The New Developments in Csound 6

John ffitch
University of Bath
Department of Computer Science

Michael Gogins
Irreducible Productions
New York

Victor Lazzarini
National University of Ireland, Maynooth
Department of Music

Andrés Cabrera
University of California, Santa Barbara
Media Arts and Technology

ABSTRACT

In this paper we introduce a major new version of Csound, the audio processing system and library. We begin with an overview of the current status of Csound (version 5), as well as its three layers of use (high, middle, and low). We then outline the design motivations and features of version 6. We continue by exploring external changes and discussing some examples of use. We conclude by looking forward to the next steps in the development of Csound.

1. INTRODUCTION

In 2012, six years after the initial release of the first major re-engineering of the well-known and widely-used software sound synthesis system Csound [1], we (its developers) decided to embark on a further revision of many of its internal and external aspects. Developments since version 5.00 [2] until the current release, 5.19, have been mostly incremental. They have also been limited by our commitment to maintaining both binary and API (Application Programming Interface) compatibility with earlier versions (although the system has actually come through a binary upgrade, after version 5.09). To allow for a number of requested changes, we decided a new major version was necessary, which would mean a break in backwards compatibility (both API and binary). This does not, however, mean a break in backwards compatibility of Csound code and pieces. Older pieces and code will always continue to work with Csound 6. This paper discusses the motivation for Csound 6, its development process, and major features of the new system.

2. WHAT IS CSOUND?

For the ICMC audience, it might not seem necessary to describe such a well-known and established software package. After all, there have been a number of papers on the subject of Csound presented here, over the years [3] [4] [5] [6] [7] [8] [9]. However, it is well worth describing what Csound is in a bit more detail, because 1) Csound has a long history of development, and much of the information describing it is outdated; and 2) the motivation for the present directions will become clearer as we outline the present system.

The best way to describe Csound, in its version 5, is to present it as a series of layers, with various ‘modes of entry’ for users and for related applications.

At the lowest level, Csound is a self-contained audio-programming language implemented in a cross-platform library, with a well-defined API, which allows software developers to create programs for audio synthesis and processing, and computer music composition. Csound supports a variety of synthesis techniques in its orchestra language, and allows various means/levels of internal and external control. Csound is extensible via plugin modules. Software that uses Csound can be written in C, C++, Objective-C, Java, Python, Lua, Tel, Lisp, and others. Csound runs on Windows, Linux, OSX, Solaris, Haiku, Android and iOS.

The middle layer is characterized by writing programs in the Csound language for performance, composition, and other audio processing tasks such as sonification. At this level, the system allows composers to design computer music instruments, and to control them in real time or deferred time. Interaction with the system comes via various frontends, many of which are third party (i.e. not maintained as part of the Csound releases). The ‘classic’ command-line interface (CLI) is the basic frontend, where the system is controlled by a single terminal command. As this was the only original means of using the software, traditionally a number of frontends have been designed to provide a simpler wrapper around CLI Csound. More commonly, today, frontends access the Csound library directly (via its API). These frontends provide diverse modes of interaction. For example, Csound can be embedded in graphical environments such as Pure Data via the csoundapi~ frontend, and in Max/MSP via csound~. Composition environments such as blue use it as a sound engine. For more general-purpose uses, there are integrated development environments (IDEs) for programming with Csound (such as CsoundQt and WinXsound), and plugin/application generators, such as Cs-LADSPA [6] and Cabbage[8].

Copyright: ©2015 John ffitch et al. This is an open-access article distributed under the terms of the Creative Commons Attribution License 3.0 Unported, which permits unrestricted use, distribution, and reproduction in any medium, provided the original author and source are credited.
At this level, Csound co-exists with a number of tools and languages which add support for activities such as algorithmic composition and graphical user interaction.

![Diagram showing the various levels of Csound, related software and end users.](image)

**Figure 1.** The various levels of Csound, related software and end users.

At the highest level, use of Csound occurs by developing applications based on the middle and lower levels. Here, the user might not even know that Csound is involved, as user programming is generally not involved. This is seen, for instance, in some frontends, such as Cecilia and blue, where the user might only need to deal with parameter setting in the graphical interface, in plugins or applications generated by Cabbage, in bundled packages such as Csound4Live (which uses the csound~ frontend), or in mobile applications for iOS and Android.

It is clear that Csound has attracted a diverse set of users, from the expert programmer to the mobile app customer. In addition, thanks to Csound’s long history, especially by composers working at the middle level, there is a legacy of music written with Csound that is worth preserving (and that in fact stretches back to 70s compositions written for Csound’s predecessors MUSIC 11 and MUSIC 360 [10]). This has focused our minds to provide a completely backwards-compatible system (as far as the language is concerned) as *sine qua non* condition for future versions.

Some may criticise this as recipe for an ever increasing language, with its associated complexity penalty, and, as often vocalised in the detractors’ corner, ‘bloat.’ But although some ‘bloat’ is inevitable in a system nearly three decades old, the Csound language is still syntactically very simple (it consists of just a few simple syntactical constructs), and the processing engine is generally efficient in terms of DSP and algorithm implementations. What criticisms fail to consider is that Csound has fostered a vibrant community of users and algorithm implementations. What criticisms fail to consider is that Csound has fostered a vibrant community of users and developers.

We understand that *community* is the biggest asset a system like Csound can have. Without users, expert and non-expert, a system withers and dies. It would not be a huge task (in comparative terms) to ditch the old system and recreate one whose language adapts completely to the flavour-of-the-moment software design. Also, creating a whole new computer music language from scratch is also not too difficult now, especially with the availability of models that exist as open-source code. In fact, there is a multiplication of in-cipient systems that claim to be the intelligent solution to perceived problems in existing software. The majority of these do not cross the 80/20 divide of development. This occurs possibly for a variety of reasons, but especially for the lack of an enthusiastic user (and developer) community. There is great value in the accumulated knowledge of the community and the large body of existing code. We understand that moving Csound away from its origins as a system does not mean ditching users and music along the way. The requirements of the community are paramount to where we want the software to go. By supporting the various levels of entry into the system, we aim to foster interest in the software and in computer music in general. This translates as well into the different levels of difficulty that the Csound language contains. It allows educators to provide a smooth learning curve for students, going from the early (and simpler) set of language elements into the expanded one that the system supports today.

3. WHY CSOUND 6?

By 2012, we began to feel that Csound 5’s incremental model of development was becoming a limitation. At the 2011 International Csound Conference in Hannover, users and developers met to agree on a number of desired features that the software should have in future versions. Some of these (like support for mobile platforms and some additional language features) were achievable in Csound 5 and indeed were soon made available. Others have required a major re-engineering of the system. Among them, we can cite:

- the capacity of new orchestra code, i.e. instruments and user-defined opcodes (UDOs), to be added to a running instance of the engine (enhancing, for instance, live-coding support and interactive sound design);
- major additions to the orchestra language, for instance, generic arrays, debugging/introspection, and a type system;
- rationalisation of the API to simplify its usage and to allow further features in frontends;
- fast loadable (FASL-like) binary formats, API construction of instruments;
- further development of concurrency (enhancement of existing support [7]).

This list was our starting point for the development of Csound 6.

4. INTERNAL CHANGES IN VERSION 6.0

A number of important changes have been made to the code base, which not only introduce significant improvements and scalability in performance (i.e. in parallel processing), but also provide a robust infrastructure for future developments.
Table 1. Relative performance with multiple threads in three existing Csound examples.

<table>
<thead>
<tr>
<th>Threads</th>
<th>CloudStrata ksmps=500 (sr=96000)</th>
<th>Xanadu ksmps=10 ksmps=100</th>
<th>Trapped In Convert ksmps=10 ksmps=100 ksmps=1000</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>0.54</td>
<td>0.57 0.55</td>
<td>0.75 0.79 0.78</td>
</tr>
<tr>
<td>3</td>
<td>0.39</td>
<td>0.40 0.40</td>
<td>0.66 0.76 0.73</td>
</tr>
<tr>
<td>4</td>
<td>0.32</td>
<td>0.39 0.33</td>
<td>0.61 0.72 0.70</td>
</tr>
</tbody>
</table>

4.1 Build system and tests

We have adopted CMake as Csound’s primary build tool, replacing scons as used in Csound 5. We have added test suites for the language and API, as well as individual CUnit tests, to the code base and build system. These changes are well aligned with modern standards of software testing and project development.

4.2 Code reorganisation

We have removed obsolete code, such as the old parser. The CSOUND class has been rationalised and refactored. Some opcodes have been rewritten/substituted, especially in cases where they incorporated special licensing issues beyond LGPL. Syntax checking in the parser has been completely overhauled, and, by extension, the old annotation system used for opcode overloading has been substituted by a simpler and more robust mechanism. Data-structure utilities such as hash tables etc. have been given a clean and easy to maintain interface and implementation.

4.3 Type system

To better support the strong typing of the Csound language and also to allow its expansion, we have implemented a new type system to replace the old hard-coded typing in the parser and compiler. This is more generic and implements tracking of variable names to types. The type system allows the creation of opcodes that accept and produce complex data structures, as well as new semantics for opcode inputs and outputs. It will also allow the development of debugging/inspection tools for Csound code. In addition, the code for the string type has been completely replaced, allowing for dynamic allocation and variable sizes. This was required to allow any size orchestra code to be manipulated as strings, and passed to the compilation stage inside a Csound instrument.

4.4 Asynchronous operations

We implemented mechanisms for the access of files in an asynchronous mode (non-blocking). These mechanisms are generic enough for the use in opcodes and plugins. In cases where the generic mechanism was not suitable (e.g. the diskin opcode), a dedicated solution was implemented. We also added support for asynchronous i-time operations to the engine, which will allow initialisation code to be performed in a separate thread to performance.

4.5 Thread-safety

In Csound 5, library users were expected to take care of thread-safety when splitting performance and control in separate threads (although some helper classes were available for this purpose in the Csound interfaces API). In Csound 6, thread-safety is built into the library, so API calls can be placed in separate threads (e.g. for control, table access and performance). The software bus channels, for instance use gcc atomic built-in functions (i.e. _sync_lock_test_and_set())

4.6 Multicore operation

Following the introduction of multicore support in Csound 5 [11] [7], we have created an improved design with better use of resources. The new design uses more conservative re-drawing of directed acyclic graphs (DAG), which is now done only at the beginning and end of events rather than on every control cycle, and uses watch-lists, as found in SAT-solvers [12]. The effect of this change is significant; in almost all cases it gives major speed-up with two or more threads being used on recent processors, delivering about 60% of the time or better. Some preliminary figures are shown in table 1.

Finding a way of using multiple cores is a major challenge to software writers, and is particularly difficult in audio processing [13]. We think that this scheme will scale to a significant number of cores and open up the possibility for complex synthesis in real time.

It is important to note that parallelism in Csound is completely automatic and provided out-of-box by a single configuration option (requesting a given number of threads). No user modification of Csound code is required, and more importantly, no expertise in how to parallelise code is required. We understand this as a compiler problem, not a user one. Initial tests have indicated that the parallelism is generic enough to provide gains and scalability for arbitrary orchestras, only failing in pathological cases, such as when ksmps=1 and the computation small when the overhead is apparent or where there is no parallelism to find. It might even be possible to have an automatic mode where the code analyser can determine whether there is likely to be and advantage in using multiple threads.

5. EXTERNAL CHANGES

In addition to the developer-level changes listed above, significant external changes are also visible to end-users.
5.1 Generic Arrays

Csound 5 introduced a new type of variable that implemented simple one-dimensional arrays, and with it a suite of operations was also added. In Csound 6 arrays have been generalised, and all types can be constructed as one- or two-dimensional objects. This provides substantial flexibility for users of the language. For instance, we can have code constructs like this, where a bank of oscillators is spawned:

```
opcode OscBank,a,kki
setksmps 1
kamp,kfr,inum xin
kph[] init inum
kcnt = 0
au = 0
until kcnt == inum do
    au += sin(kph[kcnt])
    kph[kcnt] += kfr*kcnt*(2*$M_PI)/sr
    kcnt += 1
od
xout au*kamp
endop
```

Previously, such designs would have had to be implemented via recursive user-defined opcodes or instruments. But now, more straightforward loops can be used. The only care is that, as unit generators (opcodes) are effectively anonymous classes in the current syntax, those whose internal state advances on every call cannot be directly used in loops as in the example above. We are considering a number of possible syntactical solutions, including automatic parallel expansion, so that arrays can be used more freely with opcodes. Functions, and many unit generators that don’t have an evolving internal state (e.g., a phase accumulator) can be used with no limitations. In addition to array data types, we have designed a full set of operations (such as list comprehensions, maps, copying, table access, etc.). We will implement these in subsequent updates.

5.2 New functional syntax

Another major external change to Csound is the possibility of a new functional syntax, where opcodes can be used in expressions of the general form

```
ans = opcode(arg-list)
```

This allows the inlining of opcodes in expressions, for instance, with the following code

```
out(moogladder
    (vco2
        (linen(p4,0.01,p3,0.1),p5),
        p5+linen(p5*4,0.01,p3,0.5),
        0.8))
```

being the equivalent of

```
k1 linen p4,0.01,p3,0.1
k2 linen p5*4,0.01,p3,0.5
a1 vco2 k1, p5
a2 moogladder a1, k2+p5, 0.8
out a2
```

in the traditional Csound syntax.

Given the extensive use of polymorphism in Csound, the mechanism of type annotation can be used to resolve certain ambiguous expressions and to select the required opcode for a desired output type. The general form of annotations in functional syntax is

```
opcode:type(arg-list)
```

In version 6.00, only opcodes with a single output are allowed in this form, as multiple outputs will require the introduction of tuple types (current under plans). However, the functional syntax can be intermingled with the traditional output-in syntax in Csound code. Note that, as Csound is not a purely functional language, there are no guarantees that functions will not have side effects, so the change in syntax does not imply any internal operation modifications.

5.3 On-the-fly Compilation

With Csound 5, recompilation of code running in an instance of the engine required interruption of performance. This came to seem restrictive, specially for performances involving live coding, where either two instances would be used (so one could be alternatively recompiled while the other was active), or a complete set of instruments was required to be supplied.

In Csound 6, we have removed this restriction. Any new instruments can be added at any point, and will be available for new insertions. The mechanism allows for replacement of existing instruments, with any running instances of these being unaffected. User-defined opcodes can also be added at any point. From the use-case point of view, we expect that software using Csound will allow on-the-fly scripting of instruments, loading and instantiation.

From inside the orchestra language, however, it is also possible to add new instruments, via two special opcodes, compileorc and compilestr. The first opcode reads orchestra code from a file, parses and compiles it. The second performs the same operations on a string.

Hosts can also send instruments as strings via bus channels to be compiled, or save them in plain text files. Full access to parsing and compilation is provided via the API. The parse tree is also exposed via the API, so it is feasible that in the future alternative languages might be implemented, ready for Csound compilation. This is yet another step towards the full separation of engine and language, which started in Csound 5.
5.4 Sample-level accuracy

Traditionally, sample-level accuracy had been achieved in Csound by running it with a ksmsp (block) size of 1. This has been always available as a global orchestra setting. Since Csound 5, user-defined opcodes can also have local ksmsps values, enabling sample-level processing. In Csound 6, this is extended to instrument definitions, which can have a per-instance block size. However, global (whole-orchestra) sample-by-sample processing of this kind is relatively inefficient (even though in some other systems this all is that available). For Csound 6, we have introduced a mechanism that allows sample-level accuracy that is completely independent of ksmsps. This is enabled by an engine option (‘sample-accurate’), but it is not on by default (for backward compatibility reasons, as it would possibly alter behaviour of older code). With this feature, we also have means of optimising multicores performance by processing in larger blocks, without loss of timing accuracy \(^1\) [14].

5.5 Realtime priority mode

Another new feature of Csound 6 is a realtime priority mode that allows performance to be uninterrupted by blocking or time-consuming operations. This mode effectively forces opcodes that access disk to do so asynchronously, and also performs all init-time code in a separate thread. In this case, new instrument instances will invoke their init-pass code to happen in a worker thread, then immediately resume executing their performance-pass code. For example, the loading of large tables and similar operations will no longer directly affect performance. Similarly, opcodes reading or writing to disk will not cause dropouts (which was liable to happen in Csound 5, esp. on disk writing). This should enhance the performance of Csound code in interrupt-driven callbacks.

5.6 The new API

We have carefully revised the low-level Csound API. Functions exposing the new functionality have been added, and others have been removed in an effort to simplify API use. In particular, access to the software bus has been simplified. Also, as noted above, with on-the-fly compilation, new means of starting and running Csound instances has been added. Csound performances can be started with no orchestra or score, instruments and events can be added at any time to it. New ways of configuring the engine have also been provided, previously only possible via string flags and arguments. A simple Python example demonstrating some of the new API functions is shown below:

```python
import csnd6
import time

cs = csnd6.csoundCreate(None)
csnd6.csoundSetOption(cs, ‘-odac’)  
csnd6.csoundStart(cs)

perf = csnd6.CsoundPerformanceThread(cs)
perf.Play()

csnd6.csoundCompileOrc(cs, '')
event_i ‘i’,1,0.1,1,1000,500
instr 1
k1 expon 1,p3,0.001
a2 oscili k1*p4,p5
event_i ‘i’,1,0.1,1,p4,rnd(p5)+500
out a2

endin ‘’)
time.sleep(5)
perf.Stop()
```

This script runs for the synthesis engine only for 5 seconds, but in interactive contexts Csound would be open for performance indefinitely, accepting input in terms of orchestra code or realtime events. Examples such as these can be run in a read-eval-print loop (REPL) provided by emacs, vim, ipython or similar environments, for live-coding with Csound, as well as from other languages (Lua, Java, Clojure, etc.). Such possibilities are not limited to performance and composition, but also allow flexible use in research and teaching.

5.7 Miscellaneous improvements

Utilities have been updated to provide cross-platform support in terms of file formats, which is byte-order and precision independent. Support for string data in the score has also been made more flexible, so that an unlimited number of strings can be passed from events to instrument instances (previously this was limited to one). There is a proposal for a new parser for the score language, but details of this are still in the planning stage.

6. NEXT STEPS

At the time of writing, we are providing a Release Candidate version of Csound 6, which is available for all users to test. This will be followed by the first full release of Csound 6 for Linux, OSX, Windows, Android and iOS. Beyond that, we expect that the infrastructure changes will now allow significant room for further incremental development of new features and improvements, and publication of the internal abstract syntax tree format will allow new user-level languages to access the Csound engine and unit generators. The new releases will be developed in conjunction with third-party developments of frontends and applications, whose functionality, it is hoped, will be greatly enhanced by Csound 6.

---

\(^1\) The effectiveness of parallelization of audio processes in general is tied to the granularity of processing, due to the overhead from spawning and joining the parallel processes. Larger granularity generally leads to greater event jitter and latency.
7. REFERENCES


