Pyo, the Python DSP toolbox

Olivier Bélanger
Faculty of Music, University of Montreal
200 rue Vincent d'Indy
Montréal, Québec, Canada
olivier.belanger@umontreal.ca

SOFTWARE SUBMITTED TO ACM MULTIMEDIA 2016 OPEN SOURCE SOFTWARE COMPETITION.

The software presented in this article is free of charge and open-source.

Please visit http://ajaxsoundstudio.com/software/pyo/ or the github repository at https://github.com/belangeo/pyo for source code, downloads, documentation and other resources.

ABSTRACT

This paper introduces *pyo*, a python module dedicated to the digital processing of sound. This audio engine distinguishes itself from other alternatives by being natively integrated to a common general programming language. This integration allows incorporating audio processes quickly to other programming tasks, like mathematical computations, network communications or graphical interface programming. We will expose the main features of the library as well as the different contexts of use where pyo can be of a great benefit to composers and audio software developers.

Keywords

Python, Digital Signal Processing, Audio Programming

1. INTRODUCTION

Over the years, several ways and means were developed to facilitate audio programming. Max Mathews's MUSIC-N[1] series and Barry Vercoe's Csound[5] program used the instrument/score paradigm to allow the user to create synthesizers or processing instruments in one file and specify the events controlling the notes and parameters over time into another file. Perhaps more intuitive for musicians, the patching paradigm implemented in Pure Data[4] lets the user connect objects in processing chains to generate both sound and control structures. SuperCollider[2], developed by James McCartney in 1996, uses an object-oriented programming language to allow dynamic real-time audio synthesis and algorithmic composition.

All these tools perform well for a wide variety of tasks in music composition, audio synthesis, signal analysis and live digital signal processing. So, what was the motivation for the development of an entirely new audio engine? The common thing to all these tools is the specialized language they each use to perform audio programming tasks. If one wishes to develop an audio platform answering to needs for general programming tasks, such as accessing databases, creating a custom graphical user interface or analyzing the content of a text file, these languages fall short as they were not designed to efficiently perform these kinds of tasks. They were purpose-built to perform audio signal processing. Of course, there are ways to communicate between these programs and more general programming languages, such as Python. It can be done either via an application programming interface (API) or with a communication protocol like Open Sound Control (OSC), but this can lead to several programming problems and force the developer to deal with two different environments at the same time.

The development of pyo as a Python module was motivated by the need to provide a complete audio programming toolkit for a general programming language. Pyo is a Python module where the logic and API are written in pure Python while all the signal processing parts are written in C. The communication between the two layers is entirely done with the Python C-API. This integration to a popular language allows the user to concentrate all his efforts on the development of audio algorithms, without the need to learn a whole new programming language. The program benefits from the power and speed of an extended audio toolkit and from the syntax and versatility of a mature programming language that is widely used.

This article is divided into three parts. First, we will introduce the programming environment we propose to make the sound. In the second part, we will explore the possible applications with pyo and to whom this module can be useful. Finally, we will explain the main features of the library and illustrate them with simple code examples.

2. PROGRAMMING ENVIRONMENT

Python is a cross-platform, open-source, dynamic programming language used in a wide variety of application domains. His clear and readable syntax makes it a perfect choice to learn to program. Python offers an intuitive object orientation that is very suitable for signal processing design

[1]https://en.wikipedia.org/wiki/MUSIC-N

ACMMM '16 October 15–19, 2016, Amsterdam, The Netherlands

DOI: 10.475/123_4

and algorithmic composition. One can use Python to build sophisticated software or simply as a scripting language for light duty tasks. An active community of developers regularly updates the source code and provides new versions. It is well documented and offers extensive standard libraries and third party modules for virtually any purpose. Because it allows programmers to write efficient code very quickly, it's one of the best choices to start writing audio programs.

Pyo is a Python module written in C to help digital signal processing script creation. It is entirely integrated, which means the audio engine has no need for an API or protocol like OSC to communicate with the language. It contains classes for a wide variety of signal processing types. With pyo, users can include signal processing chains directly in Python scripts or projects and manipulate them in real time through the interpreter. Tools in the pyo module offer primitives like arithmetic operations on an audio signal, basic signal processing (filters, delays, oscillators, etc.) and complex algorithms to create sound granulation or other creative audio manipulations. pyo supports the OSC protocol (Open Sound Control), to ease communications between programs and includes a full implementation of the MIDI protocol for generating sound events and controlling process parameters. pyo allows the creation of sophisticated signal processing chains with all the benefits of a mature and widely used general programming language.

One significant constraint in audio programming is that the program must run for an extended period. While most programs can just quit in a friendly way after the execution of each line of code, an audio performance must compute samples as long as they are needed. The Python interpreter (the engine evaluating the commands) can be invoked directly in a terminal window and used to create an audio processing loop. It will stay active until the user explicitly asks to terminate. The interpreter is very useful for live coding or for experimenting with some small functions. However, most of the time, it will be easier to call a graphical user interface which will keep the interpreter alive as long as a window is shown on the screen. Pyo provides such interfaces for the audio server and also for object's display and control.

3. APPLICATIONS

Pyo is well suited for any tasks that involve sound and programming. One of its primary purposes is for music creation and algorithmic composition. One can use pyo to process audio inputs and control effect parameters while performing on stage. Another way of composing with pyo is to take advantage of python's algorithmic capabilities to create complex control structures for a synthesis engine. We have created a web radio continuously processing and playing such algorithmic compositions from whoever wants its tune to be played. The radio can be listened to at http://radiopyo.acaia.ca/.

As part of a larger programming eco-system, pyo can easily be integrated into the engine of a music software. Here are some softwares where pyo provides the audio services:

- **Cecilia5**, a digital signal processing toolbox. http://ajaxsoundstudio.com/software/cecilia/
- **Soundgrain**, a granular sound synthesis interface. http://ajaxsoundstudio.com/software/soundgrain/
- **PsychoPy**, psychology and psychophysic experiments. http://www.psychopy.org/

The distribution of pyo comes with E-Pyo, a simple but powerful text editor offering live coding facilities such as a background audio server and shortcuts to increase the coding speed. It's especially useful for quickly exploring new processing techniques.

Pyo also offers a simple C-API that allow embedding a Python interpreter in another application written in C/C++. There are examples in the sources showing pyo embedded inside OpenFrameworks, PureData and in a Juce audio plugin.

4. INTENDED AUDIENCE

The intended audience of a python module like pyo is vast. Any music composer who wants to build its tools or to explore with sound can make a good use of pyo. Because there is libraries for almost any programming task in python, software developers from a variety of backgrounds can benefit from pyo if they need to include an audio part in their projects. Since 2010, I use pyo in my university classes to teach digital signal processing theory and practice and to introduce musicians to audio programming. The combination Python and pyo offers a very gentle learning curve and one or two semesters are usually enough for students to produce original and exciting work. In conjunction with scientific modules like numpy or music21, pyo should be in the toolbox of any electronic music teacher or researcher.

5. MAIN FEATURES

5.1 The audio engine

Pyo uses a callback function to allow real-time audio processing without blocking the main thread. In blocking mode, the user could not enter new commands from the interpreter or a graphical interface could not be refreshed until the process exit. This function is part of the Server object and is called by the audio driver, in a high priority thread, every time a new block of samples is needed. Audio and MIDI configurations can be modified at the Server's initialization if, for example, the default sound card or the default MIDI interface are not the ones needed by the user. We specify the sampling rate and the size of each block of samples as arguments to the Server. Another role of the Server is to manage the connections between pyo objects and the order in which they will be computed. The callback function will ask all registered objects, in the order they were created by the program, for a new block of samples unless the objects tree is explicitly modified during the execution. To be properly registered, the first thing an audio object did at its creation, is to look for the current server in the program memory. As a consequence, a server must be present and booted before creating any audio object. The following sequence of instructions create, initialize and start an audio server:

```
>>> from pyo import *
>>> # Create a stereo server
>>> s = Server(sr=48000, nchnls=2, buffersize=64)
>>> # Set the audio and MIDI devices
>>> s.setInOutDevice(2)
>>> s.setMidiInputDevice(1)
>>> # Boot the server
>>> s.boot()
>>> # Now we can create a processing chain...
>>> s.start() # Start the audio callback loop
```

5.2 Everything is computed at audio rate

One significant difference between pyo and older audio engines is the removal of the control rate. The control rate is a second rate, slower than the audio rate, used to compute variations over time for some parameters of the generators or the effect processors. The control rate is cheaper to process, as it computes only a single floating-point value instead of a block of samples, and therefore uses fewer CPU resources. The cost for this gain in CPU is that the control rate can causes artifacts, like zipper noise, in the sound when values are changing too fast. With computers growing more powerful, the need for saving CPU resources is less significant, and the audio quality can be prioritized at a reasonable cost. Within pyo, every object generates an audio signal, and almost all parameters accept audio as a control signal. This flexibility allows a broad range of modulations without having to care about audio degradation by a downsampled signal given as a modulator of a parameter such as a filter's frequency. In the example below, an oscillator modulates the center frequency of a bandpass filter to create a kind of FM (*frequency modulation*) effect on the microphone input:

```
>>> from pyo import *
>>> s = Server().boot()
>>> src = Input()
>>> mod = Sine(freq=500, mul=250, add=500)
>>> filter = ButBP(src, freq=mod, q=5).out()
>>> s.start()
```

5.3 Sample-accurate timing

One of the benefits of the fact that everything is an audio signal is the trigger framework, with which it is possible to create sample-accurate timing structure. A trigger is a signal with a value of one, surrounded by zeros. All trigger generators, such as a metronome or a step sequencer, are sampled at the audio rate, allowing the most precise possible timing. A lot of objects in the library are configured to respond to this kind of signal, so, one could create a tempo-style control structure with the guarantee of the best timing accuracy. The following piece of code generates a polyphonic melody with an exponential envelope and a square wave:

```
>>> from pyo import *
>>> s = Server().boot()
>>> wav = SquareTable()
>>> env = ExpTable([(0,0), (64,1), (8191,0)])
>>> met = Metro(time=.125, poly=8).play()
>>> amp = TrigEnv(met, table=env, dur=1, mul=.1)
>>> mid = TrigRandInt(met, max=12, add=48)
>>> hz = Snap(mid, [0,2,3,5,7,8,10], scale=1)
>>> out = Osc(table=wav, freq=hz, mul=amp).out()
>>> s.start()
```

5.4 List expansion

A powerful property, called "multichannel expansion," introduced in the textual synthesis language SuperCollider uses an array to duplicate processes on a single line of code [3]. If we give an array of frequencies to a resonator, it will create as many resonators as there are values in the array. Pyo uses the python's list type to implement a similar duplication of object's processing. Almost all parameters accept a list as an argument, creating as many audio streams as necessary to process all values in the list. A stream is a monophonic audio signal container, and a pyo object can manage any number of these streams. One can create two hundred oscillators in one line by giving a list of two hundred values to the frequency parameter of a Sine object. A pyo object is also considered as a list by Python, which means that if an object receives another pyo object in one of its parameters, the receiver will generate the same number of streams, each one with its audio variation. This system is very expressive and allows to create highly flexible and compact scripts. The next example generates a chorus of N oscillators with random frequencies, phases, and amplitudes.

```
>>> from pyo import *
>>> from random import random, uniform
>>> s = Server().boot()
>>> N = 200
>>> freqs = [uniform(100, 900) for i in range(N)]
>>> phases = [random() for i in range(N)]
>>> amps = [uniform(.001, .02) for i in range(N)]
>>> oscs = Sine(freqs, phases, amps).out()
>>> s.start()
```

5.5 Multichannel environment

Pyo can be turned into a multichannel environment, instead of stereo, simply by giving the desired number of channels as an argument to the Server object. When a pyo object is asked to send its audio streams to the sound card, the default behavior is to alternate its streams over the available channels cyclically. This behavior can be overridden in many ways, either with random functions, or by specifying the exact output for each stream managed by the object, or with the available panning functionalities of the library. The panning objects take the number of channels as an argument, allowing one to switch quickly between a stereo and a multichannel workstation.

```
>>> from pyo import *
>>> CHNLS = 8
>>> s = Server(nchnls=CHNLS).boot()
>>> n = Noise(.5)
>>> lfo = Sine(.1, mul=0.5, add=0.5)
>>> pan = Pan(n, outs=CHNLS, pan=lfo).out()
>>> s.start()
```

5.6 Arithmetic with audio objects

Pyo objects also override the basic math operators ($*$, $/$, $+$, $-$, $**$, $\%$), meaning you can do arithmetic involving audio objects. When math operations contain pyo objects, a new audio object, called Dummy, is automatically created to hold the result of the computation. The action leaves the original object untouched. This behavior can be very useful as a single audio object can be multiplied by a list of floats to return a list of similar, but slightly different, audio objects. Arithmetic operations can involve only audio objects or both audio objects and floats.

In a similar way, conditional operators ($<$, $<=$, $>$, $>=$, $==$, $!=$) are overridden in pyo objects to return an audio stream containing zeros and ones, depending on the result of the comparison.

The next sample illustrates some arithmetics with audio objects by creating a complex tone with the summation of three sine waves. The frequency of each oscillator is a harmonic of a given fundamental with an independent jitter and a common vibrato.

```
>>> from pyo import *
>>> s = Server().boot()
>>> f = 100
>>> vib = Sine(5, mul=0.02, add=1)
>>> jit = Randi(min=0.99, max=1.01, freq=[1,2,3])
>>> a1 = Sine(freq=f * vib * jit[0], mul=0.5)
>>> a2 = Sine(freq=f * 2 * vib * jit[1], mul=0.3)
>>> a3 = Sine(freq=f * 3 * vib * jit[2], mul=0.1)
>>> total = (a1 + a2 + a3).out()
>>> s.start()
```

5.7 MIDI and OSC support

Music is all about instrument and control. In this regards, pyo offers a complete support for MIDI and OSC[6] (*Open Sound Control*) communication. These two protocols are widely used and incorporated into many commercial and custom musical interfaces. It is a very efficient way to modify and generate sound events in real-time. Here is the bare minimum to play notes in pyo with a MIDI keyboard:

```
>>> from pyo import *
>>> s = Server()
>>> s.setMidiInputDevice(99) # open all devices!
>>> s.boot()
>>> note = Notein(poly=10, scale=1)
>>> amp = MidiAdsr(note['velocity'], mul=0.2)
>>> osc = RCOsc(freq=note['pitch'], mul=amp)
>>> out = osc.mix(2).out()
>>> s.start()
```

5.8 Other features

After several years of development of pyo, it is now a DSP toolkit pretty complete. In addition to essential processes like soundfile players, filters, wave shapers, oscillators, random generators, reverbs, etc., there is an all set of objects to do spectral transformations with the phase vocoder technique[1]. Scientists and psychoacoustics will find analysis tools to retrieve and display sound characteristics such as fundamental frequency, spectrum envelope, amplitude envelope, brightness and spectral center of gravity. Pyo also has many built-in operators to create and manipulate tables or matrices and much more. See the documentation for the detail about the hundreds of objects in the library.

6. CONCLUSION

In this article we have introduced a new audio engine that, developed as a Python module, is part of a larger programming environment than the sound itself. It is a dedicated module amongst thousands of other specialized modules. We have also demonstrated that this integration to a general and common language can be of a great benefit. There is no need to learn a new syntax and new paradigms before starting to make sound with code; we just need to write programs in a familiar language. Crunching numbers to store in an audio table can be easily done with the numpy module. Creating audio software with a sophisticated graphical user interface become an easy task because there is already plenty of GUI modules available. Python is very powerful when comes the time to develop algorithms and pyo now provides a complete and high quality DSP toolbox to produce sound within Python.

7. REFERENCES

[1] R. Boulanger and V. Lazzarini. *The audio programming book*. MIT Press, Cambridge, Massachussets, 2010.

[2] J. McCartney. Supercollider: a new real time synthesis language. In *ICMC'96 Conference Proceedings*, pages 257–258. International Computer Music Association, 1996.

[3] J. McCartney. Continued evolution of the supercollider real time environment. In *ICMC'98 Conference Proceedings*, pages 133–136. International Computer Music Association, 1998.

[4] M. Puckette. Pure data: another integrated computer music environment. In *Conference Proceedings*, pages 37–41. Second Intercollege Computer Music Concerts, 1996.

[5] B. Vercoe and D. Ellis. Real-time csound: Software synthesis with sensing and control. In *ICMC'90 Conference Proceedings*, pages 209–211. International Computer Music Association, 1990.

[6] M. Wright, A. Freed, and AliMomeni. Opensound control: State of the art. In *NIME'03 Conference Proceedings*. Conference on New Interfaces for Musical Expression, 2003.