Interactive Csound coding with Emacs

Hlöðver Sigurðsson

Abstract. This paper will cover the features of the Emacs package *csound-mode*, a new major-mode for Csound coding. The package is in most part a typical emacs major mode where indentation rules, completions, docstrings and syntax highlighting are provided.
With an extra feature of a REPL[1], that is based on running csound instance through the csound-api. Similar to csound-repl.vim[1] csound-mode strives to enable the Csound user a faster feedback loop by offering a REPL instance inside of the text editor. Making the gap between development and the final output reachable within a real-time interaction.

1 Introduction

After reading the changelog of Emacs 25.1[2] I discovered a new Emacs feature of dynamic modules, enabling the possibility of *Foreign Function Interface(FFI)*. Being insired by Gogins's recent Common Lisp FFI for the CsoundAPI[3], I decided to use this new feature and develop an FFI for Csound. I made the dynamic module which ports the greater part of the C-lang's csound-api and wrote some of Steven Yi's csound-api examples for Elisp, which can be found on the Gihub page for CsoundAPI-emacsLisp[4].

This sparked my idea of creating a new REPL based Csound major mode for Emacs. As a composer using Csound, I feel the need to be close to that which I'm composing at any given moment. From previous Csound front-end tools I've used, the time between writing a Csound statement and hearing its output has been for me a too long process of mouseclicking and/or changing windows. I needed something to enhance my Csound composition experience and I decided to develop a new Csound major mode for Emacs, called simply *csound-mode*.

Other Emacs packages already exist for Csound, one called *csound-x*[5] and an older one from John ffitch[6]. Both of them are based on dual mode between orchestra and score, *csound-x* takes the dual mode setup presented in John's Emacs packages to a more complete level. That which differentiates *csound-mode* from *csound-x* in terms of user experience is that *csound-mode* does not attempt to separate a csound document(csd) in two modes and seeks to keep the user in one buffer[2]. *csound-x* also makes Csound very configurable where paths and options can be configured separately from global Csound options, *csound-mode* tries to keep all Emacs configuration at minimum and respects all system

[1] REPL stands for read-eval-print-loop and is a term that is used in many programming languages that offer a language interpreter in a shell, like which can be found in Python, Node and Clojure.

[2] A buffer is an Emacs lingo for a window, as you can have multiple tabs open in many text editors you can have multiple buffers running inside Emacs.

paths and global Csound configuration should one exist. Furthermore *csound-x* or its dependencies are not compatible as of today with Emacs version 25 or newer whereas *csound-mode* is not compatible with Emacs version 24.5 or older.

I hope this tool will be as useful for other composers as it is for me. In the following sections I will go further into the mechanics of *csound-mode*.

2 Csound coding style

No official community driven Csound style guide exists, which makes Csound coding style today rather unstandardized. When deciding on indentation rules for *csound-mode* I had to make few opinionated decisions, based on common best practices. Through informal observation of Csound code from various Csound users throughout the years, it can be noticed that some coding tendencies are fading away. For example indented **instr** statements are very rare in modern Csound code.

```
          instr 2
a1        oscil     p4, p5, 1      ; p4=amp
          out       a1             ; p5=freq
          endin
```

Fig. 1. Richard Boulanger's toot2.orc

Rather in most cases, **instr** and **endin** statements are set to the beginning of line with its body indented to right.

```
instr  1
  aenv      linseg    1,p3-.05,1,.05,0,.01,0
  a1        oscili    p4, 333, 1
            outs      a1*aenv,a1*aenv
endin
```

Fig. 2. John ffitch's beast1.orc

What makes these two figures clear and easy to read, is the fact that a visual distinction is given between the return value, operators/opcodes, input parameters and optional comments at the end. While this holds true in most cases, with the introduction of indented boolean blocks, this can quickly get messy.

```
    opcode envelope, a, iiii
  iatt, idec, isus, irel  xin
    xtratim   irel
  krel     release
if (krel == 1) kgoto rel
  aenv1     linseg    0, iatt, 1, idec, isus
  aenv     =  aenv1
    kgoto     done
rel:
  aenv2     linseg    1, irel, 0
  aenv     =  aenv1 * aenv2
done:
    xout      aenv
    endop
```

Fig. 3. Jonathan Murphy's UDO envelope.udo

Despite its old fashioned indentation in **Fig 3**, it can be seen that a code with boolean blocks does not align well with code that is otherwise trying to form vertical blocks based on outputs and operators. Therefore, as a matter of opinionated taste, code that forms boolean blocks should be visually aligned to one another, and for each new depth of nesting, an equal width of indentation should be added. The following figure displays indentation pattern which follows the default *csound-mode* indentation.

```
opcode gatesig, a, ak
  atrig, khold xin
  kcount init 0
  asig init 0
  kndx = 0
  kholdsamps = khold * sr
  while (kndx < ksmps) do
    if(atrig[kndx] == 1) then
      kcount = 0
    endif
    asig[kndx] = (kcount < kholdsamps) ? 1 : 0
    kndx += 1
    kcount += 1
  od
  xout asig
endop
```

Fig. 4. Steven Yi's UDO gatesig.udo

In *csound-mode* the indentation width defaults to 2 spaces, this width is adjustable with a customizeable global variable in Emacs under *csound-indentation-spaces*. Single line indentation is in Emacs by default bound to the `<TAB>` key, like with most in Emacs, this is highly customizeable. *csound-mode* happens to work very well with the minor-mode *aggressive-indent-mode*[7], which automates indentation based on buffer's current active major-mode indentation rules and can in turn saves many keystrokes.

The score section is simpler and more straightforward in terms of indentation as a score statement is separated with newlines. If we look at spreadsheet applications like *Microsoft Excel* we see why they are useful for composing Csound scores. They provide resizable vertical and horizontal blocks, giving a clear distinction between each parameter field. *csound-mode* comes with the function *csound-score-align-block* that indents sequence of score events, separated by newline. As will be mentioned later, *csound-mode* treats series of score events seperated by newline as a score-block unit. The function *csound-score-align-block* is as of current version bound to `<C-c C-s>`, by applying this function with the cursor located anywhere within the score-block, all common score parameters will adjusted to the same width.

```
i 1 0 100 440 0.000001
i 1 100 100 850 10
i 1 2000 1 60 1|

;; C-c C-s
i 1 0    100 440 0.000001
i 1 100  100 850 10
i 1 2000 1   60  1|
```

Fig. 5. Score-block transformation in csound-mode where the pipe character represents cursor location.

By setting the variable *csound-indentation-aggressive-score* to true, the function *csound-score-align-block* will be called on every indentation command. This function was designed function with *aggressive-indentation-mode*, but for very long score blocks it may cost a lot of CPU.

In conclusion, the way which *csound-mode* indents Csound code is based on the depth of nesting of boolean statements within the orchestra part or file of given Csound code. And for the score part or file, no statement will be indented to right but a possibility of aligning blocks of score statements is available to the user.

3 Syntax highlighting

csound-mode utilizes the built in *font-lock-mode* to provide syntax highlighting. Most colors can be modified via `M-x customize-face` if wished. For example if

the user wishes to change the color of i-rate variables, it would be found under *csound-font-lock-i-rate* and global i-rate variables under *csound-font-lock-global-i-rate* etc.

By default *csound-mode* comes with enabled rainbow delimited parameter fields, meaning each parameter within a score statement will get a unique color, up to 8 different colors before they repeat. This can potentially give visual aid to scores statements that have many parameters. If wanted, this feature can be turned off with the customizeable variable *csound-rainbow-score-parameters-p*.

4 Completions and documentation

Like most Emacs major-mode packages, *csound-mode* will also provide *eldoc-mode* functionality and autocomplete, where autocomplete can be *ac-mode* or the more modern version of it, *company-mode*. All the data for completions and *eldoc-mode* is based on crude xml parsing of the Csound Manual, meaning that some opcodes and symbols could be missing. The documentation data is stored in a hash-map and is static, meaning that global symbols from UDOs (user defined opcodes) evaluated into the Csound instance, do not enjoy the benefits of *font-lock* syntax highlighting or *company-mode* completions.

The echo-buffer is where the docstrings from the autocomplete suggestions and argument names are printed into. This is where *eldoc-mode* plays a big role in *csound-mode*. While user is typing in values for a given opcode's parameter, *eldoc* will highlight the current argument at the point of the cursor. This works on multiple lines for opcodes and operators, as well as for nested (functional-style) opcodes calls, which are given argument values between parenthesis.

5 Csound interaction

Before talking about the Csound REPL, it's worth mentioning the two "offline" possibilites, namely the functions *csound-play* bound to `<C-c C-p>` and *csound-render* bound to `<C-c C-r>`. Running those functions pops up compilation buffer that gives user all logs printed to stdout and stderror from the command. Calling *csound-play* is the same as if `csound -odac file.csd` would be typed into the command line while *csound-render* would equal to `csound -o filename.wav filename.csd` where *csound-mode* will prompt the user for a filename before rendering.

When starting Csound REPL via *csound-start-repl* a new buffer called ***Csound REPL*** will open. This buffer is running on *CsoundInteractive* for major-mode and *comint-mode* as minor mode. *comint-mode* provides a prompt functionality, enabling evaluation of the user input into the prompt, as well as storing history of the commands given to the prompt. As of current version, score statements are the only commands that the prompt understands, this is expected to be extended in future versions.

With an open REPL buffer, a Csound instance is running in the background with

indefinite performance time. Through this Csound instance, live-coding and/or live-interaction can take place, trough two different functions bounded to keys. Which are *csound-evaluate-region* bound to `<C-M-x>` and *csound-evaluate-line* bound to `<C-x C-e>`.

csound-evaluate-region is more dynamic of the two, it tries to recognize a Csound score or orchestra statement at the point of cursor, which may or may not cover multiple lines. After a Csound statement is recognized, the string is sent to forementioned Csound instance to be evaluated, followed by a "special effect" where the code that was recognized gets highlighted for a sub-second. The result of this operation should get printed immediately into the REPL buffer, where Csound may print syntax errors if one were found. *csound-evaluate-line* does the same, but will only evaluate the current line of the cursor. *csound-evaluate-line* could be more convenient to run within score blocks, as *csound-evaluate-region* will send all the score statements separated by newline into the Csound instance. As a use-case example, a composer has with these functions the possibility to compose short phrases/bars at a time, separated by newlines and have Csound play immediately the evaluated phrase, as well as evaluating one single score line to hear how one note sounds within a block of notes.

Note that *csound-mode* has a built-in transformation of the score-blocks that are being evaluated. Trough this transformation the lowest value of p2 in the score-block is subtracted from all p2 and p3 fields. This eliminates all waiting time for scores-blocks that have long starting time (high p2 value), but could in some cases be unwanted, in which case playing the file "offline" via *csound-play* could be more suitable option.

6 Conclusion

csound-mode is new and growing package that provides set of functionalities aimed to enhance the flow of the composer. *csound-mode* is developed by a seasoned Emacs user for Emacs users, which may set a barrier for potential new users of *csound-mode* without prior knowledge of the text editor Emacs. Unlike CsoundQt, *csound-mode* does not come with Csound Manual lookup functionality and provides only short documentation snippets via autocomplete and *eldoc-mode*. *csound-mode* is also the only Emacs package for Csound that is now available on MELPA, which is the largest package manager for Emacs packages. Being a new package, it's not battle tested and has potentially bugs, which is why I think it's important that *csound-mode* has a github page where future users can report bugs, suggest improvements and submit pull-requests. Something that other Emacs packages for Csound have lacked up to this point.

References

1. Steven Yi. csound-repl.vim. `https://github.com/kunstmusik/csound-repl`, 2016.
2. Mickey Petersen. What's new in emacs 25.1. `https://www.masteringemacs.org/article/whats-new-in-emacs-25-1`, 2016.

3. Michael Gogins. Steel bank common lisp ffi interface to csound.h. https://github.com/csound/csound/blob/7b4cebf8cf0a3f49232123ee3d752db2116c4c6c/interfaces/sb-csound.lisp, 2017.
4. Hlöðver Sigurðsson. Emacslisp link to csound's api via emacs modules. https://github.com/hlolli/csoundAPI_emacsLisp, 2017.
5. Stéphane Rollandin. Csound-x for emacs. http://www.zogotounga.net/comp/csoundx.html, 2015.
6. John ffitch. Emacs-macros. http://www.cs.bath.ac.uk/pub/dream/utilities/Emacs-macros/, 2003.
7. Artur Malabarbara et al:. aggressive-indent-mode. https://github.com/Malabarba/aggressive-indent-mode, 2017.