhythms; all starting at the same BPM; but each, slowly moving away in timbre, space, and time. Or one could imagine using the *kstep* argument and assigning a specific MIDI note or ASCII key to jump to a step, maybe after step 8, that momentarily triggers a generative instrument that takes the piece off into a very different direction. As such, it could serve as an important structuring device in a live or generative composition. Clearly, the *iinstr* argument offers a rich set of permutational and control possibilities.

The instrument below shows using an ASCII trigger to reset the sequence manually:

```
 instr 1
   gkNumber, gkPress sensekey

  if changed(gkPress) == 1 then
   ; ascii 114 = lowercase letter 'r' - for reset
     if (gkNumber == 114) then
       kreset = 1
     else
      kreset = 0
    endif
  endif

  irhythms[] fillarray 1, 1.5,0.5, 0.5, 0.5, 0.5, 1.5, 1
  iinst[] fillarray 11,12,13,14,15,16,17,18
  inotes[] fillarray 60, 61, 62, 63, 64, 65, 66, 67
  kstep init 0
  kres  sequ  irhythms,iinst,inotes,180,8,0,0,kreset,1
endin

instr 11, 12, 13, 14, 15, 16,17, 18
  k1  linseg 0, 0.01, p3-0.11, 0.1, 0
  a1  oscil  0.9, cpsmidinn(p4)
```

```
   outall    a1*kl
  endin
```

This instrument features two sequences going out of phase with each other in the left and right channels.

```
instr 1
  irhythm[] fillarray 1, 1.5, 0.5, 0.5, 0.5, 0.5, 1.5, 1
  inst0[] fillarray 11, 12, 13, 14, 15, 16, 17, 18
  inst1[] fillarray 19, 20, 21, 22, 23, 24, 25, 26
  inotes[] fillarray 60, 61, 62, 63, 64, 65, 66, 67
  kspeed line 60, p3, 180
  kSeq0 sequ irhythm, inst0, inotes, kspeed, 8
  kSeq1 sequ irhythm, inst1, inotes, kspeed * 1.2, 8
endin

instr 11, 12, 13, 14, 15, 16, 17, 18
  kl linseg 0, p3*0.01, 1, p3*.99, 0
  a1 oscil 0.9, cpsmidinn(p4)
  outs1 a1*kl
endin

instr 19, 20, 21, 22, 23, 24, 25, 26
  kl linseg 0, p3*0.01, 1,p3*.99, 0
  a1 oscil 0.9, cpsmidinn(p4)
  outs2 a1*kl
endin
```

Another powerful feature of the *sequ* opcode is the two-dimensional data array allowing for the sending of multiple control parameters, synchronized p-fields, to an instrument from a single *sequ* opcode:

```
instr 1
  ;; rhythm array -  values are multiplied by tempo (ticks) in BPM

  irhythm0[] fillarray 1, 1.5, 0.5, 0.5, 0.5, 0.5, 1.5, 1
  ;; instrument array - instrument number to render for each step
  iinsts0[] fillarray 11, 12, 13, 14, 15, 16, 17, 18

  ;; note array - cpsmidinn(p4) amp(p5) modRatio(p6) modIndex(p7)
  ;; - esentially  'p4', 'p5', 'p6' and 'p7' are output from sequ
  ;; initialize 4 rows with 8 columns

  inotes[][] init 4,8
  inotes fillarray 60, 61, 62, 63, 64, 65, 66, 67, \
  0.9, 0.3, 0.8, 0.2, 0.7, 0.4, 0.5, 0.6, \
  1, 2, 3, 4, 5, 6, 7, 8, \
  1, 11, 2, 12, 3, 21, 4, 22

  ;; variable tempo
  kspeed linseg 85, p3*.7, 85, p3*.3, 240

  kSeq sequ irhythm0, iinsts0, inotes, kspeed, 8, p4, 0
endin

instr 11, 12, 13, 14, 15, 16, 17, 18
  kenv linseg 0, p3*0.01, 1, p3*.99, 0
  asig foscil p5, cpsmidinn(p4), 1, p6, p7
  outall asig * kenv
endin
```

Further, it is important to note that although the arrays are i-rate variables, their elements can be updated and changed on-the-fly!

Given the limits of space in this paper, the bibliography will feature links to an additional set of models and excerpts from pieces exploring the new possibilities of this powerful and versatile opcode. [3]

5 Conclusion

The new *sequ* opcode brings a standard module from the 'classic' synthesizer to Csound, and adds some powerful possibilities given its array-based design, the specific k-rate controls, and the supported playback and permutational modes. The authors welcome input and suggestions for improvements to the opcode and for other 'optional' arguments that could, and maybe should, be added to make *sequ* even more versatile.

[3] Some of these will be demonstrated live in the paper presentation at ICSC22.