THE CSOUND PLUGIN OPCODE FRAMEWORK

Victor Lazzarini
Maynooth University,
Ireland
victor.lazzarini@nuim.ie

ABSTRACT

This article introduces the Csound Plugin Opcode Framework (CPOF), which aims to provide a simple lightweight C++ framework for the development of new unit generators for Csound. The original interface for this type work is provided in the C language and it still provides the most complete set of components to cover all possible requirements. CPOF attempts to allow a simpler and more economical approach to creating plugin opcodes. The paper explores the fundamental characteristics of the framework and how it is used in practice. The helper classes that are included in CPOF are presented with examples. Finally, we look at some uses in the Csound source codebase.

1. INTRODUCTION

Plugins in Csound [1] are usually written in C, which provides a low-level access to the engine, allowing an uncompromising and complete scope for new opcode (unit generator) development. For most system developers, this will continue to be the best working environment. However, for many of the more common forms of plugins, this interface can be complex and cumbersome. In particular, we might like to take advantage of an object-oriented approach so that we can, for instance, re-use code more extensively and take advantage of existing algorithms and libraries. For this, the ideal language is probably the C++ in its more modern incarnations [2]. In this paper, we describe a lightweight framework to facilitate programming in this environment, the Csound Plugin Opcode Framework (CPOF – *see-pough* or *cipó*, *vine* in Portuguese). While there is a pre-existing C++ interface for plugin writing in the Csound codebase, this framework provides an alternative to it, that attempts to be thin, simple, complete, and handling internal Csound resources in a safe way (using the same mechanisms provided by the underlying C interface).

1.1 Design Fundamentals

The conditions in which the framework is supposed to work constrain significantly the scope of what is possible. In particular,

1. Csound is written in C. It instantiates opcodes and makes calls to opcode processing functions, but it does not know anything about C++.
2. Virtual functions are a no-go zone (due to 1). Everything needs to be set at compile time.
3. Registering processing functions requires that these are defined as static.
4. Up to three different functions should be registered (for different action times).
5. In C, an opcode dataspace "inherits" from a base structure (`OPDS`) by adding this as its first member variable.

One of the possibilities for designing a framework based on inheritance without virtual functions is to employ a method called *curiously recurring template pattern* (CRTP) [3] . A variation of this is in fact used in the pre-existing C++ opcode development interface. However, we can do better with a much simpler approach. There is no need for a compile-time mimicking of the virtual-function mechanism. This is because it is not our objective to have a general purpose framework for C++ programs, where users would be instantiating their own objects and possibly using generic pointers and references that need to bind to the correct override.

Here there is a much narrower scope: Csound does the instantiation and the calls, so we can make the decision at compile time just by providing functions that hide rather than override (in the virtual-function sense) the base class ones. In this case, hiding plays the same role as overriding. So we can have a plugin base class from which we will inherit to create the actual opcodes. This class will inherit from `OPDS` and provide some extra members that are commonly used by all opcodes. It will also provide stub methods for the processing functions, which then can be 'specialised' in the derived classes.

Given that they will not ever be called, it would seem that these stubs are surplus to requirements. However, having these allows a considerable simplification in the plugin registration process. We can just register any plugin in the same way, even if it does only provide one or two of the required processing functions. The stubs play an important role to keep the compiler happy in this scheme, even if Csound will not take any notice of them.

This mechanism requires that we provide function templates for registration. These get instantiated with exactly the derived types and are used to glue the C++ code into

Csound. Each one of them is minimal: consisting just of a call to the instantiated class processing function. For instance, one of these, used at instrument initialisation time, is defined simply as

```
template <typename T>
 int init(CSOUND *csound, T *p) {
  p->csound = (Csound *)csound;
  return p->init();
 }
```

In this case `T` is derived type passed on registration (which is defined at compile time). When running, Csound calls this function, which in its turn delegates directly to the plugin class in question. Note that this is all hidden from the framework user, who only needs to derive her classes and register them. As we will see in the following sections, this scheme enables significant economy, re-use and reduction in code verbosity (one of the issues with CRTP).

2. THE FRAMEWORK

CPOF is based on two class templates, which plugin classes derive from and instantiate. Opcode classes can then be registered by instantiating and calling one of the two overloaded registration function templates. All CPOF code is declared under the namespace **`csnd`**. In this section, we discuss the base class templates, three simple cases of derived plugins and their registration.

2.1 The Base Classes

The framework base classes are actually templates which need to be derived and instantiated by the user code. The most general of these is **`Plugin`**[1]. To use it, we program our own class by subclassing it and passing the number of output and inputs our opcode needs as its template arguments:

```
#include <plugin.h>
struct MyPlug : csnd::Plugin<1,1> { };
```

The above lines will create a plugin opcode with one output (first template argument) and one input (second template argument). This class defines a complete opcode, but since it is only using the base class stubs, it is also fully non-op.

To make it do something, we will need to reimplement one, two or three of its methods. As outlined above, this base class is derived from the Csound structure **`OPDS`** and has the following members:

- `outargs`: a `Params` object holding output arguments.
- `inargs`: input arguments (`Params`).
- `csound`: a pointer to the Csound engine object.
- `offset`: the starting position of an audio vector (for audio opcodes only).
- `nsmps`: the size of an audio vector (also for audio opcodes only).
- `init()`, `kperf()` and `aperf()` non-op methods, to be reimplemented as needed (see sec. 2.2).

The other base class in CPOF is **`FPlugin`**, derived from **`Plugin`**, which adds an extra facility for fsig (streaming frequency-domain) plugins:

- `framecount`: a member to hold a running count of fsig frames.

Input and output arguments, which in C are just raw pointers are conveniently wrapped by the Params class, which provides a number of methods to access them according to their expected Csound variable types.

2.2 Deriving opcode classes

Csound has two basic *action* times for opcodes: init and perf-time. The former runs a processing routine once per instrument instantiation (and/or once again if a re-init is called for). Code for this is placed in the **`Plugin`** class `init()` function. Perf-time code runs in a loop and is called once every control (k-)cycle. The other class methods `kperf()` and `aperf()` are called in this loop, for control (scalar) and audio (vectorial) processing. The following examples demonstrate the derivation of plugin classes for each one of these opcode types (i, k or a). Note that k and a opcodes can also use i-time functions if they require some sort of initialisation.

2.2.1 Init-time opcodes

For init-time opcodes, all we need to do is provide an implementation of the `init()` method:

```
struct Simplei : csnd::Plugin<1,1> {
  int init() {
    outargs[0] = inargs[0];
    return OK;
  }
};
```

In this simple example, we just copy the input arguments to the output once, at init-time. Each scalar input type can be accessed using array indexing. All numeric argument data is real, declared as **`MYFLT`**, the internal floating-point type used by Csound.

2.2.2 K-rate opcodes

For opcodes running only at k-rate (no init-time operation), all we need to do is provide an implementation of the `kperf()` method:

```
struct Simplek : csnd::Plugin<1,1> {
  int kperf() {
    outargs[0] = inargs[0];
    return OK;
  }
};
```

Similarly, in this simple example, we just copy the input arguments to the output at each k-period.

[1] All CPOF code is declared in `plugin.h`

2.2.3 A-rate opcodes

For opcodes running only at a-rate (and with no init-time operation), all we need to do is provide an implementation of the `aperf()` method:

```
struct Simplea : csnd::Plugin<1,1> {
 int aperf() {
  std::copy(inargs(0)+offset,
    inargs(0)+nsmps, outargs(0));
    return OK;
  }
};
```

Because audio arguments are `nsmps`-size vectors, we get these using the overloaded **`operator`**`()` for the `inargs` and `outargs` objects, which takes the argument number as input and returns a **`MYFLT`** pointer to the vector.

2.3 Registering opcodes with Csound

Once we have written our opcode classes, we need to tell Csound about them, so that they can be used, for this we use the CPOF function template `plugin()`:

```
template <typename T>
int plugin(Csound *csound,
           const char *name,
           const char *oargs,
           const char *iargs,
           uint32_t thrd,
           uint32_t flags = 0)
```

Its parameters are:

- `csound`: a pointer to the **`Csound`** object to which we want to register our opcode.
- `name`: the opcode name as it will be used in Csound code.
- `oargs`: a string containing the opcode output types, one identifier per argument
- `iargs`: a string containing the opcode input types, one identifier per argument
- `thrd`: a code to tell Csound when the opcode should be active.
- `flags`: multithread flags (generally 0 unless the opcode accesses global resources).

For opcode type identifiers, the most common types are: a (audio), k (control), i (i-time), S (string) and f (fsig). For the thread argument, we have the following options, which depend on the processing methods implemented in our plugin class:

- `thread::i`: indicates `init()`.
- `thread::k`: indicates `kperf()`.
- `thread::ik`: indicates `init()` and `kperf()`.
- `thread::a`: indicates `aperf()`.
- `thread::ia`: indicates `init()` and `aperf()`.
- `thread::ika`: indicates `init()`,`kperf()` and `aperf()`.

We instantiate and call these template functions inside the plugin dynamic library entry-point function `on_load()`. This function needs to implemented only **once** [2] in each opcode library. For example:

```
#include <modload.h>
void csnd::on_load(Csound *csound){
  csnd::plugin<Simplei>(csound,
     "simple", "i", "i",
     csnd::thread::i);
  csnd::plugin<Simplek>(csound,
      "simple", "k", "k",
       csnd::thread::k);
  csnd::plugin<Simplea>(csound,
       "simple", "a", "a",
       csnd::thread::a);
  return 0;
}
```

will register the `simple` *polymorphic* opcode, which can be used with i-, k- and a-rate variables. In each instantiation of the plugin registration template, the class name is passed as an argument to it, followed by the function call. If the class defines two specific static members, `otypes` and `itypes`, to hold the types for output and input arguments, declared as

```
struct MyPlug : csnd::Plugin<1,2> {
  static constexpr
   char const *otypes = "k";
  static constexpr
   char const *itypes = "ki";
  ...
};
```

then we can use a simpler overload of the plugin registration function:

```
template <typename T>
int plugin(Csound *csound,
           const char *name,
           uint32_t thread,
           uint32_t flags = 0)
```

For some classes, this might be a very convenient way to define the argument types. For other cases, where opcode polymorphism might be involved, we might re-use the same class for different argument types, in which case it is not desirable to define these statically in a class.

3. SUPPORT CLASSES

Plugins developed with CPOF can avail of a number of helper classes for accessing resources in the Csound engine. These include a class encapsulating the engine itself,

[2] The header file `modload.h`, where `on_load()` is declared, contains three boilerplate calls to Csound module C functions, required for Csound to load plugins properly. For this reason, each plugin library should also include this header only **once**, otherwise duplicate symbols will cause linking errors.

and classes for resource allocation, input/output access/-manipulation, threads, and support for constructing objects allocated in Csound's heap.

3.1 The Csound Engine Object

Plugins are run by an engine which is encapsulated by the **`Csound`** class. They all hold a pointer to this, called `csound`, which is needed for some of the operations invoking parameters, or for some utility methods (such as console messaging, MIDI data access, FFT operations). The following are the public methods of the Csound class:

- `init_error()`: takes a string message and signals an initialisation error.
- `perf_error()`: takes a string message, an instrument instance and signals a performance error.
- `warning()`: warning messages.
- `message()`: information messages.
- `sr()`: returns engine sampling rate.
- `_0dbfs()`: returns max amplitude reference.
- `_A4()`: returns A4 pitch reference.
- `nchnls()`: return number of output channels for the engine.
- `nchnls_i()`: same, for input channel numbers.
- `current_time_samples()`: current engine time in samples.
- `current_time_seconds()`: current engine time in seconds.
- `midi_channel()`: midi channel assigned to this instrument.
- `midi_note_num()`: midi note number (if the instrument was instantiated with a MIDI NOTE ON).
- `midi_note_vel()`: same, for velocity.
- `midi_chn_aftertouch()`, `midi_chn_polytouch()`, `midi_chn_ctl()`, `midi_chn_pitchbend()`: midi data for this channel.
- `midi_chn_list()`: list of active notes for this channel.
- `fft_setup()`, `rfft()`, `fft()`: FFT operations.

In addition to these, the Csound class also holds a deinit method registration function template for Plugin objects to use:

```
template <typename T>
 void plugin_deinit(T *p);
```

This is only needed if the Plugin has allocated extra resources using mechanisms that require de-allocation. It is not employed in most cases, as we will see below. To use it, the plugin needs to implement a `deinit()` method and then call the `plugin_deinit()` method passing itself (through a `this` pointer) in its own `init()` function:

```
csound->plugin_deinit(this);
```

3.2 Audio Signals

Audio signal variables are vectors of `nsmps` samples and we can access them through raw **`MYFLT`** pointers from input and output parameters. To facilitate a modern C++ approach, the `AudioSig` class wraps audio signal vectors conveniently, providing iterators and subscript access. Objects are constructed by passing the current plugin pointer (**`this`**) and the raw parameter pointer, e.g.:

```
csnd::AudioSig in(this,inargs(0));
csnd::AudioSig out(this,outargs(0));
std::copy(in.begin(), in.end(),
          out.begin());
```

3.3 Memory Allocation

For efficiency and to prevent leaks and undefined behaviour we need to leave all memory allocation to Csound and refrain from using C++ dynamic memory allocators or standard library containers that use dynamic allocation behind the scenes (e.g. `std::vector`).

This requires us to use the AuxAlloc mechanism implemented by Csound for opcodes. To allow for ease of use, CPOF provides a wrapper template class (which is not too dissimilar to `std::vector`) for us to allocate and use as much memory as we need. This functionality is given by the **`AuxMem`** class:

- `allocate()`: allocates memory (if required).
- **`operator`**`[]`: array-subscript access to the allocated memory.
- `data()`: returns a pointer to the data.
- `len()`: returns the length of the vector.
- `begin()`, `cbegin()` and `end()`, `cend()`: return iterators to the beginning and end of data.
- `iterator` and `const_iterator`: iterator types for this class.

As an example of use, we can implement a simple delay line [4] opcode, whose delay time is set at i-time, providing a slap-back echo effect:

```
struct DelayLine : csnd::Plugin<1,2> {
 csnd::AuxMem<MYFLT> delay;
 csnd::AuxMem<MYFLT>::iterator iter;

 int init() {
  delay.allocate(csound,
```

```
      csound->sr()*inargs[1]);
  iter = delay.begin();
  return OK;
 }

 int aperf() {
  csnd::AudioSig in(this, inargs(0));
  csnd::AudioSig out(this, outargs(0));

  std::transform(in.begin(),in.end(),
        out.begin(),[this](MYFLT s){
        MYFLT o = *iter;
        *iter = s;
        if(++iter == delay.end())
                 iter = delay.begin();
         return o;});
  return OK;
 }
};
```

In this example, we use an **AuxMem** interator to access the delay vector. It is equally possible to access each element with an array-style subscript. The memory allocated by this class is managed by Csound, so we do not need to be concerned about disposing of it. To register this opcode, we do

```
csnd::plugin<DelayLine>(csound,
        "delayline", "a", "ai",
        csnd::thread::ia);
```

3.4 Function Table Access

Access to function tables has also been facilitated by a thin wrapper class that allows us to treat it as a vector object. This is provided by the Table class:

- `init()`: initialises a table object from an opcode argument pointer.
- **`operator`**`[]`: array-subscript access to the function table.
- `data()`: returns a pointer to the function table data.
- `len()`: returns the length of the table (excluding guard point).
- `begin()`, `cbegin()` and `end()`, `cend()`: return iterators to the beginning and end of the function table.
- `iterator` and `const_iterator`: iterator types for this class.

An example of table access is given by an oscillator opcode, which is implemented in the following class:

```
struct Oscillator : csnd::Plugin<1,3> {
 csnd::Table tab;
 double scl;
 double x;
 int init() {
  tab.init(csound,inargs(2));
  scl = tab.len()/csound->sr();
  x = 0;
  return OK;
 }

 int aperf() {
  csnd::AudioSig out(this, outargs(0));
  MYFLT amp = inargs[0];
  MYFLT si = inargs[1] * scl;

  for(auto &s : out) {
    s = amp * tab[(uint32_t)x];
    x += si;
    while (x < 0) x += tab.len();
    while (x >= tab.len())
                  x -= tab.len();
  }
  return OK;
 }
};
```

The table is initialised by passing the relevant argument pointer to it (using its `data()` method). Also note that, as we need a precise phase index value, we cannot use iterators in this case (without making it very awkward), so we employ straightforward array subscripting. The opcode is registered by

```
csnd::plugin<Oscillator>(csound,
        "oscillator", "a", "kki",
        csnd::thread::ia);
```

3.5 String Types

String variables in Csound are held in a `STRINGDAT` data structure, containing a data member that holds the actual string and a size member with the allocated memory size. While CPOF does not wrap strings, it provides a translated access to string arguments through the argument objects `str_data()` function. This takes an argument index (similarly to `data()`) and returns a reference to the string variable, as demonstrated in this example:

```
struct Tprint : csnd::Plugin<0,1> {
 int init() {
  char *s = inargs.str_data(0).data;
  csound->message(s);
  return OK;
 }
};
```

This opcode will print the string to the console. Note that we have no output arguments, so we set the first template parameter to 0. We register it using

```
csnd::plugin<Tprint>(csound, "tprint",
        "", "S",  csnd::thread::i);
```

3.6 Streaming Spectral Types

For streaming spectral processing opcodes, we have a different base class with extra facilities needed for their operation (**FPlugin**). In Csound, fsig variables, which carry spectral data streams, are held in a **PVSDAT** data structure.

To facilitate their manipulation, CPOF provides the **Fsig** class, derived from **PVSDAT**. To access phase vocoder bins, a container interface is provided by **pv_frame** (**spv_frame** for the sliding mode) [3] . This holds a series of **pv_bin** (**spv_bin** for sliding) [4] objects, which have the following methods:

- `amp()`: returns the bin amplitude.
- `freq()`: returns the bin frequency.
- `amp(`**float** `a)`: sets the bin amplitude to `a`.
- `freq(`**float** `f)`: sets the bin frequency to `f`.
- **operator**`*(`**pv_bin** `f)`: multiply the amp of a pvs bin by `f.amp`.
- **operator**`*(`**MYFLT** `f)`: multiply the bin amp by `f`
- **operator**`*=()`: unary versions of the above.

The **pv_bin** class can also be translated into a `std::complex<`**float**`>`, object if needed. This class is also fully compatible with the C complex type and an object obj can be cast into a float array consisting of two items (or a float pointer), using **reinterpret_cast**`<`**float**`(&)[2]>(obj)` or **reinterpret_cast**`<float*>(&obj)`. The Fsig class has the following methods:

- `init()`: initialisation from individual parameters or from an existing fsig. Also allocates frame memory as needed.
- `dft_size()`, `hop_size()`, `win_size()`, `win_type()` and `nbins()`, returning the PV data parameters.
- `count()`: get and set fsig framecount.
- `isSliding()`: checks for sliding mode.
- **fsig_format**`()`: returns the fsig data format (**fsig_format**`::pvs`, `::polar` `::complex`, or `::tracks`).

The **pv_frame** (or **spv_frame**) class contains the following methods:

- **operator**`[]`: array-subscript access to the spectral frame
- `data()`: returns a pointer to the spectral frame data.
- `len()`: returns the length of the frame.
- `begin()`, `cbegin()` and `end()`, `cend()`: return iterators to the beginning and end of the data frame.
- `iterator` and `const_iterator`: iterator types for this class.

Fsig opcodes run at k-rate but will internally use an update rate based on the analysis hopsize. For this to work, a frame count is kept and checked to make sure we only process the input when new data is available. The following example class implements a simple gain scaler for fsigs:

```
struct PVGain : csnd::FPlugin<1, 2> {
 static constexpr
        char const *otypes = "f";
 static constexpr
         char const *itypes = "fk";

 int init() {
  if(inargs.fsig_data(0).isSliding()){
   char *s = "sliding not supported";
   return csound->init_error(s);
  }
  if(inargs.fsig_data(0).fsig_format()
     != csnd::fsig_format::pvs &&
     inargs.fsig_data(0).fsig_format()
     != csnd::fsig_format::polar){
     char *s = "format not supported";
     return csound->init_error(s);
  }
  csnd::Fsig &fout =
        outargs.fsig_data(0);
  fout.init(csound,
        inargs.fsig_data(0));
  framecount = 0;
  return OK;
 }

 int kperf() {
   csnd::pv_frame &fin =
        inargs.fsig_data(0);
   csnd::pv_frame &fout =
        outargs.fsig_data(0);
   uint32_t i;

   if(framecount < fin.count()) {
     std::transform(fin.begin(),
       fin.end(), fout.begin(),
       [this](csnd::pv_bin f){
        return f *= inargs[1];});
     framecount =
        fout.count(fin.count());
    }
    return OK;
  }
};
```

Note that, as with strings, there is a dedicated method in the arguments object that returns a ref to an Fsig class (which can also be assigned to a **pv_frame** ref). This

[3] **pv_frame** is a convenience typedef for **Pvframe**<**pv_bin**>, whereas **spv_frame** is **Pvframe**<**spv_bin**>

[4] **pv_bin** is **Pvbin**<**float**> and **spv_bin** is **Pvbin**<**MYFLT**>

is used to initialise the output object at i-time and then to obtain the input and output variable data Csound processing. The `framecount` member is provided by the base class, as well as the format check methods. This opcode is registered using

```
csnd::plugin<PVGain>(csound, "pvg",
        csnd::thread::ik);
```

3.7 Array Variables

Opcodes with array inputs or outputs use the data structure **ARRAYDAT** for parameters. Again, in order to facilitate access to these argument types, CPOF provides a wrapper class. The framework currently supports only one-dimensional arrays directly. The template container class **Vector**, derived from `ARRAYDAT`, holds the argument data. It has the following members:

- `init()`: initialises an output variable.
- **operator**`[]`: array-subscript access to the vector data.
- `data()`: returns a pointer to the vector data.
- `len()`: returns the length of the vector.
- `begin()`, `cbegin()` and `end()`, `cend()`: return iterators to the beginning and end of the vector.
- `iterator` and `const_iterator`: iterator types for this class.
- `data_array()`: returns a pointer to the vector data.

In addition to this, the `inargs` and `outargs` objects in the **Plugin** class have a template method that can be used to get a **Vector** class reference. A trivial example is shown below:

```
struct SimpleArray : csnd::Plugin<1, 1>{
  int init() {
    csnd::Vector<MYFLT> &out =
        outargs.vector_data<MYFLT>(0);
    csnd::Vector<MYFLT> &in =
        inargs.vector_data<MYFLT>(0);
    out.init(csound, in.len());
    return OK;
  }

  int kperf() {
    csnd::Vector<MYFLT> &out =
        outargs.vector_data<MYFLT>(0);
    csnd::Vector<MYFLT> &in =
        inargs.vector_data<MYFLT>(0);
    std::copy(in.begin(), in.end(),
        out.begin());
    return OK;
  }
  };
```

This opcode is registered using the following line:

```
csnd::plugin<SimpleArray>(csound,
        "simple", "k[]", "k[]",
        csnd::thread::ik);
```

3.8 Multithreading

CPOF supports the creation of threads through the **Thread** pure virtual class. Developers wanting to avail of a subthread for processing can derive their own thread class from this and implement its `run()` method.

3.9 Constructing member variables

Plugin classes can, in general, be composed of member variables of any type, built in or user defined. However, we have to remember that opcodes are allocated and instantiated by C code, which does not know anything about classes. A member variable of a non-trivial class will not be constructed at instantiation. This is perfectly fine for all CPOF classes, which are designed to expect this. However, this might cause problems for other classes that are external to the framework. In this case, a placement **new** needs to be employed at init time to construct an object declared as a plugin class member (and thus allocated in Csound's heap). To facilitate matters, CPOF includes a template function that can be used to construct any member objects:

```
template <typename T,
   typename ... Types>
   T *constr(T* p, Types ... args){
      return new(p) T(args ...);
   }
```

For instance, let's say we have in our plugin an object of type `A` called `obj`. To construct this, we just place the following line in the plugin `init()` method:

```
csnd::constr(&obj,10,10.f);
```

where the arguments are the variable address, followed by any class constructor parameters. Note that if the class allocates any resources, we will need to invoke its destructor explicitly through **csnd::**`destr(&obj)` in a `deinit()` method.

4. BUILDING THE OPCODE LIBRARY

To build a plugin opcode library, we require a C++ compiler supporting the C++11 [5] standard (`-std=c++11`), and the Csound public headers. CPOF has no link dependencies (not even to the Csound library). The opcodes should be built as a dynamic/shared module (e.g `.so` on Linux, `.dylib` on OSX or `.dll` on Windows).

5. EXAMPLES FROM THE CSOUND SOURCES

CPOF is already being used in the development of new opcodes for Csound. It allows very compact and economical code, especially in conjunction with some of the more modern facilities of C++. For example, the following class

template is used to generate a whole family of numeric array-variable operators for i-time and k-rate processing [5]:

```
template<MYFLT (*op)(MYFLT)>
struct ArrayOp : csnd::Plugin<1, 1>{
 int oprt(csnd::myfltvec &out,
          csnd::myfltvec &in){
    std::transform(in.begin(),in.end(),
           out.begin(), [](MYFLT f){
                return op(f);});
    return OK;
  }
 int init() {
    csnd::myfltvec &out =
     outargs.myfltvec_data(0);
    csnd::myfltvec &in =
     inargs.myfltvec_data(0);
    out.init(csound,in.len());
    return oprt(out, in);
  }
 int kperf() {
  return oprt(outargs.myfltvec_data(0),
         inargs.myfltvec_data(0));
  }
};
```

A plugin implementing i-time $\cos(x)$ where x is an array is created with the following line:

```
csnd::plugin<ArrayOp<std::cos>>(csound,
 "cos", "i[]", "i[]",csnd::thread::i);
```

whereas for a k-rate $\exp(x)$ is implemented by:

```
csnd::plugin<ArrayOp<std::exp>>(csound,
 "exp", "k[]", "k[]",csnd::thread::ik);
```

Forty-six such operators are currently implemented reusing the same code. Another twelve use a similar class template for binary operations (2 inputs).

Another example shows the use of standard algorithms in spectral processing. The following new opcode implements *spectral tracing* [6], which retains only a given number of bins in each frame, according to their amplitude. To select the bins, we need to sort them to find out the ones we want to retain (the loudest N). For this we collect all amplitudes from the frame and then apply *nth element* sorting, placing the threshold amplitude in element n. Then we just filter the original frame according to this threshold. Here we have the performance code (`amps` is a dynamically allocated array belonging to the **`Plugin`** object).

```
int kperf() {
 csnd::pv_frame &fin =
      inargs.fsig_data(0);
 csnd::pv_frame &fout =
      outargs.fsig_data(0);

 if(framecount < fin.count()) {
  int n =  fin.len()-(int)inargs[1];
  float thrsh;
  std::transform(fin.begin(),
   fin.end(), amps.begin(),
   [](csnd::pv_bin f){
         return f.amp();});

  std::nth_element(amps.begin(),
         amps.begin()+n, amps.end());
  thrsh = amps[n];

  std::transform(fin.begin(),
   fin.end(),fout.begin(),
   [thrsh](csnd::pv_bin f){
          return f.amp() >= thrsh ?
           f : csnd::pv_bin(); });
   framecount =
    fout.count(fin.count());
 }
 return OK;
}
```

[5] The convenience typedef **`myfltvec`** for **`Vector`**<**`MYFLT`**> is employed here

6. CONCLUSIONS

This paper described CPOF and its fundamental characteristics. We looked at how the base classes are constructed, how to derive from them, and register new opcodes in the system. The framework is designed to support modern C++ idioms and adopts the C++11 standard. All of the code examples discussed in this paper are provided in `opcodes.cpp`, found in the `examples/plugin` directory of the Csound source codebase [6]. CPOF is part of Csound and is distributed alongside its public headers. Csound is free software, licensed by the Lesser GNU Public License.

7. REFERENCES

[1] V. Lazzarini, J. ffitch, S. Yi, J. Heintz, Ø. Brandtsegg, and I. McCurdy, *Csound: A Sound and Music Computing System*. Springer Verlag, 2016.

[2] B. Stroustrup, *The C++ Programming Language*, 4th ed. Addison-Wesley, 2013.

[3] D. Abrahams and A. Gurtovoy, *C++ Template Metaprogramming: Concepts, Tools, and Techniques from Boost and Beyond (C++ in Depth Series)*. Addison-Wesley Professional, 2004.

[4] V. Lazzarini, "Time-domain signal processing," in *The Audio Programming Book*, R. Boulanger and V. Lazzarini, Eds. MIT Press, 2010, pp. 463–512.

[5] ISO/IEC, "ISO international standard ISO/IEC 4882:2011, programming language C++," 2011. [Online]. Available: http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=50372

[6] T. Wishart, *Audible Design*. Orpheus The Pantomine, 1996.

[6] `https://github.com/csound/csound`