applied sciences

MDPI

Article

Supporting an Object-Oriented Approach to Unit Generator Development: The Csound Plugin Opcode Framework †

Victor Lazzarini

Music Department, Maynooth University, Maynooth W23 X021, Ireland; victor.lazzarini@mu.ie; Tel.: +353-1-708-6936

† This article is a re-written and expanded version of "The Csound Plugin Opcode Framework", SMC 2017, Espoo, Finland, 5 July 2017.

Academic Editor: Stefania Serafin
Received: 31 July 2017; Accepted: 18 September 2017; Published: 21 September 2017

Abstract: This article presents a new framework for unit generator development for Csound, supporting a full object-oriented programming approach. It introduces the concept of unit generators and opcodes, and its centrality with regards to music programming languages in general, and Csound in specific. The layout of an opcode from the perspective of the Csound C-language API is presented, with some outline code examples. This is followed by a discussion which places the unit generator within the object-oriented paradigm and the motivation for a full C++ programming support, which is provided by the Csound Plugin Opcode Framework (CPOF). The design of CPOF is then explored in detail, supported by several opcode examples. The article concludes by discussing two key applications of object-orientation and their respective instances in the Csound code base.

Keywords: computer music languages; musical signal processing; csound; sound synthesis; object-oriented programming; C++; code re-use; opcodes; unit generators

1. Introduction

All modern music programming systems provide means for extending their capabilities [1]. In most cases, this extensibility applies to allowing new *unit generators* (UGs) to be added to the system. A UG is defined as a component of the system programming language responsible for processing input and/or generating some output [2]. These may take the form of control or audio signals, messages of some kind, or single values. UGs are core elements of any MUSIC N-type language and they are are responsible for much of the processing power provided by the system. While some languages allow the user to process signal using only primitive operations, UGs provide in most cases a more convenient and efficient way to implement a given algorithm. The efficiency gain can be of various orders of magnitude, depending on the system and the kind of operations involved [3]. In some cases, UGs are the only means possible to realise a given algorithm, as the language in question is not designed to process signals directly or does not provide the required primitive operations to implement it (on a sample-level basis).

1.1. Csound Unit Generators

Csound [4,5], is a MUSIC N-type system, which consists of an audio engine, a music programming language, and an application programming interface (API). UGs in Csound are more commonly called *opcodes* or *functions*. While the system does not distinguish between these two, we often reserve the latter name for UGs with no side effects, which are pure functions [6]. Structurally, however, there is no distinction between these as far as their implementation layout is concerned, and we will employ

the term *opcode* more generally to denote a UG of any kind. Internally, all operations are implemented by opcodes/functions (including all primitive arithmetic and control-of-flow), which places them as a central pillar of the system. Some of these of course are going to be minimal and light-weight, and others can be of significant complexity.

Csound has a collection of internal or built-in opcodes that are compiled into the system, but also provides a mechanism for adding new opcodes [7]. This mechanism is made up by two separate components, namely

1. An interface for registering new opcodes, provided by Csound API.
2. Dynamic library loading is provided by the audio engine at startup. As part of this, a given directory is searched for suitable library files containing new opcodes.

Opcodes are usually written in C, although other languages producing a compatible binary format can be employed. Each opcode will be defined by a dataspace and up to three processing functions that are called by the engine at different times depending on the type of signals that are to be processed. An opcode is invoked by Csound code in units called *instruments* or *user-defined opcodes* (UDOs). The engine will instantiate it by allocating space for its data, and call its processing functions according to one or more of two action times (*passes*):

1. **initialisation time**: code is executed once, after instantiation. It can also be invoked again if a re-initialisation pass is requested.
2. **performance time**: code is executed repeatedly each control cycle to consume and/or produce a signal. There are two separate types of functions that can be defined for performance:
   (a) **control**: this function is invoked to deal with scalar inputs or outputs (e.g., processing a single sample at a time).
   (b) **audio**: code is expected to deal with audio signals, which are passed as vectors to it.

The three processing functions defined by an opcode are each connected to one of the processing cases listed above. They are also linked to the different types of variables supported by the Csound language, which the opcodes will be designed to operate on:

- `i`-type: these variables are floating-point scalars that can only change at initialisation or re-initialisation time.
- `k`-type: also floating-point scalars, these will only change at performance time, although they can also be initialised at i-time.
- `a`-type: this is a floating-point vector, which is modified at performance time. The length of this vector is defined by a `ksmps` variable that can assume local (instrument) values or can be set globally for the whole engine. Vectors can also be initialised at i-time. Audio-rate functions are designed to operate on these variables.
- `S`-type: character strings, which can be modified at i-time and perf-time, although it is more common to do so only at i-time.
- `f`-type: frequency-domain signals (fsigs), these contain self-describing spectral data (of different formats) that are processed at performance-time by control-rate functions.
- arrays: composite-type variables of each of the above types. A very important case is `k` and `i` arrays, for which there are various important applications.

An opcode is defined to operate on a given set of input and/or output argument types. Parameters passed to them have to match the pre-defined types. Multiple versions of the same opcode can be declared for different types, which is a case of *overloading*.

1.1.1. Opcode Layout

The Csound API mandates that opcodes should have the following components:

- A data structure declared with the following format. It always contains an `OPDS` member as its first component. This holds a set of elements common to all opcodes.

```
struct NAME {
OPDS h;
// output argument addresses
// input argument addresses
// dataspace members
};
```

  where we need to declare one pointer variable for each output and input argument (in the order they should occur in the Csound language code). When the opcode is called, the output and input argument addresses are passed to Csound through these pointers. The C variable types for different Csound argument types are:

  - `MYFLT*`: pointers to the internal floating-point data type (`MYFLT` is either defined as a 64 or a 32-bit float, depending on the engine version and platform) are used for all basic numeric arguments. (`i`, `k`, or `a`).
  - `STRINGDAT*`: used for string arguments.
  - `PVSDAT*`: holds an fsig argument
  - `ARRAYDAT*`: for array arguments (of any fundamental type).

- A function with the signature

```
int func(CSOUND *, void *);
```

  for each one of the required action times (init, control, and/or audio). The first argument is a pointer to the Csound engine that instantiated this opcode. The second argument receives a pointer to the allocated dataspace for a given instance of the opcode.

New opcodes are registered with the engine using the Csound C API function `csoundAppendOpcode()`,

```
int csoundAppendOpcode(CSOUND *csound, const char *opname,
                       int dsblksiz, int flags, int thread,
                       const char *outypes, const char *intypes,
                       int (*iopadr)(CSOUND *, void *),
                       int (*kopadr)(CSOUND *, void *),
                       int (*aopadr)(CSOUND *, void *));
```

This takes in the opcode name `opname`; the size of the dataspace `dsblksiz` in bytes; multithreading `flags` (normally 0); a `thread` code for the action times, which determines whether it should be active on i-time (1), k-rate (2), and/or a-rate (4); and the functions for i-time, k-rate, and a-rate (`NULL` if not needed), for an engine given by `csound`. Two strings, `outtypes` and `intypes`, define the output and input argument types expected by the opcode. Valid values are the characters `i`, `k`, `a`, `S`, `f`, with an added `[]` to indicate array arguments.

1.1.2. Plugin libraries

As discussed in Section 1.1, the Csound engine has a dynamic library loading mechanism that scans a given directory, the *opcode directory* (which can be defined by the `OPCODE6DIR` or `OPCODE6DIR64` environment variables) and loads any suitable files containing opcodes. Alternatively, Csound can be passed the option `--opcode-lib=` to load a given plugin library file at startup.

To recognise a dynamic library as containing Csound code, the engine looks for the definition of three functions:

```
csoundModuleCreate(CSOUND *csound);
csoundModuleInit(CSOUND *csound);
csoundModuleDestroy(CSOUND *csound);
```

If these functions exist, the library is loaded and `csoundModuleCreate()` followed by `csoundModuleInit()`. One of these functions should contain the call to the opcode registration API function to append the new UGs to the list kept by the engine.

To enable libraries to be built without the need to link directly to the Csound library, an `AppendOpcode()` function exists in the `CSOUND` structure allowing code to call the API function indirectly via the engine pointer. For example,

```
PUBLIC int csoundModuleInit(CSOUND *csound) {
  csound->AppendOpcode(csound, "test", sizeof(struct OPTEST),
                               0, 1, "i", "i", test_init, NULL, NULL);
  return 0;
}
```

registers an opcode with the name `test`, running at init-time only, implemented by the function `test_init`. This function invokes the exact same code as the Csound API call, but it is more convenient for the purposes of a plugin library.

1.1.3. Discussion

Csound UG development relies on the conventions outlined in Section 1.1.1, plus a comprehensive set of API functions provided in the `CSOUND` structure (including, as we saw in Section 1.1.2, the opcode-registering function `AppendOpcode`). These support a range of facilities and access to the engine that allow a complete scope for the development of new unit generators. However, it is the case that this interface can be more complex and cumbersome than necessary, owing to the characteristics of the C language.

From an object-oriented (OO) perspective [8], we can observe that an opcode might be described as a class that inherits from `OPDS`. It will contain a variable number of argument objects ($0 - N$), depending on its outputs and inputs. An opcode class can define up to three public methods, one for each action time required. These methods take as a parameter the Csound engine object, which has a large number of public methods that can be used for a variety of means: to retrieve engine attributes; to print messages; to handle MIDI data; to list and retrieve lists of arguments; to perform memory allocation and management; to access function tables; to perform FFTs and handle frequency-domain data; to access disk files; to generate random numbers; to spawn threads, manage locks and circular buffers; plus a number of other miscellaneous operations.

The Csound engine is responsible for constructing opcodes at instantiation. Effectively, this entails only the allocation of space for its data members, if no pre-allocated space exists already. Any further initialisation, if required, needs to be performed by the init-time method. When an instance of an opcode is no longer active, there is no automatic recovery of the memory space, and so new instances can take advantage of pre-existing space and skip allocation. However, memory can be recovered at certain stages explicitly if needed. Opcodes can also optionally register a destructor method with the engine, that will be called when an instance is no longer active.

From this analysis, we conclude that the structure of an opcode effectively takes an object-oriented form as far as it is possible under C. However, the language is not conducive to the application of techniques that would maximise code re-use and encapsulation, which would allow developers to concentrate directly on the implementation of their algorithms. Code re-use not only helps to save work in reproducing boilerplate lines, but also supports a more robust development, where a given functionality is implemented once and only once facilitating the task of ensuring correctness in the

code. Encapsulation allows for certain fundamental components to be hidden away and manipulated through a logic interface, which greatly supports developers in concentrating on the task at hand.

An ideal candidate language to support this type of approach is C++ in its more modern incarnations [9]. Unlike other languages such as Java, it is possible to compile code to a binary form that can be taken directly by the Csound dynamic loading mechanism, and it does not require a virtual machine environment to run. It is also very closely integrated with C, especially if we regard it as its superset. Differently to Objective-C, there is no major syntactic chasm between C and C++ code, and its new constructs have evolved organically as extensions to the original language.

1.2. Unit Generators and Plugins in Other Systems

As outlined in Section 1.1.3, the original Csound opcode C API imposes effectively a C-based OOP idiom, with limitations determined by the choice of language. A similar arrangement for UG development is present under Pure Data (PD) [10], where an essentially object-oriented structure is implemented in C. The API for plugins provides functions to register a new class (an UG in PD), as well as to register methods to respond to various messages that the system provides. Due to this message-passing nature of the PD engine, it is slightly more complex to add a new UG than it is in Csound. However, there are a number of similarities, including the presence of an audio processing method (corresponding to the performance-time audio function in Csound) and an object constructor (with similarities) to the init-time function. New classes are registered with the system in a similar way to Csound, in a given plugin entry point function. Thus, a C++ treatment similar to what is described for Csound in Sections 2–4 is also possible for PD.

The case of SuperCollider [11] UGs (ugens) is a little more mixed. The synthesis engine is written in C++, adopting as we would expect, a full OO approach. However, the ugen API uses a mostly C-oriented approach (not unlike Csound), with heavy use of macro substitution and depending on C linkage (`extern "C"`) [12]. This is highly surprising, since we would expect that it could use a more up-to-date idiom, enabled as it is by the C++ language. Unlike Csound or PD, there would be no limitations as to what the engine might be able to handle, since there is no language barrier to speak of. Additionally, new ugen registration requires a class definition in the SuperCollider language to be provided separately, which should match the C++ code. This is, of course, not needed either in Csound or PD, and it is an aspect that does not compare very well with these systems. However, in general, the OOP approach explored in this paper could also potentially be implemented to aid SuperCollider ugen development, without some of constraints imposed by the C language as discussed in Section 2.1.

Finally, as an example of a fuller use of C++ for plugin development, we have the VST framework [13]. Under this model, we have the use of a C++ class inheritance mechanism to define new plugins, and a simple plugin registration process based on a single function call at the plugin entry point. Equally, there are programming libraries that are implemented in C++ and employ an OO approach, such as STK [14] and SndObj [15], which are based on earlier standards of the language, and AuLib [16], which takes advantage of the latest, C++14 [17]. However, VST(and audio programming libraries in general) is not fully comparable to Csound (PD or SuperCollider) as a system. Thus, we can conclude that an OOP C idiom predominates in UG development for the most important music programming environments. A more modern approach might be more conducive to better programming practices that can in particular support the implementation of signal processing code. The achievement of this result is one of the main objectives of the present work.

2. The Framework

In order to support object-oriented programming (OOP) for unit generator development, a new framework development is developed from the ground up. The main objective of the work to provide an environment that is conducive to modern programming practices discussed in Section 1.1.3. The remainder of this paper will concentrate on describing the design and implementation of the Csound Plugin Opcode Framework (CPOF (To be pronounced *see-pough* or *cipó*, *vine* in Portuguese,

appropriated from the tupi-guarani word meaning "the hand of the branch")) [18], and the discussion of results. This framework provides an alternative for opcode programming that attempts to be thin, simple, complete, and that handles internal Csound resources in a safe way (using the same mechanisms provided by the underlying C interface).

An object-oriented *framework* is a type of library or API that supports the development of new classes through inheritance/sub-classing. This is set in contrast to a *toolkit* where existing classes are expected to be used through delegation or composition [19]. In this sense, CPOF is a framework with a small associated toolkit of support classes that encapsulate a number of key operations.

2.1. Design Fundamentals

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

1. The main Csound engine is written in C. As we have noted, it instantiates opcodes and makes calls to opcode processing functions, but it does not support any C++ facilities.
2. Polymorphism [20] via virtual functions [21] is not possible (due to 1) since C does not provide dynamic dispatch. All function calls have to be resolved at compile time.
3. The process of registering an opcode with the engine requires that processing functions are defined as static. As we have seen, up to three different functions should be registered (for different action times).
4. In C, the sub-classing of the base structure (`OPDS`) is achieved by adding this as its first member variable.

In relation to the last item above, we will assume a similar behaviour in C++. While defining an opcode base class for the framework, we have the practical expectation that all C++ compilers place subclass members contiguously to their superclass object in memory. Although this layout is not imposed by the C++ standard, it is the standard practice [22]. CPOF assumes then that the following code

```
struct OPCD {
  OPDS h;
};
```

is binary equivalent to

```
struct OPCD : OPDS {
};
```

The absence of a virtual function mechanism for overriding base class methods can be overcome with different compile time strategies. One of the possibilities for designing a framework based on polymorphism without the use of dynamic binding is to employ a method called *curiously recurring template pattern* (CRTP) [23].

However, we can do better with a much simpler approach. Given the constraints in which the opcode classes are meant to operate, 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 the scope is much narrower: the developer supplies the code, but will not call it directly (the functions are effectively callbacks). 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 dynamic binding sense) the base class ones. In this case, hiding plays the same role as overriding, there is in practice no distinction between the two. A plugin base class can be defined from which we will inherit to create the actual opcodes. This class will inherit from `OPDS` (which is opaque to CPOF) 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* (hidden, in reality) by derived class methods. An initial design for an opcode base class would thus be

```
struct Plugin : OPDS {
  // dataspace
   ...
  // stub methods
  int init() { return OK; }
  int kperf() { return OK; }
  int aperf() { return OK; }
};
```

from which we would inherit our own class to implement the new opcode:

```
struct MyClass: Plugin {
...
};
```

Given that in any practical applications 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 opcode 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:

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

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

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

In this case, `T` is our derived class, which implements the new opcode. Registration can then be also implemented using a template function

```
template <typename T>
int plugin(CSOUND *cs, const char *name, const char *oargs,
           const char *iargs, uint32_t thread, uint32_t flags = 0) {
  return cs->AppendOpcode(cs, (char *)name, sizeof(T), flags, thread,
                          (char *)oargs, (char *)iargs, (SUBR)init<T>,
                          (SUBR)kperf<T>, (SUBR)aperf<T>);
}
```

In this design, a class can be registered by instantiating and invoking the template function,

```
Plugin<MyOpcode>(...);
```

This call will be resolved at compile time with the requested class argument (`MyOpcode`). When running, Csound calls the template functions for processing, which in their turn delegate directly to the ones defined in the opcode class in question (If they are not defined there, the call will default to the non-op stub). Note that this is all hidden from the framework user (in the header file `plugin.h`), 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).

2.2. Opcode Arguments

To allow for a flexible handling of opcode output and input arguments, we can refine our earlier definition of an opcode base class. It is possible to take advantage of *non-type* (numeric) template variables to define the number of arguments for a class:

```
template <uint32_t N, uint32_t M> struct Plugin : OPDS { ... };
```

where N and M will define how many outputs and inputs an opcode will take, respectively, which are defined by the class declaration (template instantiation).

As we have seen in Section 1.1.1, opcode arguments can be of different pointer types, depending on the Csound variable types required. The `Param` class in CPOF is employed to encapsulate these, providing a general interface to arguments:

```
template <uint32_t N> class Param {
 MYFLT *ptrs[N];
 ...
};
```

In the base class, we declare two of these objects, `outargs` and `inargs` as its first two members (Figure 1):

```
template <uint32_t N, uint32_t M> struct Plugin : OPDS {
  Param<N> outargs;
  Param<M> inargs;
  ...
};
```

Note that this ensures a complete binary compatibility between the C-structure form of an opcode dataspace and the CPOF template base class (assuming the standard layout discussed in Section 2.1 is obeyed by the C++ compiler).

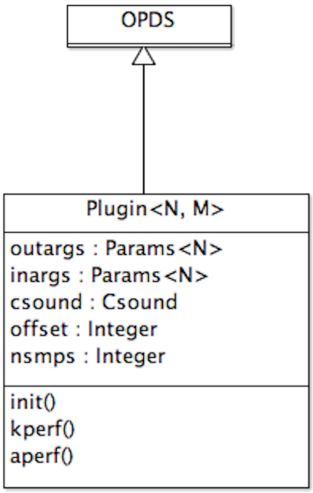

Figure 1. The `Plugin` template base class, derived from the opaque C structure `OPDS`.

2.3. The Base Class

In summary, this re-definition of the framework base class makes it a class template that needs to be instantiated by user code. To create a new opcode, we derive our own class by declaring the required number of output and inputs needs as template arguments (CPOF code uses the `csnd` namespace and is declared in the `plugin.h` header file):

```
#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. It will inherit the following members from `Plugin`:

- `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.
- `out_count()` and $in_count()$: these functions return the number of arguments for output and input, respectively. They are useful for opcodes with variable number of arguments.
- `sa_offset()`: this method calculates the correct values for `offset` and `nsmps`. User called does not need to invoke it, as it is called implicitly by the `aperf()` template function before it delegates to the plugin code.

As we have outlined in Section 1.1.1, Csound has two basic passes 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 (also known as k-period). 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.

2.3.1. Initialisation-time Opcodes

For init-time opcodes, all we need to do is provide an implementation of the `init()` method, as shown in Listing 1.

Listing 1: `i`-time opcode example

```
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`, which, as we have seen, is the internal floating-point type used in Csound.

2.3.2. Control-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, demonstrated by the code in Listing 2.

Listing 2: `k`-rate opcode example

```
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.

2.3.3. Audio-Rate Opcodes

For opcodes running only at a-rate (and with no init-time operation), we need to do provide an implementation of the `aperf()` method to process an audio vector (Listing 3).

Listing 3: `a`-rate opcode example

```
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.

2.4. Registering Opcodes with Csound

We have discussed in Section 2.1 how the opcode registration mechanism is implement through the CPOF function template `plugin()`. In order to use it, we just have to instantiate and invoke it with the required parameters. It signature is:

```
template <typename T>
```

```
int plugin(Csound *csound,
           const char *name,
           const char *oargs,
           const char *iargs,
           uint32_t thread,
           uint32_t flags = 0)
```

where we have the following arguments:

- `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 output argument types, one identifier per argument.
- `iargs`: a string containing the input argument types, one identifier per argument.
- `thread`: a code to tell Csound when the opcode should be active.
- `flags`: multithread flags (generally 0 unless the opcode accesses global resources).

For the argument type identifiers, we have seen in Section 1.1.1 that the most common types are: a (audio), k (control), i (i-time), S (string) and f (fsig). The *thread* argument, which defines what methods will be called by the opcode, can be defined by the following constants:

- `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()`.

CPOF supports the symbol `on_load()` as its entry point (Declared in the header file `modload.h`). This function needs to implemented only **once** per plugin library, and it should contain the calls to one registration function for each opcode to be added to the engine. For example, the three opcodes defined in Section 2.3 can be registered as shown in Listing 4.

Listing 4: Registering opcodes with Csound

```
#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 CSOUND_OK;
}
```

These calls 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.

2.5. Constructing and Destroying Member Variables

As opcode classes are instantiated by the Csound engine through C-language code, constructors for member variables are not invoked automatically. For member variables of non-trivial types, this may pose an issue, especially if there are specific initialisation steps to be performed at construction. All classes in the toolkit have no such requirements and do not declare constructors, but external code from other libraries might do. For these, CPOF provides a mechanism to call the member variable constructor explicitly. This is based on the use of a placement `new` via a function template, which is used to access the class constructor for an object:

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

As an example, consider an object of type `A` called `obj`, which needs to be constructed explicitly, using

```
A::A(int, float) { ... };
```

To invoke it, we should place the following call in the `init()` method of our opcode class:

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

where the arguments are the variable address, followed by any class constructor parameters. Again, given that the compiler knows that `obj` is of type `A`, it resolves the template without the need for an explicit type instantiation.

It is also important to note that if the object constructed in this form allocates any resources dynamically, we will need to free these. For this we are required to call the object destructor explicitly by using another function template, defined in CPOF as

```
template<typename T> void destr(T *p) {
  p->T::~T();
}
```

The call to the destructor should be issued at the point where we no longer need the object. For opcodes that run at perf-time, this is normally done in the opcoe `deinit()` method. For example, to clean up a member variable `obj`, we implement the following code:

```
int deinit() {
    csnd::destr(&obj);
    return OK;
}
```

3. The Engine Object

As noted in Section 1.1.1, the Csound API provides a large range of facilities to opcodes through several functions provided in the CSOUND data structure, which is made opaque to CPOF. In fact,

since the API is designed to cater for a wider variety of applications beyond extending the language, many of these functions are not designed for use in UGs. For this reason, to provide a clearer interface for opcode programming, CPOF encapsulates the engine into an object that exposes only the relevant methods to the user.

All opcodes are run inside the Csound engine, represented by the `Csound` class (Figure 2). As we have seen above, the `Plugin` class holds a pointer of this type, which can be used to access the various utility methods provided by the engine. The following are the public methods of the Csound class in each category:

- Messaging:
  - `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.
- System parameters:
  - `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.
  - `is_asig()`: check for an audio signal argument.
- MIDI data access:
  - `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()`: channel aftertouch.
  - `midi_chn_polytouch()`: polyphonic aftertouch.
  - `midi_chn_ctl()`: continuous control value.
  - `midi_chn_pitchbend()`: pitch bend data.
  - `midi_chn_list()`: list of active notes for this channel.
- FFT:
  - `fft_setup()`: FFT operation setup.
  - `rfft()`: real-to-complex, complex-to-real FFT.
  - `fft()`: complex-to-complex FFT.
- Memory allocation (Csound-managed heap):
  - `malloc()`: malloc-style memory allocation.
  - `calloc()`: calloc-style memory allocation.
  - `realloc()`: realloc-style memory allocation.
  - `strdup()`: string duplication.
  - `free()`: memory deallocation.

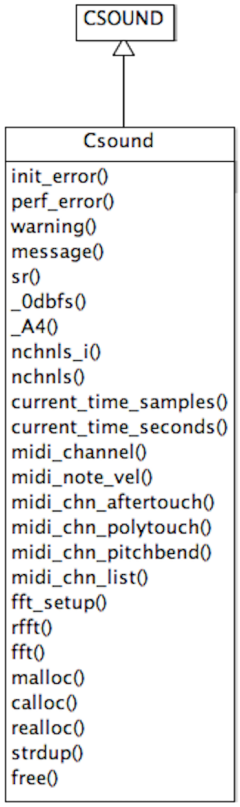

Figure 2. The `Csound` engine class, derived from the opaque C structure `CSOUND`.

In addition to these, the Csound class also holds a deinit method registration function template that can be used by opcodes to implement housekeeping tasks.

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

This is only required if the `Plugin`-derived class has allocated extra resources using mechanisms that require explicit clean-up. It is not need in most cases, as we will see in our examples. To use it, the plugin needs to declare and 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);
```

Because of the presence of the opcode object as an argument, the compiler resolves the template instantiation without requiring an explicit template parameter.

4. Toolkit Classes

Plugins developed with CPOF can avail of a number of helper classes that compose its *toolkit*. These include the aforementioned `Params` class, as well as classes for resource allocation, input/output access/manipulation, threads, and support for constructing objects allocated in Csound's heap.

4.1. Parameters

As we have already discussed in Section 2.2, parameters passed to a Csound opcode instance are encapsulated by the `Params` class so that they can be conveniently accessed. The class has the following public methods:

- `operator[]()`: array-style access to numeric (scalar) parameter values.
- `begin(), cbegin()`: begin iterators for the parameter list.

- `end(), cend()`: end iterators.
- `iterator` and `const_iterator`: iterator types for this class.
- `operator()()`: function-style access to numeric (vector) parameter pointers.
- `data()`: same as the function operator, access to the parameter address.
- `str_data()`: access to parameter as a `STRINGDAT` reference (see Section 4.5).
- `fsig_data()`: access to parameter as a `Fsig` reference (fsig data class, see Section 5).
- `vector_data()`: access to parameter as a `Vector<T>` reference (Csound 1-D numeric array data, see Section 4.6).
- `myfltvec_data()`: access to parameter as a `myfltvec` reference (Csound 1-D numeric array, see Section 4.6).

As we can see, this is is a thin wrapper over the argument pointers, which translates between the original `MYFLT*` and the various argument types, and allows for iteration over the parameter lists.

4.2. Audio Signals

As outlined in Section 1.1.1, audio signal variables are vectors of `nsmps` samples and we can access them through raw `MYFLT` pointers from input and output parameters. While this works well in a C-language environment, it is possible to provide a better object-oriented support to the manipulation of vectors through encapsulation. The `AudioSig` class wraps audio signal vectors conveniently, providing iterators and subscript access:

- `operator[]()`: array-style access to individual samples.
- `begin(), cbegin()`: begin iterators for the audio vector.
- `end(), cend()`: end iterators.
- `iterator` and `const_iterator`: iterator types for this class.
- `operator()()`: function-style access to numeric (vector) parameter pointers.

Objects are constructed by passing the current plugin pointer (`this`) and the raw parameter pointer. The final parameter is flag for an optional resetting of the audio signal vector:

```
AudioSig(OPDS *p, MYFLT *s, bool res = false);
```

With this, we can re-write the `simple` audio example opcode to use this class and its iterators in a typical C++ idiom, as demonstrated in Listing 5.

Listing 5: `a`-rate opcode example, using `AudioSig` objects

```
struct Simplea : csnd::Plugin<1,1> {
 int aperf() {
  csnd::AudioSig in(this, inargs(0));
  csnd::AudioSig out(this, outargs(0));
  std::copy(in.begin(), in.end(), out.begin());
  return OK;
 }
};
```

4.3. Memory Allocation

Csound provides its own managed heap for dynamic memory allocation. The engine provides mechanisms to allocate space as required. This ensures that there are no leaks and that there is an efficient use of resources. When an opcode requires a certain amount of space that is not known at compile time, it can avail of this mechanism to get access to it.

It is not advisable for developers to employ any other memory allocation methods. In C, this means that standard library functions `malloc` etc should be avoided. In C++, we should avoid to use the new operator (`new`) and containers that employ it (for instance, `std::vector`). They are more difficult to

integrate and use properly in this environment, especially given the fact that we are operating under in a hybrid space that includes a C platform supporting the C++ code.

The main mechanism for memory allocation in opcodes is provided by the `AuxAlloc()` function in the Csound API. This is encapsulated by the helper class `AuxMem` in CPOF, which allows an object-oriented approach to memory manipulation. This class has the following methods:

- `allocate()`: allocates new memory whenever 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()`: begin iterators to the data memory.
- `end(), cend()`: end iterators.
- `iterator` and `const_iterator`: iterator types for this class.

In Listing 6, the `DelayLine` class implements a simple comb filter [24] with three parameters (audio input, `i`-time delay time, and `k`-rate feedback amount. It demonstrates the use of the `AuxMem` template class, which holds the delay memory for the opcode.

Listing 6: Delay line opcode example

```
struct DelayLine : csnd::Plugin<1,3> {
 static constexpr char const *otypes = "a";
 static constexpr char const *itypes = "aik";

 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));
  MYFLT g = inargs[2];

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

In this example, we use an `AuxMem` iterator to access the delay vector, in a typical C++ idiom. The delay line access is implemented via a lambda that captures he opcode dataspace and processes every sample of the input producing the output vector. While this example uses an iterator for convenience, 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 can use

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

because the output and input types have already been declared in the class as the compile-time constants `itypes` and `otypes`.

In addition to the automatic `AuxMen` mechanism, Csound also offers the more conventional `malloc`-style allocation. This is not generally used in opcode development, but it is accessible via methods of the `Csound` engine object. It requires explicit de-allocation of resources on clean-up.

4.4. Function Table Access

Function tables are used by Csound for a variety of applications. They hold an array of floating-point numbers that is normally created by one of the GEN routines offered by the system [4]. These can generate tables based on trigonometric functions, polynomials, envelopes, windows, and other various mathematical formulae. Function tables are essential for many opcodes, such as oscillators, granular generators, waveshapers, and various different types of processors.

The Csound C API offers access to function tables via a `FUNC` structure. In CPOF, this access is 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()`: iterators to the beginning of the function table.
- `end(), cend()`: end iterators.
- `iterator` and `const_iterator`: iterator types for this class.

A typical usage example is given by the table-lookup oscillator algorithm [25]. In listing 7, the `Oscillator` class implements truncating lookup using a C++11 range-for facility.

Listing 7: Table-lookup oscillator opcode example

```
struct Oscillator : csnd::Plugin<1,3> {
 static constexpr char const *otypes = "a";
 static constexpr char const *itypes = "kki";
 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;
  double ph = x;

  for(auto &s : out) {
    s = amp * tab[(uint32_t) ph];
```

```
    ph += si;
    while (ph < 0) ph += tab.len();
    while (ph >= tab.len()) ph -= tab.len();
  }
  x = ph;
  return OK;
 }
};
```

A table is initialised by passing the relevant argument pointer to it (using its `data()` method). This will hold the function table number that is passed to opcode. In this example, as we need a precise index value, it is more convenient to use array index access instead of iterators, although these are also available in the class. This opcode is registered by

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

4.5. String Types

String variables in Csound are held in a `STRINGDAT` data structure, containing two members, a pointer to the address holding the zero-terminated string, and a size:

```
typedef struct {
    char *data;
    int size;
} STRINGDAT;
```

The size parameter contains the total space currently allocated for the string, which might be larger than the actual string. 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> {
 static constexpr char const *otypes = "";
 static constexpr char const *itypes = "S";
 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",  csnd::thread::i);
```

4.6. 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 template class, `Vector<typename T>`. This currently supports only one-dimensional arrays directly, but can be used with all basic Csound variable types.

This container class is derived from `ARRAYDAT` and wraps an array argument of a type defined by its template parameter. Input variables of these types are already properly initialised, but outputs need to be initialised with a given array size. The class 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()`: iterators to the beginning and end of the vector.
- `end()`, `cend()`: end iterators.
- `iterator` and `const_iterator`: iterator types for this class.
- `data_array()`: returns a pointer to the vector data address.

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 in Listing 8, implementing both `i`-time and `k`-rate array operations.

Listing 8: `i`-time and `k`-rate array opcode example

```
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());
    std::copy(in.begin(), in.end(), out.begin());
    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

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

for `i`-time operation and

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

for perf-time processing. This is an example of an *overloaded* opcode, that can operate on `i` and `k`-type variables. To facilitate the manipulation of this more common type of array, based on `MYFLT`, CPOF defines the following type:

```
typedef Vector<MYFLT> myfltvec;
```

5. Streaming Spectral Processing

Csound has a very useful mechanism for *streaming* (as opposed to *batch*) spectral processing, which is based on its fsig data type [26]. This is a self-describing type, which holds a frame of spectral data in one of several formats, defined by the C data structure PVSDAT (This C structure is provided here for reference only, it is opaque to CPOF):

```
typedef struct pvsdat {
   int32   N;
   int32  sliding;
   int32  NB;
```

```
    int32  overlap;
    int32  winsize;
    int32   wintype;
    int32  format;
    uint32  framecount;
    AUXCH  frame;
} PVSDAT;
```

This structure contains information on the DFT size used in the analysis, the window and hop sizes, the window type, the data format, and the current frame count. It also sets the analysis mode (normal or sample-by-sample sliding, where the actual hopsize is 1). The actual frame data is stored in an `AUXCH` memory structure, which is managed by Csound.

An opcode implementing fsig processing will operate nominally at the control rate, but will actually compute new frames at a rate determined by the analysis hopsize. This is implemented by checking the fsig `framecount` and only proceeding to consume and produce new frames if this is greater than the opcode internal frame counter. For streaming spectral processing opcodes, CPOF provides a separate base class, `FPlugin`, derived from `Plugin`, with an extra member variable, `framecount`, used for this purpose.

To facilitate fsig manipulation, the toolkit provides the `Fsig` class, derived from `PVSDAT`, with the following methods:

- `init()`: initialises an fsig. There are two overloads: it is possible to initialise it from individual parameters (DFT size, hop size, etc.) or directly from another fsig. Initialisation is only needed for output variables.
- `dft_size()`: DFT size.
- `hop_size()`: hopsize.
- `win_size()`: window size.
- `win_type()`: returns the window type (Hamming = 0, von Hann =1, Kaiser = 2, custom = 3, Blackman = 4 and 5, Nutall = 6, Blackman-Harris = 7 and 8, rectangular = 9).
- `nbins()`: number of bins.
- `count()`: current frame count.
- `isSliding()`: checks for sliding mode.
- `fsig_format()`: returns the data format. This will vary depending on the source. To facilitate identification, CPOF defines the following constants:
  - `fsig_format::pvs`: standard phase vocoder frame, composed of bins containing amplitude and frequency pairs.There are $N/2 + 1$ bins (N is the DFT frame size), equally spaced between between 0 Hz and the Nyquist frequency (inclusive).
  - `fsig_format::polar`: similar to the pvs type, except that bins contain pairs of magnitude and phase data.
  - `fsig_format::complex`: as above, but with bins holding complex-format data (real, imaginary).
  - `fsig_format::tracks`: this format holds tracks of amplitude, frequency, phase, and track ID (used in partial tracking opcodes).

Phase Vocoder Data

The most common format used in Csound opcodes is phase vocoder amplitude-frequency [25,27]. To provide a convinient access access to bins, a container interface is provided by `pv_frame` (`spv_frame` for the sliding mode (`pv_frame` is a convenience typedef for `Pvframe<pv_bin>`), whereas `spv_frame` is `Pvframe<spv_bin>`). This is a class derived from `Fsig` that can hold a series of `pv_bin` (`spv_bin` for sliding (`pv_bin` is `Pvbin<float>` and `spv_bin` is `Pvbin<MYFLT>`)) objects, which have the following methods (Figure 3):

- `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.

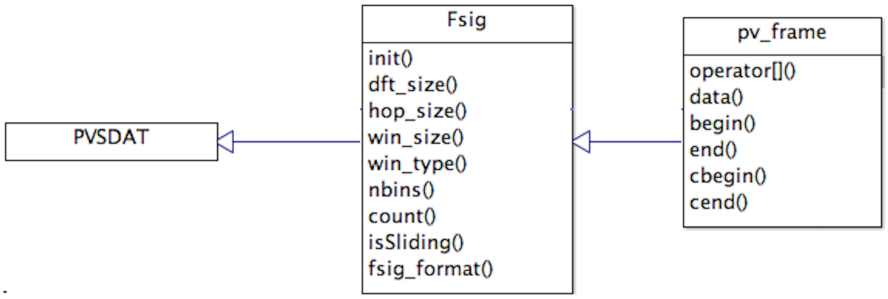

Figure 3. The `Fsig` and `pv_frame` classes and their relationship to the opaque C structure `PVSDAT`.

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 `pv_frame` (or `spv_frame`) class contains the following members:

- `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.

As noted above, 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 example in Listing 9 shows a class implementing a simple gain scaler for fsigs:

Listing 9: Fsig opcode example

```
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())
   return csound->init_error("sliding not supported");

  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;
  }
};
```

The `Params` class offers a dedicated method that is used on the arguments objects to get references to the Fsig parameters. This can also be assigned directly to a `pv_frame` reference variable. At init-time, we initialise the output based on the input fsig format. At performance time, we check the input count and process the data if necessary. The facilities offered by CPOF allow us to use a standard library transform algorithm with a lambda object and implement the gain processing very compactly. This opcode is registered using

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

6. Multithreading Support

The Csound API includes an interface for multithreading, which is implemented via pthreads [28] on POSIX systems, or other native threading libraries in non-POSIX platforms. To allow opcodes an object-oriented access to this C interface, CPOF provides the `Thread` pure virtual class. This is subclassed and instantiated to encapsulate a separate thread whose entry point is given by a `run()` method. The base class provides `join()` and `get_thread()` methods for joining a thread and getting its handle.

The example in Listing 10 illustrates the use of the `Thread` class. This implements a message printer on separate thread, picking up a string from the input and outputting it to the terminal. We derive a class that includes the `run()` method. To prevent data races, the class provides its own spin locks. A message is passed to it via the `set_messsage()` method.

Listing 10: Deriving a class from `Thread`

```
class PrintThread : public csnd::Thread {
  std::atomic_bool splock;
  std::atomic_bool on;
  std::string message;

  void lock() {
    bool tmp = false;
    while(!splock.compare_exchange_weak(tmp, true))
      tmp = false;
  }
```

```
  void unlock() {
    splock = false;
  }

  uintptr_t run() {
    std::string old;
    while(on) {
      lock();
      if(old.compare(message)) {
       csound->message(message.c_str());
       old = message;
      }
      unlock();
    }
    return 0;
  }

public:
  PrintThread(csnd::Csound *csound)
    : Thread(csound), splock(false), on(true), message("") {};

  ~PrintThread(){
    on = false;
    join();
  }

  void set_message(const char *m) {
    lock();
    message = m;
    unlock();
  }
};
```

The opcode class then is composed with it, as demonstrated in Listing 11. It will instantiate the object by calling its constructor and then pass messages to it from the performance method.

Listing 11: Threading opcode example

```
struct TPrint : csnd::Plugin<0, 1> {
  static constexpr char const *otypes = "";
  static constexpr char const *itypes = "S";
  PrintThread thread;

  int init() {
    csound->plugin_deinit(this);
    csnd::constr(&thread, csound);
    return OK;
  }

  int deinit() {
    csnd::destr(&thread);
```

```
    return OK;
  }

  int kperf() {
    thread.set_message(inargs.str_data(0).data);
    return OK;
  }
};
```

7. Results

As we have noted, CPOF supports a fully object-oriented approach to the development of new Csound opcodes, which is the main result of this work. In this section, we first oppose two versions of the same unit generator, the first using the original C API and the second based on CPOF. This demonstrates the advantages of applying the framework in the implementation of the opcode. To complement this, drawing from examples in the Csound codebase, we highlight two particular aspects of object-oriented programming that are enabled by CPOF: code re-use and the application of standard algorithms. Concluding this section, the overall contribution of this work to Csound and audio programming in general is discussed.

7.1. CPOF versus C API

As a way of comparing the CPOF approach with the original C API, we will focus on re-implementing an existing opcode using the framework. We have chose `tone`, a simple first-order lowpass filter, which provides a simple code that is easy to follow, but also demonstrates the compactness of the C++ approach. In Listing 12, we present the original code, taken from the Csound sources and adapted as a plugin. While the original code is actually an internal opcode, for the sake of making an exact comparison, we present it here in a modified version as an externally-loaded plugin. The only changes made to the original code are to do with opcode registration.

Listing 12: Tone opcode, original C API version

```
typedef struct {
   OPDS    h;
   MYFLT   *ar, *asig, *khp, *istor;
   double  c1, c2, yt1, prvhp;
} TONE;

int tonset(CSOUND *csound, TONE *p)
{
    double b;
    p->prvhp = (double)*p->khp;
    b = 2.0 - cos((double)(p->prvhp * TWOPI/csound->GetSr()));
    p->c2 = b - sqrt(b * b - 1.0);
    p->c1 = 1.0 - p->c2;

    if (LIKELY(!(*p->istor)))
      p->yt1 = 0.0;
    return OK;
}
```

```
int tone(CSOUND *csound, TONE *p)
{
    MYFLT       *ar, *asig;
    uint32_t offset = p->h.insdshead->ksmps_offset;
    uint32_t early  = p->h.insdshead->ksmps_no_end;
    uint32_t n, nsmps = CS_KSMPS;
    double      c1 = p->c1, c2 = p->c2;
    double      yt1 = p->yt1;

    if (*p->khp != (MYFLT)p->prvhp) {
      double b;
      p->prvhp = (double)*p->khp;
      b = 2.0 - cos((double)(p->prvhp * TWOPI/csound->GetSr()));
      p->c2 = c2 = b - sqrt(b * b - 1.0);
      p->c1 = c1 = 1.0 - c2;
    }
    ar = p->ar;
    asig = p->asig;
    if (UNLIKELY(offset)) memset(ar, '\0', offset*sizeof(MYFLT));
    if (UNLIKELY(early)) {
      nsmps -= early;
      memset(&ar[nsmps], '\0', early*sizeof(MYFLT));
    }
    for (n=offset; n<nsmps; n++) {
      yt1 = c1 * (double)(asig[n]) + c2 * yt1;
      ar[n] = (MYFLT)yt1;
    }
    p->yt1 = yt1;
    return OK;
}

int csoundModuleInit(CSOUND *csound) {
  csoundAppendOpcode(csound, "tone", sizeof(TONE), 0, 5, "a", "ako",
                     (SUBR) toneset, NULL, (SUBR) tone);
  return OK;
};

int csoundModuleCreate(CSOUND *csound) { return OK; };
int csoundModuleDestroy(CSOUND *csound) { return OK; };
```

This code example follows the straight C API opcode implementation: a data structure is provided, along with init-time and audio-rate perf-time functions. The opcode is registered using `csoundAppendOpcode()` called in `csoundModuleInit()`. Looking at the code, besides the typical C idiom, we see that there are a number of lines of code devoted to saving to the dataspace, setting sample-accurate parameters (`offset`, `nsmps`), and so on.

In contrast, the CPOF version, in Listing 13, is much more succinct. The functions have direct access to the dataspace, which allows them to update state directly. The use of iterators allow us to replace the loop with a single function call containing a lambda expression. The attention of the programmer is directed to the actual filter equation implemented therein. The code is made more compact also by replacing the filter update lines by inline calls to the `update()` method. All the

boilerplate code present in the C version is hidden away by the framework. Finally, opcode registration is much simplified by the use of the function template.

Listing 13: Tone opcode, CPOF version

```
struct Tone : csnd::Plugin <1, 3> {
  static constexpr char const *otypes = "a";
  static constexpr char const *itypes = "aio";
  double c1;
  double c2;
  double yt1;
  double prvhp;

  void update() {
    prvhp = (double) inargs[1];
    double b = 2.0 - cos(prvhp*csnd::twopi/csound->sr());
    c2 = b - sqrt(b * b - 1.0);
    c1 = 1.0 - c2;
  }

  int init() {
    update();
    if (!inargs[2]) yt1 = 0.;
    return OK;
  }

  int aperf() {
    csnd::AudioSig in(this, inargs(0));
    csnd::AudioSig out(this, outargs(0));
    double y = yt1;
    if (prvhp != inargs[1]) update();
    std::transform(in.begin(), in.end(), out.begin(),
                   [this, &y](MYFLT x) {
                     return (y = c1 * x + c2 * y);
                   });
    yt1 = y;
    return OK;
  }
};

void csnd::on_load(Csound *csound) {
 csnd::plugin<Tone>(csound, "tone", csnd::thread::ia);
};
```

While detailed performance considerations are beyond the scope of this paper, it is nonetheless useful to observe how the two versions of this simple opcode compare in that respect. Using standard timing tests, it was found that the elapsed CPU times of both versions are very close, as illustrated in Table 1 for various vector size (ksmps) values.

Table 1. CPU time ratio C:C++ of versions of the opcode `tone`. These results are an average of five runs per test, each processing 600 seconds of audio at $f_s = 44100$ KHz. Tests were run on the x86_64 architecture under MacOS and the clang compiler.

ksmps	1	2	4	8	16	32	64	128
CPU time	0.996369	0.995616	0.982088	0.975774	0.966242	1.000936	0.960440	0.975930

7.2. Code Re-Use

With CPOF, massive code re-use can be applied to generate whole families of opcodes. For example, the following class template is used to generate a set of numeric array-variable operators for i-time and k-rate processing. This is based on creating a template opcode class that can be instantiated with different functions that have the same signature. The class in Listing 14 implements it for single argument operators.

Listing 14: Single-argument function class template

```
template <MYFLT (*op)(MYFLT)> struct ArrayOp : csnd::Plugin<1, 1> {
  int process(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 process(out, in);
  }

  int kperf() {
    return process(outargs.myfltvec_data(0), inargs.myfltvec_data(0));
  }
};
```

We can instantiate it with a huge variety of one-in one-out functions from the standard library, thus creating over forty new opcodes. This is done by just registering each opcode using the same `ArrayOp` class with a different template parameter, as demonstrated in Listing 15.

Listing 15: Instantiating the array opcodes and registering them with Csound

```
void csnd::on_load(Csound *csound) {
  csnd::plugin<ArrayOp<std::ceil>>(csound, "ceil", "i[]", "i[]",
                                   csnd::thread::i);
  csnd::plugin<ArrayOp<std::ceil>>(csound, "ceil", "k[]", "k[]",
                                   csnd::thread::ik);
  csnd::plugin<ArrayOp<std::floor>>(csound, "floor", "i[]", "i[]",
                                    csnd::thread::i);
  csnd::plugin<ArrayOp<std::floor>>(csound, "floor", "k[]", "k[]",
                                    csnd::thread::ik);
  csnd::plugin<ArrayOp<std::round>>(csound, "round", "i[]", "i[]",
                                    csnd::thread::i);
  csnd::plugin<ArrayOp<std::round>>(csound, "round", "k[]", "k[]",
```

```
                                    csnd::thread::ik);
  csnd::plugin<ArrayOp<std::trunc>>(csound, "int", "i[]", "i[]",
                                    csnd::thread::i);
  csnd::plugin<ArrayOp<std::trunc>>(csound, "int", "k[]", "k[]",
                                    csnd::thread::i);
  csnd::plugin<ArrayOp<frac>>(csound, "frac", "i[]", "i[]",
                              csnd::thread::i);
  csnd::plugin<ArrayOp<frac>>(csound, "frac", "k[]", "k[]",
                              csnd::thread::ik);
  csnd::plugin<ArrayOp<std::exp2>>(csound, "powoftwo", "i[]", "i[]",
                                   csnd::thread::i);
  csnd::plugin<ArrayOp<std::exp2>>(csound, "powoftwo", "k[]", "k[]",
                                   csnd::thread::ik);
  csnd::plugin<ArrayOp<std::fabs>>(csound, "abs", "i[]", "i[]",
                                   csnd::thread::i);
  csnd::plugin<ArrayOp<std::fabs>>(csound, "abs", "k[]", "k[]",
                                   csnd::thread::ik);
  csnd::plugin<ArrayOp<std::log10>>(csound, "log2", "i[]", "i[]",
                                    csnd::thread::i);
  csnd::plugin<ArrayOp<std::log10>>(csound, "log2", "k[]", "k[]",
                                    csnd::thread::ik);
  csnd::plugin<ArrayOp<std::log10>>(csound, "log10", "i[]", "i[]",
                                    csnd::thread::i);
  csnd::plugin<ArrayOp<std::log10>>(csound, "log10", "k[]", "k[]",
                                    csnd::thread::ik);
  csnd::plugin<ArrayOp<std::log>>(csound, "log", "i[]", "i[]",
                                  csnd::thread::i);
  csnd::plugin<ArrayOp<std::log>>(csound, "log", "k[]", "k[]",
                                  csnd::thread::ik);
  csnd::plugin<ArrayOp<std::exp>>(csound, "exp", "i[]", "i[]",
                                  csnd::thread::i);
  csnd::plugin<ArrayOp<std::exp>>(csound, "exp", "k[]", "k[]",
                                  csnd::thread::ik);
  csnd::plugin<ArrayOp<std::sqrt>>(csound, "sqrt", "i[]", "i[]",
                                   csnd::thread::i);
  csnd::plugin<ArrayOp<std::sqrt>>(csound, "sqrt", "k[]", "k[]",
                                   csnd::thread::ik);
  csnd::plugin<ArrayOp<std::cos>>(csound, "cos", "i[]", "i[]",
                                  csnd::thread::i);
  csnd::plugin<ArrayOp<std::cos>>(csound, "cos", "k[]", "k[]",
                                  csnd::thread::ik);
  csnd::plugin<ArrayOp<std::sin>>(csound, "sin", "i[]", "i[]",
                                  csnd::thread::i);
  csnd::plugin<ArrayOp<std::sin>>(csound, "sin", "k[]", "k[]",
                                  csnd::thread::ik);
  csnd::plugin<ArrayOp<std::tan>>(csound, "tan", "i[]", "i[]",
                                  csnd::thread::i);
  csnd::plugin<ArrayOp<std::tan>>(csound, "tan", "k[]", "k[]",
                                  csnd::thread::ik);
  csnd::plugin<ArrayOp<std::acos>>(csound, "cosinv", "i[]", "i[]",
```

```
                                    csnd::thread::i);
  csnd::plugin<ArrayOp<std::acos>>(csound, "cosinv", "k[]", "k[]",
                                    csnd::thread::ik);
  csnd::plugin<ArrayOp<std::asin>>(csound, "sininv", "i[]", "i[]",
                                    csnd::thread::i);
  csnd::plugin<ArrayOp<std::asin>>(csound, "sininv", "k[]", "k[]",
                                    csnd::thread::ik);
  csnd::plugin<ArrayOp<std::atan>>(csound, "taninv", "i[]", "i[]",
                                    csnd::thread::i);
  csnd::plugin<ArrayOp<std::atan>>(csound, "taninv", "k[]", "k[]",
                                    csnd::thread::ik);
  csnd::plugin<ArrayOp<std::cosh>>(csound, "cosh", "i[]", "i[]",
                                    csnd::thread::i);
  csnd::plugin<ArrayOp<std::cosh>>(csound, "cosh", "k[]", "k[]",
                                    csnd::thread::ik);
  csnd::plugin<ArrayOp<std::sinh>>(csound, "sinh", "i[]", "i[]",
                                    csnd::thread::i);
  csnd::plugin<ArrayOp<std::sinh>>(csound, "sinh", "k[]", "k[]",
                                    csnd::thread::ik);
  csnd::plugin<ArrayOp<std::tanh>>(csound, "tanh", "i[]", "i[]",
                                    csnd::thread::i);
  csnd::plugin<ArrayOp<std::tanh>>(csound, "tanh", "k[]", "k[]",
                                    csnd::thread::ik);
  csnd::plugin<ArrayOp<std::cbrt>>(csound, "cbrt", "i[]", "i[]",
                                    csnd::thread::i);
  csnd::plugin<ArrayOp<std::cbrt>>(csound, "cbrt", "k[]", "k[]",
                                    csnd::thread::ik);
}
```

A similar approach can be used for functions of two arguments, yielding yet another large set of new opcodes. While this is a simple and indeed obvious example that can deliver re-use in a large scale, more generally the framework reduces the amount of code repetition significantly.

7.3. Standard Algorithms

Another example shows the use of standard algorithms in spectral processing. The opcode in Listing 16 implements *spectral tracing* [29], 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).

Listing 16: Phase vocoder tracing opcode

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

  int init() {
    if (inargs.fsig_data(0).isSliding())
      return csound->init_error(Str("sliding not supported"));
```

```
    if (inargs.fsig_data(0).fsig_format() != csnd::fsig_format::pvs &&
        inargs.fsig_data(0).fsig_format() != csnd::fsig_format::polar)
      return csound->init_error(Str("fsig format not supported"));

    amps.allocate(csound, inargs.fsig_data(0).nbins());
    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);

    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;
  }
};
```

7.4. Discussion

One of the main objectives of this work, as stated earlier, is to provide support for more modern C++ approaches to opcode development, which can, in particular, facilitate the writing of signal processing code. Under this point, we were able to show through the comparison of CPOF and C API code that such objective has been met. A cursory comparison of the two versions demonstrates a number of surface differences in the CPOF version: economy of expressions, code length (36 lines vs. 59), and clarity of context. A closer look will reveal how the semantics of the process (the application of a filter equation) is far better realised in the C++ version, without the intrusion of set-up and boilerplate code.

In fact, this comparative exercise points out to the fact that a lot of code re-use could actually replace the traditional approach to write components such as filters. We were able to demonstrate how this is possible in a simple case of stateless operators, but the principle is applicable to other types of opcodes, of varying levels of complexity. In the case of filters, by supplying different `update()` and filter lambdas to a template class, a whole family of such processors can be implemented. Beyond this extensive code re-use, we can also make avail of standard algorithms for these tasks, which not only simplifies some of the development stages, but also helps improve correctness of implementation.

We may conclude that the availability of CPOF in the Csound code allows for better engineering practices in the development of opcodes. A corollary of this conclusion is that CPOF provides Csound a much more advanced means of UG development than what is available in similar music programming environments. This is because, as we have noted in Section 1.2, a more traditional C-like OOP idiom is prevalent elsewhere (even where C++ is the implementation language).

Another important result of this work, which should be highlighted here is the integration of modern C++ practices into a pure-C environment. While this has been done elsewhere, the extent to which we were able to apply the C++11 standard and generate a completely C-compatible code is of note. Some of the techniques developed here, such as the use of hiding and templating as part of the polymorphism mechanism, while quite particular to the case, are original solutions introduced by this work. Some of these are applicable to other systems, including as noted in Section 1.2, PD, SuperCollider, and more generally should be of interest to developers of DSP applications.

8. Conclusions

Object-oriented programming is an established paradigm for systems implementation, which has been used in a variety of applications. In music programming systems, it is very well suited to the development of language components, unit generators, which themselves are loosely modelled on this approach. Csound originally provided through its API a C-language interface for the addition of new opcodes to the system, but the supports for object orientation under that language are incipient.

This paper described the motivations, design and implementation of a framework for opcode development in C++, CPOF, together with its support toolkit. We were able to demonstrate how it enables a range of idioms that are being adopted as standard for object-oriented programming under that language, especially following the advent of C++11 [30] standard. We have provided a detailed discussion of the framework code with several examples, complemented with the discussion of two specific cases of object-orientation that are well supported by CPOF.

This version of CPOF is already being adopted by Csound opcode developers; starting from version 6.09, a number of new UGs have appeared, which are based on the framework. While the design described here is solid and matches well the underlying opcode model, some additions and improvements might be considered in newer versions. One particular approach would be to consider a way to encapsulate the Csound type system in a more complete way under C++, representing not only the data formats (as we do now), but also the timing aspects of each type.

To build a plugin opcode library based on this framework, a C++ compiler supporting the C++11 standard (`-std=c++11`), and the Csound public headers. The Csound plugin mechanism does not depend on any particular link-time libraries. The opcode library should be built as a dynamic/shared module (e.g., `.so` on Linux, `.dylib` on OSX or `.dll` on Windows), and placed in the opcode directory (or, alternatively, it can be loaded with the `--opcode-lib=` flag).

All opcode examples in this paper, with the exception of the ones in Section 7, are provided in `opcodes.cpp`, found in the `examples/plugin` directory of the Csound source codebase (https://github.com/csound/csound). The cases discussed in Section 7 are part of the existing code base (in `Opcodes/arrayops.cpp` and `Opcodes/pvsops.cpp`). CPOF is part of Csound and is distributed alongside its public headers. Csound is free software, licensed by the Lesser GNU Public License.

Conflicts of Interest: The author declares no conflict of interest.

References

1. Lazzarini, V. The Development of Computer Music Programming Systems. *J. New Music Res.* **2013**, *1*, 97–110.
2. Dodge, C.; Jerse, T.A. *Computer Music: Synthesis, Composition and Performance*, 2nd ed.; Schirmer: New York, NY, USA, 1997.

3. Dannenberg, R.B.; Thompson, N. Real-Time Software Synthesis on Superscalar Architectures. *Comput. Music J.* **1997**, *21*, 83–94.
4. Lazzarini, V.; ffitch, J.; Yi, S.; Heintz, J.; Brandtsegg, Ø.; McCurdy, I. *Csound: A Sound and Music Computing System*; Springer: Berlin, Germany, 2016.
5. Boulanger, R. (Ed.) *The Csound Book*; MIT Press: Cambridge, MA, USA, 2000.
6. Sondergaard, H.; Sestoft, P. Referential Transparency, Definiteness and Unfoldability. *Acta Inf.* **1990**, *27*, 505–517.
7. Ffitch, J. Understanding an Opcode in Csound. In *The Audio Programming Book*; Boulanger, R., Lazzarini, V., Eds.; MIT Press: Cambridge, MA, USA, 2010; pp. 581–615.
8. Lieberman, H. Machine Tongues IX: Object Oriented Programming. In *The Well-tempered Object: Musical Applications of Object-Oriented Software Technology*; Pope, S., Ed.; MIT Press: Cambridge, MA, USA, 1991; pp. 18–31.
9. Stroustrup, B. *The C++ Programming Language*, 4th ed.; Addison-Wesley: Boston, MA, USA, 2013.
10. Puckette, M. *The Theory and Technique of Computer Music*; World Scientific Publishing: New York, NY, USA, 2007.
11. McCartney, J. Rethinking the Computer Music Language: Supercollider. *Comput. Music J.* **2002**, *26*, 61–68.
12. Stowell, D. Writing Unit Generator Plug-ins. In *The SuperCollider Book*; Wilson, S., Cottle, D., Collins, N., Eds.; MIT Press: Cambridge, MA, USA, 2010; pp. 692–720.
13. Boulanger, R.; Lazzarini, V. (Eds.) *The Audio Programming Book*; MIT Press: Cambridge, MA, USA, 2010.
14. Cook, P.; Scavone, G. The Synthesis Toolkit (STK). In Proceedings of the 1999 International Computer Music Conference, Beijing, China, 22–27 October 1999; Volume III, pp. 164–166.
15. Lazzarini, V. The SndObj Sound Object Library. *Organ. Sound* **2000**, *5*, 35–49.
16. Lazzarini, V. The Design of a Lightweight DSP Programming Library. In Proceedings of the 14th Sound and Music Computing Conference 2017, Aalto University, Espoo, Finland, 5–8 July 2017; pp. 5–12.
17. ISO/IEC. ISO International Standard ISO/IEC 14882:2014, Programming Language C++. 2014. Available online: https://www.iso.org/standard/64029.html (accessed on 25 July 2017).
18. Lazzarini, V. The Csound Plugin Opcode Framework. In Proceedings of the 14th Sound and Music Computing Conference 2017, Aalto University, Espoo, Finland, 5–8 July 2017; pp. 267–274.
19. Pope, S. Machine Tongues XI: Object-Oriented Software Design. In *The Well-tempered Object: Musical Applications of Object-oriented Software Technology*; Pope, S., Ed.; MIT Press: Cambridge, MA, USA, 1991; pp. 32–47.
20. Cardelli, L.; Wegner, P. On Understanding Types, Data Abstraction, and Polymorphism. *ACM Comput. Surv.* **1985**, *17*, 471–523.
21. Lippman, S.B. *Inside the C++ Object Model*; Addison Wesley Longman Publishing Co., Inc.: Redwood City, CA, USA, 1996.
22. Standard C++ Foundation. How to mix C and C++. Available online: https://isocpp.org/wiki/faq/mixing-c-and-cpp (accessed on 25 July 2017).
23. Abrahams, D.; Gurtovoy, A. *C++ Template Metaprogramming: Concepts, Tools, and Techniques from Boost and Beyond (C++ in Depth Series)*; Addison-Wesley Professional: Boston, MA, USA, 2004.
24. Lazzarini, V. Time-Domain Signal Processing. In *The Audio Programming Book*; Boulanger, R., Lazzarini, V., Eds.; MIT Press: Cambridge, MA, USA, 2010; pp. 463–512.
25. Moore, F.R. *Elements of Computer Music*; Prentice-Hall, Inc.: Upper Saddle River, NJ, USA, 1990.
26. Lazzarini, V. Spectral Opcodes. In *The Audio Programming Book*; Boulanger, R., Lazzarini, V., Eds.; MIT Press: Cambridge, MA, USA, 2010; pp. 617–626.
27. Dolson, M. The Phase Vocoder: A Tutorial. *Comput. Music J.* **1986**, *10*, 14–27.
28. IEEE/Open Group. The Open Group Base Specifications Issue 7, IEEE Std 1003.1-2008, 2016. Available online: http://pubs.opengroup.org/onlinepubs/9699919799/ (accessed on 25 July 2017).
29. Wishart, T. *Audible Design*; Orpheus The Pantomine Ltd.: North Yorkshire, UK, 1996.
30. ISO/IEC. ISO International Standard ISO/IEC 14882:2011, Programming Language C++, 2011. Available online: https://www.iso.org/standard/50372.html (accessed on 25 July 2017).