THE DESIGN OF A LIGHTWEIGHT DSP PROGRAMMING LIBRARY

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

ABSTRACT

This paper discusses the processes involved in designing and implementing an object-oriented library for audio signal processing in C++ (ISO/IEC C++14). The introduction presents the background and motivation for the project, which is related to providing a platform for the study and research of algorithms, with an added benefit of having an efficient and easy-to-deploy library of classes for application development. The design goals and directions are explored next, focusing on the principles of stateful representations of algorithms, abstraction/ encapsulation, code reuse and connectivity. The paper provides a general walkthrough the current classes and a detailed discussion of two algorithm implementations. Completing the discussion, an example program is presented.

1. INTRODUCTION

In 1998, I introduced version 1.0 of the SndObj library [1], which was at the time most likely one of the first generally available free and open-source general-purpose C++ class libraries for audio processing. It came out at around the same time as another early C++ toolkit, STK [2], which was mostly oriented to synthesis with physical models. The SndObj library included not only signal processing classes but also support for cross-platform realtime audio and MIDI, as it aimed to encompass all the general-purpose audio use cases [3]. The original code was completely based on pre-standard C++, as it appeared around the same time the C++98 was about to be published. It was heavily modelled on Stroustrup's prescriptions [4].

Over time, the library developed to include some of C++98 and later C++03 ideas, but its development suffered from the twists and turns of the language as it began to be standardised. In hindsight, writing thousands of lines in hundreds of source-code files based on a moving target as was C++ twenty years ago was an unwise and perilous task. However, someone had to do it. In the following decade, many C++ object-oriented library projects got developed, as the terrain got firmed underneath them.

My own attention got diverted elsewhere and from about 2005, very little was done to the SndObj library apart from adding wrappers to it and employing it as the basis for Python projects. The code, however, proved to be a good resource for teaching and I extracted many components from it as free-standing functions that I would use in classroom, creating a small function library for my students [5]. This allows the algorithms to be studied and played with, but it is not robust enough to be used more generally.

With the advent of C++11 [6], I have renewed my interest in the language and have realised that it is now in a much more stable state, which would warrant some time and resource investment. For those who work mostly in C, like myself, it is still a system of elephantine proportions that lacks some of the finesse of small languages. It has been described as having "the power, the elegance, and the simplicity of a hand grenade" [1] . However, when used in moderation, it can provide a number of benefits as an extension of C.

This is the background to the development of AuLib [7]. The motivation is to provide a simple, lightweight platform for the study, teaching, and research of digital signal processing (DSP) algorithms for audio, taking advantage of the newer, more established, C++ standards. It also has the added benefit of providing efficient and portable code that can be easily packaged and deployed in general-purpose applications. The fundamental aim is to provide wrappers for algorithms, where the audio processing code can be easily accessed, studied, and modified. This is wrapped in a very thin interface layer that allows easy connectivity of objects and a class hierarchy that emphasises common components and code re-use. The library code attempts to adhere strictly to C++14 [8] standards and best practices, as it aims to provide an example of robust software design.

This paper is organised as follows. We will present the library design and discuss the decisions taken in its development. A tour of the library and its current constituent classes is shown next, exploring it from the perspective of signal generation, processing, and input and output. Finally, two classes are singled out for a more detail discussion and a full program example is presented.

2. LIBRARY DESIGN

The library design borrows from a number of sources, which have shown the best practice in the implementation of audio programming code. One of the guiding aspects was to allow a good deal of flexibility in the construction of classes, instead of mandating the presence of specific components via an abstract base class with a number of empty

[1] Quote attributed to Kenneth C. Dyke, 05/04/97.

virtual methods. Instead, processing methods may or may not exist in a derived class, depending on what they are supposed to implement. They can be given any name, although the established informal nomenclature is to call them `process()`.

The principal base class in the library is `AudioBase`. This is subclassed to implement all different audio-handling objects, from synthesis/processing to signal buffers, function tables, delay lines, and audio IO. The layout of `AudioBase` is informed by a number of basic design decisions that underpin the principles of AuLib.

2.1 Stateful vs. stateless representations

One of the basic motivations for AuLib was to place self-standing algorithms, previously implemented as free functions, within a wrapping object allowing the safe-keeping of internal states. Let's explore this idea with a simple example of a sinusoidal oscillator, described by eq. 1.

$$s(t) = a(t)\sin\left(2\pi\int f(t)\right) \tag{1}$$

A C implementation of such function would have to take account of the sample-by-sample phase values that are produced by the integration of the time-varying frequency $f(t)$. Typically, in a sane implementation, the current value of the phase would be kept externally to the function, and modified as a side-effect (listing 1).

Listing 1. C function implementing eq. 1

```
const double twopi = 8. * atan(1.);
double sineosc(double a, double f,
        double *ph, double sr){
    double s = a * sin(*ph);
    *ph += twopi * f / sr;
    return s;
}
```

While this is entirely appropriate to demonstrate and expose the oscillator algorithm for study, it is clearly not robust enough to be incorporated into a library. Quite rightly, users would expect to be able to use such functions to implement multiple oscillators, in banks, or for amplitude or frequency modulation. In this context, a programmer could inadvertently supply a single phase address to a series of calls to such functions when implementing a bank of oscillators and would clearly fail to get the intended result. While it could work when carefully employed, such as stateless presentation of the algorithm is clearly incomplete.

While there are ways of describing a sine wave oscillator in a stateless or purely functional fashion, once we are committed to define the computation in a stateful form, we need to provide a means to keep an account of the current state. Clearly, a self-contained oscillator will need to maintain the last computed value of the phase, as the algorithm contains an integration. For this, we can wrap the whole algorithm in a class that models its state and the means to get an output sample. A minimal C++ class implementing such oscillator is shown in listing 2.

Listing 2. C++ class implementing eq. 1

```
struct SineOsc {
 double m_ph;
 double m_sr;

 SineOsc(double ph, double sr)
       m_ph(ph), m_sr(sr) {};

 double process(double a, double f){
    double s = a * sin(m_ph);
    m_ph += twopi* f / m_sr;
    return s;
 }
};
```

With an object-oriented implementation, the stateful description of the algorithm is complete and provides enough robustness for use in a variety of contexts. Likewise, if we look across the various types of DSP operations that a library would hope to implement, we will see all sorts of state variables involved. This provides enough motivation for the wrapping of such algorithms in C++ classes.

2.2 Abstraction and encapsulation

In fact, by clearly describing an algorithm as having a state and a means of computing its output, we are abstracting the DSP object as a specific data type. This encapsulates all the kinds of operations we would expect to be able to apply to such an object. What are the things we would like any DSP algorithm to contain? It would be useful for instance for it to hold its output so that we only need to compute it once. Basic attributes such as the sampling rate and the frame size (number of channels in an interleaved signal) would also be essential.

For efficient implementation, the output should not be limited to sample-by-sample computation (as in the minimal example of the oscillator in listing 2). Typically, we would want a block of frames to be generated for each call of a processing method, which may vary in size. A means of registering whether the object is in an error state would also be useful for program diagnostics. In this formulation, a class that models a generic Audio DSP object would contain the following attributes (listing 3)

Listing 3. Attributes of the Audio DSP base class

```
class AudioBase {
 protected:
  uint32_t m_nchnls;// no of channels
  uint32_t m_vframes;// vector size
  std::vector<double> m_vector;
  double m_sr;// sampling rate
  uint32_t m_error;// error code
    ...
};
```

These are protected so that no unintended modification is allowed. This class is for all practical purposes a wrapper around an audio vector (of double floating-point samples).

Methods for basic manipulation are also added: scale, offset, modulation, mixing, and sample access are provided through overloaded operators. Setting and getting samples off the vector are also provided (single channel samples, full blocks, etc.), and to modify the vector size, as well as methods to get the value of the object attributes.

2.3 Code re-use

Since we have embraced, for good reasons, the object-oriented approach, it is very useful to take advantage of inheritance, as well as composition. For this reason, I have designed the class hierarchy from the most general to the most specific, although overall the tree is not very deep (six levels at most). We will see two specific cases in more detail in section 4, but as an example, the `ResonZ` class shows how the re-use of code can be employed. In fig. 1, we see that it is subclassed from a series of parents.

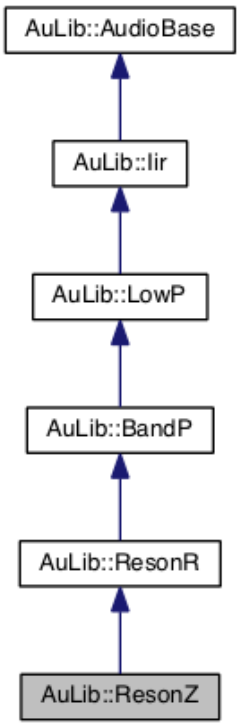

Figure 1. The `ResonZ` class and its parents.

At the top-level, `Iir` implements the basic second-order infinite impulse response (IIR) filter engine in direct form II (eq.2), with externally-defined coefficients.

$$\begin{aligned} w(t) &= x(t) - b_1 w(t-1) - b_2 w(t-2) \\ y(t) &= a_1 w(t) + a_1 w(t-1) + a_2 w(t-2) \end{aligned} \tag{2}$$

The `LowP` class holds a frequency parameter to calculate Butterworth low-pass coefficients; `BandP` adds a bandwidth attribute and re-implements the calculation of coefficients for a Butterworth band-pass configuration; `ResonR` re-implements the coefficient computation for a Resonator with an extra zero at R; `ResonZ` just sets the a_2 coefficient to -1, otherwise using the coefficient update code from its parent.

This shows an example of how each subclass represents mostly small modification of its parent, with most of its code re-used. Another benefit is that if a modification needs to be made (e.g. a bug fix), it does not need to be reproduced at several places (which opens the door for introducing small errors at these different locations).

Code re-use through composition is also employed throughout the library. For example, the `Delay` class holds an `AudioBase` object that implements its delay line, using the inlined access methods provided in that class. The `Balance` class, which implements envelope following and signal amplitude balancing, is made up of two `Rms` objects that are used to measure the RMS amplitude of input signals. `Rms` itself is a specialisation of a first-order low-pass filter class. In another example, the `TableSet` class, which is an utility class for the band-limited oscillator class `BlOsc` is made up of a vector of `FourierTable` objects containing waveform tables.

2.4 Connectivity

Some special attention has been given to the ways in which objects can be easily connected with each other and with code from other libraries (both in C and C++). For this to be achieved, there are two ways in which input and output to objects can be achieved:

1. Through direct pointers to data: these are presented in the form of **`const double*`** to allow signals from other libraries and non-AuLib sources to be inserted as inputs to processes. These, in turn, also return a **`const double*`** to the object vector so that they can be sent to other destinations. This type of connectivity is unsafe, from a C++ perspective, as it requires the programmer to carefully set the vector boundaries, although it is common place within a C-language context.

2. Object references: processing methods also allow connections to and from **`const`** `AudioBase &` variables, which provides more safety since vector boundaries are checked before access. They are the preferred way to pass signals from one library object to another. For convenience, classes overload the **`operator`**`()` as a shortcut for processing methods using object references.

While there is no mandatory way in which this is enforced in derived classes, the informal convention is to provide two processing methods as part of the public interface: one which deals with data pointers (producing and/or consuming arrays) and another that uses object references as input and/or output, as shown in listing 4. These methods delegate to a private virtual DSP method, which does the actual processing for the object and can be overriden in a derived class. This approach allows for good separation of concerns between interface and implementation.

Listing 4. Processing methods and their connectivity

```
class Proc : public AudioBase {
```

```
virtual const double *
   dsp(const double *sig);
public:
 const double *
   process(const double *sig) {
     return dsp(sig);
   }
 const Proc &
   process(const AudioBase &obj) {
     if(obj.vframes() == m_vframes &&
        obj.nchnls() == m_nchnls) {
         process(obj.vector());
      } else m_error = AULIB_ERROR;
      return *this;
   }
 const Proc &
  operator()(const AudioBase &obj) {
   return process(obj);
  }
...
};
```

No blocking operations (and/or resource allocation) should take place in a processing method. This is also the case for all inline vector manipulation methods (operators, etc.) provided by the base class, which are all real-time safe.

3. A TOUR OF THE LIBRARY

There are currently 61 classes in the library, of which 56 sit in the main `AudioBase` tree. Fig.2 shows the `AuLib::AudioBase` class hierarchy. These classes can be loosely categorised as processing (signal generators and processors, ie. they implement `process()` methods), function tables, holding mostly constant buffers, and input/output, which allow some form of audio IO through `read()` or `write` methods. In addition to these, the library also features note, instrument and score model classes.

3.1 Signal generators

Signal generators in AuLib include standard table-lookup oscillators (discussed in more detail in section 4), sampled-sound and band-limited waveform oscillators, phase generator, table readers, and envelope generators. The `SamplePlayer` class takes a buffer/function table containing recorded samples and plays it back with pitch and amplitude control either in a loop or as a single-shot performance. It can handle multichannel sample tables producing multichannel output and uses linear interpolation for table lookup.

The library supports a number of function table classes, derived from `FuncTable`, which hold waveforms, envelopes, or signal samples. These can be read by oscillators or by table lookup objects, whose indices can be derived from any signal. A phase generator connected to a table reader implements an oscillator algorithm.

The `BlOsc` class implements band-limited waveform synthesis using wavetables stored in a TableSet object. This will contain a set of band-limited tables that are selected according to the desired fundamental frequency. Currently, TableSet supports classic waveforms (such as Sawtooth, Square and Triangle) constructed using `FourierTable` objects. However the mechanism can be expanded to handle generalised band-limited waveforms.

The library contains single-segment linear and exponential signal generators, which can be triggered and reset. Extending these, a generalised multi-segment plus release envelope class `Envel` is provided. It uses a utility class, `Segments`, that is used to set up a segment list that can be shared among several envelopes (and also used for envelope tables). A pre-packaged four-segment envelope, ADSR (attack-decay-sustain-released) is derived from it as a convenient way to create simple envelopes. The release segment in these classes is triggered by a specific method (`release()`), which makes the envelope jump immediately to that stage.

3.2 Signal processors

The library contains a basic set of signal processing classes. Seven types of second-order, plus two first-order (low- and high-pass) filters are present, alongside root-mean-square detection and signal balancing. The `Delay` class implements fixed or variable delays (depending on the choice of overloaded processing functions), with or without feedback. It can implement comb filters, flangers, vibrato and chorus effects. Derived from it, we have a high-order all-pass filter and a general-purpose finite impulse response filter (implementing direct convolution, which is discussed in section 4). `Delay` objects can be tapped by `Tap` (truncating) or `Tapi` (interpolating) processors.

Some signal-processing utilities are present. A channel extractor, `Chn`, takes an interleaved multi-channel input and outputs a requested channel. A signal bus, `SigBus`, can be used as a mixer, with scaling and offset. Completing these, there is an equal-power panning class, `Pan`, that produces a stereo output from a mono input signal.

In addition to these time-domain processing classes, AuLib provides support for streaming spectral processing using the short-time Fourier transform and its derivative, the phase vocoder. Free (stateless) functions for complex and real input discrete Fourier transform (using a radix-2 algorithm) are implemented from first principles and are also available for general use. A partitioned convolution class is also implemented using these functions.

3.3 Input and output

An asynchronous (non-blocking) input and output (IO) facility is provided as part of the library, through the `SoundIn` and `SoundOut` classes. The interface is fairly agnostic as far as its implementation is concerned. Currently, it provides a frontend to libsndfile [2], for soundfile IO, portaudio [3], for realtime device IO, and `std::iostream` for standard text IO.

Users of the library do not actually depend on these two IO classes. For instance, an application could place the processing classes directly in an audio system callback (e.g. through Jack [4]), without the use of any AuLib IO object.

[2] `http://www.mega-nerd.com/libsndfile`
[3] `http://www.portaudio.com`
[4] `http://jackaudio.org`

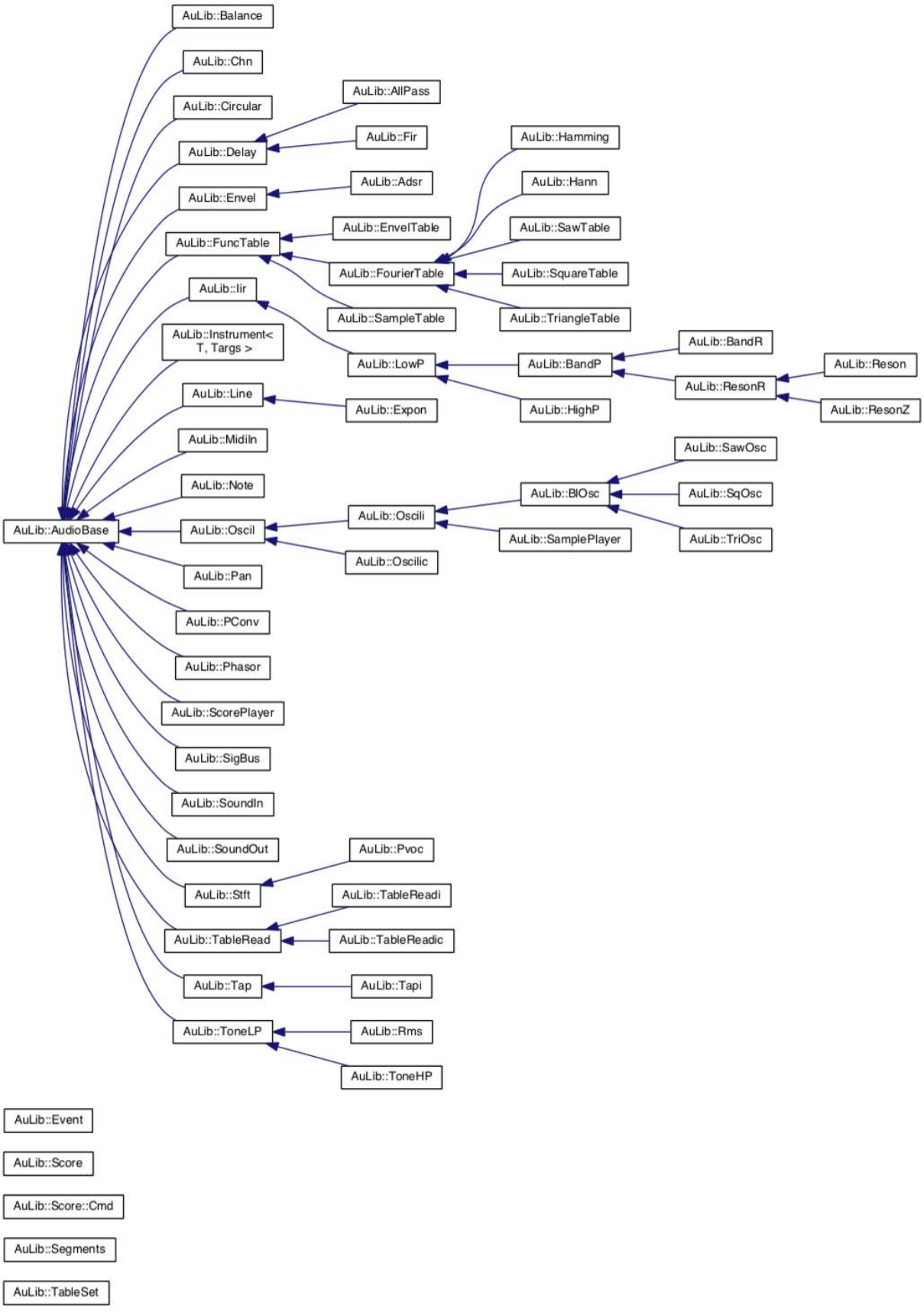

Figure 2. AuLib class hierarchy.

Equally, a processing graph based on library objects can be incorporated in a variety of settings, such as embedded hardware, mobile devices, etc..

In addition to Audio, a MidiIn class is also provided, dependent on the PortMidi library [5]. This is used to create MIDI-based applications, using the `Instrument` class.

4. EXAMPLES

In this section, we will look in detail at two examples of standard DSP algorithms and how they are implemented in the library.

4.1 Table-lookup oscillator

The table look-up oscillator, whose pedigree can be traced back to MUSIC III [9], is one of the cornerstones of digital synthesis systems. It is defined by the pair of equations 3 and 4, which characterise table lookup and phase increment, respectively, and represent a generalisation of eq.1 (N is the function size in samples and f_s is the sampling rate; $a(t)$ and $f(t)$ are the amplitude and frequency parameters).

$$s(t) = a(t)\mathrm{T}(\omega(t)) \tag{3}$$

$$\omega(t+1) = f(t)N/f_s \tag{4}$$

The function T() in eq.3 can be implemented in various forms. The simplest of them employs a floor operation, $\lfloor \omega(t) \bmod \mathrm{N} \rfloor$. This is called a truncating lookup and is implemented in listing 5, for the base class `Oscil`. Other forms can include interpolation schemes of various orders. In AuLib, linear and cubic lookups are implemented in derived classes, `Oscili` and `Oscilic` (fig.3).

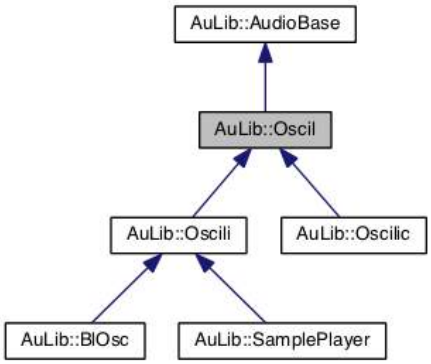

Figure 3. The `Oscil` class and its parents and children.

Listing 5. Truncating table-lookup oscillator.

```
void
AuLib::Oscil::dsp(){
 for(uint32_t i = 0;
        i < m_vframes; i++){
  am_fm(i);
  m_vector[i] =
   m_amp * m_table[(uint32_t)m_phs];
  mod();
 }
}
```

[5] http://portmedia.sourceforge.net/portmidi

Given that these share most components and only differ in how the table is accessed, it makes sense to express this commonality by inlining some operations. The functions $f(t)$ and $a(t)$ are updated from inputs in `am_fm()`, where they can vary on a sample-by-sample or vector-by-vector basis.

In order to give full flexibility and efficiency in parameter handling, a set of overloaded `process()` methods are provided, for fixed, varying, or modulating frequency and/or amplitude parameters. The actual processing code is delegated to `dsp()` virtual method (shown in listing 5 for the truncating case), which is then called by the `process()` interfaces defined in the base class.

4.2 Convolution

As a second implementation, we will have a look at direct (delay line) convolution, defined in eq. 5 for an impulse response N samples long. This effectively implements a finite impulse response (FIR) filter, which gives the name to the class containing the algorithm (`Fir`). Given that its main structure is a delay line, we inherit all of its internal components from `Delay` (fig.4).

$$y(t) = \sum_{n=0}^{N-1} s(t-n)h(n) \tag{5}$$

All we need to do is override the `dsp(const double*)` method. As in the previous example, and indeed across all of the library, signal processing is delegated to this function by the interface.

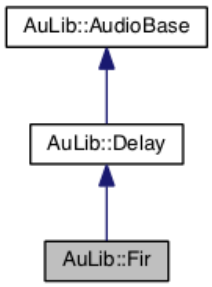

Figure 4. The `Fir` class and its parents.

The implementation is shown in listing 6, which is very compact. We can think of it as a fixed delay line tapped at each sample, with scaling factors taken from an impulse response (provided by a buffer/table object). Since the loop goes accessing the samples from maximum to no delay, the impulse response has to be read in reverse order.

Listing 6. Convolution implementation.

```
const double *
```

```
AuLib::Fir::dsp(const double *sig){
 double out = 0;
 uint32_t N = m_ir.tframes();
 for(uint32_t i = 0;
        i < m_vframes; i++){
  m_delay[m_pos] = sig[i];
  m_pos = m_pos != N-1 ?
                   m_pos + 1 : 0;
   for(uint32_t j = 0,rp = m_pos;
                j < N; j++){
    out +=
     m_delay[rp] * m_ir[N-1-j];
    rp = rp != N ? rp + 1 : 0;
   }
   m_vector[i] = out;
   out = 0.;
  }
  return vector();
}
```

5. AN ECHO PROGRAM

To complete the discussion, an echo program is shown in listing 7. Fig.5 shows the flowchart for a single audio input channel (multichannel input is allowed, with stereo output).

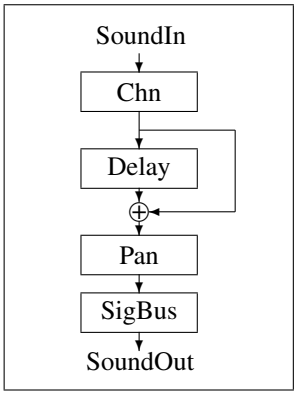

Figure 5. The signal flowchart for a single input channel in the echo program.

At the top of the program listing, lines 1 to 6 include the required AuLib classes, `SoundOut`, `SoundIn`, `Delay`, `Chn`, `SigBus` and `Pan`. A sound input object is constructed in line 17, taking as a source either the name of a soundfile, `"adc"` for realtime audio, or `"stdin"` for standard input. Arrays of channel readers, delay lines and panners are created next according to the requested number of channels. Lines 24 and 25 contain the mixer object and sound output constructors. Again, the output destination can be a soundfile name, `"dac"` for realtime, or `"stdout"` for standard output.

The processing loop, lines 33 to 44, is made up of calls to the various DSP objects through their **`operator`**`()`, plus the clearing of the signal bus mixer. It runs until the input is finished, which will depend on the source of samples. Note also the use of the overloaded operator += (in line 38) as a means of adding the dry and wet effect signals at the pan processing input.

6. CONCLUSIONS

This paper described the design of a simple, lightweight audio DSP library in C++. The main motivation is to provide a platform to develop and collect algorithms for the study, teaching and research in audio programming. The library classes are effectively thin wrappers that envelope succinct and efficient implementations of DSP operations. The code has been designed to be robust enough for general-purpose deployment in audio processing applications. Source code and multi-platform build scripts are provided at

`https://github.com/vlazzarini/aulib`

AuLib is free software, licensed by the Lesser GNU Public License.

7. REFERENCES

[1] V. Lazzarini and F. Accorsi, "Designing a sound object library," in *Proceedings of the XVIII Brazilian Computer Society Conference*, vol. III, Belo Horizonte, 1998, pp. 95–104.

[2] P. Cook and G. Scavone, "The synthesis toolkit (stk)," in *Proceedings of the ICMC 99*, vol. III, Berlin, 1999, pp. 164–166.

[3] V. Lazzarini, "The SndObj sound object library," *Organised Sound*, no. 5, pp. 35–49, 2000.

[4] B. Stroustrup, *The C++ Programming Language*, 2nd ed. Addison-Wesley, 1991.

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

[6] 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

[7] V. Lazzarini, "AuLib documentation, v.1.0 beta," 2017. [Online]. Available: http://vlazzarini.github.io/aulib/

[8] ISO/IEC, "ISO international standard ISO/IEC 14882:2014, programming language C++," 2014. [Online]. Available: http://www.iso.org/iso/home/store/catalogue_ics/catalogue_detail_ics.htm?csnumber=64029

[9] V. Lazzarini, "The development of computer music programming systems," *Journal of New Music Research*, no. 42, pp. 97–110, 2013.

Listing 7. Echo example program.

```
#include <Chn.h>
#include <Delay.h>
#include <Pan.h>
#include <SigBus.h>
#include <SoundIn.h>
#include <SoundOut.h>
#include <iostream>
#include <vector>

using namespace AuLib;
using namespace std;

int main(int argc, const char **argv) {

  if (argc > 2) {

    SoundIn input(argv[1]); // audio input

    std::vector<Chn> chn(input.nchnls()); // input channels
    std::vector<Delay> echo(input.nchnls(),
      Delay(0.5, 0.75, def_vframes, input.sr())); // delay lines
    std::vector<Pan> pan(input.nchnls()); // stereo panning

    SigBus mix(1. / input.nchnls(), 0., false, 2); // mixing bus
    SoundOut output(argv[2], 2, def_vframes, input.sr()); // audio output
    uint64_t end = input.dur() + 5*output.sr();

    std::vector<uint32_t> channels(input.nchnls()); // list of channels
    std::iota(channels.begin(), channels.end(), 0);

    cout << Info::version();

    while ((end -= def_vframes) > def_vframes) {
      input();
      for(uint32_t channel : channels) {
       chn[channel](input, channel + 1);
       echo[channel](chn[channel]);
       pan[channel](echo[channel] += chn[channel],
                    (1 + channel) * input.nchnls() / 2.);
       mix(pan[channel]);
      }
      output(mix);
      mix.clear();
    }

    return 0;
  } else
    std::cout << "usage: " << argv[0] << " <source> <dest>\n";

  return 1;
}
```