Developing User-defined and Plugin Opcodes

Csound is possibly one of the most easily extensible of all modern music programming languages. The addition of unit generators (opcodes) and function tables is generally the most common type of extension to the language. This is possible through two basic mechanisms: user-defined opcodes (UDOs), written in the Csound language itself and pre-compiled/binary opcodes, written in C or C++. To facilitate the latter case, Csound offers a simple opcode development API, from which dynamically-loadable, or 'plugin' unit generators can be built. A similar mechanism for function tables is also available.

UDOs, as mentioned above, are written using the facilities of the Csound language and they are generally put at the top of an orchestra, or sometimes included in it with the use of the `#include` macro. Their main advantage consists in the simplicity of coding, which allows users without C or C++ programming skills to implement their own processing algorithms. Another advantage for Csound programmers is the possibility of recursion, previously inexistent in the language, which allows very compact code for certain processes, such as filter and oscillator banks.

For plugin opcodes, we can use either the C++ or the C languages. C++ opcodes are written as classes derived from a template ('pseudo-virtual') base class `OpcodeBase`. In the case of C opcodes, we normally supply a module according to a basic description. The sections on plugin opcodes will use the C language. For those interested in object-oriented programming, alternative C++ class implementations for the examples discussed in this text can be extrapolated from the original C code.

1 Csound data types and signals

The Csound language provides four basic data types: i-, k-, a- and f-types (there is also a fifth type, w, which will not be discussed here). These are used to pass the data between opcodes, each opcode input or output parameter relating to one of these types. The Csound i-type variable is used for initialisation variables, which will assume only one value in performance. Once set, they will remain constant throughout the instrument or UDO code, unless there is a reinitialisation pass. In a plugin opcode, parameters that receive i-type variables are set inside the initialisation part of the code, because they will not change during processing.

The other types are used to hold scalar (k-type) , vectorial (a-type) and spectral-frame (f) signal variables. These will change in performance, so parameters assigned to these variables are set and modified in the opcode processing function. Scalars will hold a single value, whereas vectors hold an array of values (a vector). These values are floating-point numbers, either 32- or 64-bit, depending on the executable version used, defined in C/C++ as a custom MYFLT type. Spectral frame variables are PVSDAT data structures, which contain a 32-bit float array and other data relating to frequency-domain signals.

Plugin opcodes will use pointers to input and output parameters to read and write their input/output. The Csound engine will take care of allocating the memory used for its variables, so the opcodes only need to manipulate the pointers to the addresses of

these variables. An exception to this occurs with f-type variables, because the spectral data array that they contain will have to be allocated by an opcode that generates this type of signal. However, the PVSDAT structures themselves, which contain such arrays, are allocated by Csound as any other variable.

A Csound instrument code can use any of these variables, but opcodes will have to accept specific types as input and will generate data in one of those types. Certain opcodes, known as polymorphic opcodes, will be able to cope with more than one type for a specific parameter (input or output). This generally implies that more than one version of the opcode will have to be implemented, which will be called depending on the parameter types used.

2 Prelude: using the Csound language

As a prelude to the study of C-language plugin opcodes, we will have a look at UDOs, which provide the basic method of adding unit generators in the Csound language. The definition for a UDO uses 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
```

2.1 Recursion

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 and parallel, and similar operations that involve the spawning of unit generators. The basic way to implement recursion is to use an i-time counter which will keep calling the opcode until it has reached the required recursion depth. Then we can jump to the processing part of the opcode, which can then feed the input of the recursive calls to the opcode.

As an example, consider the case of a channel vocoder design: it employs a number of parallel filters to extract the spectral envelope of a sound and then impose it on another, an excitation source. Each analysis band is composed of two filters, one for analysis and the other for synthesis and the extracted sound level is imposed using a balance opcode. The outputs of each band are mixed to form the input and as the filters are connected in parallel, the same input feeds all bands. All we need is to design one vocoder band and then call the opcode recursively to form as many bands as requested. We will use the opcode i-time counter to adjust the band centre frequencies and bandwidths.

Here's the fully commented code:

```
/* opcode takes two signals, three controls
   and two i-time arguments, initialised to 1
   the first i-time argument, the number of
   bands, is set by the user; the second is
   a counter used to index the recursion call
*/
opcode Vocoder, a, aakkkpp

as1,as2,kmin,kmax,kq,ibnd,icnt  xin

/* keep calling the opcode recursively
   until the count is reached (starting from 1) */
if (icnt >= ibnd) goto bank
abnd   Vocoder as1,as2,kmin,kmax,kq,ibnd,icnt+1

/* once all calls are made, jump to processing,
   which will happen from the higher bands down
```

```
*/
bank:
/* centre frequencies are evenly (constant-Q) spaced */
kfreq = kmin*(kmax/kmin)^((icnt-1)/(ibnd-1))
kbw = kfreq/kq
/* 4th order butterworths: analysis */
an  butterbp  as2, kfreq, kbw
an  butterbp  an, kfreq, kbw
/* 4th order butterworths: synthesis */
as  butterbp  as1, kfreq, kbw
as  butterbp  as, kfreq, kbw
/* match the synthesis output to the analysis */
ao balance as, an
/* add the band output to the other bands and output it */
amix = ao + abnd
     xout amix

endop
```

The recursion code has two important sections: the recursive calls at the top, regulated by the if ... goto statement and the processing code resulting in the signal being output. For parallel arrangements such as this one, there is a need to mix the current output of each band to the output of the previous recursive call. Serial connections do not need to do this, but they need to insert their output into the next recursive call (see the Csound manual for some simple examples for this type of connections).

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 Yi, 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 Plugin opcodes

In cases where UDOs are either to cumbersome or complicated to write or efficiency of computation is a main factor, plugin opcodes offer a good mechanism for language extension. Originally, Csound opcodes could only be added to the system as statically-linked code. This required that the user recompiled the whole Csound code with the added C module. The introduction of a dynamic-loading mechanism has provided a simpler way for opcode addition, which only requires the C code to be compiled and built as a shared, dynamic library. These are known in Csound parlance as *plugin opcodes* and the following sections are dedicated to their development process.

3.1 Anatomy of an opcode

The C code for a Csound opcode has three main programming components: a data structure to hold the internal data, an initialising function and a processing function. From an object-oriented perspective, an opcode is a simple class, with its attributes, constructor and perform methods. The data structure will hold the attributes of the class: input/output parameters and internal variables (such as delays, coefficients, counters, indices etc.), which make up its dataspace.

The constructor method is the initialising function, which sets some attributes to certain values, allocates memory (if necessary) and anything that is need for an opcode to be ready for use. This method is called by the Csound engine when an instrument with its opcodes is allocated in memory, just before performance, or when a reinitialisation is required.

Performance is implemented by the processing function, or perform method, which is called when new output is to be generated. This happens at every control period, or *ksmps* samples. This implies that signals are generated at two different rates: the control rate, kr, and the audio rate, sr, which is kr *x* ksmps samples/sec. What is actually generated by the opcode, and how its perform method is implemented, will depend on its input and output Csound language data types.

3.2 Opcoding basics

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 the three main programming components outlined above: a data structure plus the initialisation and processing functions. 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 a simple data structure for an opcode with one output and three inputs, plus a couple of private internal variables:

```
#include "csdl.h"

typedef struct  _newopc {

OPDS  h;
MYFLT *out;/* output pointer  */
MYFLT *in1,*in2,*in3; /* input pointers */
MYFLT  var1;  /* internal variables */
MYFLT  var2;

} newopc;
```

The initialisation function is only there to initialise any data, such as the internal variables, or allocate memory, if needed. The plugin opcode model in Csound 5 expects both the initialisation function and the perform function to return an int value, either OK or NOTOK. Both methods take two arguments: pointers to the Csound environment and the opcode dataspace.. The following example shows an example initialisation function. It initialises one of the variables to 0 and the other to the third opcode input parameter.

```
int newopc_init(ENVIRON *csound, newopc *p){
      p->var1 = (MYFLT) 0;
      p->var2 = *p->in3;
      return OK;
}
```

The processing function implementation will depend on the type of opcode that is being created. For control rate opcodes, with k- or i-type input parameters, we will be generating one output value at a time. The example below shows an example of this type of processing function. This simple example just keeps ramping up or down depending on the value of the second input. The output is offset by the first input and the ramping is reset if it reaches the value of `var2` (which is set to the third input argument in the constructor above).

```
int newopc_process_control(ENVIRON *csound, newopc *p){
      MYFLT cnt = p->var1 + *(p->in2);
      if(cnt > p->var2) cnt = (MYFLT) 0; /* check bounds */
      *(p->out) = *(p->in1) + cnt; /* generate output */
      p->var1 = cnt; /* keep the value of cnt */
      return OK;
}
```

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, because, as we are dealing with scalar inputs and outputs, the function has to process only one sample at a time. If we were to make an audio version of the control opcode above (disregarding its usefulness), we could have the change the code slightly. The basic difference is that we have an audio rate output instead of control rate. In this case, our output is a whole vector (a MYFLT array) with ksmps samples, so we have to write a loop to fill it. It is important to point out that the control rate and audio rate processing functions will produce exactly the same result. The difference here is that in the audio case, we will produce ksmps samples, instead of just one sample. However, all the vector samples will have the same value (which actually makes the audio rate function redundant, but we will use it just to illustrate our point).

```
int newopc_process_audio(ENVIRON *csound, newopc *p){
      int i, n = csound->ksmps;
      MYFLT *aout = p->out;  /* output signal */
      MYFLT cnt = p->var1 + *(p->in2);
      if(cnt > p->var2) cnt = (MYFLT) 0; /* check bounds */

      /* processing loop    */
      for(i=0; i < n; i++) aout[i] = *(p->in1) + cnt;

      p->var1 = cnt; /* keep the value of cnt */
      return OK;
}
```

In order for the Csound to be aware of the new opcode, we will have to register it. This is done by filling an opcode registration structure OENTRY array called localops (which is static, meaning that only one such array exists in memory at a time):

```
static OENTRY localops[] = {
```

```
{ "newopc", sizeof(newopc),  7,  "s",    "kki",  (SUBR) newopc_init,
(SUBR) newopc_process_control, (SUBR) newopc_process_audio }
};
```

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 (a polymorphic opcode).
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 7 means active at i-time (1), k-rate (2) and a-rate (4). This means that the opcode has an init function, plus a k-rate and an a-rate processing functions.
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.

Since we have defined our output as ‘s’, the actual processing function called by csound will depend on the output type. For instance

```
k1  newopc  kin1, kin2, i1
```

will use `newopc_process_control()`, whereas

```
a1  newopc  kin1, kin2, i1
```

will use `newopc_process_audio()`. This type of code is found for instance in the oscillator opcodes, which can generate control or audio rate (but in that case, they actually produce a different output for each type of signal, unlike our example).

Finally, it is necessary to add, at the end of the opcode C code the LINKAGE macro, which defines some functions needed for the dynamic loading of the opcode.

3.3 Building opcodes

The plugin opcode is build as a dynamic module. On Windows, with certain compiler packages (MSVC for instance), it is necessary to add a definition file for the dynamic-link library (DLL). This file will contain the exported symbols, the functions that are used by the Csound engine for loading the opcode:

```
EXPORTS
      opcode_init @1
      opcode_size @2
```

In Linux and OSX, this is not necessary. All we need is to build the opcode as a shared object, as demonstrated by the example below (assuming that we are in the top-level csound5 directory and all the relevant csound header files are in the standard or current directories, otherwise, use –I [path to header files]):

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

3.4 A more useful example

In order to understand the concepts introduced above, we will look now at a 'real-life' example, an opcode that will probably be much more useful than the previous one. The idea for this opcode is simple, yet original in Csound: we will take an input sound and loop it, with a k-rate pitch control. The recording should be somehow triggered by one of the parameters and we will have to make a decision on how to do it. In addition, we will need some memory for the looped sound and a crossfade to tidy up the ends of the loop.

In principle, we can use a trigger value of 1 to switch the recording on, which will then proceed automatically, stopping once the memory is full. From then on, the opcode will playback the loop. The trigger parameter can be used to re-trigger the recording. For this to happen, we will have to switch off the loop playback and switch on recording again. The same parameter can control the two things, when it is 0, it only copies the input to the output, when it is 1 it starts recording and then playback. If it becomes 0 again, it goes back to copying the input to the output, until it is 1 and it starts a new recording. In order to indicate if we are recording or playing g back, we can output a 'rec on light': a control-rate output that is 1 when recording is in process and 0 otherwise.

The dataspace will have to include the two outputs, the five inputs (signal, pitch, trigger, duration and crossfade duration), a buffer, indices for writing/reading the buffer and a reset control (to switch the recording on again and control the output mode):

```
#include "csdl.h"

typedef struct _sndloop {
  OPDS h;
  MYFLT *out, *recon;  /* output, record on */
  /* input, pitch, sound on, duration, crossfade  */
  MYFLT *sig, *pitch, *on, *dur, *cfd;
  AUXCH buffer; /* loop memory */
  long  wp;     /* writer pointer */
  MYFLT rp;     /* read pointer  */
  MYFLT a;       /* fade amp */
  MYFLT inc;     /* fade inc/dec */
  long  cfds;   /* crossfade in samples */
  long durs;    /* duration in samples */
  int  rst;     /* reset indicator */
} sndloop;
```

In the initialisation function, we convert the durations from seconds to samples, reset the writer pointer and reset variable. We also allocate enough memory to hold the loop. This is done by using the opcode API function `AuxAlloc(ENVIRON *, int`

`size, AUXCH *memory)`, which resides in the ENVIRON structure itself. This structure holds the relevant opcode API functions and other global environment symbols. The `AuxAlloc()` mechanism allocates memory only if it is needed, so we check if it we need to do it first. This is because instruments can re-use already allocated memory, saving the overhead of doing it again.

```
int sndloop_init(ENVIRON *csound, sndloop *p){

p->durs = (long) (*(p->dur)*csound->esr); /* dur in samps */
p->cfds = (long) (*(p->cfd)*csound->esr); /* fade in samps */

if(p->durs < p->cfds)
   return csound->InitError(csound,
         "Crossfade time longer than loop duration");

p->inc =  (MYFLT)1/p->cfds;          /* inc/dec */
p->a  = (MYFLT) 0;
p->wp = 0;   /* intialise write pointer */
p->rst = 1;               /* reset the rec control */
if(p->buffer.auxp==NULL)   /* allocate memory if necessary */
      csound->AuxAlloc(csound,
                       p->durs*sizeof(MYFLT), &p->buffer);

return OK;
}
```

The processing algorithm is based on the management of the different output modes of the opcode. There are three of them: bypass, recording and playback. In bypass mode, we simply copy the input to the output. The recording stage can actually be subdivided into three further blocks: fade in, middle of the loop and fade out. To do the crossfade we simply overlap the fade in and fade out portions, mixing the two together. The fade in/out is linear, so the envelope gain is increased/decreased by a constant amount every sample. In playback mode, we just output the buffer signal continuously, wrapping around at the ends. Pitch change is performed by incrementing the reader pointer by values larger or smaller than 1 (negative values provide reverse playback). This opcode implements a truncated buffer readout; for better signal-to-noise ratio an interpolated readout can be used. The processing function is shown below, fully commented:

```
int sndloop_process(ENVIRON *csound, sndloop *p){

int i, on = (int) *(p->on), recon, n = csound->ksmps;
long durs = p->durs, cfds = p->cfds, wp = p->wp;
MYFLT rp = p->rp, a = p->a, inc = p->inc;
MYFLT *out = p->out, *sig = p->sig, *buffer = p->buffer.auxp;
MYFLT pitch = *(p->pitch);
if(on) recon = p->rst; /* restart recording if switched on again */
else recon = 0;        /* else do not record */

 for(i=0; i < n; i++){
      if(recon){ /* if the recording is ON */
      /* fade in portion */
      if(wp < cfds){
            buffer[wp] = sig[i]*a;
            a += inc;
      }
      else {
```

```
            if(wp >= durs){ /* fade out portion */
            buffer[wp-durs] += sig[i]*a;
            a -= inc;
        }
        else buffer[wp] = sig[i];  /* middle of loop */
        }
        /* while recording connect input to output directly */
        out[i] = sig[i];
        wp++; /* increment writer pointer */
        if(wp == durs+cfds){  /* end of recording */
              recon = 0;  /* OFF */
              p->rst = 0; /* reset to 0 */
              p->rp = (MYFLT) wp; /* rp pointer to start from here */
                    }
        }
        else {
        if(on){ /* if opcode is ON */
          out[i] = buffer[(int)rp]; /* output the looped sound */
          rp += pitch;              /* read pointer increment */
          while(rp >= durs) rp -= durs; /* wrap-around */
          while(rp < 0) rp += durs;
              }
        else {   /* if opocde is OFF */
          out[i] = sig[i]; /* copy input to the output */
          p->rst = 1;   /* reset: ready for new recording */
          wp = 0; /* zero write pointer */
              }
         }
        }
      p->rp = rp; /* keep the values */
      p->wp = wp;
      p->a = a;
      *(p->recon) = (MYFLT) recon; /* output 'rec on light' */

      return OK;
}
```

Note that we are copying the heavily-used dataspace variables into local ones, which are generally faster to access, in order to be more efficient. We try to avoid all unnecessary operations and try to reduce the code to a minimum set of them. We could perhaps have used only one pointer, for reading and writing, but for the sake of clarity, we are employing separate ones.

To register the opcode, we will provide the OENTRY values as shown below. Note that our opcode has two outputs, one at the audio rate and another at the control rate. We only have an initialisation and audio processing functions, so the third value is set to 1+4 (5):

```
static OENTRY localops[] = {
{"sndloop", sizeof(sndloop), 5, "ak", "akkii", (SUBR)sndloop_init,
  0,(SUBR)sndloop_process}
};

LINKAGE
```

4 Plugin function tables

A new type of dynamic module, introduced in Csound 5, is the plugin 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. 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.

4.1 Using function tables

Function tables have a number of different applications, they can hold audio signals, envelopes, impulse responses, polynomials, etc.. Opcodes using function tables will be able to access these by using the ENVIRON functions `FtFind()` and `Ftnp2Find()`, among others. The first one is used for the usual tables found in Csound, which are power-of-two (plus 1) size, so it guarantees that tables will conform to that format. The other function finds tables of any size, which are useful for reading deferred-allocation tables created by GEN01. These are sized to match the length of the audio data read from a soundfile.

In order to explore the use of function tables, we will develop here a variation of the sndloop opcode, which will use loop a sound read from a table, with user-defined loop start/end times and crossfade (flooper, a "crossfade looper"). This opcode will also use linear interpolation for table lookup, in order to produce a better quality output. We will use `Ftnp2Find()` so that we can also avail of deferred allocation tables. Since our table lookup algorithm will use floating-point arithmetic, there is no need to limit table sizes to powers-of-two.

Our approach is going to be slightly different from the 'live-input' looper. Since we have, at initialisation time, the loop parameters (which will not change) plus the input audio (from the function table), we can construct the loop once and just play it back later. This will simplify things and will provide a cleaner code. In terms of opcode parameters, all we need is: amplitude, pitch, start, duration, crossfade and function table number. Our dataspace will look like this:

```
typedef struct _flooper {
      OPDS h;
      MYFLT *out;  /* output, record on */

     /* amplitude, pitch, start, dur, crossfade, ftable */
      MYFLT *amp, *pitch, *start, *dur, *cfd, *ifn;
      AUXCH buffer; /* loop memory */
      FUNC  *sfunc;  /* function tabble */
      long strts;   /* start in samples */
      long  durs;    /* duration in samples */
      MYFLT  ndx;    /* table lookup ndx */
} flooper;
```

As we outlined above, most of the work will be done in the initialisation function. We will take the loop parameters and first check if they are consistent. Then we will get the function table with `Ftnp2Find()` , test if the table is long enough for the required loop, allocate the loop memory and write the loop to it. We will use basically the same code used in sndloop for this task. The loop will start at a user-defined point and end after the specified duration, with the crossfade folding the audio samples found after the loop end back over the beginning (fig. 1).

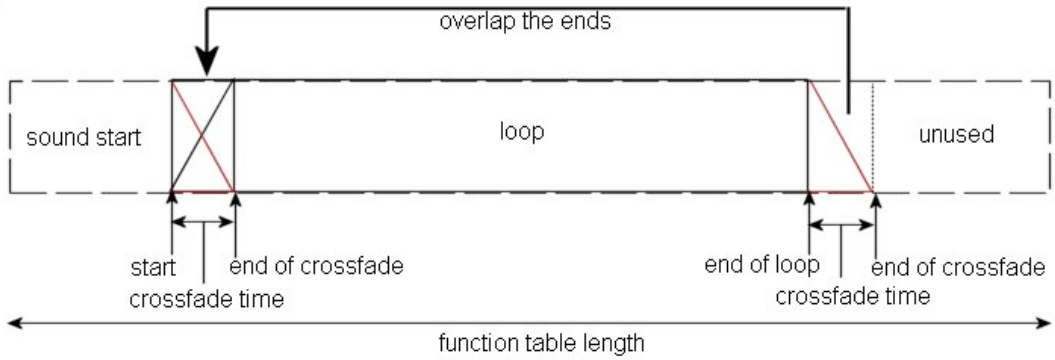

Figure 1. Loop structure

```
int flooper_init(ENVIRON *csound, flooper *p){

MYFLT *tab, *buffer, a = (MYFLT) 0, inc;
long cfds = (long) (*(p->cfd)*csound->esr);     /* fade in samps  */
long starts = (long) (*(p->start)*csound->esr); /* start in samps */
long durs = (long)  (*(p->dur)*csound->esr);    /* dur in samps   */
long len, i;

if(cfds > durs)
  return csound->InitError(csound,
        "crossfade longer than loop duration\n");


inc =  (MYFLT)1/cfds;           /* inc/dec */
p->sfunc = csound->FTnp2Find(csound, p->ifn);  /* function table */
f(p->sfunc==NULL)
    return csound->InitError(csound,"function table not found\n");

tab = p->sfunc->ftable,         /* func table pointer */
len = p->sfunc->flen;           /* function table length */
if(starts > len)
    return csound->InitError(csound,
         "start time beyond end of table\n");

if(starts+durs+cfds > len)
    return csound->InitError(csound,
          "table not long enough for loop\n");

if(p->buffer.auxp==NULL)    /* allocate memory if necessary */
      csound->AuxAlloc(csound,(durs+1)*sizeof(MYFLT), &p->buffer);

inc = (MYFLT)1/cfds;        /* fade envelope incr/decr */
buffer = p->buffer.auxp;    /* loop memory */

/* we now write the loop into memory */
for(i=0; i < durs; i++){
      if(i < cfds){
            buffer[i] = a*tab[i+starts];
            a += inc;
            }
      else buffer[i] = tab[i+starts];;
}
/*  crossfade section */
for(i=0; i  < cfds; i++){
```

```
      buffer[i] += a*tab[i+starts+durs];
      a -= inc;
}

buffer[durs] = buffer[0]; /* for wrap-around interpolation */
p->strts = starts;
p->durs = durs;
p->ndx = (MYFLT) 0;      /* lookup index */
return OK;
}
```

Since we are going to read the buffer using linear interpolation, we will actually need to allocate one extra location at the end of it. The reason behind this is that the interpolation procedure will use two samples, at the current lookup position and the next. The last sample in the buffer will be a copy of the first, so that the interpolation can be performed smoothly.

The processing function is quite simple. It is based on a table-lookup oscillator design, except that when we start looping, the oscillator will switch from reading the function table to the loop memory. The index is incremented according to the k-rate pitch control and is kept within the bounds of the loop memory. Linear interpolation is implemented as follows: if an index position p falls between table/buffer points x_1 and x_2, whose values are y_1 and y_2, then, by linear interpolation we have

$$y = y_1 + (y_2 - y_1)(p - x_1)$$

All we need to do is to find the fractional part of the index $(p - x_1)$, multiply it by the difference between the two adjacent values and add it to the current position value.

```
int flooper_process(ENVIRON *csound, flooper *p){

int i, n = csound->ksmps;
long start = p->strts+p->durs, end = p->durs+start;
MYFLT *out = p->out, *buffer = p->buffer.auxp;
MYFLT amp = *(p->amp), pitch = *(p->pitch);
MYFLT *tab = p->sfunc->ftable, ndx = p->ndx, frac;
int tndx;

for(i=0; i < n; i++){
      tndx  = (int) ndx;
      frac = ndx - tndx;
      if(ndx < start) { /* this is the start portion of the sound */
        out[i] = amp*(tab[tndx] + frac*(tab[tndx+1] - tab[tndx]));
        ndx += pitch;
      }
      else { /* this is the loop section */
        tndx -= start;
       out[i] = amp*(buffer[tndx] + frac*(tab[tndx+1] - tab[tndx]));
       ndx += pitch;
       while (ndx >= end) ndx -= end;
       while (ndx < start) ndx += start;
            }
      }
p->ndx = ndx;
return OK;
}
```

The opcode OENTRY table entry is shown below:

```
static OENTRY localops[] = {
      {"flooper", sizeof(flooper), 5,
      "a", "kkiiii", (SUBR)flooper_init, 0 ,
      (SUBR)flooper_process}
};
```

5 Processing 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 actually 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 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 bin. This is probably the most musically meaningful of the DFT-based output formats and can be generated by phase vocoder (PV) analysis (implemented by the opcode pvsanal and the utility analysis program pvanal). The fsig data type is defined by the following C structure, which :

```
typedef struct pvsdat {
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;
```

The structure holds all the necessary data to describe the signal type: the DFT size (N), which will determine the number of analysis bins (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.

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 allocated externally, in similar fashion to audio-rate vectors and control-rate scalars; however the DFT frame allocation is done by the opcode that generates 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 code example we will develop here takes advantage of one of the key aspects of the PV data format: the separation between amplitudes and frequencies. We will implement a spectral highlighter, or spectral arpeggiator. We are borrowing the idea from Trevor Wishart's specarp program from the original Composer's Desktop Project suite of PV data transformation software. The concept (and implementation) is very simple, yet effective: we will emphasize one central bin, whilst attenuating all others. This will happen for every output frame, depending on three k-rate controls: bin index, depth of attenuation and highlight gain. In order to provide a framesize-independent bin index, we will take it as normalised value, between 0 and 1, scaling it to the fftsize. The depth will control how much all bins, except the highlighted one, will be attenuated, from 0 (no attenuation) to 1 (full attenuation). The highlight gain is simply the positive gain applied to the selected bin. This transformation only concerns the amplitudes of each spectral bin, so we will pass all frequencies unchanged. The opcode dataspace and the initialisation function are shown below:

```
typedef struct _pvsarp {
        OPDS h;
        PVSDAT  *fout;
        PVSDAT  *fin;
        MYFLT   *cf,*kdepth, *gain;
        unsigned long   lastframe;
} pvsarp;

int pvsarp_init(ENVIRON *csound, pvsarp *p)
{
    long N = p->fin->N; /* fftsize */

    if (p->fout->frame.auxp==NULL)
        csound->AuxAlloc(csound,(N+2)*sizeof(float),&p->fout->frame);
    /* initialise the PVSDAT structure */
    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))
           return csound->InitError(csound,
```

```
   "pvsarp: signal format must be amp-phase or amp-freq.\n");
    return OK;
}
```

The opcode dataspace contains pointers to the output and input fsig, as well as the k-rate input parameters and a frame counter. The init function has to allocate space for the output fsig DFT frame, as well as setting the various PVSDAT parameters.

```
int pvsarp_process(ENVIRON *csound, pvsarp *p)
{
    long i,j,N = p->fout->N, bins = N/2 + 1;
    float g = (float) *p->gain;
    MYFLT kdepth = (MYFLT) *(p->kdepth), cf = (MYFLT) *(p->cf);
    float *fin = (float *) p->fin->frame.auxp;
    float *fout = (float *) p->fout->frame.auxp;

    if(fout==NULL)
      return csound->PerfError(csound,
                        "pvsarp: not initialised\n");

    /* if a new frame is ready for processing */
    if(p->lastframe < p->fin->framecount) {
        /* limit cf and kdepth to 0 - 1 range */
        /* scale cf to the number of spectral bins */
        cf = cf >= 0 ? (cf < 1 ? cf*bins : bins-1) : 0;
        kdepth = kdepth >= 0 ? (kdepth <= 1 ? kdepth :
                                    (MYFLT)1.0): (MYFLT)0.0;
            /* j counts bins, whereas i counts frame positions */
            for(i=j=0;i < N+2;i+=2, j++) {
              /* if the bin is to be highlighted */
              if(j == (int) cf) fout[i] = fin[i]*g;
              /* else attenuate it */
              else fout[i] = (float)(fin[i]*(1-kdepth));
              /* pass the frequencies unchanged */
              fout[i+1] = fin[i+1];
            }
            /* update the internal frame count */
            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. As mentioned above, this process only alters the amplitudes of each bin, indexed by `i`, passing the frequencies (`i+1`).

Finally, the localops OENTRY structure for this opcode will look like this:

```
static OENTRY localops[] = {
       {"pvsarp", sizeof(pvsarp), 3, "f", "fkkk", (SUBR)pvsarp_init,
       (SUBR)pvsarp_process}
};
```

6 Postlude: documentation

It is generally assumed that software development stops here, with the finished code, when in reality a final and important stage is still to be completed. This is of course the writing of suitable documentation for our work. Without it, any software is of limited use, perhaps only for its developers, but even in this case, its usability is not guaranteed. Therefore, it is important that a manual page and some examples are supplied with every new opcode, be it plugin or UDO.

A Csound manual page template for opcode reference is very well established. The manual currently exists in many formats: plain text, help file, html, xml, etc. The latter is particularly flexible as it can itself generate documentation in different forms. For those interested in that format, a quick look at some opcode entries will be enough for learning the xml tags used in the manual.

In general, the manual page should have the following format:

1. Title: generally the opcode name
2. Short description: one-line on what it does.
3. Description: under that heading, a longer discussion of what the opcode does and how to use it.
4. Syntax: the opcode Csound syntax.
5. Initialisation parameters and their description
6. Performance parameters and their description, including possibly their expected ranges etc.
7. Csound code examples of opcode usage.

As an example, here is the plain-text manual entry for the pvsarp opcode discussed in the previous section:

```
PVSARP

Spectral filtering and arpeggiation of PV streams.

DESCRIPTION
Pvsarp takes an input fsig and arpeggiates it according to bin index,
effect depth and boost amplitude controls. On each fsig input frame,
it will apply a gain to the target bin, as well as attenuating all
the other frequency bins according to the depth control.

fsig  pvsarp fin, kcf, kdepth, kgain

[INITIALISATION: this is omitted as the opcode does not have i-type
parameters]

PERFORMANCE
fin – input fsig
kbin –  bin index (normalised, 0-1, equivalent to the 0Hz – Nyquist
range), determining the target bin for arpeggiation.
kdepth – depth of attenuation of surrounding bins (0 <= kdepth <= 1)
kgain – gain (multiplier), applied to the target bin.

EXAMPLE
```

```
This example arpeggiates the spectrum of an input signal, using an
LFO (with a triangle wave) to control the bin index. The modulation
width is 0.05*SR/2 (1102.5 at 44.1KHz) and the lowest frequency is
0.005*SR/2 (110.25).

instr 1

ifftsize = 1024
iwtype = 1
ifr = 0.2
idepth = 1

asig  in
kbin  oscil  0.05, ifr, 1
kbin = kbin + 0.005

fsig  pvsanal   asig, ifftsize, ifftsize/4, ifftsize, iwtype
ftps  pvsarp   fsig, kbin, 0.95, 15
atps  pvsynth  ftps

       out atps
endin

f1 0 1024 7 0 512 1 512 0
i1 0 20
```

7 Final remarks

Opcode and GEN development has become more accessible with the introduction of the plugin framework, as well as the UDO facility. Now opcodes can be customised by users with a basic understanding of C programming and, for those without it, the UDO mechanism can be used a substitute. As a result, opcode additions will not have to wait for new releases of Csound, they are now independent and the distribution of opcode libraries can be made separately. This is a feature that, allied with the straightforward opcode format, makes Csound probably one of the easiest synthesis/processing systems to extend and customise.