Musical Signal Scripting with PySndObj

Victor Lazzarini
Sound and Digital Music Technology Group,
Music Technology Lab,
NUI, Maynooth
Ireland
Victor.Lazzarini@nuim.ie

Abstract

This article discusses musical signal scripting using a Python language module, PySndObj, based on the Sound Object (SndObj) Library. This module allows for advanced music and audio scripting and provides support for fast application development and on-the-fly synthesis programming. The article discusses the main concepts involved in audio programming with the library. This is complemented by an overview of the PySndObj module with a number of basic examples of its use. The article concludes with the description of a proposed Musical Signal Processing system, which would include the previously discussed SndObj and PySndObj components.

Keywords

Musical Signal Processing, Object-Oriented Programming, Scripting Languages, Music Composition

1 Introduction

The Sound Object (SndObj) Library [1] is an object-oriented[2] audio processing library. It is a collection of classes for synthesis and processing of sound, inspired by the example set by the MUSIC *N* family of programs [3]. These can be used to build applications for computer-generated music. The source code is multi-platform and can be compiled under most C++ compilers. The library is a both an audio programming framework and a fast application development toolkit, with over 100 classes in its current release. For the latter uses, the library is available on C++[4], Java[5] or Python[6]. This article will discuss aspects of audio scripting using the SndObj library Python module, PySndObj.

1.1 What is a SndObj?

A SndObj (pronounced 'Sound Object') is a programming unit that can generate signals with audio or control characteristics. It has a number of basic attributes, such as an output vector, a sampling rate, a vectorsize and an input connection (which points to another SndObj). Depending on the type of SndObj, other attributes will also be featured: an oscillator will have an input connection for a function table, a delayline will have a delay buffer, etc..

SndObjs contain their own output signal. So, at a given time, if we want to obtain the signal it generates, we can probe its output vector. This will contain a vecsize number of samples that the object has generated after it was asked to either process or synthesise a signal. This is a basic characteristic of SndObjs: signals are internal, as opposed to existing in external buffers or busses. SndObjs can interface very easily with external signals, but in a pure SndObj processing chain, signals are internal and hidden.

1.2 Generating output

The basic operation that a SndObj performs is to produce an output signal. This is done by invoking the public member function SndObj::DoProcess(). Each call will generate a new output vector full of samples, so to generate a continuous signal stream, DoProcess() should be invoked repeatedly in a loop (known as the 'processing loop'). Programs will have to feature at least one such loop in order to generate audio signals.

As an alternative to directly programming a loop, users can avail of the services of the SndThread class and its derivatives, which provide processing thread management and a hidden processing loop (see below). The DoProcess() method is overridable, so each different variety of SndObj will implement it differently so that different objects can generate different signals. In

addition, other types of processing might be achieved with some overloaded operators (see below in ‘Manipulating SndObjs’).

1.3 Connecting SndObjs

Another basic programming concept found in this library is that SndObjs do not have direct signal inputs, because of the fact that signals are internal to them. Instead, they will have input connections to other SndObjs. This way an object will read the output signal of another which is connected to it. Any type of signal input, either a processing input or a parameter modulator input is connected in the same way.

Certain processing parameters will have two types of input: an *offset value* and a *SndObj connection*. The offset value, generally a single floating point value is added to whatever signal the connected SndObj has generated. In most cases, SndObj connections for parameters are optional: if they are not present, then only the offset value is used for it. In this case, they are in fact not an ‘offset’, but the actual value for the parameter. In other cases, the user will want to set the parameter offset to 0, so that only the SndObj input is used to control that parameter.

1.4 Manipulating SndObjs

Apart from invoking processing, users can manipulate SndObjs in other ways. The first obvious operation is parameter setting, for which different varieties of SndObjs will have different methods. However, a unified message-passing interface is defined by SndObj, with the SndObj::Set() and SndObj::Connect() methods. These can be used to change the status of SndObjs via the various messages defined for them.

Messages are also inherited, so the derived object will have its own set, plus the ones defined for its superclass(es). Set() is used to set offset and single parameter values. Connect() is used to connect input objects, which can be of SndObj, SndIO (input and output objects) or Table (function table objects) types. Messages are string constants. In addition, the output signal buffer can be accessed with a variety of methods such as SndObj::Output(), SndObj::PushIn() and SndObj::PopOut() .

1.5 Input and Output

Signal input and output is handled by SndIOs (‘sound ios’), which are objects that can write and read to files, memory, devices, etc. They are modelled in similar ways to SndObjs: signals are internal, use object connections, etc.. However, they are designed to deal with a slightly different type of processing. Their main performing methods are SndIO::Read() and SndIO::Write(). When invoked, these will read or write a vectorsize full of samples from/to their source/destination, respectively. SndIOs can handle multichannel streams, so their output vector actually contains frames of samples (in interleaved format).

SndIOs interact with SndObjs in two basic ways. For signal input, SndIOs can be accessed via SndIn objects. Each channel of input audio has to be connected separately, because SndObjs in general handle only single signal streams. For signal output, SndObjs can be connected directly to SndIOs (again, one for each channel). This can be done at construction time, or more usually using SndIO::SetOutput(). For MIDI input, a number of specialist classes exist, derived from MidiIn, which work in a similar way to SndIn.

1.6 Function Tables

Certain SndObjs, for instance oscillators, will depend heavily on tabulated function tables. For this purpose, a special type of object can be used, a Table object. Tables are very simple objects whose most important attribute is their actual tabulated function, which is created at construction time. Tables can be updated at any time, by changing some of their parameters and invoking Table::MakeTable().

1.7 Frequency-domain issues

The Sound Object Library provides classes for time and frequency-domain (spectral) processing. For the latter, a few special considerations must be made. Time-domain and spectral SndObjs are designed to fit in together very snuggly in a processing chain. For this reason, a certain model was employed, which slightly limits the arrangement of such SndObjs.

For spectral processing, the FFT size must be always power-of-two multiple of the hopsize (usually a minimum four times that value). When connecting time- and frequency-domain SndObjs, the hopsize must be the same as the time-domain vectorsize. Generally for an efficient FFT, the analysis size is set to a power-of-two value. So, in practice, this limits the vectorsize/hopsize and FFT size values to a limited pairing of values. Although at first this looks limiting, it will in fact have little impact of the flexibility of spectral processing using the library. This model, in turn, will facilitate immensely the interaction between

frequency- and time-domain SndObjs. Effectively, if these conditions are met, they can be inter-connected transparently, even though they are dealing with very different types of signals.

1.8 Processing threads

In addition to the basic types of objects discussed above, the Sound Object Library also includes a special thread management class, SndThread. With this type of object, a pthread-library based thread can be instantiated and run. This object encapsulates the main processing loop, calling the basic performing methods of each object that has been add to it.

Using SndThreads ('sound threads') is very simple. Once an object has been created and a chain of SndObjs/SndIOs has been defined, a processing list is initialised using SndThread::AddObj() or SndThread::Insert(). To start processing a signal, SndThread::ProcOn() is invoked. To stop processing, SndThread::Proc-Off() can be used. SndObjs can be deleted from the processing list using SndThread::DeleteObj(). Multiple SndThreads can be used for parallel processing with SndBuffer objects being used to obtain the signals from each thread.

2 PySndObj

PySndObj is a python module that wraps the SndObj C++ code in a very useful way to provide support for Python scripting. It allows for a nice scripting interface to the library for fast application development, prototyping of applications and general on-the-fly synthesis and processing.

PySndObj can be added, provided you have the _sndobj dynamic module (.so on Linux, .dylib on OSX and .dll on Windows) and csnd.py (the python bindings) in the right places (check your Python documentation), using the import command:

```
import sndobj
```

or

```
from sndobj import *
```

This will allow access to all SndObj library classes available to your platform, plus some extra utility classes for array support. The latter form allows for accessing the SndObj classes directly, without the package name as a prefix ('namespace'). For sake of simplicity and economy of space, we will be using the latter form as the basis for all further code examples. However it is important to point out that the recommended Python coding style is to explicitely use namespaces.

2.1 Python SndObj classes and objects

Python SndObj classes look very similar to their C++ counterparts. The main difference is that in Python all objects are dynamically allocated, so they are equivalent to C++ pointers to objects. Since the library uses pointers to connect object (See 'Programming Concepts'), using SndObjs in Python is very straightforward and transparent.

Connecting objects is very simple. Let's say we want to create a sine wavetable and connect an oscillator to it:

```
tab = HarmTable()
osc = Oscili(tab,440, 16000)
```

Here as variables hold object pointers, we have the case where 'tab' can be passed directly to osc, with no need for any extra complications. The same works for SndObj and SndIO connections, so if we set up a RT output object, we can connect our oscillator to it:

```
outp = SndRTIO(1)
outp.SetOutput(1,osc)
```

This works between SndObjs as we would expect, if we want to, say, connect a modulator to our oscillator:

```
mod = Oscili(tab, 2, 44)
osc.SetFreq(440, mod)
```

2.2 Running SndObjs

In order to get audio processing out of a SndObj, it is necessary to invoke its DoProcess() method. This runs the processing once and produces an output vector full of samples, eg.

```
osc.DoProcess()
```

For a continuous output, continued calls to DoProcess() are required, so we need to set up a loop, where this can happen. In addition, any SndIOs in the chain have also to call their Read() or Write() methods (for input or output respectively).

```
# 10 seconds of audio output
timecount = 0
end = 10 * osc.GetSr()
```

```
vecsize = osc.GetVectorSize()
while(timecount < end):
  mod.DoProcess()
  osc.DoProcess()
  outp.Write()
timecount += vecsize
```

2.3 Using a processing thread

However, the simplest way to get SndObjs producing audio is to use a SndThread object to control the synthesis. This will take care of setting up a processing loop and call all the required methods. We start by setting the object up:

```
thread = SndThread()
thread.AddObj(mod)
thread.AddObj(osc)
thread.AddObj(outp, SNDIO_OUT)
```

Then we can turn on the processing to get some audio out:

```
thread.ProcOn()
```

When we are done with it, we can turn it off:

```
thread.ProcOff()
```

In addition to SndThread, it is also possible to use SndRTThread, which has default realtime objects for input and output that can be connected to. In this case the user only needs to set up his/her SndObj chain and add this to the thread object.

Any asynchronous Python code can also be called, by setting up a process callback, that will be invoked once every processing period. This happens after the input signal(s) has been obtained, but before any processing by SndObjs.

For instance, if we have a method:

```
def callb(data):
    ... # callback code
```

This sets the callback, `data` is any data object to be passed to the callback:

```
thread.SetProcessCallback(callb,data)
```

The callback can be used for things like updating a display, changing parameters cyclically, polling for control input, etc..

2.4 Support for arrays

In order to facilitate certain ways of programming and to make possible the use of C arrays with the library, some utility classes have been added for int, float and double arrays, named, respectively: intArray, floatArray and doubleArray. These classes can be used as follows

```
# create an array of two items
f = floatArray(2)
# array objects can be manipulated
# by index as in C
f[0] = 2.5
```

In addition, a special type of array is also available, the sndobjArray, which holds SndObjs (internally C++ SndObj pointers). Objects of this type can be used similarly to the above array:

```
objs = sndobjArray(2)
objs[0] = mod
objs[1] = osc
```

However, when these are used as SndObj pointer arrays, they will need to be cast as that. Hopefully the class has a handy method for doing just that:

```
objp = objs.cast()
```

These can be used with objects that take arrays of SndObjs as input, such as SndThread:

```
thread = SndThread(2, objp, outp)
```

in which case we are setting up a thread with two SndObjs (which is similar to the example above). Other objects that take SndObj arrays are for instance SndIO-derived objects and Mixer SndObjs. But remember, SndObj arrays are not sndobjArray objects, but can be retrieved using sndobjArray::cast().

3 Simple examples

Two simple scripts are provided here for realtime audio processing and synthesis.

3.1 Simple echo using a comb filter

This example demonstrates realtime audio IO and some delayline processing. The example uses a SndRTThread object as introduced above. This takes care of all realtime input/output.

```
from sndobj import *
import time
import sys

if len(sys.argv) > 1:
  dur = sys.argv[1]
else:
  dur = 60

# SndRTThread object has its own
# IO objects.
# By the default it is created wth
```

```
# 2 channels
t = SndRTThread(2)

# Echo objects take input from
# SndRTThread inputs
comb_left = Comb(0.48,0.6,t.GetInput(1))
comb_right = Comb(0.52, 0.6,t.GetInput(1))

# We now add the echo objects to
# the output channels
t.AddOutput(1, comb_left)
t.AddOutput(2, comb_right)

# This connects input to output
# directly
t.Direct(1)
t.Direct(2)

# turn on processing for
# dur seconds
t.ProcOn()
time.sleep(float(dur))
t.ProcOff()
```

3.2 Oscillator with GUI (using wxPython)

Here we present a complete GUI-based synthesis program (albeit a trivial one). This demonstrates how a GUI toolkit (such as wxPython[7]) can be used to create complete computer instruments.

```
from sndobj import *
from wxPython.wx import *
import traceback
import time

class ControlPanel(wxPanel)
# Override the base class
# constructor
def __init__(self, parent):
wxPanel.__init__(self, parent, -1)
self.ID_BUTTON1 = 10
self.button1 = wxButton(self,      \
    self.ID_BUTTON1, "On/Off",     \
    (20, 20))
# Bind the button to its event
# handler.
 EVT_BUTTON(self, self.ID_BUTTON1,\
  self.OnClickButton1)
# Create a slider to change pitch
 self.ID_SLIDER1 = 20
 self.slider1 = wxSlider(self,     \
   self.ID_SLIDER1, 300, 200, 400,\
   (20, 50), (200,50),             \
     wxSL_HORIZONTAL | wxSL_LABELS)
 self.slider1.SetTickFreq(5, 1)
# Bind the slider to its event
#  handler.
  EVT_SLIDER(self, self.ID_SLIDER1, \
        self.OnSlider1Move)
EVT_CLOSE(parent, self.OnClose)
# Default pitch.
self.pitch = 300
# Sine wave table
self.tab = HarmTable()
# Envelope (just attack actually)
self.line = Interp(0, 10000, 0.05)
# oscil
self.osc = Oscili(self.tab,    \
   self.pitch, 0, None, self.line)
self.out = SndRTIO(1,SND_OUTPUT)
self.out.SetOutput(1, self.osc)
self.thread = SndThread()
self.thread.AddObj(self.line)
self.thread.AddObj(self.osc)
self.thread.AddObj(self.out, SNDIO_OUT)
self.play = False

  def OnClickButton1(self, event):
  if(self.play):
# create an envelope decay
self.line.SetCurve(10000, 0)
   self.line.Restart()
   self.play = False
 else:
# create an envelope attack
   self.line.SetCurve(0, 10000)
   self.line.Restart()
   self.play = True
   self.thread.ProcOn()

# slider movement
def OnSlider1Move(self, event):
  self.pitch = event.GetInt()
  self.osc.SetFreq(self.pitch)

# stop performance
def OnClose(self, event):
   try:
     self.thread.ProcOff()
     time.sleep(1)
     self.GetParent().Destroy()
   except:
     print traceback.print_exc()

# Create a wx application.
application= wxPySimpleApp()
# Create a parent frame
frame= wxFrame(None,-1,"PySndObj example")
# Create the controls
controlPanel= ControlPanel(frame)
# Display the frame.
frame.Show(True)
# Run the application.
application.MainLoop()
```

4 Towards a Musical Signal Processing System

The SndObj library and PySndObj, in fact, form part of a larger picture of a proposed system that will incorporate a third software component in the form of graphic 'patching application'. These three elements would then form a three-layer Musical Signal Processing System (fig.1), allowing different entry levels of user interaction.

At the top, the patching application will provide a GUI layer. At this level, users can set up patches of SndObjs, event patterns, automation, etc., without the use (or at least with reduced use) of textual declarations.

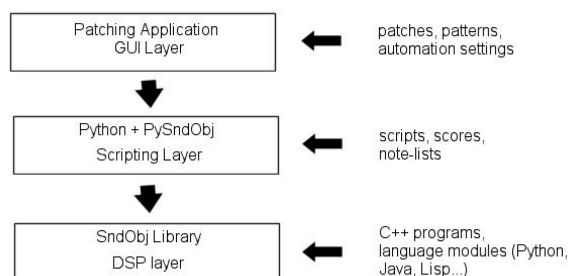

Figure 1. The three levels of a proposed Musical Signal Processing System

The middle layer is provided by Python and PySndObj (possibly with some Python-written add-ons for event processing and control). This provides the interface between the patching application (itself written in Python) and the lower-level SndObj library. Users will be able to use this layer directly, bypassing (or combining with) the patching application, for scores and, more general, scripting (for instance, alternative top-level applications can be written using this layer).

The lower level is then provided by the SndObj library itself, the DSP engine for the system. This layer can be accessed directly by the user in C++ code, providing standalone applications, modules for languages (similarly to PySndObj, for Java, Lisp, etc...).

Of the three components for this proposed system, the lower end is complete and functional. The middle layer (as discussed in this article) is functional, but perhaps needing some extensions for better event processing support. The top level does not exist yet, although some idea of how it might look like was provided by the AIDE software[8], developed using the SndObj library.

5 Conclusion

PySndObj provides a good support for audio processing in Python. The simplicity of the language, allied to the modularity and comprehesiveness of the library proves to be a powerful combination.

All of the SndObj tools are Free software, GPL licensed, and are available from sourceforge download (full releases) or anonymous CVS at:

http://sndobj.sf.net

Developers are also encouraged to join the project and can do so by contacting the author at his e-mail address.

6 References

[1] V Lazzarini. 2000. The Sound Object Library. *Organised Sound 5 (1),* pages 35-49. Cambridge Univ. Press., Cambridge.

[2] M Abadi and L Cardelli. 1996. *A Theory of Objects*, Springer-Verlag, New York.

[3] C Dodge and T Jerse. 1985. *Computer Music: Synthesis, Composition and Performance.* Schirmer Books, New York.

[4] B Stroustrop. 1991. *The C++ Programming Language*, second edition. Addison-Wesley, New York.

[5] K Arnold and J Gosling. 1996. *The Java Programming Language.* Addison-Wesley, New York.

[6] G Van Rossum and F Drake. 2003. *The Python Language Reference Manual*. Network Theory, Bristol.

[7] http://www.wxwidgets.org

[8] V Lazzarini and R Walsh. 2004. AIDE, a New digital audio effects development environment. *Proc. of the 7th Int. Conference on Digital Audio Effects (DAFx-04)*, pages 53-57. Univ.of Naples, Naples.