Extensions to the Csound Language: from User-Defined to Plugin Opcodes and Beyond.

Victor Lazzarini
Music Technology Laboratory
National University of Ireland, Maynooth
Victor.Lazzarini@nuim.ie

Abstract

This article describes the latest methods of extending the csound language. It discusses these methods in relation to the two currently available versions of the system, 4.23 and 5. After an introduction on basic aspects of the system, it explores the methods of extending it using facilities provided by the csound language itself, using user-defined opcodes. The mechanism of plugin opcodes and function table generation is then introduced as an external means of extending csound. Complementing this article, the fsig signal framework is discussed, focusing on its support for the development of spectral-processing opcodes.

Keywords: Computer Music, Music Processing Languages, Application Development, C / C++ Programming

1 Introduction

The csound (Vercoe 2004) music programming language is probably the most popular of the text-based audio processing systems. Together with cmusic (Moore 1990), it was one of the first modern C-language-based portable sound compilers (Pope 1993), but unlike it, it was adopted by composers and developers world-wide and it continued to develop into a formidable tool for sound synthesis, processing and computer music composition. This was probably due to the work of John Ffitch and others, who coordinated a large developer community who was ultimately responsible for the constant upgrading of the system. In addition, the work of composers and educators, such as Richard Boulanger, Dave Phillips and many others, supported the expansion of its user base, who also has been instrumental in pushing for new additions and improvements. In summary, csound can be seen as one of the best examples of music open-source software development, whose adoption has transcended a pool of expert-users, filtering into a wider music community.

The constant development of csound has been partly fuelled by the existence of a simple opcode API (Fftich 2000) (Resibois 2000), which is easy to understand, providing a good, if basic, support for unit generator addition. This was, for many years, the only direct means of extending csound for those who were not prepared to learn the inside details of the code. In addition, the only way of adding new unit generators to csound was to include them in the system source code and re-build the system, as there was no support for dynamically-loadable components (csound being from an age where these concepts had not entered mainstream software development). Since then, there were some important new developments in the language and the software in general providing extra support for extensions. These include the possibility of language extension both in terms of C/C++-language loadable modules and in csound's own programming language. Another important development has been the availability of a more complete C API (Goggins et al 2004), which can be used to instantiate and control csound from a calling process, opening the door for the separation of language and processing engine.

2 Csound versions

Currently there are two parallel versions of the so-called canonical csound distribution, csound 4.23, which is a code-freeze version from 2002 , and csound 5, a re-modelled system, still in beta stage of development. The developments mentioned in the introduction are present in csound 4.23, but have been further expanded in version 5. In this system, apart from the core opcodes, most of the unit generators are now in loadable library modules and further opcode addition should be in that format. The plugin opcode mechanism is already present in version 4.23, although some differences exist between opcode formats for the

two versions. These are mainly to do with arguments to functions and return types. There is also now a mechanism for dynamic-library function tables and an improved/expanded csound API. Other changes brought about in csound 5 are the move to the use of external libraries for soundfile, audio IO and MIDI.

Csound 4.23 is the stable version of csound, so at this moment, it would be the recommended one for general use and, especially, for new users. Most of the mechanisms of language extension and unit generator development discussed in this paper are supported by this version. For Linux users, a GNU building system-based source package is available for this version, making it simple to configure and install the program on most distributions. It is important to also note that csound 5 is fully operational, although with a number of issues still to be resolved. It indeed can be used by anyone, nevertheless we would recommend it for more experienced users. However, the user input is crucial to csound 5 development, so the more users adopting the new version, the better for its future.

3 Extending the language

As mentioned earlier, csound has mechanisms for addition of new components both by writing code in the csound language itself and by writing C/C++ language modules. This section will concentrate on csound language-based development, which takes the basic form of user-defined opcodes. Before examining these, a quick discussion of csound data types, signals and performance characteristics is offered

3.1 Data types and signals

The csound language provides three basic data types: i-, k- and a-types. The first is used for initialisation variables, which will assume only one value in performance, so once set, they will usually remain constant throughout the instrument code. The other types are used to hold scalar (k-type) and vectorial (a-type) variables. The first will hold a single value, whereas the second will hold an array of values (a vector) and internally, each value is a floating-point number, either 32- or 64-bit, depending on the version used.

A csound instrument code can use any of these variables, but opcodes might accept specific types as input and will generate data in one of those types. This implies that opcodes will execute at a certain update rate, depending on the output type (Ekman 2000). This can be at the audio sampling rate (sr), the control rate (kr) or only at initialisation time. Another important aspect is that csound instrument code effectively has a hidden processing loop, running at the control-rate and affecting (updating) only control and audio signals. An instrument will execute its code lines in that loop until it is switched off
Under this loop, audio variables, holding a block of samples equivalent to sr/kr (ksmps), will have their whole vector updated every pass of the loop:

```
instr 1    /* start of the loop */

iscl = 0.5 /* i-type, not affected by
              the loop */
asig  in  /* copies ksmps  samples from
             input buffer into asig  */
atten = asig*iscl /* scales every sample
                     of  asig with iscl */
out  atten /* copies kmsps samples from
              atten into output buffer */

endin     /* end of the loop */
```

This means that code that requires sample-by-sample processing, such as delays that are smaller than one control-period, will require setting the a-rate vector size, ksmps, to 1, making kr=sr. This will have a detrimental effect on performance, as the efficiency of csound depends a lot on the use of different control and audio rates.

3.2 User-defined opcodes

The basic method of adding unit generators in the csound language is provided by the user-defined opcode (UDO) facility, added by Istvan Varga to csound 4.22. The definition for a UDO is given using the keywords opcode and endop, in a similar fashion to instruments:

```
opcode  NewUgen  a,aki
/* defines an a-rate opcode, taking a,
   k and i-type inputs */
endop
```

The number of allowed input argument types is close to what is allowed for C-language opcodes. All p-field values are copied from the calling instrument. In addition to a-,k- and i-type arguments (and 0, meaning no inputs), which are audio, control and initialisation variables, we have: K, control-rate argument (with initialisation); plus o, p and j (optional arguments, i-type variables defaulting to 0,1 and -1). Output is permitted to be to any of a-, k- or i-type variables. Access to input and output is simplified through the use of a special pair of opcodes, `xin` and `xout`. UDOs will have one extra argument in addition to those defined in the declaration, the internal number of the a-signal vector samples `iksmps`. This sets the value of a local control rate (sr/iksmps) and

defaults to 0, in which case the `iksmps` value is taken from the caller instrument or opcode.

The possibility of a different a-signal vector size (and different control rates) is an important aspect of UDOs. This enables users to write code that requires the control rate to be the same as audio rate, without actually having to alter the global values for these parameters, thus improving efficiency. An opcode is also provided for setting the iksmps value to any given constant:

```
setksmps 1 /* sets a-signal vector to 1,
            making kr=sr */
```

The only caveat is that when the local ksmps value differs from the global setting, UDOs are not allowed to use global a-rate operations (global variable access, etc.). The example below implements a simple feedforward filter, as an example of UDO use:

```
#define LowPass 0
#define HighPass 1

opcode  NewFilter  a,aki

 setksmps  1     /* kr = sr */
 asig,kcoef,itype  xin
 adel init 0

 if itype == HighPass then
  kcoef = -kcoef
 endif

 afil  =  asig + kcoef*adel
 adel = asig  /* 1-sample delay,
              only because kr = sr */
     xout   afil

endop
```

Another very important aspect of UDOs is that recursion is possible and only limited to available memory. This allows, for instance, the implementation of recursive filterbanks, both serial or parallel, and similar operations that involve the spawning of unit generators. The UDO facility has added great flexibility to the csound language, enabling the fast development of musical signal processing operations. In fact, an on-line UDO database has been made available by Steven Yin, holding many interesting new operations and utilities implemented using this facility (www.csounds.com/udo). This possibly will form the foundation for a complete csound-language-based opcode library.

3.3 Adding external components

Csound can be extended in variety of ways by modifying its source code and/or adding elements to it. This is something that might require more than a passing acquaintance with its workings, as a rebuild of the software from its complete source code. However, the addition of unit generators and function tables is generally the most common type of extension to the system. So, to facilitate this, csound offers a simple opcode development API, from which new dynamically-loadable ('plugin') unit generators can be built. In addition, csound 5 also offers a similar mechanism for function tables. Opcodes can be written in the C or C++ language. In the latter, the opcode is written as a class derived from a template ('pseudo-virtual') base class `OpcodeBase`, whereas in the former, we normally supply a C module according to a basic description. The following sections will describe the process of adding an opcode in the C language. An alternative C++ class implementation would employ a similar method.

3.3.1 Plugin opcodes

C-language opcodes normally obey a few basic rules and their development require very little in terms of knowledge of the actual processes involved in csound. Plugin opcodes will have to provide three main programming components: a data structure to hold the opcode internal data, an initialising function or method, and a processing function or method. From an object-oriented perspective, all we need is a simple class, with its members, constructor and perform methods. Once these elements are supplied, all we need to do is to add a line telling csound what type of opcode it is, whether it is an i-, k- or a-rate based unit generator and what arguments it takes.

The data structure will be organised in the following fashion:

1. The OPDS data structure, holding the common components of all opcodes.
2. The output pointers (one MYFLT pointer for each output)
3. The input pointers (as above)
4. Any other internal dataspace member.

The csound opcode API is defined by csdl.h, which should be included at the top of the source file. The example below shows the data structure for same filter implemented in previous sections:

```
#include "csdl.h"

typedef struct  _newflt {
OPDS  h;
MYFLT *outsig;/* output pointer  */
MYFLT *insig,*kcoef,*itype;/* input
                            pointers */
MYFLT  delay;  /* internal variable,
```

```
                the 1-sample delay */
int    mode;   /* filter mode */
} newfilter;
```

The initialisation function is only there to initialise any data, such as the 1-sample delay, or allocate memory, if needed. The new plugin opcode model in csound5 expects both the initialisation function and the perform function to return an int value, either OK or NOTOK. In addition, both methods now take a two arguments: pointers to the csound environment and the opcode dataspace. In version 4.23 the opcode function will only take the pointer to the opcode dataspace as argument. The following example shows an initialisation function in csound 5 (all following examples are also targeted at that version):

```
int newfilter_init(ENVIRON *csound,
                   newfilter *p){
p->delay = (MYFLT) 0;
p->mode = (int) *p->itype;
return OK;
}
```

The processing function implementation will depend on the type of opcode that is being created. For audio rate opcodes, because it will be generating audio signal vectors, it will require an internal loop to process the vector samples. This is not necessary with k-rate opcodes, as we are dealing with scalar inputs and outputs, so the function has to process only one sample at a time. This means that, effectively, all processing functions are called every control period. The filter opcode is an audio-rate unit generator, so it will include the internal loop.

```
int newfilter_process(ENVIRON *csound,
                      newfilter *p){
int i;
/* signals in, out */
MYFLT *in = p->insig;
MYFLT *out = p->outsig;
/* control input */
MYFLT  coef = *p->kcoef;
/* 1-sample  delay  */
MYFLT delay = *p->delay;
MYFLT temp;

if(p->mode)coef = -coef;

/* processing loop    */
for(i=0; i < ksmps; i++){
    temp = in[i];
    out[i] = in[i] + delay*coef ;
    delay = temp;
}
/* keep delayed sample for next time */
*p->delay = delay;

return OK;
}
```

To complete the source code, we fill an opcode registration structure OENTRY array called localops (static), followed by the LINKAGE macro:

```
static OENTRY localops[] = {
{ "newfilter", S(newfilter),  5,  "a",
"aki",  (SUBR)newfilter_init,  NULL,
(SUBR)newfilter_process }
};

LINKAGE
```

The OENTRY structure defines the details of the new opcode:

1. the opcode name (a string without any spaces).
2. the size of the opcode dataspace, set using the macro S(struct_name), in most cases; otherwise this is a code indicating that the opcode will have more than one implementation, depending on the type of input arguments.
3. An int code defining when the opcode is active: 1 is for i-time, 2 is for k-rate and 4 is for a-rate. The actual value is a combination of one or more of those. The value of 5 means active at i-time (1) and a-rate (4). This means that the opcode has an init function and an a-rate processing function.
4. String definition the output type(s): a, k, s (either a or k), i, m (multiple output arguments), w or f (spectral signals).
5. Same as above, for input types: a, k, s, i, w, f, o (optional i-rate, default to 0), p (opt, default to 1), q (opt, 10), v(opt, 0.5), j(opt, –1), h(opt, 127), y (multiple inputs, a-type), z (multiple inputs, k-type), Z (multiple inputs, alternating k- and a-types), m (multiple inputs, i-type), M (multiple inputs, any type) and n (multiple inputs, odd number of inputs, i-type).
6. I-time function (init), cast to (SUBR).
7. K-rate function.
8. A-rate function.

The LINKAGE macro defines some functions needed for the dynamic loading of the opcode. This macro is present in version 5 csdl.h, but not in 4.23 (in which case the functions need to be added manually):

```
#define LINKAGE long opcode_size(void) \
{ return sizeof(localops);}   \
OENTRY *opcode_init(ENVIRON *xx)   \
{ return localops;}           \
```

The plugin opcode is build as a dynamic module, and similar code can be used both with csound versions 4.23 or 5:

```
gcc -02 -c opsrc.c -o opcode.o
ld -E --shared opcode.o –o opcode.so
```

However, due to differences in the interface, the binaries are not compatible, so they will need to built specificially for one of the two versions.Another difference is that csound 5 will load automatically all opcodes in the directory set with the environment variable OPCODEDIR, whereas version 4.23 needs the flag –opcode-lib=*myopcode.so* for loading a specific module.

3.3.2 Plugin function tables

A new type of dynamic module, which has been introduced in csound 5 is the dynamic function table generator (GEN). Similarly to opcodes, function table GENs were previously only included statically with the rest of the source code. It is possible now to provide them as dynamic loadable modules. This is a very recent feature, introduced by John Ffitch at the end of 2004, so it has not been extensively tested. The principle is similar to plugin opcodes, but the implementation is simpler. It is only necessary to provide the GEN routine that the function table implements. The example below shows the test function table, written by John Ffitch, implementing a hyperbolic tangent table:

```
#include "csdl.h"
#include <math.h>

void tanhtable(ENVIRON *csound,
       FUNC *ftp, FGDATA *ff,)
{
/* the function table */
MYFLT fp = ftp->ftable;
/* f-statement p5, the range */
MYFLT range = ff->e.p[5];
/* step is range/tablesize */
double step = (double)
              range/(ff->e.p[3]);
int i;
double x;
  /* table-filling loop   */
   for(i=0, x=FL(0.0); i<ff->e.p[3];
       i++,x+=step)
        *fp++ = (MYFLT)tanh(x);
}
```

The GEN function takes three arguments, the csound environment dataspace, a function table pointer and a gen info data pointer. The former holds the actual table, an array of MYFLTs, whereas the latter holds all the information regarding the table, e.g. its size and creation arguments. The `FGDATA` member `e` will hold a numeric array (`p`) with all p-field data passed from the score f-statement (or `ftgen` opcode).

```
static NGFENS localfgens[] = {
   { "tanh", (void(*)(void))tanhtable},
   { NULL, NULL}
};
```

The structure NFGENS holds details on the function table GENs, in the same way as OENTRY holds opcode information. It contains a string name and a pointer to the GEN function. The localfgens array is initialised with these details and terminated with NULL data. Dynamic GENs are numbered according to their loading order, starting from GEN 44 (there are 43 ‘internal’ GENs in csound 5).

```
#define S sizeof
static OENTRY *localops = NULL;
FLINKAGE
```

Since opcodes and function table GENs reside in the same directory and are loaded at the same time, setting the *localops array to NULL, will avoid confusion as to what is being loaded. The FLINKAGE macro works in the same fashion as LINKAGE.

4 Spectral signals

As discussed above, Csound provides data types for control and audio, which are all time-domain signals. For spectral domain processing, there are two separate signal types, ‘wsig’ and ‘fsig’. The former is a signal type introduced by Barry Vercoe to hold a special, non-standard, type of logarithmic frequency analysis data and is used with a few opcodes originally provided for manipulating this data type. The latter is a self-describing data type designed by Richard Dobson to provide a framework for spectral processing, in what is called streaming phase vocoder processes (to differentiate it from the original csound phase vocoder opcodes). Opcodes for converting between time-domain audio signals and fsigs, as well as a few processing opcodes, were provided as part of the original framework by Dobson. In addition, support for a self-describing, portable, spectral file format PVOCEX (Dobson 2002) has been added to csound, into the analysis utility program pvanal and with a file reader opcode. A library of processing opcodes, plus a spectral GEN, has been added to csound by this author. This section will explore the fsig framework, in relation to opcode development.

Fsig is a self-describing csound data type which will hold frames of DFT-based spectral analysis

data. Each frame will contain the positive side of the spectrum, from 0 Hz to the Nyquist (inclusive). The framework was designed to support different spectral formats, but at the moment, only an amplitude-frequency format is supported, which will hold pairs of floating-point numbers with the amplitude and frequency (in Hz) data for each DFT analysis channel (bin). This is probably the most musically meaningful of the DFT-based output formats and is generated by Phase Vocoder (PV) analysis. The fsig data type is defined by the following C structure:

```
typedef struct pvsdat {
/* framesize-2, DFT length */
long N;
/* number of frame overlaps */
long overlap;
/* window size */
long winsize;
/* window type: hamming/hanning */
int  wintype;
/* format: cur. fixed to AMP:FREQ */
long format;
/* frame counter   */
unsigned long   framecount;
/* spectral sample is a 32-bit float */
AUXCH frame;
} PVSDAT;
```

The structure holds all the necessary data to describe the signal type: the DFT size (N), which will determine the number of analysis channels (N/2 + 1) and the framesize; the number of overlaps, or decimation, which will determine analysis hopsize (N/overlaps); the size of the analysis window, generally the same as N; the window type, currently supporting PVS_WIN_HAMMING or PVS_WIN_HANN; the data format, currently only PVS_AMP_FREQ; a frame counter, for keeping track of processed frames; and finally the AUXCH structure which will hold the actual array of floats with the spectral data. The AUXCH structure and associated functions are provided by csound as a mechanism for dynamic memory allocation and are used whenever such operation is required. A number of other utility functions are provided by the csound opcode API (in csdl.h), for operations such as loading, reading and writing files, accessing function tables, handling string arguments, etc.. Two of these are used in the code below to provide simple error notification and handling (`initerror()` and `perferror()`).

A number of implementation differences exist between spectral and time-domain processing opcodes. The main one is that new output is only produced if a new input frame is ready to be processed. Because of this implementation detail, the processing function of a streaming PV opcode is actually registered as a k-rate routine. In addition, opcodes allocate space for their fsig frame outputs, unlike ordinary opcodes, which simply take floating-point buffers as input and output. The fsig dataspace is externally allocated, in similar fashion to audio-rate vectors and control-rate scalars; however the DFT frame allocation is done by the opcode generating the signal. With that in mind, and observing that type of data we are processing is frequency-domain, we can implement a spectral unit generator as an ordinary (k-rate) opcode. The following example is a frequency-domain version of the simple filter implemented in the previous sections:

```
#include "csdl.h"
#include "pstream.h" /* fsig definitions
*/

typedef struct _pvsnewfilter {
OPDS    h;
/* output fsig, its frame needs to be
   allocated */
PVSDAT  *fout;
PVSDAT  *fin;   /* input fsig */
/* other opcode args  */
MYFLT   *coef, *itype;
MYFLT    mode;  /* filter type */
unsigned long lastframe;
} pvsnewfilter;

int pvsnewfilter_init(ENVIRON *csound,
                      pvsnewfilter *p)
{
long N = p->fin->N;
p->mode = (int) *p->itype;
/* this allocates an AUXCH struct, if
   non-existing */
if(p->fout->frame.auxp==NULL)
  auxalloc((N+2)*sizeof(float),
           &p->fout->frame);
/* output fsig description */
p->fout->N =  N;
p->fout->overlap = p->fin->overlap;
p->fout->winsize = p->fin->winsize;
p->fout->wintype = p->fin->wintype;
p->fout->format = p->fin->format;
p->fout->framecount = 1;
p->lastframe = 0;

/* check format */
if (!(p->fout->format==PVS_AMP_FREQ ||
      p->fout>format==PVS_AMP_PHASE))
return  initerror("wrong format\n");
/* initerror is a utility csound
   function */

return OK;
}
```

The opcode dataspace contains pointers to the output and input fsig, as well as the k-rate coefficient and the internal variable that holds the filter mode. The init function has to allocate space for the output fsig DFT frame, using the csound

opcode API function `auxalloc()`, checking first if it is not there already.

```
int pvsnewfilter_process(ENVIRON *csound,
                         pvsnewfilter p)
{
 long i,N = p->fout->N;
 MYFLT cosw, tpon;
 MYFLT coef = *p->kcoef;
 float *fin = (float *)
           p->fin >frame.auxp;
float *fout = (float *)
           p->fout->frame.auxp;

if(fout==NULL)
 return perferror("not initialised\n");
/* perferror is a utility csound
   function */

if(mode) coef = -coef;
/* if a new input frame is ready  */
if(p->lastframe <
   p->fin->framecount) {
 /* process the input, filtering */
  pon = pi/N; /* pi is global*/
  for(i=0;i < N+2;i+=2) {
    cosw = cos(i*pon);
    /* amps */
    fout[i] =  fin[i] *
       sqrt(1+coef*coef+2*coef*cosw);
    /* freqs: unchanged */
    fout[i+1] = fin[i+1];
      }
  /* update the framecount  */
   p->fout->framecount =
   p->lastframe = p->fin->framecount;
    }
return OK;
}
```

The processing function keeps track of the frame count and only processes the input, generating a new output frame, if a new input is available. The framecount is generated by the analysis opcode and is passed from one processing opcode to the next in the chain. As mentioned before, the processing function is called every control-period, but it is independent of it, only performing when needed. The only caveat is that the fsig framework requires the control period in samples (ksmps) to be smaller or equal to the analysis hopsize. Finally, the localops OENTRY structure for this opcode will look like this:

```
static OENTRY localops[] = {
 {"pvsnewfilter", S(pvsnewfilter), 3,
  "f", "fkp", (SUBR)pvsnewfilter_init,
  (SUBR)pvsnewfilter_process}
};
```

From the above, it is clear to see that the new opcode is called pvsnewfilter and its implementation is made of i-time and k-rate functions. It takes fsig, ksig and one optional i-time arguments and it outputs fsig data.

5 Conclusion

Csound is regarded as one of the most complete synthesis and processing languages in terms of its unit generator collection. The introduction of UDOs, plugin opcode and function table mechanisms, as well as a self-describing spectral signal framework, has opened the way for further expansion of the language. These methods provide simpler and quicker ways for customisation. In fact, one of the goals of csound 5 is to enhance the possibilities of extension and integration of the language/processing engine into other systems. It is therefore expected that the developments discussed in this article are but only the start of a new phase in the evolution of csound.

6 References

Richard Dobson. 2000. PVOCEX: File format for Phase Vocoder data, based on WAVE FORMAT EXTENSIBLE. . http://www.bath.ac.uk/~masrwd/pvocex/pvocex.html.

Rasmus Ekman. 2000. Csound Control Flow. http://www.csounds.com/internals/index.html.

John Ffitch. Extending Csound. In R. Boulanger, editor, *The Csound Book*, Cambridge, Mass., MIT Press.

Michael Goggins et Al. 2004. *The Csound API*. http://www.csounds.com/developers/html/csound_8h.html

F Richard Moore. 1990. *Elements of Computer Music*, Englewood Cliffs, NJ: Prentice-Hall, 1990.

Stephen T Pope. 1993. Machine Tongues XV: Three Packages for Software Sound Synthesis. *Computer Music Journal 17 (2)*.

Mark Resibois. 2000. Adding New Unit Generators to Csound. In R. Boulanger, editor, *The Csound Book*, Cambridge, Mass., MIT Press.

Barry Vercoe. 2004. *The Csound and VSTCsound Reference Manual*, http://cvs.sourceforge.net/viewcvs.py/csound/csound5/csound.pdf.