

Extensions to the Csound Language

from user-defined to plugin opcodes and beyond

Victor Lazzarini
Music Technology Lab
NUI Maynooth
Ireland

Introduction

- *Csound* is one of the most popular of the modern computer music languages
- It is available for a number of *different platforms*
- It has the most *complete set* of unit generators (ugens, which in csound take the form of *opcodes*)
- It has a *simple* opcode structure, which makes it easy to extend
- Now it provides a mechanism for loading *plugin opcodes* from *dynamic library objects*
- It also provides an API, which allows for great *flexibility and control* over its operation, as well as increased support for *extension*

Csound versions

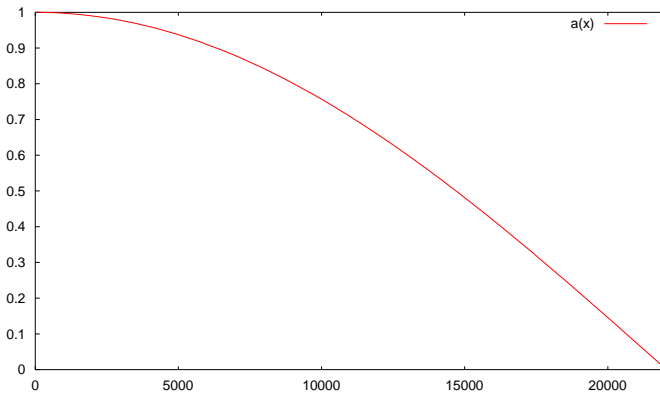
- The latest csound is available in two versions:
 - Csound 4.23, a stable, code-freeze version from 2002
 - available as an easy-to-install *GNU build system package*
 - provides an API and a library, as well as the support for plugin opcodes
 - a *reliable* system
 - Csound 5, a pre-release, 'new generation' version
 - available from sourceforge CVS & snapshots
 - built with *scons*
 - includes a re-structuring of the code, plus an *expanded API*
 - uses external libraries for soundfile (*sndfile lib*), MIDI (*portmidi*) and RT audio (*portaudio*, and also an *alsa RT audio plugin*)
 - constantly *evolving* and not completely tested
 - latest code provides a *very usable* system in GNU/Linux

Data Types & Signals

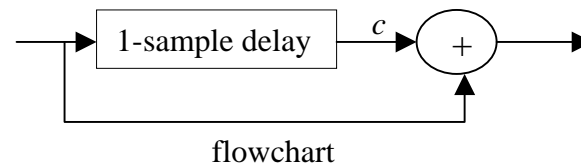
- Three basic data types:
 - **i variables:** initialisation only (*one value* in performance)
 - **k variables:** control-rate signals, these are scalars, holding a *single sample* during performance
 - **a variables:** audio-rate signals, these are *vectors*, holding a block of *ksmps* samples.
- The 'performance loop'
 - csound instrument code has an implied performance loop, which operates at the control rate $kr = sr/ksmps$
 - During performance, *k-variables* are updated every $1/kr$ seconds and *a-variables* have their whole vector updated at that rate.

The Example Opcodes

- In this presentation, we will be looking at a very simple unit generator example in three forms:
 - UDO
 - plugin opcode (time-domain)
 - plugin opcode (spectral-domain)
- The ugen will implement a 1st order feedforward filter, described by $y(n) = x(n) + cx(n-1]$



amplitude spectrum (with $c=1$)



$$amp(f) = \sqrt{1 + c^2 + 2c \cos\left(\frac{\pi f}{SR}\right)}$$

amplitude response

User-Defined Opcodes

- UDOs provide a way of extending the system using the csound language itself.
- They behave exactly like other opcodes; any number of instances can be created.
- Similar definition to an instrument code, plus arguments:

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

- *Inputs* and *outputs* are accessed using xin/xout:

```
as,ks,iv xin
      xout as
```

- A local value for *ksmps* can be set:

```
setksmps 1
```

UDO example

Here's a simple example: a simple feedforward filter

```
#define LowPass 0
#define HighPass 1

opcode NewFilter a,aki
setksmps 1 /* kr = sr */
asig,kcoef,ittype 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
```

use: ar **NewFilter** asig, kcoef, imode

Plugin Opcodes

- Csound provides a mechanism for loading plugin opcodes, as dynamic libraries written in C or C++.
- Both csound 4.23 and 5 support this feature, although the C code is not completely compatible between versions.
- Csound plugin opcodes have the following code format:
 - A C-structure defining their *dataspace*
 - Initialisation (i-time), control-rate and/or audio-rate functions implementing the *signal processing algorithm*.
 - An OENTRY array used for *registering* the opcode with csound
 - The LINKAGE *macro* (cs 5) or library entry-point functions (cs 4.23)

Example: dataspace

- The opcode dataspace has the following members (in that order):
 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

Example:

```
#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;
```

Example: init function

- The init function is called every time there is an init-pass in the orchestra (at instrument start time, for instance):

```
/* in cs 4.23, this function returns void and does not take the ENVIRON* arg */
int newfilter_init(ENVIRON *csound, newfilter *p){
p->delay = (MYFLT) 0;
p->mode = (int) *p->itype;
return OK; /* cs 5 only, cs 4.23 does not return a value */
}
```

Here we do all the necessary initialisation, including obtaining the i-time argument value from the input, since we only need to do it once.

Example: process function

- An audio-rate processing function produces a vector of samples as its output and, here, also takes a sample block as input. The size of the vector is taken from *ksmps*.

```
/* in cs 4.23 this takes one argument only (the dataspace) and returns void */
int newfilter_process(ENVIRON *csound, newfilter *p){
int i, n=csound->ksmps;                               /* in cs 4.23 ksmps is global */
MYFLT *in = p->insig, *out = p->outsig;                /* audio signals in, out */
MYFLT coef = *p->kcoef;                                /* control signal input */
MYFLT delay = p->delay, temp;                          /* 1-sample delay and temp vars */
if(p->mode)coef = -coef;
for(i=0; i < n; i++){                                  /* processing loop */
    temp = in[i];
    out[i] = in[i] + delay*coef ;
    delay = temp; }
p->delay = delay;                                     /* keep delayed sample for next time */
return OK;                                           /* cs 5 only, cs 4.23 does not return a value */
}
```

Example: opcode registration

- An OENTRY array entry is used to *register* the opcode:

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

containing: 1. *opcode name* ; 2. *size of dataspace* (in most cases); 3. *code* determining *what processing* it does: 1=init, 2=control, 4=audio, in any combination; 4. *output types*; 5. *input types*; 6. address of *init function*; 7. address of *control function*; 8. address of *audio function*.

- Either the LINKAGE macro (cs 5) or the entry point functions are used (cs 4.23) (these are always the same):

```
long opcode_size(void){ return sizeof(localops);}  
OENTRY *opcode_init(GLOBALS *xx){  
  pcglob = xx;  
  return localops; }  
}
```

Building/Using Plugin Opcodes

- Plugin opcodes can be built with the following commands:

```
# this is cs 5 and we are in the top-level dir
# cs 4.23 does not need -I./H if we are in the sources directory
gcc -O2 -c opsrc.c -I./H -o opcode.o
ld -E --shared opcode.o -o opcode.so
```

- Csound 5 will load automatically all opcodes in `OPCODE_DIR`
- Csound 4.23 can use `--opcode-lib=./opcode.so`

Plugin Function Tables

- Dynamic function table GENs can also be loaded by csound 5
- All we need to do is to define a function to implement the GEN routine:

```
#include "csdl.h"
#include <math.h>
/* hyperbolic tangent GEN by John ffitch */
void tanhtable(ENVIRON *csound, FUNC *ftp, FGDATA *ff)
{
MYFLT fp = ftp->ftable; /* the function table */
MYFLT range = ff->e.p[5]; /* f-statement p5, the range */
double step = (double)range/(ff->e.p[3]); /* step is range/tablesiz */
int i;
double x;
for(i=0, x=FL(0.0); i<ff->e.p[3]; i++,x+=step) /* table-filling loop */
    *fp++ = (MYFLT)tanh(x);
}
```

Registering the plugin GEN

- A similar method to plugin opcodes is used to register the newly-loaded GEN, a null-terminated NFGENS array:

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

- Loadable GENs are numbered by their loading order, starting from 44.
- The FLINKAGE macro is used to provide the entry point to the dynamic library module:

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

Spectral Processing

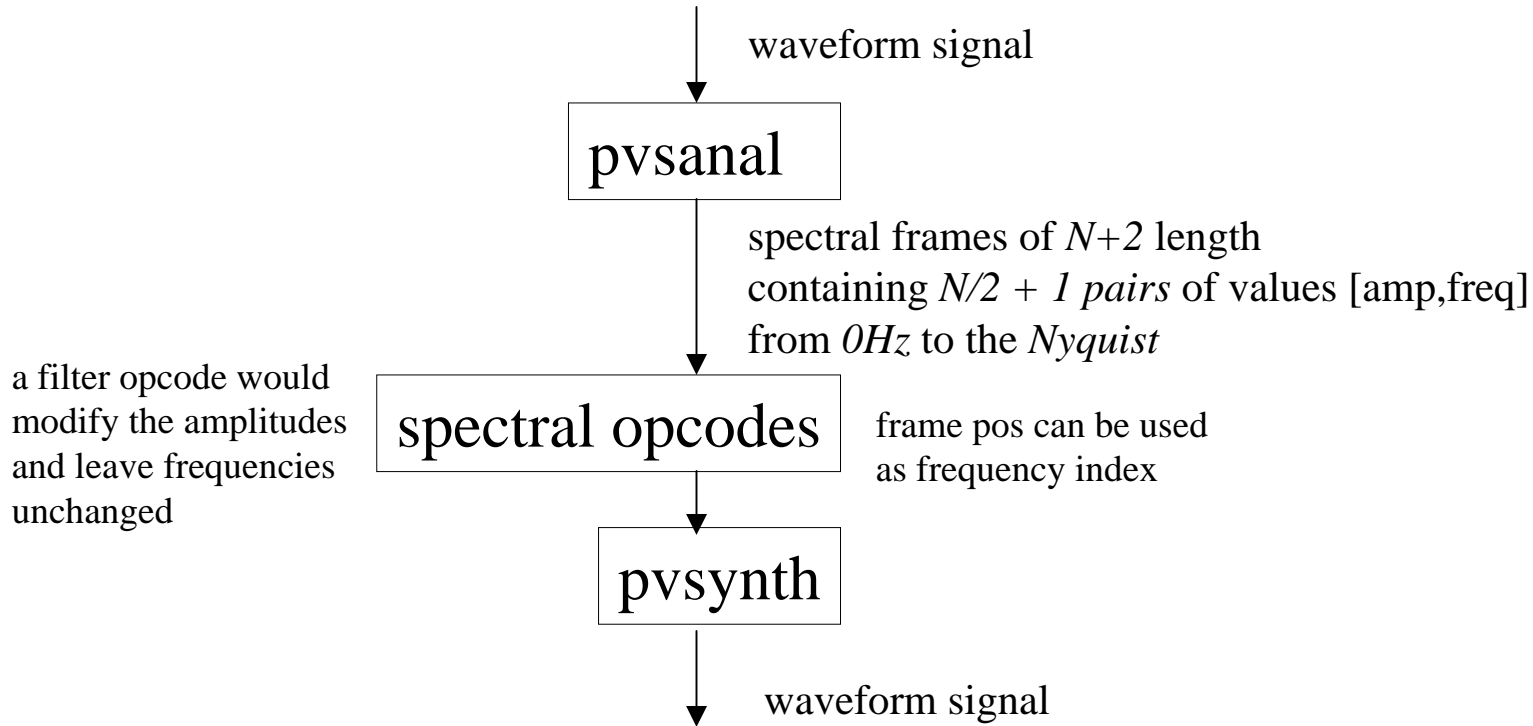
- Csound provides a framework for spectral processing, based on a fourth data type, fsig.
- These define a self-describing frequency-domain data type, described by following C data structure:

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

Spectral Opcodes

- Spectral analysis opcodes, such as `pvsanal`, will output frames of `fftsize+2` values every *hopsize* input samples.
- These frames will contain pairs of `amp,freq` values for each DFT bin from 0Hz to the Nyquist frequency, inclusive.
- Spectral-processing opcodes will take these `fsigs`, process and output them 'on-demand', as new frames are ready to be processed.
- In doing so, they will actually be implemented as `k`-rate processing functions, since they will not need to produce data at the audio rate.
- The `fsig` framework requires that the `hopsize` between analysis frames be $\geq ksmpls$.

Spectral Opcodes: flowchart



Example: dataspace

- The dataspace, as before contains the outputs and inputs, which, for fsigs, are PVSDAT object pointers

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

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

- The important thing to remember is that the PVSDAT structure contains the frame data array, which will need to be allocated by the opcode generating it.

Example: init function

- Initialise variables, allocate output frame memory

```
/* cs 4.23 only takes one argument (dataspace) and returns void */
int pvsnewfilter_init(ENVIRON *csound, pvsnewfilter *p){
    long N = p->fin->N;
    p->mode = (int) *p->itype;
    if(p->fout->frame.auxp==NULL)/* this allocates an AUXCH struct, if non-existing */
        /* in cs 4.23 we use auxalloc((N+2)*sizeof(float), &p->fout->frame) */
        csound->AuxAlloc(csound, (N+2)*sizeof(float), &p->fout->frame);
    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;
    if (!(p->fout->format==PVS_AMP_FREQ || p->fout->format==PVS_AMP_PHASE))
        /* in cs 4.23 we use initerror(...) */
        return csound->InitError(csound, "wrong format\n"); /* check format */
    return OK; /* cs 5 only */
}
```

Example: processing function

- Here we only process a frame when a new one is available

```
int pvsnewfilter_process(ENVIRON *csound, pvsnewfilter *p){
long i,N = p->fout->N;
MYFLT cosw, tpon, coef = *p->coef;
float *fin = (float *) p->fin->frame.auxp, *fout = (float *) p->fout->frame.auxp;
/* this just perferror(...) in cs 4.23 */
if(fout==NULL) return csound->PerfError(csound, "not initialised\n");
if(p->mode) coef = -coef;
if(p->lastframe < p->fin->framecount) { /* if a new input frame is ready */
    pon = (MYFLT) PI/N; /* pi is global*/
    for(i=0;i < N+2;i+=2) { /* process the input, filtering */
        cosw = cos(i*pon);
        fout[i] = fin[i] * sqrt(1+coef*coef+2*coef*cosw); /* amps: scale */
        fout[i+1] = fin[i+1]; /* freqs: unchanged */
    }
    /* update the framecount */
    p->fout->framecount = p->lastframe = p->fin->framecount;
}
return OK;
}
```

Example: opcode registration

- Since we are effectively doing k-rate processing, we register the process function as a control signal function.

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

- and add the LINKAGE macro etc. to the code.

Conclusion

- Csound is regarded as one of the most *complete* computer music languages
- UDOs, plugin opcode and function table mechanisms, as well as a self-describing spectral signal framework, have opened the way for further *expansion* of the language
- These methods provide simple and quick ways for *customisation*
- These developments are only the start of a new phase in the *evolution* of Csound

Updated Text

- Due to the recent developments in csound 5, parts of the code examples shown in the printed proceedings need updating.

The updated examples, as shown in this presentation can be obtained on-line at the following sites:

- NUIM Music Technology Lab:
<http://www.nuim.ie/academic/music/musictec>
- Csound site:
<http://csounds.com>

as well as in the updated electronic proceedings