Pyo Documentation

Release 0.9.2

Olivier Bélanger

Feb 15, 2019

CONTENTS

Pyo is a Python module written in C to help digital signal processing script creation. It provides a complete set of classes to build audio softwares, compose algorithmic musics or simply explore audio processing with a simple, mature and powerful programming language.

CHAPTER

ONE

PARTS OF THE DOCUMENTATION

1.1 About pyo

Pyo is a Python module written in C to help digital signal processing script creation. It provides a complete set of classes to build audio softwares, compose algorithmic musics or simply explore audio processing with a simple, mature and powerful programming language.

Pyo contains classes for a wide variety of audio signal processing. With pyo, the user will be able to include signal processing chains directly in Python scripts or projects, and to manipulate them in real time through the interpreter. Tools in the pyo module offer primitives, like mathematical operations on audio signals, basic signal processing (filters, delays, synthesis generators, etc.), but also complex algorithms to create sound granulation and other creative audio manipulations. pyo supports the OSC protocol (Open Sound Control) to ease communications between softwares, and 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.

Pyo is developed by Olivier Bélanger < belangeo@gmail.com >

For questions and comments, please subscribe to the pyo-discuss mailing list.

To report a bug or to request a feature, use the issues tracker on github.

Sources and binaries can be downloaded at: http://ajaxsoundstudio.com/software/pyo/

1.2 Downloading the installer

Installers are available for Windows (XP/Vista/7/8/10) and for MacOS (from 10.6 to 10.12).

To download the latest pre-compiled version of pyo, go to the pyo's web page.

Under Debian and Fedora distros, you can get pyo from the package manager. The library's name is **python-pyo** or **python2-pyo** for Python 2.7 and **python3-pyo** for Python 3.5+.

If you are running Arch linux, the package is called **python2-pyo**. There is no package yet for Python 3.5+.

1.2.1 Content of the installer

The installer installs two distinct softwares on the system. First, it will install the pyo module (compiled for both single and double precision) and its dependencies under the current python distribution. Secondly, it will install E-Pyo, a simple text editor especially tuned to edit and run audio python script.

1.2.2 Pyo is a python module...

... which means that python must be present (version 2.7, 3.5 or 3.6 (recommended)) on the system. If python is not installed, you can download it on python.org.

Pyo also offers some GUI facilities to control or visualize the audio processing. If you want to use all of pyo's GUI features, you must install WxPython 3.0 (**classic** for python 2.7 and **phoenix** for python 3.5+), available on wxpython.org.

1.3 Compiling pyo from sources

Here is how you can compile pyo from sources on Linux and MacOS (if you are interested in the adventure of compiling pyo from sources on Windows, you can take a look at my personal notes in windows-7-build-routine.txt).

1.3.1 Dependencies

To compile pyo with all its features, you will need the following dependencies:

- Python 2.7 or 3.5 or higher. On Windows, install the 32-bit version of Python.
- WxPython 3.0.2.0 (classic) or 4.0.0 (phoenix, recommended)
- Portaudio
- Portmidi
- libsndfile
- liblo
- git (if you want the latest sources)

Please note that under MacOS you will need to install the **Apple's developer tools** to compile pyo.

1.3.2 Getting sources

You can download pyo's sources by checking out the source code here:

```
git clone https://github.com/belangeo/pyo.git
```

1.3.3 Compilation

Once you have all the required dependencies, go in pyo's directory:

```
cd path/to/pyo
```

And build the library:

```
sudo python setup.py install
```

You can customize your compilation by giving some flags to the command line.

Compilation flags

If you want to be able to use coreaudio (MacOS):

```
--use-coreaudio
```

If you want JACK support (Linux, MacOS):

```
--use-jack
```

If you want to be able to use a 64-bit pyo (All platforms, this is the sample resolution, not the architecture), this will build both single and double precisions:

```
--use-double
```

If you want to disable most of messages printed to the console:

```
--no-messages
```

If you want to compile external classes defined in pyo/externals folder:

```
--compile-externals
```

By default, debug symbols are off. If you want to compile pyo with debug symbols:

```
--debug
```

By default, optimizations are activated. If you want to compile pyo without optimizations:

```
--fast-compile
```

If you want to compile pyo with minimal dependencies (mostly for integrated use in a host environment):

```
--minimal
```

This will compile pyo without portaudio, portmidi and liblo support.

Compilation scripts

In the ./scripts folder, there is some alternate scripts to simplify the compilation process a little bit. These scripts will compile pyo for the version of python pointed to by the command *python*.

To compile both 32-bit and 64-bit resolutions on linux with jack support:

```
sudo sh scripts/compile_linux_withJack.sh
```

To compile both 32-bit and 64-bit resolutions on macOS without Jack support:

```
sudo sh scripts/compile_OSX.sh
```

To compile both 32-bit and 64-bit resolutions on macOS with Jack support (Jack headers must be present on the system):

```
sudo sh scripts/compile_OSX_withJack.sh
```

1.3.4 Debian & Ubuntu (apt-get)

Under Debian & Ubuntu you can type the following commands to get pyo up and running.

For Python 2.7

```
sudo apt-get install libjack-jackd2-dev libportmidi-dev portaudio19-dev liblo-dev␣
↪libsndfile-dev
sudo apt-get install python-dev python-tk python-imaging-tk python-wxgtk3.0
git clone https://github.com/belangeo/pyo.git
cd pyo
sudo python setup.py install --use-jack --use-double
```

- On Ubuntu system prior to vivid, wxpython 3.0 must be compiled from sources.

For Python 3.5 and higher

```
sudo apt-get install libjack-jackd2-dev libportmidi-dev portaudio19-dev liblo-dev␣
↪libsndfile-dev
sudo apt-get install python3-dev python3-tk python3-pil.imagetk python3-pip
git clone https://github.com/belangeo/pyo.git
cd pyo
sudo python3 setup.py install --use-jack --use-double
```

If you want to be able to use all of pyo's gui widgets, you will need wxPython Phoenix.

- To install wxPython with pip on linux, follow the instructions on the wxPython's downloads page.

1.3.5 MacOS (Homebrew)

Under macOS, it is very simple to build pyo from sources with the Homebrew package manager.

The first step is to install the official Python from python.org.

Second step, if you want to be able to use all of pyo's gui widgets, you will need wxPython Phoenix. Install with pip:

```
sudo pip3 install -U wxpython
```

The third step is to install Homebrew.

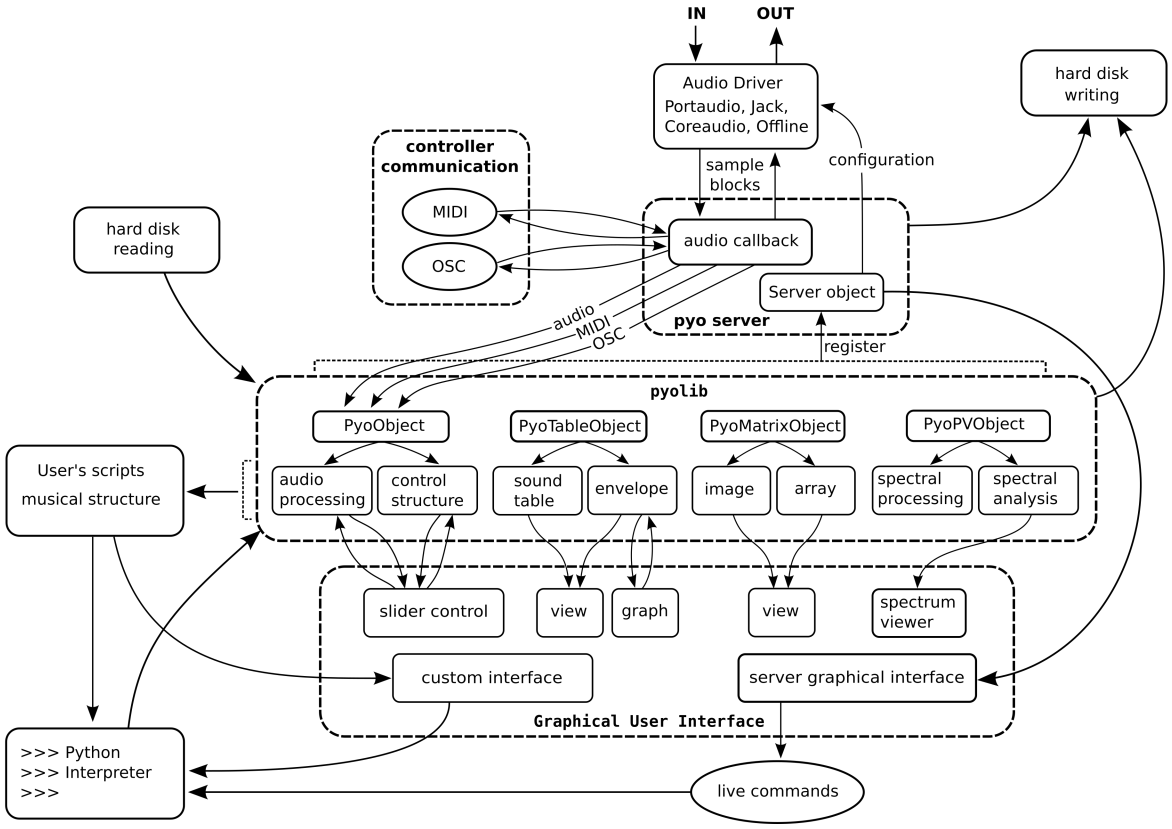
Finally, in a terminal window, install pyo's dependencies, clone and build pyo:

```
brew install liblo libsndfile portaudio portmidi
git clone https://github.com/belangeo/pyo.git
cd pyo
python setup.py install --use-coreaudio --use-double
```

1.4 Structure of the library

This diagram shows the internal structure of the library.

PYO : A Dedicated Module for Digital Signal Processing



1.5 Getting started

Here is quick introduction to Pyo. It assumes you already know Python and basics about OOP (Object-Oriented Programming).

1.5.1 The Pyo Server and GUI

The first thing you need to do to use Pyo is import the pyo python module and boot the server. This audio server will open audio and midi interfaces and will be ready to send to them the audio and MIDI produced by other pyo objects. You then need to make some sound:

```
>>> from pyo import *
>>> s = Server().boot()
>>> s.start()
>>> a = Sine(mul=0.01).out()
```

The *s* variable holds the Server instance, which has been booted, using the boot function. Booting the server includes opening audio and MIDI interfaces, and setting up the sample rate and number of channels, but the server will not be processing audio until its start() method is called. Then we create a Sine object, and store it in variable a, after calling its out method. The Sine class defines a Sine wave oscillator. The out method from this class connects the output of the oscillator to the server audio outputs. I have set the mul attribute of the Sine object to make sure you don't blow your ears when you play this, as the default amplitude multiplier is 1, i.e. a sine wave at the maximum amplitude before clipping! (But I'll talk about attributes later...) You can stop the server with:

```
>>> s.stop()
```

1.5.2 To interact or not to interact

If you tried the above script from an interactive python shell you would have heard a sine tone, but if you ran it from a python script non-interactively, you are probably asking yourself why you haven't heard anything. The reason is that the script has finished before the server has sent any audio to the outputs! So if you are using python non-interactively, the way to hear this example is:

```
from pyo import *
s = Server().boot()
s.start()
a = Sine(mul=0.01).out()
s.gui(locals())
```

In the last line, you can see a very handy method from the Server class, which creates a small control GUI for the current instance. The gui method for the Server object, keeps a script running and allows you to start and stop the server, control the output volume and record to an audio file the sound generated in the server. A handy feature of the server GUI is the interpreter text box in the bottom. From it you can send commands interactively to the interpreter, to start and stop objects, create or destroy them, etc.

1.5.3 Changing Object Characteristics

The Sine class constructor is defined as:

```
Sine(self, freq=1000, phase=0, mul=1, add=0)
```

So you can give it a frequency, starting phase, multiplier and DC offset value when you create it. Also, if you want to do without the server gui, you can use the server method start() from your script, but you might need to use the sleep function from the time module to have your script run the server for a while if you are running Python non-interactively:

```
from pyo import *
import time
s = Server().boot()
a = Sine(440, 0, 0.1).out()
s.start()
time.sleep(1)
s.stop()
```

Notice that you can set the parameters for Sine in the order in which they are defined, but you can also give the parameters a name if you want to leave the rest at their default:

```
a = Sine(mul=0.1).out()
```

Once the object has been created, you can modify its attributes using the access methods. For example, to modify the frequency of the a oscillator object after it has been created you can use:

```
a.setFreq(1000)
```

But you can also set the attributes directly:

```
a.freq = 1000
```

1.5.4 Chaining objects

Oscillators like the Sine class can be used as inputs to other classes, for example for frequency modulation:

```
from pyo import *
s = Server().boot()
mod = Sine(freq=6, mul=50)
a = Sine(freq=mod + 440, mul=0.1).out()
s.gui(locals())
```

You can create an envelope for a sine wave like this:

```
from pyo import *
s = Server().boot()
f = Adsr(attack=.01, decay=.2, sustain=.5, release=.1, dur=5, mul=.5)
a = Sine(mul=f).out()
f.play()
s.gui(locals())
```

1.5.5 Class examples

All Classes in Pyo come with an example which shows how it can be used. To execute the example you can do:

```
>>> from pyo import *
>>> example(Harmonizer)
```

This will show and execute the example for the Harmonizer class.

1.6 Configuring the audio output (especially on Windows)

Here is some tips to help you to configure the audio input/output on Windows. Some of these procedures are also valid for other systems.

1.6.1 How to choose the audio host api on Windows

Choosing the good audio API on Windows can turn out to be a real headache.

This document presents a script that will inspect your system and tell you if:

- Pyo can run in duplex mode. That means both audio input and output instead of output only.
- Pyo is able to connect to the different host APIs that are usually available on Windows.

In the hope that this will help you having a good experience with pyo under Windows!

https://github.com/belangeo/pyo/tree/master/scripts/win_audio_drivers_inspector.py

```
"""
Windows audio host inspector.

This script will check if pyo can run in duplex mode (both audio input and output)
and will test every host API to help the user in making his audio device choice.

"""
```

(continues on next page)

(continued from previous page)

```
import sys, time
from pyo import *

if sys.platform == "win32":
    host_apis = ["mme", "directsound", "asio", "wasapi", "wdm-ks"]
else:
    print("This program must be used on a windows system! Ciao!")
    exit()


print("* Checking for any available audio input... *")

input_names, input_indexes  = pa_get_input_devices()

print("* Checking audio output hosts... *")

s = Server(duplex=0)
s.verbosity = 0

host_results = []
for host in host_apis:
    print("* Testing %s... *" % host)
    try:
        s.reinit(buffersize=1024, duplex=0, winhost=host)
        s.boot().start()
        a = Sine(freq=440, mul=0.2).out()
        time.sleep(2)
        s.stop()
        s.shutdown()
        host_results.append(True)
    except:
        host_results.append(False)

print("\nResults")
print("-------\n")

if len(input_names):
    print("Duplex mode OK!")
    print("Initialize the Server with duplex=1 as argument.\n")
else:
    print("No input available. Duplex mode should be turned off.")
    print("Initialize the Server with duplex=0 as argument.\n")

for i, host in enumerate(host_apis):
    if host_results[i]:
        print("Host: %s  ==>  OK!" % host)
    else:
        print("Host: %s  ==>  Failed..." % host)

print("Initialize the Server with the desired host given to winhost argument.")

print("\nFinished!")
```

1.6.2 Tunning the Windows WASAPI driver

The Windows Audio Session API (WASAPI) is Microsoft's most modern method for talking with audio devices. It is available in Windows since Vista. Pyo's default host is DIRECTSOUND but you can change it to WASAPI by changing the *winhost* argument of the **Server** object. If the script above tells you:

> Host: wasapi ==> Failed...

there is some things you can do to make it work. Open the **Sound** window by double-clicking on the volume icon and choosing *Playback Devices*. Here, select your device and click on the *Properties* button. In the *advanced* tab, make sure that the sampling rate is the same that the one used by pyo (pyo defaults to 44100 Hz). You can check the exclusive mode box if you want, this will bypass the system mixer, default settings, and typically any effects provided by the audio driver.

Perform the same in the *recording* tab if you want to run pyo in duplex mode. If you got the message:

> No input available. Duplex mode should be turned off.

you'll have to make sure first that there is an available input device in that tab.

If you use a cheap soundcard (typically, any built in soundcard is not very good!), you may have to increase the buffer size of the pyo's Server in order to avoid glitches in the audio streams.

1.6.3 Server initialization examples

```
# sampling rate = 44100 Hz, buffer size = 256, channels = 2, full duplex, host =␣
↪DIRECTSOUND
s = Server()

# sampling rate = 48000 Hz, buffer size = 1024, channels = 2, full duplex, host =␣
↪DIRECTSOUND
s = Server(sr=48000, buffersize=1024)

# sampling rate = 48000 Hz, buffer size = 512, channels = 2, full duplex, host =␣
↪WASAPI
s = Server(sr=48000, buffersize=512, winhost="wasapi")

# sampling rate = 48000 Hz, buffer size = 512, channels = 2, output only, host = ASIO
s = Server(sr=48000, buffersize=512, duplex=0, winhost="asio")

# sampling rate = 96000 Hz, buffer size = 128, channels = 1, full duplex, host = ASIO
s = Server(sr=96000, nchnls=1, buffersize=128, duplex=1, winhost="asio")
```

1.6.4 Choosing a specific device

A single host API can target more than one available devices.

There is some useful functions that can help you in the choice of the audio device:

- **pa_list_host_apis()**: Prints the list of audio host APIs.
- **pa_list_devices()**: Prints the list of audio devices. The first column if the index of the device.
- **pa_get_default_input()**: Returns the index of the default input device.
- **pa_get_default_output()**: Returns the index of the default output device.
- **pa_get_default_devices_from_host(host)**: Returns the default input and output devices for a given audio host.

Run this code to see the current state of your audio setup:

```
from pyo import *

print("Audio host APIS:")
pa_list_host_apis()
pa_list_devices()
print("Default input device: %i" % pa_get_default_input())
print("Default output device: %i" % pa_get_default_output())
```

If the default device for the desired host is not the one you want, you can tell the Server which device you want to use with the *setInputDevice(x)* and *setOutputDevice(x)* methods. These methods take the index of the desired device and must be called before booting the Server. Ex:

```
from pyo import *

s = Server(duplex=0)
s.setOutputDevice(0)
s.boot()
```

1.7 How to improve performance of your pyo programs

This document lists various tips that help to improve the performance of your pyo programs.

1.7.1 Python tips

There is not much you can do at the Python level because once the script has finished its execution run, almost all computations are done in the C level of pyo. Nevertheless, there is these two tricks to consider:

Adjust the interpreter's "check interval"

You can change how often the interpreter checks for periodic things with *sys.setcheckinterval(interval)*. The defaults is 100, which means the check is performed every 100 Python virtual instructions. Setting it to a larger value may increase performance for programs using threads.

Use the subprocess or multiprocessing modules

You can use the subprocess or multiprocessing modules to spawn your processes on multiple processors. From the python docs:

> The multiprocessing package offers both local and remote concurrency, effectively side-stepping the Global Interpreter Lock by using subprocesses instead of threads. Due to this, the multiprocessing module allows the programmer to fully leverage multiple processors on a given machine. It runs on both Unix and Windows.

Here is a little example of using the multiprocessing module to spawn a lot of sine wave computations to multiple processors.

```
#!/usr/bin/env python
# encoding: utf-8
"""
Spawning lot of sine waves to multiple processes.
```

(continues on next page)

(continued from previous page)

```
From the command line, run the script with -i flag.

Call quit() to stop the workers and quit the program.

"""
import time
import multiprocessing
from random import uniform
from pyo import Server, SineLoop

class Group(multiprocessing.Process):
    def __init__(self, num_of_sines):
        super(Group, self).__init__()
        self.daemon = True
        self._terminated = False
        self.num_of_sines = num_of_sines

    def run(self):
        # All code that should run on a separated
        # core must be created in the run() method.
        self.server = Server()
        self.server.deactivateMidi()
        self.server.boot().start()

        freqs = [uniform(400,800) for i in range(self.num_of_sines)]
        self.oscs = SineLoop(freq=freqs, feedback=0.1, mul=.005).out()

        # Keeps the process alive...
        while not self._terminated:
            time.sleep(0.001)

        self.server.stop()

    def stop(self):
        self._terminated = True

if __name__ == '__main__':
    # Starts four processes playing 500 oscillators each.
    jobs = [Group(500) for i in range(4)]
    [job.start() for job in jobs]

    def quit():
        "Stops the workers and quit the program."
        [job.stop() for job in jobs]
        exit()
```

Avoid memory allocation after initialization

Dynamic memory allocation (malloc/calloc/realloc) tends to be nondeterministic; the time taken to allocate memory may not be predictable, making it inappropriate for real time systems. To be sure that the audio callback will run smoothly all the time, it is better to create all audio objects at the program's initialization and call their *stop()*, *play()*, *out()* methods when needed.

Be aware that a simple arithmetic operation involving an audio object will create a *Dummy* object (to hold the modified signal), thus will allocate memory for its audio stream AND add a processing task on the CPU. Run this simple example and watch the process's CPU growing:

```
from pyo import *
import random

s = Server().boot()

env = Fader(0.005, 0.09, 0.1, mul=0.2)
jit = Randi(min=1.0, max=1.02, freq=3)
sig = RCOsc(freq=[100,100], mul=env).out()

def change():
    freq = midiToHz(random.randrange(60, 72, 2))
    # Because `jit` is a PyoObject, both `freq+jit` and `freq-jit` will
    # create a `Dummy` object, for which a reference will be created and
    # saved in the `sig` object. The result is both memory and CPU
    # increase until something bad happens!
    sig.freq = [freq+jit, freq-jit]
    env.play()

pat = Pattern(change, time=0.125).play()

s.gui(locals())
```

An efficient version of this program should look like this:

```
from pyo import *
import random

s = Server().boot()

env = Fader(0.005, 0.09, 0.1, mul=0.2)
jit = Randi(min=1.0, max=1.02, freq=3)
# Create a `Sig` object to hold the frequency value.
frq = Sig(100)
# Create the `Dummy` objects only once at initialization.
sig = RCOsc(freq=[frq+jit, frq-jit], mul=env).out()

def change():
    freq = midiToHz(random.randrange(60, 72, 2))
    # Only change the `value` attribute of the Sig object.
    frq.value = freq
    env.play()

pat = Pattern(change, time=0.125).play()

s.gui(locals())
```

Don't do anything that can trigger the garbage collector

The garbage collector of python is another nondeterministic process. You should avoid doing anything that can trigger it. So, instead of deleting an audio object, which can turn out to delete many stream objects, you should just call its *stop()* method to remove it from the server's processing loop.

1.7.2 Pyo tips

Here is a list of tips specific to pyo that you should consider when trying to reduce the CPU consumption of your audio program.

Mix down before applying effects

It is very easy to over-saturate the CPU with pyo, especially if you use the multi-channel expansion feature. If your final output uses less channels than the number of audio streams in an object, don't forget to mix it down (call its *mix()* method) before applying effects on the sum of the signals.

Consider the following snippet, which create a chorus of 50 oscillators and apply a phasing effect on the resulting sound:

```
src = SineLoop(freq=[random.uniform(190,210) for i in range(50)],
               feedback=0.1, mul=0.01)
lfo = Sine(.25).range(200, 400)
phs = Phaser(src, freq=lfo, q=20, feedback=0.95).out()
```

This version uses around 47% of the CPU on my Thinkpad T430, i5 3320M @ 2.6GHz. The problem is that the 50 oscillators given in input of the Phaser object creates 50 identical Phaser objects, one for each oscillator. That is a big waste of CPU. The next version mixes the oscillators into a stereo stream before applying the effect and the CPU consumption drops to ~7% !

```
src = SineLoop(freq=[random.uniform(190,210) for i in range(50)],
               feedback=0.1, mul=0.01)
lfo = Sine(.25).range(200, 400)
phs = Phaser(src.mix(2), freq=lfo, q=20, feedback=0.95).out()
```

When costly effects are involved, this can have a very drastic impact on the CPU usage.

Stop your unused audio objects

Whenever you don't use an audio object (but you want to keep it for future uses), call its *stop()* method. This will inform the server to remove it from the computation loop. Setting the volume to 0 does not save CPU (everything is computed then multiplied by 0), the *stop()* method does. My own synth classes often looks like something like this:

```
class Glitchy:
    def __init__(self):
        self.feed = Lorenz(0.002, 0.8, True, 0.49, 0.5)
        self.amp = Sine(0.2).range(0.01, 0.3)
        self.src = SineLoop(1, self.feed, mul=self.amp)
        self.filt = ButLP(self.src, 10000)

    def play(self, chnl=0):
        self.feed.play()
        self.amp.play()
        self.src.play()
        self.filt.out(chnl)
        return self

    def stop(self):
        self.feed.stop()
        self.amp.stop()
        self.src.stop()
```

(continues on next page)

(continued from previous page)

```
        self.filt.stop()
        return self
```

Control attribute with numbers instead of PyoObjects

Objects internal processing functions are optimized when plain numbers are given to their attributes. Unless you really need audio control over some parameters, don't waste CPU cycles and give fixed numbers to every attribute that don't need to change over time. See this comparison:

```
n = Noise(.2)

# ~5% CPU
p1 = Phaser(n, freq=[100,105], spread=1.2, q=10,
            feedback=0.9, num=48).out()

# ~14% CPU
p2 = Phaser(n, freq=[100,105], spread=Sig(1.2), q=10,
            feedback=0.9, num=48).out()
```

Making the *spread* attribute of *p2* an audio signal causes the frequency of the 48 notches to be recalculated every sample, which can be a very costly process.

Check for denormal numbers

From wikipedia:

> In computer science, denormal numbers or denormalized numbers (now often called subnormal numbers) fill the underflow gap around zero in floating-point arithmetic. Any non-zero number with magnitude smaller than the smallest normal number is 'subnormal'.

The problem is that some processors compute denormal numbers very slowly, which makes grow the CPU consumption very quickly. The solution is to wrap the objects that are subject to denormals (any object with an internal recursive delay line, ie. filters, delays, reverbs, harmonizers, etc.) in a *Denorm* object. *Denorm* adds a little amount of noise, with a magnitude just above the smallest normal number, to its input. Of course, you can use the same noise for multiple denormalizations:

```
n = Noise(1e-24) # low-level noise for denormals

src = SfPlayer(SNDS_PATH+"/transparent.aif")
dly = Delay(src+n, delay=.1, feedback=0.8, mul=0.2).out()
rev = WGVerb(src+n).out()
```

Use a PyoObject when available

Always look first if a PyoObject does what you want, it will always be more efficient than the same process written from scratch.

This construct, although pedagogically valid, will never be more efficient, in term of CPU and memory usage, than a native PyoObject (Phaser) written in C.

```
a = BrownNoise(.02).mix(2).out()
```

(continues on next page)

(continued from previous page)

```
lfo = Sine(.25).range(.75, 1.25)
filters = []
for i in range(24):
    freq = rescale(i, xmin=0, xmax=24, ymin=100, ymax=10000)
    filter = Allpass2(a, freq=lfo*freq, bw=freq/2, mul=0.2).out()
    filters.append(filter)
```

It is also more efficient to use *Biquadx(stages=4)* than a cascade of four *Biquad* objects with identical arguments.

Avoid trigonometric computation

Avoid trigonometric functions computed at audio rate (*Sin*, *Cos*, *Tan*, *Atan2*, etc.), use simple approximations instead. For example, you can replace a clean *Sin/Cos* panning function with a cheaper one based on *Sqrt*:

```
# Heavier
pan = Linseg([(0,0), (2, 1)]).play()
left = Cos(pan * math.pi * 0.5, mul=0.5)
right = Sin(pan * math.pi * 0.5, mul=0.5)
a = Noise([left, right]).out()

# Cheaper
pan2 = Linseg([(0,0), (2, 1)]).play()
left2 = Sqrt(1 - pan2, mul=0.5)
right2 = Sqrt(pan2, mul=0.5)
a2 = Noise([left2, right2]).out()
```

Use approximations if absolute precision is not needed

When absolute precision is not really important, you can save precious CPU cycles by using approximations instead of the real function. *FastSine* is an approximation of the *sin* function that can be almost twice cheaper than a lookup table (Sine). I plan to add more approximations like this one in the future.

Re-use your generators

Some times it possible to use the same signal for parallel purposes. Let's study the next process:

```
# single white noise
noise = Noise()

# denormal signal
denorm = noise * 1e-24
# little jitter around 1 used to modulate frequency
jitter = noise * 0.0007 + 1.0
# excitation signal of the waveguide
source = noise * 0.7

env = Fader(fadein=0.001, fadeout=0.01, dur=0.015).play()
src = ButLP(source, freq=1000, mul=env)
wg = Waveguide(src+denorm, freq=100*jitter, dur=30).out()
```

Here the same white noise is used for three purposes at the same time. First, it is used to generate a denormal signal. Then, it is used to generate a little jitter applied to the frequency of the waveguide (that adds a little buzz to the string

sound) and finally, we use it as the excitation of the waveguide. This is surely cheaper than generating three different white noises without noticeable difference in the sound.

Leave 'mul' and 'add' attributes to their defaults when possible

There is an internal condition that bypass the object "post-processing" function when *mul=1* and *add=0*. It is a good practice to apply amplitude control in one place instead of messing with the *mul* attribute of each objects.

```
# wrong
n = Noise(mul=0.7)
bp1 = ButBP(n, freq=500, q=10, mul=0.5)
bp2 = ButBP(n, freq=1500, q=10, mul=0.5)
bp3 = ButBP(n, freq=2500, q=10, mul=0.5)
rev = Freeverb(bp1+bp2+bp3, size=0.9, bal=0.3, mul=0.7).out()

# good
n = Noise(mul=0.25)
bp1 = ButBP(n, freq=500, q=10)
bp2 = ButBP(n, freq=1500, q=10)
bp3 = ButBP(n, freq=2500, q=10)
rev = Freeverb(bp1+bp2+bp3, size=0.9, bal=0.3).out()
```

Avoid graphical updates

Even if they run in different threads, with different priorities, the audio callback and the graphical interface of a python program are parts of a unique process, sharing the same CPU. Don't use the Server's GUI if you don't need to see the meters or use the volume slider. Instead, you could start the script from command line with *-i* flag to leave the interpreter alive.

```
$ python -i myscript.py
```

List of CPU intensive objects

Here is a non-exhaustive list of the most CPU intensive objects of the library.

- **Analysis**
  - Yin
  - Centroid
  - Spectrum
  - Scope
- **Arithmetic**
  - Sin
  - Cos
  - Tan
  - Tanh
  - Atan2
- **Dynamic**

    - Compress
    - Gate
- **Special Effects**
    - Convolve
- **Prefix Expression Evaluator**
    - Expr
- **Filters**
    - Phaser
    - Vocoder
    - IRWinSinc
    - IRAverage
    - IRPulse
    - IRFM
- **Fast Fourier Transform**
    - CvlVerb
- **Phase Vocoder**
    - Almost every objects!
- **Signal Generators**
    - LFO
- **Matrix Processing**
    - MatrixMorph
- **Table Processing**
    - Granulator
    - Granule
    - Particule
    - OscBank
- **Utilities**
    - Resample

1.8 API documentation

1.8.1 Constants

- **PYO_VERSION** : string. Current version of pyo, as a string in the format "major.minor.change".
- **USE_DOUBLE** : boolean. True if using double precision (64-bit), False for single precision (32-bit).
- **WITH_EXTERNALS** : boolean. True if pyo was compiled with external classes. See *Compilation flags*.

- **SNDS_PATH** : string. Path to the pyo sound folder (located in the site-packages folder of the current Python installation.

1.8.2 Functions

Audio Setup

Set of functions to inspect the system's audio configuration.

Note: These functions are available only if pyo is built with portaudio support.

pa_get_version

pa_get_version()
Returns the version number, as an integer, of the current portaudio installation.

```
>>> v = pa_get_version()
>>> print(v)
1899
```

pa_get_version_text

pa_get_version_text()
Returns the textual description of the current portaudio installation.

```
>>> desc = pa_get_version_text()
>>> print(desc)
PortAudio V19-devel (built Oct 8 2012 16:25:16)
```

pa_count_host_apis

pa_count_host_apis()
Returns the number of host apis found by Portaudio.

```
>>> c = pa_count_host_apis()
>>> print(c)
1
```

pa_list_host_apis

pa_list_host_apis()
Prints a list of all host apis found by Portaudio.

```
>>> pa_list_host_apis()
index: 0, id: 5, name: Core Audio, num devices: 6, default in: 0, default out: 2
```

pa_get_default_host_api

pa_get_default_host_api()
Returns the index number of Portaudio's default host api.

```
>>> h = pa_get_default_host_api()
>>> print(h)
0
```

pa_get_default_devices_from_host

pa_get_default_devices_from_host(*host*)
Returns the default input and output devices for a given audio host.

This function can greatly help finding the device indexes (especially on Windows) to give to the server in order to use to desired audio host.

Args

host: string Name of the desired audio host. Possible hosts are:

- For Windows: mme, directsound, asio, wasapi or wdm-ks.
- For linux: alsa, oss, pulse or jack.
- For MacOS: core audio, jack or soundflower.

Return: (default_input_device, default_output_device)

pa_count_devices

pa_count_devices()
Returns the number of devices found by Portaudio.

```
>>> c = pa_count_devices()
>>> print(c)
6
```

pa_list_devices

pa_list_devices()
Prints a list of all devices found by Portaudio.

```
>>> pa_list_devices()
AUDIO devices:
0: IN, name: Built-in Microphone, host api index: 0, default sr: 44100 Hz,␣
↪latency: 0.001088 s
1: IN, name: Built-in Input, host api index: 0, default sr: 44100 Hz, latency: 0.
↪001088 s
2: OUT, name: Built-in Output, host api index: 0, default sr: 44100 Hz, latency:␣
↪0.001088 s
3: IN, name: UA-4FX, host api index: 0, default sr: 44100 Hz, latency: 0.010000 s
3: OUT, name: UA-4FX, host api index: 0, default sr: 44100 Hz, latency: 0.003061 s
4: IN, name: Soundflower (2ch), host api index: 0, default sr: 44100 Hz, latency:␣
↪0.010000 s
```

(continues on next page)

(continued from previous page)

```
4: OUT, name: Soundflower (2ch), host api index: 0, default sr: 44100 Hz,␣
↪latency: 0.000000 s
5: IN, name: Soundflower (16ch), host api index: 0, default sr: 44100 Hz,␣
↪latency: 0.010000 s
5: OUT, name: Soundflower (16ch), host api index: 0, default sr: 44100 Hz,␣
↪latency: 0.000000 s
```

pa_get_devices_infos

pa_get_devices_infos()

Returns informations about all devices found by Portaudio.

This function returns two dictionaries, one containing a dictionary for each input device and one containing a dictionary for each output device. Keys of outer dictionaries are the device index as returned by Portaudio. Keys of inner dictionaries are: 'name', 'host api index', 'default sr' and 'latency'.

```
>>> inputs, outputs = pa_get_devices_infos()
>>> print('- Inputs:')
>>> for index in sorted(inputs.keys()):
...     print('  Device index:', index)
...     for key in ['name', 'host api index', 'default sr', 'latency']:
...         print('    %s:' % key, inputs[index][key])
>>> print('- Outputs:')
>>> for index in sorted(outputs.keys()):
...     print('  Device index:', index)
...     for key in ['name', 'host api index', 'default sr', 'latency']:
...         print('    %s:' % key, outputs[index][key])
```

pa_get_input_devices

pa_get_input_devices()

Returns input devices (device names, device indexes) found by Portaudio.

device names is a list of strings and *device indexes* is a list of the actual Portaudio index of each device.

```
>>> ins = pa_get_input_devices()
>>> print(ins)
(['Built-in Microphone', 'Built-in Input', 'UA-4FX', 'Soundflower (2ch)',
↪'Soundflower (16ch)'], [0, 1, 3, 4, 5])
```

pa_get_output_devices

pa_get_output_devices()

Returns output devices (device names, device indexes) found by Portaudio.

device names is a list of strings and *device indexes* is a list of the actual Portaudio index of each device.

```
>>> outs = pa_get_output_devices()
>>> print(outs)
(['Built-in Output', 'UA-4FX', 'Soundflower (2ch)', 'Soundflower (16ch)'], [2, 3,␣
↪4, 5])
```

pa_get_default_input

pa_get_default_input()
Returns the index number of Portaudio's default input device.

```
>>> names, indexes = pa_get_input_devices()
>>> name = names[indexes.index(pa_get_default_input())]
>>> print(name)
'Built-in Microphone'
```

pa_get_default_output

pa_get_default_output()
Returns the index number of Portaudio's default output device.

```
>>> names, indexes = pa_get_output_devices()
>>> name = names[indexes.index(pa_get_default_output())]
>>> print(name)
'UA-4FX'
```

pa_get_input_max_channels

pa_get_input_max_channels(*x*)
Retrieve the maximum number of input channels for the specified device.

Args

x: int Device index as listed by Portaudio (see pa_get_input_devices).

```
>>> device = 'HDA Intel PCH: STAC92xx Analog (hw:0,0)'
>>> dev_list, dev_index =  pa_get_output_devices()
>>> dev = dev_index[dev_list.index(device)]
>>> print('Device index:', dev)
>>> maxouts = pa_get_output_max_channels(dev)
>>> maxins = pa_get_input_max_channels(dev)
>>> print('Max outputs', maxouts)
>>> print('Max inputs:', maxins)
>>> if maxouts >= 2 and maxins >= 2:
...     nchnls = 2
>>> else:
...     nchnls = 1
```

pa_get_output_max_channels

pa_get_output_max_channels(*x*)
Retrieve the maximum number of output channels for the specified device.

Args

x: int Device index as listed by Portaudio (see pa_get_output_devices).

```
>>> device = 'HDA Intel PCH: STAC92xx Analog (hw:0,0)'
>>> dev_list, dev_index =  pa_get_output_devices()
>>> dev = dev_index[dev_list.index(device)]
>>> print('Device index:', dev)
>>> maxouts = pa_get_output_max_channels(dev)
>>> maxins = pa_get_input_max_channels(dev)
>>> print('Max outputs:', maxouts)
>>> print('Max inputs:', maxins)
>>> if maxouts >= 2 and maxins >= 2:
...     nchnls = 2
>>> else:
...     nchnls = 1
```

Midi Setup

Set of functions to inspect the system's midi configuration.

Note: These functions are available only if pyo is built with portmidi support.

pm_get_default_output

pm_get_default_output()
Returns the index number of Portmidi's default output device.

```
>>> names, indexes = pm_get_output_devices()
>>> name = names[indexes.index(pm_get_default_output())]
>>> print(name)
'IAC Driver Bus 1'
```

pm_get_default_input

pm_get_default_input()
Returns the index number of Portmidi's default input device.

```
>>> names, indexes = pm_get_input_devices()
>>> name = names[indexes.index(pm_get_default_input())]
>>> print(name)
'IAC Driver Bus 1'
```

pm_get_output_devices

pm_get_output_devices()
Returns midi output devices (device names, device indexes) found by Portmidi.

device names is a list of strings and *device indexes* is a list of the actual Portmidi index of each device.

```
>>> outs = pm_get_output_devices()
>>> print(outs)
(['IAC Driver Bus 1', 'to MaxMSP 1', 'to MaxMSP 2'], [3, 4, 5])
```

pm_get_input_devices

pm_get_input_devices()
Returns midi input devices (device names, device indexes) found by Portmidi.

device names is a list of strings and *device indexes* is a list of the actual Portmidi index of each device.

```
>>> ins = pm_get_input_devices()
>>> print(ins)
(['IAC Driver Bus 1', 'from MaxMSP 1', 'from MaxMSP 2'], [0, 1, 2])
```

pm_list_devices

pm_list_devices()
Prints a list of all devices found by Portmidi.

```
>>> pm_list_devices()
MIDI devices:
0: IN, name: IAC Driver Bus 1, interface: CoreMIDI
1: IN, name: from MaxMSP 1, interface: CoreMIDI
2: IN, name: from MaxMSP 2, interface: CoreMIDI
3: OUT, name: IAC Driver Bus 1, interface: CoreMIDI
4: OUT, name: to MaxMSP 1, interface: CoreMIDI
5: OUT, name: to MaxMSP 2, interface: CoreMIDI
```

pm_count_devices

pm_count_devices()
Returns the number of devices found by Portmidi.

```
>>> c = pm_count_devices()
>>> print(c)
6
```

Soundfile

sndinfo

sndinfo(*path*, *print=False*)
Retrieve informations about a soundfile.

Prints the infos of the given soundfile to the console and returns a tuple containing:

(number of frames, duration in seconds, sampling rate, number of channels, file format, sample type)

Args

path: string Path of a valid soundfile.

print: boolean, optional If True, sndinfo will print sound infos to the console. Defaults to False.

```
>>> path = SNDS_PATH + '/transparent.aif'
>>> print(path)
/usr/lib/python2.7/dist-packages/pyo/lib/snds/transparent.aif
>>> info = sndinfo(path)
>>> print(info)
(29877, 0.6774829931972789, 44100.0, 1, 'AIFF', '16 bit int')
```

savefile

savefile(*samples*, *path*, *sr=44100*, *channels=1*, *fileformat=0*, *sampletype=0*, *quality=0.4*)
Creates an audio file from a list of floats.

Args

samples: list of floats List of samples data, or list of list of samples data if more than 1 channels.

path: string Full path (including extension) of the new file.

sr: int, optional Sampling rate of the new file. Defaults to 44100.

channels: int, optional Number of channels of the new file. Defaults to 1.

fileformat: int, optional

Format type of the new file. Defaults to 0. Supported formats are:

0. WAVE - Microsoft WAV format (little endian) {.wav, .wave}
1. AIFF - Apple/SGI AIFF format (big endian) {.aif, .aiff}
2. AU - Sun/NeXT AU format (big endian) {.au}
3. RAW - RAW PCM data {no extension}
4. SD2 - Sound Designer 2 {.sd2}
5. FLAC - FLAC lossless file format {.flac}
6. CAF - Core Audio File format {.caf}
7. OGG - Xiph OGG container {.ogg}

sampletype ; int, optional Bit depth encoding of the audio file. Defaults to 0. SD2 and FLAC only support 16 or 24 bit int. Supported types are:

0. 16 bit int
1. 24 bit int
2. 32 bit int
3. 32 bit float
4. 64 bit float
5. U-Law encoded
6. A-Law encoded

quality: float, optional The encoding quality value, between 0.0 (lowest quality) and 1.0 (highest quality). This argument has an effect only with FLAC and OGG compressed formats. Defaults to 0.4.

```
>>> from random import uniform
>>> import os
>>> home = os.path.expanduser('~')
>>> sr, dur, chnls, path = 44100, 5, 2, os.path.join(home, 'noise.aif')
>>> samples = [[uniform(-0.5,0.5) for i in range(sr*dur)] for i in range(chnls)]
>>> savefile(samples=samples, path=path, sr=sr, channels=chnls, fileformat=1,␣
↪sampletype=1)
```

savefileFromTable

savefileFromTable(*table*, *path*, *fileformat=0*, *sampletype=0*, *quality=0.4*)

Creates an audio file from the content of a table.

Args

table: PyoTableObject Table from which to retrieve the samples to write.

path: string Full path (including extension) of the new file.

fileformat: int, optional

Format type of the new file. Defaults to 0. Supported formats are:

0. WAVE - Microsoft WAV format (little endian) {.wav, .wave}
1. AIFF - Apple/SGI AIFF format (big endian) {.aif, .aiff}
2. AU - Sun/NeXT AU format (big endian) {.au}
3. RAW - RAW PCM data {no extension}
4. SD2 - Sound Designer 2 {.sd2}
5. FLAC - FLAC lossless file format {.flac}
6. CAF - Core Audio File format {.caf}
7. OGG - Xiph OGG container {.ogg}

sampletype ; int, optional Bit depth encoding of the audio file. Defaults to 0. SD2 and FLAC only support 16 or 24 bit int. Supported types are:

0. 16 bit int
1. 24 bit int
2. 32 bit int
3. 32 bit float
4. 64 bit float
5. U-Law encoded
6. A-Law encoded

quality: float, optional The encoding quality value, between 0.0 (lowest quality) and 1.0 (highest quality). This argument has an effect only with FLAC and OGG compressed formats. Defaults to 0.4.

```
>>> import os
>>> home = os.path.expanduser('~')
>>> path1 = SNDS_PATH + '/transparent.aif'
```

(continues on next page)

(continued from previous page)

```
>>> path2 = os.path.join(home, '/transparent2.aif')
>>> t = SndTable(path1)
>>> savefileFromTable(table=t, path=path, fileformat=1, sampletype=1)
```

Resampling

upsamp

upsamp (*path*, *outfile*, *up=4*, *order=128*)

Increases the sampling rate of an audio file.

Args

path: string Full path (including extension) of the audio file to convert.

outfile: string Full path (including extension) of the new file.

up: int, optional Upsampling factor. Defaults to 4.

order: int, optional Length, in samples, of the anti-aliasing lowpass filter. Defaults to 128.

```
>>> import os
>>> home = os.path.expanduser('~')
>>> f = SNDS_PATH+'/transparent.aif'
>>> # upsample a signal 3 times
>>> upfile = os.path.join(home, 'trans_upsamp_2.aif')
>>> upsamp(f, upfile, 2, 256)
>>> # downsample the upsampled signal 3 times
>>> downfile = os.path.join(home, 'trans_downsamp_3.aif')
>>> downsamp(upfile, downfile, 3, 256)
```

downsamp

downsamp (*path*, *outfile*, *down=4*, *order=128*)

Decreases the sampling rate of an audio file.

Args

path: string Full path (including extension) of the audio file to convert.

outfile: string Full path (including extension) of the new file.

down: int, optional Downsampling factor. Defaults to 4.

order: int, optional Length, in samples, of the anti-aliasing lowpass filter. Defaults to 128.

```
>>> import os
>>> home = os.path.expanduser('~')
>>> f = SNDS_PATH+'/transparent.aif'
>>> # upsample a signal 3 times
>>> upfile = os.path.join(home, 'trans_upsamp_2.aif')
>>> upsamp(f, upfile, 2, 256)
>>> # downsample the upsampled signal 3 times
>>> downfile = os.path.join(home, 'trans_downsamp_3.aif')
>>> downsamp(upfile, downfile, 3, 256)
```

Conversions

midiToHz

midiToHz(*x*)

Converts a midi note value to frequency in Hertz.

Args

x: int or float Midi note. *x* can be a number, a list or a tuple, otherwise the function returns None.

```
>>> a = (48, 60, 62, 67)
>>> b = midiToHz(a)
>>> print(b)
(130.8127826503271, 261.62556530066814, 293.66476791748823, 391.9954359818656)
>>> a = [48, 60, 62, 67]
>>> b = midiToHz(a)
>>> print(b)
[130.8127826503271, 261.62556530066814, 293.66476791748823, 391.9954359818656]
>>> b = midiToHz(60.0)
>>> print(b)
261.625565301
```

midiToTranspo

midiToTranspo(*x*)

Converts a midi note value to transposition factor (central key = 60).

Args

x: int or float Midi note. *x* can be a number, a list or a tuple, otherwise the function returns None.

```
>>> a = (48, 60, 62, 67)
>>> b = midiToTranspo(a)
>>> print(b)
    (0.49999999999997335, 1.0, 1.122462048309383, 1.4983070768767281)
>>> a = [48, 60, 62, 67]
>>> b = midiToTranspo(a)
>>> print(b)
[0.49999999999997335, 1.0, 1.122462048309383, 1.4983070768767281]
>>> b = midiToTranspo(60.0)
>>> print(b)
1.0
```

sampsToSec

sampsToSec(*x*)

Returns the duration in seconds equivalent to the number of samples given as an argument.

Args

x: int or float Duration in samples. *x* can be a number, a list or a tuple, otherwise function returns None.

```
>>> s = Server().boot()
>>> a = (64, 128, 256)
>>> b = sampsToSec(a)
>>> print(b)
(0.0014512471655328798, 0.0029024943310657597, 0.0058049886621315194)
>>> a = [64, 128, 256]
>>> b = sampsToSec(a)
>>> print(b)
[0.0014512471655328798, 0.0029024943310657597, 0.0058049886621315194]
>>> b = sampsToSec(8192)
>>> print(b)
0.185759637188
```

secToSamps

secToSamps(*x*)

Returns the number of samples equivalent to the duration in seconds given as an argument.

Args

x: int or float Duration in seconds. *x* can be a number, a list or a tuple, otherwise function returns None.

```
>>> s = Server().boot()
>>> a = (0.1, 0.25, 0.5, 1)
>>> b = secToSamps(a)
>>> print(b)
(4410, 11025, 22050, 44100)
>>> a = [0.1, 0.25, 0.5, 1]
>>> b = secToSamps(a)
>>> print(b)
[4410, 11025, 22050, 44100]
>>> b = secToSamps(2.5)
>>> print(b)
110250
```

linToCosCurve

linToCosCurve(*data, yrange=[0, 1], totaldur=1, points=1024, log=False*)

Creates a cosinus interpolated curve from a list of points.

A point is a tuple (or a list) of two floats, time and value.

Args

data: list of points Set of points between which will be inserted interpolated segments.

yrange: list of 2 floats, optional Minimum and maximum values on the Y axis. Defaults to [0., 1.].

totaldur: float, optional X axis duration. Defaults to 1.

points: int, optional Number of points in the output list. Defaults to 1024.

log: boolean, optional Set this value to True if the Y axis has a logarithmic scale. Defaults to False

```
>>> s = Server().boot()
>>> a = [(0,0), (0.25, 1), (0.33, 1), (1,0)]
>>> b = linToCosCurve(a, yrange=[0, 1], totaldur=1, points=8192)
>>> t = DataTable(size=len(b), init=[x[1] for x in b])
>>> t.view()
```

rescale

rescale(*data*, *xmin=0.0*, *xmax=1.0*, *ymin=0.0*, *ymax=1.0*, *xlog=False*, *ylog=False*)

Converts values from an input range to an output range.

This function takes data in the range *xmin* - *xmax* and returns corresponding values in the range *ymin* - *ymax*.

data can be either a number or a list. Return value is of the same type as *data* with all values rescaled.

Argss

data: float or list of floats Values to convert.

xmin: float, optional Minimum value of the input range.

xmax: float, optional Maximum value of the input range.

ymin: float, optional Minimum value of the output range.

ymax: float, optional Maximum value of the output range.

xlog: boolean, optional Set this argument to True if the input range has a logarithmic scaling.

ylog: boolean, optional Set this argument to True if the output range has a logarithmic scaling.

```
>>> a = 0.5
>>> b = rescale(a, 0, 1, 20, 20000, False, True)
>>> print(b)
632.453369141
>>> a = [0, .4, .8]
>>> b = rescale(a, 0, 1, 20, 20000, False, True)
>>> print(b)
[20.000001907348633, 316.97738647460938, 5023.7705078125]
```

floatmap

floatmap(*x*, *min=0.0*, *max=1.0*, *exp=1.0*)

Converts values from a 0-1 range to an output range.

This function takes data in the range *0* - *1* and returns corresponding values in the range *min* - *max*.

Argss

x: float Value to convert, in the range 0 to 1.

min: float, optional Minimum value of the output range. Defaults to 0.

max: float, optional Maximum value of the output range. Defaults to 1.

exp: float, optional Power factor (1 (default) is linear, les than 1 is logarithmic, greter than 1 is exponential).

```
>>> a = 0.5
>>> b = floatmap(a, 0, 1, 4)
>>> print(b)
0.0625
```

distanceToSegment

distanceToSegment(*p*, *p1*, *p2*, *xmin=0.0*, *xmax=1.0*, *ymin=0.0*, *ymax=1.0*, *xlog=False*, *ylog=False*)

Find the distance from a point to a line or line segment.

This function returns the shortest distance from a point to a line segment normalized between 0 and 1.

A point is a tuple (or a list) of two floats, time and value. *p* is the point for which to find the distance from line *p1* to *p2*.

Args

p: list or tuple Point for which to find the distance.

p1: list or tuple First point of the segment.

p2: list or tuple Second point of the segment.

xmin: float, optional Minimum value on the X axis.

xmax: float, optional Maximum value on the X axis.

ymin: float, optional Minimum value on the Y axis.

ymax: float, optional Maximum value on the Y axis.

xlog: boolean, optional Set this argument to True if X axis has a logarithmic scaling.

ylog: boolean, optional Set this argument to True if Y axis has a logarithmic scaling.

reducePoints

reducePoints(*pointlist*, *tolerance=0.02*)

Douglas-Peucker curve reduction algorithm.

This function receives a list of points as input and returns a simplified list by eliminating redundancies.

A point is a tuple (or a list) of two floats, time and value. A list of points looks like:

[(0, 0), (0.1, 0.7), (0.2, 0.5), ...]

Args

pointlist: list of lists or list of tuples List of points (time, value) to filter.

tolerance: float, optional Normalized distance threshold under which a point is excluded from the list. Defaults to 0.02.

Server Queries

serverCreated

serverCreated()

Returns True if a Server object is already created, otherwise, returns False.

```
>>> print(serverCreated())
False
>>> s = Server()
>>> print(serverCreated())
True
```

serverBooted

serverBooted()
Returns True if an already created Server is booted, otherwise, returns False.

```
>>> s = Server()
>>> print(serverBooted())
False
>>> s.boot()
>>> print(serverBooted())
True
```

Utilities

example

example(*cls*, *dur=5*, *toprint=True*, *double=False*)
Execute the documentation example of the object given as an argument.

Args

cls: PyoObject class or string Class reference of the desired object example. If this argument is the string of the full path of an example (as returned by the getPyoExamples() function), it will be executed.

dur: float, optional Duration of the example.

toprint: boolean, optional If True, the example script will be printed to the console. Defaults to True.

double: boolean, optional If True, force the example to run in double precision (64-bit) Defaults to False.

```
>>> example(Sine)
```

class_args

class_args(*cls*)
Returns the init line of a class reference.

This function takes a class reference (not an instance of that class) as input and returns the init line of that class with the default values.

Args

cls: PyoObject class Class reference of the desired object's init line.

```
>>> print(class_args(Sine))
>>> 'Sine(freq=1000, phase=0, mul=1, add=0)'
```

getVersion

getVersion()
Returns the version number of the current pyo installation.

This function returns the version number of the current pyo installation as a 3-ints tuple (major, minor, rev).

The returned tuple for version '0.4.1' will look like: (0, 4, 1)

```
>>> print(getVersion())
>>> (0, 5, 1)
```

getPrecision

getPrecision()
Returns the current sample precision as an integer.

This function returns the current sample precision as an integer, either 32 for 32-bit (single) or 64 for 64-bit (double).

getPyoKeywords

getPyoKeywords()
Returns a list of every keywords (classes and functions) of pyo.

```
>>> keywords = getPyoKeywords()
```

getPyoExamples

getPyoExamples(*fullpath=False*)
Returns a listing of the examples, as a dictionary, installed with pyo.

Args

fullpath: boolean If True, the full path of each file is returned. Otherwise, only the filenames are listed.

```
>>> examples = getPyoExamples()
```

withPortaudio

withPortaudio()
Returns True if pyo is built with portaudio support.

withPortmidi

withPortmidi()
Returns True if pyo is built with portmidi support.

withJack

withJack()
Returns True if pyo is built with jack support.

withCoreaudio

withCoreaudio()
Returns True if pyo is built with coreaudio support.

withOSC

withOSC()
Returns True if pyo is built with OSC (Open Sound Control) support.

convertStringToSysEncoding

convertStringToSysEncoding(*strng*)
Convert a string to the current platform file system encoding.

Returns the new encoded string.

Args

strng: string String to convert.

convertArgsToLists

convertArgsToLists(**args*)
Convert all arguments to list if not already a list or a PyoObjectBase. Return new args and maximum list length.

wrap

wrap(*arg*, *i*)
Return value at position *i* from *arg* with wrap around *arg* length.

1.8.3 Alphabetical class reference

- *AToDB* : Returns the decibel equivalent of an amplitude value.
- *Abs* : Performs an absolute function on audio signal.
- *Adsr* : Attack - Decay - Sustain - Release envelope generator.
- *Allpass2* : Second-order phase shifter allpass.

- *AllpassWG* : Out of tune waveguide model with a recursive allpass network.
- *Allpass* : Delay line based allpass filter.
- *Atan2* : Computes the principal value of the arc tangent of b/a.
- *AtanTable* : Generates an arctangent transfert function.
- *Atone* : A first-order recursive high-pass filter with variable frequency response.
- *AttackDetector* : Audio signal onset detection.
- *Average* : Moving average filter.
- *Balance* : Adjust rms power of an audio signal according to the rms power of another.
- *BandSplit* : Splits an input signal into multiple frequency bands.
- *Beat* : Generates algorithmic trigger patterns.
- *Bendin* : Get the current value of the pitch bend controller.
- *Between* : Informs when an input signal is contained in a specified range.
- *Biquad* : A sweepable general purpose biquadratic digital filter.
- *Biquada* : A general purpose biquadratic digital filter (floating-point arguments).
- *Biquadx* : A multi-stages sweepable general purpose biquadratic digital filter.
- *Blit* : Band limited impulse train synthesis.
- *BrownNoise* : A brown noise generator.
- *ButBP* : A second-order Butterworth bandpass filter.
- *ButBR* : A second-order Butterworth band-reject filter.
- *ButHP* : A second-order Butterworth highpass filter.
- *ButLP* : A second-order Butterworth lowpass filter.
- *CallAfter* : Calls a Python function after a given time.
- *CarToPol* : Performs the cartesian to polar conversion.
- *Ceil* : Rounds to smallest integral value greater than or equal to the input signal.
- *Centroid* : Computes the spectral centroid of an input signal.
- *CentsToTranspo* : Returns the transposition factor equivalent of a given cents value.
- *Change* : Sends trigger that informs when input value has changed.
- *ChebyTable* : Chebyshev polynomials of the first kind.
- *ChenLee* : Chaotic attractor for the Chen-Lee system.
- *Choice* : Periodically choose a new value from a user list.
- *Chorus* : 8 modulated delay lines chorus processor.
- *Clean_objects* : Stops and deletes PyoObjects after a given time.
- *Clip* : Clips a signal to a predefined limit.
- *Cloud* : Generates random triggers.
- *Compare* : Comparison object.
- *ComplexRes* : Complex one-pole resonator filter.

- *Compress* : Reduces the dynamic range of an audio signal.
- *ControlRead* : Reads control values previously stored in text files.
- *ControlRec* : Records control values and writes them in a text file.
- *Convolve* : Implements filtering using circular convolution.
- *CosLogTable* : Construct a table from logarithmic-cosine segments in breakpoint fashion.
- *CosTable* : Construct a table from cosine interpolated segments.
- *Cos* : Performs a cosine function on audio signal.
- *Count* : Counts integers at audio rate.
- *Counter* : Integer count generator.
- *CrossFM* : Cross frequency modulation generator.
- *CtlScan2* : Scan the Midi channel and controller number in input.
- *CtlScan* : Scan the Midi controller's number in input.
- *CurveTable* : Construct a table from curve interpolated segments.
- *CvlVerb* : Convolution based reverb.
- *DBToA* : Returns the amplitude equivalent of a decibel value.
- *DCBlock* : Implements the DC blocking filter.
- *DataTable* : Create an empty table ready for data recording.
- *Degrade* : Signal quality reducer.
- *Delay1* : Delays a signal by one sample.
- *Delay* : Sweepable recursive delay.
- *Denorm* : Mixes low level noise to an input signal.
- *Disto* : Kind of Arc tangent distortion.
- *Div* : Divides a by b.
- *Dummy* : Dummy object used to perform arithmetics on PyoObject.
- *EQ* : Equalizer filter.
- *Euclide* : Euclidean rhythm generator.
- *ExpTable* : Construct a table from exponential interpolated segments.
- *Exp* : Calculates the value of e to the power of x.
- *Expand* : Expand the dynamic range of an audio signal.
- *Expr* : Prefix audio expression evaluator.
- *Expr* : Prefix audio expression evaluator.
- *Expseg* : Draw a series of exponential segments between specified break-points.
- *FFT* : Fast Fourier Transform.
- *FM* : A simple frequency modulation generator.
- *FToM* : Returns the midi note equivalent to a frequency in Hz.
- *Fader* : Fadein - fadeout envelope generator.

- *FastSine* : A fast sine wave approximation using the formula of a parabola.
- *Floor* : Rounds to largest integral value not greater than audio signal.
- *Follower2* : Envelope follower with different attack and release times.
- *Follower* : Envelope follower.
- *FourBand* : Splits an input signal into four frequency bands.
- *FrameAccum* : Accumulates the phase differences between successive frames.
- *FrameDelta* : Computes the phase differences between successive frames.
- *Freeverb* : Implementation of Jezar's Freeverb.
- *FreqShift* : Frequency shifting using single sideband amplitude modulation.
- *Gate* : Allows a signal to pass only when its amplitude is above a set threshold.
- *Granulator* : Granular synthesis generator.
- *Granule* : Another granular synthesis generator.
- *HRTF* : Head-Related Transfert Function 3D spatialization.
- *HannTable* : Generates Hanning window function.
- *HarmTable* : Harmonic waveform generator.
- *Harmonizer* : Generates harmonizing voices in synchrony with its audio input.
- *Hilbert* : Hilbert transform.
- *IFFT* : Inverse Fast Fourier Transform.
- *IRAverage* : Moving average filter using circular convolution.
- *IRFM* : Filters a signal with a frequency modulation spectrum using circular convolution.
- *IRPulse* : Comb-like filter using circular convolution.
- *IRWinSinc* : Windowed-sinc filter using circular convolution.
- *InputFader* : Audio streams crossfader.
- *Input* : Read from a numbered channel in an external audio signal.
- *Interp* : Interpolates between two signals.
- *Iter* : Triggers iterate over a list of values.
- *LFO* : Band-limited Low Frequency Oscillator with different wave shapes.
- *LinTable* : Construct a table from segments of straight lines in breakpoint fashion.
- *Linseg* : Draw a series of line segments between specified break-points.
- *Log10* : Performs a base 10 log function on audio signal.
- *Log2* : Performs a base 2 log function on audio signal.
- *LogTable* : Construct a table from logarithmic segments in breakpoint fashion.
- *Log* : Performs a natural log function on audio signal.
- *LogiMap* : Random generator based on the logistic map.
- *Lookup* : Uses table to do waveshaping on an audio signal.
- *Looper* : Crossfading looper.

- *Lorenz* : Chaotic attractor for the Lorenz system.
- *MToF* : Returns the frequency (Hz) equivalent to a midi note.
- *MToT* : Returns the transposition factor equivalent to a midi note.
- *MatrixMorph* : Morphs between multiple PyoMatrixObjects.
- *MatrixPointer* : Matrix reader with control on the 2D pointer position.
- *MatrixRecLoop* : MatrixRecLoop records samples in loop into a previously created NewMatrix.
- *MatrixRec* : MatrixRec records samples into a previously created NewMatrix.
- *Max* : Outputs the maximum of two values.
- *Metro* : Generates isochronous trigger signals.
- *MidiAdsr* : Midi triggered ADSR envelope generator.
- *MidiDelAdsr* : Midi triggered ADSR envelope generator with pre-delay.
- *MidiLinseg* : Line segments trigger.
- *MidiListener* : Self-contained midi listener thread.
- *Midictl* : Get the current value of a Midi controller.
- *Min* : Outputs the minimum of two values.
- *Mirror* : Reflects the signal that exceeds the *min* and *max* thresholds.
- *Mix* : Mix audio streams to arbitrary number of streams.
- *Mixer* : Audio mixer.
- *MoogLP* : A fourth-order resonant lowpass filter.
- *MultiBand* : Splits an input signal into multiple frequency bands.
- *NewMatrix* : Create a new matrix ready for recording.
- *NewTable* : Create an empty table ready for recording.
- *NextTrig* : A trigger in the second stream opens a gate only for the next one in the first stream.
- *Noise* : A white noise generator.
- *NoteinRead* : Reads Notein values previously stored in text files.
- *NoteinRec* : Records Notein inputs and writes them in a text file.
- *Notein* : Generates Midi note messages.
- *OscBank* : Any number of oscillators reading a waveform table.
- *OscDataReceive* : Receives data values over a network via the Open Sound Control protocol.
- *OscDataSend* : Sends data values over a network via the Open Sound Control protocol.
- *OscListReceive* : Receives list of values over a network via the Open Sound Control protocol.
- *OscListener* : Self-contained OSC listener thread.
- *OscLoop* : A simple oscillator with feedback reading a waveform table.
- *OscReceive* : Receives values over a network via the Open Sound Control protocol.
- *OscSend* : Sends values over a network via the Open Sound Control protocol.
- *OscTrig* : An oscillator reading a waveform table with sample accurate reset signal.

- *Osc* : A simple oscillator reading a waveform table.
- *PVAddSynth* : Phase Vocoder additive synthesis object.
- *PVAmpMod* : Performs frequency independent amplitude modulations.
- *PVAnal* : Phase Vocoder analysis object.
- *PVBufLoops* : Phase vocoder buffer with bin independent speed playback.
- *PVBufTabLoops* : Phase vocoder buffer with bin independent speed playback.
- *PVBuffer* : Phase vocoder buffer and playback with transposition.
- *PVCross* : Performs cross-synthesis between two phase vocoder streaming object.
- *PVDelay* : Spectral delays.
- *PVFilter* : Spectral filter.
- *PVFreqMod* : Performs frequency independent frequency modulations.
- *PVGate* : Spectral gate.
- *PVMix* : Mix the most prominent components from two phase vocoder streaming objects.
- *PVMorph* : Performs spectral morphing between two phase vocoder streaming object.
- *PVMult* : Multiply magnitudes from two phase vocoder streaming object.
- *PVShift* : Spectral domain frequency shifter.
- *PVSynth* : Phase Vocoder synthesis object.
- *PVTranspose* : Transpose the frequency components of a pv stream.
- *PVVerb* : Spectral domain reverberation.
- *PadSynthTable* : Generates wavetable with the PadSynth algorithm from Nasca Octavian Paul.
- *Pan* : Cosinus panner with control on the spread factor.
- *ParaTable* : Generates parabola window function.
- *PartialTable* : Inharmonic waveform generator.
- *Particle2* : An even more full control granular synthesis generator.
- *Particle* : A full control granular synthesis generator.
- *Pattern* : Periodically calls a Python function.
- *PeakAmp* : Peak amplitude follower.
- *Percent* : Lets pass a certain percentage of the input triggers.
- *Phaser* : Multi-stages second-order phase shifter allpass filters.
- *Phasor* : A simple phase incrementor.
- *PinkNoise* : A pink noise generator.
- *Pointer2* : High quality table reader with control on the pointer position.
- *Pointer* : Table reader with control on the pointer position.
- *PolToCar* : Performs the polar to cartesian conversion.
- *Port* : Exponential portamento.
- *Pow* : Performs a power function on audio signal.

- *Print* : Print PyoObject's current value.
- *Programin* : Get the current value of a program change Midi controller.
- *Pulsar* : Pulsar synthesis oscillator.
- *PyoGuiControlSlider* : Floating-point control slider.
- *PyoGuiGrapher* : Multi-modes break-points function editor.
- *PyoGuiKeyboard* : Virtual MIDI keyboard.
- *PyoGuiMultiSlider* : Data multi-sliders editor.
- *PyoGuiScope* : Oscilloscope display.
- *PyoGuiSndView* : Soundfile display.
- *PyoGuiSpectrum* : Frequency spectrum display.
- *PyoGuiVuMeter* : Multi-channels Vu Meter.
- *RCOsc* : Waveform aproximation of a RC circuit.
- *RMS* : Returns the RMS (Root-Mean-Square) value of a signal.
- *RandDur* : Recursive time varying pseudo-random generator.
- *RandInt* : Periodic pseudo-random integer generator.
- *Randh* : Periodic pseudo-random generator.
- *Randi* : Periodic pseudo-random generator with interpolation.
- *RawMidi* : Raw Midi handler.
- *Record* : Writes input sound in an audio file on the disk.
- *Resample* : Realtime upsampling or downsampling of an audio signal.
- *Reson* : A second-order resonant bandpass filter.
- *Resonx* : A multi-stages second-order resonant bandpass filter.
- *Rossler* : Chaotic attractor for the Rossler system.
- *Round* : Rounds to the nearest integer value in a floating-point format.
- *SDelay* : Simple delay without interpolation.
- *SLMapDur* : SLMap with normalized values for a 'dur' slider.
- *SLMapFreq* : SLMap with normalized values for a 'freq' slider.
- *SLMapMul* : SLMap with normalized values for a 'mul' slider.
- *SLMapPan* : SLMap with normalized values for a 'pan' slider.
- *SLMapPhase* : SLMap with normalized values for a 'phase' slider.
- *SLMapQ* : SLMap with normalized values for a 'q' slider.
- *SPan* : Simple equal power panner.
- *STRev* : Stereo reverb.
- *SVF2* : Second-order state variable filter allowing continuous change of the filter type.
- *SVF* : Fourth-order state variable filter allowing continuous change of the filter type.
- *SampHold* : Performs a sample-and-hold operation on its input.

- *SawTable* : Sawtooth waveform generator.
- *Scale* : Maps an input range of audio values to an output range.
- *Scope* : Oscilloscope - audio waveform display.
- *Score* : Calls functions by incrementation of a preformatted name.
- *Select* : Sends trigger on matching integer values.
- *Selector* : Audio selector.
- *Seq* : Generates a rhythmic sequence of trigger signals.
- *SfMarkerLooper* : AIFF with markers soundfile looper.
- *SfMarkerShuffler* : AIFF with markers soundfile shuffler.
- *SfPlayer* : Soundfile player.
- *SharedTable* : Create an inter-process shared memory table.
- *SigTo* : Convert numeric value to PyoObject signal with portamento.
- *Sig* : Convert numeric value to PyoObject signal.
- *Sin* : Performs a sine function on audio signal.
- *SincTable* : Generates sinc window function.
- *SineLoop* : A simple sine wave oscillator with feedback.
- *Sine* : A simple sine wave oscillator.
- *SmoothDelay* : Artifact free sweepable recursive delay.
- *Snap* : Snap input values on a user's defined midi scale.
- *SndTable* : Transfers data from a soundfile into a function table.
- *Spectrum* : Spectrum analyzer and display.
- *Sqrt* : Performs a square-root function on audio signal.
- *SquareTable* : Square waveform generator.
- *Sub* : Substracts b from a.
- *SumOsc* : Discrete summation formulae to produce complex spectra.
- *SuperSaw* : Roland JP-8000 Supersaw emulator.
- *Switch* : Audio switcher.
- *TableFill* : Continuously fills a table with incoming samples.
- *TableIndex* : Table reader by sample position without interpolation.
- *TableMorph* : Morphs between multiple PyoTableObjects.
- *TablePut* : Writes values, without repetitions, from an audio stream into a DataTable.
- *TableRead* : Simple waveform table reader.
- *TableRec* : TableRec is for writing samples into a previously created NewTable.
- *TableScale* : Scales all the values contained in a PyoTableObject.
- *TableScan* : Reads the content of a table in loop, without interpolation.
- *TableWrite* : TableWrite writes samples into a previously created NewTable.

- *Tan* : Performs a tangent function on audio signal.
- *Tanh* : Performs a hyperbolic tangent function on audio signal.
- *Thresh* : Informs when a signal crosses a threshold.
- *Timer* : Reports elapsed time between two trigs.
- *Tone* : A first-order recursive low-pass filter with variable frequency response.
- *Touchin* : Get the current value of an after-touch Midi controller.
- *TrackHold* : Performs a track-and-hold operation on its input.
- *TranspoToCents* : Returns the cents value equivalent of a transposition factor.
- *TrigBurst* : Generates a time/amplitude expandable trigger pattern.
- *TrigChoice* : Random generator from user's defined values.
- *TrigEnv* : Envelope reader generator.
- *TrigExpseg* : Exponential segments trigger.
- *TrigFunc* : Python function callback.
- *TrigLinseg* : Line segments trigger.
- *TrigRandInt* : Pseudo-random integer generator.
- *TrigRand* : Pseudo-random number generator.
- *TrigTableRec* : TrigTableRec is for writing samples into a previously created NewTable.
- *TrigVal* : Outputs a previously defined value on a trigger signal.
- *TrigXnoiseMidi* : Triggered X-class midi notes pseudo-random generator.
- *TrigXnoise* : Triggered X-class pseudo-random generator.
- *Trig* : Sends one trigger.
- *Urn* : Periodic pseudo-random integer generator without duplicates.
- *VarPort* : Convert numeric value to PyoObject signal with portamento.
- *Vectral* : Performs magnitude smoothing between successive frames.
- *Vocoder* : Applies the spectral envelope of a first sound to the spectrum of a second sound.
- *VoiceManager* : Polyphony voice manager.
- *WGVerb* : 8 delay lines mono FDN reverb.
- *Waveguide* : Basic waveguide model.
- *WinTable* : Generates different kind of windowing functions.
- *Wrap* : Wraps-around the signal that exceeds the *min* and *max* thresholds.
- *XnoiseDur* : Recursive time varying X-class pseudo-random generator.
- *XnoiseMidi* : X-class midi notes pseudo-random generator.
- *Xnoise* : X-class pseudo-random generator.
- *Yin* : Pitch tracker using the Yin algorithm.
- *ZCross* : Zero-crossing counter.

1.8.4 Classes by category

Audio Server

Server

class Server(*sr=44100*, *nchnls=2*, *buffersize=256*, *duplex=1*, *audio='portaudio'*, *jackname='pyo'*, *ichnls=None*, *winhost='directsound'*, *midi='portmidi'*)

Main processing audio loop callback handler.

The Server object handles all communications with Portaudio and Portmidi. It keeps track of all audio streams created as well as connections between them.

An instance of the Server must be booted before defining any signal processing chain.

Args

sr: int, optional Sampling rate used by Portaudio and the Server to compute samples. Defaults to 44100.

nchnls: int, optional Number of output channels. The number of input channels will be the same if *ichnls* argument is not defined. Defaults to 2.

buffersize: int, optional Number of samples that Portaudio will request from the callback loop. Defaults to 256.

This value has an impact on CPU use (a small buffer size is harder to compute) and on the latency of the system.

Latency is *buffer size / sampling rate* in seconds.

duplex: int {0, 1}, optional Input - output mode. 0 is output only and 1 is both ways. Defaults to 1.

audio: string {'portaudio', 'pa', 'jack', 'coreaudio', 'offline', 'offline_nb', 'embedded'}, optional Audio backend to use. 'pa' is equivalent to 'portaudio'. Default is 'portaudio'.

'offline' save the audio output in a soundfile as fast as possible in blocking mode,

ie. the main program doesn't respond until the end of the computation.

'offline_nb' save the audio output in a soundfile as fast as possible in non-blocking mode,

ie. the computation is executed in a separated thread, allowing the program to respond while the computation goes on.

It is the responsibility of the user to make sure that the program doesn't exit before the computation is done.

'embedded' should be used when pyo is embedded inside an host environment via its C api.

If 'jack' is selected but jackd is not already started when the program is executed, pyo will ask jack to start in the background. Note that pyo never ask jack to close. It is the user's responsability to manage the audio configuration of its system.

jackname: string, optional Name of jack client. Defaults to 'pyo'

ichnls: int, optional Number of input channels if different of output channels. If None (default), ichnls = nchnls.

winhost: string, optional Under Windows, pyo's Server will try to use the default devices of the given host. This behaviour can be changed with the SetXXXDevice methods. Defaults to "directsound".

midi: string {'portmidi', 'pm', 'jack'}, optional Midi backend to use. 'pm' is equivalent to 'portmidi'. Default is 'portmidi'.

If 'jack' is selected but jackd is not already started when the program is executed, pyo will ask jack to start in the background. Note that pyo never ask jack to close. It is the user's responsability to manage the audio/midi configuration of its system.

Note: The following methods must be called **before** booting the server

- setInOutDevice(x): Set both input and output devices. See *pa_list_devices()*.
- setInputDevice(x): Set the audio input device number. See *pa_list_devices()*.
- setOutputDevice(x): Set the audio output device number. See *pa_list_devices()*.
- setInputOffset(x): Set the first physical input channel.
- setOutputOffset(x): Set the first physical output channel.
- setInOutOffset(x): Set the first physical input and output channels.
- setMidiInputDevice(x): Set the MIDI input device number. See *pm_list_devices()*.
- setMidiOutputDevice(x): Set the MIDI output device number. See *pm_list_devices()*.
- setSamplingRate(x): Set the sampling rate used by the server.
- setBufferSize(x): Set the buffer size used by the server.
- setNchnls(x): Set the number of output (and input if *ichnls* = None) channels used by the server.
- setIchnls(x): Set the number of input channels (if different of output channels) used by the server.
- setDuplex(x): Set the duplex mode used by the server.
- setVerbosity(x): Set the server's verbosity.
- reinit(sr, nchnls, buffersize, duplex, audio, jackname): Reinit the server's settings.
- deactivateMidi(): Deactivate Midi callback.
- setIsJackTransportSlave(x): Set if pyo's server is slave to jack transport or not.
- allowMicrosoftMidiDevices(): Allows the Microsoft Midi Mapper or GS Wavetable Synth devices.

```
>>> # For an 8 channels server in duplex mode with
>>> # a sampling rate of 48000 Hz and buffer size of 512
>>> s = Server(sr=48000, nchnls=8, buffersize=512, duplex=1).boot()
>>> s.start()
```

reinit(*sr=44100*, *nchnls=2*, *buffersize=256*, *duplex=1*, *audio='portaudio'*, *jackname='pyo'*, *ichnls=None*, *winhost='directsound'*, *midi='portmidi'*)
Reinit the server'settings. Useful to alternate between real-time and offline server.

Args Same as in the __init__ method.

setCallback(*callback*)
Register a custom process callback.

The function given as argument will be called every computation block, just before the computation of the audio object tree. Inside the callback, one can process the data of a table with numpy calls for example.

gui(*locals=None*, *meter=True*, *timer=True*, *exit=True*, *title=None*)
Show the server's user interface.

Args

locals: locals namespace {locals(), None}, optional If locals() is given, the interface will show an interpreter extension, giving a way to interact with the running script. Defaults to None.

meter: boolean, optinal If True, the interface will show a vumeter of the global output signal. Defaults to True.

timer: boolean, optional If True, the interface will show a clock of the current time. Defaults to True.

exit: boolean, optional If True, the python interpreter will exit when the 'Quit' button is pressed, Otherwise, the GUI will be closed leaving the interpreter alive. Defaults to True.

title: str, optional Alternate title for the server window. If None (default), generic title, "Pyo Server" is used.

closeGui()
Programmatically close the server's GUI.

setTimeCallable(*func*)
Set a function callback that will receive the current time as argument.

The function will receive four integers in this format: hours, minutes, seconds, milliseconds

Args

func: python callable Python function or method to call with current time as argument.

setMeterCallable(*func*)
Set a function callback that will receive the current rms values as argument.

The function will receive a list containing the rms value for each audio channel.

Args

func: python callable Python function or method to call with current rms values as argument.

setMeter(*meter*)
Registers a meter object to the server.

The object must have a method named *setRms*. This method will be called with the rms values of each audio channel as argument.

Args

meter: python object Python object with a *setRms* method.

setInOutDevice(*x*)
Set both input and output audio devices. See *pa_list_devices()*.

Args

x: int Number of the audio input and output devices.

setInputDevice(*x*)
Set the audio input device number. See *pa_list_devices()*.

Args

x: int Number of the audio device listed by Portaudio.

setOutputDevice(*x*)
Set the audio output device number. See *pa_list_devices()*.

Args

x: int Number of the audio device listed by Portaudio.

setInputOffset(*x*)
Set the first physical input channel.

Channel number *x* from the soundcard will be assigned to server's channel one, channel number *x* + 1 to server's channel two and so on.

Args

x: int Channel number.

setOutputOffset(*x*)
Set the first physical output channel.

Server's channel one will be assigned to soundcard's channel number *x*, server's channel two will be assigned to soundcard's channel number *x* + 1 and so on.

Args

x: int Channel number.

setInOutOffset(*x*)
Set the first physical input and output channels.

Set both offsets to the same value. See *setInputOffset* and *setOutputOffset* documentation for more details.

Args

x: int Channel number.

setMidiInputDevice(*x*)
Set the Midi input device number. See *pm_list_devices()*.

A number greater than the highest portmidi device index will opened all available input devices.

Args

x: int Number of the Midi device listed by Portmidi.

setMidiOutputDevice(*x*)
Set the Midi output device number. See *pm_list_devices()*.

Args

x: int Number of the Midi device listed by Portmidi.

allowMicrosoftMidiDevices()
Allows the Microsoft Midi Mapper or GS Wavetable Synth device.

These are off by default because they crash on some systems.

setSamplingRate(*x*)
Set the sampling rate used by the server.

Args

x: int New sampling rate, must be supported by the soundcard.

setBufferSize(*x*)
Set the buffer size used by the server.

Args

x: int New buffer size.

setNchnls (*x*)
Set the number of output (and input if *ichnls* = None) channels used by the server.

Args

x: int New number of channels.

setIchnls (*x*)
Set the number of input channels (if different of output channels) used by the server.

Args

x: int New number of input channels.

setDuplex (*x*)
Set the duplex mode used by the server.

Args

x: int {0 or 1} New mode. 0 is output only, 1 is both ways.

setVerbosity (*x*)
Set the server's verbosity.

Args

x: int

A sum of values to display different levels:

- 1 = error
- 2 = message
- 4 = warning
- 8 = debug

setGlobalDur (*x*)
Set the global object duration (time to wait before stopping the object).

This value, if not 0, will override the *dur* argument of object's play() and out() methods.

Args

x: float New global duration.

setGlobalDel (*x*)
Set the global object delay time (time to wait before activating the object).

This value, if not 0, will override the *del* argument of object's play() and out() methods.

Args

x: float New global delay time.

setJackAuto (*xin=True*, *xout=True*)
Tells the server to auto-connect (or not) Jack ports to System ports.

Args

xin: boolean Input Auto-connection switch. True is enabled (default) and False is disabled.

xout: boolean Output Auto-connection switch. True is enabled (default) and False is disabled.

setJackAutoConnectInputPorts(*ports*)
Tells the server to auto-connect Jack input ports to pre-defined Jack ports.

Args

ports: list of list of strings Name of the Jack ports to auto-connect to pyo input channels. There must be exactly one list of port(s) for each pyo input channel.

[['ports', 'to', 'channel', '1'], ['ports', 'to', 'channel', '2'], ...]

setJackAutoConnectOutputPorts(*ports*)
Tells the server to auto-connect Jack output ports to pre-defined Jack ports.

Args

ports: list of list of strings Name of the Jack ports to auto-connect to pyo output channels. There must be exactly one list of port(s) for each pyo output channel.

[['ports', 'to', 'channel', '1'], ['ports', 'to', 'channel', '2'], ...]

setJackInputPortNames(*name*)
Change the short name of pyo's input ports for the jack server.

This method must be called after the server is booted.

Args

name: string or list of strings New name of input ports for the jack server. If *name* is a string, '_xxx' (where xxx is the channel number) will be added to it for each input channel. If *name* is a list of strings, They will be used as is and there must be one for each input channel.

setJackOutputPortNames(*name*)
Change the short name of pyo's output ports for the jack server.

This method must be called after the server is booted.

Args

name: string or list of strings New name of output ports for the jack server. If *name* is a string, '_xxx' (where xxx is the channel number) will be added to it for each output channel. If *name* is a list of strings, They will be used as is and there must be one for each output channel.

setJackAutoConnectMidiInputPort(*ports*)
Tells the server to auto-connect Jack midi input port to pre-defined Jack ports.

Args

ports: string or list of strings Name of the Jack ports to auto-connect to pyo midi input channel.

['ports', 'to', 'midi', 'input', ...]

setJackAutoConnectMidiOutputPort(*ports*)
Tells the server to auto-connect Jack midi output port to pre-defined Jack ports.

Args

ports: string or list of strings Name of the Jack ports to auto-connect to pyo midi output channel.

['ports', 'to', 'midi', 'output', ...]

setJackMidiInputPortName(*name*)
Change the short name of pyo's midi input port for the jack server.

This method must be called after the server is booted.

Args

name: string New name of midi input port for the jack server.

setJackMidiOutputPortName(*name*)
Change the short name of pyo's midi output port for the jack server.

This method must be called after the server is booted.

Args

name: string New name of midi output port for the jack server.

setIsJackTransportSlave(*x*)
Set if pyo's server is slave to jack transport or not.

This method must be called before booting the server.

Args

x: boolean If True, the server's start and stop command will be slave to Jack transport. If False (the default) jack transport is ignored.

setGlobalSeed(*x*)
Set the server's global seed used by random objects.

Args

x: int A positive integer that will be used as the seed by random objects.

If zero, randoms will be seeded with the system clock current value.

setStartOffset(*x*)
Set the server's starting time offset. First *x* seconds will be rendered offline as fast as possible.

Args

x: float Starting time of the real-time processing.

setAmp(*x*)
Set the overall amplitude.

Args

x: float New amplitude.

beginResamplingBlock(*x*)
Starts a resampling block.

This factor must be a power-of-two. A positive value means upsampling and a negative value means downsampling. After this call, every PyoObject will be created with an internal sampling rate and buffer size relative to the resampling factor. The method *endResamplingBlock()* should be called at the end of the code block using the resampling factor.

The *Resample* object can be used inside the resampling block to perform up or down resampling of audio signal created before the block.

Args

x: int, power-of-two Resampling factor. Must be a power-of-two. A positive value starts an upsampling block while a negative value starts a downsampling block.

endResamplingBlock()
Ends a resampling block.

This call ends a code block using a sample rate different from the current sampling rate of the system.

The *Resample* object can be used after the resampling blick to perform up or down resampling of audio signal created inside the block.

shutdown()
Shut down and clear the server. This method will erase all objects from the callback loop. This method need to be called before changing server's parameters like *samplingrate*, *buffersize*, *nchnls*, ...

boot(*newBuffer=True*)
Boot the server. Must be called before defining any signal processing chain. Server's parameters like *samplingrate*, *buffersize* or *nchnls* will be effective after a call to this method.

Args

newBuffer: bool Specify if the buffers need to be allocated or not. Useful to limit the allocation of new buffers when the buffer size hasn't change.

Therefore, this is useful to limit calls to the Python interpreter to get the buffers addresses when using Pyo inside a C/C++ application with the embedded server.

Defaults to True.

start()
Start the audio callback loop and begin processing.

stop()
Stop the audio callback loop.

recordOptions(*dur=-1*, *filename=None*, *fileformat=0*, *sampletype=0*, *quality=0.4*)
Sets options for soundfile created by offline rendering or global recording.

Args

dur: float Duration, in seconds, of the recorded file. Only used by offline rendering. Must be positive. Defaults to -1.

filename: string Full path of the file to create. If None, a file called *pyo_rec.wav* will be created in the user's home directory. Defaults to None.

fileformat: int, optional Format type of the audio file. This function will first try to set the format from the filename extension.

If it's not possible, it uses the fileformat parameter. Supported formats are:

0. WAV - Microsoft WAV format (little endian) {.wav, .wave} (default)
1. AIFF - Apple/SGI AIFF format (big endian) {.aif, .aiff}
2. AU - Sun/NeXT AU format (big endian) {.au}
3. RAW - RAW PCM data {no extension}
4. SD2 - Sound Designer 2 {.sd2}
5. FLAC - FLAC lossless file format {.flac}
6. CAF - Core Audio File format {.caf}
7. OGG - Xiph OGG container {.ogg}

sampletype: int, optional Bit depth encoding of the audio file.

SD2 and FLAC only support 16 or 24 bit int. Supported types are:

0. 16 bits int (default)
1. 24 bits int
2. 32 bits int
3. 32 bits float
4. 64 bits float
5. U-Law encoded
6. A-Law encoded

quality: float, optional The encoding quality value, between 0.0 (lowest quality) and 1.0 (highest quality). This argument has an effect only with FLAC and OGG compressed formats. Defaults to 0.4.

recstart (*filename=None*)
Begins a default recording of the sound that is sent to the soundcard. This will create a file called *pyo_rec.wav* in the user's home directory if no path is supplied or defined with recordOptions method. Uses file format and sample type defined with recordOptions method.

Args

filename: string, optional Name of the file to be created. Defaults to None.

recstop()
Stop the previously started recording.

noteout (*pitch*, *velocity*, *channel=0*, *timestamp=0*)
Send a MIDI note message to the selected midi output device.

Arguments can be list of values to generate multiple events in one call.

Args

pitch: int Midi pitch, between 0 and 127.

velocity: int Amplitude of the note, between 0 and 127. A note with a velocity of 0 is equivalent to a note off.

channel: int, optional The Midi channel, between 1 and 16, on which the note is sent. A channel of 0 means all channels. Defaults to 0.

timestamp: int, optional The delay time, in milliseconds, before the message is sent to the output midi stream. A value of 0 means to play the note now. Defaults to 0.

afterout (*pitch*, *velocity*, *channel=0*, *timestamp=0*)
Send an aftertouch message to the selected midi output device.

Arguments can be list of values to generate multiple events in one call.

Args

pitch: int Midi key pressed down, between 0 and 127.

velocity: int Velocity of the pressure, between 0 and 127.

channel: int, optional The Midi channel, between 1 and 16, on which the note is sent. A channel of 0 means all channels. Defaults to 0.

timestamp: int, optional The delay time, in milliseconds, before the message is sent to the output midi stream. A value of 0 means to play the note now. Defaults to 0.

ctlout(*ctlnum*, *value*, *channel=0*, *timestamp=0*)
Send a control change message to the selected midi output device.

Arguments can be list of values to generate multiple events in one call.

Args

ctlnum: int Controller number, between 0 and 127.

value: int Value of the controller, between 0 and 127.

channel: int, optional The Midi channel, between 1 and 16, on which the message is sent. A channel of 0 means all channels. Defaults to 0.

timestamp: int, optional The delay time, in milliseconds, before the message is sent to the output midi stream. A value of 0 means to play the note now. Defaults to 0.

programout(*value*, *channel=0*, *timestamp=0*)
Send a program change message to the selected midi output device.

Arguments can be list of values to generate multiple events in one call.

Args

value: int New program number, between 0 and 127.

channel: int, optional The Midi channel, between 1 and 16, on which the message is sent. A channel of 0 means all channels. Defaults to 0.

timestamp: int, optional The delay time, in milliseconds, before the message is sent to the output midi stream. A value of 0 means to play the note now. Defaults to 0.

pressout(*value*, *channel=0*, *timestamp=0*)
Send a channel pressure message to the selected midi output device.

Arguments can be list of values to generate multiple events in one call.

Args

value: int Single greatest pressure value, between 0 and 127.

channel: int, optional The Midi channel, between 1 and 16, on which the message is sent. A channel of 0 means all channels. Defaults to 0.

timestamp: int, optional The delay time, in milliseconds, before the message is sent to the output midi stream. A value of 0 means to play the note now. Defaults to 0.

bendout(*value*, *channel=0*, *timestamp=0*)
Send a pitch bend message to the selected midi output device.

Arguments can be list of values to generate multiple events in one call.

Args

value: int 14 bits pitch bend value. 8192 is where there is no bending, 0 is full down and 16383 is full up bending.

channel: int, optional The Midi channel, between 1 and 16, on which the message is sent. A channel of 0 means all channels. Defaults to 0.

timestamp: int, optional The delay time, in milliseconds, before the message is sent to the output midi stream. A value of 0 means to play the note now. Defaults to 0.

sysexout(*msg*, *timestamp=0*)
Send a system exclusive message to the selected midi output device.

Arguments can be list of values/messages to generate multiple events in one call. Implemented only for portmidi, this method is not available when using jack as the midi backend.

Args

msg: str A valid system exclusive message as a string. The first byte must be 0xf0 and the last one must be 0xf7.

timestamp: int, optional The delay time, in milliseconds, before the message is sent to the output midi stream. A value of 0 means to play the message now. Defaults to 0.

makenote(*pitch*, *velocity*, *duration*, *channel=0*)

Send MIDI noteon/noteoff messages to the selected midi output device.

This method will send a noteon message to the selected midi output device and after a delay of *duration* milliseconds, it will send the corresponding noteoff message.

Arguments can be list of values to generate multiple events in one call.

Args

pitch: int Midi pitch, between 0 and 127.

velocity: int Amplitude of the note, between 0 and 127. A note with a velocity of 0 is equivalent to a note off.

duration: int The delay time, in milliseconds, before the noteoff message is sent to the output midi stream.

channel: int, optional The Midi channel, between 1 and 16, on which the note is sent. A channel of 0 means all channels. Defaults to 0.

addMidiEvent(*status*, *data1=0*, *data2=0*)

Add a MIDI event in the server processing loop.

This method can be used to programmatically simulate incoming MIDI events. In an embedded framework (ie. pyo inside puredata, openframeworks, etc.), this is useful to control a MIDI-driven script from the host program. Arguments can be list of values to generate multiple events in one call.

The MIDI event buffer is emptied at the end of each processing block. So, for events to be processed, addMidiEvent should be called at the beginning of the block. If you use audio objects to generate MIDI events, they should be created before the rest of the processing chain.

Args

status: int The status byte, indicating the type of event and the MIDI channel. Typical event type are:

128 -> 143: Noteoff 144 -> 159: Noteon 176 -> 191: Control change 192 -> 207: Program change 208 -> 223: After touch 224 -> 239: Pitch bend

data1: int, optional The first data byte (pitch for a midi note, controller number for a control change). Defaults to 0.

data2: int, optional The second data byte (velocity for a midi note, value for a control change). Defaults to 0.

getSamplingRate()

Return the current sampling rate.

getNchnls()

Return the current number of channels.

getBufferSize()

Return the current buffer size.

getGlobalDur()
Return the current global duration.

getGlobalDel()
Return the current global delay time.

getGlobalSeed()
Return the current global seed.

getIsStarted()
Returns 1 if the server is started, otherwise returns 0.

getIsBooted()
Returns 1 if the server is booted, otherwise returns 0.

deactivateMidi()
Deactivate Midi callback. Must be called before the boot() method.

getMidiActive()
Returns 1 if Midi callback is active, otherwise returns 0.

getStreams()
Returns the list of Stream objects currently in the Server memory.

getNumberOfStreams()
Returns the number of streams currently in the Server memory.

setServer()
Sets this server as the one to use for new objects when using the embedded device.

getInputAddr()
Return the address of the input buffer.

getOutputAddr()
Return the address of the output buffer.

getServerID()
Return the server ID.

getServerAddr()
Return the address of the server.

getEmbedICallbackAddr()
Return the address of the interleaved embedded callback function.

getCurrentTime()
Return the current time as a formatted string.

getCurrentAmp()
Return the current amplitudes as a tuple of *nchnls* length.

setAutoStartChildren(*state*)
Giving True to this method tells pyo that a call to the *play*, *out* or *stop* method of audio objects should propagate to the other audio objects given as arguments. This can be used to control an entire dsp chain just by calling methods on the very last object.

With setAutoStartChildren(True), a call to the *out* method will automatically triggers the *play* method of the Fader object given as *mul* argument.

```
>>> a = RCOsc(freq=150, mul=Fader(1, 1, mul=.3)).out()
```

With setAutoStartChildren(True), a call to the *stop* method will also propagate to audio objects given as arguments. The *stop* method has an argument *wait*, useful to postpone the moment when to really stop the

process. The *waiting value is also propagated to the other audio objects, but not for the objects assigned to a 'mul* argument. This property allows to do things like the next snippet. On a call *a.stop(1)* , while the Linseg assigned to the frequency waits for 1 second before actually stopping its process, the Fader assigned to the *mul* argument starts its fadeout immediately.

```
>>> freq = Linseg([(0,500),(1, 750),(2, 500)], loop=True)
>>> a = Sine(freq=freq, mul=Fader(1, 1, mul=.2)).out()
```

Sometime, we still want the process assigned to a *mul* attribute to wait before stopping its process. See the next case. The amplitude envelope is itself modulated in amplitude by another envelope. We still want the fadeout to start immediately, but its own envelope has to wait, otherwise, the amplitude will cut off before having the time to complete the fadeout. To do this, we call *useWaitTimeOnStop()* on the object we want to force to wait.

```
>>> lf = Linseg([(0,0),(0.2,0.1),(0.4,0)], loop=True)
>>> lf.useWaitTimeOnStop()
>>> freq = Linseg([(0,500),(1, 750),(2, 500)], loop=True)
>>> a = Sine(freq=freq, mul=Fader(1, 1, mul=lf)).out()
```

Note: The waiting time doesn't propagate to objects used in audio arithmetic. Code like this (for which you want to call out.stop()):

```
>>> a = Sine(500)
>>> a.setStopDelay(1)
>>> b = Sine(750)
>>> b.setStopDelay(2)
>>> out = Freeverb(a+b, size=0.8, mul=0.3).out()
>>> out.setStopDelay(4)
```

Should be written as:

```
>>> a = Sine(500)
>>> a.setStopDelay(1)
>>> b = Sine(750)
>>> b.setStopDelay(2)
>>> c = Mix([a, b], voices=1)
>>> c.setStopDelay(2)
>>> out = Freeverb(c, size=0.8, mul=0.3).out()
>>> out.setStopDelay(4)
```

---

amp
float. Overall amplitude.

startoffset
float. Starting time of the real-time processing.

verbosity
int. Server verbosity.

globalseed
int. Server global seed.

Controller listeners

These objects can be used to create MIDI and/or OSC listeners without the need to boot ands start an audio server before receiving messages.

MidiListener

class MidiListener(*function*, *mididev=-1*, *reportdevice=False*)

Self-contained midi listener thread.

This object allows to setup a Midi server that is independent of the audio server (mainly to be able to receive Midi data even when the audio server is stopped). Although it runs in a separated thread, the same device can't be used by this object and the audio server at the same time. It is adviced to call the deactivateMidi() method on the audio server to avoid conflicts.

Parent threading.Thread

Args

function: Python function (can't be a list) Function that will be called when a new midi event is available. This function is called with the incoming midi data as arguments. The signature of the function must be:

def myfunc(status, data1, data2)

mididev: int or list of ints, optional Sets the midi input device (see *pm_list_devices()* for the available devices). The default, -1, means the system default device. A number greater than the highest portmidi device index will open all available input devices. Specific devices can be set with a list of integers.

reportdevice: boolean, optional If True, the device ID will be reported as a fourth argument to the callback. The signature will then be:

def myfunc(status, data1, data2, id)

Available at initialization only. Defaults to False.

Note: This object is available only if pyo is built with portmidi support (see withPortmidi function).

```
>>> s = Server()
>>> s.deactivateMidi()
>>> s.boot()
>>> def midicall(status, data1, data2):
...     print(status, data1, data2)
>>> listen = MidiListener(midicall, 5)
>>> listen.start()
```

run()

Starts the process. The thread runs as daemon, so no need to stop it.

stop()

Stops the listener and properly close the midi ports.

getDeviceInfos()

Returns infos about connected midi devices.

This method returns a list of dictionaries, one per device.

Dictionary format is:

{"id": device_id (int), "name": device_name (str), "interface": interface (str)}

MidiDispatcher

class MidiDispatcher(*mididev=-1*)

Self-contained midi dispatcher thread.

This object allows to setup a Midi server that is independent of the audio server (mainly to be able to send Midi data even when the audio server is stopped). Although it runs in a separated thread, the same device can't be used by this object and the audio server at the same time. It is adviced to call the deactivateMidi() method on the audio server to avoid conflicts.

Use the *send* method to send midi event to connected devices.

Use the *sendx* method to send sysex event to connected devices.

Parent threading.Thread

Args

mididev: int or list of ints, optional Sets the midi output device (see *pm_list_devices()* for the available devices). The default, -1, means the system default device. A number greater than the highest portmidi device index will open all available input devices. Specific devices can be set with a list of integers.

Note: This object is available only if pyo is built with portmidi support (see withPortmidi function).

```
>>> s = Server()
>>> s.deactivateMidi()
>>> s.boot()
>>> dispatch = MidiDispatcher(5)
>>> dispatch.start()
>>> dispatch.send(144, 60, 127)
```

run()

Starts the process. The thread runs as daemon, so no need to stop it.

send(*status*, *data1*, *data2=0*, *timestamp=0*, *device=-1*)

Send a MIDI message to the selected midi output device.

Arguments can be list of values to generate multiple events in one call.

Args

status: int Status byte.

data1: int First data byte.

data2: int, optional Second data byte. Defaults to 0.

timestamp: int, optional The delay time, in milliseconds, before the note is sent on the portmidi stream. A value of 0 means to play the note now. Defaults to 0.

device: int, optional The index of the device to which the message will be sent. The default (-1) means all devices. See *getDeviceInfos()* to retrieve device indexes.

sendx(*msg*, *timestamp=0*, *device=-1*)

Send a MIDI system exclusive message to the selected midi output device.

Arguments can be list of values to generate multiple events in one call.

Args

msg: str A valid system exclusive message as a string. The first byte must be 0xf0 and the last one must be 0xf7.

timestamp: int, optional The delay time, in milliseconds, before the note is sent on the portmidi stream. A value of 0 means to play the note now. Defaults to 0.

device: int, optional The index of the device to which the message will be sent. The default (-1) means all devices. See *getDeviceInfos()* to retrieve device indexes.

getDeviceInfos()

Returns infos about connected midi devices.

This method returns a list of dictionaries, one per device.

Dictionary format is:

{"id": device_id (int), "name": device_name (str), "interface": interface (str)}

OscListener

class OscListener(*function*, *port=9000*)

Self-contained OSC listener thread.

This object allows to setup an OSC server that is independent of the audio server (mainly to be able to receive OSC data even when the audio server is stopped).

Parent threadind.Thread

Args

function: Python function (can't be a list) Function that will be called when a new OSC event is available. This function is called with the incoming address and values as arguments. The signature of the function must be:

def myfunc(address, *args)

port: int, optional The OSC port on which the values are received. Defaults to 9000.

```
>>> s = Server().boot()
>>> def call(address, *args):
...     print(address, args)
>>> listen = OscListener(call, 9901)
>>> listen.start()
```

run()

Starts the process. The thread runs as daemon, so no need to stop it.

Base Classes

Here are defined the base classes implementing common behaviors for the different kinds of objects in the library.

PyoObjectBase

class PyoObjectBase

Base class for all pyo objects.

This object encapsulates some common behaviors for all pyo objects.

One typically inherits from a more specific subclass of this class instead of using it directly.

Note: Operations allowed on all PyoObjectBase

```
>>> len(obj)       # Return the number of streams managed by the object.
>>> obj[x]         # Return stream `x` of the object.
>>>                # `x` is a number from 0 to len(obj)-1.
>>> dir(obj)       # Return the list of attributes of the object.
>>> for x in obj: # Can be used as an iterator (iterates over
>>>                # object's audio streams).
```

dump()
Print infos about the current state of the object.

Print the number of Stream objects managed by the instance and the current status of the object's attributes.

getBaseObjects()
Return a list of Stream objects managed by the instance.

getServer()
Return a reference to the current Server object.

getSamplingRate()
Return the current sampling rate (samples per second), as a float.

getBufferSize()
Return the current buffer size (samples per buffer), as an integer.

allowAutoStart(*switch=True*)
When autoStartChildren is activated in the Server, call this method with False as argument to stop the propagation of play/out/stop methods to this object. This is useful when an object is used at multiple places and you don't want to loose it when you stop one dsp block.

Note: This flag is ignored if you pass a _base object instead of a PyoObject. In the following code, a[0] will still be stopped by a b.stop(wait) call:

```
>>> a = Randi(min=0, max=0.3, freq=[1,2])
>>> a.allowAutoStart(False)
>>> b = Sine(mul=a[0]).out()
```

useWaitTimeOnStop()
When autoStartChildren is activated in the Server, call this method to force an object given to the *mul* attribute of another object to use the wait time from the stop method instead of being stopped immediately.

Note: Use wait time on stop is always "on" for _base objects. In the following code, a[0] will use the wait time given to b.stop(wait) even if it is used as a *mul* attribute:

```
>>> a = Randi(min=0, max=0.3, freq=[1,2])
>>> b = Sine(mul=a[0]).out()
```

addLinkedObject(*x*)
When autoStartChildren is activated in the Server, use this method to explicitly add an object in a dsp chain, which is generally controlled by the last object of the chain. Here is an example where we want an object to be linked to the chain but it can't be automatically detected:

```
>>> tab = NewTable(length=2, chnls=1)
>>> rec = TableRec(Sine(500), tab, .01)
>>> amp = TrigVal(rec["trig"], 0.5)
>>> osc = Osc(tab, tab.getRate(), mul=Fader(1, 1, mul=.2))
>>> # "osc" can't know for sure that "rec" should be linked
>>> # to this dsp chain, so we add it manually.
>>> osc.addLinkedObject(rec)
>>> osc.out()
```

setStopDelay(*x*)

Set a specific waiting time when calling the stop method on this object.

This method returns *self*, allowing it to be applied at the object creation.

Args

x: float New waiting time in seconds.

Note: if the method setStopDelay(x) was called before calling stop(wait) with a positive *wait* value, the *wait* value won't overwrite the value given to setStopDelay for the current object, but will be the one propagated to children objects. This allow to set a waiting time for a specific object with setStopDelay whithout changing the global delay time given to the stop method.

Stop delay value is ignored if you pass a _base object instead of a PyoObject. In the following code, a[0] ignores the a.setStopDelay(1) call:

```
>>> a = Randi(min=0, max=0.3, freq=[1,2])
>>> a.setStopDelay(1)
>>> b = Sine(mul=a[0]).out()
```

---

next()

Alias for __next__ method.

PyoObject

class PyoObject(*mul=1.0*, *add=0.0*)

Base class for all pyo objects that manipulate vectors of samples.

The user should never instantiate an object of this class.

Parent *PyoObjectBase*

Args

mul: float or PyoObject, optional Multiplication factor. Defaults to 1.

add: float or PyoObject, optional Addition factor. Defaults to 0.

Note: Arithmetics

Multiplication, addition, division and substraction can be applied between pyo objects or between pyo objects and numbers. Doing so returns a Dummy object with the result of the operation.

```
>>> # Creates a Dummy object `b` with `mul` set to 0.5.
>>> # Leaves `a` unchanged.
>>> b = a * 0.5
```

Inplace multiplication, addition, division and substraction can be applied between pyo objects or between pyo objects and numbers. These operations will replace the *mul* or *add* factor of the object.

```
>>> a *= 0.5 # replaces the `mul` attribute of `a`.
```

The next operators can only be used with PyoObject, not with XXX_base objects.

Exponent and **modulo**

```
>>> a ** 10 # returns a Pow object created as: Pow(a, 10)
>>> 10 ** a # returns a Pow object created as: Pow(10, a)
>>> a % 4 # returns a Wrap object created as: Wrap(a, 0, 4)
>>> a % b # returns a Wrap object created as: Wrap(a, 0, b)
```

Unary negative (**-**)

```
>>> -a # returns a Dummy object with negative values of streams in `a`.
```

Comparison operators

```
>>> # Comparison operators return a Compare object.
>>> x = a < b # same as: x = Compare(a, comp=b, mode="<")
>>> x = a <= b # same as: Compare(a, comp=b, mode="<=")
>>> x = a == b # same as: Compare(a, comp=b, mode="==")
>>> x = a != b # same as: Compare(a, comp=b, mode="!=")
>>> x = a > b # same as: Compare(a, comp=b, mode=">")
>>> x = a >= b # same as: Compare(a, comp=b, mode=">=")
```

A special case concerns the comparison of a PyoObject with None. All operators return False except *a != None*, which returns True.

---

isPlaying(*all=False*)

Returns True if the object is playing, otherwise, returns False.

Args

all: boolean, optional If True, the object returns a list with the state of all streams managed by the object.

If False, it return a boolean corresponding to the state of the first stream.

isOutputting(*all=False*)

Returns True if the object is outputting.

Returns True if the object is sending samples to dac, otherwise, returns False.

Args

all: boolean, optional If True, the object returns a list with the state of all streams managed by the object.

If False, it return a boolean corresponding to the state of the first stream.

get(*all=False*)

Return the first sample of the current buffer as a float.

Can be used to convert audio stream to usable Python data.

Object that implements string identifier for specific audio streams must use the corresponding string to specify the stream from which to get the value. See get() method definition in these object's man pages.

Args

all: boolean, optional If True, the first value of each object's stream will be returned as a list.

If False, only the value of the first object's stream will be returned as a float.

play(*dur=0*, *delay=0*)

Start processing without sending samples to output. This method is called automatically at the object creation.

This method returns *self*, allowing it to be applied at the object creation.

Args

dur: float, optional Duration, in seconds, of the object's activation. The default is 0 and means infinite duration.

delay: float, optional Delay, in seconds, before the object's activation. Defaults to 0.

out(*chnl=0*, *inc=1*, *dur=0*, *delay=0*)

Start processing and send samples to audio output beginning at *chnl*.

This method returns *self*, allowing it to be applied at the object creation.

Args

chnl: int, optional Physical output assigned to the first audio stream of the object. Defaults to 0.

inc: int, optional Output channel increment value. Defaults to 1.

dur: float, optional Duration, in seconds, of the object's activation. The default is 0 and means infinite duration.

delay: float, optional Delay, in seconds, before the object's activation. Defaults to 0.

If *chnl* >= 0, successive streams increment the output number by *inc* and wrap around the global number of channels.

If *chnl* is negative, streams begin at 0, increment the output number by *inc* and wrap around the global number of channels. Then, the list of streams is scrambled.

If *chnl* is a list, successive values in the list will be assigned to successive streams.

stop(*wait=0*)

Stop processing.

This method returns *self*, allowing it to be applied at the object creation.

Args

wait: float, optional Delay, in seconds, before the process is actually stopped. If autoStartChildren is activated in the Server, this value is propagated to the children objects. Defaults to 0.

Note: if the method setStopDelay(x) was called before calling stop(wait) with a positive *wait* value, the *wait* value won't overwrite the value given to setStopDelay for the current object, but will be the one propagated to children objects. This allow to set a waiting time for a specific object with setStopDelay whithout changing the global delay time given to the stop method.

Fader and Adsr objects ignore waiting time given to the stop method because they already implement a delayed processing triggered by the stop call.

---

mix (*voices=1*)
Mix the object's audio streams into *voices* streams and return a Mix object.

Args

voices: int, optional Number of audio streams of the Mix object created by this method. Defaults to 1.

If more than 1, object's streams are alternated and added into Mix object's streams.

range (*min*, *max*)
Adjust *mul* and *add* attributes according to a given range.

This function assumes a signal between -1 and 1. Arguments may be list of floats for multi-streams objects.

This method returns *self*, allowing it to be applied at the object creation:

```
>>> lfo = Sine(freq=1).range(500, 1000)
```

Args

min: float Minimum value of the output signal.

max: float Maximum value of the output signal.

setMul (*x*)
Replace the *mul* attribute.

Args

x: float or PyoObject New *mul* attribute.

setAdd (*x*)
Replace the *add* attribute.

Args

x: float or PyoObject New *add* attribute.

setSub (*x*)
Replace and inverse the *add* attribute.

Args

x: float or PyoObject New inversed *add* attribute.

setDiv (*x*)
Replace and inverse the *mul* attribute.

Args

x: float or PyoObject New inversed *mul* attribute.

set (*attr*, *value*, *port=0.025*, *callback=None*)
Replace any attribute with portamento.

This method is intended to be applied on attributes that are not already assigned to PyoObjects. It will work only with floats or list of floats.

Args

attr: string Name of the attribute as a string.

value: float New value.

port: float, optional Time, in seconds, to reach the new value.

callback: callable, optional A python function to be called at the end of the ramp. If the end of the ramp is not reached (ex.: called again before the end of the portamento), the callback will not be called.

ctrl(*map_list=None*, *title=None*, *wxnoserver=False*)
Opens a sliders window to control the parameters of the object. Only parameters that can be set to a PyoObject are allowed to be mapped on a slider.

If a list of values are given to a parameter, a multisliders will be used to control each stream independently.

Args

map_list: list of SLMap objects, optional Users defined set of parameters scaling. There is default scaling for each object that accept *ctrl* method.

title: string, optional Title of the window. If none is provided, the name of the class is used.

wxnoserver: boolean, optional With wxPython graphical toolkit, if True, tells the interpreter that there will be no server window.

If *wxnoserver* is set to True, the interpreter will not wait for the server GUI before showing the controller window.

mul
float or PyoObject. Multiplication factor.

add
float or PyoObject. Addition factor.

PyoTableObject

class PyoTableObject(*size=0*)
Base class for all pyo table objects.

A table object is a buffer memory to store precomputed samples.

The user should never instantiate an object of this class.

Parent *PyoObjectBase*

Args

size: int Length of the table in samples. Usually provided by the child object.

save(*path*, *format=0*, *sampletype=0*, *quality=0.4*)
Writes the content of the table in an audio file.

The sampling rate of the file is the sampling rate of the server and the number of channels is the number of table streams of the object.

Args

path: string Full path (including extension) of the new file.

format: int, optional

Format type of the new file. Supported formats are:

0. WAVE - Microsoft WAV format (little endian) {.wav, .wave}
1. AIFF - Apple/SGI AIFF format (big endian) {.aif, .aiff}

2. AU - Sun/NeXT AU format (big endian) {.au}
3. RAW - RAW PCM data {no extension}
4. SD2 - Sound Designer 2 {.sd2}
5. FLAC - FLAC lossless file format {.flac}
6. CAF - Core Audio File format {.caf}
7. OGG - Xiph OGG container {.ogg}

sampletype: int, optional Bit depth encoding of the audio file.

SD2 and FLAC only support 16 or 24 bit int. Supported types are:

0. 16 bit int (default)
1. 24 bit int
2. 32 bit int
3. 32 bit float
4. 64 bit float
5. U-Law encoded
6. A-Law encoded

quality: float, optional The encoding quality value, between 0.0 (lowest quality) and 1.0 (highest quality). This argument has an effect only with FLAC and OGG compressed formats. Defaults to 0.4.

write(*path*, *oneline=True*)

Writes the content of the table in a text file.

This function can be used to store the table data as a list of floats into a text file.

Args

path: string Full path of the generated file.

oneline: boolean, optional If True, list of samples will inserted on one line.

If False, list of samples will be truncated to 8 floats per line.

read(*path*)

Reads the content of a text file and replaces the table data with the values stored in the file.

Args

path: string Full path of the file to read.

The format is a list of lists of floats. For example, A two tablestreams object must be given a content like this:

[[0.0, 1.0, 0.5, . . .], [1.0, 0.99, 0.98, 0.97, . . .]]

Each object's tablestream will be resized according to the length of the lists.

getBuffer(*chnl=0*)

Return a reference to the underlying object implementing the buffer protocol.

With the buffer protocol, a table can be referenced and modified, without copying the data, with numpy functions. To create an array using the same memory as the table:

t = SndTable(SNDS_PATH+"/transparent.aif") arr = numpy.asarray(t.getBuffer())

Now, every changes applied to the array will be reflected in the SndTable.

For more details about the buffer protocol, see PEP 3118 and python documentation.

setSize(*size*)
Change the size of the table.

This will erase the previously drawn waveform.

Args

size: int New table size in samples.

getSize(*all=False*)
Return table size in samples.

Args

all: boolean If the table contains more than one stream and *all* is True, returns a list of all sizes. Otherwise, returns only the first size as an int. Defaults to False.

put(*value*, *pos=0*)
Puts a value at specified sample position in the table.

If the object has more than 1 tablestream, the default is to record the value in each table. User can call obj[x].put() to record into a specific table.

Args

value: float Value, as floating-point, to record in the table.

pos: int, optional Position, in samples, where to record value. Defaults to 0.

get(*pos*)
Returns the value, as float, stored at a specific position in the table.

If the object has more than 1 tablestream, the default is to return a list with the value of each tablestream. User can call obj[x].get() to get the value of a specific table.

Args

pos: int, optional Position, in samples, where to read the value. Defaults to 0.

getTable(*all=False*)
Returns the content of the table as list of floats.

Args

all: boolean, optional If True, all sub tables are retrieved and returned as a list of list of floats.

If False, a single list containing the content of the first subtable (or the only one) is returned.

normalize()
Normalize table samples between -1 and 1.

reset()
Resets table samples to 0.0.

removeDC()
Filter out DC offset from the table's data.

reverse()
Reverse the table's data in time.

invert()
Reverse the table's data in amplitude.

rectify()
Positive rectification of the table's data.

pow(*exp=10*)
Apply a power function on each sample in the table.

Args

exp: float, optional Exponent factor. Defaults to 10.

bipolarGain(*gpos=1*, *gneg=1*)
Apply different gain factor for positive and negative samples.

Args

gpos: float, optional Gain factor for positive samples. Defaults to 1.

gneg: float, optional Gain factor for negative samples. Defaults to 1.

lowpass(*freq=1000*)
Apply a one-pole lowpass filter on table's samples.

Args

freq: float, optional Filter's cutoff, in Hertz. Defaults to 1000.

fadein(*dur=0.1*)
Apply a gradual increase in the level of the table's samples.

Args

dur: float, optional Fade in duration, in seconds. Defaults to 0.1.

fadeout(*dur=0.1*)
Apply a gradual decrease in the level of the table's samples.

Args

dur: float, optional Fade out duration, in seconds. Defaults to 0.1.

add(*x*)
Performs addition on the table values.

Adds the argument to each table values, position by position if the argument is a list or another PyoTableObject.

Args

x: float, list or PyoTableObject value(s) to add.

sub(*x*)
Performs substraction on the table values.

Substracts the argument to each table values, position by position if the argument is a list or another PyoTableObject.

Args

x: float, list or PyoTableObject value(s) to substract.

mul(*x*)
Performs multiplication on the table values.

Multiply each table values by the argument, position by position if the argument is a list or another PyoTableObject.

Args

x: float, list or PyoTableObject value(s) to multiply.

copyData(*table*, *srcpos=0*, *destpos=0*, *length=-1*)
Copy samples from a source table.

Copy *length* samples from a source table to this table.

Args

table: PyoTableObject The source table.

srcpos: int, optional The start position, in samples, in the source table. Defaults to 0.

destpos ; int, optional The start position, in samples, in the destination (self) table. Defaults to 0.

length: int, optional The number of samples to copy from source to destination. if length is negative, the length of the smallest table is used. Defaults to -1.

rotate(*pos*)
Rotate the table content to the left around the position given as argument.

Samples between the given position and the end of the table will be relocated in front of the samples from the beginning to the given position.

Args

pos: int The rotation position in samples. A negative position means a rotation to the right.

copy()
Returns a deep copy of the object.

view(*title='Table waveform'*, *wxnoserver=False*)
Opens a window showing the contents of the table.

Args

title: string, optional Window title. Defaults to "Table waveform".

wxnoserver: boolean, optional With wxPython graphical toolkit, if True, tells the interpreter that there will be no server window.

If *wxnoserver* is set to True, the interpreter will not wait for the server GUI before showing the controller window.

refreshView()
Updates the graphical display of the table, if applicable.

size
int. Table size in samples.

PyoMatrixObject

class PyoMatrixObject
Base class for all pyo matrix objects.

A matrix object is a 2 dimensions buffer memory to store precomputed samples.

The user should never instantiate an object of this class.

Parent *PyoObjectBase*

write(*path*)
Writes the content of the matrix into a text file.

This function can be used to store the matrix data as a list of list of floats into a text file.

Args

path: string Full path of the generated file.

read(*path*)
Reads the content of a text file and replaces the matrix data with the values in the file.

Format is a list of lists of floats. For example, A two matrixstreams object must be given a content like this:

[[[0.0 ,1.0, 0.5, ...], [1.0, 0.99, 0.98, 0.97, ...]], [[0.0, 1.0, 0.5, ...], [1.0, 0.99, 0.98, 0.97, ...]]]

Each object's matrixstream will be resized according to the length of the lists, but the number of matrixstreams must be the same.

Args

path: string Full path of the file to read.

getSize()
Returns matrix size in samples. Size is a tuple (x, y).

normalize()
Normalize matrix samples between -1 and 1.

blur()
Apply a simple gaussian blur on the matrix.

boost(*min=-1.0*, *max=1.0*, *boost=0.01*)
Boost the constrast of values in the matrix.

Args

min: float, optional Minimum value. Defaults to -1.0.

max: float, optional Maximum value. Defaults to 1.0.

boost: float, optional Amount of boost applied on each value. Defaults to 0.01.

put(*value*, *x=0*, *y=0*)
Puts a value at specified position in the matrix.

If the object has more than 1 matrixstream, the default is to record the value in each matrix. User can call obj[x].put() to record in a specific matrix.

Args

value: float Value, as floating-point, to record in the matrix.

x: int, optional X position where to record value. Defaults to 0.

y: int, optional Y position where to record value. Defaults to 0.

get(*x=0*, *y=0*)
Returns the value, as float, at specified position in the matrix.

If the object has more than 1 matrixstream, the default is to return a list with the value of each matrixstream. User can call obj[x].get() to get the value of a specific matrix.

Args

x: int, optional X position where to get value. Defaults to 0.

y: int, optional Y position where to get value. Defaults to 0.

getInterpolated(*x=0.0*, *y=0.0*)
Returns the value, as float, at a normalized position in the matrix.

If the object has more than 1 matrixstream, the default is to return a list with the value of each matrixstream. User can call obj[x].getInterpolated() to get the value of a specific matrix.

Args

x: float {0 -> 1} X normalized position where to get value. Defaults to 0.0.

y: int {0 -> 1} Y normalized position where to get value. Defaults to 0.0.

view(*title='Matrix viewer'*, *wxnoserver=False*)
Opens a window showing the contents of the matrix.

Args

title: string, optional Window title. Defaults to "Matrix viewer".

wxnoserver: boolean, optional With wxPython graphical toolkit, if True, tells the interpreter that there will be no server window.

If *wxnoserver* is set to True, the interpreter will not wait for the server GUI before showing the controller window.

refreshView()
Updates the graphical display of the matrix, if applicable.

PyoPVObject

class PyoPVObject
Base class for objects working with phase vocoder's magnitude and frequency streams.

The user should never instantiate an object of this class.

Parent `PyoObjectBase`

isPlaying(*all=False*)
Returns True if the object is playing, otherwise, returns False.

Args

all: boolean, optional If True, the object returns a list with the state of all streams managed by the object.

If False, it return a boolean corresponding to the state of the first stream.

play(*dur=0*, *delay=0*)
Start processing without sending samples to output. This method is called automatically at the object creation.

This method returns *self*, allowing it to be applied at the object creation.

Args

dur: float, optional Duration, in seconds, of the object's activation. The default is 0 and means infinite duration.

delay: float, optional Delay, in seconds, before the object's activation. Defaults to 0.

stop(*wait=0*)
Stop processing.

This method returns *self*, allowing it to be applied at the object creation.

Args

wait: float, optional Delay, in seconds, before the process is actually stopped. If autoStartChildren is activated in the Server, this value is propagated to the children objects. Defaults to 0.

Note: if the method setStopDelay(x) was called before calling stop(wait) with a positive *wait* value, the *wait* value won't overwrite the value given to setStopDelay for the current object, but will be the one propagated to children objects. This allow to set a waiting time for a specific object with setStopDelay whithout changing the global delay time given to the stop method.

---

set(*attr*, *value*, *port=0.025*)
Replace any attribute with portamento.

This method is intended to be applied on attributes that are not already assigned to PyoObjects. It will work only with floats or list of floats.

Args

attr: string Name of the attribute as a string.

value: float New value.

port: float, optional Time, in seconds, to reach the new value.

ctrl(*map_list=None*, *title=None*, *wxnoserver=False*)
Opens a sliders window to control the parameters of the object. Only parameters that can be set to a PyoObject are allowed to be mapped on a slider.

If a list of values are given to a parameter, a multisliders will be used to control each stream independently.

Args

map_list: list of SLMap objects, optional Users defined set of parameters scaling. There is default scaling for each object that accept *ctrl* method.

title: string, optional Title of the window. If none is provided, the name of the class is used.

wxnoserver: boolean, optional With wxPython graphical toolkit, if True, tells the interpreter that there will be no server window.

If *wxnoserver* is set to True, the interpreter will not wait for the server GUI before showing the controller window.

Audio Signal Analysis

Tools to analyze audio signals.

These objects are designed to retrieve specific informations from an audio stream. Analysis are sent at audio rate, user can use them for controlling parameters of others objects.

Follower

class Follower(*input*, *freq=20*, *mul=1*, *add=0*)

Envelope follower.

Output signal is the continuous mean amplitude of an input signal.

Parent *PyoObject*

Args

input: PyoObject Input signal to process.

freq: float or PyoObject, optional Cutoff frequency of the filter in hertz. Default to 20.

Note: The out() method is bypassed. Follower's signal can not be sent to audio outs.

See also:

Follower2, :py:class: *Balance*

```
>>> s = Server().boot()
>>> s.start()
>>> sf = SfPlayer(SNDS_PATH + "/transparent.aif", loop=True, mul=.4).out()
>>> fol = Follower(sf, freq=30)
>>> n = Noise(mul=fol).out(1)
```

setInput(*x*, *fadetime=0.05*)

Replace the *input* attribute.

Args

x: PyoObject New signal to process.

fadetime: float, optional Crossfade time between old and new input. Default to 0.05.

setFreq(*x*)

Replace the *freq* attribute.

Args

x: float or PyoObject New *freq* attribute.

out(*chnl=0*, *inc=1*, *dur=0*, *delay=0*)

Start processing and send samples to audio output beginning at *chnl*.

This method returns *self*, allowing it to be applied at the object creation.

Args

chnl: int, optional Physical output assigned to the first audio stream of the object. Defaults to 0.

inc: int, optional Output channel increment value. Defaults to 1.

dur: float, optional Duration, in seconds, of the object's activation. The default is 0 and means infinite duration.

delay: float, optional Delay, in seconds, before the object's activation. Defaults to 0.

If *chnl* >= 0, successive streams increment the output number by *inc* and wrap around the global number of channels.

If *chnl* is negative, streams begin at 0, increment the output number by *inc* and wrap around the global number of channels. Then, the list of streams is scrambled.

If *chnl* is a list, successive values in the list will be assigned to successive streams.

ctrl (*map_list=None*, *title=None*, *wxnoserver=False*)
Opens a sliders window to control the parameters of the object. Only parameters that can be set to a PyoObject are allowed to be mapped on a slider.

If a list of values are given to a parameter, a multisliders will be used to control each stream independently.

Args

map_list: list of SLMap objects, optional Users defined set of parameters scaling. There is default scaling for each object that accept *ctrl* method.

title: string, optional Title of the window. If none is provided, the name of the class is used.

wxnoserver: boolean, optional With wxPython graphical toolkit, if True, tells the interpreter that there will be no server window.

If *wxnoserver* is set to True, the interpreter will not wait for the server GUI before showing the controller window.

input
PyoObject. Input signal to process.

freq
float or PyoObject. Cutoff frequency of the filter.

Follower2

class Follower2 (*input*, *risetime=0.01*, *falltime=0.1*, *mul=1*, *add=0*)
Envelope follower with different attack and release times.

Output signal is the continuous mean amplitude of an input signal.

Parent *PyoObject*

Args

input: PyoObject Input signal to process.

risetime: float or PyoObject, optional Time to reach upward value in seconds. Default to 0.01.

falltime: float or PyoObject, optional Time to reach downward value in seconds. Default to 0.1.

Note: The out() method is bypassed. Follower's signal can not be sent to audio outs.

See also:

Follower, :py:class: *Balance*

```
>>> s = Server().boot()
>>> s.start()
>>> sf = SfPlayer(SNDS_PATH + "/transparent.aif", loop=True, mul=.4).out()
>>> fol2 = Follower2(sf, risetime=0.002, falltime=.1, mul=.5)
>>> n = Noise(fol2).out(1)
```

setInput(*x*, *fadetime=0.05*)
Replace the *input* attribute.

Args

x: PyoObject New signal to process.

fadetime: float, optional Crossfade time between old and new input. Default to 0.05.

setRisetime(*x*)
Replace the *risetime* attribute.

Args

x: float or PyoObject New *risetime* attribute.

setFalltime(*x*)
Replace the *falltime* attribute.

Args

x: float or PyoObject New *falltime* attribute.

out(*chnl=0*, *inc=1*, *dur=0*, *delay=0*)
Start processing and send samples to audio output beginning at *chnl*.

This method returns *self*, allowing it to be applied at the object creation.

Args

chnl: int, optional Physical output assigned to the first audio stream of the object. Defaults to 0.

inc: int, optional Output channel increment value. Defaults to 1.

dur: float, optional Duration, in seconds, of the object's activation. The default is 0 and means infinite duration.

delay: float, optional Delay, in seconds, before the object's activation. Defaults to 0.

If *chnl* >= 0, successive streams increment the output number by *inc* and wrap around the global number of channels.

If *chnl* is negative, streams begin at 0, increment the output number by *inc* and wrap around the global number of channels. Then, the list of streams is scrambled.

If *chnl* is a list, successive values in the list will be assigned to successive streams.

ctrl(*map_list=None*, *title=None*, *wxnoserver=False*)
Opens a sliders window to control the parameters of the object. Only parameters that can be set to a PyoObject are allowed to be mapped on a slider.

If a list of values are given to a parameter, a multisliders will be used to control each stream independently.

Args

map_list: list of SLMap objects, optional Users defined set of parameters scaling. There is default scaling for each object that accept *ctrl* method.

title: string, optional Title of the window. If none is provided, the name of the class is used.

wxnoserver: boolean, optional With wxPython graphical toolkit, if True, tells the interpreter that there will be no server window.

If *wxnoserver* is set to True, the interpreter will not wait for the server GUI before showing the controller window.

input
PyoObject. Input signal to process.

risetime
float or PyoObject. Time to reach upward value in seconds.

falltime
float or PyoObject. Time to reach downward value in seconds.

ZCross

class ZCross(*input*, *thresh=0.0*, *mul=1*, *add=0*)
Zero-crossing counter.

Output signal is the number of zero-crossing occured during each buffer size, normalized between 0 and 1.

Parent *PyoObject*

Args

input: PyoObject Input signal to process.

thresh: float, optional Minimum amplitude difference allowed between adjacent samples to be included in the zeros count.

Note: The out() method is bypassed. ZCross's signal can not be sent to audio outs.

```
>>> s = Server().boot()
>>> s.start()
>>> a = SfPlayer(SNDS_PATH + "/transparent.aif", loop=True, mul=.4).out()
>>> b = ZCross(a, thresh=.02)
>>> n = Noise(b).out(1)
```

setInput(*x*, *fadetime=0.05*)
Replace the *input* attribute.

Args

x: PyoObject New signal to process.

fadetime: float, optional Crossfade time between old and new input. Default to 0.05.

setThresh(*x*)
Replace the *thresh* attribute.

Args

x: float New amplitude difference threshold.

out(*chnl=0*, *inc=1*, *dur=0*, *delay=0*)
Start processing and send samples to audio output beginning at *chnl*.

This method returns *self*, allowing it to be applied at the object creation.

Args

chnl: int, optional Physical output assigned to the first audio stream of the object. Defaults to 0.

inc: int, optional Output channel increment value. Defaults to 1.

dur: float, optional Duration, in seconds, of the object's activation. The default is 0 and means infinite duration.

delay: float, optional Delay, in seconds, before the object's activation. Defaults to 0.

If *chnl* >= 0, successive streams increment the output number by *inc* and wrap around the global number of channels.

If *chnl* is negative, streams begin at 0, increment the output number by *inc* and wrap around the global number of channels. Then, the list of streams is scrambled.

If *chnl* is a list, successive values in the list will be assigned to successive streams.

ctrl(*map_list=None*, *title=None*, *wxnoserver=False*)
Opens a sliders window to control the parameters of the object. Only parameters that can be set to a PyoObject are allowed to be mapped on a slider.

If a list of values are given to a parameter, a multisliders will be used to control each stream independently.

Args

map_list: list of SLMap objects, optional Users defined set of parameters scaling. There is default scaling for each object that accept *ctrl* method.

title: string, optional Title of the window. If none is provided, the name of the class is used.

wxnoserver: boolean, optional With wxPython graphical toolkit, if True, tells the interpreter that there will be no server window.

If *wxnoserver* is set to True, the interpreter will not wait for the server GUI before showing the controller window.

input
PyoObject. Input signal to process.

thresh
float. Amplitude difference threshold.

Yin

class Yin(*input*, *tolerance=0.2*, *minfreq=40*, *maxfreq=1000*, *cutoff=1000*, *winsize=1024*, *mul=1*, *add=0*)
Pitch tracker using the Yin algorithm.

Pitch tracker using the Yin algorithm based on the implementation in C of aubio. This algorithm was developped by A. de Cheveigne and H. Kawahara and published in

de Cheveigne, A., Kawahara, H. (2002) 'YIN, a fundamental frequency estimator for speech and music', J. Acoust. Soc. Am. 111, 1917-1930.

The audio output of the object is the estimated frequency, in Hz, of the input sound.

Parent *PyoObject*

Args

input: PyoObject Input signal to process.

tolerance: float, optional Parameter for minima selection, between 0 and 1. Defaults to 0.2.

minfreq: float, optional Minimum estimated frequency in Hz. Frequency below this threshold will be ignored. Defaults to 40.

maxfreq: float, optional Maximum estimated frequency in Hz. Frequency above this threshold will be ignored. Defaults to 1000.

cutoff: float, optional Cutoff frequency, in Hz, of the lowpass filter applied on the input sound. Defaults to 1000.

The lowpass filter helps the algorithm to detect the fundamental frequency by filtering higher harmonics.

winsize: int, optional Size, in samples, of the analysis window. Must be higher that two period of the lowest desired frequency.

Available at initialization time only. Defaults to 1024.

```
>>> s = Server(duplex=1).boot()
>>> s.start()
>>> lfo = Randh(min=100, max=500, freq=3)
>>> src = SineLoop(freq=lfo, feedback=0.1, mul=.3).out()
>>> pit = Yin(src, tolerance=0.2, winsize=1024)
>>> freq = Tone(pit, freq=10)
>>> # fifth above
>>> a = LFO(freq*1.5, type=2, mul=0.2).out(1)
```

setInput (*x*, *fadetime=0.05*)

Replace the *input* attribute.

Args

x: PyoObject New signal to process.

fadetime: float, optional Crossfade time between old and new input. Default to 0.05.

setTolerance (*x*)

Replace the *tolerance* attribute.

Args

x: float New parameter for minima selection, between 0 and 1.

setMinfreq (*x*)

Replace the *minfreq* attribute.

Args

x: float New minimum frequency detected.

setMaxfreq (*x*)

Replace the *maxfreq* attribute.

Args

x: float New maximum frequency detected.

setCutoff (*x*)

Replace the *cutoff* attribute.

Args

x: float New input lowpass filter cutoff frequency.

out (*chnl=0*, *inc=1*, *dur=0*, *delay=0*)

Start processing and send samples to audio output beginning at *chnl*.

This method returns *self*, allowing it to be applied at the object creation.

Args

chnl: int, optional Physical output assigned to the first audio stream of the object. Defaults to 0.

inc: int, optional Output channel increment value. Defaults to 1.

dur: float, optional Duration, in seconds, of the object's activation. The default is 0 and means infinite duration.

delay: float, optional Delay, in seconds, before the object's activation. Defaults to 0.

If *chnl* >= 0, successive streams increment the output number by *inc* and wrap around the global number of channels.

If *chnl* is negative, streams begin at 0, increment the output number by *inc* and wrap around the global number of channels. Then, the list of streams is scrambled.

If *chnl* is a list, successive values in the list will be assigned to successive streams.

ctrl(*map_list=None*, *title=None*, *wxnoserver=False*)
Opens a sliders window to control the parameters of the object. Only parameters that can be set to a PyoObject are allowed to be mapped on a slider.

If a list of values are given to a parameter, a multisliders will be used to control each stream independently.

Args

map_list: list of SLMap objects, optional Users defined set of parameters scaling. There is default scaling for each object that accept *ctrl* method.

title: string, optional Title of the window. If none is provided, the name of the class is used.

wxnoserver: boolean, optional With wxPython graphical toolkit, if True, tells the interpreter that there will be no server window.

If *wxnoserver* is set to True, the interpreter will not wait for the server GUI before showing the controller window.

input
PyoObject. Input signal to process.

tolerance
float. Parameter for minima selection.

minfreq
float. Minimum frequency detected.

maxfreq
float. Maximum frequency detected.

cutoff
float. Input lowpass filter cutoff frequency.

Centroid

class Centroid(*input*, *size=1024*, *mul=1*, *add=0*)
Computes the spectral centroid of an input signal.

Output signal is the spectral centroid, in Hz, of the input signal. It indicates where the "center of mass" of the spectrum is. Perceptually, it has a robust connection with the impression of "brightness" of a sound.

Centroid does its computation with two overlaps, so a new output value comes every half of the FFT window size.

Parent *PyoObject*

Args

input: PyoObject Input signal to process.

size: int, optional Size, as a power-of-two, of the FFT used to compute the centroid.

Available at initialization time only. Defaults to 1024.

Note: The out() method is bypassed. Centroid's signal can not be sent to audio outs.

```
>>> s = Server().boot()
>>> s.start()
>>> a = SfPlayer(SNDS_PATH + "/transparent.aif", loop=True, mul=.4).out()
>>> b = Centroid(a, 1024)
>>> c = Port(b, 0.05, 0.05)
>>> d = ButBP(Noise(0.2), freq=c, q=5).out(1)
```

setInput (*x*, *fadetime=0.05*)

Replace the *input* attribute.

Args

x: PyoObject New signal to process.

fadetime: float, optional Crossfade time between old and new input. Default to 0.05.

out (*chnl=0*, *inc=1*, *dur=0*, *delay=0*)

Start processing and send samples to audio output beginning at *chnl*.

This method returns *self*, allowing it to be applied at the object creation.

Args

chnl: int, optional Physical output assigned to the first audio stream of the object. Defaults to 0.

inc: int, optional Output channel increment value. Defaults to 1.

dur: float, optional Duration, in seconds, of the object's activation. The default is 0 and means infinite duration.

delay: float, optional Delay, in seconds, before the object's activation. Defaults to 0.

If *chnl* >= 0, successive streams increment the output number by *inc* and wrap around the global number of channels.

If *chnl* is negative, streams begin at 0, increment the output number by *inc* and wrap around the global number of channels. Then, the list of streams is scrambled.

If *chnl* is a list, successive values in the list will be assigned to successive streams.

ctrl (*map_list=None*, *title=None*, *wxnoserver=False*)

Opens a sliders window to control the parameters of the object. Only parameters that can be set to a PyoObject are allowed to be mapped on a slider.

If a list of values are given to a parameter, a multisliders will be used to control each stream independently.

Args

map_list: list of SLMap objects, optional Users defined set of parameters scaling. There is default scaling for each object that accept *ctrl* method.

title: string, optional Title of the window. If none is provided, the name of the class is used.

wxnoserver: boolean, optional With wxPython graphical toolkit, if True, tells the interpreter that there will be no server window.

If *wxnoserver* is set to True, the interpreter will not wait for the server GUI before showing the controller window.

input
PyoObject. Input signal to process.

AttackDetector

class AttackDetector(*input*, *deltime=0.005*, *cutoff=10*, *maxthresh=3*, *minthresh=-30*, *reltime=0.1*, *mul=1*, *add=0*)
Audio signal onset detection.

AttackDetector analyses an audio signal in input and output a trigger each time an onset is detected. An onset is a sharp amplitude rising while the signal had previously fall below a minimum threshold. Parameters must be carefully tuned depending on the nature of the analysed signal and the level of the background noise.

Parent *PyoObject*

Args

input: PyoObject Input signal to process.

deltime: float, optional Delay time, in seconds, between previous and current rms analysis to compare. Defaults to 0.005.

cutoff: float, optional Cutoff frequency, in Hz, of the amplitude follower's lowpass filter. Defaults to 10.

Higher values are more responsive and also more likely to give false onsets.

maxthresh: float, optional Attack threshold in positive dB (current rms must be higher than previous rms + maxthresh to be reported as an attack). Defaults to 3.0.

minthresh: float, optional Minimum threshold in dB (signal must fall below this threshold to allow a new attack to be detected). Defaults to -30.0.

reltime: float, optional Time, in seconds, to wait before reporting a new attack. Defaults to 0.1.

```
>>> s = Server(duplex=1).boot()
>>> s.start()
>>> a = Input()
>>> d = AttackDetector(a, deltime=0.005, cutoff=10, maxthresh=4, minthresh=-20,␣
↪reltime=0.05)
>>> exc = TrigEnv(d, HannTable(), dur=0.005, mul=BrownNoise(0.3))
>>> wgs = Waveguide(exc, freq=[100,200.1,300.3,400.5], dur=30).out()
```

setInput(*x*, *fadetime=0.05*)
Replace the *input* attribute.

Args

x: PyoObject New signal to process.

fadetime: float, optional Crossfade time between old and new input. Default to 0.05.

setDeltime(*x*)
Replace the *deltime* attribute.

Args

x: float New delay between rms analysis.

setCutoff(*x*)
Replace the *cutoff* attribute.

Args

x: float New cutoff for the follower lowpass filter.

setMaxthresh(*x*)
Replace the *maxthresh* attribute.

Args

x: float New attack threshold in dB.

setMinthresh(*x*)
Replace the *minthresh* attribute.

Args

x: float New minimum threshold in dB.

setReltime(*x*)
Replace the *reltime* attribute.

Args

x: float Time, in seconds, to wait before reporting a new attack.

readyToDetect()
Initializes variables in the ready state to detect an attack.

out(*chnl=0*, *inc=1*, *dur=0*, *delay=0*)
Start processing and send samples to audio output beginning at *chnl*.

This method returns *self*, allowing it to be applied at the object creation.

Args

chnl: int, optional Physical output assigned to the first audio stream of the object. Defaults to 0.

inc: int, optional Output channel increment value. Defaults to 1.

dur: float, optional Duration, in seconds, of the object's activation. The default is 0 and means infinite duration.

delay: float, optional Delay, in seconds, before the object's activation. Defaults to 0.

If *chnl* >= 0, successive streams increment the output number by *inc* and wrap around the global number of channels.

If *chnl* is negative, streams begin at 0, increment the output number by *inc* and wrap around the global number of channels. Then, the list of streams is scrambled.

If *chnl* is a list, successive values in the list will be assigned to successive streams.

ctrl(*map_list=None*, *title=None*, *wxnoserver=False*)
Opens a sliders window to control the parameters of the object. Only parameters that can be set to a PyoObject are allowed to be mapped on a slider.

If a list of values are given to a parameter, a multisliders will be used to control each stream independently.

Args

map_list: list of SLMap objects, optional Users defined set of parameters scaling. There is default scaling for each object that accept *ctrl* method.

title: string, optional Title of the window. If none is provided, the name of the class is used.

wxnoserver: boolean, optional With wxPython graphical toolkit, if True, tells the interpreter that there will be no server window.

If *wxnoserver* is set to True, the interpreter will not wait for the server GUI before showing the controller window.

input
PyoObject. Input signal to process.

deltime
float. Delay between rms analysis.

cutoff
float. Cutoff for the follower lowpass filter.

maxthresh
float. Attack threshold in dB.

minthresh
float. Minimum threshold in dB.

reltime
float. Time to wait before reporting a new attack.

Spectrum

class Spectrum(*input*, *size=1024*, *wintype=2*, *function=None*, *wintitle='Spectrum'*)
Spectrum analyzer and display.

Spectrum measures the magnitude of an input signal versus frequency within a user defined range. It can show both magnitude and frequency on linear or logarithmic scale.

Parent *PyoObject*

Args

input: PyoObject Input signal to process.

size: int {pow-of-two > 4}, optional FFT size. Must be a power of two greater than 4. The FFT size is the number of samples used in each analysis frame. Defaults to 1024.

wintype: int, optional Shape of the envelope used to filter each input frame. Possible shapes are :

0. rectangular (no windowing)
1. Hamming
2. Hanning
3. Bartlett (triangular)
4. Blackman 3-term

5. Blackman-Harris 4-term
6. Blackman-Harris 7-term
7. Tuckey (alpha = 0.66)
8. Sine (half-sine window)

function: python callable, optional If set, this function will be called with magnitudes (as list of lists, one list per channel). Useful if someone wants to save the analysis data into a text file. Defaults to None.

wintitle: string, optional GUI window title. Defaults to "Spectrum".

Note: Spectrum has no *out* method.

Spectrum has no *mul* and *add* attributes.

```
>>> s = Server().boot()
>>> s.start()
>>> a = SuperSaw(freq=[500,750], detune=0.6, bal=0.7, mul=0.5).out()
>>> spec = Spectrum(a, size=1024)
```

play(*dur=0*, *delay=0*)

Start processing without sending samples to output. This method is called automatically at the object creation.

This method returns *self*, allowing it to be applied at the object creation.

Args

dur: float, optional Duration, in seconds, of the object's activation. The default is 0 and means infinite duration.

delay: float, optional Delay, in seconds, before the object's activation. Defaults to 0.

out(*chnl=0*, *inc=1*, *dur=0*, *delay=0*)

Start processing and send samples to audio output beginning at *chnl*.

This method returns *self*, allowing it to be applied at the object creation.

Args

chnl: int, optional Physical output assigned to the first audio stream of the object. Defaults to 0.

inc: int, optional Output channel increment value. Defaults to 1.

dur: float, optional Duration, in seconds, of the object's activation. The default is 0 and means infinite duration.

delay: float, optional Delay, in seconds, before the object's activation. Defaults to 0.

If *chnl* >= 0, successive streams increment the output number by *inc* and wrap around the global number of channels.

If *chnl* is negative, streams begin at 0, increment the output number by *inc* and wrap around the global number of channels. Then, the list of streams is scrambled.

If *chnl* is a list, successive values in the list will be assigned to successive streams.

stop(*wait=0*)

Stop processing.

This method returns *self*, allowing it to be applied at the object creation.

Args

wait: float, optional Delay, in seconds, before the process is actually stopped. If autoStartChildren is activated in the Server, this value is propagated to the children objects. Defaults to 0.

Note: if the method setStopDelay(x) was called before calling stop(wait) with a positive *wait* value, the *wait* value won't overwrite the value given to setStopDelay for the current object, but will be the one propagated to children objects. This allow to set a waiting time for a specific object with setStopDelay whithout changing the global delay time given to the stop method.

Fader and Adsr objects ignore waiting time given to the stop method because they already implement a delayed processing triggered by the stop call.

---

setInput(*x*, *fadetime=0.05*)
Replace the *input* attribute.

Args

x: PyoObject New signal to process.

fadetime: float, optional Crossfade time between old and new input. Default to 0.05.

setSize(*x*)
Replace the *size* attribute.

Args

x: int new *size* attribute.

setWinType(*x*)
Replace the *wintype* attribute.

Args

x: int new *wintype* attribute.

setFunction(*function*)
Sets the function to be called to retrieve the analysis data.

Args

function: python callable The function called by the internal timer to retrieve the analysis data. The function must be created with one argument and will receive the data as a list of lists (one list per channel).

poll(*active*)
Turns on and off the analysis polling.

Args

active: boolean If True, starts the analysis polling, False to stop it. defaults to True.

polltime(*time*)
Sets the polling time in seconds.

Args

time: float Adjusts the frequency of the internal timer used to retrieve the current analysis frame. defaults to 0.05.

setLowFreq(*x*)
Sets the lower frequency, in Hz, returned by the analysis.

Args

x: float New low frequency in Hz. Adjusts the *lowbound* attribute, as *x / sr*.

setHighFreq(*x*)
Sets the higher frequency, in Hz, returned by the analysis.

Args

x: float New high frequency in Hz. Adjusts the *highbound* attribute, as *x / sr*.

setLowbound(*x*)
Sets the lower frequency, as multiplier of sr, returned by the analysis.

Returns the real low frequency en Hz.

Args

x: float {0 <= x <= 0.5} new *lowbound* attribute.

setHighbound(*x*)
Sets the higher frequency, as multiplier of sr, returned by the analysis.

Returns the real high frequency en Hz.

Args

x: float {0 <= x <= 0.5} new *highbound* attribute.

getLowfreq()
Returns the current lower frequency, in Hz, used by the analysis.

getHighfreq()
Returns the current higher frequency, in Hz, used by the analysis.

setWidth(*x*)
Sets the width, in pixels, of the current display.

Used internally to build the list of points to draw.

Args

x: int new *width* attribute.

setHeight(*x*)
Sets the height, in pixels, of the current display.

Used internally to build the list of points to draw.

Args

x: int new *height* attribute.

setFscaling(*x*)
Sets the frequency display to linear or logarithmic.

Args

x: boolean If True, the frequency display is logarithmic. False turns it back to linear. Defaults to False.

setMscaling(*x*)
Sets the magnitude display to linear or logarithmic.

Args

x: boolean If True, the magnitude display is logarithmic (which means in dB). False turns it back to linear. Defaults to True.

getFscaling()
Returns the scaling of the frequency display.

Returns True for logarithmic or False for linear.

getMscaling()
Returns the scaling of the magnitude display.

Returns True for logarithmic or False for linear.

setGain(*x*)
Set the gain of the analysis data. For drawing purpose.

Args

x: float new *gain* attribute, as linear values.

view(*title='Spectrum'*, *wxnoserver=False*)
Opens a window showing the result of the analysis.

Args

title: string, optional Window title. Defaults to "Spectrum".

wxnoserver: boolean, optional With wxPython graphical toolkit, if True, tells the interpreter that there will be no server window.

If *wxnoserver* is set to True, the interpreter will not wait for the server GUI before showing the controller window.

showChannelNames(*visible=True*)
If True (the default), channel names will be displayed in the window.

refreshView()
Updates the graphical display of the spectrum.

Called automatically by the internal timer.

input
PyoObject. Input signal to process.

size
int. FFT size.

wintype
int. Windowing method.

gain
float. Sets the gain of the analysis data.

lowbound
float. Lowest frequency (multiplier of sr) to output.

highbound
float. Highest frequency (multiplier of sr) to output.

width
int. Width, in pixels, of the current display.

height
int. Height, in pixels, of the current display.

fscaling
boolean. Scaling of the frequency display.

mscaling
boolean. Scaling of the magnitude display.

Scope

class Scope(*input*, *length=0.05*, *gain=0.67*, *function=None*, *wintitle='Scope'*)
Oscilloscope - audio waveform display.

Oscilloscopes are used to observe the change of an electrical signal over time.

Parent *PyoObject*

Args

input: PyoObject Input signal to process.

length: float, optional Length, in seconds, of the displayed window. Can't be a list. Defaults to 0.05.

gain: float, optional Linear gain applied to the signal to be displayed. Can't be a list. Defaults to 0.67.

function: python callable, optional If set, this function will be called with samples (as list of lists, one list per channel). Useful if someone wants to save the analysis data into a text file. Defaults to None.

wintitle: string, optional GUI window title. Defaults to "Scope".

Note: Scope has no *out* method.

Scope has no *mul* and *add* attributes.

```
>>> s = Server().boot()
>>> s.start()
>>> a = Sine([100,100.2], mul=0.7)
>>> b = Noise(0.1)
>>> scope = Scope(a+b)
```

setInput(*x*, *fadetime=0.05*)
Replace the *input* attribute.

Args

x: PyoObject New signal to process.

fadetime: float, optional Crossfade time between old and new input. Default to 0.05.

setLength(*x*)
Replace the *length* attribute.

Args

x: float new *length* attribute.

setGain(*x*)
Set the gain boost applied to the analysed data. For drawing purpose.

Args

x: float new *gain* attribute, as linear values.

poll(*active*)
Turns on and off the analysis polling.

Args

active: boolean If True, starts the analysis polling, False to stop it. defaults to True.

setWidth(*x*)
Gives the width of the display to the analyzer.

The analyzer needs this value to construct the list of points to draw on the display.

Args

x: int Width of the display in pixel value. The default width is 500.

setHeight(*x*)
Gives the height of the display to the analyzer.

The analyzer needs this value to construct the list of points to draw on the display.

Args

x: int Height of the display in pixel value. The default height is 400.

view(*title='Scope'*, *wxnoserver=False*)
Opens a window showing the incoming waveform.

Args

title: string, optional Window title. Defaults to "Scope".

wxnoserver: boolean, optional With wxPython graphical toolkit, if True, tells the interpreter that there will be no server window.

If *wxnoserver* is set to True, the interpreter will not wait for the server GUI before showing the controller window.

setFunction(*function*)
Sets the function to be called to retrieve the analysis data.

Args

function: python callable The function called by the internal timer to retrieve the analysis data. The function must be created with one argument and will receive the data as a list of lists (one list per channel).

showChannelNames(*visible=True*)
If True (the default), channel names will be displayed in the window.

refreshView()
Updates the graphical display of the scope.

Called automatically by the internal timer.

input
PyoObject. Input signal to process.

length
float. Window length.

gain
float. Sets the gain of the analysis data.

PeakAmp

class PeakAmp(*input*, *function=None*, *mul=1*, *add=0*)

Peak amplitude follower.

Output signal is the continuous peak amplitude of an input signal. A new peaking value is computed every buffer size. If *function* argument is not None, it should be a function that will be called periodically with a variable-length argument list containing the peaking values of all object's streams. Useful for meter drawing. Function definition must look like this:

```
>>> def getValues(*args)
```

Parent *PyoObject*

Args

input: PyoObject Input signal to process.

function: callable, optional Function that will be called with amplitude values in arguments. Default to None.

Note: The out() method is bypassed. PeakAmp's signal can not be sent to audio outs.

```
>>> s = Server().boot()
>>> s.start()
>>> sf = SfPlayer(SNDS_PATH + "/transparent.aif", loop=True, mul=.4).out()
>>> amp = PeakAmp(sf)
>>> n = Noise(mul=Port(amp)).out(1)
```

play(*dur=0*, *delay=0*)

Start processing without sending samples to output. This method is called automatically at the object creation.

This method returns *self*, allowing it to be applied at the object creation.

Args

dur: float, optional Duration, in seconds, of the object's activation. The default is 0 and means infinite duration.

delay: float, optional Delay, in seconds, before the object's activation. Defaults to 0.

stop(*wait=0*)

Stop processing.

This method returns *self*, allowing it to be applied at the object creation.

Args

wait: float, optional Delay, in seconds, before the process is actually stopped. If autoStartChildren is activated in the Server, this value is propagated to the children objects. Defaults to 0.

Note: if the method setStopDelay(x) was called before calling stop(wait) with a positive *wait* value, the *wait* value won't overwrite the value given to setStopDelay for the current object, but will be the one propagated to children objects. This allow to set a waiting time for a specific object with setStopDelay whithout changing the global delay time given to the stop method.

Fader and Adsr objects ignore waiting time given to the stop method because they already implement a delayed processing triggered by the stop call.

---

setInput(*x*, *fadetime=0.05*)
Replace the *input* attribute.

Args

x: PyoObject New signal to process.

fadetime: float, optional Crossfade time between old and new input. Default to 0.05.

setFunction(*x*)
Replace the *function* attribute.

Args

x: callable New function to call with amplitude values in arguments.

polltime(*x*)
Sets the delay, in seconds, between each call of the function.

Args

x: float New polling time in seconds.

out(*chnl=0*, *inc=1*, *dur=0*, *delay=0*)
Start processing and send samples to audio output beginning at *chnl*.

This method returns *self*, allowing it to be applied at the object creation.

Args

chnl: int, optional Physical output assigned to the first audio stream of the object. Defaults to 0.

inc: int, optional Output channel increment value. Defaults to 1.

dur: float, optional Duration, in seconds, of the object's activation. The default is 0 and means infinite duration.

delay: float, optional Delay, in seconds, before the object's activation. Defaults to 0.

If *chnl* >= 0, successive streams increment the output number by *inc* and wrap around the global number of channels.

If *chnl* is negative, streams begin at 0, increment the output number by *inc* and wrap around the global number of channels. Then, the list of streams is scrambled.

If *chnl* is a list, successive values in the list will be assigned to successive streams.

ctrl(*map_list=None*, *title=None*, *wxnoserver=False*)
Opens a sliders window to control the parameters of the object. Only parameters that can be set to a PyoObject are allowed to be mapped on a slider.

If a list of values are given to a parameter, a multisliders will be used to control each stream independently.

Args

map_list: list of SLMap objects, optional Users defined set of parameters scaling. There is default scaling for each object that accept *ctrl* method.

title: string, optional Title of the window. If none is provided, the name of the class is used.

wxnoserver: boolean, optional With wxPython graphical toolkit, if True, tells the interpreter that there will be no server window.

If *wxnoserver* is set to True, the interpreter will not wait for the server GUI before showing the controller window.

input
PyoObject. Input signal to process.

function
PyoObject. function signal to process.

RMS

class RMS(*input*, *function=None*, *mul=1*, *add=0*)
Returns the RMS (Root-Mean-Square) value of a signal.

Output signal is the continuous rms of an input signal. A new rms value is computed every buffer size. If *function* argument is not None, it should be a function that will be called periodically with a variable-length argument list containing the rms values of all object's streams. Useful for meter drawing. Function definition must look like this:

```
>>> def getValues(*args)
```

Parent *PyoObject*

Args

input: PyoObject Input signal to process.

function: callable, optional Function that will be called with amplitude values in arguments. Default to None.

Note: The out() method is bypassed. RMS's signal can not be sent to audio outs.

```
>>> s = Server().boot()
>>> s.start()
>>> sf = SfPlayer(SNDS_PATH + "/transparent.aif", loop=True, mul=.4).out()
>>> amp = RMS(sf)
>>> n = Noise(mul=Port(amp)).out(1)
```

play(*dur=0*, *delay=0*)
Start processing without sending samples to output. This method is called automatically at the object creation.

This method returns *self*, allowing it to be applied at the object creation.

Args

dur: float, optional Duration, in seconds, of the object's activation. The default is 0 and means infinite duration.

delay: float, optional Delay, in seconds, before the object's activation. Defaults to 0.

stop(*wait=0*)
Stop processing.

This method returns *self*, allowing it to be applied at the object creation.

Args

wait: float, optional Delay, in seconds, before the process is actually stopped. If autoStartChildren is activated in the Server, this value is propagated to the children objects. Defaults to 0.

Note: if the method setStopDelay(x) was called before calling stop(wait) with a positive *wait* value, the *wait* value won't overwrite the value given to setStopDelay for the current object, but will be the one propagated to children objects. This allow to set a waiting time for a specific object with setStopDelay whithout changing the global delay time given to the stop method.

Fader and Adsr objects ignore waiting time given to the stop method because they already implement a delayed processing triggered by the stop call.

---

setInput(*x*, *fadetime=0.05*)
Replace the *input* attribute.

Args

x: PyoObject New signal to process.

fadetime: float, optional Crossfade time between old and new input. Default to 0.05.

setFunction(*x*)
Replace the *function* attribute.

Args

x: callable New function to call with amplitude values in arguments.

polltime(*x*)
Sets the delay, in seconds, between each call of the function.

Args

x: float New polling time in seconds.

out(*chnl=0*, *inc=1*, *dur=0*, *delay=0*)
Start processing and send samples to audio output beginning at *chnl*.

This method returns *self*, allowing it to be applied at the object creation.

Args

chnl: int, optional Physical output assigned to the first audio stream of the object. Defaults to 0.

inc: int, optional Output channel increment value. Defaults to 1.

dur: float, optional Duration, in seconds, of the object's activation. The default is 0 and means infinite duration.

delay: float, optional Delay, in seconds, before the object's activation. Defaults to 0.

If *chnl* >= 0, successive streams increment the output number by *inc* and wrap around the global number of channels.

If *chnl* is negative, streams begin at 0, increment the output number by *inc* and wrap around the global number of channels. Then, the list of streams is scrambled.

If *chnl* is a list, successive values in the list will be assigned to successive streams.

ctrl(*map_list=None*, *title=None*, *wxnoserver=False*)
Opens a sliders window to control the parameters of the object. Only parameters that can be set to a PyoObject are allowed to be mapped on a slider.

If a list of values are given to a parameter, a multisliders will be used to control each stream independently.

Args

map_list: list of SLMap objects, optional Users defined set of parameters scaling. There is default scaling for each object that accept *ctrl* method.

title: string, optional Title of the window. If none is provided, the name of the class is used.

wxnoserver: boolean, optional With wxPython graphical toolkit, if True, tells the interpreter that there will be no server window.

If *wxnoserver* is set to True, the interpreter will not wait for the server GUI before showing the controller window.

input
PyoObject. Input signal to process.

function
PyoObject. function signal to process.

Arithmetic

Tools to perform arithmetic operations on audio signals.

Sin

class Sin(*input*, *mul=1*, *add=0*)
Performs a sine function on audio signal.

Returns the sine of audio signal as input.

Parent *PyoObject*

Args

input: PyoObject Input signal, angle in radians.

```
>>> s = Server().boot()
>>> s.start()
>>> import math
>>> a = Phasor(500, mul=math.pi*2)
>>> b = Sin(a, mul=.3).mix(2).out()
```

setInput(*x*, *fadetime=0.05*)
Replace the *input* attribute.

Args

x: PyoObject New signal to process.

fadetime: float, optional Crossfade time between old and new input. Default to 0.05.

input
PyoObject. Input signal to process.

Cos

class Cos(*input*, *mul=1*, *add=0*)

Performs a cosine function on audio signal.

Returns the cosine of audio signal as input.

Parent *PyoObject*

Args

input: PyoObject Input signal, angle in radians.

```
>>> s = Server().boot()
>>> s.start()
>>> import math
>>> a = Phasor(500, mul=math.pi*2)
>>> b = Cos(a, mul=.3).mix(2).out()
```

setInput(*x*, *fadetime=0.05*)

Replace the *input* attribute.

Args

x: PyoObject New signal to process.

fadetime: float, optional Crossfade time between old and new input. Default to 0.05.

input

PyoObject. Input signal to process.

Tan

class Tan(*input*, *mul=1*, *add=0*)

Performs a tangent function on audio signal.

Returns the tangent of audio signal as input.

Parent *PyoObject*

Args

input: PyoObject Input signal, angle in radians.

```
>>> s = Server().boot()
>>> s.start()
>>> # Tangent panning function
>>> import math
>>> src = Sine(mul=.3)
>>> a = Phasor(freq=1, mul=90, add=-45)
>>> b = Tan(Abs(a*math.pi/180))
>>> b1 = 1.0 - b
>>> oL = src * b
>>> oR = src * b1
>>> oL.out()
>>> oR.out(1)
```

setInput(*x*, *fadetime=0.05*)

Replace the *input* attribute.

Args

x: PyoObject New signal to process.

fadetime: float, optional Crossfade time between old and new input. Default to 0.05.

input
PyoObject. Input signal to process.

Tanh

class Tanh(*input*, *mul=1*, *add=0*)
Performs a hyperbolic tangent function on audio signal.

Returns the hyperbolic tangent of audio signal as input.

Parent *PyoObject*

Args

input: PyoObject Input signal, angle in radians.

```
>>> s = Server().boot()
>>> s.start()
>>> import math
>>> a = Phasor(250, mul=math.pi*2)
>>> b = Tanh(Sin(a, mul=10), mul=0.3).mix(2).out()
```

setInput(*x*, *fadetime=0.05*)
Replace the *input* attribute.

Args

x: PyoObject New signal to process.

fadetime: float, optional Crossfade time between old and new input. Default to 0.05.

input
PyoObject. Input signal to process.

Abs

class Abs(*input*, *mul=1*, *add=0*)
Performs an absolute function on audio signal.

Returns the absolute value of audio signal as input.

Parent *PyoObject*

Args

input: PyoObject Input signal to process.

```
>>> s = Server().boot()
>>> s.start()
>>> # Back-and-Forth playback
>>> t = SndTable(SNDS_PATH + "/transparent.aif")
>>> a = Phasor(freq=t.getRate()*0.5, mul=2, add=-1)
>>> b = Pointer(table=t, index=Abs(a), mul=0.5).mix(2).out()
```

setInput(*x*, *fadetime=0.05*)
Replace the *input* attribute.

Args

x: PyoObject New signal to process.

fadetime: float, optional Crossfade time between old and new input. Default to 0.05.

input
PyoObject. Input signal to process.

Sqrt

class Sqrt(*input*, *mul=1*, *add=0*)
Performs a square-root function on audio signal.

Returns the square-root value of audio signal as input.

Parent *PyoObject*

Args

input: PyoObject Input signal to process.

```
>>> s = Server().boot()
>>> s.start()
>>> # Equal-power panning function
>>> src = Sine(mul=.3)
>>> a = Abs(Phasor(freq=1, mul=2, add=-1))
>>> left = Sqrt(1.0 - a)
>>> right = Sqrt(a)
>>> oL = src * left
>>> oR = src * right
>>> oL.out()
>>> oR.out(1)
```

setInput(*x*, *fadetime=0.05*)
Replace the *input* attribute.

Args

x: PyoObject New signal to process.

fadetime: float, optional Crossfade time between old and new input. Default to 0.05.

input
PyoObject. Input signal to process.

Log

class Log(*input*, *mul=1*, *add=0*)
Performs a natural log function on audio signal.

Returns the natural log value of of audio signal as input. Values less than 0.0 return 0.0.

Parent *PyoObject*

Args

input: PyoObject Input signal to process.

```
>>> s = Server().boot()
>>> s.start()
# Logarithmic amplitude envelope
>>> a = LFO(freq=1, type=3, mul=0.2, add=1.2) # triangle
>>> b = Log(a)
>>> c = SineLoop(freq=[300,301], feedback=0.05, mul=b).out()
```

setInput (*x*, *fadetime=0.05*)
Replace the *input* attribute.

Args

x: PyoObject New signal to process.

fadetime: float, optional Crossfade time between old and new input. Default to 0.05.

input
PyoObject. Input signal to process.

Log2

class Log2 (*input*, *mul=1*, *add=0*)
Performs a base 2 log function on audio signal.

Returns the base 2 log value of audio signal as input. Values less than 0.0 return 0.0.

Parent *PyoObject*

Args

input: PyoObject Input signal to process.

```
>>> s = Server().boot()
>>> s.start()
# Logarithmic amplitude envelope
>>> a = LFO(freq=1, type=3, mul=0.1, add=1.1) # triangle
>>> b = Log2(a)
>>> c = SineLoop(freq=[300,301], feedback=0.05, mul=b).out()
```

setInput (*x*, *fadetime=0.05*)
Replace the *input* attribute.

Args

x: PyoObject New signal to process.

fadetime: float, optional Crossfade time between old and new input. Default to 0.05.

input
PyoObject. Input signal to process.

Log10

class Log10 (*input*, *mul=1*, *add=0*)
Performs a base 10 log function on audio signal.

Returns the base 10 log value of audio signal as input. Values less than 0.0 return 0.0.

Parent *PyoObject*

Args

input: PyoObject Input signal to process.

```
>>> s = Server().boot()
>>> s.start()
# Logarithmic amplitude envelope
>>> a = LFO(freq=1, type=3, mul=0.4, add=1.4) # triangle
>>> b = Log10(a)
>>> c = SineLoop(freq=[300,301], feedback=0.05, mul=b).out()
```

setInput(*x*, *fadetime=0.05*)
Replace the *input* attribute.

Args

x: PyoObject New signal to process.

fadetime: float, optional Crossfade time between old and new input. Default to 0.05.

input
PyoObject. Input signal to process.

Atan2

class Atan2(*b=1*, *a=1*, *mul=1*, *add=0*)
Computes the principal value of the arc tangent of b/a.

Computes the principal value of the arc tangent of b/a, using the signs of both arguments to determine the quadrant of the return value.

Parent *PyoObject*

Args

b: float or PyoObject, optional Numerator. Defaults to 1.

a: float or PyoObject, optional Denominator. Defaults to 1.

```
>>> s = Server().boot()
>>> s.start()
>>> # Simple distortion
>>> a = Sine(freq=[200,200.3])
>>> lf = Sine(freq=1, mul=.2, add=.2)
>>> dist = Atan2(a, lf)
>>> lp = Tone(dist, freq=2000, mul=.1).out()
```

setB(*x*)
Replace the *b* attribute.

Args

x: float or PyoObject new *b* attribute.

setA(*x*)
Replace the *a* attribute.

Args

x: float or PyoObject new *a* attribute.

b
float or PyoObject. Numerator.

a
float or PyoObject. Denominator.

Floor

class Floor(*input*, *mul=1*, *add=0*)
Rounds to largest integral value not greater than audio signal.

For each samples in the input signal, rounds to the largest integral value not greater than the sample value.

Parent *PyoObject*

Args

input: PyoObject Input signal to process.

```
>>> s = Server().boot()
>>> s.start()
>>> # Clipping frequencies
>>> sweep = Phasor(freq=[1,.67], mul=4)
>>> flo = Floor(sweep, mul=50, add=200)
>>> a = SineLoop(freq=flo, feedback=.1, mul=.3).out()
```

setInput(*x*, *fadetime=0.05*)
Replace the *input* attribute.

Args

x: PyoObject New signal to process.

fadetime: float, optional Crossfade time between old and new input. Default to 0.05.

input
PyoObject. Input signal to process.

Ceil

class Ceil(*input*, *mul=1*, *add=0*)
Rounds to smallest integral value greater than or equal to the input signal.

For each samples in the input signal, rounds to the smallest integral value greater than or equal to the sample value.

Parent *PyoObject*

Args

input: PyoObject Input signal to process.

```
>>> s = Server().boot()
>>> s.start()
>>> # Clipping frequencies
>>> sweep = Phasor(freq=[1,.67], mul=4)
>>> flo = Ceil(sweep, mul=50, add=200)
>>> a = SineLoop(freq=flo, feedback=.1, mul=.3).out()
```

setInput(*x*, *fadetime=0.05*)
Replace the *input* attribute.

Args

x: PyoObject New signal to process.

fadetime: float, optional Crossfade time between old and new input. Default to 0.05.

input
PyoObject. Input signal to process.

Round

class Round(*input*, *mul=1*, *add=0*)
Rounds to the nearest integer value in a floating-point format.

For each samples in the input signal, rounds to the nearest integer value of the sample value.

Parent *PyoObject*

Args

input: PyoObject Input signal to process.

```
>>> s = Server().boot()
>>> s.start()
>>> # Clipping frequencies
>>> sweep = Phasor(freq=[1,.67], mul=4)
>>> flo = Round(sweep, mul=50, add=200)
>>> a = SineLoop(freq=flo, feedback=.1, mul=.3).out()
```

setInput(*x*, *fadetime=0.05*)
Replace the *input* attribute.

Args

x: PyoObject New signal to process.

fadetime: float, optional Crossfade time between old and new input. Default to 0.05.

input
PyoObject. Input signal to process.

Pow

class Pow(*base=10*, *exponent=1*, *mul=1*, *add=0*)
Performs a power function on audio signal.

Parent *PyoObject*

Args

base: float or PyoObject, optional Base composant. Defaults to 10.

exponent: float or PyoObject, optional Exponent composant. Defaults to 1.

```
>>> s = Server().boot()
>>> s.start()
>>> # Exponential amplitude envelope
```

(continues on next page)

(continued from previous page)

```
>>> a = LFO(freq=1, type=3, mul=0.5, add=0.5)
>>> b = Pow(Clip(a, 0, 1), 4, mul=.3)
>>> c = SineLoop(freq=[300,301], feedback=0.05, mul=b).out()
```

setBase(*x*)
Replace the *base* attribute.

Args

x: float or PyoObject new *base* attribute.

setExponent(*x*)
Replace the *exponent* attribute.

Args

x: float or PyoObject new *exponent* attribute.

base
float or PyoObject. Base composant.

exponent
float or PyoObject. Exponent composant.

Exp

class Exp(*input*, *mul=1*, *add=0*)
Calculates the value of e to the power of x.

Returns the value of e to the power of x, where e is the base of the natural logarithm, 2.718281828...

Parent *PyoObject*

Args

input: PyoObject Input signal, the exponent.

```
>>> s = Server().boot()
>>> s.start()
>>> a = Sine(freq=200)
>>> lf = Sine(freq=.5, mul=5, add=6)
>>> # Tanh style distortion
>>> t = Exp(2 * a * lf)
>>> th = (t - 1) / (t + 1)
>>> out = (th * 0.3).out()
```

setInput(*x*, *fadetime=0.05*)
Replace the *input* attribute.

Args

x: PyoObject New signal to process.

fadetime: float, optional Crossfade time between old and new input. Default to 0.05.

input
PyoObject. Input signal to process.

Div

class Div(*a=1*, *b=1*, *mul=1*, *add=0*)

Divides a by b.

Parent *PyoObject*

Args

a: float or PyoObject, optional Numerator. Defaults to 1.

b: float or PyoObject, optional Denominator. Defaults to 1.

```
>>> s = Server().boot()
>>> s.start()
>>> a = Sine(freq=[400, 500])
>>> b = Randi(min=1, max=10, freq=5.00)
>>> c = Div(a, b, mul=0.3).out()
```

setA(*x*)

Replace the *a* attribute.

Args

x: float or PyoObject new *a* attribute.

setB(*x*)

Replace the *b* attribute.

Args

x: float or PyoObject new *b* attribute.

a

float or PyoObject. Numerator.

b

float or PyoObject. Denominator.

Sub

class Sub(*a=1*, *b=1*, *mul=1*, *add=0*)

Substracts b from a.

Parent *PyoObject*

Args

a: float or PyoObject, optional Left operand. Defaults to 1.

b: float or PyoObject, optional Right operand. Defaults to 1.

```
>>> s = Server().boot()
>>> s.start()
>>> a = Sig([400, 500])
>>> b = Randi(min=50, max=100, freq=5.00)
>>> c = Sub(a, b)
>>> d = SineLoop(freq=c, feedback=0.08, mul=0.3).out()
```

setA(*x*)

Replace the *a* attribute.

Args

x: float or PyoObject new *a* attribute.

setB(*x*)
Replace the *b* attribute.

Args

x: float or PyoObject new *b* attribute.

a
float or PyoObject. Left operand.

b
float or PyoObject. Right operand.

Control Signals

Objects designed to create parameter's control at audio rate.

These objects can be used to create envelopes, line segments and conversion from python number to audio signal.

The audio streams of these objects can't be sent to the output soundcard.

Fader

class Fader(*fadein=0.01*, *fadeout=0.1*, *dur=0*, *mul=1*, *add=0*)
Fadein - fadeout envelope generator.

Generate an amplitude envelope between 0 and 1 with control on fade times and total duration of the envelope.

The play() method starts the envelope and is not called at the object creation time.

Parent *PyoObject*

Args

fadein: float, optional Rising time of the envelope in seconds. Defaults to 0.01.

fadeout: float, optional Falling time of the envelope in seconds. Defaults to 0.1.

dur: float, optional Total duration of the envelope in seocnds. Defaults to 0, which means wait for the stop() method to start the fadeout.

Note: The out() method is bypassed. Fader's signal can not be sent to audio outs.

The play() method starts the envelope.

The stop() method calls the envelope's release phase if *dur* = 0.

As of version 0.8.0, exponential or logarithmic envelopes can be created with the exponent factor (see setExp() method).

```
>>> s = Server().boot()
>>> s.start()
>>> f = Fader(fadein=0.5, fadeout=0.5, dur=2, mul=.5)
>>> a = BrownNoise(mul=f).mix(2).out()
>>> def repeat():
```

(continues on next page)

(continued from previous page)

```
...     f.play()
>>> pat = Pattern(function=repeat, time=2).play()
```

out (*chnl=0*, *inc=1*, *dur=0*, *delay=0*)
Start processing and send samples to audio output beginning at *chnl*.

This method returns *self*, allowing it to be applied at the object creation.

Args

chnl: int, optional Physical output assigned to the first audio stream of the object. Defaults to 0.

inc: int, optional Output channel increment value. Defaults to 1.

dur: float, optional Duration, in seconds, of the object's activation. The default is 0 and means infinite duration.

delay: float, optional Delay, in seconds, before the object's activation. Defaults to 0.

If *chnl* >= 0, successive streams increment the output number by *inc* and wrap around the global number of channels.

If *chnl* is negative, streams begin at 0, increment the output number by *inc* and wrap around the global number of channels. Then, the list of streams is scrambled.

If *chnl* is a list, successive values in the list will be assigned to successive streams.

setFadein (*x*)
Replace the *fadein* attribute.

Args

x: float new *fadein* attribute.

setFadeout (*x*)
Replace the *fadeout* attribute.

Args

x: float new *fadeout* attribute.

setDur (*x*)
Replace the *dur* attribute.

Args

x: float new *dur* attribute.

setExp (*x*)
Sets an exponent factor to create exponential or logarithmic envelope.

The default value is 1.0, which means linear segments. A value higher than 1.0 will produce exponential segments while a value between 0 and 1 will produce logarithmic segments. Must be > 0.0.

Args

x: float new *exp* attribute.

ctrl (*map_list=None*, *title=None*, *wxnoserver=False*)
Opens a sliders window to control the parameters of the object. Only parameters that can be set to a PyoObject are allowed to be mapped on a slider.

If a list of values are given to a parameter, a multisliders will be used to control each stream independently.

Args

map_list: list of SLMap objects, optional Users defined set of parameters scaling. There is default scaling for each object that accept *ctrl* method.

title: string, optional Title of the window. If none is provided, the name of the class is used.

wxnoserver: boolean, optional With wxPython graphical toolkit, if True, tells the interpreter that there will be no server window.

If *wxnoserver* is set to True, the interpreter will not wait for the server GUI before showing the controller window.

fadein
float. Rising time of the envelope in seconds.

fadeout
float. Falling time of the envelope in seconds.

dur
float. Total duration of the envelope.

exp
float. Exponent factor of the envelope.

Adsr

class Adsr(*attack=0.01*, *decay=0.05*, *sustain=0.707*, *release=0.1*, *dur=0*, *mul=1*, *add=0*)
Attack - Decay - Sustain - Release envelope generator.

Calculates the classical ADSR envelope using linear segments. Duration can be set to 0 to give an infinite sustain. In this case, the stop() method calls the envelope release part.

The play() method starts the envelope and is not called at the object creation time.

Parent *PyoObject*

Args

attack: float, optional Duration of the attack phase in seconds. Defaults to 0.01.

decay: float, optional Duration of the decay in seconds. Defaults to 0.05.

sustain: float, optional Amplitude of the sustain phase. Defaults to 0.707.

release: float, optional Duration of the release in seconds. Defaults to 0.1.

dur: float, optional Total duration of the envelope in seconds. Defaults to 0, which means wait for the stop() method to start the release phase.

Note: The out() method is bypassed. Adsr's signal can not be sent to audio outs.

The play() method starts the envelope.

The stop() method calls the envelope's release phase if *dur* = 0.

As of version 0.8.0, exponential or logarithmic envelopes can be created with the exponent factor (see setExp() method).

```
>>> s = Server().boot()
>>> s.start()
>>> f = Adsr(attack=.01, decay=.2, sustain=.5, release=.1, dur=2, mul=.5)
>>> a = BrownNoise(mul=f).mix(2).out()
>>> def repeat():
...     f.play()
>>> pat = Pattern(function=repeat, time=2).play()
```

out (*chnl=0*, *inc=1*, *dur=0*, *delay=0*)

Start processing and send samples to audio output beginning at *chnl*.

This method returns *self*, allowing it to be applied at the object creation.

Args

chnl: int, optional Physical output assigned to the first audio stream of the object. Defaults to 0.

inc: int, optional Output channel increment value. Defaults to 1.

dur: float, optional Duration, in seconds, of the object's activation. The default is 0 and means infinite duration.

delay: float, optional Delay, in seconds, before the object's activation. Defaults to 0.

If *chnl* >= 0, successive streams increment the output number by *inc* and wrap around the global number of channels.

If *chnl* is negative, streams begin at 0, increment the output number by *inc* and wrap around the global number of channels. Then, the list of streams is scrambled.

If *chnl* is a list, successive values in the list will be assigned to successive streams.

setAttack (*x*)

Replace the *attack* attribute.

Args

x: float new *attack* attribute.

setDecay (*x*)

Replace the *decay* attribute.

Args

x: float new *decay* attribute.

setSustain (*x*)

Replace the *sustain* attribute.

Args

x: float new *sustain* attribute.

setRelease (*x*)

Replace the *sustain* attribute.

Args

x: float new *sustain* attribute.

setDur (*x*)

Replace the *dur* attribute.

Args

x: float new *dur* attribute.

setExp(*x*)
Sets an exponent factor to create exponential or logarithmic envelope.

The default value is 1.0, which means linear segments. A value higher than 1.0 will produce exponential segments while a value between 0 and 1 will produce logarithmic segments. Must be > 0.0.

Args

x: float new *exp* attribute.

ctrl(*map_list=None*, *title=None*, *wxnoserver=False*)
Opens a sliders window to control the parameters of the object. Only parameters that can be set to a PyoObject are allowed to be mapped on a slider.

If a list of values are given to a parameter, a multisliders will be used to control each stream independently.

Args

map_list: list of SLMap objects, optional Users defined set of parameters scaling. There is default scaling for each object that accept *ctrl* method.

title: string, optional Title of the window. If none is provided, the name of the class is used.

wxnoserver: boolean, optional With wxPython graphical toolkit, if True, tells the interpreter that there will be no server window.

If *wxnoserver* is set to True, the interpreter will not wait for the server GUI before showing the controller window.

attack
float. Duration of the attack phase in seconds.

decay
float. Duration of the decay phase in seconds.

sustain
float. Amplitude of the sustain phase.

release
float. Duration of the release phase in seconds.

dur
float. Total duration of the envelope.

exp
float. Exponent factor of the envelope.

Linseg

class Linseg(*list*, *loop=False*, *initToFirstVal=False*, *mul=1*, *add=0*)
Draw a series of line segments between specified break-points.

The play() method starts the envelope and is not called at the object creation time.

Parent `PyoObject`

Args

list: list of tuples Points used to construct the line segments. Each tuple is a new point in the form (time, value).

Times are given in seconds and must be in increasing order.

loop: boolean, optional Looping mode. Defaults to False.

initToFirstVal: boolean, optional If True, audio buffer will be filled at initialization with the first value of the line. Defaults to False.

Note: The out() method is bypassed. Linseg's signal can not be sent to audio outs.

```
>>> s = Server().boot()
>>> s.start()
>>> l = Linseg([(0,500),(.03,1000),(.1,700),(1,500),(2,500)], loop=True)
>>> a = Sine(freq=l, mul=.3).mix(2).out()
>>> # then call:
>>> l.play()
```

out(*chnl=0*, *inc=1*, *dur=0*, *delay=0*)

Start processing and send samples to audio output beginning at *chnl*.

This method returns *self*, allowing it to be applied at the object creation.

Args

chnl: int, optional Physical output assigned to the first audio stream of the object. Defaults to 0.

inc: int, optional Output channel increment value. Defaults to 1.

dur: float, optional Duration, in seconds, of the object's activation. The default is 0 and means infinite duration.

delay: float, optional Delay, in seconds, before the object's activation. Defaults to 0.

If *chnl* >= 0, successive streams increment the output number by *inc* and wrap around the global number of channels.

If *chnl* is negative, streams begin at 0, increment the output number by *inc* and wrap around the global number of channels. Then, the list of streams is scrambled.

If *chnl* is a list, successive values in the list will be assigned to successive streams.

setList(*x*)

Replace the *list* attribute.

Args

x: list of tuples new *list* attribute.

replace(*x*)

Alias for *setList* method.

Args

x: list of tuples new *list* attribute.

setLoop(*x*)

Replace the *loop* attribute.

Args

x: boolean new *loop* attribute.

pause()

Toggles between play and stop mode without reset.

graph (*xlen=None*, *yrange=None*, *title=None*, *wxnoserver=False*)
Opens a grapher window to control the shape of the envelope.

When editing the grapher with the mouse, the new set of points will be send to the object on mouse up.

Ctrl+C with focus on the grapher will copy the list of points to the clipboard, giving an easy way to insert the new shape in a script.

Args

xlen: float, optional Set the maximum value of the X axis of the graph. If None, the maximum value is retrieve from the current list of points.

yrange: tuple, optional Set the min and max values of the Y axis of the graph. If None, min and max are retrieve from the current list of points.

title: string, optional Title of the window. If none is provided, the name of the class is used.

wxnoserver: boolean, optional With wxPython graphical toolkit, if True, tells the interpreter that there will be no server window.

If *wxnoserver* is set to True, the interpreter will not wait for the server GUI before showing the controller window.

list
float. List of points (time, value).

loop
boolean. Looping mode.

Expseg

class Expseg (*list*, *loop=False*, *exp=10*, *inverse=True*, *initToFirstVal=False*, *mul=1*, *add=0*)
Draw a series of exponential segments between specified break-points.

The play() method starts the envelope and is not called at the object creation time.

Parent *PyoObject*

Args

list: list of tuples Points used to construct the line segments. Each tuple is a new point in the form (time, value).

Times are given in seconds and must be in increasing order.

loop: boolean, optional Looping mode. Defaults to False.

exp: float, optional Exponent factor. Used to control the slope of the curves. Defaults to 10.

inverse: boolean, optional If True, downward slope will be inversed. Useful to create biexponential curves. Defaults to True.

initToFirstVal: boolean, optional If True, audio buffer will be filled at initialization with the first value of the line. Defaults to False.

Note: The out() method is bypassed. Expseg's signal can not be sent to audio outs.

```
>>> s = Server().boot()
>>> s.start()
>>> l = Expseg([(0,500),(.03,1000),(.1,700),(1,500),(2,500)], loop=True)
>>> a = Sine(freq=l, mul=.3).mix(2).out()
>>> # then call:
>>> l.play()
```

out (*chnl=0*, *inc=1*, *dur=0*, *delay=0*)
Start processing and send samples to audio output beginning at *chnl*.

This method returns *self*, allowing it to be applied at the object creation.

Args

chnl: int, optional Physical output assigned to the first audio stream of the object. Defaults to 0.

inc: int, optional Output channel increment value. Defaults to 1.

dur: float, optional Duration, in seconds, of the object's activation. The default is 0 and means infinite duration.

delay: float, optional Delay, in seconds, before the object's activation. Defaults to 0.

If *chnl* >= 0, successive streams increment the output number by *inc* and wrap around the global number of channels.

If *chnl* is negative, streams begin at 0, increment the output number by *inc* and wrap around the global number of channels. Then, the list of streams is scrambled.

If *chnl* is a list, successive values in the list will be assigned to successive streams.

setList (*x*)
Replace the *list* attribute.

Args

x: list of tuples new *list* attribute.

setLoop (*x*)
Replace the *loop* attribute.

Args

x: boolean new *loop* attribute.

setExp (*x*)
Replace the *exp* attribute.

Args

x: float new *exp* attribute.

setInverse (*x*)
Replace the *inverse* attribute.

Args

x: boolean new *inverse* attribute.

replace (*x*)
Alias for *setList* method.

Args

x: list of tuples new *list* attribute.

pause()
Toggles between play and stop mode without reset.

graph(*xlen=None*, *yrange=None*, *title=None*, *wxnoserver=False*)
Opens a grapher window to control the shape of the envelope.

When editing the grapher with the mouse, the new set of points will be send to the object on mouse up.

Ctrl+C with focus on the grapher will copy the list of points to the clipboard, giving an easy way to insert the new shape in a script.

Args

xlen: float, optional Set the maximum value of the X axis of the graph. If None, the maximum value is retrieve from the current list of points. Defaults to None.

yrange: tuple, optional Set the min and max values of the Y axis of the graph. If None, min and max are retrieve from the current list of points. Defaults to None.

title: string, optional Title of the window. If none is provided, the name of the class is used.

wxnoserver: boolean, optional With wxPython graphical toolkit, if True, tells the interpreter that there will be no server window.

If *wxnoserver* is set to True, the interpreter will not wait for the server GUI before showing the controller window.

list
float. List of points (time, value).

loop
boolean. Looping mode.

exp
float. Exponent factor.

inverse
boolean. Inverse downward slope.

Sig

class Sig(*value*, *mul=1*, *add=0*)
Convert numeric value to PyoObject signal.

Parent *PyoObject*

Args

value: float or PyoObject Numerical value to convert.

```
>>> import random
>>> s = Server().boot()
>>> s.start()
>>> fr = Sig(value=400)
>>> p = Port(fr, risetime=0.001, falltime=0.001)
>>> a = SineLoop(freq=p, feedback=0.08, mul=.3).out()
>>> b = SineLoop(freq=p*1.005, feedback=0.08, mul=.3).out(1)
>>> def pick_new_freq():
...     fr.value = random.randrange(300,601,50)
>>> pat = Pattern(function=pick_new_freq, time=0.5).play()
```

setValue(*x*)
Changes the value of the signal stream.

Args

x: float or PyoObject Numerical value to convert.

ctrl(*map_list=None*, *title=None*, *wxnoserver=False*)
Opens a sliders window to control the parameters of the object. Only parameters that can be set to a PyoObject are allowed to be mapped on a slider.

If a list of values are given to a parameter, a multisliders will be used to control each stream independently.

Args

map_list: list of SLMap objects, optional Users defined set of parameters scaling. There is default scaling for each object that accept *ctrl* method.

title: string, optional Title of the window. If none is provided, the name of the class is used.

wxnoserver: boolean, optional With wxPython graphical toolkit, if True, tells the interpreter that there will be no server window.

If *wxnoserver* is set to True, the interpreter will not wait for the server GUI before showing the controller window.

value
float or PyoObject. Numerical value to convert.

SigTo

class SigTo(*value*, *time=0.025*, *init=0.0*, *mul=1*, *add=0*)
Convert numeric value to PyoObject signal with portamento.

When *value* is changed, a ramp is applied from the current value to the new value. Can be used with PyoObject to apply a linear portamento on an audio signal.

Parent *PyoObject*

Args

value: float or PyoObject Numerical value to convert.

time: float or PyoObject, optional Ramp time, in seconds, to reach the new value. Defaults to 0.025.

init: float, optional Initial value of the internal memory. Defaults to 0.

Note: The out() method is bypassed. SigTo's signal can not be sent to audio outs.

```
>>> import random
>>> s = Server().boot()
>>> s.start()
>>> fr = SigTo(value=200, time=0.5, init=200)
>>> a = SineLoop(freq=fr, feedback=0.08, mul=.3).out()
>>> b = SineLoop(freq=fr*1.005, feedback=0.08, mul=.3).out(1)
>>> def pick_new_freq():
...     fr.value = random.randrange(200,501,50)
>>> pat = Pattern(function=pick_new_freq, time=1).play()
```

setValue(*x*)
Changes the value of the signal stream.

Args

x: float or PyoObject Numerical value to convert.

setTime(*x*)
Changes the ramp time of the object.

Args

x: float or PyoObject New ramp time in seconds.

ctrl(*map_list=None*, *title=None*, *wxnoserver=False*)
Opens a sliders window to control the parameters of the object. Only parameters that can be set to a PyoObject are allowed to be mapped on a slider.

If a list of values are given to a parameter, a multisliders will be used to control each stream independently.

Args

map_list: list of SLMap objects, optional Users defined set of parameters scaling. There is default scaling for each object that accept *ctrl* method.

title: string, optional Title of the window. If none is provided, the name of the class is used.

wxnoserver: boolean, optional With wxPython graphical toolkit, if True, tells the interpreter that there will be no server window.

If *wxnoserver* is set to True, the interpreter will not wait for the server GUI before showing the controller window.

value
float or PyoObject. Numerical value to convert.

time
floator PyoObject. Ramp time in seconds.

Dynamic management

Objects to modify the dynamic range and sample quality of audio signals.

Clip

class Clip(*input*, *min=-1.0*, *max=1.0*, *mul=1*, *add=0*)
Clips a signal to a predefined limit.

Parent *PyoObject*

Args

input: PyoObject Input signal to process.

min: float or PyoObject, optional Minimum possible value. Defaults to -1.

max: float or PyoObject, optional Maximum possible value. Defaults to 1.

```
>>> s = Server().boot()
>>> s.start()
>>> a = SfPlayer(SNDS_PATH + "/transparent.aif", loop=True)
>>> lfoup = Sine(freq=.25, mul=.48, add=.5)
>>> lfodown = 0 - lfoup
>>> c = Clip(a, min=lfodown, max=lfoup, mul=.4).mix(2).out()
```

setInput (*x*, *fadetime=0.05*)
Replace the *input* attribute.

Args

x: PyoObject New signal to process.

fadetime: float, optional Crossfade time between old and new input. Defaults to 0.05.

setMin (*x*)
Replace the *min* attribute.

Args

x: float or PyoObject New *min* attribute.

setMax (*x*)
Replace the *max* attribute.

Args

x: float or PyoObject New *max* attribute.

ctrl (*map_list=None*, *title=None*, *wxnoserver=False*)
Opens a sliders window to control the parameters of the object. Only parameters that can be set to a PyoObject are allowed to be mapped on a slider.

If a list of values are given to a parameter, a multisliders will be used to control each stream independently.

Args

map_list: list of SLMap objects, optional Users defined set of parameters scaling. There is default scaling for each object that accept *ctrl* method.

title: string, optional Title of the window. If none is provided, the name of the class is used.

wxnoserver: boolean, optional With wxPython graphical toolkit, if True, tells the interpreter that there will be no server window.

If *wxnoserver* is set to True, the interpreter will not wait for the server GUI before showing the controller window.

input
PyoObject. Input signal to process.

min
float or PyoObject. Minimum possible value.

max
float or PyoObject. Maximum possible value.

Degrade

class Degrade (*input*, *bitdepth=16*, *srscale=1.0*, *mul=1*, *add=0*)

Signal quality reducer.

Degrade takes an audio signal and reduces the sampling rate and/or bit-depth as specified.

Parent *PyoObject*

Args

input: PyoObject Input signal to process.

bitdepth: float or PyoObject, optional Signal quantization in bits. Must be in range 1 -> 32. Defaults to 16.

srscale: float or PyoObject, optional Sampling rate multiplier. Must be in range 0.0009765625 -> 1. Defaults to 1.

```
>>> s = Server().boot()
>>> s.start()
>>> t = SquareTable()
>>> a = Osc(table=t, freq=[100,101], mul=.5)
>>> lfo = Sine(freq=.2, mul=6, add=8)
>>> lfo2 = Sine(freq=.25, mul=.45, add=.55)
>>> b = Degrade(a, bitdepth=lfo, srscale=lfo2, mul=.3).out()
```

setInput (*x*, *fadetime=0.05*)

Replace the *input* attribute.

Args

x: PyoObject New signal to process.

fadetime: float, optional Crossfade time between old and new input. Defaults to 0.05.

setBitdepth (*x*)

Replace the *bitdepth* attribute.

Args

x: float or PyoObject New *bitdepth* attribute.

setSrscale (*x*)

Replace the *srscale* attribute.

Args

x: float or PyoObject New *srscale* attribute.

ctrl (*map_list=None*, *title=None*, *wxnoserver=False*)

Opens a sliders window to control the parameters of the object. Only parameters that can be set to a PyoObject are allowed to be mapped on a slider.

If a list of values are given to a parameter, a multisliders will be used to control each stream independently.

Args

map_list: list of SLMap objects, optional Users defined set of parameters scaling. There is default scaling for each object that accept *ctrl* method.

title: string, optional Title of the window. If none is provided, the name of the class is used.

wxnoserver: boolean, optional With wxPython graphical toolkit, if True, tells the interpreter that there will be no server window.

If *wxnoserver* is set to True, the interpreter will not wait for the server GUI before showing the controller window.

input
PyoObject. Input signal to process.

bitdepth
float or PyoObject. Signal quantization in bits.

srscale
float or PyoObject. Sampling rate multiplier.

Mirror

class Mirror(*input*, *min=0.0*, *max=1.0*, *mul=1*, *add=0*)
Reflects the signal that exceeds the *min* and *max* thresholds.

This object is useful for table indexing or for clipping and modeling an audio signal.

Parent *PyoObject*

Args

input: PyoObject Input signal to process.

min: float or PyoObject, optional Minimum possible value. Defaults to 0.

max: float or PyoObject, optional Maximum possible value. Defaults to 1.

Note: If *min* is higher than *max*, then the output will be the average of the two.

```
>>> s = Server().boot()
>>> s.start()
>>> a = Sine(freq=[300,301])
>>> lfmin = Sine(freq=1.5, mul=.25, add=-0.75)
>>> lfmax = Sine(freq=2, mul=.25, add=0.75)
>>> b = Mirror(a, min=lfmin, max=lfmax)
>>> c = Tone(b, freq=2500, mul=.15).out()
```

setInput(*x*, *fadetime=0.05*)
Replace the *input* attribute.

Args

x: PyoObject New signal to process.

fadetime: float, optional Crossfade time between old and new input. Defaults to 0.05.

setMin(*x*)
Replace the *min* attribute.

Args

x: float or PyoObject New *min* attribute.

setMax(*x*)
Replace the *max* attribute.

Args

x: float or PyoObject New *max* attribute.

ctrl(*map_list=None*, *title=None*, *wxnoserver=False*)
Opens a sliders window to control the parameters of the object. Only parameters that can be set to a PyoObject are allowed to be mapped on a slider.

If a list of values are given to a parameter, a multisliders will be used to control each stream independently.

Args

map_list: list of SLMap objects, optional Users defined set of parameters scaling. There is default scaling for each object that accept *ctrl* method.

title: string, optional Title of the window. If none is provided, the name of the class is used.

wxnoserver: boolean, optional With wxPython graphical toolkit, if True, tells the interpreter that there will be no server window.

If *wxnoserver* is set to True, the interpreter will not wait for the server GUI before showing the controller window.

input
PyoObject. Input signal to process.

min
float or PyoObject. Minimum possible value.

max
float or PyoObject. Maximum possible value.

Compress

class Compress(*input*, *thresh=-20*, *ratio=2*, *risetime=0.01*, *falltime=0.1*, *lookahead=5.0*, *knee=0*, *outputAmp=False*, *mul=1*, *add=0*)
Reduces the dynamic range of an audio signal.

Compress reduces the volume of loud sounds or amplifies quiet sounds by narrowing or compressing an audio signal's dynamic range.

Parent *PyoObject*

Args

input: PyoObject Input signal to process.

thresh: float or PyoObject, optional Level, expressed in dB, above which the signal is reduced. Reference level is 0dB. Defaults to -20.

ratio: float or PyoObject, optional Determines the input/output ratio for signals above the threshold. Defaults to 2.

risetime: float or PyoObject, optional Used in amplitude follower, time to reach upward value in seconds. Defaults to 0.01.

falltime: float or PyoObject, optional Used in amplitude follower, time to reach downward value in seconds. Defaults to 0.1.

lookahead: float, optional Delay length, in ms, for the "look-ahead" buffer. Range is 0 -> 25 ms. Defaults to 5.0.

knee: float optional Shape of the transfert function around the threshold, specified in the range 0 -> 1.

A value of 0 means a hard knee and a value of 1.0 means a softer knee. Defaults to 0.

outputAmp: boolean, optional If True, the object's output signal will be the compression level alone, not the compressed signal.

It can be useful if 2 or more channels need to be linked on the same compression slope. Defaults to False.

Available at initialization only.

```
>>> s = Server().boot()
>>> s.start()
>>> a = SfPlayer(SNDS_PATH + '/transparent.aif', loop=True)
>>> b = Compress(a, thresh=-24, ratio=6, risetime=.01, falltime=.2, knee=0.5).
↪mix(2).out()
```

setInput (*x*, *fadetime=0.05*)
Replace the *input* attribute.

Args

x: PyoObject New signal to process.

fadetime: float, optional Crossfade time between old and new input. Defaults to 0.05.

setThresh (*x*)
Replace the *thresh* attribute.

Args

x: float or PyoObject New *thresh* attribute.

setRatio (*x*)
Replace the *ratio* attribute.

Args

x: float or PyoObject New *ratio* attribute.

setRiseTime (*x*)
Replace the *risetime* attribute.

Args

x: float or PyoObject New *risetime* attribute.

setFallTime (*x*)
Replace the *falltime* attribute.

Args

x: float or PyoObject New *falltime* attribute.

setLookAhead (*x*)
Replace the *lookahead* attribute.

Args

x: float New *lookahead* attribute.

setKnee (*x*)
Replace the *knee* attribute.

Args

x: float New *knee* attribute.

ctrl(*map_list=None*, *title=None*, *wxnoserver=False*)
Opens a sliders window to control the parameters of the object. Only parameters that can be set to a PyoObject are allowed to be mapped on a slider.

If a list of values are given to a parameter, a multisliders will be used to control each stream independently.

Args

map_list: list of SLMap objects, optional Users defined set of parameters scaling. There is default scaling for each object that accept *ctrl* method.

title: string, optional Title of the window. If none is provided, the name of the class is used.

wxnoserver: boolean, optional With wxPython graphical toolkit, if True, tells the interpreter that there will be no server window.

If *wxnoserver* is set to True, the interpreter will not wait for the server GUI before showing the controller window.

input
PyoObject. Input signal to process.

thresh
float or PyoObject. Level above which the signal is reduced.

ratio
float or PyoObject. in/out ratio for signals above the threshold.

risetime
float or PyoObject. Time to reach upward value in seconds.

falltime
float or PyoObject. Time to reach downward value in seconds.

lookahead
float. Delay length, in ms, of the "look-ahead" buffer.

knee
float. Shape of the transfert function around the threshold.

Gate

class Gate(*input*, *thresh=-70*, *risetime=0.01*, *falltime=0.05*, *lookahead=5.0*, *outputAmp=False*, *mul=1*, *add=0*)
Allows a signal to pass only when its amplitude is above a set threshold.

A noise gate is used when the level of the signal is below the level of the noise floor. The threshold is set above the level of the noise and so when there is no signal the gate is closed. A noise gate does not remove noise from the signal. When the gate is open both the signal and the noise will pass through.

Parent `PyoObject`

Args

input: PyoObject Input signal to process.

thresh: float or PyoObject, optional Level, expressed in dB, below which the gate is closed. Reference level is 0dB. Defaults to -70.

risetime: float or PyoObject, optional Time to open the gate in seconds. Defaults to 0.01.

falltime: float or PyoObject, optional Time to close the gate in seconds. Defaults to 0.05.

lookahead: float, optional Delay length, in ms, for the "look-ahead" buffer. Range is 0 -> 25 ms. Defaults to 5.0.

outputAmp: boolean, optional If True, the object's output signal will be the gating level alone, not the gated signal.

It can be useful if 2 or more channels need to linked on the same gating slope. Defaults to False.

Available at initialization only.

```
>>> s = Server().boot()
>>> s.start()
>>> sf = SfPlayer(SNDS_PATH + '/transparent.aif', speed=[1,.5], loop=True)
>>> gt = Gate(sf, thresh=-24, risetime=0.005, falltime=0.01, lookahead=5, mul=.4).
↪out()
```

setInput (*x*, *fadetime=0.05*)

Replace the *input* attribute.

Args

x: PyoObject New signal to process.

fadetime: float, optional Crossfade time between old and new input. Defaults to 0.05.

setThresh (*x*)

Replace the *thresh* attribute.

Args

x: float or PyoObject New *thresh* attribute.

setRiseTime (*x*)

Replace the *risetime* attribute.

Args

x: float or PyoObject New *risetime* attribute.

setFallTime (*x*)

Replace the *falltime* attribute.

Args

x: float or PyoObject New *falltime* attribute.

setLookAhead (*x*)

Replace the *lookahead* attribute.

Args

x: float New *lookahead* attribute.

ctrl (*map_list=None*, *title=None*, *wxnoserver=False*)

Opens a sliders window to control the parameters of the object. Only parameters that can be set to a PyoObject are allowed to be mapped on a slider.

If a list of values are given to a parameter, a multisliders will be used to control each stream independently.

Args

map_list: list of SLMap objects, optional Users defined set of parameters scaling. There is default scaling for each object that accept *ctrl* method.

title: string, optional Title of the window. If none is provided, the name of the class is used.

wxnoserver: boolean, optional With wxPython graphical toolkit, if True, tells the interpreter that there will be no server window.

If *wxnoserver* is set to True, the interpreter will not wait for the server GUI before showing the controller window.

input
PyoObject. Input signal to process.

thresh
float or PyoObject. Level below which the gate is closed.

risetime
float or PyoObject. Time to open the gate in seconds.

falltime
float or PyoObject. Time to close the gate in seconds.

lookahead
float. Delay length, in ms, of the "look-ahead" buffer.

Balance

class Balance(*input*, *input2*, *freq=10*, *mul=1*, *add=0*)
Adjust rms power of an audio signal according to the rms power of another.

The rms power of a signal is adjusted to match that of a comparator signal.

Parent *PyoObject*

Args

input: PyoObject Input signal to process.

input2: PyoObject Comparator signal.

freq: float or PyoObject, optional Cutoff frequency of the lowpass filter in hertz. Default to 10.

```
>>> s = Server().boot()
>>> s.start()
>>> sf = SfPlayer(SNDS_PATH + '/accord.aif', speed=[.99,1], loop=True, mul=.3)
>>> comp = SfPlayer(SNDS_PATH + '/transparent.aif', speed=[.99,1], loop=True,␣
↪mul=.3)
>>> out = Balance(sf, comp, freq=10).out()
```

setInput(*x*, *fadetime=0.05*)
Replace the *input* attribute.

Input signal to process.

Args

x: PyoObject New signal to process.

fadetime: float, optional Crossfade time between old and new input. Default to 0.05.

setInput2(*x*, *fadetime=0.05*)
Replace the *input2* attribute.

Comparator signal.

Args

x: PyoObject New signal to process.

fadetime: float, optional Crossfade time between old and new input. Default to 0.05.

setFreq(*x*)
Replace the *freq* attribute.

Cutoff frequency of the lowpass filter, in Hertz.

Args

x: float or PyoObject New *freq* attribute.

ctrl(*map_list=None*, *title=None*, *wxnoserver=False*)
Opens a sliders window to control the parameters of the object. Only parameters that can be set to a PyoObject are allowed to be mapped on a slider.

If a list of values are given to a parameter, a multisliders will be used to control each stream independently.

Args

map_list: list of SLMap objects, optional Users defined set of parameters scaling. There is default scaling for each object that accept *ctrl* method.

title: string, optional Title of the window. If none is provided, the name of the class is used.

wxnoserver: boolean, optional With wxPython graphical toolkit, if True, tells the interpreter that there will be no server window.

If *wxnoserver* is set to True, the interpreter will not wait for the server GUI before showing the controller window.

input
PyoObject. Input signal to process.

input2
PyoObject. Comparator signal.

freq
float or PyoObject. Cutoff frequency of the lowpass filter.

Min

class Min(*input*, *comp=0.5*, *mul=1*, *add=0*)
Outputs the minimum of two values.

Parent *PyoObject*

Args

input: PyoObject Input signal to process.

comp: float or PyoObject, optional Comparison value. If *input* is lower than this value, *input* is send to the output, otherwise, *comp* is outputted.

```
>>> s = Server().boot()
>>> s.start()
>>> # Triangle wave
```

(continues on next page)

(continued from previous page)

```
>>> a = Phasor([249,250])
>>> b = Min(a, comp=a*-1+1, mul=4, add=-1)
>>> c = Tone(b, freq=1500, mul=.5).out()
```

setInput(*x*, *fadetime=0.05*)
Replace the *input* attribute.

Args

x: PyoObject New signal to process.

fadetime: float, optional Crossfade time between old and new input. Default to 0.05.

setComp(*x*)
Replace the *comp* attribute.

Args

x: float or PyoObject New *comp* attribute.

ctrl(*map_list=None*, *title=None*, *wxnoserver=False*)
Opens a sliders window to control the parameters of the object. Only parameters that can be set to a PyoObject are allowed to be mapped on a slider.

If a list of values are given to a parameter, a multisliders will be used to control each stream independently.

Args

map_list: list of SLMap objects, optional Users defined set of parameters scaling. There is default scaling for each object that accept *ctrl* method.

title: string, optional Title of the window. If none is provided, the name of the class is used.

wxnoserver: boolean, optional With wxPython graphical toolkit, if True, tells the interpreter that there will be no server window.

If *wxnoserver* is set to True, the interpreter will not wait for the server GUI before showing the controller window.

input
PyoObject. Input signal to process.

comp
float or PyoObject. Comparison value.

Max

class Max(*input*, *comp=0.5*, *mul=1*, *add=0*)
Outputs the maximum of two values.

Parent *PyoObject*

Args

input: PyoObject Input signal to process.

comp: float or PyoObject, optional Comparison value. If *input* is higher than this value, *input* is send to the output, otherwise, *comp* is outputted.

```
>>> s = Server().boot()
>>> s.start()
>>> # Assimetrical clipping
>>> a = Phasor(500, mul=2, add=-1)
>>> b = Max(a, comp=-0.3)
>>> c = Tone(b, freq=1500, mul=.5).out()
```

setInput(*x*, *fadetime=0.05*)
Replace the *input* attribute.

Args

x: PyoObject New signal to process.

fadetime: float, optional Crossfade time between old and new input. Default to 0.05.

setComp(*x*)
Replace the *comp* attribute.

Args

x: float or PyoObject New *comp* attribute.

ctrl(*map_list=None*, *title=None*, *wxnoserver=False*)
Opens a sliders window to control the parameters of the object. Only parameters that can be set to a PyoObject are allowed to be mapped on a slider.

If a list of values are given to a parameter, a multisliders will be used to control each stream independently.

Args

map_list: list of SLMap objects, optional Users defined set of parameters scaling. There is default scaling for each object that accept *ctrl* method.

title: string, optional Title of the window. If none is provided, the name of the class is used.

wxnoserver: boolean, optional With wxPython graphical toolkit, if True, tells the interpreter that there will be no server window.

If *wxnoserver* is set to True, the interpreter will not wait for the server GUI before showing the controller window.

input
PyoObject. Input signal to process.

comp
float or PyoObject. Comparison value.

Wrap

class Wrap(*input*, *min=0.0*, *max=1.0*, *mul=1*, *add=0*)
Wraps-around the signal that exceeds the *min* and *max* thresholds.

This object is useful for table indexing, phase shifting or for clipping and modeling an audio signal.

Parent *PyoObject*

Args

input: PyoObject Input signal to process.

min: float or PyoObject, optional Minimum possible value. Defaults to 0.

max: float or PyoObject, optional Maximum possible value. Defaults to 1.

Note: If *min* is higher than *max*, then the output will be the average of the two.

```
>>> s = Server().boot()
>>> s.start()
>>> # Time-varying overlaping envelopes
>>> env = HannTable()
>>> lff = Sine(.5, mul=3, add=4)
>>> ph1 = Phasor(lff)
>>> ph2 = Wrap(ph1+0.5, min=0, max=1)
>>> amp1 = Pointer(env, ph1, mul=.25)
>>> amp2 = Pointer(env, ph2, mul=.25)
>>> a = SineLoop(250, feedback=.1, mul=amp1).out()
>>> b = SineLoop(300, feedback=.1, mul=amp2).out(1)
```

setInput(*x*, *fadetime=0.05*)

Replace the *input* attribute.

Args

x: PyoObject New signal to process.

fadetime: float, optional Crossfade time between old and new input. Defaults to 0.05.

setMin(*x*)

Replace the *min* attribute.

Args

x: float or PyoObject New *min* attribute.

setMax(*x*)

Replace the *max* attribute.

Args

x: float or PyoObject New *max* attribute.

ctrl(*map_list=None*, *title=None*, *wxnoserver=False*)

Opens a sliders window to control the parameters of the object. Only parameters that can be set to a PyoObject are allowed to be mapped on a slider.

If a list of values are given to a parameter, a multisliders will be used to control each stream independently.

Args

map_list: list of SLMap objects, optional Users defined set of parameters scaling. There is default scaling for each object that accept *ctrl* method.

title: string, optional Title of the window. If none is provided, the name of the class is used.

wxnoserver: boolean, optional With wxPython graphical toolkit, if True, tells the interpreter that there will be no server window.

If *wxnoserver* is set to True, the interpreter will not wait for the server GUI before showing the controller window.

input

PyoObject. Input signal to process.

min
float or PyoObject. Minimum possible value.

max
float or PyoObject. Maximum possible value.

Expand

class Expand(*input*, *downthresh=-40*, *upthresh=-10*, *ratio=2*, *risetime=0.01*, *falltime=0.1*, *lookahead=5.0*, *outputAmp=False*, *mul=1*, *add=0*)
Expand the dynamic range of an audio signal.

The Expand object will boost the volume of the input sound if it rises above the upper threshold. It will also reduce the volume of the input sound if it falls below the lower threshold. This process will "expand" the audio signal's dynamic range.

Parent *PyoObject*

Args

input: PyoObject Input signal to process.

downthresh: float or PyoObject, optional Level, expressed in dB, below which the signal is getting softer, according to the *ratio*. Reference level is 0dB. Defaults to -20.

upthresh: float or PyoObject, optional Level, expressed in dB, above which the signal is getting louder, according to the same *ratio* as the lower threshold. Reference level is 0dB. Defaults to -20.

ratio: float or PyoObject, optional The *ratio* argument controls is the amount of expansion (with a ratio of 4, if there is a rise of 2 dB above the upper threshold, the output signal will rises by 8 dB), and contrary for the lower threshold. Defaults to 2.

risetime: float or PyoObject, optional Used in amplitude follower, time to reach upward value in seconds. Defaults to 0.01.

falltime: float or PyoObject, optional Used in amplitude follower, time to reach downward value in seconds. Defaults to 0.1.

lookahead: float, optional Delay length, in ms, for the "look-ahead" buffer. Range is 0 -> 25 ms. Defaults to 5.0.

outputAmp: boolean, optional If True, the object's output signal will be the expansion level alone, not the expanded signal.

It can be useful if 2 or more channels need to be linked on the same expansion slope. Defaults to False.

Available at initialization only.

```
>>> s = Server().boot()
>>> s.start()
>>> # original to the left channel
>>> sf = SfPlayer(SNDS_PATH + "/transparent.aif", loop=True, mul=0.7)
>>> ori = Delay(sf, 0.005).out()
>>> # expanded to the right channel
>>> ex = Expand(sf, downthresh=-20, upthresh=-20, ratio=4, mul=0.5).out(1)
```

setInput(*x*, *fadetime=0.05*)
Replace the *input* attribute.

Args

x: PyoObject New signal to process.

fadetime: float, optional Crossfade time between old and new input. Defaults to 0.05.

setDownThresh(*x*)
Replace the *downthresh* attribute.

Args

x: float or PyoObject New *downthresh* attribute.

setUpThresh(*x*)
Replace the *upthresh* attribute.

Args

x: float or PyoObject New *upthresh* attribute.

setRatio(*x*)
Replace the *ratio* attribute.

Args

x: float or PyoObject New *ratio* attribute.

setRiseTime(*x*)
Replace the *risetime* attribute.

Args

x: float or PyoObject New *risetime* attribute.

setFallTime(*x*)
Replace the *falltime* attribute.

Args

x: float or PyoObject New *falltime* attribute.

setLookAhead(*x*)
Replace the *lookahead* attribute.

Args

x: float New *lookahead* attribute.

ctrl(*map_list=None*, *title=None*, *wxnoserver=False*)
Opens a sliders window to control the parameters of the object. Only parameters that can be set to a PyoObject are allowed to be mapped on a slider.

If a list of values are given to a parameter, a multisliders will be used to control each stream independently.

Args

map_list: list of SLMap objects, optional Users defined set of parameters scaling. There is default scaling for each object that accept *ctrl* method.

title: string, optional Title of the window. If none is provided, the name of the class is used.

wxnoserver: boolean, optional With wxPython graphical toolkit, if True, tells the interpreter that there will be no server window.

If *wxnoserver* is set to True, the interpreter will not wait for the server GUI before showing the controller window.

input
PyoObject. Input signal to process.

downthresh
float or PyoObject. Level below which the signal is reduced.

upthresh
float or PyoObject. Level above which the signal is boosted.

ratio
float or PyoObject. in/out ratio for signals outside thresholds.

risetime
float or PyoObject. Time to reach upward value in seconds.

falltime
float or PyoObject. Time to reach downward value in seconds.

lookahead
float. Delay length, in ms, of the "look-ahead" buffer.

Special Effects

Objects to perform specific audio signal processing effects such as distortions, delays, chorus and reverbs.

Disto

class Disto(*input*, *drive=0.75*, *slope=0.5*, *mul=1*, *add=0*)
Kind of Arc tangent distortion.

Apply a kind of arc tangent distortion with controllable drive, followed by a one pole lowpass filter, to the input signal.

As of version 0.8.0, this object use a simple but very efficient (4x faster than tanh or atan2 functions) waveshaper formula describe here:

http://musicdsp.org/archive.php?classid=4#46

The waveshaper algorithm is:

y[n] = (1 + k) * x[n] / (1 + k * abs(x[n]))

where:

k = (2 * drive) / (1 - drive)

Parent *PyoObject*

Args

input: PyoObject Input signal to process.

drive: float or PyoObject, optional Amount of distortion applied to the signal, between 0 and 1. Defaults to 0.75.

slope: float or PyoObject, optional Slope of the lowpass filter applied after distortion, between 0 and 1. Defaults to 0.5.

```
>>> s = Server().boot()
>>> s.start()
>>> a = SfPlayer(SNDS_PATH + "/transparent.aif", loop=True)
>>> lfo = Sine(freq=[.2,.25], mul=.5, add=.5)
>>> d = Disto(a, drive=lfo, slope=.8, mul=.15).out()
```

setInput(*x*, *fadetime=0.05*)
Replace the *input* attribute.

Args

x: PyoObject New signal to process.

fadetime: float, optional Crossfade time between old and new input. Defaults to 0.05.

setDrive(*x*)
Replace the *drive* attribute.

Args

x: float or PyoObject New *drive* attribute.

setSlope(*x*)
Replace the *slope* attribute.

Args

x: float or PyoObject New *slope* attribute.

ctrl(*map_list=None*, *title=None*, *wxnoserver=False*)
Opens a sliders window to control the parameters of the object. Only parameters that can be set to a PyoObject are allowed to be mapped on a slider.

If a list of values are given to a parameter, a multisliders will be used to control each stream independently.

Args

map_list: list of SLMap objects, optional Users defined set of parameters scaling. There is default scaling for each object that accept *ctrl* method.

title: string, optional Title of the window. If none is provided, the name of the class is used.

wxnoserver: boolean, optional With wxPython graphical toolkit, if True, tells the interpreter that there will be no server window.

If *wxnoserver* is set to True, the interpreter will not wait for the server GUI before showing the controller window.

input
PyoObject. Input signal to process.

drive
float or PyoObject. Amount of distortion.

slope
float or PyoObject. Slope of the lowpass filter.

Delay

class Delay(*input*, *delay=0.25*, *feedback=0*, *maxdelay=1*, *mul=1*, *add=0*)
Sweepable recursive delay.

Parent *PyoObject*

Args

input: PyoObject Input signal to delayed.

delay: float or PyoObject, optional Delay time in seconds. Defaults to 0.25.

feedback: float or PyoObject, optional Amount of output signal sent back into the delay line. Defaults to 0.

maxdelay: float, optional Maximum delay length in seconds. Available only at initialization. Defaults to 1.

Note: The minimum delay time allowed with Delay is one sample. It can be computed with :

onesamp = 1.0 / s.getSamplingRate()

See also:

SDelay, *Waveguide*

```
>>> s = Server().boot()
>>> s.start()
>>> a = SfPlayer(SNDS_PATH + "/transparent.aif", loop=True, mul=.3).mix(2).out()
>>> d = Delay(a, delay=[.15,.2], feedback=.5, mul=.4).out()
```

setInput(*x*, *fadetime=0.05*)
Replace the *input* attribute.

Args

x: PyoObject New signal to delayed.

fadetime: float, optional Crossfade time between old and new input. Defaults to 0.05.

setDelay(*x*)
Replace the *delay* attribute.

Args

x: float or PyoObject New *delay* attribute.

setFeedback(*x*)
Replace the *feedback* attribute.

Args

x: float or PyoObject New *feedback* attribute.

reset()
Reset the memory buffer to zeros.

ctrl(*map_list=None*, *title=None*, *wxnoserver=False*)
Opens a sliders window to control the parameters of the object. Only parameters that can be set to a PyoObject are allowed to be mapped on a slider.

If a list of values are given to a parameter, a multisliders will be used to control each stream independently.

Args

map_list: list of SLMap objects, optional Users defined set of parameters scaling. There is default scaling for each object that accept *ctrl* method.

title: string, optional Title of the window. If none is provided, the name of the class is used.

wxnoserver: boolean, optional With wxPython graphical toolkit, if True, tells the interpreter that there will be no server window.

If *wxnoserver* is set to True, the interpreter will not wait for the server GUI before showing the controller window.

input
PyoObject. Input signal to delayed.

delay
float or PyoObject. Delay time in seconds.

feedback
float or PyoObject. Amount of output signal sent back into the delay line.

SDelay

class SDelay(*input*, *delay=0.25*, *maxdelay=1*, *mul=1*, *add=0*)
Simple delay without interpolation.

Parent *PyoObject*

Args

input: PyoObject Input signal to delayed.

delay: float or PyoObject, optional Delay time in seconds. Defaults to 0.25.

maxdelay: float, optional Maximum delay length in seconds. Available only at initialization. Defaults to 1.

See also:

Delay, *Delay1*

```
>>> s = Server().boot()
>>> s.start()
>>> srPeriod = 1. / s.getSamplingRate()
>>> dlys = [srPeriod * i * 5 for i in range(1, 7)]
>>> a = SfPlayer(SNDS_PATH + "/transparent.aif", loop=True)
>>> d = SDelay(a, delay=dlys, mul=.1).out(1)
```

setInput(*x*, *fadetime=0.05*)
Replace the *input* attribute.

Args

x: PyoObject New signal to process.

fadetime: float, optional Crossfade time between old and new input. Defaults to 0.05.

setDelay(*x*)
Replace the *delay* attribute.

Args

x: float or PyoObject New *delay* attribute.

reset()
Reset the memory buffer to zeros.

ctrl(*map_list=None*, *title=None*, *wxnoserver=False*)
Opens a sliders window to control the parameters of the object. Only parameters that can be set to a PyoObject are allowed to be mapped on a slider.

If a list of values are given to a parameter, a multisliders will be used to control each stream independently.

Args

map_list: list of SLMap objects, optional Users defined set of parameters scaling. There is default scaling for each object that accept *ctrl* method.

title: string, optional Title of the window. If none is provided, the name of the class is used.

wxnoserver: boolean, optional With wxPython graphical toolkit, if True, tells the interpreter that there will be no server window.

If *wxnoserver* is set to True, the interpreter will not wait for the server GUI before showing the controller window.

input
PyoObject. Input signal to delayed.

delay
float or PyoObject. Delay time in seconds.

Delay1

class Delay1(*input*, *mul=1*, *add=0*)
Delays a signal by one sample.

Parent *PyoObject*

Args

input: PyoObject Input signal to process.

```
>>> s = Server().boot()
>>> s.start()
>>> # 50th order FIR lowpass filter
>>> order = 50
>>> objs = [Noise(.3)]
>>> for i in range(order):
...     objs.append(Delay1(objs[-1], add=objs[-1]))
...     objs.append(objs[-1] * 0.5)
>>> out = Sig(objs[-1]).out()
```

setInput(*x*, *fadetime=0.05*)
Replace the *input* attribute.

Args

x: PyoObject New signal to process.

fadetime: float, optional Crossfade time between old and new input. Default to 0.05.

input
PyoObject. Input signal to delayed.

Waveguide

class Waveguide(*input*, *freq=100*, *dur=10*, *minfreq=20*, *mul=1*, *add=0*)

Basic waveguide model.

This waveguide model consisting of one delay-line with a simple lowpass filtering and lagrange interpolation.

Parent *PyoObject*

Args

input: PyoObject Input signal to process.

freq: float or PyoObject, optional Frequency, in cycle per second, of the waveguide (i.e. the inverse of delay time). Defaults to 100.

dur: float or PyoObject, optional Duration, in seconds, for the waveguide to drop 40 dB below it's maxima. Defaults to 10.

minfreq: float, optional Minimum possible frequency, used to initialized delay length. Available only at initialization. Defaults to 20.

```
>>> s = Server().boot()
>>> s.start()
>>> sf = SfPlayer(SNDS_PATH + '/transparent.aif', speed=[.98,1.02], loop=True)
>>> gt = Gate(sf, thresh=-24, risetime=0.005, falltime=0.01, lookahead=5, mul=.2)
>>> w = Waveguide(gt, freq=[60,120.17,180.31,240.53], dur=20, minfreq=20, mul=.4).
↪out()
```

setInput(*x*, *fadetime=0.05*)

Replace the *input* attribute.

Args

x: PyoObject New signal to process.

fadetime: float, optional Crossfade time between old and new input. Defaults to 0.05.

setFreq(*x*)

Replace the *freq* attribute.

Args

x: float or PyoObject New *freq* attribute.

setDur(*x*)

Replace the *dur* attribute.

Args

x: float or PyoObject New *dur* attribute.

reset()

Reset the memory buffer to zeros.

ctrl(*map_list=None*, *title=None*, *wxnoserver=False*)

Opens a sliders window to control the parameters of the object. Only parameters that can be set to a PyoObject are allowed to be mapped on a slider.

If a list of values are given to a parameter, a multisliders will be used to control each stream independently.

Args

map_list: list of SLMap objects, optional Users defined set of parameters scaling. There is default scaling for each object that accept *ctrl* method.

title: string, optional Title of the window. If none is provided, the name of the class is used.

wxnoserver: boolean, optional With wxPython graphical toolkit, if True, tells the interpreter that there will be no server window.

If *wxnoserver* is set to True, the interpreter will not wait for the server GUI before showing the controller window.

input
PyoObject. Input signal to process.

freq
float or PyoObject. Frequency in cycle per second.

dur
float or PyoObject. Resonance duration in seconds.

AllpassWG

class AllpassWG(*input*, *freq=100*, *feed=0.95*, *detune=0.5*, *minfreq=20*, *mul=1*, *add=0*)
Out of tune waveguide model with a recursive allpass network.

This waveguide model consisting of one delay-line with a 3-stages recursive allpass filter which made the resonances of the waveguide out of tune.

Parent *PyoObject*

Args

input: PyoObject Input signal to process.

freq: float or PyoObject, optional Frequency, in cycle per second, of the waveguide (i.e. the inverse of delay time). Defaults to 100.

feed: float or PyoObject, optional Amount of output signal (between 0 and 1) sent back into the delay line. Defaults to 0.95.

detune: float or PyoObject, optional Control the depth of the allpass delay-line filter, i.e. the depth of the detuning. Should be in the range 0 to 1. Defaults to 0.5.

minfreq: float, optional Minimum possible frequency, used to initialized delay length. Available only at initialization. Defaults to 20.

```
>>> s = Server().boot()
>>> s.start()
>>> sf = SfPlayer(SNDS_PATH + '/transparent.aif', speed=[.98,1.02], loop=True)
>>> gt = Gate(sf, thresh=-24, risetime=0.005, falltime=0.01, lookahead=5, mul=.2)
>>> rnd = Randi(min=.5, max=1.0, freq=[.13,.22,.155,.171])
>>> rnd2 = Randi(min=.95, max=1.05, freq=[.145,.2002,.1055,.071])
>>> fx = AllpassWG(gt, freq=rnd2*[74.87,75,75.07,75.21], feed=1, detune=rnd, mul=.
↪15).out()
```

setInput(*x*, *fadetime=0.05*)
Replace the *input* attribute.

Args

x: PyoObject New signal to process.

fadetime: float, optional Crossfade time between old and new input. Defaults to 0.05.

setFreq(*x*)
Replace the *freq* attribute.

Args

x: float or PyoObject New *freq* attribute.

setFeed(*x*)
Replace the *feed* attribute.

Args

x: float or PyoObject New *feed* attribute.

setDetune(*x*)
Replace the *detune* attribute.

Args

x: float or PyoObject New *detune* attribute.

reset()
Reset the memory buffer to zeros.

ctrl(*map_list=None*, *title=None*, *wxnoserver=False*)
Opens a sliders window to control the parameters of the object. Only parameters that can be set to a PyoObject are allowed to be mapped on a slider.

If a list of values are given to a parameter, a multisliders will be used to control each stream independently.

Args

map_list: list of SLMap objects, optional Users defined set of parameters scaling. There is default scaling for each object that accept *ctrl* method.

title: string, optional Title of the window. If none is provided, the name of the class is used.

wxnoserver: boolean, optional With wxPython graphical toolkit, if True, tells the interpreter that there will be no server window.

If *wxnoserver* is set to True, the interpreter will not wait for the server GUI before showing the controller window.

input
PyoObject. Input signal to process.

freq
float or PyoObject. Frequency in cycle per second.

feed
float or PyoObject. Amount of output signal sent back into the delay line.

detune
float or PyoObject. Depth of the detuning.

Freeverb

class Freeverb(*input*, *size=0.5*, *damp=0.5*, *bal=0.5*, *mul=1*, *add=0*)
Implementation of Jezar's Freeverb.

Freeverb is a reverb unit generator based on Jezar's public domain C++ sources, composed of eight parallel comb filters, followed by four allpass units in series. Filters on each stream are slightly detuned in order to create multi-channel effects.

Parent *PyoObject*

Args

input: PyoObject Input signal to process.

size: float or PyoObject, optional Controls the length of the reverb, between 0 and 1. A higher value means longer reverb. Defaults to 0.5.

damp: float or PyoObject, optional High frequency attenuation, between 0 and 1. A higher value will result in a faster decay of the high frequency range. Defaults to 0.5.

bal: float or PyoObject, optional Balance between wet and dry signal, between 0 and 1. 0 means no reverb. Defaults to 0.5.

```
>>> s = Server().boot()
>>> s.start()
>>> a = SfPlayer(SNDS_PATH + "/transparent.aif", loop=True, mul=.4)
>>> b = Freeverb(a, size=[.79,.8], damp=.9, bal=.3).out()
```

setInput(*x*, *fadetime=0.05*)
Replace the *input* attribute.

Args

x: PyoObject New signal to process.

fadetime: float, optional Crossfade time between old and new input. Defaults to 0.05.

setSize(*x*)
Replace the *size* attribute.

Args

x: float or PyoObject New *size* attribute.

setDamp(*x*)
Replace the *damp* attribute.

Args

x: float or PyoObject New *damp* attribute.

setBal(*x*)
Replace the *bal* attribute.

Args

x: float or PyoObject New *bal* attribute.

reset()
Reset the memory buffer to zeros.

ctrl(*map_list=None*, *title=None*, *wxnoserver=False*)
Opens a sliders window to control the parameters of the object. Only parameters that can be set to a PyoObject are allowed to be mapped on a slider.

If a list of values are given to a parameter, a multisliders will be used to control each stream independently.

Args

map_list: list of SLMap objects, optional Users defined set of parameters scaling. There is default scaling for each object that accept *ctrl* method.

title: string, optional Title of the window. If none is provided, the name of the class is used.

wxnoserver: boolean, optional With wxPython graphical toolkit, if True, tells the interpreter that there will be no server window.

If *wxnoserver* is set to True, the interpreter will not wait for the server GUI before showing the controller window.

input
PyoObject. Input signal to process.

size
float or PyoObject. Room size.

damp
float or PyoObject. High frequency damping.

bal
float or PyoObject. Balance between wet and dry signal.

Convolve

class Convolve(*input*, *table*, *size*, *mul=1*, *add=0*)
Implements filtering using circular convolution.

A circular convolution is defined as the integral of the product of two functions after one is reversed and shifted.

Parent *PyoObject*

Args

input: PyoObject Input signal to process.

table: PyoTableObject Table containning the impulse response.

size: int Length, in samples, of the convolution. Available at initialization time only.

If the table changes during the performance, its size must egal or greater than this value.

If greater only the first *size* samples will be used.

Note: Convolution is very expensive to compute, so the impulse response must be kept very short to run in real time.

Usually convolution generates a high amplitude level, take care of the *mul* parameter!

See also:

Follower

```
>>> s = Server().boot()
>>> s.start()
>>> snd = SNDS_PATH + '/transparent.aif'
>>> sf = SfPlayer(snd, speed=[.999,1], loop=True, mul=.25).out()
>>> a = Convolve(sf, SndTable(SNDS_PATH+'/accord.aif'), size=512, mul=.2).out()
```

setInput(*x*, *fadetime=0.05*)
Replace the *input* attribute.

Args

x: PyoObject New signal to process.

fadetime: float, optional Crossfade time between old and new input. Default to 0.05.

setTable(*x*)
Replace the *table* attribute.

Args

x: PyoTableObject new *table* attribute.

input
PyoObject. Input signal to filter.

table
PyoTableObject. Table containing the impulse response.

WGVerb

class WGVerb(*input*, *feedback=0.5*, *cutoff=5000*, *bal=0.5*, *mul=1*, *add=0*)
8 delay lines mono FDN reverb.

8 delay lines FDN reverb, with feedback matrix based upon physical modeling scattering junction of 8 lossless waveguides of equal characteristic impedance.

Parent *PyoObject*

Args

input: PyoObject Input signal to process.

feedback: float or PyoObject, optional Amount of output signal sent back into the delay lines. Defaults to 0.5.

0.6 gives a good small "live" room sound, 0.8 a small hall, and 0.9 a large hall.

cutoff: float or PyoObject, optional cutoff frequency of simple first order lowpass filters in the feedback loop of delay lines, in Hz. Defaults to 5000.

bal: float or PyoObject, optional Balance between wet and dry signal, between 0 and 1. 0 means no reverb. Defaults to 0.5.

```
>>> s = Server().boot()
>>> s.start()
>>> a = SfPlayer(SNDS_PATH + "/transparent.aif", loop=True)
>>> d = WGVerb(a, feedback=[.74,.75], cutoff=5000, bal=.25, mul=.3).out()
```

setInput(*x*, *fadetime=0.05*)
Replace the *input* attribute.

Args

x: PyoObject New signal to process.

fadetime: float, optional Crossfade time between old and new input. Defaults to 0.05.

setFeedback(*x*)
Replace the *feedback* attribute.

Args

x: float or PyoObject New *feedback* attribute.

setCutoff(*x*)
Replace the *cutoff* attribute.

Args

x: float or PyoObject New *cutoff* attribute.

setBal(*x*)
Replace the *bal* attribute.

Args

x: float or PyoObject New *bal* attribute.

reset()
Reset the memory buffer to zeros.

ctrl(*map_list=None*, *title=None*, *wxnoserver=False*)
Opens a sliders window to control the parameters of the object. Only parameters that can be set to a PyoObject are allowed to be mapped on a slider.

If a list of values are given to a parameter, a multisliders will be used to control each stream independently.

Args

map_list: list of SLMap objects, optional Users defined set of parameters scaling. There is default scaling for each object that accept *ctrl* method.

title: string, optional Title of the window. If none is provided, the name of the class is used.

wxnoserver: boolean, optional With wxPython graphical toolkit, if True, tells the interpreter that there will be no server window.

If *wxnoserver* is set to True, the interpreter will not wait for the server GUI before showing the controller window.

input
PyoObject. Input signal to process.

feedback
float or PyoObject. Amount of output signal sent back into the delay lines.

cutoff
float or PyoObject. Lowpass filter cutoff in Hz.

bal
float or PyoObject. wet - dry balance.

Chorus

class Chorus(*input*, *depth=1*, *feedback=0.25*, *bal=0.5*, *mul=1*, *add=0*)
8 modulated delay lines chorus processor.

A chorus effect occurs when individual sounds with roughly the same timbre and nearly (but never exactly) the same pitch converge and are perceived as one.

Parent *PyoObject*

Args

input: PyoObject Input signal to process.

depth: float or PyoObject, optional Chorus depth, between 0 and 5. Defaults to 1.

feedback: float or PyoObject, optional Amount of output signal sent back into the delay lines. Defaults to 0.25.

bal: float or PyoObject, optional Balance between wet and dry signals, between 0 and 1. 0 means no chorus. Defaults to 0.5.

```
>>> s = Server().boot()
>>> s.start()
>>> sf = SfPlayer(SNDS_PATH + '/transparent.aif', loop=True, mul=.5)
>>> chor = Chorus(sf, depth=[1.5,1.6], feedback=0.5, bal=0.5).out()
```

setInput(*x*, *fadetime=0.05*)
Replace the *input* attribute.

Args

x: PyoObject New signal to process.

fadetime: float, optional Crossfade time between old and new input. Defaults to 0.05.

setDepth(*x*)
Replace the *depth* attribute.

Args

x: float or PyoObject New *depth* attribute.

setFeedback(*x*)
Replace the *feedback* attribute.

Args

x: float or PyoObject New *feedback* attribute.

setBal(*x*)
Replace the *bal* attribute.

Args

x: float or PyoObject New *bal* attribute.

reset()
Reset the memory buffer to zeros.

ctrl(*map_list=None*, *title=None*, *wxnoserver=False*)
Opens a sliders window to control the parameters of the object. Only parameters that can be set to a PyoObject are allowed to be mapped on a slider.

If a list of values are given to a parameter, a multisliders will be used to control each stream independently.

Args

map_list: list of SLMap objects, optional Users defined set of parameters scaling. There is default scaling for each object that accept *ctrl* method.

title: string, optional Title of the window. If none is provided, the name of the class is used.

wxnoserver: boolean, optional With wxPython graphical toolkit, if True, tells the interpreter that there will be no server window.

If *wxnoserver* is set to True, the interpreter will not wait for the server GUI before showing the controller window.

input
PyoObject. Input signal to process.

depth
float or PyoObject. Chorus depth, between 0 and 5.

feedback
float or PyoObject. Amount of output signal sent back into the delay lines.

bal
float or PyoObject. wet - dry balance.

Harmonizer

class Harmonizer(*input*, *transpo=-7.0*, *feedback=0*, *winsize=0.1*, *mul=1*, *add=0*)
Generates harmonizing voices in synchrony with its audio input.

Parent *PyoObject*

Args

input: PyoObject Input signal to process.

transpo: float or PyoObject, optional Transposition factor in semitone. Defaults to -7.0.

feedback: float or PyoObject, optional Amount of output signal sent back into the delay line. Defaults to 0.

winsize: float, optional Window size in seconds (max = 1.0). Defaults to 0.1.

```
>>> s = Server().boot()
>>> s.start()
>>> sf = SfPlayer(SNDS_PATH + '/transparent.aif', loop=True, mul=.3).out()
>>> harm = Harmonizer(sf, transpo=-5, winsize=0.05).out(1)
```

setInput(*x*, *fadetime=0.05*)
Replace the *input* attribute.

Args

x: PyoObject New signal to process.

fadetime: float, optional Crossfade time between old and new input. Defaults to 0.05.

setTranspo(*x*)
Replace the *transpo* attribute.

Args

x: float or PyoObject New *transpo* attribute.

setFeedback(*x*)
Replace the *feedback* attribute.

Args

x: float or PyoObject New *feedback* attribute.

setWinsize(*x*)
Replace the *winsize* attribute.

Args

x: float New *winsize* attribute.

reset()
Reset the memory buffer to zeros.

ctrl(*map_list=None*, *title=None*, *wxnoserver=False*)
Opens a sliders window to control the parameters of the object. Only parameters that can be set to a PyoObject are allowed to be mapped on a slider.

If a list of values are given to a parameter, a multisliders will be used to control each stream independently.

Args

map_list: list of SLMap objects, optional Users defined set of parameters scaling. There is default scaling for each object that accept *ctrl* method.

title: string, optional Title of the window. If none is provided, the name of the class is used.

wxnoserver: boolean, optional With wxPython graphical toolkit, if True, tells the interpreter that there will be no server window.

If *wxnoserver* is set to True, the interpreter will not wait for the server GUI before showing the controller window.

input
PyoObject. Input signal to delayed.

transpo
float or PyoObject. Transposition factor in semitone.

feedback
float or PyoObject. Amount of output signal sent back into the delay line.

winsize
float. Window size in seconds (max = 1.0).

FreqShift

class FreqShift(*input*, *shift=100*, *mul=1*, *add=0*)
Frequency shifting using single sideband amplitude modulation.

Shifting frequencies means that the input signal can be detuned, where the harmonic components of the signal are shifted out of harmonic alignment with each other, e.g. a signal with harmonics at 100, 200, 300, 400 and 500 Hz, shifted up by 50 Hz, will have harmonics at 150, 250, 350, 450, and 550 Hz.

Parent *PyoObject*

Args

input: PyoObject Input signal to process.

shift: float or PyoObject, optional Amount of shifting in Hertz. Defaults to 100.

```
>>> s = Server().boot()
>>> s.start()
>>> a = SineLoop(freq=300, feedback=.1, mul=.3)
>>> lf1 = Sine(freq=.04, mul=10)
>>> lf2 = Sine(freq=.05, mul=10)
```

(continues on next page)

(continued from previous page)

```
>>> b = FreqShift(a, shift=lf1, mul=.5).out()
>>> c = FreqShift(a, shift=lf2, mul=.5).out(1)
```

play(*dur=0*, *delay=0*)

Start processing without sending samples to output. This method is called automatically at the object creation.

This method returns *self*, allowing it to be applied at the object creation.

Args

dur: float, optional Duration, in seconds, of the object's activation. The default is 0 and means infinite duration.

delay: float, optional Delay, in seconds, before the object's activation. Defaults to 0.

stop(*wait=0*)

Stop processing.

This method returns *self*, allowing it to be applied at the object creation.

Args

wait: float, optional Delay, in seconds, before the process is actually stopped. If autoStartChildren is activated in the Server, this value is propagated to the children objects. Defaults to 0.

Note: if the method setStopDelay(x) was called before calling stop(wait) with a positive *wait* value, the *wait* value won't overwrite the value given to setStopDelay for the current object, but will be the one propagated to children objects. This allow to set a waiting time for a specific object with setStopDelay whithout changing the global delay time given to the stop method.

Fader and Adsr objects ignore waiting time given to the stop method because they already implement a delayed processing triggered by the stop call.

---

out(*chnl=0*, *inc=1*, *dur=0*, *delay=0*)

Start processing and send samples to audio output beginning at *chnl*.

This method returns *self*, allowing it to be applied at the object creation.

Args

chnl: int, optional Physical output assigned to the first audio stream of the object. Defaults to 0.

inc: int, optional Output channel increment value. Defaults to 1.

dur: float, optional Duration, in seconds, of the object's activation. The default is 0 and means infinite duration.

delay: float, optional Delay, in seconds, before the object's activation. Defaults to 0.

If *chnl* >= 0, successive streams increment the output number by *inc* and wrap around the global number of channels.

If *chnl* is negative, streams begin at 0, increment the output number by *inc* and wrap around the global number of channels. Then, the list of streams is scrambled.

If *chnl* is a list, successive values in the list will be assigned to successive streams.

setInput(*x*, *fadetime=0.05*)
Replace the *input* attribute.

Parameters:

x: PyoObject New signal to process.

fadetime: float, optional Crossfade time between old and new input. Defaults to 0.05.

setShift(*x*)
Replace the *shift* attribute.

Parameters:

x: float or PyoObject New *shift* attribute.

ctrl(*map_list=None*, *title=None*, *wxnoserver=False*)
Opens a sliders window to control the parameters of the object. Only parameters that can be set to a PyoObject are allowed to be mapped on a slider.

If a list of values are given to a parameter, a multisliders will be used to control each stream independently.

Args

map_list: list of SLMap objects, optional Users defined set of parameters scaling. There is default scaling for each object that accept *ctrl* method.

title: string, optional Title of the window. If none is provided, the name of the class is used.

wxnoserver: boolean, optional With wxPython graphical toolkit, if True, tells the interpreter that there will be no server window.

If *wxnoserver* is set to True, the interpreter will not wait for the server GUI before showing the controller window.

input
PyoObject. Input signal to pitch shift.

shift
float or PyoObject. Amount of pitch shift in Hertz.

STRev

class STRev(*input*, *inpos=0.5*, *revtime=1*, *cutoff=5000*, *bal=0.5*, *roomSize=1*, *firstRefGain=-3*, *mul=1*, *add=0*)
Stereo reverb.

Stereo reverb based on WGVerb (8 delay line FDN reverb). A mono input will produce two audio streams, left and right channels. Therefore, a stereo input will produce four audio streams, left and right channels for each input channel. Position of input streams can be set with the *inpos* argument. To achieve a stereo reverb, delay line lengths are slightly differents on both channels, but also, pre-delays length and filter cutoff of both channels will be affected to reflect the input position.

Parent `PyoObject`

Args

input: PyoObject Input signal to process.

inpos: float or PyoObject, optional Position of the source, between 0 and 1. 0 means fully left and 1 means fully right. Defaults to 0.5.

revtime: float or PyoObject, optional Duration, in seconds, of the reverberated sound, defined as the time needed to the sound to drop 40 dB below its peak. Defaults to 1.

cutoff: float or PyoObject, optional cutoff frequency, in Hz, of a first order lowpass filters in the feedback loop of delay lines. Defaults to 5000.

bal: float or PyoObject, optional Balance between wet and dry signal, between 0 and 1. 0 means no reverb. Defaults to 0.5.

roomSize: float, optional Delay line length scaler, between 0.25 and 4. Values higher than 1 make the delay lines longer and simulate larger rooms. Defaults to 1.

firstRefGain: float, optional Gain, in dB, of the first reflexions of the room. Defaults to -3.

```
>>> s = Server().boot()
>>> s.start()
>>> t = SndTable(SNDS_PATH + "/transparent.aif")
>>> sf = Looper(t, dur=t.getDur()*2, xfade=0, mul=0.5)
>>> rev = STRev(sf, inpos=0.25, revtime=2, cutoff=5000, bal=0.25, roomSize=1).
↪out()
```

setInput (*x*, *fadetime=0.05*)
Replace the *input* attribute.

Args

x: PyoObject New signal to process.

fadetime: float, optional Crossfade time between old and new input. Defaults to 0.05.

setInpos (*x*)
Replace the *inpos* attribute.

Args

x: float or PyoObject New *inpos* attribute.

setRevtime (*x*)
Replace the *revtime* attribute.

Args

x: float or PyoObject New *revtime* attribute.

setCutoff (*x*)
Replace the *cutoff* attribute.

Args

x: float or PyoObject New *cutoff* attribute.

setBal (*x*)
Replace the *bal* attribute.

Args

x: float or PyoObject New *bal* attribute.

setRoomSize (*x*)
Set the room size scaler, between 0.25 and 4.

Args

x: float Room size scaler, between 0.25 and 4.0.

setFirstRefGain(*x*)
Set the gain of the first reflexions.

Args

x: float Gain, in dB, of the first reflexions.

reset()
Reset the memory buffer to zeros.

ctrl(*map_list=None*, *title=None*, *wxnoserver=False*)
Opens a sliders window to control the parameters of the object. Only parameters that can be set to a PyoObject are allowed to be mapped on a slider.

If a list of values are given to a parameter, a multisliders will be used to control each stream independently.

Args

map_list: list of SLMap objects, optional Users defined set of parameters scaling. There is default scaling for each object that accept *ctrl* method.

title: string, optional Title of the window. If none is provided, the name of the class is used.

wxnoserver: boolean, optional With wxPython graphical toolkit, if True, tells the interpreter that there will be no server window.

If *wxnoserver* is set to True, the interpreter will not wait for the server GUI before showing the controller window.

input
PyoObject. Input signal to process.

inpos
float or PyoObject. Position of the source.

revtime
float or PyoObject. Room size.

cutoff
float or PyoObject. High frequency damping.

bal
float or PyoObject. Balance between wet and dry signal.

roomSize
float. Room size scaler, between 0.25 and 4.0.

firstRefGain
float. Gain, in dB, of the first reflexions.

SmoothDelay

class SmoothDelay(*input*, *delay=0.25*, *feedback=0*, *crossfade=0.05*, *maxdelay=1*, *mul=1*, *add=0*)
Artifact free sweepable recursive delay.

SmoothDelay implements a delay line that does not produce clicks or pitch shifting when the delay time is changing.

Parent *PyoObject*

Args

input: PyoObject Input signal to delayed.

delay: float or PyoObject, optional Delay time in seconds. Defaults to 0.25.

feedback: float or PyoObject, optional Amount of output signal sent back into the delay line. Defaults to 0.

crossfade: float, optional Crossfade time, in seconds, between overlaped readers. Defaults to 0.05.

maxdelay: float, optional Maximum delay length in seconds. Available only at initialization. Defaults to 1.

Note: The minimum delay time allowed with SmoothDelay is one sample. It can be computed with :

onesamp = 1.0 / s.getSamplingRate()

See also:

Delay, *Waveguide*

```
>>> s = Server().boot()
>>> s.start()
>>> sf = SfPlayer(SNDS_PATH+"/transparent.aif", loop=True, mul=0.3).mix(2).out()
>>> lf = Sine(freq=0.1, mul=0.24, add=0.25)
>>> sd = SmoothDelay(sf, delay=lf, feedback=0.5, crossfade=0.05, mul=0.7).out()
```

setInput (*x*, *fadetime=0.05*)
Replace the *input* attribute.

Args

x: PyoObject New signal to delayed.

fadetime: float, optional Crossfade time between old and new input. Defaults to 0.05.

setDelay (*x*)
Replace the *delay* attribute.

Args

x: float or PyoObject New *delay* attribute.

setFeedback (*x*)
Replace the *feedback* attribute.

Args

x: float or PyoObject New *feedback* attribute.

setCrossfade (*x*)
Replace the *crossfade* attribute.

Args

x: float New *crossfade* attribute.

reset ()
Reset the memory buffer to zeros.

ctrl (*map_list=None*, *title=None*, *wxnoserver=False*)
Opens a sliders window to control the parameters of the object. Only parameters that can be set to a PyoObject are allowed to be mapped on a slider.

If a list of values are given to a parameter, a multisliders will be used to control each stream independently.

Args

map_list: list of SLMap objects, optional Users defined set of parameters scaling. There is default scaling for each object that accept *ctrl* method.

title: string, optional Title of the window. If none is provided, the name of the class is used.

wxnoserver: boolean, optional With wxPython graphical toolkit, if True, tells the interpreter that there will be no server window.

If *wxnoserver* is set to True, the interpreter will not wait for the server GUI before showing the controller window.

input
PyoObject. Input signal to delayed.

delay
float or PyoObject. Delay time in seconds.

feedback
float or PyoObject. Amount of output signal sent back into the delay line.

crossfade
float. Crossfade time, in seconds, between overlaps.

Prefix expression evaluators

Prefix audio expression evaluator.

This family implements a tiny functional programming language that can be used to write synthesis or signal processing algorithms.

API documentation

This API is in alpha stage and subject to future changes!

Builtin functions

Arithmetic operators

- (+ x y) : returns the sum of two values.
- (- x y) : substracts the second value to the first and returns the result.
- (* x y) : returns the multiplication of two values.
- (/ x y) : returns the quotient of x/y.
- (^ x y) : returns x to the power y.
- (% x y) : returns the floating-point remainder of x/y.
- (neg x) : returns the negative of x.

Moving phase operators

- (++ x y) : increments its internal state by x and wrap around 0.0 and y.

- (– x y) : decrements its internal state by x and wrap around 0.0 and y.
- (~ x y) : generates a periodic ramp from 0 to 1 with frequency x and phase y.

Conditional operators

- (< x y) : returns 1 if x is less than y, otherwise returns 0.
- (<= x y) : returns 1 if x is less than or equal to y, otherwise returns 0.
- (> x y) : returns 1 if x is greater than y, otherwise returns 0.
- (>= x y) : returns 1 if x is greater than or equal to y, otherwise returns 0.
- (== x y) : returns 1 if x is equal to y, otherwise returns 0.
- (!= x y) : returns 1 if x is not equal to y, otherwise returns 0.
- (if (cond) (then) (else)) : returns then for any non-zero value of cond, otherwise returns else.
- (and x y) : returns 1 if both x and y are not 0, otherwise returns 0.
- (or x y) : returns 1 if one of x or y are not 0, otherwise returns 0.

Trigonometric functions

- (sin x) : returns the sine of an angle of x radians.
- (cos x) : returns the cosine of an angle of x radians.
- (tan x) : returns the tangent of x radians.
- (tanh x) : returns the hyperbolic tangent of x radians.
- (atan x) : returns the principal value of the arc tangent of x, expressed in radians.
- (atan2 x y) : returns the principal value of the arc tangent of y/x, expressed in radians.

Power and logarithmic functions

- (sqrt x) : returns the square root of x.
- (log x) : returns the natural logarithm of x.
- (log2 x) : returns the binary (base-2) logarithm of x.
- (log10 x) : returns the common (base-10) logarithm of x.
- (pow x y) : returns x to the power y.

Clipping functions

- (abs x) : returns the absolute value of x.
- (floor x) : rounds x downward, returning the largest integral value that is not greater than x.
- (ceil x) : rounds x upward, returning the smallest integral value that is not less than x.
- (exp x) : returns the constant e to the power x.
- (round x) : returns the integral value that is nearest to x.
- (min x y) : returns the smaller of its arguments: either x or y.
- (max x y) : returns the larger of its arguments: either x or y.
- (wrap x) : wraps x between 0 and 1.

Random fuctions

- (randf x y) : returns a pseudo-random floating-point number in the range between x and y.

- (randi x y) : returns a pseudo-random integral number in the range between x and y.

Complex numbers

- (complex x y) : returns a complex number where x is the real part and y the imaginary part.
- (real x) : returns the real part of the complex number x.
- (imag x) : returns the imaginary part of the complex number x.

Filter functions

- (delay x) : one sample delay.
- (sah x y) : samples and holds x value whenever y is smaller than its previous state.
- (rpole x y) : real one-pole recursive filter. returns x + last_out * y.
- (rzero x y) : real one-zero non-recursive filter. returns x - last_x * y.
- (cpole x y) : complex one-pole recursive filter. x is the complex signal to filter, y is a complex coefficient, it returns a complex signal.
- (czero x y) : complex one-zero non-recursive filter. x is the complex signal to filter, y is a complex coefficient, it returns a complex signal.

Constants

- (const x) : returns x.
- (pi) : returns an approximated value of pi.
- (twopi) : returns a constant with value pi*2.
- (e) : returns an approximated value of e.
- (sr) : returns the curretn sampling rate.

Comments

A comment starts with two slashs (//) and ends at the end of the line:

```
// This is a comment!
```

Input and Output signals

User has access to the last buffer size of input and output samples.

To use samples from past input, use $x[n] notation, where n is the position from the current time. $x[0] is the current input, $x[-1] is the previous one and $x[-buffersize] is the last available input sample.

To use samples from past output, use $y[n] notation, where n is the position from the current time. $y[-1] is the previous output and $y[-buffersize] is the last available output sample.

Here an example of a first-order IIR lowpass filter expression:

```
// A first-order IIR lowpass filter
+ $x[0] (* (- $y[-1] $x[0]) 0.99)
```

Defining custom functions

The *define* keyword starts the definition of a custom function:

```
(define funcname (body))
```

funcname is the name used to call the function in the expression and body is the sequence of functions to execute. Arguments of the function are extracted directly from the body. They must be named $1, $2, $3, . . . , $9.

Example of a sine wave function:

```
(define osc (
    sin (* (twopi) (~ $1))
    )
)
// play a sine wave
* (osc 440) 0.3
```

State variables

User can create state variable with the keyword *let*. This is useful to set an intermediate state to be used in multiple places in the processing chain. The syntax is:

```
(let #var (body))
```

The variable name must begin with a #:

```
(let #sr 44100)
(let #freq 1000)
(let #coeff (
    ^ (e) (/ (* (* -2 (pi)) #freq) #sr)
    )
)
+ $x[0] (* (- $y[-1] $x[0]) #coeff)
```

The variable is private to a function if created inside a custom function:

```
(let #freq 250) // global #freq variable
(define osc (
    (let #freq (* $1 $2)) // local #freq variable
    sin (* (twopi) (~ #freq))
    )
)
* (+ (osc 1 #freq) (osc 2 #freq)) 0.2
```

State variables can be used to do 1 sample feedback if used before created. Undefined variables are initialized to 0:

```
(define oscloop (
        (let #xsin
            (sin (+ (* (~ $1) (twopi)) (* #xsin $2))) // #xsin used before...
        ) // ... "let" statement finished!
        #xsin // oscloop function outputs #xsin variable
    )
)
* (oscloop 200 0.7) 0.3
```

A state variable can only be a single value. The variable created with the *let* keyword can't hold a complex number (*complex*, *cpole*, *czero*). In the following statement, the state variable will only hold the real part of the complex number:

```
(let #v (complex 0.2 0.7)) // #v = 0.2
```

User variables

User variables are created with the keyword *var*:

```
(var #var (init))
```

The variable name must begin with a *#*.

They are computed only at initialization, but can be changed from the python script with method calls (varname is a string and value is a float):

```
obj.setVar(varname, value)
```

Library importation

Custom functions can be defined in an external file and imported with the *load* function:

```
(load path/to/the/file)
```

The content of the file will be inserted where the load function is called and all functions defined inside the file will then be accessible. The path can be absolute or relative to the current working directory.

Complex numbers

A complex number is created with the *complex* function:

```
(complex x y)
```

We can retrieve one part of a complex number with *real* and *imag* functions:

```
// get the real part (x) of a number
(real (complex x y))
```

If a complex number is used somewhere not waiting for a complex, real value will be used.

If a real number is used somewhere waiting for a complex, the imaginary part is set to 0.0.

Examples

Here is some expression examples.

A first-order IIR lowpass filter:

```
(var #sr 44100)
(var #cutoff 1000)
(let #coeff (exp (/ (* (* -2 (pi)) #cutoff) #sr)))
+ $x[0] (* (- $y[-1] $x[0]) #coeff)
```

A LFO'ed hyperbolic tangent distortion:

```
// $1 = lfo frequency, $2 = lfo depth
(define lfo (
        (+ (* (sin (* (twopi) (~ $1))) (- $2 1)) $2)
    )
)
tanh (* $x[0] (lfo .25 10))
```

A triangle waveform generator (use Sig(0) as input argument to bypass input):

```
(var #freq 440)
// $1 = oscillator frequency
(define triangle (
        (let #ph (~ $1))
        (- (* (min #ph (- 1 #ph)) 4) 1)
    )
)
triangle #freq
```

Objects

Expr

class Expr(*input*, *expr=''*, *mul=1*, *add=0*)

Prefix audio expression evaluator.

Expr implements a tiny functional programming language that can be used to write synthesis or signal processing algorithms.

For documentation about the language, see the module's documentation.

Parent *PyoObject*

Args

input: PyoObject Input signal to process.

expr: str, optional Expression to evaluate as a string.

```
>>> s = Server().boot()
>>> s.start()
>>> proc = '''
>>> (var #boost 1)
>>> (tanh (* $x[0] #boost))
>>> '''
>>> sf = SfPlayer(SNDS_PATH + "/transparent.aif", loop=True)
>>> ex = Expr(sf, proc, mul=0.4).out()
>>> lfo = Sine(freq=1).range(1, 20)
>>> def change():
...     ex.setVar("#boost", lfo.get())
>>> pat = Pattern(change, 0.02).play()
```

setInput(*x*, *fadetime=0.05*)

Replace the *input* attribute.

Args

x: PyoObject New signal to process.

fadetime: float, optional Crossfade time between old and new input. Defaults to 0.05.

setExpr(*x*)
Replace the *expr* attribute.

Args

x: string New expression to process.

printNodes()
Print the list of current nodes.

editor(*title='Expr Editor'*, *wxnoserver=False*)
Opens the text editor for this object.

Args

title: string, optional Title of the window. If none is provided, the name of the class is used.

wxnoserver: boolean, optional With wxPython graphical toolkit, if True, tells the interpreter that there will be no server window.

If *wxnoserver* is set to True, the interpreter will not wait for the server GUI before showing the controller window.

input
PyoObject. Input signal to process.

expr
string. New expression to process.

Filters

Different kinds of audio filtering operations.

An audio filter is designed to amplify, pass or attenuate (negative amplification) some frequency ranges. Common types include low-pass, which pass through frequencies below their cutoff frequencies, and progressively attenuates frequencies above the cutoff frequency. A high-pass filter does the opposite, passing high frequencies above the cutoff frequency, and progressively attenuating frequencies below the cutoff frequency. A bandpass filter passes frequencies between its two cutoff frequencies, while attenuating those outside the range. A band-reject filter, attenuates frequencies between its two cutoff frequencies, while passing those outside the 'reject' range.

An all-pass filter, passes all frequencies, but affects the phase of any given sinusoidal component according to its frequency.

Biquad

class Biquad(*input*, *freq=1000*, *q=1*, *type=0*, *mul=1*, *add=0*)
A sweepable general purpose biquadratic digital filter.

Parent `PyoObject`

Args

input: PyoObject Input signal to process.

freq: float or PyoObject, optional Cutoff or center frequency of the filter. Defaults to 1000.

q: float or PyoObject, optional Q of the filter, defined (for bandpass filters) as freq/bandwidth. Should be between 1 and 500. Defaults to 1.

type: int, optional

Filter type. Five possible values :

0. lowpass (default)
1. highpass
2. bandpass
3. bandstop
4. allpass

```
>>> s = Server().boot()
>>> s.start()
>>> a = Noise(mul=.7)
>>> lfo = Sine(freq=[.2, .25], mul=1000, add=1500)
>>> f = Biquad(a, freq=lfo, q=5, type=2).out()
```

setInput(*x*, *fadetime=0.05*)
Replace the *input* attribute.

Args

x: PyoObject New signal to process.

fadetime: float, optional Crossfade time between old and new input. Defaults to 0.05.

setFreq(*x*)
Replace the *freq* attribute.

Args

x: float or PyoObject New *freq* attribute.

setQ(*x*)
Replace the *q* attribute. Should be between 1 and 500.

Args

x: float or PyoObject New *q* attribute.

setType(*x*)
Replace the *type* attribute.

Args

x: int

New *type* attribute. 0.lowpass 1. highpass 2. bandpass 3. bandstop 4. allpass

ctrl(*map_list=None*, *title=None*, *wxnoserver=False*)
Opens a sliders window to control the parameters of the object. Only parameters that can be set to a PyoObject are allowed to be mapped on a slider.

If a list of values are given to a parameter, a multisliders will be used to control each stream independently.

Args

map_list: list of SLMap objects, optional Users defined set of parameters scaling. There is default scaling for each object that accept *ctrl* method.

title: string, optional Title of the window. If none is provided, the name of the class is used.

wxnoserver: boolean, optional With wxPython graphical toolkit, if True, tells the interpreter that there will be no server window.

If *wxnoserver* is set to True, the interpreter will not wait for the server GUI before showing the controller window.

input
PyoObject. Input signal to process.

freq
float or PyoObject. Cutoff or center frequency of the filter.

q
float or PyoObject. Q of the filter.

type
int. Filter type.

Biquadx

class Biquadx(*input*, *freq=1000*, *q=1*, *type=0*, *stages=4*, *mul=1*, *add=0*)
A multi-stages sweepable general purpose biquadratic digital filter.

Biquadx is equivalent to a filter consisting of more layers of Biquad with the same arguments, serially connected. It is faster than using a large number of instances of the Biquad object, It uses less memory and allows filters with sharper cutoff.

Parent *PyoObject*

Args

input: PyoObject Input signal to process.

freq: float or PyoObject, optional Cutoff or center frequency of the filter. Defaults to 1000.

q: float or PyoObject, optional Q of the filter, defined (for bandpass filters) as freq/bandwidth. Should be between 1 and 500. Defaults to 1.

type: int, optional

Filter type. Five possible values :

0. lowpass (default)
1. highpass
2. bandpass
3. bandstop
4. allpass

stages: int, optional The number of filtering stages in the filter stack. Defaults to 4.

```
>>> s = Server().boot()
>>> s.start()
>>> a = Noise(mul=.7)
>>> lfo = Sine(freq=[.2, .25], mul=1000, add=1500)
>>> f = Biquadx(a, freq=lfo, q=5, type=2).out()
```

setInput(*x*, *fadetime=0.05*)
Replace the *input* attribute.

Args

x: PyoObject New signal to process.

fadetime: float, optional Crossfade time between old and new input. Defaults to 0.05.

setFreq(*x*)
Replace the *freq* attribute.

Args

x: float or PyoObject New *freq* attribute.

setQ(*x*)
Replace the *q* attribute. Should be between 1 and 500.

Args

x: float or PyoObject New *q* attribute.

setType(*x*)
Replace the *type* attribute.

Args

x: int New *type* attribute. 0. lowpass 1. highpass 2. bandpass 3. bandstop 4. allpass

setStages(*x*)
Replace the *stages* attribute.

Args

x: int New *stages* attribute.

ctrl(*map_list=None*, *title=None*, *wxnoserver=False*)
Opens a sliders window to control the parameters of the object. Only parameters that can be set to a PyoObject are allowed to be mapped on a slider.

If a list of values are given to a parameter, a multisliders will be used to control each stream independently.

Args

map_list: list of SLMap objects, optional Users defined set of parameters scaling. There is default scaling for each object that accept *ctrl* method.

title: string, optional Title of the window. If none is provided, the name of the class is used.

wxnoserver: boolean, optional With wxPython graphical toolkit, if True, tells the interpreter that there will be no server window.

If *wxnoserver* is set to True, the interpreter will not wait for the server GUI before showing the controller window.

input
PyoObject. Input signal to process.

freq
float or PyoObject. Cutoff or center frequency of the filter.

q
float or PyoObject. Q of the filter.

type
int. Filter type.

stages
int. The number of filtering stages.

Biquada

class Biquada(*input*, *b0=0.005066*, *b1=0.010132*, *b2=0.005066*, *a0=1.070997*, *a1=-1.979735*, *a2=0.929003*, *mul=1*, *add=0*)

A general purpose biquadratic digital filter (floating-point arguments).

A digital biquad filter is a second-order recursive linear filter, containing two poles and two zeros. Biquada is a "Direct Form 1" implementation of a Biquad filter:

y[n] = (b0*x[n] + b1*x[n-1] + b2*x[n-2] - a1*y[n-1] - a2*y[n-2]) / a0

This object is directly controlled via the six coefficients, as floating-point values or audio stream, of the filter. There is no clipping of the values given as coefficients, so, unless you know what you do, it is recommended to use the Biquad object, which is controlled with frequency, Q and type arguments.

The default values of the object give a lowpass filter with a 1000 Hz cutoff frequency.

Parent *PyoObject*

Args

input: PyoObject Input signal to process.

b0: float or PyoObject, optional Amplitude of the current sample. Defaults to 0.005066.

b1: float or PyoObject, optional Amplitude of the first input sample delayed. Defaults to 0.010132.

b2: float or PyoObject, optional Amplitude of the second input sample delayed. Defaults to 0.005066.

a0: float or PyoObject, optional Overall gain coefficient. Defaults to 1.070997.

a1: float or PyoObject, optional Amplitude of the first output sample delayed. Defaults to -1.979735.

a2: float or PyoObject, optional Amplitude of the second output sample delayed. Defaults to 0.929003.

```
>>> s = Server().boot()
>>> s.start()
>>> a = Noise(mul=.7)
>>> lf = Sine([1.5, 2], mul=.07, add=-1.9)
>>> f = Biquada(a, 0.005066, 0.010132, 0.005066, 1.070997, lf, 0.929003).out()
```

setInput(*x*, *fadetime=0.05*)

Replace the *input* attribute.

Args

x: PyoObject New signal to process.

fadetime: float, optional Crossfade time between old and new input. Defaults to 0.05.

setB0(*x*)

Replace the *b0* attribute.

Args

x: float or PyoObject New *b0* attribute.

setB1(*x*)

Replace the *b1* attribute.

Args

x: float or PyoObject New *b1* attribute.

setB2(*x*)
Replace the *b2* attribute.

Args

x: float or PyoObject New *b2* attribute.

setA0(*x*)
Replace the *a0* attribute.

Args

x: float or PyoObject New *a0* attribute.

setA1(*x*)
Replace the *a1* attribute.

Args

x: float or PyoObject New *a1* attribute.

setA2(*x*)
Replace the *a2* attribute.

Args

x: float or PyoObject New *a2* attribute.

setCoeffs(**args*, ***kwds*)
Replace all filter coefficients.

Args

b0: float or PyoObject, optional New *b0* attribute.

b1: float or PyoObject, optional New *b1* attribute.

b2: float or PyoObject, optional New *b2* attribute.

a0: float or PyoObject, optional New *a0* attribute.

a1: float or PyoObject, optional New *a1* attribute.

a2: float or PyoObject, optional New *a2* attribute.

ctrl(*map_list=None*, *title=None*, *wxnoserver=False*)
Opens a sliders window to control the parameters of the object. Only parameters that can be set to a PyoObject are allowed to be mapped on a slider.

If a list of values are given to a parameter, a multisliders will be used to control each stream independently.

Args

map_list: list of SLMap objects, optional Users defined set of parameters scaling. There is default scaling for each object that accept *ctrl* method.

title: string, optional Title of the window. If none is provided, the name of the class is used.

wxnoserver: boolean, optional With wxPython graphical toolkit, if True, tells the interpreter that there will be no server window.

If *wxnoserver* is set to True, the interpreter will not wait for the server GUI before showing the controller window.

input
PyoObject. Input signal to process.

b0
float or PyoObject. *b0* coefficient.

b1
float or PyoObject. *b1* coefficient.

b2
float or PyoObject. *b2* coefficient.

a0
float or PyoObject. *a0* coefficient.

a1
float or PyoObject. *a1* coefficient.

a2
float or PyoObject. *a2* coefficient.

EQ

class EQ(*input*, *freq=1000*, *q=1*, *boost=-3.0*, *type=0*, *mul=1*, *add=0*)
Equalizer filter.

EQ is a biquadratic digital filter designed for equalization. It provides peak/notch and lowshelf/highshelf filters for building parametric equalizers.

Parent *PyoObject*

Args

input: PyoObject Input signal to process.

freq: float or PyoObject, optional Cutoff or center frequency of the filter. Defaults to 1000.

q: float or PyoObject, optional Q of the filter, defined as freq/bandwidth. Should be between 1 and 500. Defaults to 1.

boost: float or PyoObject, optional Gain, expressed in dB, to add or remove at the center frequency. Default to -3.

type: int, optional

Filter type. Three possible values :

0. peak/notch (default)
1. lowshelf
2. highshelf

```
>>> s = Server().boot()
>>> s.start()
>>> amp = Fader(1, 1, mul=.15).play()
>>> src = PinkNoise(amp)
>>> fr = Sine(.2, 0, 500, 1500)
>>> boo = Sine([4, 4], 0, 6)
>>> out = EQ(src, freq=fr, q=1, boost=boo, type=0).out()
```

setInput(*x*, *fadetime=0.05*)
Replace the *input* attribute.

Args

x: PyoObject New signal to process.

fadetime: float, optional Crossfade time between old and new input. Defaults to 0.05.

setFreq(*x*)
Replace the *freq* attribute.

Args

x: float or PyoObject New *freq* attribute.

setQ(*x*)
Replace the *q* attribute. Should be between 1 and 500.

Args

x: float or PyoObject New *q* attribute.

setBoost(*x*)
Replace the *boost* attribute, expressed in dB.

Args

x: float or PyoObject New *boost* attribute.

setType(*x*)
Replace the *type* attribute.

Args

x: int New *type* attribute. 0. peak 1. lowshelf 2. highshelf

ctrl(*map_list=None*, *title=None*, *wxnoserver=False*)
Opens a sliders window to control the parameters of the object. Only parameters that can be set to a PyoObject are allowed to be mapped on a slider.

If a list of values are given to a parameter, a multisliders will be used to control each stream independently.

Args

map_list: list of SLMap objects, optional Users defined set of parameters scaling. There is default scaling for each object that accept *ctrl* method.

title: string, optional Title of the window. If none is provided, the name of the class is used.

wxnoserver: boolean, optional With wxPython graphical toolkit, if True, tells the interpreter that there will be no server window.

If *wxnoserver* is set to True, the interpreter will not wait for the server GUI before showing the controller window.

input
PyoObject. Input signal to process.

freq
float or PyoObject. Cutoff or center frequency of the filter.

q
float or PyoObject. Q of the filter.

boost
float or PyoObject. Boost factor of the filter.

type
int. Filter type.

Tone

class Tone(*input*, *freq=1000*, *mul=1*, *add=0*)
A first-order recursive low-pass filter with variable frequency response.

Parent *PyoObject*

Args

input: PyoObject Input signal to process.

freq: float or PyoObject, optional Cutoff frequency of the filter in hertz. Default to 1000.

```
>>> s = Server().boot()
>>> s.start()
>>> n = Noise(.3)
>>> lf = Sine(freq=.2, mul=800, add=1000)
>>> f = Tone(n, lf).mix(2).out()
```

setInput(*x*, *fadetime=0.05*)
Replace the *input* attribute.

Args

x: PyoObject New signal to process.

fadetime: float, optional Crossfade time between old and new input. Default to 0.05.

setFreq(*x*)
Replace the *freq* attribute.

Args

x: float or PyoObject New *freq* attribute.

ctrl(*map_list=None*, *title=None*, *wxnoserver=False*)
Opens a sliders window to control the parameters of the object. Only parameters that can be set to a PyoObject are allowed to be mapped on a slider.

If a list of values are given to a parameter, a multisliders will be used to control each stream independently.

Args

map_list: list of SLMap objects, optional Users defined set of parameters scaling. There is default scaling for each object that accept *ctrl* method.

title: string, optional Title of the window. If none is provided, the name of the class is used.

wxnoserver: boolean, optional With wxPython graphical toolkit, if True, tells the interpreter that there will be no server window.

If *wxnoserver* is set to True, the interpreter will not wait for the server GUI before showing the controller window.

input
PyoObject. Input signal to process.

freq
float or PyoObject. Cutoff frequency of the filter.

Atone

class Atone(*input*, *freq=1000*, *mul=1*, *add=0*)

A first-order recursive high-pass filter with variable frequency response.

Parent *PyoObject*

Args

input: PyoObject Input signal to process.

freq: float or PyoObject, optional Cutoff frequency of the filter in hertz. Default to 1000.

```
>>> s = Server().boot()
>>> s.start()
>>> n = Noise(.3)
>>> lf = Sine(freq=.2, mul=5000, add=6000)
>>> f = Atone(n, lf).mix(2).out()
```

setInput(*x*, *fadetime=0.05*)

Replace the *input* attribute.

Args

x: PyoObject New signal to process.

fadetime: float, optional Crossfade time between old and new input. Default to 0.05.

setFreq(*x*)

Replace the *freq* attribute.

Args

x: float or PyoObject New *freq* attribute.

ctrl(*map_list=None*, *title=None*, *wxnoserver=False*)

Opens a sliders window to control the parameters of the object. Only parameters that can be set to a PyoObject are allowed to be mapped on a slider.

If a list of values are given to a parameter, a multisliders will be used to control each stream independently.

Args

map_list: list of SLMap objects, optional Users defined set of parameters scaling. There is default scaling for each object that accept *ctrl* method.

title: string, optional Title of the window. If none is provided, the name of the class is used.

wxnoserver: boolean, optional With wxPython graphical toolkit, if True, tells the interpreter that there will be no server window.

If *wxnoserver* is set to True, the interpreter will not wait for the server GUI before showing the controller window.

input

PyoObject. Input signal to process.

freq

float or PyoObject. Cutoff frequency of the filter.

Port

class Port (*input*, *risetime=0.05*, *falltime=0.05*, *init=0*, *mul=1*, *add=0*)

Exponential portamento.

Perform an exponential portamento on an audio signal with different rising and falling times.

Parent *PyoObject*

Args

input: PyoObject Input signal to process.

risetime: float or PyoObject, optional Time to reach upward value in seconds. Defaults to 0.05.

falltime: float or PyoObject, optional Time to reach downward value in seconds. Defaults to 0.05.

init: float, optional Initial state of the internal memory. Available at intialization time only. Defaults to 0.

```
>>> from random import uniform
>>> s = Server().boot()
>>> s.start()
>>> x = Sig(value=500)
>>> p = Port(x, risetime=.1, falltime=1)
>>> a = Sine(freq=[p, p*1.01], mul=.2).out()
>>> def new_freq():
...     x.value = uniform(400, 800)
>>> pat = Pattern(function=new_freq, time=1).play()
```

setInput (*x*, *fadetime=0.05*)

Replace the *input* attribute.

Args

x: PyoObject New signal to process.

fadetime: float, optional Crossfade time between old and new input. Defaults to 0.05.

setRiseTime (*x*)

Replace the *risetime* attribute.

Args

x: float or PyoObject New *risetime* attribute.

setFallTime (*x*)

Replace the *falltime* attribute.

Args

x: float or PyoObject New *falltime* attribute.

ctrl (*map_list=None*, *title=None*, *wxnoserver=False*)

Opens a sliders window to control the parameters of the object. Only parameters that can be set to a PyoObject are allowed to be mapped on a slider.

If a list of values are given to a parameter, a multisliders will be used to control each stream independently.

Args

map_list: list of SLMap objects, optional Users defined set of parameters scaling. There is default scaling for each object that accept *ctrl* method.

title: string, optional Title of the window. If none is provided, the name of the class is used.

wxnoserver: boolean, optional With wxPython graphical toolkit, if True, tells the interpreter that there will be no server window.

If *wxnoserver* is set to True, the interpreter will not wait for the server GUI before showing the controller window.

input
PyoObject. Input signal to process.

risetime
float or PyoObject. Time to reach upward value in seconds.

falltime
float or PyoObject. Time to reach downward value in seconds.

DCBlock

class DCBlock(*input*, *mul=1*, *add=0*)
Implements the DC blocking filter.

Parent *PyoObject*

Args

input: PyoObject Input signal to process.

```
>>> s = Server().boot()
>>> s.start()
>>> n = Noise(.01)
>>> w = Delay(n, delay=[0.02, 0.01], feedback=.995, mul=.5)
>>> f = DCBlock(w).out()
```

setInput(*x*, *fadetime=0.05*)
Replace the *input* attribute.

Args

x: PyoObject New signal to process.

fadetime: float, optional Crossfade time between old and new input. Default to 0.05.

input
PyoObject. Input signal to process.

BandSplit

class BandSplit(*input*, *num=6*, *min=20*, *max=20000*, *q=1*, *mul=1*, *add=0*)
Splits an input signal into multiple frequency bands.

The input signal will be separated into *num* bands between *min* and *max* frequencies using second-order band-pass filters. Each band will then be assigned to an independent audio stream. Useful for multiband processing.

Parent *PyoObject*

Args

input: PyoObject Input signal to process.

num: int, optional Number of frequency bands created. Available at initialization time only. Defaults to 6.

min: float, optional Lowest frequency. Available at initialization time only. Defaults to 20.

max: float, optional Highest frequency. Available at initialization time only. Defaults to 20000.

q: float or PyoObject, optional Q of the filters, defined as center frequency / bandwidth. Should be between 1 and 500. Defaults to 1.

See also:

FourBand, *MultiBand*

```
>>> s = Server().boot()
>>> s.start()
>>> lfos = Sine(freq=[.3,.4,.5,.6,.7,.8], mul=.5, add=.5)
>>> n = PinkNoise(.5)
>>> a = BandSplit(n, num=6, min=250, max=4000, q=5, mul=lfos).out()
```

setInput(*x*, *fadetime=0.05*)

Replace the *input* attribute.

Args

x: PyoObject New signal to process.

fadetime: float, optional Crossfade time between old and new input. Defaults to 0.05.

setQ(*x*)

Replace the *q* attribute.

Args

x: float or PyoObject new *q* attribute.

ctrl(*map_list=None*, *title=None*, *wxnoserver=False*)

Opens a sliders window to control the parameters of the object. Only parameters that can be set to a PyoObject are allowed to be mapped on a slider.

If a list of values are given to a parameter, a multisliders will be used to control each stream independently.

Args

map_list: list of SLMap objects, optional Users defined set of parameters scaling. There is default scaling for each object that accept *ctrl* method.

title: string, optional Title of the window. If none is provided, the name of the class is used.

wxnoserver: boolean, optional With wxPython graphical toolkit, if True, tells the interpreter that there will be no server window.

If *wxnoserver* is set to True, the interpreter will not wait for the server GUI before showing the controller window.

input

PyoObject. Input signal to process.

q

float or PyoObject. Q of the filters.

FourBand

class FourBand(*input*, *freq1=150*, *freq2=500*, *freq3=2000*, *mul=1*, *add=0*)

Splits an input signal into four frequency bands.

The input signal will be separated into 4 bands around *freqs* arguments using fourth-order Linkwitz-Riley lowpass and highpass filters. Each band will then be assigned to an independent audio stream. The sum of the four bands reproduces the same signal as the *input*. Useful for multiband processing.

Parent *PyoObject*

Args

input: PyoObject Input signal to process.

freq1: float or PyoObject, optional First crossover frequency. First band will contain signal from 0 Hz to *freq1* Hz. Defaults to 150.

freq2: float or PyoObject, optional Second crossover frequency. Second band will contain signal from *freq1* Hz to *freq2*. *freq2* is the lower limit of the third band signal. Defaults to 500.

freq3: float or PyoObject, optional Third crossover frequency. It's the upper limit of the third band signal and fourth band will contain signal from *freq3* to sr/2. Defaults to 2000.

See also:

BandSplit, *MultiBand*

```
>>> s = Server().boot()
>>> s.start()
>>> lfos = Sine(freq=[.3,.4,.5,.6], mul=.5, add=.5)
>>> n = PinkNoise(.3)
>>> a = FourBand(n, freq1=250, freq2=1000, freq3=2500, mul=lfos).out()
```

setInput(*x*, *fadetime=0.05*)

Replace the *input* attribute.

Args

x: PyoObject New signal to process.

fadetime: float, optional Crossfade time between old and new input. Defaults to 0.05.

setFreq1(*x*)

Replace the *freq1* attribute.

Args

x: float or PyoObject new *freq1* attribute.

setFreq2(*x*)

Replace the *freq2* attribute.

Args

x: float or PyoObject new *freq2* attribute.

setFreq3(*x*)

Replace the *freq3* attribute.

Args

x: float or PyoObject new *freq3* attribute.

ctrl(*map_list=None*, *title=None*, *wxnoserver=False*)
Opens a sliders window to control the parameters of the object. Only parameters that can be set to a PyoObject are allowed to be mapped on a slider.

If a list of values are given to a parameter, a multisliders will be used to control each stream independently.

Args

map_list: list of SLMap objects, optional Users defined set of parameters scaling. There is default scaling for each object that accept *ctrl* method.

title: string, optional Title of the window. If none is provided, the name of the class is used.

wxnoserver: boolean, optional With wxPython graphical toolkit, if True, tells the interpreter that there will be no server window.

If *wxnoserver* is set to True, the interpreter will not wait for the server GUI before showing the controller window.

input
PyoObject. Input signal to process.

freq1
float or PyoObject. First crossover frequency.

freq2
float or PyoObject. Second crossover frequency.

freq3
float or PyoObject. Third crossover frequency.

MultiBand

class MultiBand(*input*, *num=8*, *mul=1*, *add=0*)
Splits an input signal into multiple frequency bands.

The input signal will be separated into *num* logarithmically spaced frequency bands using fourth-order Linkwitz-Riley lowpass and highpass filters. Each band will then be assigned to an independent audio stream. The sum of all bands reproduces the same signal as the *input*. Useful for multiband processing.

User-defined frequencies can be assigned with the *setFrequencies()* method.

Parent *PyoObject*

Args

input: PyoObject Input signal to process.

num: int, optional Number of frequency bands created, between 2 and 16. Available at initialization time only. Defaults to 8.

See also:

BandSplit, *FourBand*

```
>>> s = Server().boot()
>>> s.start()
>>> lfos = Sine(freq=[.1,.2,.3,.4,.5,.6,.7,.8], mul=.5, add=.5)
>>> n = PinkNoise(.3)
>>> a = MultiBand(n, num=8, mul=lfos).out()
```

setInput(*x*, *fadetime=0.05*)
Replace the *input* attribute.

Args

x: PyoObject New signal to process.

fadetime: float, optional Crossfade time between old and new input. Defaults to 0.05.

setFrequencies(*freqs*)
Change the filters splitting frequencies.

This method can be used to change filter frequencies to a particular set. Frequency list length must be egal to the number of bands minus 1.

Args

freqs: list of floats New frequencies used as filter cutoff frequencies.

input
PyoObject. Input signal to process.

Hilbert

class Hilbert(*input*, *mul=1*, *add=0*)
Hilbert transform.

Hilbert is an IIR filter based implementation of a broad-band 90 degree phase difference network. The outputs of hilbert have an identical frequency response to the input (i.e. they sound the same), but the two outputs have a constant phase difference of 90 degrees, plus or minus some small amount of error, throughout the entire frequency range. The outputs are in quadrature.

Hilbert is useful in the implementation of many digital signal processing techniques that require a signal in phase quadrature. The real part corresponds to the cosine output of hilbert, while the imaginary part corresponds to the sine output. The two outputs have a constant phase difference throughout the audio range that corresponds to the phase relationship between cosine and sine waves.

Parent *PyoObject*

Args

input: PyoObject Input signal to process.

Note: Real and imaginary parts are two separated set of streams. The user should call :

Hilbert['real'] to retrieve the real part.
Hilbert['imag'] to retrieve the imaginary part.

```
>>> s = Server().boot()
>>> s.start()
>>> a = SfPlayer(SNDS_PATH + "/accord.aif", loop=True, mul=0.5).out(0)
>>> b = Hilbert(a)
>>> quad = Sine([250, 500], [0, .25])
>>> mod1 = b['real'] * quad[0]
>>> mod2 = b['imag'] * quad[1]
```

(continues on next page)

(continued from previous page)

```
>>> up = (mod1 - mod2) * 0.7
>>> down = mod1 + mod2
>>> up.out(1)
```

get(*identifier='real'*, *all=False*)
Return the first sample of the current buffer as a float.

Can be used to convert audio stream to usable Python data.

"real" or "imag" must be given to *identifier* to specify which stream to get value from.

Args

identifier: string {"real", "imag"} Address string parameter identifying audio stream. Defaults to "real".

all: boolean, optional If True, the first value of each object's stream will be returned as a list. Otherwise, only the value of the first object's stream will be returned as a float. Defaults to False.

setInput(*x*, *fadetime=0.05*)
Replace the *input* attribute.

Args

x: PyoObject New signal to process.

fadetime: float, optional Crossfade time between old and new input. Default to 0.05.

input
PyoObject. Input signal to process.

Allpass

class Allpass(*input*, *delay=0.01*, *feedback=0*, *maxdelay=1*, *mul=1*, *add=0*)
Delay line based allpass filter.

Allpass is based on the combination of feedforward and feedback comb filter. This kind of filter is often used in simple digital reverb implementations.

Parent *PyoObject*

Args

input: PyoObject Input signal to process.

delay: float or PyoObject, optional Delay time in seconds. Defaults to 0.01.

feedback: float or PyoObject, optional Amount of output signal sent back into the delay line. Defaults to 0.

maxdelay: float, optional Maximum delay length in seconds. Available only at initialization. Defaults to 1.

```
>>> # SIMPLE REVERB
>>> s = Server().boot()
>>> s.start()
>>> a = SfPlayer(SNDS_PATH + "/transparent.aif", loop=True, mul=0.25).mix(2).out()
>>> b1 = Allpass(a, delay=[.0204,.02011], feedback=0.25)
>>> b2 = Allpass(b1, delay=[.06653,.06641], feedback=0.31)
```

(continues on next page)

(continued from previous page)

```
>>> b3 = Allpass(b2, delay=[.035007,.03504], feedback=0.4)
>>> b4 = Allpass(b3, delay=[.023021 ,.022987], feedback=0.55)
>>> c1 = Tone(b1, 5000, mul=0.2).out()
>>> c2 = Tone(b2, 3000, mul=0.2).out()
>>> c3 = Tone(b3, 1500, mul=0.2).out()
>>> c4 = Tone(b4, 500, mul=0.2).out()
```

setInput (*x*, *fadetime=0.05*)
Replace the *input* attribute.

Args

x: PyoObject New signal to process.

fadetime: float, optional Crossfade time between old and new input. Defaults to 0.05.

setDelay (*x*)
Replace the *delay* attribute.

Args

x: float or PyoObject New *delay* attribute.

setFeedback (*x*)
Replace the *feedback* attribute.

Args

x: float or PyoObject New *feedback* attribute.

ctrl (*map_list=None*, *title=None*, *wxnoserver=False*)
Opens a sliders window to control the parameters of the object. Only parameters that can be set to a PyoObject are allowed to be mapped on a slider.

If a list of values are given to a parameter, a multisliders will be used to control each stream independently.

Args

map_list: list of SLMap objects, optional Users defined set of parameters scaling. There is default scaling for each object that accept *ctrl* method.

title: string, optional Title of the window. If none is provided, the name of the class is used.

wxnoserver: boolean, optional With wxPython graphical toolkit, if True, tells the interpreter that there will be no server window.

If *wxnoserver* is set to True, the interpreter will not wait for the server GUI before showing the controller window.

input
PyoObject. Input signal to process.

delay
float or PyoObject. Delay time in seconds.

feedback
float or PyoObject. Amount of output signal sent back into the delay line.

Allpass2

class Allpass2(*input*, *freq=1000*, *bw=100*, *mul=1*, *add=0*)
Second-order phase shifter allpass.

This kind of filter is used in phaser implementation. The signal of this filter, when added to original sound, creates a notch in the spectrum at frequencies that are in phase opposition.

Parent *PyoObject*

Args

input: PyoObject Input signal to process.

freq: float or PyoObject, optional Center frequency of the filter. Defaults to 1000.

bw: float or PyoObject, optional Bandwidth of the filter in Hertz. Defaults to 100.

```
>>> s = Server().boot()
>>> s.start()
>>> # 3 STAGES PHASER
>>> a = BrownNoise(.025).mix(2).out()
>>> blfo = Sine(freq=.1, mul=250, add=500)
>>> b = Allpass2(a, freq=blfo, bw=125).out()
>>> clfo = Sine(freq=.14, mul=500, add=1000)
>>> c = Allpass2(b, freq=clfo, bw=350).out()
>>> dlfo = Sine(freq=.17, mul=1000, add=2500)
>>> d = Allpass2(c, freq=dlfo, bw=800).out()
```

setInput(*x*, *fadetime=0.05*)
Replace the *input* attribute.

Args

x: PyoObject New signal to process.

fadetime: float, optional Crossfade time between old and new input. Defaults to 0.05.

setFreq(*x*)
Replace the *freq* attribute.

Args

x: float or PyoObject New *freq* attribute.

setBw(*x*)
Replace the *bw* attribute.

Args

x: float or PyoObject New *bw* attribute.

ctrl(*map_list=None*, *title=None*, *wxnoserver=False*)
Opens a sliders window to control the parameters of the object. Only parameters that can be set to a PyoObject are allowed to be mapped on a slider.

If a list of values are given to a parameter, a multisliders will be used to control each stream independently.

Args

map_list: list of SLMap objects, optional Users defined set of parameters scaling. There is default scaling for each object that accept *ctrl* method.

title: string, optional Title of the window. If none is provided, the name of the class is used.

wxnoserver: boolean, optional With wxPython graphical toolkit, if True, tells the interpreter that there will be no server window.

If *wxnoserver* is set to True, the interpreter will not wait for the server GUI before showing the controller window.

input
PyoObject. Input signal to filter.

freq
float or PyoObject. Center frequency of the filter.

bw
float or PyoObject. Bandwidth of the filter.

Phaser

class Phaser(*input*, *freq=1000*, *spread=1.1*, *q=10*, *feedback=0*, *num=8*, *mul=1*, *add=0*)
Multi-stages second-order phase shifter allpass filters.

Phaser implements *num* number of second-order allpass filters.

Parent *PyoObject*

Args

input: PyoObject Input signal to process.

freq: float or PyoObject, optional Center frequency of the first notch. Defaults to 1000.

spread: float or PyoObject, optional Spreading factor for upper notch frequencies. Defaults to 1.1.

q: float or PyoObject, optional Q of the filter as center frequency / bandwidth. Defaults to 10.

feedback: float or PyoObject, optional Amount of output signal which is fed back into the input of the allpass chain. Defaults to 0.

num: int, optional The number of allpass stages in series. Defines the number of notches in the spectrum. Defaults to 8.

Available at initialization only.

```
>>> s = Server().boot()
>>> s.start()
>>> fade = Fader(fadein=.1, mul=.07).play()
>>> a = Noise(fade).mix(2).out()
>>> lf1 = Sine(freq=[.1, .15], mul=100, add=250)
>>> lf2 = Sine(freq=[.18, .15], mul=.4, add=1.5)
>>> b = Phaser(a, freq=lf1, spread=lf2, q=1, num=20, mul=.5).out(0)
```

setInput(*x*, *fadetime=0.05*)
Replace the *input* attribute.

Args

x: PyoObject New signal to process.

fadetime: float, optional Crossfade time between old and new input. Defaults to 0.05.

setFreq(*x*)
Replace the *freq* attribute.

Args

x: float or PyoObject New *freq* attribute.

setSpread(*x*)
Replace the *spread* attribute.

Args

x: float or PyoObject New *spread* attribute.

setQ(*x*)
Replace the *q* attribute.

Args

x: float or PyoObject New *q* attribute.

setFeedback(*x*)
Replace the *feedback* attribute.

Args

x: float or PyoObject New *feedback* attribute.

ctrl(*map_list=None*, *title=None*, *wxnoserver=False*)
Opens a sliders window to control the parameters of the object. Only parameters that can be set to a PyoObject are allowed to be mapped on a slider.

If a list of values are given to a parameter, a multisliders will be used to control each stream independently.

Args

map_list: list of SLMap objects, optional Users defined set of parameters scaling. There is default scaling for each object that accept *ctrl* method.

title: string, optional Title of the window. If none is provided, the name of the class is used.

wxnoserver: boolean, optional With wxPython graphical toolkit, if True, tells the interpreter that there will be no server window.

If *wxnoserver* is set to True, the interpreter will not wait for the server GUI before showing the controller window.

input
PyoObject. Input signal to process.

freq
float or PyoObject. Center frequency of the first notch.

spread
float or PyoObject. Spreading factor for upper notch frequencies.

q
float or PyoObject. Q factor of the filter.

feedback
float or PyoObject. Feedback factor of the filter.

Vocoder

class Vocoder(*input*, *input2*, *freq=60*, *spread=1.25*, *q=20*, *slope=0.5*, *stages=24*, *mul=1*, *add=0*)
Applies the spectral envelope of a first sound to the spectrum of a second sound.

The vocoder is an analysis/synthesis system, historically used to reproduce human speech. In the encoder, the first input (spectral envelope) is passed through a multiband filter, each band is passed through an envelope follower, and the control signals from the envelope followers are communicated to the decoder. The decoder applies these (amplitude) control signals to corresponding filters modifying the second source (exciter).

Parent *PyoObject*

Args

input: PyoObject Spectral envelope. Gives the spectral properties of the bank of filters. For best results, this signal must have a dynamic spectrum, both for amplitudes and frequencies.

input2: PyoObject Exciter. Spectrum to filter. For best results, this signal must have a broadband spectrum with few amplitude variations.

freq: float or PyoObject, optional Center frequency of the first band. This is the base frequency used to compute the upper bands. Defaults to 60.

spread: float or PyoObject, optional Spreading factor for upper band frequencies. Each band is *freq * pow(order, spread)*, where order is the harmonic rank of the band. Defaults to 1.25.

q: float or PyoObject, optional Q of the filters as *center frequency / bandwidth*. Higher values imply more resonance around the center frequency. Defaults to 20.

slope: float or PyoObject, optional Time response of the envelope follower. Lower values mean smoother changes, while higher values mean a better time accuracy. Defaults to 0.5.

stages: int, optional The number of bands in the filter bank. Defines the number of notches in the spectrum. Defaults to 24.

Note: Altough parameters can be audio signals, values are sampled only four times per buffer size. To avoid artefacts, it is recommended to keep variations at low rate (< 20 Hz).

```
>>> s = Server().boot()
>>> s.start()
>>> sf = SfPlayer(SNDS_PATH+'/transparent.aif', loop=True)
>>> ex = BrownNoise(0.5)
>>> voc = Vocoder(sf, ex, freq=80, spread=1.2, q=20, slope=0.5)
>>> out = voc.mix(2).out()
```

setInput(*x*, *fadetime=0.05*)

Replace the *input* attribute.

Args

x: PyoObject The spectral envelope.

fadetime: float, optional Crossfade time between old and new input. Defaults to 0.05.

setInput2(*x*, *fadetime=0.05*)

Replace the *input2* attribute.

Args

x: PyoObject The exciter.

fadetime: float, optional Crossfade time between old and new input. Defaults to 0.05.

setFreq(*x*)
Replace the *freq* attribute.

Args

x: float or PyoObject New *freq* attribute.

setSpread(*x*)
Replace the *spread* attribute.

Args

x: float or PyoObject New *spread* attribute.

setQ(*x*)
Replace the *q* attribute.

Args

x: float or PyoObject New *q* attribute.

setSlope(*x*)
Replace the *slope* attribute.

Args

x: float or PyoObject New *slope* attribute.

setStages(*x*)
Replace the *stages* attribute.

Args

x: int New *stages* attribute.

ctrl(*map_list=None*, *title=None*, *wxnoserver=False*)
Opens a sliders window to control the parameters of the object. Only parameters that can be set to a PyoObject are allowed to be mapped on a slider.

If a list of values are given to a parameter, a multisliders will be used to control each stream independently.

Args

map_list: list of SLMap objects, optional Users defined set of parameters scaling. There is default scaling for each object that accept *ctrl* method.

title: string, optional Title of the window. If none is provided, the name of the class is used.

wxnoserver: boolean, optional With wxPython graphical toolkit, if True, tells the interpreter that there will be no server window.

If *wxnoserver* is set to True, the interpreter will not wait for the server GUI before showing the controller window.

input
PyoObject. The spectral envelope.

input2
PyoObject. The exciter.

freq
float or PyoObject. Center frequency of the first band.

spread
float or PyoObject. Spreading factor for upper band frequencies.

q
float or PyoObject. Q factor of the filters.

slope
float or PyoObject. Time response of the envelope follower.

stages
int. The number of bands in the filter bank.

IRWinSinc

class IRWinSinc(*input*, *freq=1000*, *bw=500*, *type=0*, *order=256*, *mul=1*, *add=0*)
Windowed-sinc filter using circular convolution.

IRWinSinc uses circular convolution to implement standard filters like lowpass, highpass, bandreject and bandpass with very flat passband response and sharp roll-off. User can defined the length, in samples, of the impulse response, also known as the filter kernel.

Parent *PyoObject*

Args

input: PyoObject Input signal to process.

freq: float or PyoObject, optional Frequency cutoff for lowpass and highpass and center frequency for bandjrect and bandpass filters, expressed in Hz. Defaults to 1000.

bw: float or PyoObject, optional Bandwidth, expressed in Hertz, for bandreject and bandpass filters. Defaults to 500.

type: int, optional

Filter type. Four possible values :

0. lowpass (default)
1. highpass
2. bandreject
3. bandpass

order: int {even number}, optional Length, in samples, of the filter kernel used for convolution. Available at initialization time only. Defaults to 256.

This value must be even. Higher is the order and sharper is the roll-off of the filter, but it is also more expensive to compute.

Note: Convolution is very expensive to compute, so the length of the impulse response (the *order* parameter) must be kept very short to run in real time.

Note that although *freq* and *bw* can be PyoObjects, the impulse response of the filter is only updated once per buffer size.

```
>>> s = Server().boot()
>>> s.start()
>>> a = Noise(.3)
>>> lfr = Sine([.15, .2], mul=2000, add=3500)
>>> lbw = Sine([.3, .25], mul=1000, add=1500)
>>> b = IRWinSinc(a, freq=lfr, bw=lbw, type=3, order=256).out()
```

setInput(*x*, *fadetime=0.05*)
Replace the *input* attribute.

Args

x: PyoObject New signal to process.

fadetime: float, optional Crossfade time between old and new input. Default to 0.05.

setFreq(*x*)
Replace the *freq* attribute.

Args

x: float or PyoObject New *freq* attribute.

setBw(*x*)
Replace the *bw* attribute.

Args

x: float or PyoObject New *bw* attribute.

setType(*x*)
Replace the *type* attribute.

Args

x: int New *type* attribute. 0. lowpass 1. highpass 2. bandreject 3. bandpass

ctrl(*map_list=None*, *title=None*, *wxnoserver=False*)
Opens a sliders window to control the parameters of the object. Only parameters that can be set to a PyoObject are allowed to be mapped on a slider.

If a list of values are given to a parameter, a multisliders will be used to control each stream independently.

Args

map_list: list of SLMap objects, optional Users defined set of parameters scaling. There is default scaling for each object that accept *ctrl* method.

title: string, optional Title of the window. If none is provided, the name of the class is used.

wxnoserver: boolean, optional With wxPython graphical toolkit, if True, tells the interpreter that there will be no server window.

If *wxnoserver* is set to True, the interpreter will not wait for the server GUI before showing the controller window.

input
PyoObject. Input signal to process.

freq
float or PyoObject. Cutoff or Center frequency of the filter.

bw
float or PyoObject. Bandwidth for bandreject and bandpass filters.

type
int. Filter type {0 = lowpass, 1 = highpass, 2 = bandreject, 3 = bandpass}.

IRAverage

class IRAverage(*input*, *order=256*, *mul=1*, *add=0*)
Moving average filter using circular convolution.

IRAverage uses circular convolution to implement an average filter. This filter is designed to reduce the noise in the input signal while keeping as much as possible the step response of the original signal. User can defined the length, in samples, of the impulse response, also known as the filter kernel. This controls the ratio of removed noise vs the fidelity of the original step response.

Parent *PyoObject*

Args

input: PyoObject Input signal to process.

order: int {even number}, optional Length, in samples, of the filter kernel used for convolution. Available at initialization time only. Defaults to 256.

This value must be even. A high order will reduced more noise and will have a higher damping effect on the step response, but it is also more expensive to compute.

Note: Convolution is very expensive to compute, so the length of the impulse response (the *order* parameter) must be kept very short to run in real time.

```
>>> s = Server().boot()
>>> s.start()
>>> nz = Noise(.05)
>>> a = Sine([300, 400], mul=.25, add=nz)
>>> b = IRAverage(a, order=128).out()
```

setInput(*x*, *fadetime=0.05*)
Replace the *input* attribute.

Args

x: PyoObject New signal to process.

fadetime: float, optional Crossfade time between old and new input. Default to 0.05.

input
PyoObject. Input signal to process.

IRPulse

class IRPulse(*input*, *freq=500*, *bw=2500*, *type=0*, *order=256*, *mul=1*, *add=0*)
Comb-like filter using circular convolution.

IRPulse uses circular convolution to implement standard comb-like filters consisting of an harmonic series with fundamental *freq* and a comb filter with the first notch at *bw* frequency. The *type* parameter defines variations of this pattern. User can defined the length, in samples, of the impulse response, also known as the filter kernel.

Parent *PyoObject*

Args

input: PyoObject Input signal to process.

freq: float or PyoObject, optional Fundamental frequency of the spikes in the filter's spectrum, expressed in Hertz. Defaults to 500.

bw: float or PyoObject, optional Frequency, expressed in Hertz, of the first notch in the comb filtering. Defaults to 2500.

type: int, optional

Filter type. Four possible values :

0. Pulse & comb (default)
1. Pulse & comb & lowpass
2. Pulse (odd harmonics) & comb
3. Pulse (odd harmonics) & comb & lowpass

order: int {even number}, optional Length, in samples, of the filter kernel used for convolution. Available at initialization time only. Defaults to 256.

This value must be even. Higher is the order and sharper is the roll-off of the filter, but it is also more expensive to compute.

Note: Convolution is very expensive to compute, so the length of the impulse response (the *order* parameter) must be kept very short to run in real time.

Note that although *freq* and *bw* can be PyoObjects, the impulse response of the filter is only updated once per buffer size.

```
>>> s = Server().boot()
>>> s.start()
>>> a = Noise(.5)
>>> b = IRPulse(a, freq=[245, 250], bw=2500, type=3, order=256).out()
```

setInput(*x*, *fadetime=0.05*)
Replace the *input* attribute.

Args

x: PyoObject New signal to process.

fadetime: float, optional Crossfade time between old and new input. Default to 0.05.

setFreq(*x*)
Replace the *freq* attribute.

Args

x: float or PyoObject New *freq* attribute.

setBw(*x*)
Replace the *bw* attribute.

Args

x: float or PyoObject New *bw* attribute.

setType(*x*)
Replace the *type* attribute.

Args

x: int

New *type* attribute. Filter type. Four possible values:

0. Pulse & comb (default)
1. Pulse & comb & lowpass
2. Pulse (odd harmonics) & comb
3. Pulse (odd harmonics) & comb & lowpass

ctrl(*map_list=None*, *title=None*, *wxnoserver=False*)
Opens a sliders window to control the parameters of the object. Only parameters that can be set to a PyoObject are allowed to be mapped on a slider.

If a list of values are given to a parameter, a multisliders will be used to control each stream independently.

Args

map_list: list of SLMap objects, optional Users defined set of parameters scaling. There is default scaling for each object that accept *ctrl* method.

title: string, optional Title of the window. If none is provided, the name of the class is used.

wxnoserver: boolean, optional With wxPython graphical toolkit, if True, tells the interpreter that there will be no server window.

If *wxnoserver* is set to True, the interpreter will not wait for the server GUI before showing the controller window.

input
PyoObject. Input signal to process.

freq
float or PyoObject. Cutoff or Center frequency of the filter.

bw
float or PyoObject. Bandwidth for bandreject and bandpass filters.

type
int. Filter type {0 = pulse, 1 = pulse_lp, 2 = pulse_odd, 3 = pulse_odd_lp}.

IRFM

class IRFM(*input*, *carrier=1000*, *ratio=0.5*, *index=3*, *order=256*, *mul=1*, *add=0*)
Filters a signal with a frequency modulation spectrum using circular convolution.

IRFM uses circular convolution to implement filtering with a frequency modulation spectrum. User can defined the length, in samples, of the impulse response, also known as the filter kernel. The higher the *order*, the narrower the bandwidth around each of the FM components.

Parent `PyoObject`

Args

input: PyoObject Input signal to process.

carrier: float or PyoObject, optional Carrier frequency in cycles per second. Defaults to 1000.

ratio: float or PyoObject, optional A factor that, when multiplied by the *carrier* parameter, gives the modulator frequency. Defaults to 0.5.

index: float or PyoObject, optional The modulation index. This value multiplied by the modulator frequency gives the modulator amplitude. Defaults to 3.

order: int {even number}, optional Length, in samples, of the filter kernel used for convolution. Available at initialization time only. Defaults to 256.

This value must be even. Higher is the order and sharper is the roll-off of the filter, but it is also more expensive to compute.

Note: Convolution is very expensive to compute, so the length of the impulse response (the *order* parameter) must be kept very short to run in real time.

Note that although *carrier*, *ratio* and *index* can be PyoObjects, the impulse response of the filter is only updated once per buffer size.

```
>>> s = Server().boot()
>>> s.start()
>>> nz = Noise(.7)
>>> lf = Sine(freq=[.2, .25], mul=.125, add=.5)
>>> b = IRFM(nz, carrier=3500, ratio=lf, index=3, order=256).out()
```

setInput(*x*, *fadetime=0.05*)

Replace the *input* attribute.

Args

x: PyoObject New signal to process.

fadetime: float, optional Crossfade time between old and new input. Default to 0.05.

setCarrier(*x*)

Replace the *carrier* attribute.

Args

x: float or PyoObject New *carrier* attribute.

setRatio(*x*)

Replace the *ratio* attribute.

Args

x: float or PyoObject New *ratio* attribute.

setIndex(*x*)

Replace the *index* attribute.

Args

x: float or PyoObject New *index* attribute.

ctrl(*map_list=None*, *title=None*, *wxnoserver=False*)

Opens a sliders window to control the parameters of the object. Only parameters that can be set to a PyoObject are allowed to be mapped on a slider.

If a list of values are given to a parameter, a multisliders will be used to control each stream independently.

Args

map_list: list of SLMap objects, optional Users defined set of parameters scaling. There is default scaling for each object that accept *ctrl* method.

title: string, optional Title of the window. If none is provided, the name of the class is used.

wxnoserver: boolean, optional With wxPython graphical toolkit, if True, tells the interpreter that there will be no server window.

If *wxnoserver* is set to True, the interpreter will not wait for the server GUI before showing the controller window.

input
PyoObject. Input signal to process.

carrier
float or PyoObject. Carrier frequency in Hz.

ratio
float or PyoObject. Modulator/carrier ratio.

index
float or PyoObject. Modulation index.

SVF

class SVF(*input*, *freq=1000*, *q=1*, *type=0*, *mul=1*, *add=0*)
Fourth-order state variable filter allowing continuous change of the filter type.

Parent *PyoObject*

Args

input: PyoObject Input signal to process.

freq: float or PyoObject, optional Cutoff or center frequency of the filter. Defaults to 1000.

Because this filter becomes unstable at higher frequencies, the *freq* parameter is limited to one-sixth of the sampling rate.

q: float or PyoObject, optional Q of the filter, defined (for bandpass filters) as freq/bandwidth. Should be between 0.5 and 50. Defaults to 1.

type: float or PyoObject, optional This value, in the range 0 to 1, controls the filter type crossfade on the continuum lowpass-bandpass-highpass.

- 0.0 = lowpass (default)
- 0.5 = bandpass
- 1.0 = highpass

```
>>> s = Server().boot()
>>> s.start()
>>> a = Noise(.2)
>>> lf1 = Sine([.22,.25]).range(500, 2000)
>>> lf2 = Sine([.17,.15]).range(1, 4)
>>> lf3 = Sine([.23,.2]).range(0, 1)
>>> b = SVF(a, freq=lf1, q=lf2, type=lf3).out()
```

setInput(*x*, *fadetime=0.05*)
Replace the *input* attribute.

Args

x: PyoObject New signal to process.

fadetime: float, optional Crossfade time between old and new input. Defaults to 0.05.

setFreq(*x*)
Replace the *freq* attribute. Limited in the upper bound to one-sixth of the sampling rate.

Args

x: float or PyoObject New *freq* attribute.

setQ(*x*)
Replace the *q* attribute. Should be between 0.5 and 50.

Args

x: float or PyoObject New *q* attribute.

setType(*x*)
Replace the *type* attribute. Must be in the range 0 to 1.

This value allows the filter type to sweep from a lowpass (0) to a bandpass (0.5) and then, from the bandpass to a highpass (1).

Args

x: float or PyoObject New *type* attribute.

ctrl(*map_list=None*, *title=None*, *wxnoserver=False*)
Opens a sliders window to control the parameters of the object. Only parameters that can be set to a PyoObject are allowed to be mapped on a slider.

If a list of values are given to a parameter, a multisliders will be used to control each stream independently.

Args

map_list: list of SLMap objects, optional Users defined set of parameters scaling. There is default scaling for each object that accept *ctrl* method.

title: string, optional Title of the window. If none is provided, the name of the class is used.

wxnoserver: boolean, optional With wxPython graphical toolkit, if True, tells the interpreter that there will be no server window.

If *wxnoserver* is set to True, the interpreter will not wait for the server GUI before showing the controller window.

input
PyoObject. Input signal to process.

freq
float or PyoObject. Cutoff or center frequency of the filter.

q
float or PyoObject. Q of the filter.

type
float or PyoObject. Crossfade between filter types.

SVF2

class SVF2(*input*, *freq=1000*, *q=1*, *shelf=-3*, *type=0*, *mul=1*, *add=0*)
Second-order state variable filter allowing continuous change of the filter type.

This 2-pole multimode filter is described in the book "The Art of VA Filter Design" by Vadim Zavalishin (version 2.1.0 when this object was created).

Several filter types are available with continuous change between them. The default order (controlled with the *type* argument) is:

- lowpass
- bandpass
- highpass
- highshelf
- bandshelf
- lowshelf
- notch
- peak
- allpass
- unit gain bandpass

The filter types order can be changed with the *setOrder* method. The first filter type is always copied at the end of the order list so we can create a glitch-free loop of filter types with a Phasor given as *type* argument. Ex.:

```
>>> lfo = Phasor(freq=.5, mul=3)
>>> svf2 = SVF2(Noise(.25), freq=2500, q=2, shelf=-6, type=lfo).out()
>>> svf2.order = [2, 1, 0] # highpass -> bandpass -> lowpass -> highpass
```

Parent *PyoObject*

Args

input: PyoObject Input signal to process.

freq: float or PyoObject, optional Cutoff or center frequency of the filter. Defaults to 1000.

q: float or PyoObject, optional Q of the filter, defined (for bandpass filters) as freq/bandwidth. Should be between 0.5 and 50. Defaults to 1.

shelf: float or PyoObject, optional Gain, between -24 dB and 24 dB, used by shelving filters. Defaults to -3.

type: float or PyoObject, optional This value, in the range 0 to 10, controls the filter type crossfade on the continuum defined by the filter types order. The default order (which can be changed with the *setOrder* method) is:

- 0 = lowpass
- 1 = bandpass
- 2 = highpass
- 3 = highshelf
- 4 = bandshelf
- 5 = lowshelf
- 6 = notch
- 7 = peak

- 8 = allpass
- 9 = unit gain bandpass

```
>>> s = Server().boot()
>>> s.start()
>>> n = Noise(0.25).mix(2)
>>> tp = Phasor(.25, mul=10)
>>> fr = Sine(.3).range(500, 5000)
>>> svf2 = SVF2(n, freq=fr, q=3, shelf=-6, type=tp).out()
```

setInput(*x*, *fadetime=0.05*)
Replace the *input* attribute.

Args

x: PyoObject New signal to process.

fadetime: float, optional Crossfade time between old and new input. Defaults to 0.05.

setFreq(*x*)
Replace the *freq* attribute.

Args

x: float or PyoObject New *freq* attribute.

setQ(*x*)
Replace the *q* attribute. Should be between 0.5 and 50.

Args

x: float or PyoObject New *q* attribute.

setShelf(*x*)
Replace the *shelf* attribute. Should be between -24 dB and 24 dB.

Args

x: float or PyoObject New *shelf* attribute.

setType(*x*)
Replace the *type* attribute. Must be in the range 0 to 10.

This value allows the filter type to crossfade between several filter types.

Args

x: float or PyoObject New *type* attribute.

setOrder(*x*)
Change the filter types ordering.

The ordering is a list of one to ten integers indicating the filter types order used by the *type* argument to crossfade between them. Types, as integer, are:

- 0 = lowpass
- 1 = bandpass
- 2 = highpass
- 3 = highshelf
- 4 = bandshelf
- 5 = lowshelf

- 6 = notch
- 7 = peak
- 8 = allpass
- 9 = unit gain bandpass

Args

x: list of ints New types ordering.

ctrl(*map_list=None*, *title=None*, *wxnoserver=False*)
Opens a sliders window to control the parameters of the object. Only parameters that can be set to a PyoObject are allowed to be mapped on a slider.

If a list of values are given to a parameter, a multisliders will be used to control each stream independently.

Args

map_list: list of SLMap objects, optional Users defined set of parameters scaling. There is default scaling for each object that accept *ctrl* method.

title: string, optional Title of the window. If none is provided, the name of the class is used.

wxnoserver: boolean, optional With wxPython graphical toolkit, if True, tells the interpreter that there will be no server window.

If *wxnoserver* is set to True, the interpreter will not wait for the server GUI before showing the controller window.

input
PyoObject. Input signal to process.

freq
float or PyoObject. Cutoff or center frequency of the filter.

q
float or PyoObject. Q of the filter.

shelf
float or PyoObject. Shelving gain of the filter.

type
float or PyoObject. Crossfade between filter types.

order
list of ints. Filter types ordering.

Average

class Average(*input*, *size=10*, *mul=1*, *add=0*)
Moving average filter.

As the name implies, the moving average filter operates by averaging a number of points from the input signal to produce each point in the output signal. In spite of its simplicity, the moving average filter is optimal for a common task: reducing random noise while retaining a sharp step response.

Parent *PyoObject*

Args

input: PyoObject Input signal to process.

size: int, optional Filter kernel size, which is the number of samples used in the moving average. Default to 10.

```
>>> s = Server().boot()
>>> s.start()
>>> a = Noise(.025)
>>> b = Sine(250, mul=0.3)
>>> c = Average(a+b, size=100).out()
```

setInput(*x*, *fadetime=0.05*)
Replace the *input* attribute.

Args

x: PyoObject New signal to process.

fadetime: float, optional Crossfade time between old and new input. Default to 0.05.

setSize(*x*)
Replace the *size* attribute.

Args

x: int New *size* attribute.

ctrl(*map_list=None*, *title=None*, *wxnoserver=False*)
Opens a sliders window to control the parameters of the object. Only parameters that can be set to a PyoObject are allowed to be mapped on a slider.

If a list of values are given to a parameter, a multisliders will be used to control each stream independently.

Args

map_list: list of SLMap objects, optional Users defined set of parameters scaling. There is default scaling for each object that accept *ctrl* method.

title: string, optional Title of the window. If none is provided, the name of the class is used.

wxnoserver: boolean, optional With wxPython graphical toolkit, if True, tells the interpreter that there will be no server window.

If *wxnoserver* is set to True, the interpreter will not wait for the server GUI before showing the controller window.

input
PyoObject. Input signal to process.

size
int. Filter kernel size in samples.

Reson

class Reson(*input*, *freq=1000*, *q=1*, *mul=1*, *add=0*)
A second-order resonant bandpass filter.

Reson implements a classic resonant bandpass filter, as described in:

Dodge, C., Jerse, T., "Computer Music, Synthesis, Composition and Performance".

Reson uses less CPU than the equivalent filter with a Biquad object.

Parent *PyoObject*

Args

input: PyoObject Input signal to process.

freq: float or PyoObject, optional Center frequency of the filter. Defaults to 1000.

q: float or PyoObject, optional Q of the filter, defined as freq/bandwidth. Should be between 1 and 500. Defaults to 1.

```
>>> s = Server().boot()
>>> s.start()
>>> a = Noise(mul=.7)
>>> lfo = Sine(freq=[.2, .25], mul=1000, add=1500)
>>> f = Reson(a, freq=lfo, q=5).out()
```

setInput(*x*, *fadetime=0.05*)
Replace the *input* attribute.

Args

x: PyoObject New signal to process.

fadetime: float, optional Crossfade time between old and new input. Defaults to 0.05.

setFreq(*x*)
Replace the *freq* attribute.

Args

x: float or PyoObject New *freq* attribute.

setQ(*x*)
Replace the *q* attribute. Should be between 1 and 500.

Args

x: float or PyoObject New *q* attribute.

ctrl(*map_list=None*, *title=None*, *wxnoserver=False*)
Opens a sliders window to control the parameters of the object. Only parameters that can be set to a PyoObject are allowed to be mapped on a slider.

If a list of values are given to a parameter, a multisliders will be used to control each stream independently.

Args

map_list: list of SLMap objects, optional Users defined set of parameters scaling. There is default scaling for each object that accept *ctrl* method.

title: string, optional Title of the window. If none is provided, the name of the class is used.

wxnoserver: boolean, optional With wxPython graphical toolkit, if True, tells the interpreter that there will be no server window.

If *wxnoserver* is set to True, the interpreter will not wait for the server GUI before showing the controller window.

input
PyoObject. Input signal to process.

freq
float or PyoObject. Center frequency of the filter.

q
float or PyoObject. Q of the filter.

Resonx

class Resonx(*input*, *freq=1000*, *q=1*, *stages=4*, *mul=1*, *add=0*)
A multi-stages second-order resonant bandpass filter.

Resonx implements a stack of the classic resonant bandpass filter, as described in:

Dodge, C., Jerse, T., "Computer Music, Synthesis, Composition and Performance".

Resonx is equivalent to a filter consisting of more layers of Reson with the same arguments, serially connected. It is faster than using a large number of instances of the Reson object, it uses less memory and allows filters with sharper cutoff.

Parent *PyoObject*

Args

input: PyoObject Input signal to process.

freq: float or PyoObject, optional Center frequency of the filter. Defaults to 1000.

q: float or PyoObject, optional Q of the filter, defined as freq/bandwidth. Should be between 1 and 500. Defaults to 1.

stages: int, optional The number of filtering stages in the filter stack. Defaults to 4.

```
>>> s = Server().boot()
>>> s.start()
>>> a = Noise(mul=.7)
>>> lfo = Sine(freq=[.2, .25], mul=1000, add=1500)
>>> f = Resonx(a, freq=lfo, q=5).out()
```

setInput(*x*, *fadetime=0.05*)
Replace the *input* attribute.

Args

x: PyoObject New signal to process.

fadetime: float, optional Crossfade time between old and new input. Defaults to 0.05.

setFreq(*x*)
Replace the *freq* attribute.

Args

x: float or PyoObject New *freq* attribute.

setQ(*x*)
Replace the *q* attribute. Should be between 1 and 500.

Args

x: float or PyoObject New *q* attribute.

setStages(*x*)
Replace the *stages* attribute.

Args

x: int New *stages* attribute.

ctrl (*map_list=None*, *title=None*, *wxnoserver=False*)
Opens a sliders window to control the parameters of the object. Only parameters that can be set to a PyoObject are allowed to be mapped on a slider.

If a list of values are given to a parameter, a multisliders will be used to control each stream independently.

Args

map_list: list of SLMap objects, optional Users defined set of parameters scaling. There is default scaling for each object that accept *ctrl* method.

title: string, optional Title of the window. If none is provided, the name of the class is used.

wxnoserver: boolean, optional With wxPython graphical toolkit, if True, tells the interpreter that there will be no server window.

If *wxnoserver* is set to True, the interpreter will not wait for the server GUI before showing the controller window.

input
PyoObject. Input signal to process.

freq
float or PyoObject. Center frequency of the filter.

q
float or PyoObject. Q of the filter.

stages
int. The number of filtering stages.

ButLP

class ButLP (*input*, *freq=1000*, *mul=1*, *add=0*)
A second-order Butterworth lowpass filter.

ButLP implements a second-order IIR Butterworth lowpass filter, which has a maximally flat passband and a very good precision and stopband attenuation.

Parent *PyoObject*

Args

input: PyoObject Input signal to process.

freq: float or PyoObject, optional Cutoff frequency of the filter in hertz. Default to 1000.

```
>>> s = Server().boot()
>>> s.start()
>>> n = Noise(.3)
>>> lf = Sine(freq=.2, mul=800, add=1000)
>>> f = ButLP(n, lf).mix(2).out()
```

setInput (*x*, *fadetime=0.05*)
Replace the *input* attribute.

Args

x: PyoObject New signal to process.

fadetime: float, optional Crossfade time between old and new input. Default to 0.05.

setFreq(*x*)
Replace the *freq* attribute.

Args

x: float or PyoObject New *freq* attribute.

ctrl(*map_list=None*, *title=None*, *wxnoserver=False*)
Opens a sliders window to control the parameters of the object. Only parameters that can be set to a PyoObject are allowed to be mapped on a slider.

If a list of values are given to a parameter, a multisliders will be used to control each stream independently.

Args

map_list: list of SLMap objects, optional Users defined set of parameters scaling. There is default scaling for each object that accept *ctrl* method.

title: string, optional Title of the window. If none is provided, the name of the class is used.

wxnoserver: boolean, optional With wxPython graphical toolkit, if True, tells the interpreter that there will be no server window.

If *wxnoserver* is set to True, the interpreter will not wait for the server GUI before showing the controller window.

input
PyoObject. Input signal to process.

freq
float or PyoObject. Cutoff frequency of the filter.

ButHP

class ButHP(*input*, *freq=1000*, *mul=1*, *add=0*)
A second-order Butterworth highpass filter.

ButHP implements a second-order IIR Butterworth highpass filter, which has a maximally flat passband and a very good precision and stopband attenuation.

Parent *PyoObject*

Args

input: PyoObject Input signal to process.

freq: float or PyoObject, optional Cutoff frequency of the filter in hertz. Default to 1000.

```
>>> s = Server().boot()
>>> s.start()
>>> n = Noise(.2)
>>> lf = Sine(freq=.2, mul=1500, add=2500)
>>> f = ButHP(n, lf).mix(2).out()
```

setInput(*x*, *fadetime=0.05*)
Replace the *input* attribute.

Args

x: PyoObject New signal to process.

fadetime: float, optional Crossfade time between old and new input. Default to 0.05.

setFreq(*x*)
Replace the *freq* attribute.

Args

x: float or PyoObject New *freq* attribute.

ctrl(*map_list=None*, *title=None*, *wxnoserver=False*)
Opens a sliders window to control the parameters of the object. Only parameters that can be set to a PyoObject are allowed to be mapped on a slider.

If a list of values are given to a parameter, a multisliders will be used to control each stream independently.

Args

map_list: list of SLMap objects, optional Users defined set of parameters scaling. There is default scaling for each object that accept *ctrl* method.

title: string, optional Title of the window. If none is provided, the name of the class is used.

wxnoserver: boolean, optional With wxPython graphical toolkit, if True, tells the interpreter that there will be no server window.

If *wxnoserver* is set to True, the interpreter will not wait for the server GUI before showing the controller window.

input
PyoObject. Input signal to process.

freq
float or PyoObject. Cutoff frequency of the filter.

ButBP

class ButBP(*input*, *freq=1000*, *q=1*, *mul=1*, *add=0*)
A second-order Butterworth bandpass filter.

ButBP implements a second-order IIR Butterworth bandpass filter, which has a maximally flat passband and a very good precision and stopband attenuation.

Parent *PyoObject*

Args

input: PyoObject Input signal to process.

freq: float or PyoObject, optional Center frequency of the filter. Defaults to 1000.

q: float or PyoObject, optional Q of the filter, defined as freq/bandwidth. Should be between 1 and 500. Defaults to 1.

```
>>> s = Server().boot()
>>> s.start()
>>> a = Noise(mul=.7)
>>> lfo = Sine(freq=[.2, .25], mul=1000, add=1500)
>>> f = ButBP(a, freq=lfo, q=5).out()
```

setInput(*x*, *fadetime=0.05*)
Replace the *input* attribute.

Args

x: PyoObject New signal to process.

fadetime: float, optional Crossfade time between old and new input. Defaults to 0.05.

setFreq(*x*)
Replace the *freq* attribute.

Args

x: float or PyoObject New *freq* attribute.

setQ(*x*)
Replace the *q* attribute. Should be between 1 and 500.

Args

x: float or PyoObject New *q* attribute.

ctrl(*map_list=None*, *title=None*, *wxnoserver=False*)
Opens a sliders window to control the parameters of the object. Only parameters that can be set to a PyoObject are allowed to be mapped on a slider.

If a list of values are given to a parameter, a multisliders will be used to control each stream independently.

Args

map_list: list of SLMap objects, optional Users defined set of parameters scaling. There is default scaling for each object that accept *ctrl* method.

title: string, optional Title of the window. If none is provided, the name of the class is used.

wxnoserver: boolean, optional With wxPython graphical toolkit, if True, tells the interpreter that there will be no server window.

If *wxnoserver* is set to True, the interpreter will not wait for the server GUI before showing the controller window.

input
PyoObject. Input signal to process.

freq
float or PyoObject. Center frequency of the filter.

q
float or PyoObject. Q of the filter.

ButBR

class ButBR(*input*, *freq=1000*, *q=1*, *mul=1*, *add=0*)
A second-order Butterworth band-reject filter.

ButBR implements a second-order IIR Butterworth band-reject filter, which has a maximally flat passband and a very good precision and stopband attenuation.

Parent *PyoObject*

Args

input: PyoObject Input signal to process.

freq: float or PyoObject, optional Center frequency of the filter. Defaults to 1000.

q: float or PyoObject, optional Q of the filter, defined as freq/bandwidth. Should be between 1 and 500. Defaults to 1.

```
>>> s = Server().boot()
>>> s.start()
>>> a = Noise(mul=.2)
>>> lfo = Sine(freq=[.2, .25], mul=1000, add=1500)
>>> f = ButBR(a, freq=lfo, q=1).out()
```

setInput(*x*, *fadetime=0.05*)
Replace the *input* attribute.

Args

x: PyoObject New signal to process.

fadetime: float, optional Crossfade time between old and new input. Defaults to 0.05.

setFreq(*x*)
Replace the *freq* attribute.

Args

x: float or PyoObject New *freq* attribute.

setQ(*x*)
Replace the *q* attribute. Should be between 1 and 500.

Args

x: float or PyoObject New *q* attribute.

ctrl(*map_list=None*, *title=None*, *wxnoserver=False*)
Opens a sliders window to control the parameters of the object. Only parameters that can be set to a PyoObject are allowed to be mapped on a slider.

If a list of values are given to a parameter, a multisliders will be used to control each stream independently.

Args

map_list: list of SLMap objects, optional Users defined set of parameters scaling. There is default scaling for each object that accept *ctrl* method.

title: string, optional Title of the window. If none is provided, the name of the class is used.

wxnoserver: boolean, optional With wxPython graphical toolkit, if True, tells the interpreter that there will be no server window.

If *wxnoserver* is set to True, the interpreter will not wait for the server GUI before showing the controller window.

input
PyoObject. Input signal to process.

freq
float or PyoObject. Center frequency of the filter.

q
float or PyoObject. Q of the filter.

ComplexRes

class ComplexRes(*input*, *freq=1000*, *decay=0.25*, *mul=1*, *add=0*)

Complex one-pole resonator filter.

ComplexRes implements a resonator derived from a complex multiplication, which is very similar to a digital filter.

Parent *PyoObject*

Args

input: PyoObject Input signal to process.

freq: float or PyoObject, optional Center frequency of the filter. Defaults to 1000.

decay: float or PyoObject, optional Decay time, in seconds, for the filter's response. Defaults to 0.25.

```
>>> s = Server().boot()
>>> s.start()
>>> env = HannTable()
>>> trigs = Metro(.2, poly=4).play()
>>> amp = TrigEnv(trigs, table=env, dur=0.005, mul=2)
>>> im = Noise(mul=amp)
>>> res = ComplexRes(im, freq=[950,530,780,1490], decay=1).out()
```

setInput(*x*, *fadetime=0.05*)

Replace the *input* attribute.

Args

x: PyoObject New signal to process.

fadetime: float, optional Crossfade time between old and new input. Defaults to 0.05.

setFreq(*x*)

Replace the *freq* attribute.

Args

x: float or PyoObject New *freq* attribute.

setDecay(*x*)

Replace the *decay* attribute.

Args

x: float or PyoObject New *decay* attribute.

ctrl(*map_list=None*, *title=None*, *wxnoserver=False*)

Opens a sliders window to control the parameters of the object. Only parameters that can be set to a PyoObject are allowed to be mapped on a slider.

If a list of values are given to a parameter, a multisliders will be used to control each stream independently.

Args

map_list: list of SLMap objects, optional Users defined set of parameters scaling. There is default scaling for each object that accept *ctrl* method.

title: string, optional Title of the window. If none is provided, the name of the class is used.

wxnoserver: boolean, optional With wxPython graphical toolkit, if True, tells the interpreter that there will be no server window.

If *wxnoserver* is set to True, the interpreter will not wait for the server GUI before showing the controller window.

input
PyoObject. Input signal to filter.

freq
float or PyoObject. Center frequency of the filter.

decay
float or PyoObject. Decay time of the filter's response.

MoogLP

class MoogLP(*input*, *freq=1000*, *res=0*, *mul=1*, *add=0*)
A fourth-order resonant lowpass filter.

Digital approximation of the Moog VCF, giving a decay of 24dB/oct.

Parent *PyoObject*

Args

input: PyoObject Input signal to process.

freq: float or PyoObject, optional Cutoff frequency of the filter. Defaults to 1000.

res: float or PyoObject, optional Amount of Resonance of the filter, usually between 0 (no resonance) and 1 (medium resonance). Self-oscillation occurs when the resonance is >= 1. Can go up to 10. Defaults to 0.

```
>>> s = Server().boot()
>>> s.start()
>>> ph = Phasor(40)
>>> sqr = Round(ph, add=-0.5)
>>> lfo = Sine(freq=[.4, .5], mul=2000, add=2500)
>>> fil = MoogLP(sqr, freq=lfo, res=1.25).out()
```

setInput(*x*, *fadetime=0.05*)
Replace the *input* attribute.

Args

x: PyoObject New signal to process.

fadetime: float, optional Crossfade time between old and new input. Defaults to 0.05.

setFreq(*x*)
Replace the *freq* attribute.

Args

x: float or PyoObject New *freq* attribute.

setRes(*x*)
Replace the *res* attribute.

Args

x: float or PyoObject New *res* attribute.

ctrl(*map_list=None*, *title=None*, *wxnoserver=False*)
Opens a sliders window to control the parameters of the object. Only parameters that can be set to a PyoObject are allowed to be mapped on a slider.

If a list of values are given to a parameter, a multisliders will be used to control each stream independently.

Args

map_list: list of SLMap objects, optional Users defined set of parameters scaling. There is default scaling for each object that accept *ctrl* method.

title: string, optional Title of the window. If none is provided, the name of the class is used.

wxnoserver: boolean, optional With wxPython graphical toolkit, if True, tells the interpreter that there will be no server window.

If *wxnoserver* is set to True, the interpreter will not wait for the server GUI before showing the controller window.

input
PyoObject. Input signal to process.

freq
float or PyoObject. Cutoff frequency of the filter.

res
float or PyoObject. Amount of resonance of the filter.

Fast Fourier Transform

A Fast Fourier Transform (FFT) is an efficient algorithm to compute the discrete Fourier transform (DFT) and its inverse (IFFT).

The objects below can be used to perform sound processing in the spectral domain.

FFT

class FFT(*input*, *size=1024*, *overlaps=4*, *wintype=2*)
Fast Fourier Transform.

FFT analyses an input signal and converts it into the spectral domain. Three audio signals are sent out of the object, the *real* part, from bin 0 (DC) to bin size/2 (Nyquist), the *imaginary* part, from bin 0 to bin size/2-1, and the bin number, an increasing count from 0 to size-1. *real* and *imaginary* buffer's left samples up to size-1 are filled with zeros. See notes below for an example of how to retrieve each signal component.

Parent `PyoObject`

Args

input: PyoObject Input signal to process.

size: int {pow-of-two > 4}, optional FFT size. Must be a power of two greater than 4. The FFT size is the number of samples used in each analysis frame. Defaults to 1024.

overlaps: int, optional The number of overlaped analysis block. Must be a positive integer. More overlaps can greatly improved sound quality synthesis but it is also more CPU expensive. Defaults to 4.

wintype: int, optional Shape of the envelope used to filter each input frame. Possible shapes are :

0. rectangular (no windowing)
1. Hamming
2. Hanning
3. Bartlett (triangular)
4. Blackman 3-term
5. Blackman-Harris 4-term
6. Blackman-Harris 7-term
7. Tuckey (alpha = 0.66)
8. Sine (half-sine window)

Note: FFT has no *out* method. Signal must be converted back to time domain, with IFFT, before being sent to output.

FFT has no *mul* and *add* attributes.

Real, imaginary and bin_number parts are three separated set of audio streams. The user should call :

FFT['real'] to retrieve the real part.
FFT['imag'] to retrieve the imaginary part.
FFT['bin'] to retrieve the bin number part.

```
>>> s = Server().boot()
>>> s.start()
>>> a = Noise(.25).mix(2)
>>> fin = FFT(a, size=1024, overlaps=4, wintype=2)
>>> t = ExpTable([(0,0),(3,0),(10,1),(20,0),(30,.8),(50,0),(70,.6),(150,0),(512,
↪0)], size=512)
>>> amp = TableIndex(t, fin["bin"])
>>> re = fin["real"] * amp
>>> im = fin["imag"] * amp
>>> fout = IFFT(re, im, size=1024, overlaps=4, wintype=2).mix(2).out()
```

get (*identifier='real'*, *all=False*)

Return the first sample of the current buffer as a float.

Can be used to convert audio stream to usable Python data.

"real", "imag" or "bin" must be given to *identifier* to specify which stream to get value from.

Args

identifier: string {"real", "imag", "bin"} Address string parameter identifying audio stream. Defaults to "real".

all: boolean, optional If True, the first value of each object's stream will be returned as a list. Otherwise, only the value of the first object's stream will be returned as a float. Defaults to False.

setInput (*x*, *fadetime=0.05*)

Replace the *input* attribute.

Args

x: PyoObject New signal to process.

fadetime: float, optional Crossfade time between old and new input. Default to 0.05.

play(*dur=0*, *delay=0*)

Start processing without sending samples to output. This method is called automatically at the object creation.

This method returns *self*, allowing it to be applied at the object creation.

Args

dur: float, optional Duration, in seconds, of the object's activation. The default is 0 and means infinite duration.

delay: float, optional Delay, in seconds, before the object's activation. Defaults to 0.

stop(*wait=0*)

Stop processing.

This method returns *self*, allowing it to be applied at the object creation.

Args

wait: float, optional Delay, in seconds, before the process is actually stopped. If autoStartChildren is activated in the Server, this value is propagated to the children objects. Defaults to 0.

Note: if the method setStopDelay(x) was called before calling stop(wait) with a positive *wait* value, the *wait* value won't overwrite the value given to setStopDelay for the current object, but will be the one propagated to children objects. This allow to set a waiting time for a specific object with setStopDelay whithout changing the global delay time given to the stop method.

Fader and Adsr objects ignore waiting time given to the stop method because they already implement a delayed processing triggered by the stop call.

---

out(*chnl=0*, *inc=1*, *dur=0*, *delay=0*)

Start processing and send samples to audio output beginning at *chnl*.

This method returns *self*, allowing it to be applied at the object creation.

Args

chnl: int, optional Physical output assigned to the first audio stream of the object. Defaults to 0.

inc: int, optional Output channel increment value. Defaults to 1.

dur: float, optional Duration, in seconds, of the object's activation. The default is 0 and means infinite duration.

delay: float, optional Delay, in seconds, before the object's activation. Defaults to 0.

If *chnl* >= 0, successive streams increment the output number by *inc* and wrap around the global number of channels.

If *chnl* is negative, streams begin at 0, increment the output number by *inc* and wrap around the global number of channels. Then, the list of streams is scrambled.

If *chnl* is a list, successive values in the list will be assigned to successive streams.

setSize(*x*)
Replace the *size* attribute.

Args

x: int new *size* attribute.

setWinType(*x*)
Replace the *wintype* attribute.

Args

x: int new *wintype* attribute.

input
PyoObject. Input signal to process.

size
int. FFT size.

wintype
int. Windowing method.

IFFT

class IFFT(*inreal*, *inimag*, *size=1024*, *overlaps=4*, *wintype=2*, *mul=1*, *add=0*)
Inverse Fast Fourier Transform.

IFFT takes a signal in the spectral domain and converts it to a real audio signal using an inverse fast fourier transform. IFFT takes two signals in input, the *real* and *imaginary* parts of an FFT analysis and returns the corresponding real signal. These signals must correspond to *real* and *imaginary* parts from an FFT object.

Parent `PyoObject`

Args

inreal: PyoObject Input *real* signal.

inimag: PyoObject Input *imaginary* signal.

size: int {pow-of-two > 4}, optional FFT size. Must be a power of two greater than 4. The FFT size is the number of samples used in each analysis frame. This value must match the *size* attribute of the former FFT object. Defaults to 1024.

overlaps: int, optional The number of overlaped analysis block. Must be a positive integer. More overlaps can greatly improved sound quality synthesis but it is also more CPU expensive. This value must match the *overlaps* atribute of the former FFT object. Defaults to 4.

wintype: int, optional Shape of the envelope used to filter each output frame. Possible shapes are :

0. rectangular (no windowing)
1. Hamming
2. Hanning
3. Bartlett (triangular)
4. Blackman 3-term
5. Blackman-Harris 4-term

6. Blackman-Harris 7-term
7. Tuckey (alpha = 0.66)
8. Sine (half-sine window)

Note: The number of streams in *inreal* and *inimag* attributes must be egal to the output of the former FFT object. In most case, it will be *channels of processed sound * overlaps*.

The output of IFFT must be mixed to reconstruct the real signal from the overlapped streams. It is left to the user to call the mix(channels of the processed sound) method on an IFFT object.

```
>>> s = Server().boot()
>>> s.start()
>>> a = Noise(.25).mix(2)
>>> fin = FFT(a, size=1024, overlaps=4, wintype=2)
>>> t = ExpTable([(0,0),(3,0),(10,1),(20,0),(30,.8),(50,0),(70,.6),(150,0),(512,
↪0)], size=512)
>>> amp = TableIndex(t, fin["bin"])
>>> re = fin["real"] * amp
>>> im = fin["imag"] * amp
>>> fout = IFFT(re, im, size=1024, overlaps=4, wintype=2).mix(2).out()
```

setInReal(*x*, *fadetime=0.05*)
Replace the *inreal* attribute.

Args

x: PyoObject New input *real* signal.

fadetime: float, optional Crossfade time between old and new input. Default to 0.05.

setInImag(*x*, *fadetime=0.05*)
Replace the *inimag* attribute.

Args

x: PyoObject New input *imag* signal.

fadetime: float, optional Crossfade time between old and new input. Default to 0.05.

setSize(*x*)
Replace the *size* attribute.

Args

x: int new *size* attribute.

setWinType(*x*)
Replace the *wintype* attribute.

Args

x: int new *wintype* attribute.

ctrl(*map_list=None*, *title=None*, *wxnoserver=False*)
Opens a sliders window to control the parameters of the object. Only parameters that can be set to a PyoObject are allowed to be mapped on a slider.

If a list of values are given to a parameter, a multisliders will be used to control each stream independently.

Args

map_list: list of SLMap objects, optional Users defined set of parameters scaling. There is default scaling for each object that accept *ctrl* method.

title: string, optional Title of the window. If none is provided, the name of the class is used.

wxnoserver: boolean, optional With wxPython graphical toolkit, if True, tells the interpreter that there will be no server window.

If *wxnoserver* is set to True, the interpreter will not wait for the server GUI before showing the controller window.

inreal
PyoObject. Real input signal.

inimag
PyoObject. Imaginary input signal.

size
int. FFT size.

wintype
int. Windowing method.

PolToCar

class PolToCar(*inmag*, *inang*, *mul=1*, *add=0*)
Performs the polar to cartesian conversion.

The Polar system locates the point by measuring the straight line distance, usually denoted by R, from the origin to the point and the angle of an imaginary line from the origin to the point measured counterclockwise from the positive X axis.

The Cartesian system locates points on a plane by measuring the horizontal and vertical distances from an arbitrary origin to a point. These are usually denoted as a pair of values (X,Y).

Parent *PyoObject*

Args

inmag: PyoObject Magintude input signal.

inang: PyoObject Angle input signal.

Note: Cartesians coordinates can be retrieve by calling :

PolToCar['real'] to retrieve the real part.
CarToPol['imag'] to retrieve the imaginary part.

PolToCar has no *out* method. Signal must be converted back to time domain, with IFFT, before being sent to output.

```
>>> s = Server().boot()
>>> snd1 = SfPlayer(SNDS_PATH+"/transparent.aif", loop=True, mul=.7).mix(2)
>>> snd2 = FM(carrier=[75,100,125,150], ratio=[.499,.5,.501,.502], index=20, mul=.
↪1).mix(2)
```

(continues on next page)

(continued from previous page)

```
>>> fin1 = FFT(snd1, size=1024, overlaps=4)
>>> fin2 = FFT(snd2, size=1024, overlaps=4)
>>> # get magnitudes and phases of input sounds
>>> pol1 = CarToPol(fin1["real"], fin1["imag"])
>>> pol2 = CarToPol(fin2["real"], fin2["imag"])
>>> # times magnitudes and adds phases
>>> mag = pol1["mag"] * pol2["mag"] * 100
>>> pha = pol1["ang"] + pol2["ang"]
>>> # converts back to rectangular
>>> car = PolToCar(mag, pha)
>>> fout = IFFT(car["real"], car["imag"], size=1024, overlaps=4).mix(2).out()
>>> s.start()
```

get (*identifier='real'*, *all=False*)
Return the first sample of the current buffer as a float.

Can be used to convert audio stream to usable Python data.

"real" or "imag" must be given to *identifier* to specify which stream to get value from.

Args

identifier: string {"real", "imag"} Address string parameter identifying audio stream. Defaults to "mag".

all: boolean, optional If True, the first value of each object's stream will be returned as a list. Otherwise, only the value of the first object's stream will be returned as a float. Defaults to False.

setInMag (*x*, *fadetime=0.05*)
Replace the *inmag* attribute.

Args

x: PyoObject New signal to process.

fadetime: float, optional Crossfade time between old and new input. Default to 0.05.

setInAng (*x*, *fadetime=0.05*)
Replace the *inang* attribute.

Args

x: PyoObject New signal to process.

fadetime: float, optional Crossfade time between old and new input. Default to 0.05.

inmag
PyoObject. Magnitude input signal.

inang
PyoObject. Angle input signal.

CarToPol

class CarToPol (*inreal*, *inimag*, *mul=1*, *add=0*)
Performs the cartesian to polar conversion.

The Cartesian system locates points on a plane by measuring the horizontal and vertical distances from an arbitrary origin to a point. These are usually denoted as a pair of values (X,Y).

The Polar system locates the point by measuring the straight line distance, usually denoted by R, from the origin to the point and the angle of an imaginary line from the origin to the point measured counterclockwise from the positive X axis.

Parent *PyoObject*

Args

inreal: PyoObject Real input signal.

inimag: PyoObject Imaginary input signal.

Note: Polar coordinates can be retrieve by calling :

CarToPol['mag'] to retrieve the magnitude part.
CarToPol['ang'] to retrieve the angle part.

CarToPol has no *out* method. Signal must be converted back to time domain, with IFFT, before being sent to output.

```
>>> s = Server().boot()
>>> snd1 = SfPlayer(SNDS_PATH+"/transparent.aif", loop=True, mul=.7).mix(2)
>>> snd2 = FM(carrier=[75,100,125,150], ratio=[.499,.5,.501,.502], index=20, mul=.
↪1).mix(2)
>>> fin1 = FFT(snd1, size=1024, overlaps=4)
>>> fin2 = FFT(snd2, size=1024, overlaps=4)
>>> # get magnitudes and phases of input sounds
>>> pol1 = CarToPol(fin1["real"], fin1["imag"])
>>> pol2 = CarToPol(fin2["real"], fin2["imag"])
>>> # times magnitudes and adds phases
>>> mag = pol1["mag"] * pol2["mag"] * 100
>>> pha = pol1["ang"] + pol2["ang"]
>>> # converts back to rectangular
>>> car = PolToCar(mag, pha)
>>> fout = IFFT(car["real"], car["imag"], size=1024, overlaps=4).mix(2).out()
>>> s.start()
```

get(*identifier='mag'*, *all=False*)

Return the first sample of the current buffer as a float.

Can be used to convert audio stream to usable Python data.

"mag" or "ang" must be given to *identifier* to specify which stream to get value from.

Args

identifier: string {"mag", "ang"} Address string parameter identifying audio stream. Defaults to "mag".

all: boolean, optional If True, the first value of each object's stream will be returned as a list. Otherwise, only the value of the first object's stream will be returned as a float. Defaults to False.

setInReal(*x*, *fadetime=0.05*)

Replace the *inreal* attribute.

Args

x: PyoObject New signal to process.

fadetime: float, optional Crossfade time between old and new input. Default to 0.05.

setInImag(*x*, *fadetime=0.05*)
Replace the *inimag* attribute.

Args

x: PyoObject New signal to process.

fadetime: float, optional Crossfade time between old and new input. Default to 0.05.

inreal
PyoObject. Real input signal.

inimag
PyoObject. Imaginary input signal.

FrameAccum

class FrameAccum(*input*, *framesize=1024*, *overlaps=4*, *mul=1*, *add=0*)
Accumulates the phase differences between successive frames.

The difference between the phase values of successive FFT frames for a given bin determines the exact frequency of the energy centered in that bin. This is often known as the phase difference (and sometimes also referred to as phase derivative or instantaneous frequency if it's been subjected to a few additional calculations).

In order to reconstruct a plausible playback of re-ordered FFT frames, we need to calculate the phase difference between successive frames, with FrameDelta, and use it to construct a *running phase* (by simply summing the successive differences) for the output FFT frames.

Parent *PyoObject*

Args

input: PyoObject Phase input signal.

framesize: int, optional Frame size in samples. Usually same as the FFT size. Defaults to 1024.

overlaps: int, optional Number of overlaps in incomming signal. Usually the same as the FFT overlaps. Defaults to 4.

Note: FrameAccum has no *out* method. Signal must be converted back to time domain, with IFFT, before being sent to output.

```
>>> s = Server().boot()
>>> s.start()
>>> snd = SNDS_PATH + '/transparent.aif'
>>> size, hop = 1024, 256
>>> nframes = sndinfo(snd)[0] / size
>>> a = SfPlayer(snd, mul=.3)
>>> m_mag = [NewMatrix(width=size, height=nframes) for i in range(4)]
>>> m_pha = [NewMatrix(width=size, height=nframes) for i in range(4)]
>>> fin = FFT(a, size=size, overlaps=4)
>>> pol = CarToPol(fin["real"], fin["imag"])
>>> delta = FrameDelta(pol["ang"], framesize=size, overlaps=4)
>>> m_mag_rec = MatrixRec(pol["mag"], m_mag, 0, [i*hop for i in range(4)]).play()
```

(continues on next page)

(continued from previous page)

```
>>> m_pha_rec = MatrixRec(delta, m_pha, 0, [i*hop for i in range(4)]).play()
>>> m_mag_read = MatrixPointer(m_mag, fin["bin"]/size, Sine(freq=0.25, mul=.5,␣
↪add=.5))
>>> m_pha_read = MatrixPointer(m_pha, fin["bin"]/size, Sine(freq=0.25, mul=.5,␣
↪add=.5))
>>> accum = FrameAccum(m_pha_read, framesize=size, overlaps=4)
>>> car = PolToCar(m_mag_read, accum)
>>> fout = IFFT(car["real"], car["imag"], size=size, overlaps=4).mix(1).out()
>>> right = Delay(fout, delay=0.013).out(1)
```

out (*chnl=0*, *inc=1*, *dur=0*, *delay=0*)

Start processing and send samples to audio output beginning at *chnl*.

This method returns *self*, allowing it to be applied at the object creation.

Args

chnl: int, optional Physical output assigned to the first audio stream of the object. Defaults to 0.

inc: int, optional Output channel increment value. Defaults to 1.

dur: float, optional Duration, in seconds, of the object's activation. The default is 0 and means infinite duration.

delay: float, optional Delay, in seconds, before the object's activation. Defaults to 0.

If *chnl* >= 0, successive streams increment the output number by *inc* and wrap around the global number of channels.

If *chnl* is negative, streams begin at 0, increment the output number by *inc* and wrap around the global number of channels. Then, the list of streams is scrambled.

If *chnl* is a list, successive values in the list will be assigned to successive streams.

setInput (*x*, *fadetime=0.05*)

Replace the *input* attribute.

Args

x: PyoObject New signal to process.

fadetime: float, optional Crossfade time between old and new input. Default to 0.05.

setFrameSize (*x*)

Replace the *framesize* attribute.

Args

x: int new *framesize* attribute.

input

PyoObject. Phase input signal.

framesize

PyoObject. Frame size in samples.

FrameDelta

class FrameDelta (*input*, *framesize=1024*, *overlaps=4*, *mul=1*, *add=0*)

Computes the phase differences between successive frames.

The difference between the phase values of successive FFT frames for a given bin determines the exact frequency of the energy centered in that bin. This is often known as the phase difference (and sometimes also referred to as phase derivative or instantaneous frequency if it's been subjected to a few additional calculations).

In order to reconstruct a plausible playback of re-ordered FFT frames, we need to calculate the phase difference between successive frames and use it to construct a *running phase* (by simply summing the successive differences with FrameAccum) for the output FFT frames.

Parent *PyoObject*

Args

input: PyoObject Phase input signal, usually from an FFT analysis.

framesize: int, optional Frame size in samples. Usually the same as the FFT size. Defaults to 1024.

overlaps: int, optional Number of overlaps in incomming signal. Usually the same as the FFT overlaps. Defaults to 4.

Note: FrameDelta has no *out* method. Signal must be converted back to time domain, with IFFT, before being sent to output.

```
>>> s = Server().boot()
>>> s.start()
>>> snd = SNDS_PATH + '/transparent.aif'
>>> size, hop = 1024, 256
>>> nframes = sndinfo(snd)[0] / size
>>> a = SfPlayer(snd, mul=.3)
>>> m_mag = [NewMatrix(width=size, height=nframes) for i in range(4)]
>>> m_pha = [NewMatrix(width=size, height=nframes) for i in range(4)]
>>> fin = FFT(a, size=size, overlaps=4)
>>> pol = CarToPol(fin["real"], fin["imag"])
>>> delta = FrameDelta(pol["ang"], framesize=size, overlaps=4)
>>> m_mag_rec = MatrixRec(pol["mag"], m_mag, 0, [i*hop for i in range(4)]).play()
>>> m_pha_rec = MatrixRec(delta, m_pha, 0, [i*hop for i in range(4)]).play()
>>> m_mag_read = MatrixPointer(m_mag, fin["bin"]/size, Sine(freq=0.25, mul=.5,␣
↪add=.5))
>>> m_pha_read = MatrixPointer(m_pha, fin["bin"]/size, Sine(freq=0.25, mul=.5,␣
↪add=.5))
>>> accum = FrameAccum(m_pha_read, framesize=size, overlaps=4)
>>> car = PolToCar(m_mag_read, accum)
>>> fout = IFFT(car["real"], car["imag"], size=size, overlaps=4).mix(1).out()
>>> right = Delay(fout, delay=0.013).out(1)
```

out(*chnl=0*, *inc=1*, *dur=0*, *delay=0*)

Start processing and send samples to audio output beginning at *chnl*.

This method returns *self*, allowing it to be applied at the object creation.

Args

chnl: int, optional Physical output assigned to the first audio stream of the object. Defaults to 0.

inc: int, optional Output channel increment value. Defaults to 1.

dur: float, optional Duration, in seconds, of the object's activation. The default is 0 and means infinite duration.

delay: float, optional Delay, in seconds, before the object's activation. Defaults to 0.

If *chnl* >= 0, successive streams increment the output number by *inc* and wrap around the global number of channels.

If *chnl* is negative, streams begin at 0, increment the output number by *inc* and wrap around the global number of channels. Then, the list of streams is scrambled.

If *chnl* is a list, successive values in the list will be assigned to successive streams.

setInput(*x*, *fadetime=0.05*)
Replace the *input* attribute.

Args

x: PyoObject New signal to process.

fadetime: float, optional Crossfade time between old and new input. Default to 0.05.

setFrameSize(*x*)
Replace the *framesize* attribute.

Args

x: int new *framesize* attribute.

input
PyoObject. Phase input signal.

framesize
PyoObject. Frame size in samples.

CvlVerb

class CvlVerb(*input*, *impulse='/Library/Frameworks/Python.framework/Versions/3.7/lib/python3.7/site-packages/pyo-0.9.2-py3.7-macosx-10.9-x86_64.egg/pyo/lib/snds/IRMediumHallStereo.wav'*, *bal=0.25*, *size=1024*, *mul=1*, *add=0*)
Convolution based reverb.

CvlVerb implements convolution based on a uniformly partitioned overlap-save algorithm. This object can be used to convolve an input signal with an impulse response soundfile to simulate real acoustic spaces.

Parent *PyoObject*

Args

input: PyoObject Input signal to process.

impulse: string, optional Path to the impulse response soundfile. The file must have the same sampling rate as the server to get the proper convolution. Available at initialization time only. Defaults to 'IRMediumHallStereo.wav', located in pyo SNDS_PATH folder.

size: int {pow-of-two}, optional The size in samples of each partition of the impulse file. Small size means smaller latency but more computation time. If not a power-of-2, the object will find the next power-of-2 greater and use that as the actual partition size. This value must also be greater or equal than the server's buffer size. Available at initialization time only. Defaults to 1024.

bal: float or PyoObject, optional Balance between wet and dry signal, between 0 and 1. 0 means no reverb. Defaults to 0.25.

```
>>> s = Server().boot()
>>> s.start()
>>> sf = SfPlayer(SNDS_PATH+"/transparent.aif", loop=True, mul=0.5)
>>> cv = CvlVerb(sf, SNDS_PATH+"/IRMediumHallStereo.wav", size=1024, bal=0.4).
↪out()
```

setInput (*x*, *fadetime=0.05*)
Replace the *input* attribute.

Args

x: PyoObject New signal to process.

fadetime: float, optional Crossfade time between old and new input. Default to 0.05.

setBal (*x*)
Replace the *bal* attribute.

Args

x: float or PyoObject new *bal* attribute.

ctrl (*map_list=None*, *title=None*, *wxnoserver=False*)
Opens a sliders window to control the parameters of the object. Only parameters that can be set to a PyoObject are allowed to be mapped on a slider.

If a list of values are given to a parameter, a multisliders will be used to control each stream independently.

Args

map_list: list of SLMap objects, optional Users defined set of parameters scaling. There is default scaling for each object that accept *ctrl* method.

title: string, optional Title of the window. If none is provided, the name of the class is used.

wxnoserver: boolean, optional With wxPython graphical toolkit, if True, tells the interpreter that there will be no server window.

If *wxnoserver* is set to True, the interpreter will not wait for the server GUI before showing the controller window.

input
PyoObject. Input signal to process.

bal
float or PyoObject. Wet / dry balance.

Vectral

class Vectral (*input*, *framesize=1024*, *overlaps=4*, *up=1.0*, *down=0.7*, *damp=0.9*, *mul=1*, *add=0*)
Performs magnitude smoothing between successive frames.

Vectral applies filter with different coefficients for increasing and decreasing magnitude vectors, bin by bin.

Parent *PyoObject*

Args

input: PyoObject Magnitude input signal, usually from an FFT analysis.

framesize: int, optional Frame size in samples. Usually the same as the FFT size. Defaults to 1024.

overlaps: int, optional Number of overlaps in incomming signal. Usually the same as the FFT overlaps. Defaults to 4.

up: float or PyoObject, optional Filter coefficient for increasing bins, between 0 and 1. Lower values results in a longer ramp time for bin magnitude. Defaults to 1.

down: float or PyoObject, optional Filter coefficient for decreasing bins, between 0 and 1. Lower values results in a longer decay time for bin magnitude. Defaults to 0.7

damp: float or PyoObject, optional High frequencies damping factor, between 0 and 1. Lower values mean more damping. Defaults to 0.9.

Note: Vectral has no *out* method. Signal must be converted back to time domain, with IFFT, before being sent to output.

```
>>> s = Server().boot()
>>> snd = SNDS_PATH + '/accord.aif'
>>> size, olaps = 1024, 4
>>> snd = SfPlayer(snd, speed=[.75,.8], loop=True, mul=.3)
>>> fin = FFT(snd, size=size, overlaps=olaps)
>>> pol = CarToPol(fin["real"], fin["imag"])
>>> vec = Vectral(pol["mag"], framesize=size, overlaps=olaps, down=.2, damp=.6)
>>> car = PolToCar(vec, pol["ang"])
>>> fout = IFFT(car["real"], car["imag"], size=size, overlaps=olaps).mix(2).out()
>>> s.start()
```

out(*chnl=0*, *inc=1*, *dur=0*, *delay=0*)

Start processing and send samples to audio output beginning at *chnl*.

This method returns *self*, allowing it to be applied at the object creation.

Args

chnl: int, optional Physical output assigned to the first audio stream of the object. Defaults to 0.

inc: int, optional Output channel increment value. Defaults to 1.

dur: float, optional Duration, in seconds, of the object's activation. The default is 0 and means infinite duration.

delay: float, optional Delay, in seconds, before the object's activation. Defaults to 0.

If *chnl* >= 0, successive streams increment the output number by *inc* and wrap around the global number of channels.

If *chnl* is negative, streams begin at 0, increment the output number by *inc* and wrap around the global number of channels. Then, the list of streams is scrambled.

If *chnl* is a list, successive values in the list will be assigned to successive streams.

setInput(*x*, *fadetime=0.05*)

Replace the *input* attribute.

Args

x: PyoObject New signal to process.

fadetime: float, optional Crossfade time between old and new input. Default to 0.05.

setFrameSize(*x*)

Replace the *framesize* attribute.

Args

x: int new *framesize* attribute.

setUp(*x*)
Replace the *up* attribute.

Args

x: float or PyoObject new *up* attribute.

setDown(*x*)
Replace the *down* attribute.

Args

x: float or PyoObject new *down* attribute.

setDamp(*x*)
Replace the *damp* attribute.

Args

x: float or PyoObject new *damp* attribute.

ctrl(*map_list=None*, *title=None*, *wxnoserver=False*)
Opens a sliders window to control the parameters of the object. Only parameters that can be set to a PyoObject are allowed to be mapped on a slider.

If a list of values are given to a parameter, a multisliders will be used to control each stream independently.

Args

map_list: list of SLMap objects, optional Users defined set of parameters scaling. There is default scaling for each object that accept *ctrl* method.

title: string, optional Title of the window. If none is provided, the name of the class is used.

wxnoserver: boolean, optional With wxPython graphical toolkit, if True, tells the interpreter that there will be no server window.

If *wxnoserver* is set to True, the interpreter will not wait for the server GUI before showing the controller window.

input
PyoObject. Magnitude input signal.

framesize
int. Frame size in samples.

up
float or PyoObject. Filter coefficient for increasing bins.

down
float or PyoObject. Filter coefficient for decreasing bins.

damp
float or PyoObject. High frequencies damping factor.

Phase Vocoder

The phase vocoder is a digital signal processing technique of potentially great musical significance. It can be used to perform very high fidelity time scaling, pitch transposition, and myriad other modifications of sounds.

PVAnal

class PVAnal(*input*, *size=1024*, *overlaps=4*, *wintype=2*, *callback=None*)

Phase Vocoder analysis object.

PVAnal takes an input sound and performs the phase vocoder analysis on it. This results in two streams, one for the bin's magnitudes and the other for the bin's true frequencies. These two streams are used by the PVxxx object family to transform the input signal using spectral domain algorithms. The last object in the phase vocoder chain must be a PVSynth to perform the spectral to time domain conversion.

Parent *PyoPVObject*

Args

input: PyoObject Input signal to process.

size: int {pow-of-two > 4}, optional FFT size. Must be a power of two greater than 4. Defaults to 1024.

The FFT size is the number of samples used in each analysis frame.

overlaps: int, optional The number of overlaped analysis block. Must be a power of two. Defaults to 4.

More overlaps can greatly improved sound quality synthesis but it is also more CPU expensive.

wintype: int, optional Shape of the envelope used to filter each input frame. Possible shapes are:

0. rectangular (no windowing)
1. Hamming
2. Hanning (default)
3. Bartlett (triangular)
4. Blackman 3-term
5. Blackman-Harris 4-term
6. Blackman-Harris 7-term
7. Tuckey (alpha = 0.66)
8. Sine (half-sine window)

callback: callable, optional If not None (default), this function will be called with the result of the analysis at the end of every overlap. The function will receive two arguments, a list of floats for both the magnitudes and the frequencies. The signature is:

callback(magnitudes, frequencies)

If you analyse a multi-channel signal, you should pass a list of callables, one per channel to analyse.

```
>>> s = Server().boot()
>>> s.start()
>>> a = SfPlayer(SNDS_PATH+"/transparent.aif", loop=True, mul=0.7)
>>> pva = PVAnal(a, size=1024, overlaps=4, wintype=2)
>>> pvs = PVSynth(pva).mix(2).out()
```

setInput(*x*, *fadetime=0.05*)

Replace the *input* attribute.

Args

x: PyoObject New signal to process.

fadetime: float, optional Crossfade time between old and new input. Default to 0.05.

setSize(*x*)
Replace the *size* attribute.

Args

x: int new *size* attribute.

setOverlaps(*x*)
Replace the *overlaps* attribute.

Args

x: int new *overlaps* attribute.

setWinType(*x*)
Replace the *wintype* attribute.

Args

x: int new *wintype* attribute.

setCallback(*x*)
Replace the *callback* attribute.

Args

x: callable new *callback* attribute.

input
PyoObject. Input signal to process.

size
int. FFT size.

overlaps
int. FFT overlap factor.

wintype
int. Windowing method.

PVSynth

class PVSynth(*input*, *wintype=2*, *mul=1*, *add=0*)
Phase Vocoder synthesis object.

PVSynth takes a PyoPVObject as its input and performed the spectral to time domain conversion on it. This step converts phase vocoder magnitude and true frequency's streams back to a real signal.

Parent `PyoObject`

Args

input: PyoPVObject Phase vocoder streaming object to process.

wintype: int, optional Shape of the envelope used to filter each input frame. Possible shapes are:

0. rectangular (no windowing)

1. Hamming
2. Hanning (default)
3. Bartlett (triangular)
4. Blackman 3-term
5. Blackman-Harris 4-term
6. Blackman-Harris 7-term
7. Tuckey (alpha = 0.66)
8. Sine (half-sine window)

```
>>> s = Server().boot()
>>> s.start()
>>> a = SfPlayer(SNDS_PATH+"/transparent.aif", loop=True, mul=0.7)
>>> pva = PVAnal(a, size=1024, overlaps=4, wintype=2)
>>> pvs = PVSynth(pva).mix(2).out()
```

setInput (*x*)
Replace the *input* attribute.

Args

x: PyoPVObject New signal to process.

setWinType (*x*)
Replace the *wintype* attribute.

Args

x: int new *wintype* attribute.

ctrl (*map_list=None*, *title=None*, *wxnoserver=False*)
Opens a sliders window to control the parameters of the object. Only parameters that can be set to a PyoObject are allowed to be mapped on a slider.

If a list of values are given to a parameter, a multisliders will be used to control each stream independently.

Args

map_list: list of SLMap objects, optional Users defined set of parameters scaling. There is default scaling for each object that accept *ctrl* method.

title: string, optional Title of the window. If none is provided, the name of the class is used.

wxnoserver: boolean, optional With wxPython graphical toolkit, if True, tells the interpreter that there will be no server window.

If *wxnoserver* is set to True, the interpreter will not wait for the server GUI before showing the controller window.

input
PyoPVObject. Input signal to process.

wintype
int. Windowing method.

PVAddSynth

class PVAddSynth(*input*, *pitch=1*, *num=100*, *first=0*, *inc=1*, *mul=1*, *add=0*)

Phase Vocoder additive synthesis object.

PVAddSynth takes a PyoPVObject as its input and resynthesize the real signal using the magnitude and true frequency's streams to control amplitude and frequency envelopes of an oscillator bank.

Parent *PyoObject*

Args

input: PyoPVObject Phase vocoder streaming object to process.

pitch: float or PyoObject, optional Transposition factor. Defaults to 1.

num: int, optional Number of oscillators used to synthesize the output sound. Defaults to 100.

first: int, optional The first bin to synthesize, starting from 0. Defaults to 0.

inc: int, optional Starting from bin *first*, resynthesize bins *inc* apart. Defaults to 1.

```
>>> s = Server().boot()
>>> s.start()
>>> a = SfPlayer(SNDS_PATH+"/transparent.aif", loop=True, mul=0.7)
>>> pva = PVAnal(a, size=1024, overlaps=4, wintype=2)
>>> pvs = PVAddSynth(pva, pitch=1.25, num=100, first=0, inc=2).out()
```

setInput(*x*)

Replace the *input* attribute.

Args

x: PyoPVObject New signal to process.

setPitch(*x*)

Replace the *pitch* attribute.

Args

x: float or PyoObject new *pitch* attribute.

setNum(*x*)

Replace the *num* attribute.

Args

x: int new *num* attribute.

setFirst(*x*)

Replace the *first* attribute.

Args

x: int new *first* attribute.

setInc(*x*)

Replace the *inc* attribute.

Args

x: int new *inc* attribute.

ctrl(*map_list=None*, *title=None*, *wxnoserver=False*)
Opens a sliders window to control the parameters of the object. Only parameters that can be set to a PyoObject are allowed to be mapped on a slider.

If a list of values are given to a parameter, a multisliders will be used to control each stream independently.

Args

map_list: list of SLMap objects, optional Users defined set of parameters scaling. There is default scaling for each object that accept *ctrl* method.

title: string, optional Title of the window. If none is provided, the name of the class is used.

wxnoserver: boolean, optional With wxPython graphical toolkit, if True, tells the interpreter that there will be no server window.

If *wxnoserver* is set to True, the interpreter will not wait for the server GUI before showing the controller window.

input
PyoPVObject. Input signal to process.

pitch
float or PyoObject. Transposition factor.

num
int. Number of oscillators.

first
int. First bin to synthesize.

inc
int. Synthesized bin increment.

PVTranspose

class PVTranspose(*input*, *transpo=1*)
Transpose the frequency components of a pv stream.

Parent *PyoPVObject*

Args

input: PyoPVObject Phase vocoder streaming object to process.

transpo: float or PyoObject, optional Transposition factor. Defaults to 1.

```
>>> s = Server().boot()
>>> s.start()
>>> sf = SfPlayer(SNDS_PATH+"/transparent.aif", loop=True, mul=.7)
>>> pva = PVAnal(sf, size=1024)
>>> pvt = PVTranspose(pva, transpo=1.5)
>>> pvs = PVSynth(pvt).out()
>>> dry = Delay(sf, delay=1024./s.getSamplingRate(), mul=.7).out(1)
```

setInput(*x*)
Replace the *input* attribute.

Args

x: PyoPVObject New signal to process.

setTranspo(*x*)
Replace the *transpo* attribute.

Args

x: int new *transpo* attribute.

ctrl(*map_list=None*, *title=None*, *wxnoserver=False*)
Opens a sliders window to control the parameters of the object. Only parameters that can be set to a PyoObject are allowed to be mapped on a slider.

If a list of values are given to a parameter, a multisliders will be used to control each stream independently.

Args

map_list: list of SLMap objects, optional Users defined set of parameters scaling. There is default scaling for each object that accept *ctrl* method.

title: string, optional Title of the window. If none is provided, the name of the class is used.

wxnoserver: boolean, optional With wxPython graphical toolkit, if True, tells the interpreter that there will be no server window.

If *wxnoserver* is set to True, the interpreter will not wait for the server GUI before showing the controller window.

input
PyoPVObject. Input signal to process.

transpo
float or PyoObject. Transposition factor.

PVVerb

class PVVerb(*input*, *revtime=0.75*, *damp=0.75*)
Spectral domain reverberation.

Parent *PyoPVObject*

Args

input: PyoPVObject Phase vocoder streaming object to process.

revtime: float or PyoObject, optional Reverberation factor, between 0 and 1. Defaults to 0.75.

damp: float or PyoObject, optional High frequency damping factor, between 0 and 1. 1 means no damping and 0 is the most damping. Defaults to 0.75.

```
>>> s = Server().boot()
>>> s.start()
>>> sf = SfPlayer(SNDS_PATH+"/transparent.aif", loop=True, mul=.5)
>>> pva = PVAnal(sf, size=2048)
>>> pvg = PVGate(pva, thresh=-36, damp=0)
>>> pvv = PVVerb(pvg, revtime=0.95, damp=0.95)
>>> pvs = PVSynth(pvv).mix(2).out()
>>> dry = Delay(sf, delay=2048./s.getSamplingRate(), mul=.4).mix(2).out()
```

setInput(*x*)
Replace the *input* attribute.

Args

x: PyoPVObject New signal to process.

setRevtime(*x*)
Replace the *revtime* attribute.

Args

x: int new *revtime* attribute.

setDamp(*x*)
Replace the *damp* attribute.

Args

x: int new *damp* attribute.

ctrl(*map_list=None*, *title=None*, *wxnoserver=False*)
Opens a sliders window to control the parameters of the object. Only parameters that can be set to a PyoObject are allowed to be mapped on a slider.

If a list of values are given to a parameter, a multisliders will be used to control each stream independently.

Args

map_list: list of SLMap objects, optional Users defined set of parameters scaling. There is default scaling for each object that accept *ctrl* method.

title: string, optional Title of the window. If none is provided, the name of the class is used.

wxnoserver: boolean, optional With wxPython graphical toolkit, if True, tells the interpreter that there will be no server window.

If *wxnoserver* is set to True, the interpreter will not wait for the server GUI before showing the controller window.

input
PyoPVObject. Input signal to process.

revtime
float or PyoObject. Reverberation factor.

damp
float or PyoObject. High frequency damping factor.

PVGate

class PVGate(*input*, *thresh=-20*, *damp=0.0*, *inverse=False*)
Spectral gate.

Parent *PyoObject*

Args

input: PyoPVObject Phase vocoder streaming object to process.

thresh: float or PyoObject, optional Threshold factor in dB. Bins below that threshold will be scaled by *damp* factor. Defaults to -20.

damp: float or PyoObject, optional Damping factor for low amplitude bins. Defaults to 0.

inverse: boolean, optional If True, the damping factor will be applied to the bins with amplitude above the given threshold. If False, the damping factor is applied to bins with amplitude below the given threshold. Defaults to False.

```
>>> s = Server().boot()
>>> s.start()
>>> sf = SfPlayer(SNDS_PATH+"/transparent.aif", loop=True, mul=.5)
>>> pva = PVAnal(sf, size=2048)
>>> pvg = PVGate(pva, thresh=-50, damp=0)
>>> pvs = PVSynth(pvg).mix(2).out()
```

setInput(*x*)
Replace the *input* attribute.

Args

x: PyoPVObject New signal to process.

setThresh(*x*)
Replace the *thresh* attribute.

Args

x: int new *thresh* attribute.

setDamp(*x*)
Replace the *damp* attribute.

Args

x: int new *damp* attribute.

setInverse(*x*)
Replace the *inverse* attribute.

Args

x: boolean new *inverse* attribute.

ctrl(*map_list=None*, *title=None*, *wxnoserver=False*)
Opens a sliders window to control the parameters of the object. Only parameters that can be set to a PyoObject are allowed to be mapped on a slider.

If a list of values are given to a parameter, a multisliders will be used to control each stream independently.

Args

map_list: list of SLMap objects, optional Users defined set of parameters scaling. There is default scaling for each object that accept *ctrl* method.

title: string, optional Title of the window. If none is provided, the name of the class is used.

wxnoserver: boolean, optional With wxPython graphical toolkit, if True, tells the interpreter that there will be no server window.

If *wxnoserver* is set to True, the interpreter will not wait for the server GUI before showing the controller window.

input
PyoPVObject. Input signal to process.

thresh
float or PyoObject. Threshold factor.

damp
float or PyoObject. Damping factor for low amplitude bins.

inverse
boolean. If True, the gate is applied to high amplitude bins.

PVCross

class PVCross(*input*, *input2*, *fade=1*)
Performs cross-synthesis between two phase vocoder streaming object.

The amplitudes from *input* and *input2* (scaled by *fade* argument) are applied to the frequencies of *input*.

Parent *PyoPVObject*

Args

input: PyoPVObject Phase vocoder streaming object to process. Frequencies from this pv stream are used to compute the output signal.

input2: PyoPVObject Phase vocoder streaming object which gives the second set of magnitudes. Frequencies from this pv stream are not used.

fade: float or PyoObject, optional Scaling factor for the output amplitudes, between 0 and 1. 0 means amplitudes from *input* and 1 means amplitudes from *input2*. Defaults to 1.

Note: The two input pv stream must have the same size and overlaps. It is the responsibility of the user to be sure they are consistent. To change the size (or the overlaps) of the phase vocoder process, one must write a function to change both at the same time (see the example below). Another possibility is to use channel expansion to analyse both sounds with the same PVAnal object.

```
>>> s = Server().boot()
>>> s.start()
>>> sf = SineLoop(freq=[80,81], feedback=0.07, mul=.5)
>>> sf2 = SfPlayer(SNDS_PATH+"/transparent.aif", loop=True, mul=.5)
>>> pva = PVAnal(sf)
>>> pva2 = PVAnal(sf2)
>>> pvc = PVCross(pva, pva2, fade=1)
>>> pvs = PVSynth(pvc).out()
>>> def size(x):
...     pva.size = x
...     pva2.size = x
>>> def olaps(x):
...     pva.overlaps = x
...     pva2.overlaps = x
```

setInput(*x*)
Replace the *input* attribute.

Args

x: PyoPVObject New signal to process.

setInput2(*x*)
Replace the *input2* attribute.

Args

x: PyoPVObject New signal to process.

setFade(*x*)
Replace the *fade* attribute.

Args

x: float or PyoObject new *fade* attribute.

ctrl(*map_list=None*, *title=None*, *wxnoserver=False*)
Opens a sliders window to control the parameters of the object. Only parameters that can be set to a PyoObject are allowed to be mapped on a slider.

If a list of values are given to a parameter, a multisliders will be used to control each stream independently.

Args

map_list: list of SLMap objects, optional Users defined set of parameters scaling. There is default scaling for each object that accept *ctrl* method.

title: string, optional Title of the window. If none is provided, the name of the class is used.

wxnoserver: boolean, optional With wxPython graphical toolkit, if True, tells the interpreter that there will be no server window.

If *wxnoserver* is set to True, the interpreter will not wait for the server GUI before showing the controller window.

input
PyoPVObject. Input signal to process.

input2
PyoPVObject. Second set of amplitudes.

fade
float or PyoObject. Scaling factor.

PVMult

class PVMult(*input*, *input2*)
Multiply magnitudes from two phase vocoder streaming object.

Parent *PyoPVObject*

Args

input: PyoPVObject Phase vocoder streaming object to process. Frequencies from this pv stream are used to compute the output signal.

input2: PyoPVObject Phase vocoder streaming object which gives the second set of magnitudes. Frequencies from this pv stream are not used.

Note: The two input pv stream must have the same size and overlaps. It is the responsibility of the user to be sure they are consistent. To change the size (or the overlaps) of the phase vocoder process, one must write a function to change both at the same time (see the example below). Another possibility is to use channel expansion to analyse both sounds with the same PVAnal object.

```
>>> s = Server().boot()
>>> s.start()
>>> sf = FM(carrier=[100,150], ratio=[.999,.5005], index=20, mul=.4)
```

(continues on next page)

(continued from previous page)

```
>>> sf2 = SfPlayer(SNDS_PATH+"/transparent.aif", loop=True, mul=.5)
>>> pva = PVAnal(sf)
>>> pva2 = PVAnal(sf2)
>>> pvc = PVMult(pva, pva2)
>>> pvs = PVSynth(pvc).out()
>>> def size(x):
...     pva.size = x
...     pva2.size = x
>>> def olaps(x):
...     pva.overlaps = x
...     pva2.overlaps = x
```

setInput(*x*)
Replace the *input* attribute.

Args

x: PyoPVObject New signal to process.

setInput2(*x*)
Replace the *input2* attribute.

Args

x: PyoPVObject New signal to process.

input
PyoPVObject. Input signal to process.

input2
PyoPVObject. Second set of magnitudes.

PVMorph

class PVMorph(*input*, *input2*, *fade=0.5*)
Performs spectral morphing between two phase vocoder streaming object.

According to *fade* argument, the amplitudes from *input* and *input2* are interpolated linearly while the frequencies are interpolated exponentially.

Parent *PyoPVObject*

Args

input: PyoPVObject Phase vocoder streaming object which gives the first set of magnitudes and frequencies.

input2: PyoPVObject Phase vocoder streaming object which gives the second set of magnitudes and frequencies.

fade: float or PyoObject, optional Scaling factor for the output amplitudes and frequencies, between 0 and 1. 0 is *input* and 1 in *input2*. Defaults to 0.5.

Note: The two input pv stream must have the same size and overlaps. It is the responsibility of the user to be sure they are consistent. To change the size (or the overlaps) of the phase vocoder process, one must write a function to change both at the same time (see the example below). Another possibility is to use channel expansion to analyse both sounds with the same PVAnal object.

```
>>> s = Server().boot()
>>> s.start()
>>> sf = SineLoop(freq=[100,101], feedback=0.12, mul=.5)
>>> sf2 = SfPlayer(SNDS_PATH+"/transparent.aif", loop=True, mul=.5)
>>> pva = PVAnal(sf)
>>> pva2 = PVAnal(sf2)
>>> pvc = PVMorph(pva, pva2, fade=0.5)
>>> pvs = PVSynth(pvc).out()
>>> def size(x):
...     pva.size = x
...     pva2.size = x
>>> def olaps(x):
...     pva.overlaps = x
...     pva2.overlaps = x
```

setInput (*x*)
Replace the *input* attribute.

Args

x: PyoPVObject New signal to process.

setInput2 (*x*)
Replace the *input2* attribute.

Args

x: PyoPVObject New signal to process.

setFade (*x*)
Replace the *fade* attribute.

Args

x: float or PyoObject new *fade* attribute.

ctrl (*map_list=None*, *title=None*, *wxnoserver=False*)
Opens a sliders window to control the parameters of the object. Only parameters that can be set to a PyoObject are allowed to be mapped on a slider.

If a list of values are given to a parameter, a multisliders will be used to control each stream independently.

Args

map_list: list of SLMap objects, optional Users defined set of parameters scaling. There is default scaling for each object that accept *ctrl* method.

title: string, optional Title of the window. If none is provided, the name of the class is used.

wxnoserver: boolean, optional With wxPython graphical toolkit, if True, tells the interpreter that there will be no server window.

If *wxnoserver* is set to True, the interpreter will not wait for the server GUI before showing the controller window.

input
PyoPVObject. First input signal.

input2
PyoPVObject. Second input signal.

fade
float or PyoObject. Scaling factor.

PVFilter

class PVFilter(*input*, *table*, *gain=1*, *mode=0*)
Spectral filter.

PVFilter filters frequency components of a pv stream according to the shape drawn in the table given in argument.

Parent *PyoPVObject*

Args

input: PyoPVObject Phase vocoder streaming object to process.

table: PyoTableObject Table containing the filter shape. If the table length is smaller than fftsize/2, remaining bins will be set to 0.

gain: float or PyoObject, optional Gain of the filter applied to the input spectrum. Defaults to 1.

mode: int, optional Table scanning mode. Defaults to 0.

If 0, bin indexes outside table size are set to 0. If 1, bin indexes are scaled over table length.

```
>>> s = Server().boot()
>>> s.start()
>>> t = ExpTable([(0,1),(61,1),(71,0),(131,1),(171,0),(511,0)], size=512)
>>> src = Noise(.4)
>>> pva = PVAnal(src, size=1024)
>>> pvf = PVFilter(pva, t)
>>> pvs = PVSynth(pvf).out()
```

setInput(*x*)
Replace the *input* attribute.

Args

x: PyoPVObject New signal to process.

setTable(*x*)
Replace the *table* attribute.

Args

x: PyoTableObject new *table* attribute.

setGain(*x*)
Replace the *gain* attribute.

Args

x: float or PyoObject new *gain* attribute.

setMode(*x*)
Replace the *mode* attribute.

Args

x: int new *mode* attribute.

ctrl(*map_list=None*, *title=None*, *wxnoserver=False*)
Opens a sliders window to control the parameters of the object. Only parameters that can be set to a PyoObject are allowed to be mapped on a slider.

If a list of values are given to a parameter, a multisliders will be used to control each stream independently.

Args

map_list: list of SLMap objects, optional Users defined set of parameters scaling. There is default scaling for each object that accept *ctrl* method.

title: string, optional Title of the window. If none is provided, the name of the class is used.

wxnoserver: boolean, optional With wxPython graphical toolkit, if True, tells the interpreter that there will be no server window.

If *wxnoserver* is set to True, the interpreter will not wait for the server GUI before showing the controller window.

input
PyoPVObject. Input signal to process.

table
PyoTableObject. Table containing the filter shape.

gain
float or PyoObject. Gain of the filter.

mode
int. Table scanning mode.

PVDelay

class PVDelay(*input*, *deltable*, *feedtable*, *maxdelay=1.0*, *mode=0*)
Spectral delays.

PVDelay applies different delay times and feedbacks for each bin of a phase vocoder analysis. Delay times and feedbacks are specified with PyoTableObjects.

Parent *PyoPVObject*

Args

input: PyoPVObject Phase vocoder streaming object to process.

deltable: PyoTableObject Table containing delay times, as integer multipliers of the FFT hopsize (fftsize / overlaps).

If the table length is smaller than fftsize/2, remaining bins will be set to 0.

feedtable: PyoTableObject Table containing feedback values, between -1 and 1.

If the table length is smaller than fftsize/2, remaining bins will be set to 0.

maxdelay: float, optional Maximum delay time in seconds. Available at initialization time only. Defaults to 1.0.

mode: int, optional Tables scanning mode. Defaults to 0.

If 0, bin indexes outside table size are set to 0. If 1, bin indexes are scaled over table length.

```
>>> s = Server().boot()
>>> s.start()
>>> SIZE = 1024
>>> SIZE2 = int(SIZE / 2)
>>> OLAPS = 4
>>> MAXDEL = 2.0 # two seconds delay memories
>>> FRAMES = int(MAXDEL * s.getSamplingRate() / (SIZE / OLAPS))
>>> # Edit tables with the graph() method. yrange=(0, FRAMES) for delays table
>>> dt = DataTable(size=SIZE2, init=[i / float(SIZE2) * FRAMES for i in␣
↪range(SIZE2)])
>>> ft = DataTable(size=SIZE2, init=[0.5]*SIZE2)
>>> src = SfPlayer(SNDS_PATH+"/transparent.aif", loop=True, mul=0.5)
>>> pva = PVAnal(src, size=SIZE, overlaps=OLAPS)
>>> pvd = PVDelay(pva, dt, ft, maxdelay=MAXDEL)
>>> pvs = PVSynth(pvd).out()
```

setInput (*x*)
Replace the *input* attribute.

Args

x: PyoPVObject New signal to process.

setDeltable (*x*)
Replace the *deltable* attribute.

Args

x: PyoTableObject new *deltable* attribute.

setFeedtable (*x*)
Replace the *feedtable* attribute.

Args

x: PyoTableObject new *feedtable* attribute.

setMode (*x*)
Replace the *mode* attribute.

Args

x: int new *mode* attribute.

input
PyoPVObject. Input signal to process.

deltable
PyoTableObject. Table containing the delay times.

feedtable
PyoTableObject. Table containing feedback values.

mode
int. Table scanning mode.

PVBuffer

class PVBuffer (*input*, *index*, *pitch=1.0*, *length=1.0*)
Phase vocoder buffer and playback with transposition.

PVBuffer keeps *length* seconds of pv analysis in memory and gives control on playback position and transposition.

Parent *PyoPVObject*

Args

input: PyoPVObject Phase vocoder streaming object to process.

index: PyoObject Playback position, as audio stream, normalized between 0 and 1.

pitch: float or PyoObject, optional Transposition factor. Defaults to 1.

length: float, optional Memory length in seconds. Available at initialization time only. Defaults to 1.0.

Note: The play() method can be called to start a new recording of the current pv input.

```
>>> s = Server().boot()
>>> s.start()
>>> f = SNDS_PATH+'/transparent.aif'
>>> f_len = sndinfo(f)[1]
>>> src = SfPlayer(f, mul=0.5)
>>> index = Phasor(freq=1.0/f_len*0.25, phase=0.9)
>>> pva = PVAnal(src, size=1024, overlaps=8)
>>> pvb = PVBuffer(pva, index, pitch=1.25, length=f_len)
>>> pvs = PVSynth(pvb).out()
```

setInput(*x*)
Replace the *input* attribute.

Args

x: PyoPVObject New signal to process.

setIndex(*x*)
Replace the *index* attribute.

Args

x: PyoObject new *index* attribute.

setPitch(*x*)
Replace the *pitch* attribute.

Args

x: float or PyoObject new *pitch* attribute.

ctrl(*map_list=None*, *title=None*, *wxnoserver=False*)
Opens a sliders window to control the parameters of the object. Only parameters that can be set to a PyoObject are allowed to be mapped on a slider.

If a list of values are given to a parameter, a multisliders will be used to control each stream independently.

Args

map_list: list of SLMap objects, optional Users defined set of parameters scaling. There is default scaling for each object that accept *ctrl* method.

title: string, optional Title of the window. If none is provided, the name of the class is used.

wxnoserver: boolean, optional With wxPython graphical toolkit, if True, tells the interpreter that there will be no server window.

If *wxnoserver* is set to True, the interpreter will not wait for the server GUI before showing the controller window.

input
PyoPVObject. Input signal to process.

index
PyoObject. Reader's normalized position.

pitch
float or PyoObject. Transposition factor.

PVShift

class PVShift(*input*, *shift=0*)
Spectral domain frequency shifter.

PVShift linearly moves the analysis bins by the amount, in Hertz, specified by the the *shift* argument.

Parent *PyoPVObject*

Args

input: PyoPVObject Phase vocoder streaming object to process.

shift: float or PyoObject, optional Frequency shift factor. Defaults to 0.

```
>>> s = Server().boot()
>>> s.start()
>>> sf = SfPlayer(SNDS_PATH+"/transparent.aif", loop=True, mul=.7)
>>> pva = PVAnal(sf, size=1024)
>>> pvt = PVShift(pva, shift=500)
>>> pvs = PVSynth(pvt).out()
```

setInput(*x*)
Replace the *input* attribute.

Args

x: PyoPVObject New signal to process.

setShift(*x*)
Replace the *shift* attribute.

Args

x: float or PyoObject new *shift* attribute.

ctrl(*map_list=None*, *title=None*, *wxnoserver=False*)
Opens a sliders window to control the parameters of the object. Only parameters that can be set to a PyoObject are allowed to be mapped on a slider.

If a list of values are given to a parameter, a multisliders will be used to control each stream independently.

Args

map_list: list of SLMap objects, optional Users defined set of parameters scaling. There is default scaling for each object that accept *ctrl* method.

title: string, optional Title of the window. If none is provided, the name of the class is used.

wxnoserver: boolean, optional With wxPython graphical toolkit, if True, tells the interpreter that there will be no server window.

If *wxnoserver* is set to True, the interpreter will not wait for the server GUI before showing the controller window.

input
PyoPVObject. Input signal to process.

shift
float or PyoObject. Frequency shift factor.

PVAmpMod

class PVAmpMod(*input*, *basefreq=1*, *spread=0*, *shape=0*)
Performs frequency independent amplitude modulations.

PVAmpMod modulates the magnitude of each bin of a pv stream with an independent oscillator. *basefreq* and *spread* are used to derive the frequency of each modulating oscillator.

Internally, the following operations are applied to derive oscillator frequencies (*i* is the bin number):

spread = spread * 0.001 + 1.0

f_i = basefreq * pow(spread, i)

Parent *PyoPVObject*

Args

input: PyoPVObject Phase vocoder streaming object to process.

basefreq: float or PyoObject, optional Base modulation frequency, in Hertz. Defaults to 1.

spread: float or PyoObject, optional Spreading factor for oscillator frequencies, between -1 and 1. 0 means every oscillator has the same frequency.

shape: int, optional

Modulation oscillator waveform. Possible shapes are:

0. Sine (default)
1. Sawtooth
2. Ramp (inverse sawtooth)
3. Square
4. Triangle
5. Brown Noise
6. Pink Noise
7. White Noise

```
>>> s = Server().boot()
>>> s.start()
>>> src = PinkNoise(.3)
>>> pva = PVAnal(src, size=1024, overlaps=4)
>>> pvm = PVAmpMod(pva, basefreq=4, spread=0.5)
>>> pvs = PVSynth(pvm).out()
```

setInput(*x*)
Replace the *input* attribute.

Args

x: PyoPVObject New signal to process.

setBasefreq(*x*)
Replace the *basefreq* attribute.

Args

x: float or PyoObject new *basefreq* attribute.

setSpread(*x*)
Replace the *spread* attribute.

Args

x: float or PyoObject new *spread* attribute.

setShape(*x*)
Replace the *shape* attribute.

Args

x: int

new *shape* attribute. Possible shapes are:

0. Sine (default)
1. Sawtooth
2. Ramp (inverse sawtooth)
3. Square
4. Triangle
5. Brown Noise
6. Pink Noise
7. White Noise

reset()
Resets modulation pointers to 0.

ctrl(*map_list=None*, *title=None*, *wxnoserver=False*)
Opens a sliders window to control the parameters of the object. Only parameters that can be set to a PyoObject are allowed to be mapped on a slider.

If a list of values are given to a parameter, a multisliders will be used to control each stream independently.

Args

map_list: list of SLMap objects, optional Users defined set of parameters scaling. There is default scaling for each object that accept *ctrl* method.

title: string, optional Title of the window. If none is provided, the name of the class is used.

wxnoserver: boolean, optional With wxPython graphical toolkit, if True, tells the interpreter that there will be no server window.

If *wxnoserver* is set to True, the interpreter will not wait for the server GUI before showing the controller window.

input
PyoPVObject. Input signal to process.

basefreq
float or PyoObject. Modulator's base frequency.

spread
float or PyoObject. Modulator's frequency spreading factor.

shape
int. Modulation oscillator waveform.

PVFreqMod

class PVFreqMod(*input*, *basefreq=1*, *spread=0*, *depth=0.1*, *shape=0*)
Performs frequency independent frequency modulations.

PVFreqMod modulates the frequency of each bin of a pv stream with an independent oscillator. *basefreq* and *spread* are used to derive the frequency of each modulating oscillator.

Internally, the following operations are applied to derive oscillator frequencies (*i* is the bin number):

spread = spread * 0.001 + 1.0

f_i = basefreq * pow(spread, i)

Parent *PyoPVObject*

Args

input: PyoPVObject Phase vocoder streaming object to process.

basefreq: float or PyoObject, optional Base modulation frequency, in Hertz. Defaults to 1.

spread: float or PyoObject, optional Spreading factor for oscillator frequencies, between -1 and 1. 0 means every oscillator has the same frequency.

depth: float or PyoObject, optional Amplitude of the modulating oscillators, between 0 and 1. Defaults to 0.1.

shape: int, optional

Modulation oscillator waveform. Possible shapes are:

0. Sine (default)
1. Sawtooth
2. Ramp (inverse sawtooth)
3. Square
4. Triangle
5. Brown Noise

6. Pink Noise
7. White Noise

```
>>> s = Server().boot()
>>> s.start()
>>> src = SfPlayer(SNDS_PATH+"/accord.aif", loop=True, mul=0.5)
>>> pva = PVAnal(src, size=1024, overlaps=4)
>>> pvm = PVFreqMod(pva, basefreq=8, spread=0.75, depth=0.05)
>>> pvs = PVSynth(pvm).out()
```

setInput(*x*)
Replace the *input* attribute.

Args

x: PyoPVObject New signal to process.

setBasefreq(*x*)
Replace the *basefreq* attribute.

Args

x: float or PyoObject new *basefreq* attribute.

setSpread(*x*)
Replace the *spread* attribute.

Args

x: float or PyoObject new *spread* attribute.

setDepth(*x*)
Replace the *depth* attribute.

Args

x: float or PyoObject new *depth* attribute.

setShape(*x*)
Replace the *shape* attribute.

Args

x: int

new *shape* attribute. Possible shapes are:

0. Sine (default)
1. Sawtooth
2. Ramp (inverse sawtooth)
3. Square
4. Triangle
5. Brown Noise
6. Pink Noise
7. White Noise

reset()
Resets modulation pointers to 0.

ctrl(*map_list=None*, *title=None*, *wxnoserver=False*)
Opens a sliders window to control the parameters of the object. Only parameters that can be set to a PyoObject are allowed to be mapped on a slider.

If a list of values are given to a parameter, a multisliders will be used to control each stream independently.

Args

map_list: list of SLMap objects, optional Users defined set of parameters scaling. There is default scaling for each object that accept *ctrl* method.

title: string, optional Title of the window. If none is provided, the name of the class is used.

wxnoserver: boolean, optional With wxPython graphical toolkit, if True, tells the interpreter that there will be no server window.

If *wxnoserver* is set to True, the interpreter will not wait for the server GUI before showing the controller window.

input
PyoPVObject. Input signal to process.

basefreq
float or PyoObject. Modulator's base frequency.

spread
float or PyoObject. Modulator's frequencies spreading factor.

depth
float or PyoObject. Amplitude of the modulators.

shape
int. Modulation oscillator waveform.

PVBufLoops

class PVBufLoops(*input*, *low=1.0*, *high=1.0*, *mode=0*, *length=1.0*)
Phase vocoder buffer with bin independent speed playback.

PVBufLoops keeps *length* seconds of pv analysis in memory and gives control on playback position independently for every frequency bin.

Parent `PyoPVObject`

Args

input: PyoPVObject Phase vocoder streaming object to process.

low: float or PyoObject, optional Lowest bin speed factor. Defaults to 1.0.

high: float or PyoObject, optional Highest bin speed factor. Defaults to 1.0.

mode: int, optional

Speed distribution algorithm. Available algorithms are:

0. linear, line between *low* and *high* (default)
1. exponential, exponential line between *low* and *high*
2. logarithmic, logarithmic line between *low* and *high*
3. random, uniform random between *low* and *high*

4. rand expon min, exponential random from *low* to *high*
5. rand expon max, exponential random from *high* to *low*
6. rand bi-expon, bipolar exponential random between *low* and *high*

length: float, optional Memory length in seconds. Available at initialization time only. Defaults to 1.0.

Note: The play() method can be called to start a new recording of the current pv input.

```
>>> s = Server().boot()
>>> s.start()
>>> f = SNDS_PATH+'/transparent.aif'
>>> f_len = sndinfo(f)[1]
>>> src = SfPlayer(f, mul=0.5)
>>> pva = PVAnal(src, size=1024, overlaps=8)
>>> pvb = PVBufLoops(pva, low=0.9, high=1.1, mode=3, length=f_len)
>>> pvs = PVSynth(pvb).out()
```

setInput (*x*)
Replace the *input* attribute.

Args

x: PyoPVObject New signal to process.

setLow (*x*)
Replace the *low* attribute.

Args

x: float or PyoObject new *low* attribute.

setHigh (*x*)
Replace the *high* attribute.

Args

x: float or PyoObject new *high* attribute.

setMode (*x*)
Replace the *mode* attribute.

Args

x: int new *mode* attribute.

reset ()
Reset pointer positions to 0.

ctrl (*map_list=None*, *title=None*, *wxnoserver=False*)
Opens a sliders window to control the parameters of the object. Only parameters that can be set to a PyoObject are allowed to be mapped on a slider.

If a list of values are given to a parameter, a multisliders will be used to control each stream independently.

Args

map_list: list of SLMap objects, optional Users defined set of parameters scaling. There is default scaling for each object that accept *ctrl* method.

title: string, optional Title of the window. If none is provided, the name of the class is used.

wxnoserver: boolean, optional With wxPython graphical toolkit, if True, tells the interpreter that there will be no server window.

If *wxnoserver* is set to True, the interpreter will not wait for the server GUI before showing the controller window.

input
PyoPVObject. Input signal to process.

low
float or PyoObject. Lowest bin speed factor.

high
float or PyoObject. Highest bin speed factor.

mode
int. Speed distribution algorithm.

PVBufTabLoops

class PVBufTabLoops(*input*, *speed*, *length=1.0*)
Phase vocoder buffer with bin independent speed playback.

PVBufTabLoops keeps *length* seconds of pv analysis in memory and gives control on playback position, using a PyoTableObject, independently for every frequency bin.

Parent *PyoPVObject*

Args

input: PyoPVObject Phase vocoder streaming object to process.

speed: PyoTableObject Table which specify the speed of bin playback readers.

length: float, optional Memory length in seconds. Available at initialization time only. Defaults to 1.0.

Note: The play() method can be called to start a new recording of the current pv input.

```
>>> s = Server().boot()
>>> s.start()
>>> f = SNDS_PATH+'/transparent.aif'
>>> f_len = sndinfo(f)[1]
>>> src = SfPlayer(f, mul=0.5)
>>> spd = ExpTable([(0,1), (512,0.5)], exp=6, size=512)
>>> pva = PVAnal(src, size=1024, overlaps=8)
>>> pvb = PVBufTabLoops(pva, spd, length=f_len)
>>> pvs = PVSynth(pvb).out()
```

setInput(*x*)
Replace the *input* attribute.

Args

x: PyoPVObject New signal to process.

setSpeed(*x*)
Replace the *speed* attribute.

Args

x: PyoTableObject new *speed* attribute.

reset()
Reset pointer positions to 0.

input
PyoPVObject. Input signal to process.

speed
PyoTableObject. Table which specify the speed of bin playback readers.

PVMix

class PVMix(*input*, *input2*)
Mix the most prominent components from two phase vocoder streaming objects.

Parent *PyoPVObject*

Args

input: PyoPVObject Phase vocoder streaming object 1.

input2: PyoPVObject Phase vocoder streaming object 2.

Note: The two input pv stream must have the same size and overlaps. It is the responsibility of the user to be sure they are consistent. To change the size (or the overlaps) of the phase vocoder process, one must write a function to change both at the same time (see the example below). Another possibility is to use channel expansion to analyse both sounds with the same PVAnal object.

```
>>> s = Server().boot()
>>> s.start()
>>> sf = SfPlayer(SNDS_PATH+"/transparent.aif", loop=True, mul=.5)
>>> sf2 = SfPlayer(SNDS_PATH+"/accord.aif", loop=True, mul=.5)
>>> pva = PVAnal(sf)
>>> pva2 = PVAnal(sf2)
>>> pvm = PVMix(pva, pva2)
>>> pvs = PVSynth(pvm).out()
>>> def size(x):
...     pva.size = x
...     pva2.size = x
>>> def olaps(x):
...     pva.overlaps = x
...     pva2.overlaps = x
```

setInput(*x*)
Replace the *input* attribute.

Args

x: PyoPVObject New signal to process.

setInput2(*x*)
Replace the *input2* attribute.

Args

x: PyoPVObject New signal to process.

input
PyoPVObject. Phase vocoder streaming object 1.

input2
PyoPVObject. Phase vocoder streaming object 2.

Signal Generators

Set of synthesis generators that can be used as sources of a signal processing chain or as parameter's modifiers.

Blit

class Blit (*freq=100*, *harms=40*, *mul=1*, *add=0*)
Band limited impulse train synthesis.

Impulse train generator with control over the number of harmonics in the spectrum, which gives oscillators with very low aliasing.

Parent *PyoObject*

Args

freq: float or PyoObject, optional Frequency in cycles per second. Defaults to 100.

harms: float or PyoObject, optional Number of harmonics in the generated spectrum. Defaults to 40.

```
>>> s = Server().boot()
>>> s.start()
>>> lfo = Sine(freq=4, mul=.02, add=1)
>>> lf2 = Sine(freq=.25, mul=10, add=30)
>>> a = Blit(freq=[100, 99.7]*lfo, harms=lf2, mul=.3).out()
```

setFreq (*x*)
Replace the *freq* attribute.

Args

x: float or PyoObject new *freq* attribute.

setHarms (*x*)
Replace the *harms* attribute.

Args

x: float or PyoObject new *harms* attribute.

ctrl (*map_list=None*, *title=None*, *wxnoserver=False*)
Opens a sliders window to control the parameters of the object. Only parameters that can be set to a PyoObject are allowed to be mapped on a slider.

If a list of values are given to a parameter, a multisliders will be used to control each stream independently.

Args

map_list: list of SLMap objects, optional Users defined set of parameters scaling. There is default scaling for each object that accept *ctrl* method.

title: string, optional Title of the window. If none is provided, the name of the class is used.

wxnoserver: boolean, optional With wxPython graphical toolkit, if True, tells the interpreter that there will be no server window.

If *wxnoserver* is set to True, the interpreter will not wait for the server GUI before showing the controller window.

freq
float or PyoObject. Frequency in cycles per second.

harms
float or PyoObject. Number of harmonics.

BrownNoise

class BrownNoise(*mul=1*, *add=0*)
A brown noise generator.

The spectrum of a brown noise has a power density which decreases 6 dB per octave with increasing frequency (density proportional to 1/f^2).

Parent *PyoObject*

```
>>> s = Server().boot()
>>> s.start()
>>> a = BrownNoise(.1).mix(2).out()
```

ctrl(*map_list=None*, *title=None*, *wxnoserver=False*)
Opens a sliders window to control the parameters of the object. Only parameters that can be set to a PyoObject are allowed to be mapped on a slider.

If a list of values are given to a parameter, a multisliders will be used to control each stream independently.

Args

map_list: list of SLMap objects, optional Users defined set of parameters scaling. There is default scaling for each object that accept *ctrl* method.

title: string, optional Title of the window. If none is provided, the name of the class is used.

wxnoserver: boolean, optional With wxPython graphical toolkit, if True, tells the interpreter that there will be no server window.

If *wxnoserver* is set to True, the interpreter will not wait for the server GUI before showing the controller window.

ChenLee

class ChenLee(*pitch=0.25*, *chaos=0.5*, *stereo=False*, *mul=1*, *add=0*)
Chaotic attractor for the Chen-Lee system.

The ChenLee attractor is a system of three non-linear ordinary differential equations. These differential equations define a continuous-time dynamical system that exhibits chaotic dynamics associated with the fractal properties of the attractor.

Parent *PyoObject*

Args

pitch: float or PyoObject, optional Controls the speed, in the range 0 -> 1, of the variations. With values below 0.2, this object can be used as a low frequency oscillator (LFO) and above 0.2, it will generate a broad spectrum noise with harmonic peaks. Defaults to 0.25.

chaos: float or PyoObject, optional Controls the chaotic behavior, in the range 0 -> 1, of the oscillator. 0 means nearly periodic while 1 is totally chaotic. Defaults to 0.5

stereo, boolean, optional If True, 2 streams will be generated, one with the X variable signal of the algorithm and a second composed of the Y variable signal of the algorithm. These two signal are strongly related in their frequency spectrum but the Y signal is slightly out-of-phase. Useful to create alternating LFOs. Available at initialization only. Defaults to False.

See also:

Rossler, *Lorenz*

```
>>> s = Server().boot()
>>> s.start()
>>> a = ChenLee(pitch=.001, chaos=0.2, stereo=True, mul=.5, add=.5)
>>> b = ChenLee(pitch=1, chaos=a, mul=0.5).out()
```

setPitch(*x*)
Replace the *pitch* attribute.

Args

x: float or PyoObject new *pitch* attribute. {0. -> 1.}

setChaos(*x*)
Replace the *chaos* attribute.

Args

x: float or PyoObject new *chaos* attribute. {0. -> 1.}

ctrl(*map_list=None*, *title=None*, *wxnoserver=False*)
Opens a sliders window to control the parameters of the object. Only parameters that can be set to a PyoObject are allowed to be mapped on a slider.

If a list of values are given to a parameter, a multisliders will be used to control each stream independently.

Args

map_list: list of SLMap objects, optional Users defined set of parameters scaling. There is default scaling for each object that accept *ctrl* method.

title: string, optional Title of the window. If none is provided, the name of the class is used.

wxnoserver: boolean, optional With wxPython graphical toolkit, if True, tells the interpreter that there will be no server window.

If *wxnoserver* is set to True, the interpreter will not wait for the server GUI before showing the controller window.

pitch
float or PyoObject. Speed of the variations.

chaos
float or PyoObject. Chaotic behavior.

CrossFM

class CrossFM(*carrier=100*, *ratio=0.5*, *ind1=2*, *ind2=2*, *mul=1*, *add=0*)

Cross frequency modulation generator.

Frequency modulation synthesis where the output of both oscillators modulates the frequency of the other one.

Parent *PyoObject*

Args

carrier: float or PyoObject, optional Carrier frequency in cycles per second. Defaults to 100.

ratio: float or PyoObject, optional A factor that, when multiplied by the *carrier* parameter, gives the modulator frequency. Defaults to 0.5.

ind1: float or PyoObject, optional The carrier index. This value multiplied by the carrier frequency gives the carrier amplitude for modulating the modulation oscillator frequency. Defaults to 2.

ind1: float or PyoObject, optional The modulation index. This value multiplied by the modulation frequency gives the modulation amplitude for modulating the carrier oscillator frequency. Defaults to 2.

```
>>> s = Server().boot()
>>> s.start()
>>> ind = LinTable([(0,20), (200,5), (1000,2), (8191,1)])
>>> m = Metro(4).play()
>>> tr = TrigEnv(m, table=ind, dur=4)
>>> f = CrossFM(carrier=[250.5,250], ratio=[.2499,.2502], ind1=tr, ind2=tr, mul=.
↪2).out()
```

setCarrier(*x*)

Replace the *carrier* attribute.

Args

x: float or PyoObject new *carrier* attribute.

setRatio(*x*)

Replace the *ratio* attribute.

Args

x: float or PyoObject new *ratio* attribute.

setInd1(*x*)

Replace the *ind1* attribute.

Args

x: float or PyoObject new *ind1* attribute.

setInd2(*x*)

Replace the *ind2* attribute.

Args

x: float or PyoObject new *ind2* attribute.

ctrl(*map_list=None*, *title=None*, *wxnoserver=False*)

Opens a sliders window to control the parameters of the object. Only parameters that can be set to a PyoObject are allowed to be mapped on a slider.

If a list of values are given to a parameter, a multisliders will be used to control each stream independently.

Args

map_list: list of SLMap objects, optional Users defined set of parameters scaling. There is default scaling for each object that accept *ctrl* method.

title: string, optional Title of the window. If none is provided, the name of the class is used.

wxnoserver: boolean, optional With wxPython graphical toolkit, if True, tells the interpreter that there will be no server window.

If *wxnoserver* is set to True, the interpreter will not wait for the server GUI before showing the controller window.

carrier
float or PyoObject. Carrier frequency in cycles per second.

ratio
float or PyoObject. Modulator/Carrier ratio.

ind1
float or PyoObject. Carrier index.

ind2
float or PyoObject. Modulation index.

FastSine

class FastSine(*freq=1000*, *initphase=0.0*, *quality=1*, *mul=1*, *add=0*)
A fast sine wave approximation using the formula of a parabola.

This object implements two sin approximations that are even faster than a linearly interpolated table lookup. With *quality* set to 1, the approximation is more accurate but also more expensive on the CPU (still cheaper than a Sine object). With *quality* = 0, the algorithm gives a worse approximation of the sin function but it is very fast (and well suitable for generating LFO).

Parent *PyoObject*

Args

freq: float or PyoObject, optional Frequency in cycles per second. Defaults to 1000.

initphase: float, optional Initial phase of the oscillator, between 0 and 1. Available at initialization time only. Defaults to 0.

quality: int, optional Sets the approximation quality. 1 is more accurate but also more expensive on the CPU. 0 is a cheaper algorithm but is very fast. Defaults to 1.

See also:

Sine

```
>>> s = Server().boot()
>>> s.start()
>>> lfo = FastSine(freq=[4,5], quality=0, mul=0.02, add=1)
>>> syn = FastSine(freq=500*lfo, quality=1, mul=0.4).out()
```

setFreq(*x*)
Replace the *freq* attribute.

Args

x: float or PyoObject new *freq* attribute.

setQuality(*x*)
Replace the *quality* attribute.

Args

x: int {0 or 1} new *quality* attribute.

reset()
Resets current phase to 0.

ctrl(*map_list=None*, *title=None*, *wxnoserver=False*)
Opens a sliders window to control the parameters of the object. Only parameters that can be set to a PyoObject are allowed to be mapped on a slider.

If a list of values are given to a parameter, a multisliders will be used to control each stream independently.

Args

map_list: list of SLMap objects, optional Users defined set of parameters scaling. There is default scaling for each object that accept *ctrl* method.

title: string, optional Title of the window. If none is provided, the name of the class is used.

wxnoserver: boolean, optional With wxPython graphical toolkit, if True, tells the interpreter that there will be no server window.

If *wxnoserver* is set to True, the interpreter will not wait for the server GUI before showing the controller window.

freq
float or PyoObject. Frequency in cycles per second.

quality
int. Quality of the sin approximation.

FM

class FM(*carrier=100*, *ratio=0.5*, *index=5*, *mul=1*, *add=0*)
A simple frequency modulation generator.

Implements frequency modulation synthesis based on Chowning's algorithm.

Parent `PyoObject`

Args

carrier: float or PyoObject, optional Carrier frequency in cycles per second. Defaults to 100.

ratio: float or PyoObject, optional A factor that, when multiplied by the *carrier* parameter, gives the modulator frequency. Defaults to 0.5.

index: float or PyoObject, optional The modulation index. This value multiplied by the modulator frequency gives the modulator amplitude. Defaults to 5.

```
>>> s = Server().boot()
>>> s.start()
>>> ind = LinTable([(0,3), (20,40), (300,10), (1000,5), (8191,3)])
>>> m = Metro(4).play()
>>> tr = TrigEnv(m, table=ind, dur=4)
>>> f = FM(carrier=[251,250], ratio=[.2498,.2503], index=tr, mul=.2).out()
```

setCarrier(*x*)
Replace the *carrier* attribute.

Args

x: float or PyoObject new *carrier* attribute.

setRatio(*x*)
Replace the *ratio* attribute.

Args

x: float or PyoObject new *ratio* attribute.

setIndex(*x*)
Replace the *index* attribute.

Args

x: float or PyoObject new *index* attribute.

ctrl(*map_list=None*, *title=None*, *wxnoserver=False*)
Opens a sliders window to control the parameters of the object. Only parameters that can be set to a PyoObject are allowed to be mapped on a slider.

If a list of values are given to a parameter, a multisliders will be used to control each stream independently.

Args

map_list: list of SLMap objects, optional Users defined set of parameters scaling. There is default scaling for each object that accept *ctrl* method.

title: string, optional Title of the window. If none is provided, the name of the class is used.

wxnoserver: boolean, optional With wxPython graphical toolkit, if True, tells the interpreter that there will be no server window.

If *wxnoserver* is set to True, the interpreter will not wait for the server GUI before showing the controller window.

carrier
float or PyoObject. Carrier frequency in cycles per second.

ratio
float or PyoObject. Modulator/Carrier ratio.

index
float or PyoObject. Modulation index.

Input

class Input(*chnl=0*, *mul=1*, *add=0*)
Read from a numbered channel in an external audio signal.

Parent *PyoObject*

Args

chnl: int, optional Input channel to read from. Defaults to 0.

Note: Requires that the Server's duplex mode is set to 1.

```
>>> s = Server(duplex=1).boot()
>>> s.start()
>>> a = Input(chnl=0, mul=.7)
>>> b = Delay(a, delay=.25, feedback=.5, mul=.5).out()
```

ctrl(*map_list=None*, *title=None*, *wxnoserver=False*)

Opens a sliders window to control the parameters of the object. Only parameters that can be set to a PyoObject are allowed to be mapped on a slider.

If a list of values are given to a parameter, a multisliders will be used to control each stream independently.

Args

map_list: list of SLMap objects, optional Users defined set of parameters scaling. There is default scaling for each object that accept *ctrl* method.

title: string, optional Title of the window. If none is provided, the name of the class is used.

wxnoserver: boolean, optional With wxPython graphical toolkit, if True, tells the interpreter that there will be no server window.

If *wxnoserver* is set to True, the interpreter will not wait for the server GUI before showing the controller window.

LFO

class LFO(*freq=100*, *sharp=0.5*, *type=0*, *mul=1*, *add=0*)

Band-limited Low Frequency Oscillator with different wave shapes.

Parent *PyoObject*

Args

freq: float or PyoObject, optional Oscillator frequency in cycles per second. The frequency is internally clamped between 0.00001 and sr/4. Defaults to 100.

sharp: float or PyoObject, optional Sharpness factor between 0 and 1. Sharper waveform results in more harmonics in the spectrum. Defaults to 0.5.

type: int, optional

Waveform type. eight possible values :

0. Saw up (default)
1. Saw down
2. Square
3. Triangle
4. Pulse

5. Bipolar pulse
6. Sample and hold
7. Modulated Sine

```
>>> s = Server().boot()
>>> s.start()
>>> lf = Sine([.31,.34], mul=15, add=20)
>>> lf2 = LFO([.43,.41], sharp=.7, type=2, mul=.4, add=.4)
>>> a = LFO(freq=lf, sharp=lf2, type=7, mul=100, add=300)
>>> b = SineLoop(freq=a, feedback=0.12, mul=.2).out()
```

setFreq(*x*)
Replace the *freq* attribute.

Args

x: float or PyoObject New *freq* attribute, in cycles per seconds.

setSharp(*x*)
Replace the *sharp* attribute.

Args

x: float or PyoObject New *sharp* attribute, in the range 0 -> 1.

setType(*x*)
Replace the *type* attribute.

Args

x: int

New *type* attribute. Choices are :

0. Saw up
1. Saw down
2. Square
3. Triangle
4. Pulse
5. Bipolar pulse
6. Sample and hold
7. Modulated Sine

reset()
Resets current phase to 0.

ctrl(*map_list=None*, *title=None*, *wxnoserver=False*)
Opens a sliders window to control the parameters of the object. Only parameters that can be set to a PyoObject are allowed to be mapped on a slider.

If a list of values are given to a parameter, a multisliders will be used to control each stream independently.

Args

map_list: list of SLMap objects, optional Users defined set of parameters scaling. There is default scaling for each object that accept *ctrl* method.

title: string, optional Title of the window. If none is provided, the name of the class is used.

wxnoserver: boolean, optional With wxPython graphical toolkit, if True, tells the interpreter that there will be no server window.

If *wxnoserver* is set to True, the interpreter will not wait for the server GUI before showing the controller window.

freq
float or PyoObject. Oscillator frequency in cycles per second.

sharp
float or PyoObject. Sharpness factor {0 -> 1}.

type
int. Waveform type.

Lorenz

class Lorenz(*pitch=0.25*, *chaos=0.5*, *stereo=False*, *mul=1*, *add=0*)
Chaotic attractor for the Lorenz system.

The Lorenz attractor is a system of three non-linear ordinary differential equations. These differential equations define a continuous-time dynamical system that exhibits chaotic dynamics associated with the fractal properties of the attractor.

Parent *PyoObject*

Args

pitch: float or PyoObject, optional Controls the speed, in the range 0 -> 1, of the variations. With values below 0.2, this object can be used as a low frequency oscillator (LFO) and above 0.2, it will generate a broad spectrum noise with harmonic peaks. Defaults to 0.25.

chaos: float or PyoObject, optional Controls the chaotic behavior, in the range 0 -> 1, of the oscillator. 0 means nearly periodic while 1 is totally chaotic. Defaults to 0.5

stereo, boolean, optional If True, 2 streams will be generated, one with the X variable signal of the algorithm and a second composed of the Y variable signal of the algorithm. These two signal are strongly related in their frequency spectrum but the Y signal is out-of-phase by approximatly 180 degrees. Useful to create alternating LFOs. Available at initialization only. Defaults to False.

See also:

Rossler, *ChenLee*

```
>>> s = Server().boot()
>>> s.start()
>>> a = Lorenz(pitch=.003, stereo=True, mul=.2, add=.2)
>>> b = Lorenz(pitch=[.4,.38], mul=a).out()
```

setPitch(*x*)
Replace the *pitch* attribute.

Args

x: float or PyoObject new *pitch* attribute. {0. -> 1.}

setChaos(*x*)
Replace the *chaos* attribute.

Args

x: float or PyoObject new *chaos* attribute. {0. -> 1.}

ctrl(*map_list=None*, *title=None*, *wxnoserver=False*)
Opens a sliders window to control the parameters of the object. Only parameters that can be set to a PyoObject are allowed to be mapped on a slider.

If a list of values are given to a parameter, a multisliders will be used to control each stream independently.

Args

map_list: list of SLMap objects, optional Users defined set of parameters scaling. There is default scaling for each object that accept *ctrl* method.

title: string, optional Title of the window. If none is provided, the name of the class is used.

wxnoserver: boolean, optional With wxPython graphical toolkit, if True, tells the interpreter that there will be no server window.

If *wxnoserver* is set to True, the interpreter will not wait for the server GUI before showing the controller window.

pitch
float or PyoObject. Speed of the variations.

chaos
float or PyoObject. Chaotic behavior.

Noise

class Noise(*mul=1*, *add=0*)
A white noise generator.

Parent *PyoObject*

```
>>> s = Server().boot()
>>> s.start()
>>> a = Noise(.1).mix(2).out()
```

setType(*x*)
Sets the generation algorithm.

Args

x: int, {0, 1} 0 uses the system rand() method to generate number. Used as default. 1 uses a simple linear congruential generator, cheaper than rand().

ctrl(*map_list=None*, *title=None*, *wxnoserver=False*)
Opens a sliders window to control the parameters of the object. Only parameters that can be set to a PyoObject are allowed to be mapped on a slider.

If a list of values are given to a parameter, a multisliders will be used to control each stream independently.

Args

map_list: list of SLMap objects, optional Users defined set of parameters scaling. There is default scaling for each object that accept *ctrl* method.

title: string, optional Title of the window. If none is provided, the name of the class is used.

wxnoserver: boolean, optional With wxPython graphical toolkit, if True, tells the interpreter that there will be no server window.

If *wxnoserver* is set to True, the interpreter will not wait for the server GUI before showing the controller window.

type
int {0, 1}. Sets the generation algorithm.

Phasor

class Phasor(*freq=100*, *phase=0*, *mul=1*, *add=0*)
A simple phase incrementor.

Output is a periodic ramp from 0 to 1.

Parent *PyoObject*

Args

freq: float or PyoObject, optional Frequency in cycles per second. Defaults to 100.

phase: float or PyoObject, optional Phase of sampling, expressed as a fraction of a cycle (0 to 1). Defaults to 0.

See also:

Osc, *Sine*

```
>>> s = Server().boot()
>>> s.start()
>>> f = Phasor(freq=[1, 1.5], mul=1000, add=500)
>>> sine = Sine(freq=f, mul=.2).out()
```

setFreq(*x*)
Replace the *freq* attribute.

Args

x: float or PyoObject new *freq* attribute.

setPhase(*x*)
Replace the *phase* attribute.

Args

x: float or PyoObject new *phase* attribute.

reset()
Resets current phase to 0.

ctrl(*map_list=None*, *title=None*, *wxnoserver=False*)
Opens a sliders window to control the parameters of the object. Only parameters that can be set to a PyoObject are allowed to be mapped on a slider.

If a list of values are given to a parameter, a multisliders will be used to control each stream independently.

Args

map_list: list of SLMap objects, optional Users defined set of parameters scaling. There is default scaling for each object that accept *ctrl* method.

title: string, optional Title of the window. If none is provided, the name of the class is used.

wxnoserver: boolean, optional With wxPython graphical toolkit, if True, tells the interpreter that there will be no server window.

If *wxnoserver* is set to True, the interpreter will not wait for the server GUI before showing the controller window.

freq
float or PyoObject. Frequency in cycles per second.

phase
float or PyoObject. Phase of sampling.

PinkNoise

class PinkNoise(*mul=1*, *add=0*)
A pink noise generator.

Paul Kellet's implementation of pink noise generator.

This is an approximation to a -10dB/decade filter using a weighted sum of first order filters. It is accurate to within +/-0.05dB above 9.2Hz (44100Hz sampling rate).

Parent *PyoObject*

```
>>> s = Server().boot()
>>> s.start()
>>> a = PinkNoise(.1).mix(2).out()
```

ctrl(*map_list=None*, *title=None*, *wxnoserver=False*)
Opens a sliders window to control the parameters of the object. Only parameters that can be set to a PyoObject are allowed to be mapped on a slider.

If a list of values are given to a parameter, a multisliders will be used to control each stream independently.

Args

map_list: list of SLMap objects, optional Users defined set of parameters scaling. There is default scaling for each object that accept *ctrl* method.

title: string, optional Title of the window. If none is provided, the name of the class is used.

wxnoserver: boolean, optional With wxPython graphical toolkit, if True, tells the interpreter that there will be no server window.

If *wxnoserver* is set to True, the interpreter will not wait for the server GUI before showing the controller window.

RCOsc

class RCOsc(*freq=100*, *sharp=0.25*, *mul=1*, *add=0*)
Waveform aproximation of a RC circuit.

A RC circuit is a capacitor and a resistor in series, giving a logarithmic growth followed by an exponential decay.

Parent *PyoObject*

Args

freq: float or PyoObject, optional Frequency in cycles per second. Defaults to 100.

sharp: float or PyoObject, optional Slope of the attack and decay of the waveform, between 0 and 1. A value of 0 gives a triangular waveform and 1 gives almost a square wave. Defaults to 0.25.

See also:

Osc, *LFO*, *SineLoop*, *SumOsc*

```
>>> s = Server().boot()
>>> s.start()
>>> fr = RCOsc(freq=[.48,.5], sharp=.2, mul=300, add=600)
>>> a = RCOsc(freq=fr, sharp=.1, mul=.2).out()
```

setFreq(*x*)
Replace the *freq* attribute.

Args

x: float or PyoObject new *freq* attribute.

setSharp(*x*)
Replace the *sharp* attribute.

Args

x: float or PyoObject new *sharp* attribute.

reset()
Resets current phase to 0.

ctrl(*map_list=None*, *title=None*, *wxnoserver=False*)
Opens a sliders window to control the parameters of the object. Only parameters that can be set to a PyoObject are allowed to be mapped on a slider.

If a list of values are given to a parameter, a multisliders will be used to control each stream independently.

Args

map_list: list of SLMap objects, optional Users defined set of parameters scaling. There is default scaling for each object that accept *ctrl* method.

title: string, optional Title of the window. If none is provided, the name of the class is used.

wxnoserver: boolean, optional With wxPython graphical toolkit, if True, tells the interpreter that there will be no server window.

If *wxnoserver* is set to True, the interpreter will not wait for the server GUI before showing the controller window.

freq
float or PyoObject. Frequency in cycles per second.

sharp
float or PyoObject. Sharpness of the waveform.

Rossler

class Rossler(*pitch=0.25*, *chaos=0.5*, *stereo=False*, *mul=1*, *add=0*)

Chaotic attractor for the Rossler system.

The Rossler attractor is a system of three non-linear ordinary differential equations. These differential equations define a continuous-time dynamical system that exhibits chaotic dynamics associated with the fractal properties of the attractor.

Parent *PyoObject*

Args

pitch: float or PyoObject, optional Controls the speed, in the range 0 -> 1, of the variations. With values below 0.2, this object can be used as a low frequency oscillator (LFO) and above 0.2, it will generate a broad spectrum noise with harmonic peaks. Defaults to 0.25.

chaos: float or PyoObject, optional Controls the chaotic behavior, in the range 0 -> 1, of the oscillator. 0 means nearly periodic while 1 is totally chaotic. Defaults to 0.5.

stereo, boolean, optional If True, 2 streams will be generated, one with the X variable signal of the algorithm and a second composed of the Y variable signal of the algorithm. These two signal are strongly related in their frequency spectrum but the Y signal is out-of-phase by approximatly 180 degrees. Useful to create alternating LFOs. Available at initialization only. Defaults to False.

See also:

Lorenz, *ChenLee*

```
>>> s = Server().boot()
>>> s.start()
>>> a = Rossler(pitch=.003, stereo=True, mul=.2, add=.2)
>>> b = Rossler(pitch=[.5,.48], mul=a).out()
```

setPitch(*x*)

Replace the *pitch* attribute.

Args

x: float or PyoObject new *pitch* attribute. {0. -> 1.}

setChaos(*x*)

Replace the *chaos* attribute.

Args

x: float or PyoObject new *chaos* attribute. {0. -> 1.}

ctrl(*map_list=None*, *title=None*, *wxnoserver=False*)

Opens a sliders window to control the parameters of the object. Only parameters that can be set to a PyoObject are allowed to be mapped on a slider.

If a list of values are given to a parameter, a multisliders will be used to control each stream independently.

Args

map_list: list of SLMap objects, optional Users defined set of parameters scaling. There is default scaling for each object that accept *ctrl* method.

title: string, optional Title of the window. If none is provided, the name of the class is used.

wxnoserver: boolean, optional With wxPython graphical toolkit, if True, tells the interpreter that there will be no server window.

If *wxnoserver* is set to True, the interpreter will not wait for the server GUI before showing the controller window.

pitch
float or PyoObject. Speed of the variations.

chaos
float or PyoObject. Chaotic behavior.

Sine

class Sine(*freq=1000*, *phase=0*, *mul=1*, *add=0*)
A simple sine wave oscillator.

Parent *PyoObject*

Args

freq: float or PyoObject, optional Frequency in cycles per second. Defaults to 1000.

phase: float or PyoObject, optional Phase of sampling, expressed as a fraction of a cycle (0 to 1). Defaults to 0.

See also:

Osc, *Phasor*

```
>>> s = Server().boot()
>>> s.start()
>>> sine = Sine(freq=[400,500], mul=.2).out()
```

setFreq(*x*)
Replace the *freq* attribute.

Args

x: float or PyoObject new *freq* attribute.

setPhase(*x*)
Replace the *phase* attribute.

Args

x: float or PyoObject new *phase* attribute.

reset()
Resets current phase to 0.

ctrl(*map_list=None*, *title=None*, *wxnoserver=False*)
Opens a sliders window to control the parameters of the object. Only parameters that can be set to a PyoObject are allowed to be mapped on a slider.

If a list of values are given to a parameter, a multisliders will be used to control each stream independently.

Args

map_list: list of SLMap objects, optional Users defined set of parameters scaling. There is default scaling for each object that accept *ctrl* method.

title: string, optional Title of the window. If none is provided, the name of the class is used.

wxnoserver: boolean, optional With wxPython graphical toolkit, if True, tells the interpreter that there will be no server window.

If *wxnoserver* is set to True, the interpreter will not wait for the server GUI before showing the controller window.

freq
float or PyoObject. Frequency in cycles per second.

phase
float or PyoObject. Phase of sampling.

SineLoop

class SineLoop(*freq=1000*, *feedback=0*, *mul=1*, *add=0*)
A simple sine wave oscillator with feedback.

The oscillator output, multiplied by *feedback*, is added to the position increment and can be used to control the brightness of the oscillator.

Parent *PyoObject*

Args

freq: float or PyoObject, optional Frequency in cycles per second. Defaults to 1000.

feedback: float or PyoObject, optional Amount of the output signal added to position increment, between 0 and 1. Controls the brightness. Defaults to 0.

See also:

Sine, *OscLoop*

```
>>> s = Server().boot()
>>> s.start()
>>> lfo = Sine(.25, 0, .1, .1)
>>> a = SineLoop(freq=[400,500], feedback=lfo, mul=.2).out()
```

setFreq(*x*)
Replace the *freq* attribute.

Args

x: float or PyoObject new *freq* attribute.

setFeedback(*x*)
Replace the *feedback* attribute.

Args

x: float or PyoObject new *feedback* attribute.

ctrl(*map_list=None*, *title=None*, *wxnoserver=False*)
Opens a sliders window to control the parameters of the object. Only parameters that can be set to a PyoObject are allowed to be mapped on a slider.

If a list of values are given to a parameter, a multisliders will be used to control each stream independently.

Args

map_list: list of SLMap objects, optional Users defined set of parameters scaling. There is default scaling for each object that accept *ctrl* method.

title: string, optional Title of the window. If none is provided, the name of the class is used.

wxnoserver: boolean, optional With wxPython graphical toolkit, if True, tells the interpreter that there will be no server window.

If *wxnoserver* is set to True, the interpreter will not wait for the server GUI before showing the controller window.

freq
float or PyoObject. Frequency in cycles per second.

feedback
float or PyoObject. Brightness control.

SumOsc

class SumOsc(*freq=100*, *ratio=0.5*, *index=0.5*, *mul=1*, *add=0*)
Discrete summation formulae to produce complex spectra.

This object implements a discrete summation formulae taken from the paper 'The synthesis of complex audio spectra by means of discrete summation formulae' by James A. Moorer. The formulae used is of this form:

(sin(theta) - a * sin(theta - beta)) / (1 + a**2 - 2 * a * cos(beta))

where 'theta' and 'beta' are periodic functions and 'a' is the modulation index, providing control over the damping of the partials.

The resulting sound is related to the family of modulation techniques but this formulae express 'one-sided' spectra, useful to avoid aliasing from the negative frequencies.

Parent *PyoObject*

Args

freq: float or PyoObject, optional Base frequency in cycles per second. Defaults to 100.

ratio: float or PyoObject, optional A factor used to stretch or compress the partial serie by manipulating the frequency of the modulation oscillator. Integer ratios give harmonic spectra. Defaults to 0.5.

index: float or PyoObject, optional Damping of successive partials, between 0 and 1. With a value of 0.5, each partial is 6dB lower than the previous partial. Defaults to 0.5.

```
>>> s = Server().boot()
>>> s.start()
>>> ind = LinTable([(0,.3), (20,.85), (300,.7), (1000,.5), (8191,.3)])
>>> m = Metro(4).play()
>>> tr = TrigEnv(m, table=ind, dur=4)
>>> f = SumOsc(freq=[301,300], ratio=[.2498,.2503], index=tr, mul=.2).out()
```

setFreq(*x*)
Replace the *freq* attribute.

Args

x: float or PyoObject new *freq* attribute.

setRatio(*x*)
Replace the *ratio* attribute.

Args

x: float or PyoObject new *ratio* attribute.

setIndex(*x*)
Replace the *index* attribute.

Args

x: float or PyoObject new *index* attribute.

ctrl(*map_list=None*, *title=None*, *wxnoserver=False*)
Opens a sliders window to control the parameters of the object. Only parameters that can be set to a PyoObject are allowed to be mapped on a slider.

If a list of values are given to a parameter, a multisliders will be used to control each stream independently.

Args

map_list: list of SLMap objects, optional Users defined set of parameters scaling. There is default scaling for each object that accept *ctrl* method.

title: string, optional Title of the window. If none is provided, the name of the class is used.

wxnoserver: boolean, optional With wxPython graphical toolkit, if True, tells the interpreter that there will be no server window.

If *wxnoserver* is set to True, the interpreter will not wait for the server GUI before showing the controller window.

freq
float or PyoObject. Base frequency in cycles per second.

ratio
float or PyoObject. Base/modulator frequency ratio.

index
float or PyoObject. Index, high frequency damping.

SuperSaw

class SuperSaw(*freq=100*, *detune=0.5*, *bal=0.7*, *mul=1*, *add=0*)
Roland JP-8000 Supersaw emulator.

This object implements an emulation of the Roland JP-8000 Supersaw algorithm. The shape of the waveform is produced from 7 sawtooth oscillators detuned against each other over a period of time. It allows control over the depth of the detuning and the balance between central and sideband oscillators.

Parent `PyoObject`

Args

freq: float or PyoObject, optional Frequency in cycles per second. Defaults to 100.

detune: float or PyoObject, optional Depth of the detuning, between 0 and 1. 0 means all oscillators are tuned to the same frequency and 1 means sideband oscillators are at maximum detuning regarding the central frequency. Defaults to 0.5.

bal: float or PyoObject, optional Balance between central oscillator and sideband oscillators. A value of 0 outputs only the central oscillator while a value of 1 gives a mix of all oscillators with the central one lower than the sidebands. Defaults to 0.7.

See also:

Phasor, *SineLoop*

```
>>> s = Server().boot()
>>> s.start()
>>> lfd = Sine([.4,.3], mul=.2, add=.5)
>>> a = SuperSaw(freq=[49,50], detune=lfd, bal=0.7, mul=0.2).out()
```

setFreq(*x*)
Replace the *freq* attribute.

Args

x: float or PyoObject new *freq* attribute.

setDetune(*x*)
Replace the *detune* attribute.

Args

x: float or PyoObject new *detune* attribute.

setBal(*x*)
Replace the *bal* attribute.

Args

x: float or PyoObject new *bal* attribute.

ctrl(*map_list=None*, *title=None*, *wxnoserver=False*)
Opens a sliders window to control the parameters of the object. Only parameters that can be set to a PyoObject are allowed to be mapped on a slider.

If a list of values are given to a parameter, a multisliders will be used to control each stream independently.

Args

map_list: list of SLMap objects, optional Users defined set of parameters scaling. There is default scaling for each object that accept *ctrl* method.

title: string, optional Title of the window. If none is provided, the name of the class is used.

wxnoserver: boolean, optional With wxPython graphical toolkit, if True, tells the interpreter that there will be no server window.

If *wxnoserver* is set to True, the interpreter will not wait for the server GUI before showing the controller window.

freq
float or PyoObject. Frequency in cycles per second.

detune
float or PyoObject. Depth of the detuning.

bal
float or PyoObject. Balance between central and sideband oscillators.

Internal objects

These objects are mainly used by pyo itself, inside other objects.

Dummy

class Dummy(*objs_list*)

Dummy object used to perform arithmetics on PyoObject.

The user should never instantiate an object of this class.

Parent *PyoObject*

Args

objs_list: list of audio Stream objects List of Stream objects return by the PyoObject hidden method getBaseObjects().

Note: Multiplication, addition, division and substraction don't changed the PyoObject on which the operation is performed. A dummy object is created, which is just a copy of the audio Streams of the object, and the operation is applied on the Dummy, leaving the original object unchanged. This lets the user performs multiple different arithmetic operations on an object without conficts. Here, *b* is a Dummy object with *a* as its input with a *mul* attribute of 0.5. attribute:

```
>>> a = Sine()
>>> b = a * .5
>>> print(a)
<pyo.lib.input.Sine object at 0x11fd610>
>>> print(b)
<pyo.lib._core.Dummy object at 0x11fd710>
```

```
>>> s = Server().boot()
>>> s.start()
>>> m = Metro(time=0.25).play()
>>> p = TrigChoice(m, choice=[midiToHz(n) for n in [60,62,65,67,69]])
>>> a = SineLoop(p, feedback=.05, mul=.1).mix(2).out()
>>> b = SineLoop(p*1.253, feedback=.05, mul=.06).mix(2).out()
>>> c = SineLoop(p*1.497, feedback=.05, mul=.03).mix(2).out()
```

InputFader

class InputFader(*input*)

Audio streams crossfader.

Args

input: PyoObject Input signal.

Note: The setInput method, available to object with *input* attribute, uses an InputFader object internally to perform crossfade between the old and the new audio input assigned to the object.

```
>>> s = Server().boot()
>>> s.start()
>>> a = SineLoop([449,450], feedback=0.05, mul=.2)
>>> b = SineLoop([650,651], feedback=0.05, mul=.2)
>>> c = InputFader(a).out()
>>> # to created a crossfade, assign a new audio input:
>>> c.setInput(b, fadetime=5)
```

setInput(*x*, *fadetime=0.05*)
Replace the *input* attribute.

Args

x: PyoObject New signal to process.

fadetime: float, optional Crossfade time between old and new input. Defaults to 0.05.

input
PyoObject. Input signal.

Mix

class Mix(*input*, *voices=1*, *mul=1*, *add=0*)
Mix audio streams to arbitrary number of streams.

Mix the object's audio streams as *input* argument into *voices* streams.

Parent *PyoObject*

Args

input: PyoObject or list of PyoObjects Input signal(s) to mix the streams.

voices: int, optional Number of streams of the Mix object. If more than 1, input object's streams are alternated and added into Mix object's streams. Defaults to 1.

Note: The mix method of PyoObject creates and returns a new Mix object with mixed streams of the object that called the method. User don't have to instantiate this class directly. These two calls are identical:

```
>>> b = a.mix()
>>> b = Mix(a)
```

```
>>> s = Server().boot()
>>> s.start()
>>> a = Sine([random.uniform(400,600) for i in range(50)], mul=.02)
>>> b = Mix(a, voices=2).out()
>>> print(len(a))
50
>>> print(len(b))
1
```

VarPort

class VarPort(*value*, *time=0.025*, *init=0.0*, *function=None*, *arg=None*, *mul=1*, *add=0*)
Convert numeric value to PyoObject signal with portamento.

When *value* attribute is changed, a smoothed ramp is applied from the current value to the new value. If a callback is provided as *function* argument, it will be called at the end of the line.

Parent *PyoObject*

Args

value: float Numerical value to convert.

time: float, optional Ramp time, in seconds, to reach the new value. Defaults to 0.025.

init: float, optional Initial value of the internal memory. Defaults to 0.

function: Python callable, optional If provided, it will be called at the end of the line. Defaults to None.

arg: any Python object, optional Optional argument sent to the function called at the end of the line. Defaults to None.

Note: The out() method is bypassed. VarPort's signal can not be sent to audio outs.

```
>>> s = Server().boot()
>>> s.start()
>>> def callback(arg):
...     print("end of line")
...     print(arg)
...
>>> fr = VarPort(value=500, time=2, init=250, function=callback, arg="YEP!")
>>> a = SineLoop(freq=[fr,fr*1.01], feedback=0.05, mul=.2).out()
```

setValue(*x*)
Changes the value of the signal stream.

Args

x: float Numerical value to convert.

setTime(*x*)
Changes the ramp time of the object.

Args

x: float New ramp time.

setFunction(*x*)
Replace the *function* attribute.

Args

x: Python function new *function* attribute.

value
float. Numerical value to convert.

time
float. Ramp time.

function
Python callable. Function to be called.

Stream

class **Stream**

Audio stream objects. For internal use only.

A Stream object must never be instantiated by the user.

A Stream is a mono buffer of audio samples. It is used to pass audio between objects and the server. A PyoObject can manage many streams if, for example, a list is given to a parameter.

A Sine object with only one stream:

```
>>> a = Sine(freq=1000)
>>> print len(a)
1
```

A Sine object with four streams:

```
>>> a = Sine(freq=[250,500,750,100])
>>> print len(a)
4
```

The first stream of this object contains the samples from the 250Hz waveform. The second stream contains the samples from the 500Hz waveform, and so on.

User can call a specific stream of an object by giving the position of the stream between brackets, beginning at 0. To retrieve only the third stream of our object:

```
>>> a[2].out()
```

The method getStreamObject() can be called on a Stream object to retrieve the XXX_base object associated with this Stream. This method can be used by developers who are debugging their programs!

Matrix Processing

PyoObjects to perform operations on PyoMatrixObjects.

PyoMatrixObjects are 2 dimensions table containers. They can be used to store audio samples or algorithmic sequences. Writing and reading are done by giving row and column positions.

MatrixMorph

class MatrixMorph(*input*, *matrix*, *sources*)

Morphs between multiple PyoMatrixObjects.

Uses an index into a list of PyoMatrixObjects to morph between adjacent matrices in the list. The resulting morphed function is written into the *matrix* object at the beginning of each buffer size. The matrices in the list and the resulting matrix must be equal in size.

Parent *PyoObject*

Args

input: PyoObject Morphing index between 0 and 1. 0 is the first matrix in the list and 1 is the last.

matrix: NewMatrix The matrix where to write morphed function.

sources: list of PyoMatrixObject List of matrices to interpolate from.

Note: The out() method is bypassed. MatrixMorph returns no signal.

MatrixMorph has no *mul* and *add* attributes.

```
>>> s = Server().boot()
>>> s.start()
>>> m1 = NewMatrix(256, 256)
>>> m1.genSineTerrain(1, 4)
>>> m2 = NewMatrix(256, 256)
>>> m2.genSineTerrain(2, 8)
>>> mm = NewMatrix(256, 256)
>>> inter = Sine(.2, 0, .5, .5)
>>> morph = MatrixMorph(inter, mm, [m1,m2])
>>> x = Sine([49,50], 0, .45, .5)
>>> y = Sine([49.49,50.5], 0, .45, .5)
>>> c = MatrixPointer(mm, x, y, .2).out()
```

out (*chnl=0*, *inc=1*, *dur=0*, *delay=0*)

Start processing and send samples to audio output beginning at *chnl*.

This method returns *self*, allowing it to be applied at the object creation.

Args

chnl: int, optional Physical output assigned to the first audio stream of the object. Defaults to 0.

inc: int, optional Output channel increment value. Defaults to 1.

dur: float, optional Duration, in seconds, of the object's activation. The default is 0 and means infinite duration.

delay: float, optional Delay, in seconds, before the object's activation. Defaults to 0.

If *chnl* >= 0, successive streams increment the output number by *inc* and wrap around the global number of channels.

If *chnl* is negative, streams begin at 0, increment the output number by *inc* and wrap around the global number of channels. Then, the list of streams is scrambled.

If *chnl* is a list, successive values in the list will be assigned to successive streams.

setMul (*x*)

Replace the *mul* attribute.

Args

x: float or PyoObject New *mul* attribute.

setAdd (*x*)

Replace the *add* attribute.

Args

x: float or PyoObject New *add* attribute.

setInput (*x*, *fadetime=0.05*)

Replace the *input* attribute.

Args

x: PyoObject New signal to process.

fadetime: float, optional Crossfade time between old and new input. Defaults to 0.05.

setMatrix(*x*)
Replace the *matrix* attribute.

Args

x: NewMatrix new *matrix* attribute.

setSources(*x*)

Replace the *sources* attribute.

Args

x: list of PyoMatrixObject new *sources* attribute.

input
PyoObject. Morphing index between 0 and 1.

matrix
NewMatrix. The matrix where to write samples.

sources
list of PyoMatrixObject. List of matrices to interpolate from.

MatrixPointer

class MatrixPointer(*matrix*, *x*, *y*, *mul=1*, *add=0*)
Matrix reader with control on the 2D pointer position.

Parent *PyoObject*

Args

matrix: PyoMatrixObject Matrix containing the waveform samples.

x: PyoObject Normalized X position in the matrix between 0 and 1.

y: PyoObject Normalized Y position in the matrix between 0 and 1.

```
>>> s = Server().boot()
>>> s.start()
>>> SIZE = 256
>>> mm = NewMatrix(SIZE, SIZE)
>>> fmind = Sine(.2, 0, 2, 2.5)
>>> fmrat = Sine(.33, 0, .05, .5)
>>> aa = FM(carrier=10, ratio=fmrat, index=fmind)
>>> rec = MatrixRec(aa, mm, 0).play()
>>> lfx = Sine(.1, 0, .24, .25)
>>> lfy = Sine(.15, 0, .124, .25)
>>> x = Sine([500,501], 0, lfx, .5)
>>> y = Sine([10.5,10], 0, lfy, .5)
>>> c = MatrixPointer(mm, x, y, .2).out()
```

setMatrix(*x*)
Replace the *matrix* attribute.

Args

x: PyoTableObject new *matrix* attribute.

setX(*x*)
Replace the *x* attribute.

Args

x: PyoObject new *x* attribute.

setY(*x*)
Replace the *y* attribute.

Args

y: PyoObject new *y* attribute.

ctrl(*map_list=None*, *title=None*, *wxnoserver=False*)
Opens a sliders window to control the parameters of the object. Only parameters that can be set to a PyoObject are allowed to be mapped on a slider.

If a list of values are given to a parameter, a multisliders will be used to control each stream independently.

Args

map_list: list of SLMap objects, optional Users defined set of parameters scaling. There is default scaling for each object that accept *ctrl* method.

title: string, optional Title of the window. If none is provided, the name of the class is used.

wxnoserver: boolean, optional With wxPython graphical toolkit, if True, tells the interpreter that there will be no server window.

If *wxnoserver* is set to True, the interpreter will not wait for the server GUI before showing the controller window.

matrix
PyoMatrixObject. Matrix containing the samples.

x
PyoObject. Normalized X position in the matrix.

y
PyoObject. Normalized Y position in the matrix.

MatrixRec

class MatrixRec(*input*, *matrix*, *fadetime=0*, *delay=0*)
MatrixRec records samples into a previously created NewMatrix.

See `NewMatrix` to create an empty matrix.

The play method is not called at the object creation time. It starts the recording into the matrix, row after row, until the matrix is full. Calling the play method again restarts the recording and overwrites previously recorded samples. The stop method stops the recording. Otherwise, the default behaviour is to record through the end of the matrix.

Parent `PyoObject`

Args

input: PyoObject Audio signal to write in the matrix.

matrix: PyoMatrixObject The matrix where to write samples.

fadetime: float, optional Fade time at the beginning and the end of the recording in seconds. Defaults to 0.

delay: int, optional Delay time, in samples, before the recording begins. Available at initialization time only. Defaults to 0.

Note: The out() method is bypassed. MatrixRec returns no signal.

MatrixRec has no *mul* and *add* attributes.

MatrixRec will send a trigger signal at the end of the recording. User can retreive the trigger streams by calling obj['trig']. See *TableRec* documentation for an example.

See also:

NewMatrix

```
>>> s = Server().boot()
>>> s.start()
>>> SIZE = 256
>>> mm = NewMatrix(SIZE, SIZE)
>>> fmind = Sine(.2, 0, 2, 2.5)
>>> fmrat = Sine(.33, 0, .05, .5)
>>> aa = FM(carrier=10, ratio=fmrat, index=fmind)
>>> rec = MatrixRec(aa, mm, 0).play()
>>> lfx = Sine(.1, 0, .24, .25)
>>> lfy = Sine(.15, 0, .124, .25)
>>> x = Sine([500,501], 0, lfx, .5)
>>> y = Sine([10.5,10], 0, lfy, .5)
>>> c = MatrixPointer(mm, x, y, .2).out()
```

out (*chnl=0*, *inc=1*, *dur=0*, *delay=0*)

Start processing and send samples to audio output beginning at *chnl*.

This method returns *self*, allowing it to be applied at the object creation.

Args

chnl: int, optional Physical output assigned to the first audio stream of the object. Defaults to 0.

inc: int, optional Output channel increment value. Defaults to 1.

dur: float, optional Duration, in seconds, of the object's activation. The default is 0 and means infinite duration.

delay: float, optional Delay, in seconds, before the object's activation. Defaults to 0.

If *chnl* >= 0, successive streams increment the output number by *inc* and wrap around the global number of channels.

If *chnl* is negative, streams begin at 0, increment the output number by *inc* and wrap around the global number of channels. Then, the list of streams is scrambled.

If *chnl* is a list, successive values in the list will be assigned to successive streams.

setMul (*x*)

Replace the *mul* attribute.

Args

x: float or PyoObject New *mul* attribute.

setAdd(*x*)
Replace the *add* attribute.

Args

x: float or PyoObject New *add* attribute.

setInput(*x*, *fadetime=0.05*)
Replace the *input* attribute.

Args

x: PyoObject New signal to process.

fadetime: float, optional Crossfade time between old and new input. Defaults to 0.05.

setMatrix(*x*)
Replace the *matrix* attribute.

Args

x: NewMatrix new *matrix* attribute.

input
PyoObject. Audio signal to record in the matrix.

matrix
PyoMatrixObject. The matrix where to write samples.

MatrixRecLoop

class MatrixRecLoop(*input*, *matrix*)
MatrixRecLoop records samples in loop into a previously created NewMatrix.

See *NewMatrix* to create an empty matrix.

MatrixRecLoop records samples into the matrix, row after row, until the matrix is full and then loop back to the beginning.

Parent *PyoObject*

Args

input: PyoObject Audio signal to write in the matrix.

matrix: PyoMatrixObject The matrix where to write samples.

Note: The out() method is bypassed. MatrixRecLoop returns no signal.

MatrixRecLoop has no *mul* and *add* attributes.

MatrixRecLoop will send a trigger signal when reaching the end of the matrix. User can retreive the trigger streams by calling obj['trig']. See *TableRec* documentation for an example.

See also:

NewMatrix

```
>>> s = Server().boot()
>>> s.start()
>>> env = CosTable([(0,0), (300,1), (1000,.4), (8191,0)])
>>> matrix = NewMatrix(8192, 8)
```

(continues on next page)

(continued from previous page)

```
>>> src = SfPlayer(SNDS_PATH+'/transparent.aif', loop=True, mul=.3)
>>> m_rec = MatrixRecLoop(src, matrix)
>>> period = 8192 / s.getSamplingRate()
>>> metro = Metro(time=period/2, poly=2).play()
>>> x = TrigLinseg(metro, [(0,0), (period,1)])
>>> y = TrigRandInt(metro, max=2, mul=0.125)
>>> amp = TrigEnv(metro, table=env, dur=period)
>>> out = MatrixPointer(matrix, x, y, amp).out()
```

out (*chnl=0*, *inc=1*, *dur=0*, *delay=0*)

Start processing and send samples to audio output beginning at *chnl*.

This method returns *self*, allowing it to be applied at the object creation.

Args

chnl: int, optional Physical output assigned to the first audio stream of the object. Defaults to 0.

inc: int, optional Output channel increment value. Defaults to 1.

dur: float, optional Duration, in seconds, of the object's activation. The default is 0 and means infinite duration.

delay: float, optional Delay, in seconds, before the object's activation. Defaults to 0.

If *chnl* >= 0, successive streams increment the output number by *inc* and wrap around the global number of channels.

If *chnl* is negative, streams begin at 0, increment the output number by *inc* and wrap around the global number of channels. Then, the list of streams is scrambled.

If *chnl* is a list, successive values in the list will be assigned to successive streams.

setMul (*x*)

Replace the *mul* attribute.

Args

x: float or PyoObject New *mul* attribute.

setAdd (*x*)

Replace the *add* attribute.

Args

x: float or PyoObject New *add* attribute.

setInput (*x*, *fadetime=0.05*)

Replace the *input* attribute.

Args

x: PyoObject New signal to process.

fadetime: float, optional Crossfade time between old and new input. Defaults to 0.05.

setMatrix (*x*)

Replace the *matrix* attribute.

Args

x: NewMatrix new *matrix* attribute.

input
PyoObject. Audio signal to record in the matrix.

matrix
PyoMatrixObject. The matrix where to write samples.

Midi Handling

Objects to retrieve Midi informations for a specific Midi port and channel.

Objects creates and returns audio streams from the value in their Midi input.

The audio streams of these objects are essentially intended to be used as controls and can't be sent to the output soundcard.

Bendin

class Bendin(*brange=2*, *scale=0*, *channel=0*, *mul=1*, *add=0*)
Get the current value of the pitch bend controller.

Get the current value of the pitch bend controller and optionally maps it inside a specified range.

Parent *PyoObject*

Args

brange: float, optional Bipolar range of the pitch bend in semitones. Defaults to 2. -brange <= value < brange.

scale: int, optional

Output format. Defaults to 0.

0. Midi

1. transpo.

The transpo mode is useful if you want to transpose values that are in a frequency (Hz) format.

channel: int, optional Midi channel. 0 means all channels. Defaults to 0.

Note: The out() method is bypassed. Bendin's signal can not be sent to audio outs.

```
>>> s = Server().boot()
>>> s.start()
>>> notes = Notein(poly=10, scale=1, mul=.5)
>>> bend = Bendin(brange=2, scale=1)
>>> p = Port(notes['velocity'], .001, .5)
>>> b = Sine(freq=notes['pitch'] * bend, mul=p).out()
>>> c = Sine(freq=notes['pitch'] * bend * 0.997, mul=p).out()
>>> d = Sine(freq=notes['pitch'] * bend * 1.005, mul=p).out()
```

out(*chnl=0*, *inc=1*, *dur=0*, *delay=0*)
Start processing and send samples to audio output beginning at *chnl*.

This method returns *self*, allowing it to be applied at the object creation.

Args

chnl: int, optional Physical output assigned to the first audio stream of the object. Defaults to 0.

inc: int, optional Output channel increment value. Defaults to 1.

dur: float, optional Duration, in seconds, of the object's activation. The default is 0 and means infinite duration.

delay: float, optional Delay, in seconds, before the object's activation. Defaults to 0.

If *chnl* >= 0, successive streams increment the output number by *inc* and wrap around the global number of channels.

If *chnl* is negative, streams begin at 0, increment the output number by *inc* and wrap around the global number of channels. Then, the list of streams is scrambled.

If *chnl* is a list, successive values in the list will be assigned to successive streams.

setBrange(*x*)
Replace the *brange* attribute.

Args

x: float new *brange* attribute.

setScale(*x*)
Replace the *scale* attribute.

Args

x: int new *scale* attribute.

setChannel(*x*)
Replace the *channel* attribute.

Args

x: int new *channel* attribute.

setInterpolation(*x*)
Deprecated method. If needed, use Port or SigTo to interpolate between values.

ctrl(*map_list=None*, *title=None*, *wxnoserver=False*)
Opens a sliders window to control the parameters of the object. Only parameters that can be set to a PyoObject are allowed to be mapped on a slider.

If a list of values are given to a parameter, a multisliders will be used to control each stream independently.

Args

map_list: list of SLMap objects, optional Users defined set of parameters scaling. There is default scaling for each object that accept *ctrl* method.

title: string, optional Title of the window. If none is provided, the name of the class is used.

wxnoserver: boolean, optional With wxPython graphical toolkit, if True, tells the interpreter that there will be no server window.

If *wxnoserver* is set to True, the interpreter will not wait for the server GUI before showing the controller window.

brange
float. Bipolar range of the pitch bend in semitones.

scale
int. Output format. 0 = Midi, 1 = transpo.

channel
int. Midi channel. 0 means all channels.

CtlScan

class CtlScan(*function*, *toprint=True*)
Scan the Midi controller's number in input.

Scan the Midi controller's number in input and send it to a standard python *function*. Useful to implement a MidiLearn algorithm.

Parent *PyoObject*

Args

function: Python function (can't be a list) Function to be called. The function must be declared with an argument for the controller number in input. Ex.:

def ctl_scan(ctlnum): print(ctlnum)

toprint: boolean, optional If True, controller number and value will be printed to the console.

Note: The out() method is bypassed. CtlScan's signal can not be sent to audio outs.

```
>>> s = Server()
>>> s.setMidiInputDevice(0) # enter your device number (see pm_list_devices())
>>> s.boot()
>>> s.start()
>>> def ctl_scan(ctlnum):
...     print(ctlnum)
>>> a = CtlScan(ctl_scan)
```

out(*chnl=0*, *inc=1*, *dur=0*, *delay=0*)
Start processing and send samples to audio output beginning at *chnl*.

This method returns *self*, allowing it to be applied at the object creation.

Args

chnl: int, optional Physical output assigned to the first audio stream of the object. Defaults to 0.

inc: int, optional Output channel increment value. Defaults to 1.

dur: float, optional Duration, in seconds, of the object's activation. The default is 0 and means infinite duration.

delay: float, optional Delay, in seconds, before the object's activation. Defaults to 0.

If *chnl* >= 0, successive streams increment the output number by *inc* and wrap around the global number of channels.

If *chnl* is negative, streams begin at 0, increment the output number by *inc* and wrap around the global number of channels. Then, the list of streams is scrambled.

If *chnl* is a list, successive values in the list will be assigned to successive streams.

setMul(*x*)
Replace the *mul* attribute.

Args

x: float or PyoObject New *mul* attribute.

setAdd(*x*)
Replace the *add* attribute.

Args

x: float or PyoObject New *add* attribute.

setSub(*x*)
Replace and inverse the *add* attribute.

Args

x: float or PyoObject New inversed *add* attribute.

setDiv(*x*)
Replace and inverse the *mul* attribute.

Args

x: float or PyoObject New inversed *mul* attribute.

reset()
Resets the scanner.

setFunction(*x*)
Replace the *function* attribute.

Args

x: Python function new *function* attribute.

setToprint(*x*)
Replace the *toprint* attribute.

Args

x: int new *toprint* attribute.

function
Python function. Function to be called.

toprint
boolean. If True, prints values to the console.

CtlScan2

class CtlScan2(*function*, *toprint=True*)
Scan the Midi channel and controller number in input.

Scan the Midi channel and controller number in input and send them to a standard python *function*. Useful to implement a MidiLearn algorithm.

Parent `PyoObject`

Args

function: Python function (can't be a list) Function to be called. The function must be declared with two arguments, one for the controller number and one for the midi channel. Ex.:

def ctl_scan(ctlnum, midichnl): print(ctlnum, midichnl)

toprint: boolean, optional If True, controller number and value will be printed to the console.

Note: The out() method is bypassed. CtlScan2's signal can not be sent to audio outs.

```
>>> s = Server()
>>> s.setMidiInputDevice(0) # enter your device number (see pm_list_devices())
>>> s.boot()
>>> s.start()
>>> def ctl_scan(ctlnum, midichnl):
...     print(ctlnum, midichnl)
>>> a = CtlScan2(ctl_scan)
```

out(*chnl=0*, *inc=1*, *dur=0*, *delay=0*)

Start processing and send samples to audio output beginning at *chnl*.

This method returns *self*, allowing it to be applied at the object creation.

Args

chnl: int, optional Physical output assigned to the first audio stream of the object. Defaults to 0.

inc: int, optional Output channel increment value. Defaults to 1.

dur: float, optional Duration, in seconds, of the object's activation. The default is 0 and means infinite duration.

delay: float, optional Delay, in seconds, before the object's activation. Defaults to 0.

If *chnl* >= 0, successive streams increment the output number by *inc* and wrap around the global number of channels.

If *chnl* is negative, streams begin at 0, increment the output number by *inc* and wrap around the global number of channels. Then, the list of streams is scrambled.

If *chnl* is a list, successive values in the list will be assigned to successive streams.

setMul(*x*)

Replace the *mul* attribute.

Args

x: float or PyoObject New *mul* attribute.

setAdd(*x*)

Replace the *add* attribute.

Args

x: float or PyoObject New *add* attribute.

setSub(*x*)

Replace and inverse the *add* attribute.

Args

x: float or PyoObject New inversed *add* attribute.

setDiv(*x*)
Replace and inverse the *mul* attribute.

Args

x: float or PyoObject New inversed *mul* attribute.

reset()
Resets the scanner.

setFunction(*x*)
Replace the *function* attribute.

Args

x: Python function new *function* attribute.

setToprint(*x*)
Replace the *toprint* attribute.

Args

x: int new *toprint* attribute.

function
Python function. Function to be called.

toprint
boolean. If True, prints values to the console.

MidiAdsr

class MidiAdsr(*input*, *attack=0.01*, *decay=0.05*, *sustain=0.7*, *release=0.1*, *mul=1*, *add=0*)
Midi triggered ADSR envelope generator.

Calculates the classical ADSR envelope using linear segments. The envelope starts when it receives a positive value in input, this value is used as the peak amplitude of the envelope. The *sustain* parameter is a fraction of the peak value and sets the real sustain value. A 0 in input (note off) starts the release part of the envelope.

Parent *PyoObject*

Args

input: PyoObject Input signal used to trigger the envelope. A positive value sets the peak amplitude and starts the envelope. A 0 starts the release part of the envelope.

attack: float, optional Duration of the attack phase in seconds. Defaults to 0.01.

decay: float, optional Duration of the decay phase in seconds. Defaults to 0.05.

sustain: float, optional Amplitude of the sustain phase, as a fraction of the peak amplitude at the start of the envelope. Defaults to 0.7.

release: float, optional Duration of the release phase in seconds. Defaults to 0.1.

Note: The out() method is bypassed. MidiAdsr's signal can not be sent to audio outs.

As of version 0.8.0, exponential or logarithmic envelopes can be created with the exponent factor (see setExp() method).

```
>>> s = Server().boot()
>>> s.start()
>>> mid = Notein(scale=1)
>>> env = MidiAdsr(mid['velocity'], attack=.005, decay=.1, sustain=.4, release=1)
>>> a = SineLoop(freq=mid['pitch'], feedback=.1, mul=env).out()
>>> b = SineLoop(freq=mid['pitch']*1.005, feedback=.1, mul=env).out(1)
```

out (*chnl=0*, *inc=1*, *dur=0*, *delay=0*)

Start processing and send samples to audio output beginning at *chnl*.

This method returns *self*, allowing it to be applied at the object creation.

Args

chnl: int, optional Physical output assigned to the first audio stream of the object. Defaults to 0.

inc: int, optional Output channel increment value. Defaults to 1.

dur: float, optional Duration, in seconds, of the object's activation. The default is 0 and means infinite duration.

delay: float, optional Delay, in seconds, before the object's activation. Defaults to 0.

If *chnl* >= 0, successive streams increment the output number by *inc* and wrap around the global number of channels.

If *chnl* is negative, streams begin at 0, increment the output number by *inc* and wrap around the global number of channels. Then, the list of streams is scrambled.

If *chnl* is a list, successive values in the list will be assigned to successive streams.

setInput (*x*, *fadetime=0.05*)

Replace the *input* attribute.

Args

x: PyoObject New signal used to trigger the envelope.

fadetime: float, optional Crossfade time between old and new input. Defaults to 0.05.

setAttack (*x*)

Replace the *attack* attribute.

Args

x: float new *attack* attribute.

setDecay (*x*)

Replace the *decay* attribute.

Args

x: float new *decay* attribute.

setSustain (*x*)

Replace the *sustain* attribute.

Args

x: float new *sustain* attribute.

setRelease (*x*)

Replace the *sustain* attribute.

Args

x: float new *sustain* attribute.

setExp(*x*)
Sets an exponent factor to create exponential or logarithmic envelopes.

The default value is 1.0, which means linear segments. A value higher than 1.0 will produce exponential segments while a value between 0 and 1 will produce logarithmic segments. Must be > 0.0.

Args

x: float new *exp* attribute.

ctrl(*map_list=None*, *title=None*, *wxnoserver=False*)
Opens a sliders window to control the parameters of the object. Only parameters that can be set to a PyoObject are allowed to be mapped on a slider.

If a list of values are given to a parameter, a multisliders will be used to control each stream independently.

Args

map_list: list of SLMap objects, optional Users defined set of parameters scaling. There is default scaling for each object that accept *ctrl* method.

title: string, optional Title of the window. If none is provided, the name of the class is used.

wxnoserver: boolean, optional With wxPython graphical toolkit, if True, tells the interpreter that there will be no server window.

If *wxnoserver* is set to True, the interpreter will not wait for the server GUI before showing the controller window.

attack
float. Duration of the attack phase in seconds.

decay
float. Duration of the decay phase in seconds.

sustain
float. Amplitude of the sustain phase, as fraction of the peak amplitude.

release
float. Duration of the release phase in seconds.

exp
float. Exponent factor of the envelope.

MidiDelAdsr

class MidiDelAdsr(*input*, *delay=0*, *attack=0.01*, *decay=0.05*, *sustain=0.7*, *release=0.1*, *mul=1*, *add=0*)
Midi triggered ADSR envelope generator with pre-delay.

Calculates the classical ADSR envelope using linear segments. The envelope starts after *delay* seconds when it receives a positive value in input, this value is used as the peak amplitude of the envelope. The *sustain* parameter is a fraction of the peak value and sets the real sustain value. A 0 in input (note off) starts the release part of the envelope.

Parent *PyoObject*

Args

input: PyoObject Input signal used to trigger the envelope. A positive value sets the peak amplitude and starts the envelope. A 0 starts the release part of the envelope.

delay: float, optional Duration of the delay phase, before calling the envelope in seconds. Defaults to 0.

attack: float, optional Duration of the attack phase in seconds. Defaults to 0.01.

decay: float, optional Duration of the decay phase in seconds. Defaults to 0.05.

sustain: float, optional Amplitude of the sustain phase, as a fraction of the peak amplitude at the start of the envelope. Defaults to 0.7.

release: float, optional Duration of the release phase in seconds. Defaults to 0.1.

Note: The out() method is bypassed. MidiDelAdsr's signal can not be sent to audio outs.

As of version 0.8.0, exponential or logarithmic envelopes can be created with the exponent factor (see setExp() method).

```
>>> s = Server().boot()
>>> s.start()
>>> mid = Notein(scale=1)
>>> env = MidiDelAdsr(mid['velocity'], delay=.25, attack=.005, decay=.1, sustain=.
↪4, release=1)
>>> a = SineLoop(freq=mid['pitch'], feedback=.1, mul=env).out()
>>> b = SineLoop(freq=mid['pitch']*1.005, feedback=.1, mul=env).out(1)
```

out (*chnl=0*, *inc=1*, *dur=0*, *delay=0*)

Start processing and send samples to audio output beginning at *chnl*.

This method returns *self*, allowing it to be applied at the object creation.

Args

chnl: int, optional Physical output assigned to the first audio stream of the object. Defaults to 0.

inc: int, optional Output channel increment value. Defaults to 1.

dur: float, optional Duration, in seconds, of the object's activation. The default is 0 and means infinite duration.

delay: float, optional Delay, in seconds, before the object's activation. Defaults to 0.

If *chnl* >= 0, successive streams increment the output number by *inc* and wrap around the global number of channels.

If *chnl* is negative, streams begin at 0, increment the output number by *inc* and wrap around the global number of channels. Then, the list of streams is scrambled.

If *chnl* is a list, successive values in the list will be assigned to successive streams.

setInput (*x*, *fadetime=0.05*)

Replace the *input* attribute.

Args

x: PyoObject New signal used to trigger the envelope.

fadetime: float, optional Crossfade time between old and new input. Defaults to 0.05.

setDelay (*x*)

Replace the *delay* attribute.

Args

x: float new *delay* attribute.

setAttack(*x*)
Replace the *attack* attribute.

Args

x: float new *attack* attribute.

setDecay(*x*)
Replace the *decay* attribute.

Args

x: float new *decay* attribute.

setSustain(*x*)
Replace the *sustain* attribute.

Args

x: float new *sustain* attribute.

setRelease(*x*)
Replace the *sustain* attribute.

Args

x: float new *sustain* attribute.

setExp(*x*)
Sets an exponent factor to create exponential or logarithmic envelope.

The default value is 1.0, which means linear segments. A value higher than 1.0 will produce exponential segments while a value between 0 and 1 will produce logarithmic segments. Must be > 0.0.

Args

x: float new *exp* attribute.

ctrl(*map_list=None*, *title=None*, *wxnoserver=False*)
Opens a sliders window to control the parameters of the object. Only parameters that can be set to a PyoObject are allowed to be mapped on a slider.

If a list of values are given to a parameter, a multisliders will be used to control each stream independently.

Args

map_list: list of SLMap objects, optional Users defined set of parameters scaling. There is default scaling for each object that accept *ctrl* method.

title: string, optional Title of the window. If none is provided, the name of the class is used.

wxnoserver: boolean, optional With wxPython graphical toolkit, if True, tells the interpreter that there will be no server window.

If *wxnoserver* is set to True, the interpreter will not wait for the server GUI before showing the controller window.

delay
float. Duration of the delay phase in seconds.

attack
float. Duration of the attack phase in seconds.

decay
float. Duration of the decay phase in seconds.

sustain
float. Amplitude of the sustain phase, as fraction of the peak amplitude.

release
float. Duration of the release phase in seconds.

exp
float. Exponent factor of the envelope.

Midictl

class Midictl(*ctlnumber*, *minscale=0*, *maxscale=1*, *init=0*, *channel=0*, *mul=1*, *add=0*)
Get the current value of a Midi controller.

Get the current value of a controller and optionally map it inside a specified range.

Parent *PyoObject*

Args

ctlnumber: int Controller number.

minscale: float, optional Low range value for mapping. Defaults to 0.

maxscale: float, optional High range value for mapping. Defaults to 1.

init: float, optional Initial value. Defaults to 0.

channel: int, optional Midi channel. 0 means all channels. Defaults to 0.

Note: The out() method is bypassed. Midictl's signal can not be sent to audio outs.

```
>>> s = Server().boot()
>>> s.start()
>>> m = Midictl(ctlnumber=[107,102], minscale=250, maxscale=1000)
>>> p = Port(m, .02)
>>> a = Sine(freq=p, mul=.3).out()
>>> a1 = Sine(freq=p*1.25, mul=.3).out()
>>> a2 = Sine(freq=p*1.5, mul=.3).out()
```

out(*chnl=0*, *inc=1*, *dur=0*, *delay=0*)
Start processing and send samples to audio output beginning at *chnl*.

This method returns *self*, allowing it to be applied at the object creation.

Args

chnl: int, optional Physical output assigned to the first audio stream of the object. Defaults to 0.

inc: int, optional Output channel increment value. Defaults to 1.

dur: float, optional Duration, in seconds, of the object's activation. The default is 0 and means infinite duration.

delay: float, optional Delay, in seconds, before the object's activation. Defaults to 0.

If *chnl* >= 0, successive streams increment the output number by *inc* and wrap around the global number of channels.

If *chnl* is negative, streams begin at 0, increment the output number by *inc* and wrap around the global number of channels. Then, the list of streams is scrambled.

If *chnl* is a list, successive values in the list will be assigned to successive streams.

setCtlNumber(*x*)
Replace the *ctlnumber* attribute.

Args

x: int new *ctlnumber* attribute.

setMinScale(*x*)
Replace the *minscale* attribute.

Args

x: float new *minscale* attribute.

setMaxScale(*x*)
Replace the *maxscale* attribute.

Args

x: float new *maxscale* attribute.

setChannel(*x*)
Replace the *channel* attribute.

Args

x: int new *channel* attribute.

setValue(*x*)
Reset audio stream to value in argument.

Args

x: float new current value.

setInterpolation(*x*)
Deprecated method. If needed, use Port or SigTo to interpolate between values.

ctrl(*map_list=None*, *title=None*, *wxnoserver=False*)
Opens a sliders window to control the parameters of the object. Only parameters that can be set to a PyoObject are allowed to be mapped on a slider.

If a list of values are given to a parameter, a multisliders will be used to control each stream independently.

Args

map_list: list of SLMap objects, optional Users defined set of parameters scaling. There is default scaling for each object that accept *ctrl* method.

title: string, optional Title of the window. If none is provided, the name of the class is used.

wxnoserver: boolean, optional With wxPython graphical toolkit, if True, tells the interpreter that there will be no server window.

If *wxnoserver* is set to True, the interpreter will not wait for the server GUI before showing the controller window.

ctlnumber
int. Controller number.

minscale
float. Minimum value for scaling.

maxscale
float. Maximum value for scaling.

channel
int. Midi channel. 0 means all channels.

Notein

class Notein(*poly=10*, *scale=0*, *first=0*, *last=127*, *channel=0*, *mul=1*, *add=0*)
Generates Midi note messages.

From a Midi device, takes the notes in the range defined with *first* and *last* parameters, and outputs up to *poly* noteon - noteoff streams in the *scale* format (Midi, hertz or transpo).

Parent `PyoObject`

Args

poly: int, optional Number of streams of polyphony generated. Defaults to 10.

scale: int, optional

Pitch output format.

0. Midi
1. Hertz
2. transpo

In the transpo mode, the default central key (the key where there is no transposition) is (*first* + *last*) / 2.

The central key can be changed with the setCentralKey method.

first: int, optional Lowest Midi value. Defaults to 0.

last: int, optional Highest Midi value. Defaults to 127.

channel: int, optional Midi channel. 0 means all channels. Defaults to 0.

Note: Pitch and velocity are two separated set of streams. The user should call :

Notein['pitch'] to retrieve pitch streams.
Notein['velocity'] to retrieve velocity streams.

Velocity is automatically scaled between 0 and 1.

Notein also outputs trigger streams on noteon and noteoff. These streams can be retrieved with :

Notein['trigon'] to retrieve noteon trigger streams.
Notein['trigoff'] to retrieve noteoff trigger streams.

The out() method is bypassed. Notein's signal can not be sent to audio outs.

```
>>> s = Server().boot()
>>> s.start()
>>> notes = Notein(poly=10, scale=1, mul=.5)
>>> p = Port(notes['velocity'], .001, .5)
>>> b = Sine(freq=notes['pitch'], mul=p).out()
>>> c = Sine(freq=notes['pitch'] * 0.997, mul=p).out()
>>> d = Sine(freq=notes['pitch'] * 1.005, mul=p).out()
```

setScale(*x*)
Replace the *scale* attribute.

Args

x: int new *scale* attribute. 0 = midi, 1 = hertz, 2 = transpo.

setFirst(*x*)
Replace the *first* attribute.

Args

x: int new *first* attribute, between 0 and 127.

setLast(*x*)
Replace the *last* attribute.

Args

x: int new *last* attribute, between 0 and 127.

setChannel(*x*)
Replace the *channel* attribute.

Args

x: int new *channel* attribute.

setCentralKey(*x*)
Set the midi key where there is no transposition.

Used for transpo conversion. This value must be greater than or equal to *first* and lower than or equal to *last*.

Args

x: int new centralkey value.

setStealing(*x*)
Activates the stealing mode if True. Defaults to False.

In stealing mode, a new note will overwrite the oldest one according to the polyphony. In non-stealing mode, if the polyphony is already full, the new notes will be ignored.

Args

x: boolean True for stealing mode, False for non-stealing.

get(*identifier='pitch'*, *all=False*)
Return the first sample of the current buffer as a float.

Can be used to convert audio stream to usable Python data.

"pitch" or "velocity" must be given to *identifier* to specify which stream to get value from.

Args

identifier: string {"pitch", "velocity"} Address string parameter identifying audio stream. Defaults to "pitch".

all: boolean, optional If True, the first value of each object's stream will be returned as a list.

Otherwise, only the value of the first object's stream will be returned as a float.

play(*dur=0*, *delay=0*)

Start processing without sending samples to output. This method is called automatically at the object creation.

This method returns *self*, allowing it to be applied at the object creation.

Args

dur: float, optional Duration, in seconds, of the object's activation. The default is 0 and means infinite duration.

delay: float, optional Delay, in seconds, before the object's activation. Defaults to 0.

out(*chnl=0*, *inc=1*, *dur=0*, *delay=0*)

Start processing and send samples to audio output beginning at *chnl*.

This method returns *self*, allowing it to be applied at the object creation.

Args

chnl: int, optional Physical output assigned to the first audio stream of the object. Defaults to 0.

inc: int, optional Output channel increment value. Defaults to 1.

dur: float, optional Duration, in seconds, of the object's activation. The default is 0 and means infinite duration.

delay: float, optional Delay, in seconds, before the object's activation. Defaults to 0.

If *chnl* >= 0, successive streams increment the output number by *inc* and wrap around the global number of channels.

If *chnl* is negative, streams begin at 0, increment the output number by *inc* and wrap around the global number of channels. Then, the list of streams is scrambled.

If *chnl* is a list, successive values in the list will be assigned to successive streams.

stop(*wait=0*)

Stop processing.

This method returns *self*, allowing it to be applied at the object creation.

Args

wait: float, optional Delay, in seconds, before the process is actually stopped. If autoStartChildren is activated in the Server, this value is propagated to the children objects. Defaults to 0.

Note: if the method setStopDelay(x) was called before calling stop(wait) with a positive *wait* value, the *wait* value won't overwrite the value given to setStopDelay for the current object, but will be the one propagated to children objects. This allow to set a waiting time for a specific object with setStopDelay whithout changing the global delay time given to the stop method.

Fader and Adsr objects ignore waiting time given to the stop method because they already implement a delayed processing triggered by the stop call.

---

ctrl(*map_list=None*, *title=None*, *wxnoserver=False*)
Opens a sliders window to control the parameters of the object. Only parameters that can be set to a PyoObject are allowed to be mapped on a slider.

If a list of values are given to a parameter, a multisliders will be used to control each stream independently.

Args

map_list: list of SLMap objects, optional Users defined set of parameters scaling. There is default scaling for each object that accept *ctrl* method.

title: string, optional Title of the window. If none is provided, the name of the class is used.

wxnoserver: boolean, optional With wxPython graphical toolkit, if True, tells the interpreter that there will be no server window.

If *wxnoserver* is set to True, the interpreter will not wait for the server GUI before showing the controller window.

keyboard(*title='Notein keyboard'*, *wxnoserver=False*)
Opens a virtual midi keyboard for this object.

Args

title: string, optional Title of the window. If none is provided, the name of the class is used.

wxnoserver: boolean, optional With wxPython graphical toolkit, if True, tells the interpreter that there will be no server window.

If *wxnoserver* is set to True, the interpreter will not wait for the server GUI before showing the controller window.

scale
int. Output format. 0 = midi, 1 = hertz, 2 = transpo.

first
int. Lowest midi value.

last
int. Highest midi value.

channel
int. Midi channel. 0 means all channels.

Programin

class Programin(*channel=0*, *mul=1*, *add=0*)
Get the current value of a program change Midi controller.

Get the current value of a program change Midi controller.

Parent `PyoObject`

Args

channel: int, optional Midi channel. 0 means all channels. Defaults to 0.

Note: The out() method is bypassed. Programin's signal can not be sent to audio outs.

```
>>> s = Server().boot()
>>> s.start()
>>> notes = Notein(poly=10, scale=1, mul=.5)
>>> pchg = Programin(mul=1./12, add=1)
>>> p = Port(notes['velocity'], .001, .5)
>>> b = Sine(freq=notes['pitch'] * pchg, mul=p).out()
>>> c = Sine(freq=notes['pitch'] * pchg * 0.997, mul=p).out()
>>> d = Sine(freq=notes['pitch'] * pchg * 1.005, mul=p).out()
```

out (*chnl=0*, *inc=1*, *dur=0*, *delay=0*)

Start processing and send samples to audio output beginning at *chnl*.

This method returns *self*, allowing it to be applied at the object creation.

Args

chnl: int, optional Physical output assigned to the first audio stream of the object. Defaults to 0.

inc: int, optional Output channel increment value. Defaults to 1.

dur: float, optional Duration, in seconds, of the object's activation. The default is 0 and means infinite duration.

delay: float, optional Delay, in seconds, before the object's activation. Defaults to 0.

If *chnl* >= 0, successive streams increment the output number by *inc* and wrap around the global number of channels.

If *chnl* is negative, streams begin at 0, increment the output number by *inc* and wrap around the global number of channels. Then, the list of streams is scrambled.

If *chnl* is a list, successive values in the list will be assigned to successive streams.

setChannel (*x*)

Replace the *channel* attribute.

Args

x: int new *channel* attribute.

ctrl (*map_list=None*, *title=None*, *wxnoserver=False*)

Opens a sliders window to control the parameters of the object. Only parameters that can be set to a PyoObject are allowed to be mapped on a slider.

If a list of values are given to a parameter, a multisliders will be used to control each stream independently.

Args

map_list: list of SLMap objects, optional Users defined set of parameters scaling. There is default scaling for each object that accept *ctrl* method.

title: string, optional Title of the window. If none is provided, the name of the class is used.

wxnoserver: boolean, optional With wxPython graphical toolkit, if True, tells the interpreter that there will be no server window.

If *wxnoserver* is set to True, the interpreter will not wait for the server GUI before showing the controller window.

channel
int. Midi channel. 0 means all channels.

Touchin

class Touchin(*minscale=0*, *maxscale=1*, *init=0*, *channel=0*, *mul=1*, *add=0*)
Get the current value of an after-touch Midi controller.

Get the current value of an after-touch Midi controller and optionally maps it inside a specified range.

Parent *PyoObject*

Args

minscale: float, optional Low range value for mapping. Defaults to 0.

maxscale: float, optional High range value for mapping. Defaults to 1.

init: float, optional Initial value. Defaults to 0.

channel: int, optional Midi channel. 0 means all channels. Defaults to 0.

Note: The out() method is bypassed. Touchin's signal can not be sent to audio outs.

```
>>> s = Server().boot()
>>> s.start()
>>> notes = Notein(poly=10, scale=1, mul=.5)
>>> touch = Touchin(minscale=1, maxscale=2, init=1)
>>> p = Port(notes['velocity'], .001, .5)
>>> b = Sine(freq=notes['pitch'] * touch, mul=p).out()
>>> c = Sine(freq=notes['pitch'] * touch * 0.997, mul=p).out()
>>> d = Sine(freq=notes['pitch'] * touch * 1.005, mul=p).out()
```

out(*chnl=0*, *inc=1*, *dur=0*, *delay=0*)
Start processing and send samples to audio output beginning at *chnl*.

This method returns *self*, allowing it to be applied at the object creation.

Args

chnl: int, optional Physical output assigned to the first audio stream of the object. Defaults to 0.

inc: int, optional Output channel increment value. Defaults to 1.

dur: float, optional Duration, in seconds, of the object's activation. The default is 0 and means infinite duration.

delay: float, optional Delay, in seconds, before the object's activation. Defaults to 0.

If *chnl* >= 0, successive streams increment the output number by *inc* and wrap around the global number of channels.

If *chnl* is negative, streams begin at 0, increment the output number by *inc* and wrap around the global number of channels. Then, the list of streams is scrambled.

If *chnl* is a list, successive values in the list will be assigned to successive streams.

setMinScale(*x*)
Replace the *minscale* attribute.

Args

x: float new *minscale* attribute.

setMaxScale(*x*)
Replace the *maxscale* attribute.

Args

x: float new *maxscale* attribute.

setChannel(*x*)
Replace the *channel* attribute.

Args

x: int new *channel* attribute.

setInterpolation(*x*)
Deprecated method. If needed, use Port or SigTo to interpolate between values.

ctrl(*map_list=None*, *title=None*, *wxnoserver=False*)
Opens a sliders window to control the parameters of the object. Only parameters that can be set to a PyoObject are allowed to be mapped on a slider.

If a list of values are given to a parameter, a multisliders will be used to control each stream independently.

Args

map_list: list of SLMap objects, optional Users defined set of parameters scaling. There is default scaling for each object that accept *ctrl* method.

title: string, optional Title of the window. If none is provided, the name of the class is used.

wxnoserver: boolean, optional With wxPython graphical toolkit, if True, tells the interpreter that there will be no server window.

If *wxnoserver* is set to True, the interpreter will not wait for the server GUI before showing the controller window.

minscale
float. Minimum value for scaling.

maxscale
float. Maximum value for scaling.

channel
int. Midi channel. 0 means all channels.

RawMidi

class RawMidi(*function*)
Raw Midi handler.

This object calls a python function for each raw midi data (status, data1, data2) event for further processing in Python.

Parent `PyoObject`

Args

function: Python function (can't be a list) Function to be called. The function must be declared with three arguments, one for the status byte and two for the data bytes. Ex.:

def event(status, data1, data2): print(status, data1, data2)

Note: The out() method is bypassed. RawMidi's signal can not be sent to audio outs.

```
>>> s = Server()
>>> s.setMidiInputDevice(99) # opens all devices
>>> s.boot()
>>> s.start()
>>> def event(status, data1, data2):
...     print(status, data1, data2)
>>> a = RawMidi(event)
```

out (*chnl=0*, *inc=1*, *dur=0*, *delay=0*)

Start processing and send samples to audio output beginning at *chnl*.

This method returns *self*, allowing it to be applied at the object creation.

Args

chnl: int, optional Physical output assigned to the first audio stream of the object. Defaults to 0.

inc: int, optional Output channel increment value. Defaults to 1.

dur: float, optional Duration, in seconds, of the object's activation. The default is 0 and means infinite duration.

delay: float, optional Delay, in seconds, before the object's activation. Defaults to 0.

If *chnl* >= 0, successive streams increment the output number by *inc* and wrap around the global number of channels.

If *chnl* is negative, streams begin at 0, increment the output number by *inc* and wrap around the global number of channels. Then, the list of streams is scrambled.

If *chnl* is a list, successive values in the list will be assigned to successive streams.

setMul (*x*)

Replace the *mul* attribute.

Args

x: float or PyoObject New *mul* attribute.

setAdd (*x*)

Replace the *add* attribute.

Args

x: float or PyoObject New *add* attribute.

setSub (*x*)

Replace and inverse the *add* attribute.

Args

x: float or PyoObject New inversed *add* attribute.

setDiv (*x*)

Replace and inverse the *mul* attribute.

Args

x: float or PyoObject New inversed *mul* attribute.

setFunction(*x*)
Replace the *function* attribute.

Args

x: Python function new *function* attribute.

function
Python function. Function to be called.

MidiLinseg

class MidiLinseg(*input*, *list*, *hold=1*, *mul=1*, *add=0*)
Line segments trigger.

TrigLinseg starts reading a break-points line segments each time it receives a trigger in its *input* parameter.

Parent *PyoObject*

Args

input: PyoObject Input signal used to trigger the envelope. A positive value sets the peak amplitude and starts the envelope. A 0 starts the release part of the envelope.

list: list of tuples Points used to construct the line segments. Each tuple is a new point in the form (time, value).

Times are given in seconds and must be in increasing order.

hold: int, optional The point, starting at 0, acting as the sustain point. The envelope will hold this value as long as the input signal is positive. The release part is the remaining points. Defaults to 1.

Note: MidiLinseg will send a trigger signal at the end of the playback. User can retreive the trigger streams by calling obj['trig']. Useful to synchronize other processes.

The out() method is bypassed. MidiLinseg's signal can not be sent to audio outs.

```
>>> s = Server().boot()
>>> s.start()
>>> mid = Notein(scale=1)
>>> env = [(0,0), (0.1,1), (0.2,0.5), (0.4,0.7), (0.5,0.3), (1,1), (2,0)]
>>> env = MidiLinseg(mid['velocity'], env, hold=4)
>>> a = SineLoop(freq=mid['pitch'], feedback=.1, mul=env).out()
>>> b = SineLoop(freq=mid['pitch']*1.005, feedback=.1, mul=env).out(1)
```

out(*chnl=0*, *inc=1*, *dur=0*, *delay=0*)
Start processing and send samples to audio output beginning at *chnl*.

This method returns *self*, allowing it to be applied at the object creation.

Args

chnl: int, optional Physical output assigned to the first audio stream of the object. Defaults to 0.

inc: int, optional Output channel increment value. Defaults to 1.

dur: float, optional Duration, in seconds, of the object's activation. The default is 0 and means infinite duration.

delay: float, optional Delay, in seconds, before the object's activation. Defaults to 0.

If *chnl* >= 0, successive streams increment the output number by *inc* and wrap around the global number of channels.

If *chnl* is negative, streams begin at 0, increment the output number by *inc* and wrap around the global number of channels. Then, the list of streams is scrambled.

If *chnl* is a list, successive values in the list will be assigned to successive streams.

setInput(*x*, *fadetime=0.05*)
Replace the *input* attribute.

Args

x: PyoObject New signal to process.

fadetime: float, optional Crossfade time between old and new input. Defaults to 0.05.

setList(*x*)
Replace the *list* attribute.

Args

x: list of tuples new *list* attribute.

replace(*x*)
Alias for *setList* method.

Args

x: list of tuples new *list* attribute.

setHold(*x*)
Replace the *hold* attribute.

Args

x: int new *hold* attribute.

ctrl(*map_list=None*, *title=None*, *wxnoserver=False*)
Opens a sliders window to control the parameters of the object. Only parameters that can be set to a PyoObject are allowed to be mapped on a slider.

If a list of values are given to a parameter, a multisliders will be used to control each stream independently.

Args

map_list: list of SLMap objects, optional Users defined set of parameters scaling. There is default scaling for each object that accept *ctrl* method.

title: string, optional Title of the window. If none is provided, the name of the class is used.

wxnoserver: boolean, optional With wxPython graphical toolkit, if True, tells the interpreter that there will be no server window.

If *wxnoserver* is set to True, the interpreter will not wait for the server GUI before showing the controller window.

graph(*xlen=None*, *yrange=None*, *title=None*, *wxnoserver=False*)
Opens a grapher window to control the shape of the envelope.

When editing the grapher with the mouse, the new set of points will be send to the object on mouse up.

Ctrl+C with focus on the grapher will copy the list of points to the clipboard, giving an easy way to insert the new shape in a script.

Args

xlen: float, optional Set the maximum value of the X axis of the graph. If None, the maximum value is retrieve from the current list of points. Defaults to None.

yrange: tuple, optional Set the min and max values of the Y axis of the graph. If None, min and max are retrieve from the current list of points. Defaults to None.

title: string, optional Title of the window. If none is provided, the name of the class is used.

wxnoserver: boolean, optional With wxPython graphical toolkit, if True, tells the interpreter that there will be no server window.

If *wxnoserver* is set to True, the interpreter will not wait for the server GUI before showing the controller window.

input
PyoObject. Audio trigger signal.

list
list of tuples. Points used to construct the line segments.

hold
int. The sustain point.

Open Sound Control

Objects to manage values on an Open Sound Control port.

OscSend takes the first value of each buffersize and send it on an OSC port.

OscReceive creates and returns audio streams from the value in its input port.

The audio streams of these objects are essentially intended to be controls and can't be sent to the output soundcard.

Note: These objects are available only if pyo is built with OSC (Open Sound Control) support.

OscDataReceive

class OscDataReceive(*port*, *address*, *function*)
Receives data values over a network via the Open Sound Control protocol.

Uses the OSC protocol to receive data values from other softwares or other computers. When a message is received, the function given at the argument *function* is called with the current address destination in argument followed by a tuple of values.

Parent *PyoObject*

Args

port: int Port on which values are received. Sender should output on the same port. Unlike OscDataSend object, there can be only one port per OscDataReceive object. Available at initialization time only.

address: string Address used on the port to identify values. Address is in the form of a Unix path (ex.: "/pitch"). There can be as many addresses as needed on a single port.

function: callable (can't be a list) This function will be called whenever a message with a known address is received. there can be only one function per OscDataReceive object. Available at initialization time only.

Note: The header of the callable given at *function* argument must be in this form :

def my_func(address, *args): ...

The out() method is bypassed. OscDataReceive has no audio signal.

OscDataReceive has no *mul* and *add* attributes.

```
>>> s = Server().boot()
>>> s.start()
>>> def pp(address, *args):
...     print(address)
...     print(args)
>>> r = OscDataReceive(9900, "/data/test", pp)
>>> # Send various types
>>> a = OscDataSend("fissif", 9900, "/data/test")
>>> msg = [3.14159, 1, "Hello", "world!", 2, 6.18]
>>> a.send(msg)
>>> # Send a blob
>>> b = OscDataSend("b", 9900, "/data/test")
>>> msg = [[chr(i) for i in range(10)]]
>>> b.send(msg)
>>> # Send a MIDI noteon on port 0
>>> c = OscDataSend("m", 9900, "/data/test")
>>> msg = [[0, 144, 60, 100]]
>>> c.send(msg)
```

setMul(*x*)

Replace the *mul* attribute.

Args

x: float or PyoObject New *mul* attribute.

setAdd(*x*)

Replace the *add* attribute.

Args

x: float or PyoObject New *add* attribute.

out(*chnl=0*, *inc=1*, *dur=0*, *delay=0*)

Start processing and send samples to audio output beginning at *chnl*.

This method returns *self*, allowing it to be applied at the object creation.

Args

chnl: int, optional Physical output assigned to the first audio stream of the object. Defaults to 0.

inc: int, optional Output channel increment value. Defaults to 1.

dur: float, optional Duration, in seconds, of the object's activation. The default is 0 and means infinite duration.

delay: float, optional Delay, in seconds, before the object's activation. Defaults to 0.

If *chnl* >= 0, successive streams increment the output number by *inc* and wrap around the global number of channels.

If *chnl* is negative, streams begin at 0, increment the output number by *inc* and wrap around the global number of channels. Then, the list of streams is scrambled.

If *chnl* is a list, successive values in the list will be assigned to successive streams.

getAddresses()
Returns the addresses managed by the object.

addAddress(*path*)
Adds new address(es) to the object's handler.

Args

path: string or list of strings New path(s) to receive from.

delAddress(*path*)
Removes address(es) from the object's handler.

Args

path: string or list of strings Path(s) to remove.

OscDataSend

class OscDataSend(*types*, *port*, *address*, *host='127.0.0.1'*)
Sends data values over a network via the Open Sound Control protocol.

Uses the OSC protocol to share values to other softwares or other computers. Values are sent on the form of a list containing *types* elements.

Parent `PyoObject`

Args

types: str String specifying the types sequence of the message to be sent. Possible values are:

- "i": integer
- "h": long integer
- "f": float
- "d": double
- "s" ; string
- "b": blob (list of chars)
- "m": MIDI packet (list of 4 bytes: [midi port, status, data1, data2])
- "c": char
- "T": True
- "F": False
- "N": None (nil)

The string "ssfi" indicates that the value to send will be a list containing two strings followed by a float and an integer.

port: int Port on which values are sent. Receiver should listen on the same port.

address: string Address used on the port to identify values. Address is in the form of a Unix path (ex.: '/pitch').

host: string, optional IP address of the target computer. The default, '127.0.0.1', is the localhost.

Note: The out() method is bypassed. OscDataSend has no audio signal.

OscDataSend has no *mul* and *add* attributes.

```
>>> s = Server().boot()
>>> s.start()
>>> def pp(address, *args):
...     print(address)
...     print(args)
>>> r = OscDataReceive(9900, "/data/test", pp)
>>> # Send various types
>>> a = OscDataSend("fissif", 9900, "/data/test")
>>> msg = [3.14159, 1, "Hello", "world!", 2, 6.18]
>>> a.send(msg)
>>> # Send a blob
>>> b = OscDataSend("b", 9900, "/data/test")
>>> msg = [[chr(i) for i in range(10)]]
>>> b.send(msg)
>>> # Send a MIDI noteon on port 0
>>> c = OscDataSend("m", 9900, "/data/test")
>>> msg = [[0, 144, 60, 100]]
>>> c.send(msg)
```

out (*chnl=0*, *inc=1*, *dur=0*, *delay=0*)

Start processing and send samples to audio output beginning at *chnl*.

This method returns *self*, allowing it to be applied at the object creation.

Args

chnl: int, optional Physical output assigned to the first audio stream of the object. Defaults to 0.

inc: int, optional Output channel increment value. Defaults to 1.

dur: float, optional Duration, in seconds, of the object's activation. The default is 0 and means infinite duration.

delay: float, optional Delay, in seconds, before the object's activation. Defaults to 0.

If *chnl* >= 0, successive streams increment the output number by *inc* and wrap around the global number of channels.

If *chnl* is negative, streams begin at 0, increment the output number by *inc* and wrap around the global number of channels. Then, the list of streams is scrambled.

If *chnl* is a list, successive values in the list will be assigned to successive streams.

setMul (*x*)

Replace the *mul* attribute.

Args

x: float or PyoObject New *mul* attribute.

setAdd(*x*)
Replace the *add* attribute.

Args

x: float or PyoObject New *add* attribute.

getAddresses()
Returns the addresses managed by the object.

addAddress(*types*, *port*, *address*, *host='127.0.0.1'*)
Adds new address(es) to the object's handler.

Args

types: str String specifying the types sequence of the message to be sent. Possible values are: - "i": integer - "h": long integer - "f": float - "d": double - "s" ; string - "b": blob (list of chars) - "m": MIDI packet (list of 4 bytes: [midi port, status, data1, data2]) - "c": char - "T": True - "F": False - "N": None (nil)

The string "ssfi" indicates that the value to send will be a list containing two strings followed by a float and an integer.

port: int Port on which values are sent. Receiver should listen on the same port.

address: string Address used on the port to identify values. Address is in the form of a Unix path (ex.: '/pitch').

host: string, optional IP address of the target computer. The default, '127.0.0.1', is the localhost.

delAddress(*path*)
Removes address(es) from the object's handler.

Args

path: string or list of strings Path(s) to remove.

send(*msg*, *address=None*)
Method used to send *msg* values as a list.

Args

msg: list List of values to send. Types of values in list must be of the kind defined of *types* argument given at the object's initialization.

address: string, optional Address destination to send values. If None, values will be sent to all addresses managed by the object.

OscListReceive

class OscListReceive(*port*, *address*, *num=8*, *mul=1*, *add=0*)
Receives list of values over a network via the Open Sound Control protocol.

Uses the OSC protocol to receive list of floating-point values from other softwares or other computers. The list are converted into audio streams. Get values at the beginning of each buffersize and fill buffers with them.

Parent `PyoObject`

Args

port: int Port on which values are received. Sender should output on the same port. Unlike OscSend object, there can be only one port per OscListReceive object. Available at initialization time only.

address: string Address used on the port to identify values. Address is in the form of a Unix path (ex.: '/pitch').

num: int, optional Length of the lists in input. The object will generate *num* audio streams per given address. Available at initialization time only. This value can't be a list. That means all addresses managed by an OscListReceive object are of the same length. Defaults to 8.

Note: Audio streams are accessed with the *address* string parameter. The user should call :

OscReceive['/pitch'] to retreive list of streams named '/pitch'.

The out() method is bypassed. OscReceive's signal can not be sent to audio outs.

```
>>> s = Server().boot()
>>> s.start()
>>> # 8 oscillators
>>> a = OscListReceive(port=10001, address=['/pitch', '/amp'], num=8)
>>> b = Sine(freq=a['/pitch'], mul=a['/amp']).mix(2).out()
```

getAddresses()
Returns the addresses managed by the object.

addAddress(*path*, *mul=1*, *add=0*)
Adds new address(es) to the object's handler.

Args

path: string or list of strings New path(s) to receive from.

mul: float or PyoObject Multiplication factor. Defaults to 1.

add: float or PyoObject Addition factor. Defaults to 0.

delAddress(*path*)
Removes address(es) from the object's handler.

Args

path: string or list of strings Path(s) to remove.

setInterpolation(*x*)
Activate/Deactivate interpolation. Activated by default.

Args

x: boolean True activates the interpolation, False deactivates it.

setValue(*path*, *value*)
Sets value for a given address.

Args

path: string Address to which the value should be attributed.

value: list of floats List of values to attribute to the given address.

get(*identifier=None*, *all=False*)
Return the first list of samples of the current buffer as floats.

Can be used to convert audio stream to usable Python data.

Address as string must be given to *identifier* to specify which stream to get value from.

Args

identifier: string Address string parameter identifying audio stream. Defaults to None, useful when *all* is True to retreive all streams values.

all: boolean, optional If True, the first list of values of each object's stream will be returned as a list of lists. Otherwise, only the the list of the object's identifier will be returned as a list of floats. Defaults to False.

play(*dur=0*, *delay=0*)
Start processing without sending samples to output. This method is called automatically at the object creation.

This method returns *self*, allowing it to be applied at the object creation.

Args

dur: float, optional Duration, in seconds, of the object's activation. The default is 0 and means infinite duration.

delay: float, optional Delay, in seconds, before the object's activation. Defaults to 0.

out(*chnl=0*, *inc=1*, *dur=0*, *delay=0*)
Start processing and send samples to audio output beginning at *chnl*.

This method returns *self*, allowing it to be applied at the object creation.

Args

chnl: int, optional Physical output assigned to the first audio stream of the object. Defaults to 0.

inc: int, optional Output channel increment value. Defaults to 1.

dur: float, optional Duration, in seconds, of the object's activation. The default is 0 and means infinite duration.

delay: float, optional Delay, in seconds, before the object's activation. Defaults to 0.

If *chnl* >= 0, successive streams increment the output number by *inc* and wrap around the global number of channels.

If *chnl* is negative, streams begin at 0, increment the output number by *inc* and wrap around the global number of channels. Then, the list of streams is scrambled.

If *chnl* is a list, successive values in the list will be assigned to successive streams.

stop(*wait=0*)
Stop processing.

This method returns *self*, allowing it to be applied at the object creation.

Args

wait: float, optional Delay, in seconds, before the process is actually stopped. If autoStartChildren is activated in the Server, this value is propagated to the children objects. Defaults to 0.

Note: if the method setStopDelay(x) was called before calling stop(wait) with a positive *wait* value, the *wait* value won't overwrite the value given to setStopDelay for the current object, but will be the one propagated to children objects. This allow to set a waiting time for a specific object with setStopDelay whithout changing the global delay time given to the stop method.

Fader and Adsr objects ignore waiting time given to the stop method because they already implement a delayed processing triggered by the stop call.

OscReceive

class OscReceive(*port*, *address*, *mul=1*, *add=0*)

Receives values over a network via the Open Sound Control protocol.

Uses the OSC protocol to receive values from other softwares or other computers. Get a value at the beginning of each buffersize and fill its buffer with it.

Parent *PyoObject*

Args

port: int Port on which values are received. Sender should output on the same port.

Unlike OscSend object, there can be only one port per OscReceive object.

Available at initialization time only.

address: string Address used on the port to identify values. Address is in the form of a Unix path (ex.: '/pitch').

Note: Audio streams are accessed with the *address* string parameter. The user should call :

OscReceive['/pitch'] to retreive stream named '/pitch'.

The out() method is bypassed. OscReceive's signal can not be sent to audio outs.

```
>>> s = Server().boot()
>>> s.start()
>>> a = OscReceive(port=10001, address=['/pitch', '/amp'])
>>> b = Sine(freq=a['/pitch'], mul=a['/amp']).mix(2).out()
```

getAddresses()

Returns the addresses managed by the object.

addAddress(*path*, *mul=1*, *add=0*)

Adds new address(es) to the object's handler.

Args

path: string or list of strings New path(s) to receive from.

mul: float or PyoObject Multiplication factor. Defaults to 1.

add: float or PyoObject Addition factor. Defaults to 0.

delAddress(*path*)

Removes address(es) from the object's handler.

Args

path: string or list of strings Path(s) to remove.

setInterpolation(*x*)
Activate/Deactivate interpolation. Activated by default.

Args

x: boolean True activates the interpolation, False deactivates it.

setValue(*path*, *value*)
Sets value for a given address.

Args

path: string Address to which the value should be attributed.

value: float Value to attribute to the given address.

get(*identifier=None*, *all=False*)
Return the first sample of the current buffer as a float.

Can be used to convert audio stream to usable Python data.

Address as string must be given to *identifier* to specify which stream to get value from.

Args

identifier: string Address string parameter identifying audio stream. Defaults to None, useful when *all* is True to retreive all streams values.

all: boolean, optional If True, the first value of each object's stream will be returned as a list. Otherwise, only the value of the first object's stream will be returned as a float. Defaults to False.

play(*dur=0*, *delay=0*)
Start processing without sending samples to output. This method is called automatically at the object creation.

This method returns *self*, allowing it to be applied at the object creation.

Args

dur: float, optional Duration, in seconds, of the object's activation. The default is 0 and means infinite duration.

delay: float, optional Delay, in seconds, before the object's activation. Defaults to 0.

out(*chnl=0*, *inc=1*, *dur=0*, *delay=0*)
Start processing and send samples to audio output beginning at *chnl*.

This method returns *self*, allowing it to be applied at the object creation.

Args

chnl: int, optional Physical output assigned to the first audio stream of the object. Defaults to 0.

inc: int, optional Output channel increment value. Defaults to 1.

dur: float, optional Duration, in seconds, of the object's activation. The default is 0 and means infinite duration.

delay: float, optional Delay, in seconds, before the object's activation. Defaults to 0.

If *chnl* >= 0, successive streams increment the output number by *inc* and wrap around the global number of channels.

If *chnl* is negative, streams begin at 0, increment the output number by *inc* and wrap around the global number of channels. Then, the list of streams is scrambled.

If *chnl* is a list, successive values in the list will be assigned to successive streams.

stop(*wait=0*)
Stop processing.

This method returns *self*, allowing it to be applied at the object creation.

Args

wait: float, optional Delay, in seconds, before the process is actually stopped. If autoStartChildren is activated in the Server, this value is propagated to the children objects. Defaults to 0.

Note: if the method setStopDelay(x) was called before calling stop(wait) with a positive *wait* value, the *wait* value won't overwrite the value given to setStopDelay for the current object, but will be the one propagated to children objects. This allow to set a waiting time for a specific object with setStopDelay whithout changing the global delay time given to the stop method.

Fader and Adsr objects ignore waiting time given to the stop method because they already implement a delayed processing triggered by the stop call.

OscSend

class OscSend(*input*, *port*, *address*, *host='127.0.0.1'*)
Sends values over a network via the Open Sound Control protocol.

Uses the OSC protocol to share values to other softwares or other computers. Only the first value of each input buffersize will be sent on the OSC port.

Parent *PyoObject*

Args

input: PyoObject Input signal.

port: int Port on which values are sent. Receiver should listen on the same port.

address: string Address used on the port to identify values. Address is in the form of a Unix path (ex.: '/pitch').

host: string, optional IP address of the target computer. The default, '127.0.0.1', is the localhost.

Note: The out() method is bypassed. OscSend's signal can not be sent to audio outs.

OscSend has no *mul* and *add* attributes.

```
>>> s = Server().boot()
>>> s.start()
>>> a = Sine(freq=[1,1.5], mul=[100,.1], add=[600, .1])
>>> b = OscSend(a, port=10001, address=['/pitch','/amp'])
```

setInput(*x*, *fadetime=0.05*)
Replace the *input* attribute.

Args

x: PyoObject New signal to process.

fadetime: float, optional Crossfade time between old and new input. Defaults to 0.05.

out(*chnl=0*, *inc=1*, *dur=0*, *delay=0*)

Start processing and send samples to audio output beginning at *chnl*.

This method returns *self*, allowing it to be applied at the object creation.

Args

chnl: int, optional Physical output assigned to the first audio stream of the object. Defaults to 0.

inc: int, optional Output channel increment value. Defaults to 1.

dur: float, optional Duration, in seconds, of the object's activation. The default is 0 and means infinite duration.

delay: float, optional Delay, in seconds, before the object's activation. Defaults to 0.

If *chnl* >= 0, successive streams increment the output number by *inc* and wrap around the global number of channels.

If *chnl* is negative, streams begin at 0, increment the output number by *inc* and wrap around the global number of channels. Then, the list of streams is scrambled.

If *chnl* is a list, successive values in the list will be assigned to successive streams.

setMul(*x*)

Replace the *mul* attribute.

Args

x: float or PyoObject New *mul* attribute.

setAdd(*x*)

Replace the *add* attribute.

Args

x: float or PyoObject New *add* attribute.

setBufferRate(*x*)

Sets how many buffers to wait before sending a new value.

Args

x: int Changes the data output frequency in multiples of the buffer size. Should be greater or equal to 1.

input

PyoObject. Input signal.

Routing

Set of objects to manage audio voice routing and spread of a sound signal into a stereo or multi-channel sound field.

Mixer

class Mixer(*outs=2*, *chnls=1*, *time=0.025*, *mul=1*, *add=0*)

Audio mixer.

Mixer mixes multiple inputs to an arbitrary number of outputs with independant amplitude values per mixing channel and a user defined portamento applied on amplitude changes.

Parent *PyoObject*

Args

outs: int, optional Number of outputs of the mixer. Available at initialization time only. Defaults to 2.

chnls: int, optional Number of channels per output. Available at initialization time only. Defaults to 1.

time: float, optional Duration, in seconds, of a portamento applied on a new amplitude value for a mixing channel. Defaults to 0.025.

Note: User can retrieve each of the output channels by calling the Mixer object with the desired channel between square brackets (see example).

```
>>> s = Server().boot()
>>> s.start()
>>> a = SfPlayer(SNDS_PATH+"/transparent.aif", loop=True, mul=.2)
>>> b = FM(carrier=200, ratio=[.5013,.4998], index=6, mul=.2)
>>> mm = Mixer(outs=3, chnls=2, time=.025)
>>> fx1 = Disto(mm[0], drive=.9, slope=.9, mul=.1).out()
>>> fx2 = Freeverb(mm[1], size=.8, damp=.8, mul=.5).out()
>>> fx3 = Harmonizer(mm[2], transpo=1, feedback=.75, mul=.5).out()
>>> mm.addInput(0, a)
>>> mm.addInput(1, b)
>>> mm.setAmp(0,0,.5)
>>> mm.setAmp(0,1,.5)
>>> mm.setAmp(1,2,.5)
>>> mm.setAmp(1,1,.5)
```

setTime(*x*)

Sets the portamento duration in seconds.

Args

x: float New portamento duration.

addInput(*voice*, *input*)

Adds an audio object in the mixer's inputs.

This method returns the key (voice argument or generated key if voice=None).

Args

voice: int or string Key in the mixer dictionary for this input. If None, a unique key between 0 and 32767 will be automatically generated.

input: PyoObject Audio object to add to the mixer.

delInput(*voice*)

Removes an audio object from the mixer's inputs.

Args

voice: int or string Key in the mixer dictionary assigned to the input to remove.

setAmp(*vin*, *vout*, *amp*)
Sets the amplitude of a mixing channel.

Args

vin: int or string Key in the mixer dictionary of the desired input.

vout: int Ouput channel where to send the signal.

amp: float Amplitude value for this mixing channel.

getChannels()
Returns the Mixer's channels dictionary.

getKeys()
Returns the list of current keys in the Mixer's channels dictionary.

ctrl(*map_list=None*, *title=None*, *wxnoserver=False*)
Opens a sliders window to control the parameters of the object. Only parameters that can be set to a PyoObject are allowed to be mapped on a slider.

If a list of values are given to a parameter, a multisliders will be used to control each stream independently.

Args

map_list: list of SLMap objects, optional Users defined set of parameters scaling. There is default scaling for each object that accept *ctrl* method.

title: string, optional Title of the window. If none is provided, the name of the class is used.

wxnoserver: boolean, optional With wxPython graphical toolkit, if True, tells the interpreter that there will be no server window.

If *wxnoserver* is set to True, the interpreter will not wait for the server GUI before showing the controller window.

time
float. Portamento.

Pan

class Pan(*input*, *outs=2*, *pan=0.5*, *spread=0.5*, *mul=1*, *add=0*)
Cosinus panner with control on the spread factor.

Parent *PyoObject*

Args

input: PyoObject Input signal to process.

outs: int, optional Number of channels on the panning circle. Available at initialization time only. Defaults to 2.

pan: float or PyoObject Position of the sound on the panning circle, between 0 and 1. Defaults to 0.5.

spread: float or PyoObject Amount of sound leaking to the surrounding channels, between 0 and 1. Defaults to 0.5.

Note: When used with two output channels, the algorithm used is the cosine/sine constant power panning law. The SPan object uses the square root of intensity constant power panning law.

See also:

SPan, *Switch*, *Selector*

```
>>> s = Server(nchnls=2).boot()
>>> s.start()
>>> a = Noise(mul=.2)
>>> lfo = Sine(freq=1, mul=.5, add=.5)
>>> p = Pan(a, outs=2, pan=lfo).out()
```

setInput(*x*, *fadetime=0.05*)
Replace the *input* attribute.

Args

x: PyoObject New signal to process.

fadetime: float, optional Crossfade time between old and new input. Default to 0.05.

setPan(*x*)
Replace the *pan* attribute.

Args

x: float or PyoObject new *pan* attribute.

setSpread(*x*)
Replace the *spread* attribute.

Args

x: float or PyoObject new *spread* attribute.

ctrl(*map_list=None*, *title=None*, *wxnoserver=False*)
Opens a sliders window to control the parameters of the object. Only parameters that can be set to a PyoObject are allowed to be mapped on a slider.

If a list of values are given to a parameter, a multisliders will be used to control each stream independently.

Args

map_list: list of SLMap objects, optional Users defined set of parameters scaling. There is default scaling for each object that accept *ctrl* method.

title: string, optional Title of the window. If none is provided, the name of the class is used.

wxnoserver: boolean, optional With wxPython graphical toolkit, if True, tells the interpreter that there will be no server window.

If *wxnoserver* is set to True, the interpreter will not wait for the server GUI before showing the controller window.

input
PyoObject. Input signal to process.

pan
float or PyoObject. Position of the sound on the panning circle.

spread
float or PyoObject. Amount of sound leaking to the surrounding channels.

SPan

class SPan(*input*, *outs=2*, *pan=0.5*, *mul=1*, *add=0*)
Simple equal power panner.

Parent *PyoObject*

Args

input: PyoObject Input signal to process.

outs: int, optional Number of channels on the panning circle. Available at initialization time only. Defaults to 2.

pan: float or PyoObject Position of the sound on the panning circle, between 0 and 1. Defaults to 0.5.

Note: When used with two output channels, the algorithm used is the square root of intensity constant power panning law. The Pan object uses the cosine/sine constant power panning law.

See also:

Pan, *Switch*, *Selector*

```
>>> s = Server(nchnls=2).boot()
>>> s.start()
>>> a = Noise(mul=.2)
>>> lfo = Sine(freq=1, mul=.5, add=.5)
>>> p = SPan(a, outs=2, pan=lfo).out()
```

setInput(*x*, *fadetime=0.05*)
Replace the *input* attribute.

Args

x: PyoObject New signal to process.

fadetime: float, optional Crossfade time between old and new input. Default to 0.05.

setPan(*x*)
Replace the *pan* attribute.

Args

x: float or PyoObject new *pan* attribute.

ctrl(*map_list=None*, *title=None*, *wxnoserver=False*)
Opens a sliders window to control the parameters of the object. Only parameters that can be set to a PyoObject are allowed to be mapped on a slider.

If a list of values are given to a parameter, a multisliders will be used to control each stream independently.

Args

map_list: list of SLMap objects, optional Users defined set of parameters scaling. There is default scaling for each object that accept *ctrl* method.

title: string, optional Title of the window. If none is provided, the name of the class is used.

wxnoserver: boolean, optional With wxPython graphical toolkit, if True, tells the interpreter that there will be no server window.

If *wxnoserver* is set to True, the interpreter will not wait for the server GUI before showing the controller window.

input
PyoObject. Input signal to process.

pan
float or PyoObject. Position of the sound on the panning circle.

Selector

class Selector(*inputs*, *voice=0.0*, *mul=1*, *add=0*)
Audio selector.

Selector takes multiple PyoObjects in input and interpolates between them to generate a single output.

Parent *PyoObject*

Args

inputs: list of PyoObject Audio objects to interpolate from.

voice: float or PyoObject, optional Voice position pointer, between 0 and len(inputs)-1. Defaults to 0.

```
>>> s = Server().boot()
>>> s.start()
>>> a = SfPlayer(SNDS_PATH + "/transparent.aif", speed=[.999,1], loop=True, mul=.
↪3)
>>> b = Noise(mul=.1)
>>> c = SfPlayer(SNDS_PATH + "/accord.aif", speed=[.999,1], loop=True, mul=.5)
>>> lf = Sine(freq=.1, add=1)
>>> d = Selector(inputs=[a,b,c], voice=lf).out()
```

setInputs(*x*)
Replace the *inputs* attribute.

Args

x: list of PyoObject new *inputs* attribute.

setVoice(*x*)
Replace the *voice* attribute.

Args

x: float or PyoObject new *voice* attribute.

setMode(*x*)
Change the algorithm used to interpolate between inputs.

if inputs are phase correlated you should use a linear fade.

Args

x: int {0, 1} If 0 (the default) the equal power law is used to interpolate bewtween sources. If 1, linear fade is used instead.

ctrl(*map_list=None*, *title=None*, *wxnoserver=False*)
Opens a sliders window to control the parameters of the object. Only parameters that can be set to a PyoObject are allowed to be mapped on a slider.

If a list of values are given to a parameter, a multisliders will be used to control each stream independently.

Args

map_list: list of SLMap objects, optional Users defined set of parameters scaling. There is default scaling for each object that accept *ctrl* method.

title: string, optional Title of the window. If none is provided, the name of the class is used.

wxnoserver: boolean, optional With wxPython graphical toolkit, if True, tells the interpreter that there will be no server window.

If *wxnoserver* is set to True, the interpreter will not wait for the server GUI before showing the controller window.

inputs
List of PyoObjects. Audio objects to interpolate from.

voice
float or PyoObject. Voice position pointer.

Switch

class Switch(*input*, *outs=2*, *voice=0.0*, *mul=1*, *add=0*)
Audio switcher.

Switch takes an audio input and interpolates between multiple outputs.

User can retrieve the different streams by calling the output number between brackets. obj[0] retrieve the first stream, obj[outs-1] the last one.

Parent *PyoObject*

Args

input: PyoObject Input signal to process.

outs: int, optional Number of outputs. Available at initialization time only. Defaults to 2.

voice: float or PyoObject Voice position pointer, between 0 and (outs-1) / len(input). Defaults to 0.

```
>>> s = Server(nchnls=2).boot()
>>> s.start()
>>> a = SfPlayer(SNDS_PATH + "/transparent.aif", speed=[.999,1], loop=True, mul=.
↪3)
>>> lf = Sine(freq=.25, mul=1, add=1)
>>> b = Switch(a, outs=6, voice=lf)
>>> c = WGVerb(b[0:2], feedback=.8).out()
>>> d = Disto(b[2:4], drive=.9, mul=.1).out()
>>> e = Delay(b[4:6], delay=.2, feedback=.6).out()
```

setInput(*x*, *fadetime=0.05*)
Replace the *input* attribute.

Args

x: PyoObject New signal to process.

fadetime: float, optional Crossfade time between old and new input. Default to 0.05.

setVoice(*x*)
Replace the *voice* attribute.

Args

x: float or PyoObject new *voice* attribute.

ctrl(*map_list=None*, *title=None*, *wxnoserver=False*)
Opens a sliders window to control the parameters of the object. Only parameters that can be set to a PyoObject are allowed to be mapped on a slider.

If a list of values are given to a parameter, a multisliders will be used to control each stream independently.

Args

map_list: list of SLMap objects, optional Users defined set of parameters scaling. There is default scaling for each object that accept *ctrl* method.

title: string, optional Title of the window. If none is provided, the name of the class is used.

wxnoserver: boolean, optional With wxPython graphical toolkit, if True, tells the interpreter that there will be no server window.

If *wxnoserver* is set to True, the interpreter will not wait for the server GUI before showing the controller window.

input
PyoObject. Input signal to process.

voice
float or PyoObject. Voice position pointer.

VoiceManager

class VoiceManager(*input*, *triggers=None*, *mul=1*, *add=0*)
Polyphony voice manager.

A trigger in input ask the object for a voice number and the object returns the first free one. The voice number is then disable until a trig comes at the same position in the list of triggers given at the argument *triggers*.

Usually, the trigger enabling the voice number will come from the process started with the object output. So, it's common to leave the *triggers* argument to None and set the list of triggers afterward with the *setTriggers* method. The maximum number of voices generated by the object is the length of the trigger list.

If there is no free voice, the object outputs -1.0 continuously.

Parent `PyoObject`

Args

input: PyoObject Trigger stream asking for new voice numbers.

triggers: PyoObject or list of PyoObject, optional List of mono PyoObject sending triggers. Can be a multi-streams PyoObject but not a mix of both.

Ordering in the list corresponds to voice numbers. Defaults to None.

```
>>> s = Server().boot()
>>> s.start()
>>> env = CosTable([(0,0),(100,1),(500,.5),(8192,0)])
>>> delta = RandDur(min=.05, max=.1)
>>> vm = VoiceManager(Change(delta))
>>> sel = Select(vm, value=[0,1,2,3])
>>> pit = TrigChoice(sel, choice=[midiToHz(x) for x in [60,63,67,70,72]])
>>> amp = TrigEnv(sel, table=env, dur=.5, mul=.25)
>>> synth1 = SineLoop(pit, feedback=.07, mul=amp).out()
>>> vm.setTriggers(amp["trig"])
```

setInput (*x*, *fadetime=0.05*)
Replace the *input* attribute.

Args

x: PyoObject New signal to process.

fadetime: float, optional Crossfade time between old and new input. Defaults to 0.05.

setTriggers (*x*)
Replace the *triggers* attribute.

Args

x: PyoObject or list of PyoObject New *triggers* attribute.

input
PyoObject. Trigger stream asking for new voice numbers.

triggers
list of PyoObject. Trigger streams enabling voices.

HRTF

class HRTF (*input*, *azimuth=0.0*, *elevation=0.0*, *hrtfdata=None*, *mul=1*, *add=0*)
Head-Related Transfert Function 3D spatialization.

HRTF describes how a given sound wave input is filtered by the diffraction and reflection properties of the head, pinna, and torso, before the sound reaches the transduction machinery of the eardrum and inner ear.

This object takes a source signal and spatialises it in the 3D space around a listener by convolving the source with stored Head Related Impulse Response (HRIR) based filters. This works under ideal listening context, ie. when listening with headphones.

HRTF generates two outpout streams per input stream.

Parent *PyoObject*

Args

input: PyoObject Input signal to process.

azimuth: float or PyoObject Position of the sound on the horizontal plane, between -180 and 180 degrees. Defaults to 0.

elevation: float or PyoObject Position of the sound on the vertical plane, between -40 and 90 degrees. Defaults to 0.

hrtfdata: HRTFData, optional Custom impulse responses dataset. Not used yet. Leave it to None.

Note: Currently, HRTF uses the HRIRs from Gardner and Martin at MIT lab.

http://alumni.media.mit.edu/~kdm/hrtfdoc/hrtfdoc.html

These impulses where recorded at a sampling rate of 44.1 kHz. They will probably work just as well at 48 kHz, but surely not at higher sampling rate.

There is plan to add the possibility for the user to load its own dataset in the future.

See also:

SPan, *Pan*

```
>>> s = Server(nchnls=2).boot()
>>> s.start()
>>> sf = SfPlayer(SNDS_PATH + "/transparent.aif", loop=True)
>>> azi = Phasor(0.2, mul=360)
>>> ele = Sine(0.1).range(0, 90)
>>> mv = HRTF(sf, azi, ele, mul=0.5).out()
```

setInput (*x*, *fadetime=0.05*)

Replace the *input* attribute.

Args

x: PyoObject New signal to process.

fadetime: float, optional Crossfade time between old and new input. Default to 0.05.

setAzimuth (*x*)

Replace the *azimuth* attribute.

Args

x: float or PyoObject new *azimuth* attribute.

setElevation (*x*)

Replace the *elevation* attribute.

Args

x: float or PyoObject new *elevation* attribute.

ctrl (*map_list=None*, *title=None*, *wxnoserver=False*)

Opens a sliders window to control the parameters of the object. Only parameters that can be set to a PyoObject are allowed to be mapped on a slider.

If a list of values are given to a parameter, a multisliders will be used to control each stream independently.

Args

map_list: list of SLMap objects, optional Users defined set of parameters scaling. There is default scaling for each object that accept *ctrl* method.

title: string, optional Title of the window. If none is provided, the name of the class is used.

wxnoserver: boolean, optional With wxPython graphical toolkit, if True, tells the interpreter that there will be no server window.

If *wxnoserver* is set to True, the interpreter will not wait for the server GUI before showing the controller window.

input
PyoObject. Input signal to process.

azimuth
float or PyoObject. Position of the sound on the horizontal plane.

elevation
float or PyoObject. Position of the sound on the vertical plane.

Event Sequencing

Set of objects that call Python functions from triggers or number counts. Useful for event sequencing.

CallAfter

class CallAfter(*function*, *time=1*, *arg=None*)
Calls a Python function after a given time.

Parent *PyoObject*

Args

function: Python callable (function or method) Python callable execute after *time* seconds.

time: float, optional Time, in seconds, before the call. Default to 1.

arg: any Python object, optional Argument sent to the called function. Default to None.

Note: The out() method is bypassed. CallAfter doesn't return signal.

CallAfter has no *mul* and *add* attributes.

If *arg* is None, the function must be defined without argument:

```
>>> def tocall():
>>>     # function's body
```

If *arg* is not None, the function must be defined with one argument:

```
>>> def tocall(arg):
>>>     print(arg)
```

The object is not deleted after the call. The user must delete it himself.

```
>>> s = Server().boot()
>>> s.start()
>>> # Start an oscillator with a frequency of 250 Hz
>>> syn = SineLoop(freq=[250,251], feedback=.07, mul=.2).out()
>>> def callback(arg):
...     # Change the oscillator's frequency to 300 Hz after 2 seconds
...     syn.freq = arg
>>> a = CallAfter(callback, 2, [300,301])
```

out(*x=0*, *inc=1*, *dur=0*, *delay=0*)
Start processing and send samples to audio output beginning at *chnl*.

This method returns *self*, allowing it to be applied at the object creation.

Args

chnl: int, optional Physical output assigned to the first audio stream of the object. Defaults to 0.

inc: int, optional Output channel increment value. Defaults to 1.

dur: float, optional Duration, in seconds, of the object's activation. The default is 0 and means infinite duration.

delay: float, optional Delay, in seconds, before the object's activation. Defaults to 0.

If *chnl* >= 0, successive streams increment the output number by *inc* and wrap around the global number of channels.

If *chnl* is negative, streams begin at 0, increment the output number by *inc* and wrap around the global number of channels. Then, the list of streams is scrambled.

If *chnl* is a list, successive values in the list will be assigned to successive streams.

setMul(*x*)
Replace the *mul* attribute.

Args

x: float or PyoObject New *mul* attribute.

setAdd(*x*)
Replace the *add* attribute.

Args

x: float or PyoObject New *add* attribute.

setSub(*x*)
Replace and inverse the *add* attribute.

Args

x: float or PyoObject New inversed *add* attribute.

setDiv(*x*)
Replace and inverse the *mul* attribute.

Args

x: float or PyoObject New inversed *mul* attribute.

Pattern

class Pattern(*function*, *time=1*, *arg=None*)
Periodically calls a Python function.

The play() method starts the pattern timer and is not called at the object creation time.

Parent *PyoObject*

Args

function: Python callable (function or method) Python function to be called periodically.

time: float or PyoObject, optional Time, in seconds, between each call. Default to 1.

arg: anything, optional Argument sent to the function's call. If None, the function will be called without argument. Defaults to None.

Note: The out() method is bypassed. Pattern doesn't return signal.

Pattern has no *mul* and *add* attributes.

If *arg* is None, the function must be defined without argument:

```
>>> def tocall():
>>>     # function's body
```

If *arg* is not None, the function must be defined with one argument:

```
>>> def tocall(arg):
>>>     print(arg)
```

```
>>> s = Server().boot()
>>> s.start()
>>> t = HarmTable([1,0,.33,0,.2,0,.143,0,.111])
>>> a = Osc(table=t, freq=[250,251], mul=.2).out()
>>> def pat():
...     f = random.randrange(200, 401, 25)
...     a.freq = [f, f+1]
>>> p = Pattern(pat, .125)
>>> p.play()
```

setFunction(*x*)

Replace the *function* attribute.

Args

x: Python callable (function or method) new *function* attribute.

setTime(*x*)

Replace the *time* attribute.

Args

x: float or PyoObject New *time* attribute.

setArg(*x*)

Replace the *arg* attribute.

Args

x: Anything new *arg* attribute.

out(*x=0*, *inc=1*, *dur=0*, *delay=0*)

Start processing and send samples to audio output beginning at *chnl*.

This method returns *self*, allowing it to be applied at the object creation.

Args

chnl: int, optional Physical output assigned to the first audio stream of the object. Defaults to 0.

inc: int, optional Output channel increment value. Defaults to 1.

dur: float, optional Duration, in seconds, of the object's activation. The default is 0 and means infinite duration.

delay: float, optional Delay, in seconds, before the object's activation. Defaults to 0.

If *chnl* >= 0, successive streams increment the output number by *inc* and wrap around the global number of channels.

If *chnl* is negative, streams begin at 0, increment the output number by *inc* and wrap around the global number of channels. Then, the list of streams is scrambled.

If *chnl* is a list, successive values in the list will be assigned to successive streams.

setMul(*x*)
Replace the *mul* attribute.

Args

x: float or PyoObject New *mul* attribute.

setAdd(*x*)
Replace the *add* attribute.

Args

x: float or PyoObject New *add* attribute.

setSub(*x*)
Replace and inverse the *add* attribute.

Args

x: float or PyoObject New inversed *add* attribute.

setDiv(*x*)
Replace and inverse the *mul* attribute.

Args

x: float or PyoObject New inversed *mul* attribute.

ctrl(*map_list=None*, *title=None*, *wxnoserver=False*)
Opens a sliders window to control the parameters of the object. Only parameters that can be set to a PyoObject are allowed to be mapped on a slider.

If a list of values are given to a parameter, a multisliders will be used to control each stream independently.

Args

map_list: list of SLMap objects, optional Users defined set of parameters scaling. There is default scaling for each object that accept *ctrl* method.

title: string, optional Title of the window. If none is provided, the name of the class is used.

wxnoserver: boolean, optional With wxPython graphical toolkit, if True, tells the interpreter that there will be no server window.

If *wxnoserver* is set to True, the interpreter will not wait for the server GUI before showing the controller window.

function
Python callable. Function to be called.

time
float or PyoObject. Time, in seconds, between each call.

arg
Anything. Callable's argument.

Score

class Score(*input*, *fname='event_'*)

Calls functions by incrementation of a preformatted name.

Score takes audio stream containning integers in input and calls a function whose name is the concatenation of *fname* and the changing integer.

Can be used to sequence events, first by creating functions p0, p1, p2, etc. and then, by passing a counter to a Score object with "p" as *fname* argument. Functions are called without parameters.

Parent *PyoObject*

Args

input: PyoObject Audio signal. Must contains integer numbers. Integer must change before calling its function again.

fname: string, optional Name of the functions to be called. Defaults to "**event_**", meaning that the object will call the function "event_0", "event_1", "event_2", and so on... Available at initialization time only.

Note: The out() method is bypassed. Score's signal can not be sent to audio outs.

Score has no *mul* and *add* attributes.

See also:

Pattern, *TrigFunc*

```
>>> s = Server().boot()
>>> s.start()
>>> a = SineLoop(freq=[200,300,400,500], feedback=0.05, mul=.1).out()
>>> def event_0():
...     a.freq=[200,300,400,500]
>>> def event_1():
...     a.freq=[300,400,450,600]
>>> def event_2():
...     a.freq=[150,375,450,525]
>>> m = Metro(1).play()
>>> c = Counter(m, min=0, max=3)
>>> sc = Score(c)
```

out(*chnl=0*, *inc=1*, *dur=0*, *delay=0*)

Start processing and send samples to audio output beginning at *chnl*.

This method returns *self*, allowing it to be applied at the object creation.

Args

chnl: int, optional Physical output assigned to the first audio stream of the object. Defaults to 0.

inc: int, optional Output channel increment value. Defaults to 1.

dur: float, optional Duration, in seconds, of the object's activation. The default is 0 and means infinite duration.

delay: float, optional Delay, in seconds, before the object's activation. Defaults to 0.

If *chnl* >= 0, successive streams increment the output number by *inc* and wrap around the global number of channels.

If *chnl* is negative, streams begin at 0, increment the output number by *inc* and wrap around the global number of channels. Then, the list of streams is scrambled.

If *chnl* is a list, successive values in the list will be assigned to successive streams.

setMul(*x*)
Replace the *mul* attribute.

Args

x: float or PyoObject New *mul* attribute.

setAdd(*x*)
Replace the *add* attribute.

Args

x: float or PyoObject New *add* attribute.

setInput(*x*, *fadetime=0.05*)
Replace the *input* attribute.

Args

x: PyoObject New signal to process.

fadetime: float, optional Crossfade time between old and new input. Defaults to 0.05.

input
PyoObject. Audio signal sending integer numbers.

Soundfile Players

Play soundfiles from the disk.

SfMarkerXXX objects use markers features (store in the header) from an AIFF file to create more specific reading patterns.

SfPlayer

class SfPlayer(*path*, *speed=1*, *loop=False*, *offset=0*, *interp=2*, *mul=1*, *add=0*)
Soundfile player.

Reads audio data from a file using one of several available interpolation types. User can alter its pitch with the *speed* attribute. The object takes care of sampling rate conversion to match the Server sampling rate setting.

Parent `PyoObject`

Args

path: string Full path name of the sound to read.

speed: float or PyoObject, optional Transpose the pitch of input sound by this factor. Defaults to 1.

1 is the original pitch, lower values play sound slower, and higher values play sound faster.

Negative values results in playing sound backward.

Although the *speed* attribute accepts audio rate signal, its value is updated only once per buffer size.

loop: bool, optional If set to True, sound will play in loop. Defaults to False.

offset: float, optional Time in seconds of input sound to be skipped, assuming speed = 1. Defaults to 0.

interp: int, optional

Interpolation type. Defaults to 2.

1. no interpolation
2. linear
3. cosinus
4. cubic

Note: SfPlayer will send a trigger signal at the end of the playback if loop is off or any time it wraps around if loop is on. User can retrieve the trigger streams by calling obj['trig']:

```
>>> sf = SfPlayer(SNDS_PATH + "/transparent.aif").out()
>>> trig = TrigRand(sf['trig'])
```

Note that the object will send as many trigs as there is channels in the sound file. If you want to retrieve only one trig, only give the first stream to the next object:

```
>>> def printing():
...     print("one trig!")
>>> sf = SfPlayer("/stereo/sound/file.aif").out()
>>> trig = TrigFunc(sf['trig'][0], printing)
```

```
>>> s = Server().boot()
>>> s.start()
>>> snd = SNDS_PATH + "/transparent.aif"
>>> sf = SfPlayer(snd, speed=[.75,.8], loop=True, mul=.3).out()
```

setPath (*path*)

Sets a new sound to read.

The number of channels of the new sound must match those of the sound loaded at initialization time.

Args

path: string Full path of the new sound.

setSound (*path*)

Sets a new sound to read.

The number of channels of the new sound must match those of the sound loaded at initialization time.

Args

path: string Full path of the new sound.

setSpeed (*x*)

Replace the *speed* attribute.

Args

x: float or PyoObject new *speed* attribute.

setLoop(*x*)
Replace the *loop* attribute.

Args

x: bool {True, False} new *loop* attribute.

setOffset(*x*)
Replace the *offset* attribute.

Args

x: float new *offset* attribute.

setInterp(*x*)
Replace the *interp* attribute.

Args

x: int {1, 2, 3, 4} new *interp* attribute.

ctrl(*map_list=None*, *title=None*, *wxnoserver=False*)
Opens a sliders window to control the parameters of the object. Only parameters that can be set to a PyoObject are allowed to be mapped on a slider.

If a list of values are given to a parameter, a multisliders will be used to control each stream independently.

Args

map_list: list of SLMap objects, optional Users defined set of parameters scaling. There is default scaling for each object that accept *ctrl* method.

title: string, optional Title of the window. If none is provided, the name of the class is used.

wxnoserver: boolean, optional With wxPython graphical toolkit, if True, tells the interpreter that there will be no server window.

If *wxnoserver* is set to True, the interpreter will not wait for the server GUI before showing the controller window.

path
string. Full path of the sound.

sound
string. Alias to the *path* attribute.

speed
float or PyoObject. Transposition factor.

loop
bool. Looping mode.

offset
float. Time, in seconds, of the first sample to read.

interp
int {1, 2, 3, 4}. Interpolation method.

SfMarkerLooper

class SfMarkerLooper(*path*, *speed=1*, *mark=0*, *interp=2*, *mul=1*, *add=0*)

AIFF with markers soundfile looper.

Reads audio data from a AIFF file using one of several available interpolation types. User can alter its pitch with the *speed* attribute. The object takes care of sampling rate conversion to match the Server sampling rate setting.

The reading pointer loops a specific marker (from the MARK chunk in the header of the AIFF file) until it received a new integer in the *mark* attribute.

Parent *PyoObject*

Args

path: string Full path name of the sound to read.

speed: float or PyoObject, optional Transpose the pitch of input sound by this factor. Defaults to 1.

1 is the original pitch, lower values play sound slower, and higher values play sound faster.

Negative values results in playing sound backward.

Although the *speed* attribute accepts audio rate signal, its value is updated only once per buffer size.

mark: float or PyoObject, optional Integer denoting the marker to loop, in the range 0 -> len(getMarkers()). Defaults to 0.

interp: int, optional

Choice of the interpolation method. Defaults to 2.

1. no interpolation
2. linear
3. cosinus
4. cubic

```
>>> s = Server().boot()
>>> s.start()
>>> a = SfMarkerLooper(SNDS_PATH + '/transparent.aif', speed=[.999,1], mul=.3).
↪out()
>>> rnd = RandInt(len(a.getMarkers()), 2)
>>> a.mark = rnd
```

setSpeed(*x*)

Replace the *speed* attribute.

Args

x: float or PyoObject new *speed* attribute.

setMark(*x*)

Replace the *mark* attribute.

Args

x: float or PyoObject new *mark* attribute.

setInterp(*x*)
Replace the *interp* attribute.

Args

x: int {1, 2, 3, 4} new *interp* attribute.

getMarkers()
Returns a list of marker time values in samples.

ctrl(*map_list=None*, *title=None*, *wxnoserver=False*)
Opens a sliders window to control the parameters of the object. Only parameters that can be set to a PyoObject are allowed to be mapped on a slider.

If a list of values are given to a parameter, a multisliders will be used to control each stream independently.

Args

map_list: list of SLMap objects, optional Users defined set of parameters scaling. There is default scaling for each object that accept *ctrl* method.

title: string, optional Title of the window. If none is provided, the name of the class is used.

wxnoserver: boolean, optional With wxPython graphical toolkit, if True, tells the interpreter that there will be no server window.

If *wxnoserver* is set to True, the interpreter will not wait for the server GUI before showing the controller window.

speed
float or PyoObject. Transposition factor.

mark
float or PyoObject. Marker to loop.

interp
int {1, 2, 3, 4}. Interpolation method.

SfMarkerShuffler

class SfMarkerShuffler(*path*, *speed=1*, *interp=2*, *mul=1*, *add=0*)
AIFF with markers soundfile shuffler.

Reads audio data from a AIFF file using one of several available interpolation types. User can alter its pitch with the *speed* attribute. The object takes care of sampling rate conversion to match the Server sampling rate setting.

The reading pointer randomly choose a marker (from the MARK chunk in the header of the AIFF file) as its starting point and reads the samples until it reaches the following marker. Then, it choose another marker and reads from the new position and so on...

Parent `PyoObject`

Args

path: string Full path name of the sound to read. Can't e changed after initialization.

speed: float or PyoObject, optional Transpose the pitch of input sound by this factor. Defaults to 1.

1 is the original pitch, lower values play sound slower, and higher values play sound faster.

Negative values results in playing sound backward.

Although the *speed* attribute accepts audio rate signal, its value is updated only once per buffer size.

interp: int, optional

Choice of the interpolation method. Defaults to 2.

1. no interpolation
2. linear
3. cosinus
4. cubic

```
>>> s = Server().boot()
>>> s.start()
>>> sound = SNDS_PATH + "/transparent.aif"
>>> sf = SfMarkerShuffler(sound, speed=[1,1], mul=.3).out()
>>> sf.setRandomType("expon_min", 0.6)
```

setSpeed(*x*)

Replace the *speed* attribute.

Args

x: float or PyoObject new *speed* attribute.

setInterp(*x*)

Replace the *interp* attribute.

Args

x: int {1, 2, 3, 4} new *interp* attribute.

setRandomType(*dist=0*, *x=0.5*)

Set the random distribution type used to choose the markers.

Args

dist: int or string

The distribution type. Available distributions are:

0. uniform (default)
1. linear minimum
2. linear maximum
3. triangular
4. exponential minimum
5. exponential maximum
6. double (bi)exponential
7. cauchy
8. weibull
9. gaussian

x: float Distribution specific parameter, if applicable, as a float between 0 and 1. Defaults to 0.5.

Note: Depending on the distribution type, *x* parameter is applied as follow (names as string, or associated number can be used as *dist* parameter):

0. **uniform**
    - x: not used
1. **linear_min**
    - x: not used
2. **linear_max**
    - x: not used
3. **triangle**
    - x: not used
4. **expon_min**
    - x: slope {0 = no slope -> 1 = sharp slope}
5. **expon_max**
    - x: slope {0 = no slope -> 1 = sharp slope}
6. **biexpon**
    - x: bandwidth {0 = huge bandwidth -> 1 = narrow bandwidth}
7. **cauchy**
    - x: bandwidth {0 = huge bandwidth -> 1 = narroe bandwidth}
8. **weibull**
    - x: shape {0 = expon min => linear min => 1 = gaussian}
9. **gaussian**
    - x: bandwidth {0 = huge bandwidth -> 1 = narrow bandwidth}

getMarkers()
Returns a list of marker time values in samples.

ctrl(*map_list=None*, *title=None*, *wxnoserver=False*)
Opens a sliders window to control the parameters of the object. Only parameters that can be set to a PyoObject are allowed to be mapped on a slider.

If a list of values are given to a parameter, a multisliders will be used to control each stream independently.

Args

map_list: list of SLMap objects, optional Users defined set of parameters scaling. There is default scaling for each object that accept *ctrl* method.

title: string, optional Title of the window. If none is provided, the name of the class is used.

wxnoserver: boolean, optional With wxPython graphical toolkit, if True, tells the interpreter that there will be no server window.

If *wxnoserver* is set to True, the interpreter will not wait for the server GUI before showing the controller window.

speed
float or PyoObject. Transposition factor.

interp
int {1, 2, 3, 4}. Interpolation method.

Random generators

Set of objects that implement different kinds of random noise generators.

Choice

class Choice(*choice*, *freq=1.0*, *mul=1*, *add=0*)
Periodically choose a new value from a user list.

Choice chooses a new value from a predefined list of floats *choice* at a frequency specified by *freq* parameter. Choice will hold choosen value until next generation.

Parent *PyoObject*

Args

choice: list of floats or list of lists of floats Possible values for the random generation.

freq: float or PyoObject, optional Polling frequency. Defaults to 1.

```
>>> s = Server().boot()
>>> s.start()
>>> freqs = midiToHz([60,62,64,65,67,69,71,72])
>>> rnd = Choice(choice=freqs, freq=[3,4])
>>> a = SineLoop(rnd, feedback=0.05, mul=.2).out()
```

setChoice(*x*)
Replace the *choice* attribute.

Args

x: list of floats or list of lists of floats new *choice* attribute.

setFreq(*x*)
Replace the *freq* attribute.

Args

x: float or PyoObject new *freq* attribute.

ctrl(*map_list=None*, *title=None*, *wxnoserver=False*)
Opens a sliders window to control the parameters of the object. Only parameters that can be set to a PyoObject are allowed to be mapped on a slider.

If a list of values are given to a parameter, a multisliders will be used to control each stream independently.

Args

map_list: list of SLMap objects, optional Users defined set of parameters scaling. There is default scaling for each object that accept *ctrl* method.

title: string, optional Title of the window. If none is provided, the name of the class is used.

wxnoserver: boolean, optional With wxPython graphical toolkit, if True, tells the interpreter that there will be no server window.

If *wxnoserver* is set to True, the interpreter will not wait for the server GUI before showing the controller window.

choice
list of floats or list of lists of floats. Possible choices.

freq
float or PyoObject. Polling frequency.

LogiMap

class LogiMap(*chaos=0.6*, *freq=1.0*, *init=0.5*, *mul=1*, *add=0*)
Random generator based on the logistic map.

The logistic equation (sometimes called the Verhulst model or logistic growth curve) is a model of population growth first published by Pierre Verhulst (1845, 1847). The logistic map is a discrete quadratic recurrence equation derived from the logistic equation that can be effectively used as a number generator that exibit chaotic behavior. This object uses the following equation:

x[n] = (r + 3) * x[n-1] * (1.0 - x[n-1])

where 'r' is the randomization factor between 0 and 1.

Parent *PyoObject*

Args

chaos: float or PyoObject, optional Randomization factor, 0.0 < chaos < 1.0. Defaults to 0.6.

freq: float or PyoObject, optional Polling frequency. Defaults to 1.

init: float, optional Initial value, 0.0 < init < 1.0. Defaults to 0.5.

Note: The method play() resets the internal state to the initial value.

```
>>> s = Server().boot()
>>> s.start()
>>> val = LogiMap([0.6,0.65], [4,8])
>>> mid = Round(Scale(val, 0, 1, [36,48], [72,84]))
>>> hz = Snap(mid, [0,2,4,5,7,9,11], 1)
>>> env = CosTable([(0,0), (32,1), (4064,1), (4096,0), (8192,0)])
>>> amp = TrigEnv(Change(val), table=env, dur=[.25,.125], mul=0.3)
>>> osc = RCOsc(hz, mul=amp).out()
```

out(*chnl=0*, *inc=1*, *dur=0*, *delay=0*)
Start processing and send samples to audio output beginning at *chnl*.

This method returns *self*, allowing it to be applied at the object creation.

Args

chnl: int, optional Physical output assigned to the first audio stream of the object. Defaults to 0.

inc: int, optional Output channel increment value. Defaults to 1.

dur: float, optional Duration, in seconds, of the object's activation. The default is 0 and means infinite duration.

delay: float, optional Delay, in seconds, before the object's activation. Defaults to 0.

If *chnl* >= 0, successive streams increment the output number by *inc* and wrap around the global number of channels.

If *chnl* is negative, streams begin at 0, increment the output number by *inc* and wrap around the global number of channels. Then, the list of streams is scrambled.

If *chnl* is a list, successive values in the list will be assigned to successive streams.

setChaos(*x*)
Replace the *chaos* attribute.

Args

x: float or PyoObject new *chaos* attribute.

setFreq(*x*)
Replace the *freq* attribute.

Args

x: float or PyoObject new *freq* attribute.

ctrl(*map_list=None*, *title=None*, *wxnoserver=False*)
Opens a sliders window to control the parameters of the object. Only parameters that can be set to a PyoObject are allowed to be mapped on a slider.

If a list of values are given to a parameter, a multisliders will be used to control each stream independently.

Args

map_list: list of SLMap objects, optional Users defined set of parameters scaling. There is default scaling for each object that accept *ctrl* method.

title: string, optional Title of the window. If none is provided, the name of the class is used.

wxnoserver: boolean, optional With wxPython graphical toolkit, if True, tells the interpreter that there will be no server window.

If *wxnoserver* is set to True, the interpreter will not wait for the server GUI before showing the controller window.

chaos
float or PyoObject. Randomization factor.

freq
float or PyoObject. Polling frequency.

RandDur

class RandDur(*min=0.0*, *max=1.0*, *mul=1*, *add=0*)
Recursive time varying pseudo-random generator.

RandDur generates a pseudo-random number between *min* and *max* arguments and uses that number to set the delay time before the next generation. RandDur will hold the generated value until next generation.

Parent `PyoObject`

Args

min: float or PyoObject, optional Minimum value for the random generation. Defaults to 0.

max: float or PyoObject, optional Maximum value for the random generation. Defaults to 1.

```
>>> s = Server().boot()
>>> s.start()
>>> dur = RandDur(min=[.05,0.1], max=[.4,.5])
>>> trig = Change(dur)
>>> amp = TrigEnv(trig, table=HannTable(), dur=dur, mul=.2)
>>> freqs = midiToHz([60,63,67,70,72])
>>> freq = TrigChoice(trig, choice=freqs)
>>> a = LFO(freq=freq, type=2, mul=amp).out()
```

setMin(*x*)
Replace the *min* attribute.

Args

x: float or PyoObject new *min* attribute.

setMax(*x*)
Replace the *max* attribute.

Args

x: float or PyoObject new *max* attribute.

ctrl(*map_list=None*, *title=None*, *wxnoserver=False*)
Opens a sliders window to control the parameters of the object. Only parameters that can be set to a PyoObject are allowed to be mapped on a slider.

If a list of values are given to a parameter, a multisliders will be used to control each stream independently.

Args

map_list: list of SLMap objects, optional Users defined set of parameters scaling. There is default scaling for each object that accept *ctrl* method.

title: string, optional Title of the window. If none is provided, the name of the class is used.

wxnoserver: boolean, optional With wxPython graphical toolkit, if True, tells the interpreter that there will be no server window.

If *wxnoserver* is set to True, the interpreter will not wait for the server GUI before showing the controller window.

min
float or PyoObject. Minimum value.

max
float or PyoObject. Maximum value.

RandInt

class RandInt(*max=100*, *freq=1.0*, *mul=1*, *add=0*)
Periodic pseudo-random integer generator.

RandInt generates a pseudo-random integer number between 0 and *max* values at a frequency specified by *freq* parameter. RandInt will hold generated value until the next generation.

Parent *PyoObject*

Args

max: float or PyoObject, optional Maximum value for the random generation. Defaults to 100.

freq: float or PyoObject, optional Polling frequency. Defaults to 1.

```
>>> s = Server().boot()
>>> s.start()
>>> freq = RandInt(max=10, freq=5, mul=100, add=500)
>>> jit = Randi(min=0.99, max=1.01, freq=[2.33,3.41])
>>> a = SineLoop(freq*jit, feedback=0.03, mul=.2).out()
```

setMax(*x*)
Replace the *max* attribute.

Args

x: float or PyoObject new *max* attribute.

setFreq(*x*)
Replace the *freq* attribute.

Args

x: float or PyoObject new *freq* attribute.

ctrl(*map_list=None*, *title=None*, *wxnoserver=False*)
Opens a sliders window to control the parameters of the object. Only parameters that can be set to a PyoObject are allowed to be mapped on a slider.

If a list of values are given to a parameter, a multisliders will be used to control each stream independently.

Args

map_list: list of SLMap objects, optional Users defined set of parameters scaling. There is default scaling for each object that accept *ctrl* method.

title: string, optional Title of the window. If none is provided, the name of the class is used.

wxnoserver: boolean, optional With wxPython graphical toolkit, if True, tells the interpreter that there will be no server window.

If *wxnoserver* is set to True, the interpreter will not wait for the server GUI before showing the controller window.

max
float or PyoObject. Maximum value.

freq
float or PyoObject. Polling frequency.

Randh

class Randh(*min=0.0*, *max=1.0*, *freq=1.0*, *mul=1*, *add=0*)
Periodic pseudo-random generator.

Randh generates a pseudo-random number between *min* and *max* values at a frequency specified by *freq* parameter. Randh will hold generated value until next generation.

Parent *PyoObject*

Args

min: float or PyoObject, optional Minimum value for the random generation. Defaults to 0.

max: float or PyoObject, optional Maximum value for the random generation. Defaults to 1.

freq: float or PyoObject, optional Polling frequency. Defaults to 1.

```
>>> s = Server().boot()
>>> s.start()
>>> freq = Randh(500, 3000, 4)
>>> noze = Noise().mix(2)
>>> a = Biquad(noze, freq=freq, q=5, type=2, mul=.5).out()
```

setMin(*x*)
Replace the *min* attribute.

Args

x: float or PyoObject new *min* attribute.

setMax(*x*)
Replace the *max* attribute.

Args

x: float or PyoObject new *max* attribute.

setFreq(*x*)
Replace the *freq* attribute.

Args

x: float or PyoObject new *freq* attribute.

ctrl(*map_list=None*, *title=None*, *wxnoserver=False*)
Opens a sliders window to control the parameters of the object. Only parameters that can be set to a PyoObject are allowed to be mapped on a slider.

If a list of values are given to a parameter, a multisliders will be used to control each stream independently.

Args

map_list: list of SLMap objects, optional Users defined set of parameters scaling. There is default scaling for each object that accept *ctrl* method.

title: string, optional Title of the window. If none is provided, the name of the class is used.

wxnoserver: boolean, optional With wxPython graphical toolkit, if True, tells the interpreter that there will be no server window.

If *wxnoserver* is set to True, the interpreter will not wait for the server GUI before showing the controller window.

min
float or PyoObject. Minimum value.

max
float or PyoObject. Maximum value.

freq
float or PyoObject. Polling frequency.

Randi

class Randi(*min=0.0*, *max=1.0*, *freq=1.0*, *mul=1*, *add=0*)
Periodic pseudo-random generator with interpolation.

Randi generates a pseudo-random number between *min* and *max* values at a frequency specified by *freq* parameter. Randi will produce straight-line interpolation between current number and the next.

Parent *PyoObject*

Args

min: float or PyoObject, optional Minimum value for the random generation. Defaults to 0.

max: float or PyoObject, optional Maximum value for the random generation. Defaults to 1.

freq: float or PyoObject, optional Polling frequency. Defaults to 1.

```
>>> s = Server().boot()
>>> s.start()
>>> freq = Randi(500, 3000, 4)
>>> noze = Noise().mix(2)
>>> a = Biquad(noze, freq=freq, q=5, type=2, mul=.5).out()
```

setMin(*x*)
Replace the *min* attribute.

Args

x: float or PyoObject new *min* attribute.

setMax(*x*)
Replace the *max* attribute.

Args

x: float or PyoObject new *max* attribute.

setFreq(*x*)
Replace the *freq* attribute.

Args

x: float or PyoObject new *freq* attribute.

ctrl(*map_list=None*, *title=None*, *wxnoserver=False*)
Opens a sliders window to control the parameters of the object. Only parameters that can be set to a PyoObject are allowed to be mapped on a slider.

If a list of values are given to a parameter, a multisliders will be used to control each stream independently.

Args

map_list: list of SLMap objects, optional Users defined set of parameters scaling. There is default scaling for each object that accept *ctrl* method.

title: string, optional Title of the window. If none is provided, the name of the class is used.

wxnoserver: boolean, optional With wxPython graphical toolkit, if True, tells the interpreter that there will be no server window.

If *wxnoserver* is set to True, the interpreter will not wait for the server GUI before showing the controller window.

min
float or PyoObject. Minimum value.

max
float or PyoObject. Maximum value.

freq
float or PyoObject. Polling frequency.

Urn

class Urn(*max=100*, *freq=1.0*, *mul=1*, *add=0*)
Periodic pseudo-random integer generator without duplicates.

Urn generates a pseudo-random integer number between 0 and *max* values at a frequency specified by *freq* parameter. Urn will hold generated value until the next generation. Urn works like RandInt, except that it keeps track of each number which has been generated. After all numbers have been outputed, the pool is reseted and the object send a trigger signal.

Parent *PyoObject*

Args

max: int, optional Maximum value for the random generation. Defaults to 100.

freq: float or PyoObject, optional Polling frequency. Defaults to 1.

Note: Urn will send a trigger signal when the pool is empty. User can retreive the trigger streams by calling obj['trig']. Useful to synchronize other processes.

```
>>> s = Server().boot()
>>> s.start()
>>> mid = Urn(max=12, freq=10, add=60)
>>> fr = MToF(mid)
>>> sigL = SineLoop(freq=fr, feedback=.08, mul=0.3).out()
>>> amp = TrigExpseg(mid["trig"], [(0,0),(.01,.25),(1,0)])
>>> sigR = SineLoop(midiToHz(84), feedback=0.05, mul=amp).out(1)
```

out(*chnl=0*, *inc=1*, *dur=0*, *delay=0*)
Start processing and send samples to audio output beginning at *chnl*.

This method returns *self*, allowing it to be applied at the object creation.

Args

chnl: int, optional Physical output assigned to the first audio stream of the object. Defaults to 0.

inc: int, optional Output channel increment value. Defaults to 1.

dur: float, optional Duration, in seconds, of the object's activation. The default is 0 and means infinite duration.

delay: float, optional Delay, in seconds, before the object's activation. Defaults to 0.

If *chnl* >= 0, successive streams increment the output number by *inc* and wrap around the global number of channels.

If *chnl* is negative, streams begin at 0, increment the output number by *inc* and wrap around the global number of channels. Then, the list of streams is scrambled.

If *chnl* is a list, successive values in the list will be assigned to successive streams.

setMax(*x*)
Replace the *max* attribute.

Args

x: int new *max* attribute.

setFreq(*x*)
Replace the *freq* attribute.

Args

x: float or PyoObject new *freq* attribute.

ctrl(*map_list=None*, *title=None*, *wxnoserver=False*)
Opens a sliders window to control the parameters of the object. Only parameters that can be set to a PyoObject are allowed to be mapped on a slider.

If a list of values are given to a parameter, a multisliders will be used to control each stream independently.

Args

map_list: list of SLMap objects, optional Users defined set of parameters scaling. There is default scaling for each object that accept *ctrl* method.

title: string, optional Title of the window. If none is provided, the name of the class is used.

wxnoserver: boolean, optional With wxPython graphical toolkit, if True, tells the interpreter that there will be no server window.

If *wxnoserver* is set to True, the interpreter will not wait for the server GUI before showing the controller window.

max
int. Maximum value.

freq
float or PyoObject. Polling frequency.

Xnoise

class Xnoise(*dist=0*, *freq=1.0*, *x1=0.5*, *x2=0.5*, *mul=1*, *add=0*)
X-class pseudo-random generator.

Xnoise implements a few of the most common noise distributions. Each distribution generates values in the range 0 and 1.

Parent *PyoObject*

Args

dist: string or int, optional Distribution type. Defaults to 0.

freq: float or PyoObject, optional Polling frequency. Defaults to 1.

x1: float or PyoObject, optional First parameter. Defaults to 0.5.

x2: float or PyoObject, optional Second parameter. Defaults to 0.5.

Note:

Available distributions are:

0. uniform
1. linear minimum
2. linear maximum
3. triangular
4. exponential minimum
5. exponential maximum
6. double (bi)exponential
7. cauchy
8. weibull
9. gaussian
10. poisson
11. walker (drunk)
12. loopseg (drunk with looped segments)

Depending on the distribution, *x1* and *x2* parameters are applied as follow (names as string, or associated number can be used as *dist* parameter):

0. **uniform**
    - x1: not used
    - x2: not used
1. **linear_min**
    - x1: not used
    - x2: not used
2. **linear_max**
    - x1: not used
    - x2: not used
3. **triangle**
    - x1: not used
    - x2: not used
4. **expon_min**
    - x1: slope {0 = no slope -> 10 = sharp slope}
    - x2: not used
5. **expon_max**
    - x1: slope {0 = no slope -> 10 = sharp slope}

- x2: not used

6. **biexpon**
   - x1: bandwidth {0 = huge bandwidth -> 10 = narrow bandwidth}
   - x2: not used
7. **cauchy**
   - x1: bandwidth {0 = narrow bandwidth -> 10 = huge bandwidth}
   - x2: not used
8. **weibull**
   - x1: mean location {0 -> 1}
   - x2: shape {0.5 = linear min, 1.5 = expon min, 3.5 = gaussian}
9. **gaussian**
   - x1: mean location {0 -> 1}
   - x2: bandwidth {0 = narrow bandwidth -> 10 = huge bandwidth}
10. **poisson**
    - x1: gravity center {0 = low values -> 10 = high values}
    - x2: compress/expand range {0.1 = full compress -> 4 full expand}
11. **walker**
    - x1: maximum value {0.1 -> 1}
    - x2: maximum step {0.1 -> 1}
12. **loopseg**
    - x1: maximum value {0.1 -> 1}
    - x2: maximum step {0.1 -> 1}

```
>>> s = Server().boot()
>>> s.start()
>>> lfo = Phasor(.1, 0, .5, .15)
>>> freq = Xnoise(dist=12, freq=8, x1=1, x2=lfo, mul=1000, add=500)
>>> jit = Randi(min=0.99, max=1.01, freq=[2.33,3.41])
>>> a = SineLoop(freq*jit, feedback=0.03, mul=.2).out()
```

setDist (*x*)
Replace the *dist* attribute.

Args

x: string or int new *dist* attribute.

setX1 (*x*)
Replace the *x1* attribute.

Args

x: float or PyoObject new *x1* attribute.

setX2 (*x*)
Replace the *x2* attribute.

Args

x: float or PyoObject new *x2* attribute.

setFreq(*x*)
Replace the *freq* attribute.

Args

x: float or PyoObject new *freq* attribute.

ctrl(*map_list=None*, *title=None*, *wxnoserver=False*)
Opens a sliders window to control the parameters of the object. Only parameters that can be set to a PyoObject are allowed to be mapped on a slider.

If a list of values are given to a parameter, a multisliders will be used to control each stream independently.

Args

map_list: list of SLMap objects, optional Users defined set of parameters scaling. There is default scaling for each object that accept *ctrl* method.

title: string, optional Title of the window. If none is provided, the name of the class is used.

wxnoserver: boolean, optional With wxPython graphical toolkit, if True, tells the interpreter that there will be no server window.

If *wxnoserver* is set to True, the interpreter will not wait for the server GUI before showing the controller window.

dist
string or int. Distribution type.

freq
float or PyoObject. Polling frequency.

x1
float or PyoObject. First parameter.

x2
float or PyoObject. Second parameter.

XnoiseDur

class XnoiseDur(*dist=0*, *min=0.0*, *max=1.0*, *x1=0.5*, *x2=0.5*, *mul=1*, *add=0*)
Recursive time varying X-class pseudo-random generator.

Xnoise implements a few of the most common noise distributions. Each distribution generates values in the range 0 to 1, which are then rescaled between *min* and *max* arguments. The object uses the generated value to set the delay time before the next generation. XnoiseDur will hold the value until next generation.

Parent `PyoObject`

Args

dist: string or int, optional Distribution type. Can be the name of the distribution as a string or its associated number. Defaults to 0.

min: float or PyoObject, optional Minimum value for the random generation. Defaults to 0.

max: float or PyoObject, optional Maximum value for the random generation. Defaults to 1.

x1: float or PyoObject, optional First parameter. Defaults to 0.5.

x2: float or PyoObject, optional Second parameter. Defaults to 0.5.

Note:

Available distributions are:

0. uniform
1. linear minimum
2. linear maximum
3. triangular
4. exponential minimum
5. exponential maximum
6. double (bi)exponential
7. cauchy
8. weibull
9. gaussian
10. poisson
11. walker (drunk)
12. loopseg (drunk with looped segments)

Depending on the distribution, *x1* and *x2* parameters are applied as follow (names as string, or associated number can be used as *dist* parameter):

0. **uniform**
    - x1: not used
    - x2: not used
1. **linear_min**
    - x1: not used
    - x2: not used
2. **linear_max**
    - x1: not used
    - x2: not used
3. **triangle**
    - x1: not used
    - x2: not used
4. **expon_min**
    - x1: slope {0 = no slope -> 10 = sharp slope}
    - x2: not used

5. **expon_max**
    - x1: slope {0 = no slope -> 10 = sharp slope}
    - x2: not used
6. **biexpon**
    - x1: bandwidth {0 = huge bandwidth -> 10 = narrow bandwidth}
    - x2: not used
7. **cauchy**
    - x1: bandwidth {0 = narrow bandwidth -> 10 = huge bandwidth}
    - x2: not used
8. **weibull**
    - x1: mean location {0 -> 1}
    - x2: shape {0.5 = linear min, 1.5 = expon min, 3.5 = gaussian}
9. **gaussian**
    - x1: mean location {0 -> 1}
    - x2: bandwidth {0 = narrow bandwidth -> 10 = huge bandwidth}
10. **poisson**
    - x1: gravity center {0 = low values -> 10 = high values}
    - x2: compress/expand range {0.1 = full compress -> 4 full expand}
11. **walker**
    - x1: maximum value {0.1 -> 1}
    - x2: maximum step {0.1 -> 1}
12. **loopseg**
    - x1: maximum value {0.1 -> 1}
    - x2: maximum step {0.1 -> 1}

```
>>> s = Server().boot()
>>> s.start()
>>> dur = XnoiseDur(dist="expon_min", min=[.05,0.1], max=[.4,.5], x1=3)
>>> trig = Change(dur)
>>> amp = TrigEnv(trig, table=HannTable(), dur=dur, mul=.2)
>>> freqs = midiToHz([60,63,67,70,72])
>>> freq = TrigChoice(trig, choice=freqs)
>>> a = LFO(freq=freq, type=2, mul=amp).out()
```

setDist (*x*)

Replace the *dist* attribute.

Args

x: string or int new *dist* attribute.

setMin (*x*)

Replace the *min* attribute.

Args

x: float or PyoObject new *min* attribute.

setMax(*x*)
Replace the *max* attribute.

Args

x: float or PyoObject new *max* attribute.

setX1(*x*)
Replace the *x1* attribute.

Args

x: float or PyoObject new *x1* attribute.

setX2(*x*)
Replace the *x2* attribute.

Args

x: float or PyoObject new *x2* attribute.

ctrl(*map_list=None*, *title=None*, *wxnoserver=False*)
Opens a sliders window to control the parameters of the object. Only parameters that can be set to a PyoObject are allowed to be mapped on a slider.

If a list of values are given to a parameter, a multisliders will be used to control each stream independently.

Args

map_list: list of SLMap objects, optional Users defined set of parameters scaling. There is default scaling for each object that accept *ctrl* method.

title: string, optional Title of the window. If none is provided, the name of the class is used.

wxnoserver: boolean, optional With wxPython graphical toolkit, if True, tells the interpreter that there will be no server window.

If *wxnoserver* is set to True, the interpreter will not wait for the server GUI before showing the controller window.

dist
string or int. Distribution type.

min
float or PyoObject. Minimum value.

max
float or PyoObject. Maximum value.

x1
float or PyoObject. First parameter.

x2
float or PyoObject. Second parameter.

XnoiseMidi

class XnoiseMidi(*dist=0*, *freq=1.0*, *x1=0.5*, *x2=0.5*, *scale=0*, *mrange=(0, 127)*, *mul=1*, *add=0*)
X-class midi notes pseudo-random generator.

XnoiseMidi implements a few of the most common noise distributions. Each distribution generates integer values in the range defined with *mrange* parameter and output can be scaled on midi notes, hertz or transposition factor.

Parent `PyoObject`

Args

dist: string or int, optional Distribution type. Defaults to 0.

freq: float or PyoObject, optional Polling frequency. Defaults to 1.

x1: float or PyoObject, optional First parameter. Defaults to 0.5.

x2: float or PyoObject, optional Second parameter. Defaults to 0.5.

scale: int {0, 1, 2}, optional Output format. 0 = Midi, 1 = Hertz, 2 = transposition factor. In the transposition mode, the central key (the key where there is no transposition) is (*minrange* + *maxrange*) / 2. Defaults to 0.

mrange: tuple of int, optional Minimum and maximum possible values, in Midi notes. Available only at initialization time. Defaults to (0, 127).

Note:

Available distributions are:

0. uniform
1. linear minimum
2. linear maximum
3. triangular
4. exponential minimum
5. exponential maximum
6. double (bi)exponential
7. cauchy
8. weibull
9. gaussian
10. poisson
11. walker (drunk)
12. loopseg (drunk with looped segments)

Depending on the distribution, *x1* and *x2* parameters are applied as follow (names as string, or associated number can be used as *dist* parameter):

0. **uniform**
    - x1: not used
    - x2: not used
1. **linear_min**
    - x1: not used
    - x2: not used

2. **linear_max**
    - x1: not used
    - x2: not used
3. **triangle**
    - x1: not used
    - x2: not used
4. **expon_min**
    - x1: slope {0 = no slope -> 10 = sharp slope}
    - x2: not used
5. **expon_max**
    - x1: slope {0 = no slope -> 10 = sharp slope}
    - x2: not used
6. **biexpon**
    - x1: bandwidth {0 = huge bandwidth -> 10 = narrow bandwidth}
    - x2: not used
7. **cauchy**
    - x1: bandwidth {0 = narrow bandwidth -> 10 = huge bandwidth}
    - x2: not used
8. **weibull**
    - x1: mean location {0 -> 1}
    - x2: shape {0.5 = linear min, 1.5 = expon min, 3.5 = gaussian}
9. **gaussian**
    - x1: mean location {0 -> 1}
    - x2: bandwidth {0 = narrow bandwidth -> 10 = huge bandwidth}
10. **poisson**
    - x1: gravity center {0 = low values -> 10 = high values}
    - x2: compress/expand range {0.1 = full compress -> 4 full expand}
11. **walker**
    - x1: maximum value {0.1 -> 1}
    - x2: maximum step {0.1 -> 1}
12. **loopseg**
    - x1: maximum value {0.1 -> 1}
    - x2: maximum step {0.1 -> 1}

```
>>> s = Server().boot()
>>> s.start()
>>> l = Phasor(.4)
>>> rnd = XnoiseMidi('loopseg', freq=8, x1=1, x2=l, scale=0, mrange=(60,96))
>>> freq = Snap(rnd, choice=[0, 2, 3, 5, 7, 8, 11], scale=1)
>>> jit = Randi(min=0.99, max=1.01, freq=[2.33,3.41])
>>> a = SineLoop(freq*jit, feedback=0.03, mul=.2).out()
```

setDist (*x*)
Replace the *dist* attribute.

Args

x: string or int new *dist* attribute.

setScale (*x*)
Replace the *scale* attribute.

Possible values are:

0. Midi notes
1. Hertz
2. transposition factor (centralkey is (*minrange* + *maxrange*) / 2

Args

x: int {0, 1, 2} new *scale* attribute.

setRange (*mini*, *maxi*)
Replace the *mrange* attribute.

Args

mini: int minimum output midi range.

maxi: int maximum output midi range.

setX1 (*x*)
Replace the *x1* attribute.

Args

x: float or PyoObject new *x1* attribute.

setX2 (*x*)
Replace the *x2* attribute.

Args

x: float or PyoObject new *x2* attribute.

setFreq (*x*)
Replace the *freq* attribute.

Args

x: float or PyoObject new *freq* attribute.

ctrl (*map_list=None*, *title=None*, *wxnoserver=False*)
Opens a sliders window to control the parameters of the object. Only parameters that can be set to a PyoObject are allowed to be mapped on a slider.

If a list of values are given to a parameter, a multisliders will be used to control each stream independently.

Args

map_list: list of SLMap objects, optional Users defined set of parameters scaling. There is default scaling for each object that accept *ctrl* method.

title: string, optional Title of the window. If none is provided, the name of the class is used.

wxnoserver: boolean, optional With wxPython graphical toolkit, if True, tells the interpreter that there will be no server window.

If *wxnoserver* is set to True, the interpreter will not wait for the server GUI before showing the controller window.

dist
string or int. Distribution type.

freq
float or PyoObject. Polling frequency.

x1
float or PyoObject. First parameter.

x2
float or PyoObject. Second parameter.

scale
int. Output format.

Table Processing

Set of objects to perform operations on PyoTableObjects.

PyoTableObjects are 1 dimension containers. They can be used to store audio samples or algorithmic sequences for future uses.

Granulator

class Granulator(*table*, *env*, *pitch=1*, *pos=0*, *dur=0.1*, *grains=8*, *basedur=0.1*, *mul=1*, *add=0*)
Granular synthesis generator.

Parent *PyoObject*

Args

table: PyoTableObject Table containing the waveform samples.

env: PyoTableObject Table containing the grain envelope.

pitch: float or PyoObject, optional Overall pitch of the granulator. This value transpose the pitch of all grains. Defaults to 1.

pos: float or PyoObject, optional Pointer position, in samples, in the waveform table. Each grain sampled the current value of this stream at the beginning of its envelope and hold it until the end of the grain. Defaults to 0.

dur: float or PyoObject, optional Duration, in seconds, of the grain. Each grain sampled the current value of this stream at the beginning of its envelope and hold it until the end of the grain. Defaults to 0.1.

grains: int, optional Number of grains. Defaults to 8.

basedur: float, optional Base duration used to calculate the speed of the pointer to read the grain at its original pitch. By changing the value of the *dur* parameter, transposition per grain can be generated. Defaults to 0.1.

```
>>> s = Server().boot()
>>> s.start()
>>> snd = SndTable(SNDS_PATH + "/transparent.aif")
>>> env = HannTable()
>>> pos = Phasor(snd.getRate()*.25, 0, snd.getSize())
>>> dur = Noise(.001, .1)
>>> g = Granulator(snd, env, [1, 1.001], pos, dur, 24, mul=.1).out()
```

setTable(*x*)
Replace the *table* attribute.

Args

x: PyoTableObject new *table* attribute.

setEnv(*x*)
Replace the *env* attribute.

Args

x: PyoTableObject new *env* attribute.

setPitch(*x*)
Replace the *pitch* attribute.

Args

x: float or PyoObject new *pitch* attribute.

setPos(*x*)
Replace the *pos* attribute.

Args

x: float or PyoObject new *pos* attribute.

setDur(*x*)
Replace the *dur* attribute.

Args

x: float or PyoObject new *dur* attribute.

setGrains(*x*)
Replace the *grains* attribute.

Args

x: int new *grains* attribute.

setBaseDur(*x*)
Replace the *basedur* attribute.

Args

x: float new *basedur* attribute.

ctrl(*map_list=None*, *title=None*, *wxnoserver=False*)
Opens a sliders window to control the parameters of the object. Only parameters that can be set to a PyoObject are allowed to be mapped on a slider.

If a list of values are given to a parameter, a multisliders will be used to control each stream independently.

Args

map_list: list of SLMap objects, optional Users defined set of parameters scaling. There is default scaling for each object that accept *ctrl* method.

title: string, optional Title of the window. If none is provided, the name of the class is used.

wxnoserver: boolean, optional With wxPython graphical toolkit, if True, tells the interpreter that there will be no server window.

If *wxnoserver* is set to True, the interpreter will not wait for the server GUI before showing the controller window.

table
PyoTableObject. Table containing the waveform samples.

env
PyoTableObject. Table containing the grain envelope.

pitch
float or PyoObject. Overall pitch of the granulator.

pos
float or PyoObject. Position of the pointer in the sound table.

dur
float or PyoObject. Duration, in seconds, of the grain.

grains
int. Number of grains.

basedur
float. Duration to read the grain at its original pitch.

Granule

class Granule(*table*, *env*, *dens=50*, *pitch=1*, *pos=0*, *dur=0.1*, *mul=1*, *add=0*)
Another granular synthesis generator.

As of pyo 0.7.4, users can call the *setSync* method to change the granulation mode (either synchronous or asynchronous) of the object.

Parent `PyoObject`

Args

table: PyoTableObject Table containing the waveform samples.

env: PyoTableObject Table containing the grain envelope.

dens: float or PyoObject, optional Density of grains per second. Defaults to 50.

pitch: float or PyoObject, optional Pitch of the grains. A new grain gets the current value of *pitch* as its reading speed. Defaults to 1.

pos: float or PyoObject, optional Pointer position, in samples, in the waveform table. Each grain samples the current value of this stream at the beginning of its envelope and holds it until the end of the grain. Defaults to 0.

dur: float or PyoObject, optional Duration, in seconds, of the grain. Each grain samples the current value of this stream at the beginning of its envelope and hold it until the end of the grain. Defaults to 0.1.

```
>>> s = Server().boot()
>>> s.start()
>>> snd = SndTable(SNDS_PATH+"/transparent.aif")
>>> end = snd.getSize() - s.getSamplingRate() * 0.2
>>> env = HannTable()
>>> pos = Randi(min=0, max=1, freq=[0.25, 0.3], mul=end)
>>> dns = Randi(min=10, max=30, freq=.1)
>>> pit = Randi(min=0.99, max=1.01, freq=100)
>>> grn = Granule(snd, env, dens=dns, pitch=pit, pos=pos, dur=.2, mul=.1).out()
```

setTable(*x*)
Replace the *table* attribute.

Args

x: PyoTableObject new *table* attribute.

setEnv(*x*)
Replace the *env* attribute.

Args

x: PyoTableObject new *env* attribute.

setDens(*x*)
Replace the *dens* attribute.

Args

x: float or PyoObject new *dens* attribute.

setPitch(*x*)
Replace the *pitch* attribute.

Args

x: float or PyoObject new *pitch* attribute.

setPos(*x*)
Replace the *pos* attribute.

Args

x: float or PyoObject new *pos* attribute.

setDur(*x*)
Replace the *dur* attribute.

Args

x: float or PyoObject new *dur* attribute.

setSync(*x*)
Sets the granulation mode, synchronous or asynchronous.

Args

x: boolean True means synchronous granulation (the default) while False means asynchronous.

ctrl(*map_list=None*, *title=None*, *wxnoserver=False*)
Opens a sliders window to control the parameters of the object. Only parameters that can be set to a PyoObject are allowed to be mapped on a slider.

If a list of values are given to a parameter, a multisliders will be used to control each stream independently.

Args

map_list: list of SLMap objects, optional Users defined set of parameters scaling. There is default scaling for each object that accept *ctrl* method.

title: string, optional Title of the window. If none is provided, the name of the class is used.

wxnoserver: boolean, optional With wxPython graphical toolkit, if True, tells the interpreter that there will be no server window.

If *wxnoserver* is set to True, the interpreter will not wait for the server GUI before showing the controller window.

table
PyoTableObject. Table containing the waveform samples.

env
PyoTableObject. Table containing the grain envelope.

dens
float or PyoObject. Density of grains per second.

pitch
float or PyoObject. Transposition factor of the grain.

pos
float or PyoObject. Position of the pointer in the sound table.

dur
float or PyoObject. Duration, in seconds, of the grain.

Lookup

class Lookup(*table*, *index*, *mul=1*, *add=0*)
Uses table to do waveshaping on an audio signal.

Lookup uses a table to apply waveshaping on an input signal *index*. The index must be between -1 and 1, it is automatically scaled between 0 and len(table)-1 and is used as a position pointer in the table.

Parent *PyoObject*

Args

table: PyoTableObject Table containing the transfert function.

index: PyoObject Audio signal, between -1 and 1, internally converted to be used as the index position in the table.

```
>>> s = Server().boot()
>>> s.start()
>>> lfo = Sine(freq=[.15,.2], mul=.2, add=.25)
>>> a = Sine(freq=[100,150], mul=lfo)
>>> t = CosTable([(0,-1),(3072,-0.85),(4096,0),(5520,.85),(8192,1)])
>>> b = Lookup(table=t, index=a, mul=.5-lfo).out()
```

setTable(*x*)
Replace the *table* attribute.

Args

x: PyoTableObject new *table* attribute.

setIndex(*x*)
Replace the *index* attribute.

Args

x: PyoObject new *index* attribute.

ctrl(*map_list=None*, *title=None*, *wxnoserver=False*)
Opens a sliders window to control the parameters of the object. Only parameters that can be set to a PyoObject are allowed to be mapped on a slider.

If a list of values are given to a parameter, a multisliders will be used to control each stream independently.

Args

map_list: list of SLMap objects, optional Users defined set of parameters scaling. There is default scaling for each object that accept *ctrl* method.

title: string, optional Title of the window. If none is provided, the name of the class is used.

wxnoserver: boolean, optional With wxPython graphical toolkit, if True, tells the interpreter that there will be no server window.

If *wxnoserver* is set to True, the interpreter will not wait for the server GUI before showing the controller window.

table
PyoTableObject. Table containing the transfert function.

index
PyoObject. Audio input used as the table index.

Looper

class Looper(*table*, *pitch=1*, *start=0*, *dur=1.0*, *xfade=20*, *mode=1*, *xfadeshape=0*, *startfromloop=False*, *interp=2*, *autosmooth=False*, *mul=1*, *add=0*)
Crossfading looper.

Looper reads audio from a PyoTableObject and plays it back in a loop with user-defined pitch, start time, duration and crossfade time. The *mode* argument allows the user to choose different looping modes.

Parent *PyoObject*

Args

table: PyoTableObject Table containing the waveform samples.

pitch: float or PyoObject, optional Transposition factor. 1 is normal pitch, 0.5 is one octave lower, 2 is one octave higher. Negative values are not allowed. Defaults to 1.

start: float or PyoObject, optional Starting point, in seconds, of the loop, updated only once per loop cycle. Defaults to 0.

dur: float or PyoObject, optional Duration, in seconds, of the loop, updated only once per loop cycle. Defaults to 1.

xfade: float or PyoObject {0 -> 50}, optional Percent of the loop time used to crossfade readers, updated only once per loop cycle and clipped between 0 and 50. Defaults to 20.

mode: int {0, 1, 2, 3}, optional

Loop modes. Defaults to 1.

0. no loop
1. forward
2. backward
3. back-and-forth

xfadeshape: int {0, 1, 2}, optional

Crossfade envelope shape. Defaults to 0.

0. linear
1. equal power
2. sigmoid

startfromloop: boolean, optional If True, reading will begin directly at the loop start point. Otherwise, it begins at the beginning of the table. Defaults to False.

interp: int {1, 2, 3, 4}, optional

Choice of the interpolation method. Defaults to 2.

1. no interpolation
2. linear
3. cosinus
4. cubic

autosmooth: boolean, optional If True, a lowpass filter, following the pitch, is applied on the output signal to reduce the quantization noise produced by very low transpositions. Defaults to False.

Note: Looper will send a trigger signal every new playback starting point (i.e. when the object is activated and at the beginning of the crossfade of a loop. User can retrieve the trigger stream by calling obj['trig'].

Looper also outputs a time stream, given the current position of the reading pointer, normalized between 0.0 and 1.0 (1.0 means the beginning of the crossfade), inside the loop. User can retrieve the time stream by calling obj['time']. New in version 0.8.1.

See also:

Granulator, *Pointer*

```
>>> s = Server().boot()
>>> s.start()
>>> tab = SndTable(SNDS_PATH + '/transparent.aif')
>>> pit = Choice(choice=[.5,.75,1,1.25,1.5], freq=[3,4])
>>> start = Phasor(freq=.2, mul=tab.getDur())
>>> dur = Choice(choice=[.0625,.125,.125,.25,.33], freq=4)
>>> a = Looper(table=tab, pitch=pit, start=start, dur=dur, startfromloop=True,␣
↪mul=.25).out()
```

stop(*wait=0*)

Stop processing.

This method returns *self*, allowing it to be applied at the object creation.

Args

wait: float, optional Delay, in seconds, before the process is actually stopped. If autoStartChildren is activated in the Server, this value is propagated to the children objects. Defaults to 0.

Note: if the method setStopDelay(x) was called before calling stop(wait) with a positive *wait* value, the *wait* value won't overwrite the value given to setStopDelay for the current object, but will be the one propagated to children objects. This allow to set a waiting time for a specific object with setStopDelay whithout changing the global delay time given to the stop method.

Fader and Adsr objects ignore waiting time given to the stop method because they already implement a delayed processing triggered by the stop call.

---

setTable(*x*)
Replace the *table* attribute.

Args

x: PyoTableObject new *table* attribute.

setPitch(*x*)
Replace the *pitch* attribute.

Args

x: float or PyoObject new *pitch* attribute.

setStart(*x*)
Replace the *start* attribute.

Args

x: float or PyoObject new *start* attribute.

setDur(*x*)
Replace the *dur* attribute.

Args

x: float or PyoObject new *dur* attribute.

setXfade(*x*)
Replace the *xfade* attribute.

Args

x: float or PyoObject new *xfade* attribute.

setXfadeShape(*x*)
Replace the *xfadeshape* attribute.

Args

x: int new *xfadeshape* attribute.

setStartFromLoop(*x*)
Replace the *startfromloop* attribute.

Args

x: boolean new *startfromloop* attribute.

setMode(*x*)
Replace the *mode* attribute.

Args

x: int new *mode* attribute.

setInterp(*x*)
Replace the *interp* attribute.

Args

x: int new *interp* attribute.

setAutoSmooth(*x*)
Replace the *autosmooth* attribute.

Args

x: boolean new *autosmooth* attribute.

reset()
Resets internal reading pointers to 0.

loopnow()
Sarts a new loop immediately.

appendFadeTime(*x*)
Change the position of the crossfade regarding loop duration.

Args

x: boolean If 0 (the default), the crossfade duration lies inside the loop duration, producing a loop that is shorter than the *dur* value.

If 1, the crossfade starts after the loop duration, expecting samples after the loop to perform the fadeout. This mode gives a loop of a length of the *dur* value.

fadeInSeconds(*x*)
Change the crossfade time unit (*xfade* unit type).

Args

x: boolean If 0 (the default), the crossfade duration (*xfade* value) is a percent of the loop time, between 0 and 50.

If 1, the crossfade duration (*xfade* value) is set in seconds, between 0 and half the loop length.

ctrl(*map_list=None*, *title=None*, *wxnoserver=False*)
Opens a sliders window to control the parameters of the object. Only parameters that can be set to a PyoObject are allowed to be mapped on a slider.

If a list of values are given to a parameter, a multisliders will be used to control each stream independently.

Args

map_list: list of SLMap objects, optional Users defined set of parameters scaling. There is default scaling for each object that accept *ctrl* method.

title: string, optional Title of the window. If none is provided, the name of the class is used.

wxnoserver: boolean, optional With wxPython graphical toolkit, if True, tells the interpreter that there will be no server window.

If *wxnoserver* is set to True, the interpreter will not wait for the server GUI before showing the controller window.

table
PyoTableObject. Table containing the waveform samples.

pitch
float or PyoObject. Transposition factor.

start
float or PyoObject. Loop start position in seconds.

dur
float or PyoObject. Loop duration in seconds.

xfade
float or PyoObject. Crossfade duration in percent.

xfadeshape
int. Crossfade envelope.

startfromloop
boolean. Starts from loop point if True, otherwise starts from beginning of the sound.

mode
int. Looping mode.

interp
int. Interpolation method.

autosmooth
boolean. Activates a lowpass filter applied on output signal.

Osc

class Osc(*table*, *freq=1000*, *phase=0*, *interp=2*, *mul=1*, *add=0*)
A simple oscillator reading a waveform table.

Parent *PyoObject*

Args

table: PyoTableObject Table containing the waveform samples.

freq: float or PyoObject, optional Frequency in cycles per second. Defaults to 1000.

phase: float or PyoObject, optional Phase of sampling, expressed as a fraction of a cycle (0 to 1). Defaults to 0.

interp: int, optional

Choice of the interpolation method. Defaults to 2.

1. no interpolation
2. linear
3. cosinus
4. cubic

See also:

Phasor, *Sine*

```
>>> s = Server().boot()
>>> s.start()
>>> t = HarmTable([1,0,.33,0,.2,0,.143,0,.111,0,.091])
>>> a = Osc(table=t, freq=[100,99.2], mul=.2).out()
```

setTable(*x*)
Replace the *table* attribute.

Args

x: PyoTableObject new *table* attribute.

setFreq(*x*)
Replace the *freq* attribute.

Args

x: float or PyoObject new *freq* attribute.

setPhase(*x*)
Replace the *phase* attribute.

Args

x: float or PyoObject new *phase* attribute.

setInterp(*x*)
Replace the *interp* attribute.

Args

x: int {1, 2, 3, 4} new *interp* attribute.

reset()
Resets current phase to 0.

ctrl(*map_list=None*, *title=None*, *wxnoserver=False*)
Opens a sliders window to control the parameters of the object. Only parameters that can be set to a PyoObject are allowed to be mapped on a slider.

If a list of values are given to a parameter, a multisliders will be used to control each stream independently.

Args

map_list: list of SLMap objects, optional Users defined set of parameters scaling. There is default scaling for each object that accept *ctrl* method.

title: string, optional Title of the window. If none is provided, the name of the class is used.

wxnoserver: boolean, optional With wxPython graphical toolkit, if True, tells the interpreter that there will be no server window.

If *wxnoserver* is set to True, the interpreter will not wait for the server GUI before showing the controller window.

table
PyoTableObject. Table containing the waveform samples.

freq
float or PyoObject. Frequency in cycles per second.

phase
float or PyoObject. Phase of sampling.

interp
int {1, 2, 3, 4}. Interpolation method.

OscBank

class OscBank(*table*, *freq=100*, *spread=1*, *slope=0.9*, *frndf=1*, *frnda=0*, *arndf=1*, *arnda=0*, *num=24*, *fjit=False*, *mul=1*, *add=0*)
Any number of oscillators reading a waveform table.

OscBank mixes the output of any number of oscillators. The frequencies of each oscillator is controlled with two parameters, the base frequency *freq* and a coefficient of expansion *spread*. Frequencies are computed with the following formula (*n* is the order of the partial):

f_n = freq + freq * spread * n

The frequencies and amplitudes can be modulated by two random generators with interpolation (each partial have a different set of randoms).

Parent *PyoObject*

Args

table: PyoTableObject Table containing the waveform samples.

freq: float or PyoObject, optional Base frequency in cycles per second. Defaults to 100.

spread: float or PyoObject, optional Coefficient of expansion used to compute partial frequencies. If *spread* is 0, all partials will be at the base frequency. A value of 1 will generate integer harmonics, a value of 2 will skip even harmonics and non-integer values will generate different series of inharmonic frequencies. Defaults to 1.

slope: float or PyoObject, optional specifies the multiplier in the series of amplitude coefficients. This is a power series: the nth partial will have an amplitude of (slope ** n), i.e. strength values trace an exponential curve. Defaults to 1.

frndf: float or PyoObject, optional Frequency, in cycle per second, of the frequency modulations. Defaults to 1.

frnda: float or PyoObject, optional Maximum frequency deviation (positive and negative) in portion of the partial frequency. A value of 1 means that the frequency can drift from 0 Hz to twice the partial frequency. A value of 0 deactivates the frequency deviations. Defaults to 0.

arndf: float or PyoObject, optional Frequency, in cycle per second, of the amplitude modulations. Defaults to 1.

arnda: float or PyoObject, optional Amount of amplitude deviation. 0 deactivates the amplitude modulations and 1 gives full amplitude modulations. Defaults to 0.

num: int, optional Number of oscillators. Available at initialization only. Defaults to 24.

fjit: boolean, optional If True, a small jitter is added to the frequency of each partial. For a large number of oscillators and a very small *spread*, the periodicity between partial frequencies can cause very strange artefact. Adding a jitter breaks the periodicity. Defaults to False.

See also:

Osc

Note: Altough parameters can be audio signals, values are sampled only once per buffer size. To avoid artefacts, it is recommended to keep variations at low rate (< 20 Hz).

```
>>> s = Server().boot()
>>> s.start()
>>> ta = HarmTable([1,.3,.2])
>>> tb = HarmTable([1])
>>> f = Fader(fadein=.1).play()
>>> a = OscBank(ta,100,spread=0,frndf=.25,frnda=.01,num=[10,10],fjit=True,mul=f*0.
↪5).out()
>>> b = OscBank(tb,250,spread=.25,slope=.8,arndf=4,arnda=1,num=[10,10],mul=f*0.4).
↪out()
```

setTable(*x*)
Replace the *table* attribute.

Args

x: PyoTableObject new *table* attribute.

setFreq(*x*)
Replace the *freq* attribute.

Args

x: float or PyoObject new *freq* attribute.

setSpread(*x*)
Replace the *spread* attribute.

Args

x: float or PyoObject new *spread* attribute.

setSlope(*x*)
Replace the *slope* attribute.

Args

x: float or PyoObject new *slope* attribute.

setFrndf(*x*)
Replace the *frndf* attribute.

Args

x: float or PyoObject new *frndf* attribute.

setFrnda(*x*)
Replace the *frnda* attribute.

Args

x: float or PyoObject new *frnda* attribute.

setArndf(*x*)
Replace the *arndf* attribute.

Args

x: float or PyoObject new *arndf* attribute.

setArnda(*x*)
Replace the *arnda* attribute.

Args

x: float or PyoObject new *arnda* attribute.

setFjit(*x*)
Replace the *fjit* attribute.

Args

x: boolean new *fjit* attribute.

ctrl(*map_list=None*, *title=None*, *wxnoserver=False*)
Opens a sliders window to control the parameters of the object. Only parameters that can be set to a PyoObject are allowed to be mapped on a slider.

If a list of values are given to a parameter, a multisliders will be used to control each stream independently.

Args

map_list: list of SLMap objects, optional Users defined set of parameters scaling. There is default scaling for each object that accept *ctrl* method.

title: string, optional Title of the window. If none is provided, the name of the class is used.

wxnoserver: boolean, optional With wxPython graphical toolkit, if True, tells the interpreter that there will be no server window.

If *wxnoserver* is set to True, the interpreter will not wait for the server GUI before showing the controller window.

table
PyoTableObject. Table containing the waveform samples.

freq
float or PyoObject. Frequency in cycles per second.

spread
float or PyoObject. Frequency expansion factor.

slope
float or PyoObject. Multiplier in the series of amplitudes.

frndf
float or PyoObject. Frequency of the frequency modulations.

frnda
float or PyoObject. Maximum frequency deviation from 0 to 1.

arndf
float or PyoObject. Frequency of the amplitude modulations.

arnda
float or PyoObject. Amount of amplitude deviation from 0 to 1.

fjit
boolean. Jitter added to the parial frequencies.

OscLoop

class OscLoop(*table*, *freq=1000*, *feedback=0*, *mul=1*, *add=0*)

A simple oscillator with feedback reading a waveform table.

OscLoop reads a waveform table with linear interpolation and feedback control. The oscillator output, multiplied by *feedback*, is added to the position increment and can be used to control the brightness of the oscillator.

Parent *PyoObject*

Args

table: PyoTableObject Table containing the waveform samples.

freq: float or PyoObject, optional Frequency in cycles per second. Defaults to 1000.

feedback: float or PyoObject, optional Amount of the output signal added to position increment, between 0 and 1. Controls the brightness. Defaults to 0.

See also:

Osc, *SineLoop*

```
>>> s = Server().boot()
>>> s.start()
>>> t = HarmTable([1,0,.33,0,.2,0,.143])
>>> lfo = Sine(.5, 0, .05, .05)
>>> a = OscLoop(table=t, freq=[100,99.3], feedback=lfo, mul=.2).out()
```

setTable(*x*)

Replace the *table* attribute.

Args

x: PyoTableObject new *table* attribute.

setFreq(*x*)

Replace the *freq* attribute.

Args

x: float or PyoObject new *freq* attribute.

setFeedback(*x*)

Replace the *feedback* attribute.

Args

x: float or PyoObject new *feedback* attribute.

ctrl(*map_list=None*, *title=None*, *wxnoserver=False*)

Opens a sliders window to control the parameters of the object. Only parameters that can be set to a PyoObject are allowed to be mapped on a slider.

If a list of values are given to a parameter, a multisliders will be used to control each stream independently.

Args

map_list: list of SLMap objects, optional Users defined set of parameters scaling. There is default scaling for each object that accept *ctrl* method.

title: string, optional Title of the window. If none is provided, the name of the class is used.

wxnoserver: boolean, optional With wxPython graphical toolkit, if True, tells the interpreter that there will be no server window.

If *wxnoserver* is set to True, the interpreter will not wait for the server GUI before showing the controller window.

table
PyoTableObject. Table containing the waveform samples.

freq
float or PyoObject. Frequency in cycles per second.

feedback
float or PyoObject. Brightness control.

OscTrig

class OscTrig(*table*, *trig*, *freq=1000*, *phase=0*, *interp=2*, *mul=1*, *add=0*)
An oscillator reading a waveform table with sample accurate reset signal.

Parent *PyoObject*

Args

table: PyoTableObject Table containing the waveform samples.

trig: PyoObject Trigger signal. Reset the table pointer position to zero on each trig.

freq: float or PyoObject, optional Frequency in cycles per second. Defaults to 1000.

phase: float or PyoObject, optional Phase of sampling, expressed as a fraction of a cycle (0 to 1). Defaults to 0.

interp: int, optional

Choice of the interpolation method. Defaults to 2.

1. no interpolation
2. linear
3. cosinus
4. cubic

See also:

Osc, *Phasor*, *Sine*

```
>>> s = Server().boot()
>>> s.start()
>>> tab = SndTable(SNDS_PATH+"/transparent.aif")
>>> tim = Phasor([-0.2,-0.25], mul=tab.getDur()-0.005, add=0.005)
>>> rst = Metro(tim).play()
>>> a = OscTrig(tab, rst, freq=tab.getRate(), mul=.4).out()
```

setTable(*x*)
Replace the *table* attribute.

Args

x: PyoTableObject new *table* attribute.

setTrig(*x*)
Replace the *trig* attribute.

Args

x: PyoObject new *trig* attribute.

setFreq(*x*)
Replace the *freq* attribute.

Args

x: float or PyoObject new *freq* attribute.

setPhase(*x*)
Replace the *phase* attribute.

Args

x: float or PyoObject new *phase* attribute.

setInterp(*x*)
Replace the *interp* attribute.

Args

x: int {1, 2, 3, 4} new *interp* attribute.

reset()
Resets current phase to 0.

ctrl(*map_list=None*, *title=None*, *wxnoserver=False*)
Opens a sliders window to control the parameters of the object. Only parameters that can be set to a PyoObject are allowed to be mapped on a slider.

If a list of values are given to a parameter, a multisliders will be used to control each stream independently.

Args

map_list: list of SLMap objects, optional Users defined set of parameters scaling. There is default scaling for each object that accept *ctrl* method.

title: string, optional Title of the window. If none is provided, the name of the class is used.

wxnoserver: boolean, optional With wxPython graphical toolkit, if True, tells the interpreter that there will be no server window.

If *wxnoserver* is set to True, the interpreter will not wait for the server GUI before showing the controller window.

table
PyoTableObject. Table containing the waveform samples.

trig
PyoObject. Trigger signal. Reset pointer position to zero

freq
float or PyoObject. Frequency in cycles per second.

phase
float or PyoObject. Phase of sampling.

interp
int {1, 2, 3, 4}. Interpolation method.

Particle

class Particle(*table*, *env*, *dens=50*, *pitch=1*, *pos=0*, *dur=0.1*, *dev=0.01*, *pan=0.5*, *chnls=1*, *mul=1*, *add=0*)

A full control granular synthesis generator.

Parent *PyoObject*

Args

table: PyoTableObject Table containing the waveform samples.

env: PyoTableObject Table containing the grain envelope.

dens: float or PyoObject, optional Density of grains per second. Defaults to 50.

pitch: float or PyoObject, optional Pitch of the grains. Each grain samples the current value of this stream at the beginning of its envelope and holds it until the end of the grain. Defaults to 1.

pos: float or PyoObject, optional Pointer position, in samples, in the waveform table. Each grain sampled the current value of this stream at the beginning of its envelope and holds it until the end of the grain. Defaults to 0.

dur: float or PyoObject, optional Duration, in seconds, of the grain. Each grain samples the current value of this stream at the beginning of its envelope and holds it until the end of the grain. Defaults to 0.1.

dev: float or PyoObject, optional Maximum deviation of the starting time of the grain, between 0 and 1 (relative to the current duration of the grain). Each grain samples the current value of this stream at the beginning of its envelope and holds it until the end of the grain. Defaults to 0.01.

pan: float or PyoObject, optional Panning factor of the grain (if chnls=1, this value is skipped). Each grain samples the current value of this stream at the beginning of its envelope and holds it until the end of the grain. Defaults to 0.5.

chnls: integer, optional Number of output channels per audio stream (if chnls=2 and a stereo sound table is given at the table argument, the objet will create 4 output streams, 2 per table channel). Available at initialization only. Defaults to 1.

Note: Particle object compensate for the difference between sampling rate of the loaded sound and the current sampling rate of the Server.

```
>>> s = Server().boot()
>>> s.start()
>>> snd = SndTable(SNDS_PATH+"/transparent.aif")
>>> end = snd.getSize() - s.getSamplingRate() * 0.25
>>> env = HannTable()
>>> dns = Randi(min=5, max=100, freq=.1)
>>> pit = Randh(min=0.99, max=1.01, freq=100)
>>> pos = Randi(min=0, max=1, freq=0.25, mul=end)
>>> dur = Randi(min=0.01, max=0.25, freq=0.15)
>>> dev = Randi(min=0, max=1, freq=0.2)
>>> pan = Noise(0.5, 0.5)
>>> grn = Particle(snd, env, dns, pit, pos, dur, dev, pan, chnls=2, mul=.2).out()
```

setTable(*x*)

Replace the *table* attribute.

Args

x: PyoTableObject new *table* attribute.

setEnv(*x*)
Replace the *env* attribute.

Args

x: PyoTableObject new *env* attribute.

setDens(*x*)
Replace the *dens* attribute.

Args

x: float or PyoObject new *dens* attribute.

setPitch(*x*)
Replace the *pitch* attribute.

Args

x: float or PyoObject new *pitch* attribute.

setPos(*x*)
Replace the *pos* attribute.

Args

x: float or PyoObject new *pos* attribute.

setDur(*x*)
Replace the *dur* attribute.

Args

x: float or PyoObject new *dur* attribute.

setDev(*x*)
Replace the *dev* attribute.

Args

x: float or PyoObject new *dev* attribute.

setPan(*x*)
Replace the *pan* attribute.

Args

x: float or PyoObject new *pan* attribute.

ctrl(*map_list=None*, *title=None*, *wxnoserver=False*)
Opens a sliders window to control the parameters of the object. Only parameters that can be set to a PyoObject are allowed to be mapped on a slider.

If a list of values are given to a parameter, a multisliders will be used to control each stream independently.

Args

map_list: list of SLMap objects, optional Users defined set of parameters scaling. There is default scaling for each object that accept *ctrl* method.

title: string, optional Title of the window. If none is provided, the name of the class is used.

wxnoserver: boolean, optional With wxPython graphical toolkit, if True, tells the interpreter that there will be no server window.

If *wxnoserver* is set to True, the interpreter will not wait for the server GUI before showing the controller window.

table
PyoTableObject. Table containing the waveform samples.

env
PyoTableObject. Table containing the grain envelope.

dens
float or PyoObject. Density of grains per second.

pitch
float or PyoObject. Transposition factor of the grain.

pos
float or PyoObject. Position of the pointer in the sound table.

dur
float or PyoObject. Duration, in seconds, of the grain.

dev
float or PyoObject. Deviation of the starting point of the grain in the table.

pan
float or PyoObject.Panning of the grain.

Particle2

class Particle2(*table*, *env*, *dens=50*, *pitch=1*, *pos=0*, *dur=0.1*, *dev=0.01*, *pan=0.5*, *filterfreq=18000*, *filterq=0.7*, *filtertype=0*, *chnls=1*, *mul=1*, *add=0*)

An even more full control granular synthesis generator.

This granulator object offers all the same controls as the Particle object with additionally an independently controllable filter per grain. The filters use the same implementation as the Biquad object.

Parent *PyoObject*

Args

table: PyoTableObject Table containing the waveform samples.

env: PyoTableObject Table containing the grain envelope.

dens: float or PyoObject, optional Density of grains per second. Defaults to 50.

pitch: float or PyoObject, optional Pitch of the grains. Each grain samples the current value of this stream at the beginning of its envelope and holds it until the end of the grain. Defaults to 1.

pos: float or PyoObject, optional Pointer position, in samples, in the waveform table. Each grain samples the current value of this stream at the beginning of its envelope and holds it until the end of the grain. Defaults to 0.

dur: float or PyoObject, optional Duration, in seconds, of the grain. Each grain samples the current value of this stream at the beginning of its envelope and holds it until the end of the grain. Defaults to 0.1.

dev: float or PyoObject, optional Maximum deviation of the starting time of the grain, between 0 and 1 (relative to the current duration of the grain). Each grain samples the current value of this stream at the beginning of its envelope and holds it until the end of the grain. Defaults to 0.01.

pan: float or PyoObject, optional Panning factor of the grain (if chnls=1, this value is skipped). Each grain samples the current value of this stream at the beginning of its envelope and holds it until the end of the grain. Defaults to 0.5.

filterfreq: float or PyoObject, optional Center or cutoff frequency of the grain filter. Each grain samples the current value of this stream at the beginning of its envelope and hold it until the end of the grain. Defaults to 18000.

filterq: float or PyoObject, optional Q of the grain filter. Each grain samples the current value of this stream at the beginning of its envelope and hold it until the end of the grain. Defaults to 0.7.

filtertype: float or PyoObject, optional Type of the grain filter. Each grain samples the current value of this stream at the beginning of its envelope and hold it until the end of the grain. Thw value is rounded to the nearest integer. Possible values are:

0. lowpass (default)
1. highpass
2. bandpass
3. bandstop
4. allpass

chnls: integer, optional Number of output channels per audio stream (if chnls=2 and a stereo sound table is given at the table argument, the objet will create 4 output streams, 2 per table channel). Available at initialization only. Defaults to 1.

Note: Particle object compensate for the difference between sampling rate of the loaded sound and the current sampling rate of the Server.

```
>>> s = Server().boot()
>>> s.start()
>>> snd = SndTable(SNDS_PATH+"/transparent.aif")
>>> end = snd.getSize() - s.getSamplingRate() * 0.25
>>> env = HannTable()
>>> dns = Randi(min=8, max=24, freq=.1)
>>> pit = Randh(min=0.49, max=0.51, freq=100)
>>> pos = Sine(0.05).range(0, end)
>>> dur = Randi(min=0.05, max=0.15, freq=0.15)
>>> dev = 0.001
>>> pan = Noise(0.5, 0.5)
>>> cf = Sine(0.07).range(75, 125)
>>> fcf = Choice(list(range(1, 40)), freq=150, mul=cf)
>>> grn = Particle2(snd, env, dns, pit, pos, dur, dev, pan, fcf, 20, 2, chnls=2,␣
↪mul=.3)
>>> grn.out()
```

setTable(*x*)

Replace the *table* attribute.

Args

x: PyoTableObject new *table* attribute.

setEnv(*x*)
Replace the *env* attribute.

Args

x: PyoTableObject new *env* attribute.

setDens(*x*)
Replace the *dens* attribute.

Args

x: float or PyoObject new *dens* attribute.

setPitch(*x*)
Replace the *pitch* attribute.

Args

x: float or PyoObject new *pitch* attribute.

setPos(*x*)
Replace the *pos* attribute.

Args

x: float or PyoObject new *pos* attribute.

setDur(*x*)
Replace the *dur* attribute.

Args

x: float or PyoObject new *dur* attribute.

setDev(*x*)
Replace the *dev* attribute.

Args

x: float or PyoObject new *dev* attribute.

setPan(*x*)
Replace the *pan* attribute.

Args

x: float or PyoObject new *pan* attribute.

setFilterfreq(*x*)
Replace the *filterfreq* attribute.

Args

x: float or PyoObject new *filterfreq* attribute.

setFilterq(*x*)
Replace the *filterq* attribute.

Args

x: float or PyoObject new *filterq* attribute.

setFiltertype(*x*)
Replace the *filtertype* attribute.

Args

x: float or PyoObject new *filtertype* attribute.

ctrl(*map_list=None*, *title=None*, *wxnoserver=False*)
Opens a sliders window to control the parameters of the object. Only parameters that can be set to a PyoObject are allowed to be mapped on a slider.

If a list of values are given to a parameter, a multisliders will be used to control each stream independently.

Args

map_list: list of SLMap objects, optional Users defined set of parameters scaling. There is default scaling for each object that accept *ctrl* method.

title: string, optional Title of the window. If none is provided, the name of the class is used.

wxnoserver: boolean, optional With wxPython graphical toolkit, if True, tells the interpreter that there will be no server window.

If *wxnoserver* is set to True, the interpreter will not wait for the server GUI before showing the controller window.

table
PyoTableObject. Table containing the waveform samples.

env
PyoTableObject. Table containing the grain envelope.

dens
float or PyoObject. Density of grains per second.

pitch
float or PyoObject. Transposition factor of the grain.

pos
float or PyoObject. Position of the pointer in the sound table.

dur
float or PyoObject. Duration, in seconds, of the grain.

dev
float or PyoObject. Deviation of the starting point of the grain in the table.

pan
float or PyoObject.Panning of the grain.

filterfreq
float or PyoObject.Grain's filter center/cutoff frequency.

filterq
float or PyoObject.Grain's filter Q.

filtertype
float or PyoObject.Grain's filter type.

Pointer

class Pointer(*table*, *index*, *mul=1*, *add=0*)
Table reader with control on the pointer position.

Parent *PyoObject*

Args

table: PyoTableObject Table containing the waveform samples.

index: PyoObject Normalized position in the table between 0 and 1.

```
>>> s = Server().boot()
>>> s.start()
>>> t = SndTable(SNDS_PATH + '/transparent.aif')
>>> freq = t.getRate()
>>> p = Phasor(freq=[freq*0.5, freq*0.45])
>>> a = Pointer(table=t, index=p, mul=.3).out()
```

setTable(*x*)
Replace the *table* attribute.

Args

x: PyoTableObject new *table* attribute.

setIndex(*x*)
Replace the *index* attribute.

Args

x: PyoObject new *index* attribute.

ctrl(*map_list=None*, *title=None*, *wxnoserver=False*)
Opens a sliders window to control the parameters of the object. Only parameters that can be set to a PyoObject are allowed to be mapped on a slider.

If a list of values are given to a parameter, a multisliders will be used to control each stream independently.

Args

map_list: list of SLMap objects, optional Users defined set of parameters scaling. There is default scaling for each object that accept *ctrl* method.

title: string, optional Title of the window. If none is provided, the name of the class is used.

wxnoserver: boolean, optional With wxPython graphical toolkit, if True, tells the interpreter that there will be no server window.

If *wxnoserver* is set to True, the interpreter will not wait for the server GUI before showing the controller window.

table
PyoTableObject. Table containing the waveform samples.

index
PyoObject. Index pointer position in the table.

Pointer2

class Pointer2(*table*, *index*, *interp=4*, *autosmooth=True*, *mul=1*, *add=0*)
High quality table reader with control on the pointer position.

Parent *PyoObject*

Args

table: PyoTableObject Table containing the waveform samples.

index: PyoObject Normalized position in the table between 0 and 1.

interp: int {1, 2, 3, 4}, optional

Choice of the interpolation method. Defaults to 4.

1. no interpolation
2. linear
3. cosinus
4. cubic

autosmooth: boolean, optional If True, a lowpass filter, following the pitch, is applied on the output signal to reduce the quantization noise produced by very low transpositions. Defaults to True.

```
>>> s = Server().boot()
>>> s.start()
>>> t = SndTable(SNDS_PATH + '/transparent.aif')
>>> freq = t.getRate()
>>> p = Phasor(freq=[freq*0.5, freq*0.45])
>>> a = Pointer2(table=t, index=p, mul=.3).out()
```

setTable(*x*)

Replace the *table* attribute.

Args

x: PyoTableObject new *table* attribute.

setIndex(*x*)

Replace the *index* attribute.

Args

x: PyoObject new *index* attribute.

setInterp(*x*)

Replace the *interp* attribute.

Args

x: int {1, 2, 3, 4}

new *interp* attribute.

1. no interpolation
2. linear interpolation
3. cosine interpolation
4. cubic interpolation (default)

setAutoSmooth(*x*)

Replace the *autosmooth* attribute.

If True, a lowpass filter, following the playback speed, is applied on the output signal to reduce the quantization noise produced by very low transpositions.

Args

x: boolean new *autosmooth* attribute.

ctrl(*map_list=None*, *title=None*, *wxnoserver=False*)
Opens a sliders window to control the parameters of the object. Only parameters that can be set to a PyoObject are allowed to be mapped on a slider.

If a list of values are given to a parameter, a multisliders will be used to control each stream independently.

Args

map_list: list of SLMap objects, optional Users defined set of parameters scaling. There is default scaling for each object that accept *ctrl* method.

title: string, optional Title of the window. If none is provided, the name of the class is used.

wxnoserver: boolean, optional With wxPython graphical toolkit, if True, tells the interpreter that there will be no server window.

If *wxnoserver* is set to True, the interpreter will not wait for the server GUI before showing the controller window.

table
PyoTableObject. Table containing the waveform samples.

index
PyoObject. Index pointer position in the table.

interp
int {1, 2, 3, 4}. Interpolation method.

autosmooth
boolean. Quantization noise filter.

Pulsar

class Pulsar(*table*, *env*, *freq=100*, *frac=0.5*, *phase=0*, *interp=2*, *mul=1*, *add=0*)
Pulsar synthesis oscillator.

Pulsar synthesis produces a train of sound particles called pulsars that can make rhythms or tones, depending on the fundamental frequency of the train. Varying the *frac* parameter changes the portion of the period assigned to the waveform and the portion of the period assigned to its following silence, but maintain the overall pulsar period. This results in an effect much like a sweeping band-pass filter.

Parent `PyoObject`

Args

table: PyoTableObject Table containing the waveform samples.

env: PyoTableObject Table containing the envelope samples.

freq: float or PyoObject, optional Frequency in cycles per second. Defaults to 100.

frac: float or PyoObject, optional Fraction of the whole period (0 -> 1) given to the waveform. The rest will be filled with zeros. Defaults to 0.5.

phase: float or PyoObject, optional Phase of sampling, expressed as a fraction of a cycle (0 to 1). Defaults to 0.

interp: int, optional

Choice of the interpolation method. Defaults to 2.

1. no interpolation

2. linear
3. cosinus
4. cubic

See also:

Osc

```
>>> s = Server().boot()
>>> s.start()
>>> w = HarmTable([1,0,.33,0,2,0,.143,0,.111])
>>> e = HannTable()
>>> lfo = Sine([.1,.15], mul=.2, add=.5)
>>> a = Pulsar(table=w, env=e, freq=80, frac=lfo, mul=.08).out()
```

setTable(*x*)
Replace the *table* attribute.

Args

x: PyoTableObject new *table* attribute.

setEnv(*x*)
Replace the *env* attribute.

Args

x: PyoTableObject new *env* attribute.

setFreq(*x*)
Replace the *freq* attribute.

Args

x: float or PyoObject new *freq* attribute.

setFrac(*x*)
Replace the *frac* attribute.

Args

x: float or PyoObject new *frac* attribute.

setPhase(*x*)
Replace the *phase* attribute.

Args

x: float or PyoObject new *phase* attribute.

setInterp(*x*)
Replace the *interp* attribute.

Args

x: int {1, 2, 3, 4}

Choice of the interpolation method.

1. no interpolation
2. linear
3. cosinus
4. cubic

ctrl(*map_list=None*, *title=None*, *wxnoserver=False*)
Opens a sliders window to control the parameters of the object. Only parameters that can be set to a PyoObject are allowed to be mapped on a slider.

If a list of values are given to a parameter, a multisliders will be used to control each stream independently.

Args

map_list: list of SLMap objects, optional Users defined set of parameters scaling. There is default scaling for each object that accept *ctrl* method.

title: string, optional Title of the window. If none is provided, the name of the class is used.

wxnoserver: boolean, optional With wxPython graphical toolkit, if True, tells the interpreter that there will be no server window.

If *wxnoserver* is set to True, the interpreter will not wait for the server GUI before showing the controller window.

table
PyoTableObject. Table containing the waveform samples.

env
PyoTableObject. Table containing the envelope samples.

freq
float or PyoObject. Frequency in cycles per second.

frac
float or PyoObject. Fraction of the period assigned to waveform.

phase
float or PyoObject. Phase of sampling.

interp
int {1, 2, 3, 4}. Interpolation method.

TableIndex

class TableIndex(*table*, *index*, *mul=1*, *add=0*)
Table reader by sample position without interpolation.

Parent *PyoObject*

Args

table: PyoTableObject Table containing the samples.

index: PyoObject Position in the table, as integer audio stream, between 0 and table's size - 1.

```
>>> s = Server().boot()
>>> s.start()
>>> import random
>>> notes = [midiToHz(random.randint(60,84)) for i in range(10)]
>>> tab = DataTable(size=10, init=notes)
>>> ind = RandInt(10, [4,8])
>>> pit = TableIndex(tab, ind)
>>> a = SineLoop(freq=pit, feedback = 0.05, mul=.2).out()
```

setTable(*x*)
Replace the *table* attribute.

Args

x: PyoTableObject new *table* attribute.

setIndex(*x*)
Replace the *index* attribute.

Args

x: PyoObject new *index* attribute.

ctrl(*map_list=None*, *title=None*, *wxnoserver=False*)
Opens a sliders window to control the parameters of the object. Only parameters that can be set to a PyoObject are allowed to be mapped on a slider.

If a list of values are given to a parameter, a multisliders will be used to control each stream independently.

Args

map_list: list of SLMap objects, optional Users defined set of parameters scaling. There is default scaling for each object that accept *ctrl* method.

title: string, optional Title of the window. If none is provided, the name of the class is used.

wxnoserver: boolean, optional With wxPython graphical toolkit, if True, tells the interpreter that there will be no server window.

If *wxnoserver* is set to True, the interpreter will not wait for the server GUI before showing the controller window.

table
PyoTableObject. Table containing the samples.

index
PyoObject. Position in the table.

TableMorph

class TableMorph(*input*, *table*, *sources*)
Morphs between multiple PyoTableObjects.

Uses an index into a list of PyoTableObjects to morph between adjacent tables in the list. The resulting morphed function is written into the *table* object at the beginning of each buffer size. The tables in the list and the resulting table must be equal in size.

Parent `PyoObject`

Args

input: PyoObject Morphing index between 0 and 1. 0 is the first table in the list and 1 is the last.

table: NewTable The table where to write morphed waveform.

sources: list of PyoTableObject List of tables to interpolate from.

Note: The out() method is bypassed. TableMorph returns no signal.

TableMorph has no *mul* and *add* attributes.

```
>>> s = Server(duplex=1).boot()
>>> s.start()
>>> t1 = HarmTable([1,.5,.33,.25,.2,.167,.143,.125,.111,.1,.091])
>>> t2 = HarmTable([1,0,.33,0,.2,0,.143,0,.111,0,.091])
>>> t3 = NewTable(length=8192./s.getSamplingRate(), chnls=1)
>>> lfo = Sine(.25, 0, .5, .5)
>>> mor = TableMorph(lfo, t3, [t1,t2])
>>> osc = Osc(t3, freq=[199.5,200], mul=.08).out()
```

out (*chnl=0*, *inc=1*, *dur=0*, *delay=0*)

Start processing and send samples to audio output beginning at *chnl*.

This method returns *self*, allowing it to be applied at the object creation.

Args

chnl: int, optional Physical output assigned to the first audio stream of the object. Defaults to 0.

inc: int, optional Output channel increment value. Defaults to 1.

dur: float, optional Duration, in seconds, of the object's activation. The default is 0 and means infinite duration.

delay: float, optional Delay, in seconds, before the object's activation. Defaults to 0.

If *chnl* >= 0, successive streams increment the output number by *inc* and wrap around the global number of channels.

If *chnl* is negative, streams begin at 0, increment the output number by *inc* and wrap around the global number of channels. Then, the list of streams is scrambled.

If *chnl* is a list, successive values in the list will be assigned to successive streams.

setMul (*x*)

Replace the *mul* attribute.

Args

x: float or PyoObject New *mul* attribute.

setAdd (*x*)

Replace the *add* attribute.

Args

x: float or PyoObject New *add* attribute.

setInput (*x*, *fadetime=0.05*)

Replace the *input* attribute.

Args

x: PyoObject New signal to process.

fadetime: float, optional Crossfade time between old and new input. Defaults to 0.05.

setTable (*x*)

Replace the *table* attribute.

Args

x: NewTable new *table* attribute.

setSources (*x*)

Replace the *sources* attribute.

Args

x: list of PyoTableObject new *sources* attribute.

input
PyoObject. Morphing index between 0 and 1.

table
NewTable. The table where to write samples.

sources
list of PyoTableObject. List of tables to interpolate from.

TablePut

class TablePut (*input*, *table*)
Writes values, without repetitions, from an audio stream into a DataTable.

See *DataTable* to create an empty table.

TablePut takes an audio input and writes values into a DataTable but only when value changes. This allow to record only new values, without repetitions.

The play method is not called at the object creation time. It starts the recording into the table until the table is full. Calling the play method again restarts the recording and overwrites previously recorded values. The stop method stops the recording. Otherwise, the default behaviour is to record through the end of the table.

Parent *PyoObject*

Args

input: PyoObject Audio signal to write in the table.

table: DataTable The table where to write values.

Note: The out() method is bypassed. TablePut returns no signal.

TablePut has no *mul* and *add* attributes.

TablePut will send a trigger signal at the end of the recording. User can retrieve the trigger streams by calling obj['trig'].

See also:

DataTable, *NewTable*, *TableRec*

```
>>> s = Server().boot()
>>> s.start()
>>> t = DataTable(size=16)
>>> rnd = Choice(range(200, 601, 50), freq=16)
>>> rec = TablePut(rnd, t).play()
>>> met = Metro(.125).play()
>>> ind = Counter(met, max=16)
>>> fr = TableIndex(t, ind, mul=[1,1.005])
>>> osc = SineLoop(fr, feedback=.08, mul=.3).out()
```

out(*chnl=0*, *inc=1*, *dur=0*, *delay=0*)
Start processing and send samples to audio output beginning at *chnl*.

This method returns *self*, allowing it to be applied at the object creation.

Args

chnl: int, optional Physical output assigned to the first audio stream of the object. Defaults to 0.

inc: int, optional Output channel increment value. Defaults to 1.

dur: float, optional Duration, in seconds, of the object's activation. The default is 0 and means infinite duration.

delay: float, optional Delay, in seconds, before the object's activation. Defaults to 0.

If *chnl* >= 0, successive streams increment the output number by *inc* and wrap around the global number of channels.

If *chnl* is negative, streams begin at 0, increment the output number by *inc* and wrap around the global number of channels. Then, the list of streams is scrambled.

If *chnl* is a list, successive values in the list will be assigned to successive streams.

setMul(*x*)
Replace the *mul* attribute.

Args

x: float or PyoObject New *mul* attribute.

setAdd(*x*)
Replace the *add* attribute.

Args

x: float or PyoObject New *add* attribute.

setInput(*x*, *fadetime=0.05*)
Replace the *input* attribute.

Args

x: PyoObject New signal to process.

fadetime: float, optional Crossfade time between old and new input. Defaults to 0.05.

setTable(*x*)
Replace the *table* attribute.

Args

x: DataTable new *table* attribute.

input
PyoObject. Audio signal to write in the table.

table
DataTable. The table where to write values.

TableRead

class TableRead(*table*, *freq=1*, *loop=0*, *interp=2*, *mul=1*, *add=0*)
Simple waveform table reader.

Read sampled sound from a table, with optional looping mode.

The play() method starts the playback and is not called at the object creation time.

Parent *PyoObject*

Args

table: PyoTableObject Table containing the waveform samples.

freq: float or PyoObject, optional Frequency in cycles per second. Defaults to 1.

loop: int {0, 1}, optional Looping mode, 0 means off, 1 means on. Defaults to 0.

interp: int, optional

Choice of the interpolation method. Defaults to 2.

1. no interpolation
2. linear
3. cosinus
4. cubic

Note: TableRead will send a trigger signal at the end of the playback if loop is off or any time it wraps around if loop is on. User can retreive the trigger streams by calling obj['trig']:

```
>>> tabr = TableRead(SNDS_PATH + "/transparent.aif").out()
>>> trig = TrigRand(tabr['trig'])
```

See also:

Osc

```
>>> s = Server().boot()
>>> s.start()
>>> snd = SndTable(SNDS_PATH + '/transparent.aif')
>>> freq = snd.getRate()
>>> a = TableRead(table=snd, freq=[freq,freq*.99], loop=True, mul=.3).out()
```

setTable(*x*)

Replace the *table* attribute.

Args

x: PyoTableObject new *table* attribute.

setFreq(*x*)

Replace the *freq* attribute.

Args

x: float or PyoObject new *freq* attribute.

setLoop(*x*)

Replace the *loop* attribute.

Args

x: int {0, 1} new *loop* attribute.

setInterp(*x*)
Replace the *interp* attribute.

Args

x: int {1, 2, 3, 4} new *interp* attribute.

setKeepLast(*x*)
If anything but zero, the object will hold the last value when stopped.

Args

x: bool If 0, the object will be reset to 0 when stopped, otherwise it will hold its last value.

reset()
Resets current phase to 0.

ctrl(*map_list=None*, *title=None*, *wxnoserver=False*)
Opens a sliders window to control the parameters of the object. Only parameters that can be set to a PyoObject are allowed to be mapped on a slider.

If a list of values are given to a parameter, a multisliders will be used to control each stream independently.

Args

map_list: list of SLMap objects, optional Users defined set of parameters scaling. There is default scaling for each object that accept *ctrl* method.

title: string, optional Title of the window. If none is provided, the name of the class is used.

wxnoserver: boolean, optional With wxPython graphical toolkit, if True, tells the interpreter that there will be no server window.

If *wxnoserver* is set to True, the interpreter will not wait for the server GUI before showing the controller window.

table
PyoTableObject. Table containing the waveform samples.

freq
float or PyoObject. Frequency in cycles per second.

loop
int. Looping mode.

interp
int {1, 2, 3, 4}. Interpolation method.

TableRec

class TableRec(*input*, *table*, *fadetime=0*)
TableRec is for writing samples into a previously created NewTable.

See *NewTable* to create an empty table.

The play method is not called at the object creation time. It starts the recording into the table until the table is full. Calling the play method again restarts the recording and overwrites previously recorded samples. The stop method stops the recording. Otherwise, the default behaviour is to record through the end of the table.

Parent *PyoObject*

Args

input: PyoObject Audio signal to write in the table.

table: NewTable The table where to write samples.

fadetime: float, optional Fade time at the beginning and the end of the recording in seconds. Defaults to 0.

Note: The out() method is bypassed. TableRec returns no signal.

TableRec has no *mul* and *add* attributes.

TableRec will send a trigger signal at the end of the recording. User can retrieve the trigger streams by calling obj['trig']. In this example, the recorded table will be read automatically after a recording:

```
>>> a = Input(0)
>>> t = NewTable(length=1, chnls=1)
>>> rec = TableRec(a, table=t, fadetime=0.01)
>>> tr = TrigEnv(rec['trig'], table=t, dur=1).out()
```

obj['time'] outputs an audio stream of the current recording time, in samples.

See also:

NewTable, *TrigTableRec*

```
>>> s = Server(duplex=1).boot()
>>> s.start()
>>> t = NewTable(length=2, chnls=1)
>>> a = Input(0)
>>> b = TableRec(a, t, .01)
>>> amp = Iter(b["trig"], [.5])
>>> freq = t.getRate()
>>> c = Osc(t, [freq, freq*.99], mul=amp).out()
>>> # to record in the empty table, call:
>>> # b.play()
```

out(*chnl=0*, *inc=1*, *dur=0*, *delay=0*)

Start processing and send samples to audio output beginning at *chnl*.

This method returns *self*, allowing it to be applied at the object creation.

Args

chnl: int, optional Physical output assigned to the first audio stream of the object. Defaults to 0.

inc: int, optional Output channel increment value. Defaults to 1.

dur: float, optional Duration, in seconds, of the object's activation. The default is 0 and means infinite duration.

delay: float, optional Delay, in seconds, before the object's activation. Defaults to 0.

If *chnl* >= 0, successive streams increment the output number by *inc* and wrap around the global number of channels.

If *chnl* is negative, streams begin at 0, increment the output number by *inc* and wrap around the global number of channels. Then, the list of streams is scrambled.

If *chnl* is a list, successive values in the list will be assigned to successive streams.

setMul(*x*)
Replace the *mul* attribute.

Args

x: float or PyoObject New *mul* attribute.

setAdd(*x*)
Replace the *add* attribute.

Args

x: float or PyoObject New *add* attribute.

setInput(*x*, *fadetime=0.05*)
Replace the *input* attribute.

Args

x: PyoObject New signal to process.

fadetime: float, optional Crossfade time between old and new input. Defaults to 0.05.

setTable(*x*)
Replace the *table* attribute.

Args

x: NewTable new *table* attribute.

input
PyoObject. Audio signal to write in the table.

table
NewTable. The table where to write samples.

TableWrite

class TableWrite(*input*, *pos*, *table*, *mode=0*, *maxwindow=1024*)
TableWrite writes samples into a previously created NewTable.

See *NewTable* to create an empty table.

TableWrite takes samples from its *input* stream and writes them at a normalized or raw position given by the *pos* stream. Position must be an audio stream, ie. a PyoObject. This object allows fast recording of values coming from an X-Y pad into a table object.

Parent `PyoObject`

Args

input: PyoObject Audio signal to write in the table.

pos: PyoObject Audio signal specifying the position where to write the *input* samples. It is a normalized position (in the range 0 to 1) in mode=0 or the raw position (in samples) for any other value of mode.

table: NewTable The table where to write samples.

mode: int, optional Sets the writing pointer mode. If 0, the position must be normalized between 0 (beginning of the table) and 1 (end of the table). For any other value, the position must be in samples between 0 and the length of the table. Available at initialization time only.

maxwindow: int optional Maximum length, in samples, of the interpolated window when the position is moving fast. Useful to avoid interpolation over the entire table if using a circular writing position. Available at initialization time only. Defaults to 1024.

Note: The out() method is bypassed. TableWrite returns no signal.

TableWrite has no *mul* and *add* attributes.

See also:

NewTable, *TableRec*

```
>>> s = Server(duplex=1).boot()
>>> s.start()
>>> tab = NewTable(8192/s.getSamplingRate())
>>> tab.view()
>>> pos = Phasor(.5)
>>> val = Sine(.25)
>>> rec = TableWrite(val, pos, tab)
>>> pat = Pattern(tab.refreshView, 0.05).play()
```

out (*chnl=0*, *inc=1*, *dur=0*, *delay=0*)

Start processing and send samples to audio output beginning at *chnl*.

This method returns *self*, allowing it to be applied at the object creation.

Args

chnl: int, optional Physical output assigned to the first audio stream of the object. Defaults to 0.

inc: int, optional Output channel increment value. Defaults to 1.

dur: float, optional Duration, in seconds, of the object's activation. The default is 0 and means infinite duration.

delay: float, optional Delay, in seconds, before the object's activation. Defaults to 0.

If *chnl* >= 0, successive streams increment the output number by *inc* and wrap around the global number of channels.

If *chnl* is negative, streams begin at 0, increment the output number by *inc* and wrap around the global number of channels. Then, the list of streams is scrambled.

If *chnl* is a list, successive values in the list will be assigned to successive streams.

setMul (*x*)

Replace the *mul* attribute.

Args

x: float or PyoObject New *mul* attribute.

setAdd (*x*)

Replace the *add* attribute.

Args

x: float or PyoObject New *add* attribute.

setInput (*x*, *fadetime=0.05*)

Replace the *input* attribute.

Args

x: PyoObject New signal to process.

fadetime: float, optional Crossfade time between old and new input. Defaults to 0.05.

setTable(*x*)
Replace the *table* attribute.

Args

x: NewTable new *table* attribute.

setPos(*x*)
Replace the *pos* attribute.

Args

x: PyoObject new *pos* attribute.

input
PyoObject. Audio signal to write in the table.

table
NewTable. The table where to write samples.

pos
PyoObject. Normalized position, as audio stream, where to write samples.

TableScale

class TableScale(*table*, *outtable*, *mul=1*, *add=0*)
Scales all the values contained in a PyoTableObject.

TableScale scales the values of *table* argument according to *mul* and *add* arguments and writes the new values in *outtable*.

Parent *PyoObject*

Args

table: PyoTableObject Table containing the original values.

outtable: PyoTableObject Table where to write the scaled values.

```
>>> s = Server().boot()
>>> s.start()
>>> t = DataTable(size=12, init=midiToHz(range(48, 72, 2)))
>>> t2 = DataTable(size=12)
>>> m = Metro(.2).play()
>>> c = Counter(m ,min=0, max=12)
>>> f1 = TableIndex(t, c)
>>> syn1 = SineLoop(f1, feedback=0.08, mul=0.3).out()
>>> scl = TableScale(t, t2, mul=1.5)
>>> f2 = TableIndex(t2, c)
>>> syn2 = SineLoop(f2, feedback=0.08, mul=0.3).out(1)
```

setTable(*x*)
Replace the *table* attribute.

Args

x: PyoTableObject new *table* attribute.

setOuttable(*x*)
Replace the *outtable* attribute.

Args

x: PyoTableObject new *outtable* attribute.

ctrl(*map_list=None*, *title=None*, *wxnoserver=False*)
Opens a sliders window to control the parameters of the object. Only parameters that can be set to a PyoObject are allowed to be mapped on a slider.

If a list of values are given to a parameter, a multisliders will be used to control each stream independently.

Args

map_list: list of SLMap objects, optional Users defined set of parameters scaling. There is default scaling for each object that accept *ctrl* method.

title: string, optional Title of the window. If none is provided, the name of the class is used.

wxnoserver: boolean, optional With wxPython graphical toolkit, if True, tells the interpreter that there will be no server window.

If *wxnoserver* is set to True, the interpreter will not wait for the server GUI before showing the controller window.

table
PyoTableObject. Table containing the original values.

outtable
PyoTableObject. Scaled output table.

TableFill

class TableFill(*input*, *table*)
Continuously fills a table with incoming samples.

See *DataTable* or *NewTable* to create an empty table.

TableFill takes an audio input and writes values into a PyoTableObject, samples by samples. It wraps around the table length when reaching the end of the table.

Calling the play method reset the writing position to 0.

Parent *PyoObject*

Args

input: PyoObject Audio signal to write in the table.

table: PyoTableObject The table where to write values.

Note: The out() method is bypassed. TableFill returns no signal.

TableFill has no *mul* and *add* attributes.

See also:

TableWrite, *TablePut*, *TableRec*

```
>>> s = Server().boot()
>>> s.start()
>>> table = DataTable(s.getBufferSize(), chnls=2)
>>> sig = Sine([400,500], mul=0.3)
>>> fill = TableFill(sig, table)
>>> read = TableRead(table, table.getRate(), loop=True).out()
```

out (*chnl=0*, *inc=1*, *dur=0*, *delay=0*)

Start processing and send samples to audio output beginning at *chnl*.

This method returns *self*, allowing it to be applied at the object creation.

Args

chnl: int, optional Physical output assigned to the first audio stream of the object. Defaults to 0.

inc: int, optional Output channel increment value. Defaults to 1.

dur: float, optional Duration, in seconds, of the object's activation. The default is 0 and means infinite duration.

delay: float, optional Delay, in seconds, before the object's activation. Defaults to 0.

If *chnl* >= 0, successive streams increment the output number by *inc* and wrap around the global number of channels.

If *chnl* is negative, streams begin at 0, increment the output number by *inc* and wrap around the global number of channels. Then, the list of streams is scrambled.

If *chnl* is a list, successive values in the list will be assigned to successive streams.

setMul (*x*)

Replace the *mul* attribute.

Args

x: float or PyoObject New *mul* attribute.

setAdd (*x*)

Replace the *add* attribute.

Args

x: float or PyoObject New *add* attribute.

setInput (*x*, *fadetime=0.05*)

Replace the *input* attribute.

Args

x: PyoObject New signal to process.

fadetime: float, optional Crossfade time between old and new input. Defaults to 0.05.

setTable (*x*)

Replace the *table* attribute.

Args

x: PyoTableObject new *table* attribute.

getCurrentPos (*all=False*)

Returns the current pointer position, in samples, in the table.

Args

all: boolean, optional If True, returns the current position of all internal objects as a list.

If False, only the current position of the first object's stream will be returned as a float.

input
PyoObject. Audio signal to write in the table.

table
PyoTableObject. The table where to write values.

TableScan

class TableScan(*table*, *mul=1*, *add=0*)
Reads the content of a table in loop, without interpolation.

A simple table reader, sample by sample, with wrap-around when reaching the end of the table.

Parent *PyoObject*

Args

table: PyoTableObject Table containing the waveform samples.

See also:

Osc, *TableRead*

```
>>> s = Server().boot()
>>> s.start()
>>> tab = DataTable(s.getBufferSize(), 2)
>>> sig = Sine([500, 600], mul=0.3)
>>> fill = TableFill(sig, tab)
>>> scan = TableScan(tab).out()
```

setTable(*x*)
Replace the *table* attribute.

Args

x: PyoTableObject new *table* attribute.

reset()
Resets current phase to 0.

table
PyoTableObject. Table containing the waveform samples.

Sample Accurate Timing (Triggers)

Set of objects to manage triggers streams.

A trigger is an audio signal with a value of 1 surrounded by 0s.

TrigXXX objects use this kind of signal to generate different processes with sample rate timing accuracy.

Beat

class Beat(*time=0.125*, *taps=16*, *w1=80*, *w2=50*, *w3=30*, *poly=1*, *onlyonce=False*)
Generates algorithmic trigger patterns.

A trigger is an audio signal with a value of 1 surrounded by 0s.

Beat generates measures of length *taps* and uses weight parameters (*w1*, *w2* and *w3*) to compute the chances of a beat to be present in the generated measure.

User can store the current pattern in one of the 32 preset slots with the store() method and recall it later with recall(x).

A preset is a list where the first value is the number of beats in the measure, followed by 1s and 0s. For a 4/4 measure with only down beats:

[16, 1, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0]

The play() method starts the Beat and is not called at the object creation time.

Parent *PyoObject*

Args

time: float or PyoObject, optional Time, in seconds, between each beat of the pattern. Defaults to 0.125.

taps: int, optional Number of beats in the generated pattern, max = 64. Defaults to 16.

w1: int {0 -> 100}, optional Probability for down beats. Defaults to 80.

w2: int {0 -> 100}, optional Probability for up beats. Defaults to 50.

w3: int {0 -> 100}, optional Probability for the weakest beats. Defaults to 30.

poly: int, optional Beat polyphony. Denotes how many independent streams are generated by the object, allowing overlapping processes.

Available only at initialization. Defaults to 1.

onlyonce: boolean, optional If True, the sequence will play only once and automatically stop. Defaults to False.

Note: Beat outputs many signals identified with a string between brackets:

obj['tap'] returns audio stream of the current tap of the measure.
obj['amp'] returns audio stream of the current beat amplitude.
obj['dur'] returns audio stream of the current beat duration in seconds.
obj['end'] returns audio stream with a trigger just before the end of the measure.

obj without brackets returns the generated trigger stream of the measure.

The out() method is bypassed. Beat's signal can not be sent to audio outs.

Beat has no *mul* and *add* attributes.

```
>>> s = Server().boot()
>>> s.start()
>>> t = CosTable([(0,0), (100,1), (500,.3), (8191,0)])
>>> beat = Beat(time=.125, taps=16, w1=[90,80], w2=50, w3=35, poly=1).play()
>>> trmid = TrigXnoiseMidi(beat, dist=12, mrange=(60, 96))
>>> trhz = Snap(trmid, choice=[0,2,3,5,7,8,10], scale=1)
>>> tr2 = TrigEnv(beat, table=t, dur=beat['dur'], mul=beat['amp'])
>>> a = Sine(freq=trhz, mul=tr2*0.3).out()
```

get(*identifier='amp'*, *all=False*)
Return the first sample of the current buffer as a float.

Can be used to convert audio stream to usable Python data.

"tap", "amp" or "dur" must be given to *identifier* to specify which stream to get value from.

Args

identifier: string {"tap", "amp", "dur"} Address string parameter identifying audio stream. Defaults to "amp".

all: boolean, optional If True, the first value of each object's stream will be returned as a list.

If False, only the value of the first object's stream will be returned as a float.

reset()
Reset internal counters to initialization values.

new(*now=False*)
Generates a new pattern with the current parameters.

Args

now: boolean If True, the new sequence is immediately generated, otherwise (the default), the new sequence is generated after the current sequence is completed.

fill()
Generates a fill-in pattern and then restore the current one.

store(*x*)
Store the current pattern in memory *x*.

Args

x: int Memory number. 0 <= x < 32.

recall(*x*)
Recall the pattern previously stored in memory *x*.

Args

x: int Memory number. 0 <= x < 32.

getPresets()
Returns the list of stored presets.

setPresets(*x*)
Store a list presets.

Args

x: list List of presets.

setTime(*x*)
Replace the *time* attribute.

Args

x: float or PyoObject New *time* attribute.

setTaps(*x*)
Replace the *taps* attribute.

Args

x: int New *taps* attribute.

setW1(*x*)
Replace the *w1* attribute.

Args

x: int New *w1* attribute.

setW2(*x*)
Replace the *w2* attribute.

Args

x: int New *w2* attribute.

setW3(*x*)
Replace the *w3* attribute.

Args

x: int New *w3* attribute.

setWeights(*w1=None*, *w2=None*, *w3=None*)
Replace the weight attributes.

Arguments set to *None* remain unchanged.

Args

w1: int, optional New *w1* attribute. Defaults to None.

w2: int, optional New *w2* attribute. Defaults to None.

w3: int, optional New *w3* attribute. Defaults to None.

setOnlyonce(*x*)
Replace the *onlyonce* attribute.

Args

x: boolean New *onlyonce* attribute.

play(*dur=0*, *delay=0*)
Start processing without sending samples to output. This method is called automatically at the object creation.

This method returns *self*, allowing it to be applied at the object creation.

Args

dur: float, optional Duration, in seconds, of the object's activation. The default is 0 and means infinite duration.

delay: float, optional Delay, in seconds, before the object's activation. Defaults to 0.

stop(*wait=0*)
Stop processing.

This method returns *self*, allowing it to be applied at the object creation.

Args

wait: float, optional Delay, in seconds, before the process is actually stopped. If autoStartChildren is activated in the Server, this value is propagated to the children objects. Defaults to 0.

Note: if the method setStopDelay(x) was called before calling stop(wait) with a positive *wait* value, the *wait* value won't overwrite the value given to setStopDelay for the current object, but will be the one propagated to children objects. This allow to set a waiting time for a specific object with setStopDelay whithout changing the global delay time given to the stop method.

Fader and Adsr objects ignore waiting time given to the stop method because they already implement a delayed processing triggered by the stop call.

out(*chnl=0*, *inc=1*, *dur=0*, *delay=0*)
Start processing and send samples to audio output beginning at *chnl*.

This method returns *self*, allowing it to be applied at the object creation.

Args

chnl: int, optional Physical output assigned to the first audio stream of the object. Defaults to 0.

inc: int, optional Output channel increment value. Defaults to 1.

dur: float, optional Duration, in seconds, of the object's activation. The default is 0 and means infinite duration.

delay: float, optional Delay, in seconds, before the object's activation. Defaults to 0.

If *chnl* >= 0, successive streams increment the output number by *inc* and wrap around the global number of channels.

If *chnl* is negative, streams begin at 0, increment the output number by *inc* and wrap around the global number of channels. Then, the list of streams is scrambled.

If *chnl* is a list, successive values in the list will be assigned to successive streams.

setMul(*x*)
Replace the *mul* attribute.

Args

x: float or PyoObject New *mul* attribute.

setAdd(*x*)
Replace the *add* attribute.

Args

x: float or PyoObject New *add* attribute.

setSub(*x*)
Replace and inverse the *add* attribute.

Args

x: float or PyoObject New inversed *add* attribute.

setDiv(*x*)
Replace and inverse the *mul* attribute.

Args

x: float or PyoObject New inversed *mul* attribute.

ctrl(*map_list=None*, *title=None*, *wxnoserver=False*)
Opens a sliders window to control the parameters of the object. Only parameters that can be set to a PyoObject are allowed to be mapped on a slider.

If a list of values are given to a parameter, a multisliders will be used to control each stream independently.

Args

map_list: list of SLMap objects, optional Users defined set of parameters scaling. There is default scaling for each object that accept *ctrl* method.

title: string, optional Title of the window. If none is provided, the name of the class is used.

wxnoserver: boolean, optional With wxPython graphical toolkit, if True, tells the interpreter that there will be no server window.

If *wxnoserver* is set to True, the interpreter will not wait for the server GUI before showing the controller window.

time
float or PyoObject. Time, in seconds, between each beat.

taps
int. Number of beats in the generated pattern.

w1
int. Probability for down beats.

w2
int. Probability for up beats.

w3
int. Probability for other beats.

onlyonce
boolean. Trigger the sequence only once.

Change

class Change(*input*, *mul=1*, *add=0*)
Sends trigger that informs when input value has changed.

Parent *PyoObject*

Args

input: PyoObject Audio signal. Must contains integer numbers.

Note: The out() method is bypassed. Change's signal can not be sent to audio outs.

```
>>> s = Server().boot()
>>> s.start()
>>> t = CosTable([(0,0), (100,1), (500,.3), (8191,0)])
>>> a = XnoiseMidi(dist="loopseg", freq=[2, 3], x1=1, scale=1, mrange=(60,73))
>>> b = Change(a)
>>> amp = TrigEnv(b, table=t, dur=[.5,.333], mul=.3)
>>> out = SineLoop(freq=a, feedback=.05, mul=amp).out()
```

out(*chnl=0*, *inc=1*, *dur=0*, *delay=0*)
Start processing and send samples to audio output beginning at *chnl*.

This method returns *self*, allowing it to be applied at the object creation.

Args

chnl: int, optional Physical output assigned to the first audio stream of the object. Defaults to 0.

inc: int, optional Output channel increment value. Defaults to 1.

dur: float, optional Duration, in seconds, of the object's activation. The default is 0 and means infinite duration.

delay: float, optional Delay, in seconds, before the object's activation. Defaults to 0.

If *chnl* >= 0, successive streams increment the output number by *inc* and wrap around the global number of channels.

If *chnl* is negative, streams begin at 0, increment the output number by *inc* and wrap around the global number of channels. Then, the list of streams is scrambled.

If *chnl* is a list, successive values in the list will be assigned to successive streams.

setInput(*x*, *fadetime=0.05*)
Replace the *input* attribute.

Args

x: PyoObject New signal to process.

fadetime: float, optional Crossfade time between old and new input. Defaults to 0.05.

input
PyoObject. Audio signal.

Cloud

class Cloud(*density=10*, *poly=1*)
Generates random triggers.

Generates random triggers with control over the generation density.

A trigger is an audio signal with a value of 1 surrounded by 0s.

The play() method starts the Cloud and is not called at the object creation time.

Parent *PyoObject*

Args

density: float or PyoObject, optional Average number of triggers per second. Defaults to 10.

poly: int, optional Cloud polyphony. Denotes how many independent streams are generated by the object, allowing overlapping processes.

Available only at initialization. Defaults to 1.

Note: The out() method is bypassed. Cloud's signal can not be sent to audio outs.

Cloud has no *mul* and *add* attributes.

```
>>> s = Server().boot()
>>> s.start()
>>> dens = Expseg([(0,1),(5,50)], loop=True, exp=5, initToFirstVal=True).play()
>>> m = Cloud(density=dens, poly=2).play()
>>> tr = TrigRand(m, min=300, max=1000)
>>> tr_p = Port(tr, risetime=0.001, falltime=0.001)
>>> a = Sine(freq=tr, mul=0.2).out()
```

setDensity(*x*)
Replace the *density* attribute.

Args

x: float or PyoObject New *density* attribute.

out(*chnl=0*, *inc=1*, *dur=0*, *delay=0*)
Start processing and send samples to audio output beginning at *chnl*.

This method returns *self*, allowing it to be applied at the object creation.

Args

chnl: int, optional Physical output assigned to the first audio stream of the object. Defaults to 0.

inc: int, optional Output channel increment value. Defaults to 1.

dur: float, optional Duration, in seconds, of the object's activation. The default is 0 and means infinite duration.

delay: float, optional Delay, in seconds, before the object's activation. Defaults to 0.

If *chnl* >= 0, successive streams increment the output number by *inc* and wrap around the global number of channels.

If *chnl* is negative, streams begin at 0, increment the output number by *inc* and wrap around the global number of channels. Then, the list of streams is scrambled.

If *chnl* is a list, successive values in the list will be assigned to successive streams.

setMul(*x*)
Replace the *mul* attribute.

Args

x: float or PyoObject New *mul* attribute.

setAdd(*x*)
Replace the *add* attribute.

Args

x: float or PyoObject New *add* attribute.

setSub(*x*)
Replace and inverse the *add* attribute.

Args

x: float or PyoObject New inversed *add* attribute.

setDiv(*x*)
Replace and inverse the *mul* attribute.

Args

x: float or PyoObject New inversed *mul* attribute.

ctrl(*map_list=None*, *title=None*, *wxnoserver=False*)
Opens a sliders window to control the parameters of the object. Only parameters that can be set to a PyoObject are allowed to be mapped on a slider.

If a list of values are given to a parameter, a multisliders will be used to control each stream independently.

Args

map_list: list of SLMap objects, optional Users defined set of parameters scaling. There is default scaling for each object that accept *ctrl* method.

title: string, optional Title of the window. If none is provided, the name of the class is used.

wxnoserver: boolean, optional With wxPython graphical toolkit, if True, tells the interpreter that there will be no server window.

If *wxnoserver* is set to True, the interpreter will not wait for the server GUI before showing the controller window.

density
float or PyoObject. Average density of triggers generation.

Count

class Count(*input*, *min=0*, *max=0*, *mul=1*, *add=0*)
Counts integers at audio rate.

Count generates a signal increasing by 1 each sample when it receives a trigger. It can be used to do sample playback using TableIndex.

Parent *PyoObject*

Args

input: PyoObject Trigger signal. Start or Restart the count.

min: int, optional Minimum value of the count, included in the count. Defaults to 0.

max: int, optional Maximum value of the count. excluded of the count. Defaults to 0.

A value of 0 eliminates the maximum, and the count continues increasing without resetting.

```
>>> s = Server().boot()
>>> s.start()
>>> t = SndTable(SNDS_PATH+'/accord.aif')
>>> ind = Count(Trig().play(), [0,100], t.getSize())
>>> read = TableIndex(t, ind).out()
```

setInput(*x*, *fadetime=0.05*)
Replace the *input* attribute.

Args

x: PyoObject New input signal.

fadetime: float, optional Crossfade time between old and new input. Default to 0.05.

setMin(*x*)
Replace the *min* attribute.

Args

x: int new *min* attribute.

setMax(*x*)
Replace the *max* attribute.

Args

x: int new *max* attribute.

ctrl(*map_list=None*, *title=None*, *wxnoserver=False*)
Opens a sliders window to control the parameters of the object. Only parameters that can be set to a PyoObject are allowed to be mapped on a slider.

If a list of values are given to a parameter, a multisliders will be used to control each stream independently.

Args

map_list: list of SLMap objects, optional Users defined set of parameters scaling. There is default scaling for each object that accept *ctrl* method.

title: string, optional Title of the window. If none is provided, the name of the class is used.

wxnoserver: boolean, optional With wxPython graphical toolkit, if True, tells the interpreter that there will be no server window.

If *wxnoserver* is set to True, the interpreter will not wait for the server GUI before showing the controller window.

input
PyoObject. Trigger signal. Start/Restart the count.

min
int. Minimum value.

max
int. Maximum value.

Counter

class Counter(*input*, *min=0*, *max=100*, *dir=0*, *mul=1*, *add=0*)
Integer count generator.

Counter keeps track of all triggers received, outputs the current count constrained within *min* and *max* range, and can be set to count up, down, or up-and-down.

Parent *PyoObject*

Args

input: PyoObject Audio signal sending triggers.

min: int, optional Minimum value of the count, included in the count. Defaults to 0.

max: int, optional Maximum value of the count. excluded of the count. The counter will count up to max - 1. Defaults to 100.

dir: int {0, 1, 2}, optional

Direction of the count. Defaults to 0. Three possible values:

0. up

1. down
2. up-and-down

Note: The out() method is bypassed. Counter's signal can not be sent to audio outs.

See also:

Select

```
>>> s = Server().boot()
>>> s.start()
>>> m = Metro(.125).play()
>>> c = Counter(m, min=3, max=8, dir=2, mul=100)
>>> a = Sine(freq=c, mul=.2).mix(2).out()
```

out (*chnl=0*, *inc=1*, *dur=0*, *delay=0*)

Start processing and send samples to audio output beginning at *chnl*.

This method returns *self*, allowing it to be applied at the object creation.

Args

chnl: int, optional Physical output assigned to the first audio stream of the object. Defaults to 0.

inc: int, optional Output channel increment value. Defaults to 1.

dur: float, optional Duration, in seconds, of the object's activation. The default is 0 and means infinite duration.

delay: float, optional Delay, in seconds, before the object's activation. Defaults to 0.

If *chnl* >= 0, successive streams increment the output number by *inc* and wrap around the global number of channels.

If *chnl* is negative, streams begin at 0, increment the output number by *inc* and wrap around the global number of channels. Then, the list of streams is scrambled.

If *chnl* is a list, successive values in the list will be assigned to successive streams.

setInput (*x*, *fadetime=0.05*)

Replace the *input* attribute.

Args

x: PyoObject New signal to process.

fadetime: float, optional Crossfade time between old and new input. Defaults to 0.05.

setMin (*x*)

Replace the *min* attribute.

Args

x: int new *min* attribute.

setMax (*x*)

Replace the *max* attribute.

Args

x: int new *max* attribute.

setDir(*x*)
Replace the *dir* attribute.

Args

x: int {0, 1, 2} new *dir* attribute.

reset(*value=None*)
Reset the current count of the counter. If *value* is None, the counter resets to the beginning of the count.

Args

value: int, optional Value where to reset the count. Defaults to None.

ctrl(*map_list=None*, *title=None*, *wxnoserver=False*)
Opens a sliders window to control the parameters of the object. Only parameters that can be set to a PyoObject are allowed to be mapped on a slider.

If a list of values are given to a parameter, a multisliders will be used to control each stream independently.

Args

map_list: list of SLMap objects, optional Users defined set of parameters scaling. There is default scaling for each object that accept *ctrl* method.

title: string, optional Title of the window. If none is provided, the name of the class is used.

wxnoserver: boolean, optional With wxPython graphical toolkit, if True, tells the interpreter that there will be no server window.

If *wxnoserver* is set to True, the interpreter will not wait for the server GUI before showing the controller window.

input
PyoObject. Audio trigger signal.

min
int. Minimum value.

max
int. Maximum value.

dir
int. Direction of the count.

Euclide

class Euclide(*time=0.125*, *taps=16*, *onsets=10*, *poly=1*)
Euclidean rhythm generator.

This object generates euclidean trigger patterns, resulting in onsets in the rhythm to be as equidistant as possible.

A trigger is an audio signal with a value of 1 surrounded by 0s.

The play() method starts the Euclide and is not called at the object creation time.

Parent *PyoObject*

Args

time: float or PyoObject, optional Time, in seconds, between each beat of the pattern. Defaults to 0.125.

taps: int, optional Number of beats in the generated pattern (measure length), max = 64. Defaults to 16.

onsets: int, optional Number of onsets (a positive tap) in the generated pattern. Defaults to 10.

poly: int, optional Beat polyphony. Denotes how many independent streams are generated by the object, allowing overlapping processes.

Available only at initialization. Defaults to 1.

Note: Euclide outputs many signals identified with a string between brackets:

obj['tap'] returns audio stream of the current tap of the measure.
obj['amp'] returns audio stream of the current beat amplitude.
obj['dur'] returns audio stream of the current beat duration in seconds.
obj['end'] returns audio stream with a trigger just before the end of the measure.

obj without brackets returns the generated trigger stream of the measure.

The out() method is bypassed. Euclide's signal can not be sent to audio outs.

Euclide has no *mul* and *add* attributes.

```
>>> s = Server().boot()
>>> s.start()
>>> t = CosTable([(0,0), (100,1), (500,.3), (8191,0)])
>>> beat = Euclide(time=.125, taps=16, onsets=[8,7], poly=1).play()
>>> trmid = TrigXnoiseMidi(beat, dist=12, mrange=(60, 96))
>>> trhz = Snap(trmid, choice=[0,2,3,5,7,8,10], scale=1)
>>> tr2 = TrigEnv(beat, table=t, dur=beat['dur'], mul=beat['amp'])
>>> a = Sine(freq=trhz, mul=tr2*0.3).out()
```

get(*identifier='amp'*, *all=False*)

Return the first sample of the current buffer as a float.

Can be used to convert audio stream to usable Python data.

"tap", "amp" or "dur" must be given to *identifier* to specify which stream to get value from.

Args

identifier: string {"tap", "amp", "dur"} Address string parameter identifying audio stream. Defaults to "amp".

all: boolean, optional If True, the first value of each object's stream will be returned as a list.

If False, only the value of the first object's stream will be returned as a float.

setTime(*x*)

Replace the *time* attribute.

Args

x: float or PyoObject New *time* attribute.

setTaps(*x*)
Replace the *taps* attribute.

Args

x: int New *taps* attribute.

setOnsets(*x*)
Replace the *onsets* attribute.

Args

x: int New *onsets* attribute.

reset()
Reset internal counters to initialization values.

play(*dur=0*, *delay=0*)
Start processing without sending samples to output. This method is called automatically at the object creation.

This method returns *self*, allowing it to be applied at the object creation.

Args

dur: float, optional Duration, in seconds, of the object's activation. The default is 0 and means infinite duration.

delay: float, optional Delay, in seconds, before the object's activation. Defaults to 0.

stop(*wait=0*)
Stop processing.

This method returns *self*, allowing it to be applied at the object creation.

Args

wait: float, optional Delay, in seconds, before the process is actually stopped. If autoStartChildren is activated in the Server, this value is propagated to the children objects. Defaults to 0.

Note: if the method setStopDelay(x) was called before calling stop(wait) with a positive *wait* value, the *wait* value won't overwrite the value given to setStopDelay for the current object, but will be the one propagated to children objects. This allow to set a waiting time for a specific object with setStopDelay whithout changing the global delay time given to the stop method.

Fader and Adsr objects ignore waiting time given to the stop method because they already implement a delayed processing triggered by the stop call.

---

out(*chnl=0*, *inc=1*, *dur=0*, *delay=0*)
Start processing and send samples to audio output beginning at *chnl*.

This method returns *self*, allowing it to be applied at the object creation.

Args

chnl: int, optional Physical output assigned to the first audio stream of the object. Defaults to 0.

inc: int, optional Output channel increment value. Defaults to 1.

dur: float, optional Duration, in seconds, of the object's activation. The default is 0 and means infinite duration.

delay: float, optional Delay, in seconds, before the object's activation. Defaults to 0.

If *chnl* >= 0, successive streams increment the output number by *inc* and wrap around the global number of channels.

If *chnl* is negative, streams begin at 0, increment the output number by *inc* and wrap around the global number of channels. Then, the list of streams is scrambled.

If *chnl* is a list, successive values in the list will be assigned to successive streams.

setMul(*x*)
Replace the *mul* attribute.

Args

x: float or PyoObject New *mul* attribute.

setAdd(*x*)
Replace the *add* attribute.

Args

x: float or PyoObject New *add* attribute.

setSub(*x*)
Replace and inverse the *add* attribute.

Args

x: float or PyoObject New inversed *add* attribute.

setDiv(*x*)
Replace and inverse the *mul* attribute.

Args

x: float or PyoObject New inversed *mul* attribute.

ctrl(*map_list=None*, *title=None*, *wxnoserver=False*)
Opens a sliders window to control the parameters of the object. Only parameters that can be set to a PyoObject are allowed to be mapped on a slider.

If a list of values are given to a parameter, a multisliders will be used to control each stream independently.

Args

map_list: list of SLMap objects, optional Users defined set of parameters scaling. There is default scaling for each object that accept *ctrl* method.

title: string, optional Title of the window. If none is provided, the name of the class is used.

wxnoserver: boolean, optional With wxPython graphical toolkit, if True, tells the interpreter that there will be no server window.

If *wxnoserver* is set to True, the interpreter will not wait for the server GUI before showing the controller window.

time
float or PyoObject. Time, in seconds, between each beat.

taps
int. Number of beats in the generated pattern.

onsets
int. Number of onsets in the generated pattern.

Iter

class Iter(*input*, *choice*, *init=0.0*, *mul=1*, *add=0*)

Triggers iterate over a list of values.

Iter loops over a list of user-defined values. When a trigger is received in *input*, Iter moves up to the next value in the list, with wrap-around.

Parent *PyoObject*

Args

input: PyoObject Audio signal sending triggers.

choice: list of floats or PyoObjects Sequence of values over which to iterate. If a PyoObject with more than one audio streams is given, the streams will be flattened and inserted in the main list. See setChoice method for more details.

init: float, optional Initial value. Available at initialization time only. Defaults to 0.

Note: Iter will send a trigger signal when the iterator hits the last value of the list *choice*. User can retreive the trigger streams by calling obj['trig']. Useful to synchronize other processes.

```
>>> s = Server().boot()
>>> s.start()
>>> l1 = [300, 350, 400, 450, 500, 550]
>>> l2 = [300, 350, 450, 500, 550]
>>> t = CosTable([(0,0), (50,1), (250,.3), (8191,0)])
>>> met = Metro(time=.125, poly=2).play()
>>> amp = TrigEnv(met, table=t, dur=.25, mul=.3)
>>> it = Iter(met, choice=[l1, l2])
>>> si = Sine(freq=it, mul=amp).out()
```

setInput(*x*, *fadetime=0.05*)

Replace the *input* attribute.

Args

x: PyoObject New signal to process.

fadetime: float, optional Crossfade time between old and new input. Defaults to 0.05.

setChoice(*x*)

Replace the *choice* attribute.

x is a sequence of values over which to iterate. If a PyoObject with more than one audio streams is given, the streams will be flattened and inserted in the main list. For example, the choices:

[100, Randi(100,200,4), 200, Sig(250, mul=[1, 2])]

will expand to:

[100, rand_val, 200, 250, 500] # the last two are audio streams.

Args

x: list of floats or PyoObjects new *choice* attribute.

reset(*x=0*)

Resets the current count.

Args

x: int, optional Value where to reset the count. Defaults to 0.

input
PyoObject. Audio trigger signal.

choice
list of floats or PyoObjects. Possible values.

Metro

class Metro(*time=1*, *poly=1*)
Generates isochronous trigger signals.

A trigger is an audio signal with a value of 1 surrounded by 0s.

The play() method starts the metro and is not called at the object creation time.

Parent *PyoObject*

Args

time: float or PyoObject, optional Time between each trigger in seconds. Defaults to 1.

poly: int, optional Metronome polyphony. Denotes how many independent streams are generated by the metronome, allowing overlapping processes.

Available only at initialization. Defaults to 1.

Note: The out() method is bypassed. Metro's signal can not be sent to audio outs.

Metro has no *mul* and *add* attributes.

```
>>> s = Server().boot()
>>> s.start()
>>> t = CosTable([(0,0), (50,1), (250,.3), (8191,0)])
>>> met = Metro(time=.125, poly=2).play()
>>> amp = TrigEnv(met, table=t, dur=.25, mul=.3)
>>> freq = TrigRand(met, min=400, max=1000)
>>> a = Sine(freq=freq, mul=amp).out()
```

setTime(*x*)
Replace the *time* attribute.

Args

x: float or PyoObject New *time* attribute.

out(*chnl=0*, *inc=1*, *dur=0*, *delay=0*)
Start processing and send samples to audio output beginning at *chnl*.

This method returns *self*, allowing it to be applied at the object creation.

Args

chnl: int, optional Physical output assigned to the first audio stream of the object. Defaults to 0.

inc: int, optional Output channel increment value. Defaults to 1.

dur: float, optional Duration, in seconds, of the object's activation. The default is 0 and means infinite duration.

delay: float, optional Delay, in seconds, before the object's activation. Defaults to 0.

If *chnl* >= 0, successive streams increment the output number by *inc* and wrap around the global number of channels.

If *chnl* is negative, streams begin at 0, increment the output number by *inc* and wrap around the global number of channels. Then, the list of streams is scrambled.

If *chnl* is a list, successive values in the list will be assigned to successive streams.

setMul(*x*)
Replace the *mul* attribute.

Args

x: float or PyoObject New *mul* attribute.

setAdd(*x*)
Replace the *add* attribute.

Args

x: float or PyoObject New *add* attribute.

setSub(*x*)
Replace and inverse the *add* attribute.

Args

x: float or PyoObject New inversed *add* attribute.

setDiv(*x*)
Replace and inverse the *mul* attribute.

Args

x: float or PyoObject New inversed *mul* attribute.

ctrl(*map_list=None*, *title=None*, *wxnoserver=False*)
Opens a sliders window to control the parameters of the object. Only parameters that can be set to a PyoObject are allowed to be mapped on a slider.

If a list of values are given to a parameter, a multisliders will be used to control each stream independently.

Args

map_list: list of SLMap objects, optional Users defined set of parameters scaling. There is default scaling for each object that accept *ctrl* method.

title: string, optional Title of the window. If none is provided, the name of the class is used.

wxnoserver: boolean, optional With wxPython graphical toolkit, if True, tells the interpreter that there will be no server window.

If *wxnoserver* is set to True, the interpreter will not wait for the server GUI before showing the controller window.

time
float or PyoObject. Time between each trigger in seconds.

NextTrig

class NextTrig(*input*, *input2*, *mul=1*, *add=0*)

A trigger in the second stream opens a gate only for the next one in the first stream.

When the gate is opened by a trigger in *input2* signal, the next trigger in *input* signal is allowed to pass and automatically closes the gate.

Parent *PyoObject*

Args

input: PyoObject Trigger signal. Trigger stream waiting for the gate to be opened.

input2: PyoObject Trigger signal. Trigger stream opening the gate.

Note: The *input* signal is evaluated before the *input2* signal, so it's safe to send triggers in both inputs at the same time and wait for the next one.

```
>>> s = Server().boot()
>>> s.start()
>>> mid = Urn(max=4, freq=4, add=60)
>>> sigL = SineLoop(freq=MToF(mid), feedback=.08, mul=0.3).out()
>>> first = NextTrig(Change(mid), mid["trig"])
>>> amp = TrigExpseg(first, [(0,0),(.01,.25),(1,0)])
>>> sigR = SineLoop(midiToHz(84), feedback=0.05, mul=amp).out(1)
```

setInput(*x*, *fadetime=0.05*)

Replace the *input* attribute.

Args

x: PyoObject New signal to process.

fadetime: float, optional Crossfade time between old and new input. Default to 0.05.

setInput2(*x*, *fadetime=0.05*)

Replace the *input2* attribute.

Args

x: PyoObject New signal to process.

fadetime: float, optional Crossfade time between old and new input. Default to 0.05.

input

PyoObject. Incoming trigger stream signal.

input2

PyoObject. Trigger stream opening the gate.

Percent

class Percent(*input*, *percent=50.0*, *mul=1*, *add=0*)

Lets pass a certain percentage of the input triggers.

Percent looks at the triggers received in *input* and lets them pass *percent* of the time.

Parent *PyoObject*

Args

input: PyoObject Audio signal sending triggers.

percent: float or PyoObject, optional How much percentage of triggers to let pass, between 0 and 100. Defaults to 50.

Note: The out() method is bypassed. Percent's signal can not be sent to audio outs.

```
>>> s = Server().boot()
>>> s.start()
>>> t = CosTable([(0,0), (50,1), (250,.3), (8191,0)])
>>> met = Metro(time=.125, poly=2).play()
>>> trig = Percent(met, percent=50)
>>> amp = TrigEnv(trig, table=t, dur=.25, mul=.3)
>>> fr = TrigRand(trig, min=400, max=1000)
>>> freq = Port(fr, risetime=0.001, falltime=0.001)
>>> a = Sine(freq=freq, mul=amp).out()
```

out (*chnl=0*, *inc=1*, *dur=0*, *delay=0*)

Start processing and send samples to audio output beginning at *chnl*.

This method returns *self*, allowing it to be applied at the object creation.

Args

chnl: int, optional Physical output assigned to the first audio stream of the object. Defaults to 0.

inc: int, optional Output channel increment value. Defaults to 1.

dur: float, optional Duration, in seconds, of the object's activation. The default is 0 and means infinite duration.

delay: float, optional Delay, in seconds, before the object's activation. Defaults to 0.

If *chnl* >= 0, successive streams increment the output number by *inc* and wrap around the global number of channels.

If *chnl* is negative, streams begin at 0, increment the output number by *inc* and wrap around the global number of channels. Then, the list of streams is scrambled.

If *chnl* is a list, successive values in the list will be assigned to successive streams.

setInput (*x*, *fadetime=0.05*)

Replace the *input* attribute.

Args

x: PyoObject New signal to process.

fadetime: float, optional Crossfade time between old and new input. Defaults to 0.05.

setPercent (*x*)

Replace the *percent* attribute.

Args

x: float or PyoObject new *percent* attribute.

ctrl (*map_list=None*, *title=None*, *wxnoserver=False*)

Opens a sliders window to control the parameters of the object. Only parameters that can be set to a PyoObject are allowed to be mapped on a slider.

If a list of values are given to a parameter, a multisliders will be used to control each stream independently.

Args

map_list: list of SLMap objects, optional Users defined set of parameters scaling. There is default scaling for each object that accept *ctrl* method.

title: string, optional Title of the window. If none is provided, the name of the class is used.

wxnoserver: boolean, optional With wxPython graphical toolkit, if True, tells the interpreter that there will be no server window.

If *wxnoserver* is set to True, the interpreter will not wait for the server GUI before showing the controller window.

input
PyoObject. Audio signal.

percent
float or PyoObject. Percentage value.

Select

class Select(*input*, *value=0*, *mul=1*, *add=0*)
Sends trigger on matching integer values.

Select takes in input an audio signal containing integer numbers and sends a trigger when the input matches *value* parameter. This object is especially designed to be used with Counter object.

Parent *PyoObject*

Args

input: PyoObject Audio signal. Must contains integer numbers.

value: int, optional Value to be matched to send a trigger. Defaults to 0.

Note: The out() method is bypassed. Select's signal can not be sent to audio outs.

See also:

Counter

```
>>> s = Server().boot()
>>> s.start()
>>> env = HannTable()
>>> m = Metro(.125, poly=2).play()
>>> te = TrigEnv(m, table=env, dur=.2, mul=.2)
>>> c = Counter(m, min=0, max=4)
>>> se = Select(c, 0)
>>> tr = TrigRand(se, 400, 600)
>>> a = Sine(freq=tr, mul=te).out()
```

out(*chnl=0*, *inc=1*, *dur=0*, *delay=0*)
Start processing and send samples to audio output beginning at *chnl*.

This method returns *self*, allowing it to be applied at the object creation.

Args

chnl: int, optional Physical output assigned to the first audio stream of the object. Defaults to 0.

inc: int, optional Output channel increment value. Defaults to 1.

dur: float, optional Duration, in seconds, of the object's activation. The default is 0 and means infinite duration.

delay: float, optional Delay, in seconds, before the object's activation. Defaults to 0.

If *chnl* >= 0, successive streams increment the output number by *inc* and wrap around the global number of channels.

If *chnl* is negative, streams begin at 0, increment the output number by *inc* and wrap around the global number of channels. Then, the list of streams is scrambled.

If *chnl* is a list, successive values in the list will be assigned to successive streams.

setInput(*x*, *fadetime=0.05*)
Replace the *input* attribute.

Args

x: PyoObject New signal to process.

fadetime: float, optional Crossfade time between old and new input. Defaults to 0.05.

setValue(*x*)
Replace the *value* attribute.

Args

x: int new *value* attribute.

ctrl(*map_list=None*, *title=None*, *wxnoserver=False*)
Opens a sliders window to control the parameters of the object. Only parameters that can be set to a PyoObject are allowed to be mapped on a slider.

If a list of values are given to a parameter, a multisliders will be used to control each stream independently.

Args

map_list: list of SLMap objects, optional Users defined set of parameters scaling. There is default scaling for each object that accept *ctrl* method.

title: string, optional Title of the window. If none is provided, the name of the class is used.

wxnoserver: boolean, optional With wxPython graphical toolkit, if True, tells the interpreter that there will be no server window.

If *wxnoserver* is set to True, the interpreter will not wait for the server GUI before showing the controller window.

input
PyoObject. Audio signal.

value
int. Matching value.

Seq

class Seq(*time=1, seq=[1], poly=1, onlyonce=False, speed=1*)
Generates a rhythmic sequence of trigger signals.

A trigger is an audio signal with a value of 1 surrounded by 0s.

The play() method starts the sequence and is not called at the object creation time.

Parent *PyoObject*

Args

time: float or PyoObject, optional Base time between each trigger in seconds. Defaults to 1.

seq: list of floats, optional Sequence of beat durations in time's unit. Defaults to [1].

poly: int, optional Seq polyphony. Denotes how many independent streams are generated by the metronome, allowing overlapping processes.

Available only at initialization. Defaults to 1.

onlyonce: boolean, optional If True, the sequence will play only once and automatically stop. Defaults to False.

speed: float or PyoObject, optional Continuous speed factor. This factor multiplies the speed of the internal timer continuously. It can be useful to create time deceleration or acceleration. Defaults to 1.

Note: The out() method is bypassed. Seq's signal can not be sent to audio outs.

Seq has no *mul* and *add* attributes.

```
>>> s = Server().boot()
>>> s.start()
>>> env = CosTable([(0,0),(300,1),(1000,.3),(8191,0)])
>>> seq = Seq(time=.125, seq=[2,1,1,2], poly=2).play()
>>> tr = TrigRand(seq, min=250, max=500, port=.005)
>>> amp = TrigEnv(seq, table=env, dur=.25, mul=.25)
>>> a = SineLoop(tr, feedback=0.07, mul=amp).out()
```

setTime(*x*)

Replace the *time* attribute.

Args

x: float or PyoObject New *time* attribute.

setSpeed(*x*)

Replace the *speed* attribute.

Args

x: float or PyoObject New *speed* attribute.

setSeq(*x*)

Replace the *seq* attribute.

Args

x: list of floats New *seq* attribute.

setOnlyonce(*x*)

Replace the *onlyonce* attribute.

Args

x: boolean New *onlyonce* attribute.

out(*chnl=0*, *inc=1*, *dur=0*, *delay=0*)
Start processing and send samples to audio output beginning at *chnl*.

This method returns *self*, allowing it to be applied at the object creation.

Args

chnl: int, optional Physical output assigned to the first audio stream of the object. Defaults to 0.

inc: int, optional Output channel increment value. Defaults to 1.

dur: float, optional Duration, in seconds, of the object's activation. The default is 0 and means infinite duration.

delay: float, optional Delay, in seconds, before the object's activation. Defaults to 0.

If *chnl* >= 0, successive streams increment the output number by *inc* and wrap around the global number of channels.

If *chnl* is negative, streams begin at 0, increment the output number by *inc* and wrap around the global number of channels. Then, the list of streams is scrambled.

If *chnl* is a list, successive values in the list will be assigned to successive streams.

setMul(*x*)
Replace the *mul* attribute.

Args

x: float or PyoObject New *mul* attribute.

setAdd(*x*)
Replace the *add* attribute.

Args

x: float or PyoObject New *add* attribute.

setSub(*x*)
Replace and inverse the *add* attribute.

Args

x: float or PyoObject New inversed *add* attribute.

setDiv(*x*)
Replace and inverse the *mul* attribute.

Args

x: float or PyoObject New inversed *mul* attribute.

ctrl(*map_list=None*, *title=None*, *wxnoserver=False*)
Opens a sliders window to control the parameters of the object. Only parameters that can be set to a PyoObject are allowed to be mapped on a slider.

If a list of values are given to a parameter, a multisliders will be used to control each stream independently.

Args

map_list: list of SLMap objects, optional Users defined set of parameters scaling. There is default scaling for each object that accept *ctrl* method.

title: string, optional Title of the window. If none is provided, the name of the class is used.

wxnoserver: boolean, optional With wxPython graphical toolkit, if True, tells the interpreter that there will be no server window.

If *wxnoserver* is set to True, the interpreter will not wait for the server GUI before showing the controller window.

time
float or PyoObject. Base time between each trigger in seconds.

speed
float or PyoObject. Continuous speed factor.

seq
List of floats. Sequence of beat durations in time's unit.

onlyonce
boolean. Trigger the sequence only once.

Thresh

class Thresh(*input*, *threshold=0.0*, *dir=0*, *mul=1*, *add=0*)
Informs when a signal crosses a threshold.

Thresh sends a trigger when a signal crosses a threshold. The *dir* parameter can be used to set the crossing mode, down-up, up-down, or both.

Parent *PyoObject*

Args

input: PyoObject Audio signal sending triggers.

threshold: float or PyoObject, optional Threshold value. Defaults to 0.

dir: int {0, 1, 2}, optional

There are three modes of using Thresh:

0. **down-up (default)** sends a trigger when current value is higher than the threshold, while old value was equal to or lower than the threshold.
1. **up-down** sends a trigger when current value is lower than the threshold, while old value was equal to or higher than the threshold.
2. **both direction** sends a trigger in both the two previous cases.

Note: The out() method is bypassed. Thresh's signal can not be sent to audio outs.

```
>>> s = Server().boot()
>>> s.start()
>>> a = Phasor(1)
>>> b = Thresh(a, threshold=[0.25, 0.5, 0.66], dir=0)
>>> t = LinTable([(0,0), (50,1), (250,.3), (8191,0)])
>>> env = TrigEnv(b, table=t, dur=.5, mul=.3)
>>> sine = Sine(freq=[500,600,700], mul=env).out()
```

out(*chnl=0*, *inc=1*, *dur=0*, *delay=0*)
Start processing and send samples to audio output beginning at *chnl*.

This method returns *self*, allowing it to be applied at the object creation.

Args

chnl: int, optional Physical output assigned to the first audio stream of the object. Defaults to 0.

inc: int, optional Output channel increment value. Defaults to 1.

dur: float, optional Duration, in seconds, of the object's activation. The default is 0 and means infinite duration.

delay: float, optional Delay, in seconds, before the object's activation. Defaults to 0.

If *chnl* >= 0, successive streams increment the output number by *inc* and wrap around the global number of channels.

If *chnl* is negative, streams begin at 0, increment the output number by *inc* and wrap around the global number of channels. Then, the list of streams is scrambled.

If *chnl* is a list, successive values in the list will be assigned to successive streams.

setInput(*x*, *fadetime=0.05*)
Replace the *input* attribute.

Args

x: PyoObject New signal to process.

fadetime: float, optional Crossfade time between old and new input. Defaults to 0.05.

setThreshold(*x*)
Replace the *threshold* attribute.

Args

x: float or PyoObject new *threshold* attribute.

setDir(*x*)
Replace the *dir* attribute.

Args

x: int {0, 1, 2} new *dir* attribute.

input
PyoObject. Audio signal.

threshold
float or PyoObject. Threshold value.

dir
int. User mode.

Timer

class Timer(*input*, *input2*, *mul=1*, *add=0*)
Reports elapsed time between two trigs.

A trigger in *input2* signal starts an internal timer. The next trigger in *input* signal stops it and reports the elapsed time between the two triggers. Useful for filtering triggers that are too close to each other.

Parent *PyoObject*

Args

input: PyoObject Trigger signal. Stops the timer and reports elapsed time.

input2: PyoObject Trigger signal. Starts the timer if not already started.

Note: The *input* signal is evaluated before the *input2* signal, so it's safe to stop and start the timer with the same trigger signal.

```
>>> s = Server().boot()
>>> s.start()
>>> cl = Cloud(density=20, poly=2).play()
>>> ti = Timer(cl, cl)
>>> # Minimum waiting time before a new trig
>>> cp = Compare(ti, comp=.05, mode=">")
>>> trig = cl * cp
>>> amp = TrigEnv(trig, table=HannTable(), dur=.05, mul=.25)
>>> freq = TrigChoice(trig, choice=[100,150,200,250,300,350,400])
>>> a = LFO(freq=freq, type=2, mul=amp).out()
```

setInput(*x*, *fadetime=0.05*)
Replace the *input* attribute.

Args

x: PyoObject New signal to process.

fadetime: float, optional Crossfade time between old and new input. Default to 0.05.

setInput2(*x*, *fadetime=0.05*)
Replace the *input2* attribute.

Args

x: PyoObject New signal to process.

fadetime: float, optional Crossfade time between old and new input. Default to 0.05.

input
PyoObject. Timer stop signal.

input2
PyoObject. Timer start signal.

Trig

class Trig
Sends one trigger.

A trigger is an audio signal with a value of 1 surrounded by 0s.

Trig sends a trigger each time it's play() method is called.

Parent *PyoObject*

Note: The out() method is bypassed. Trig's signal can not be sent to audio outs.

Trig has no *mul* and *add* attributes.

```
>>> s = Server().boot()
>>> s.start()
>>> a = Trig()
>>> env = HannTable()
>>> tenv = TrigEnv(a, table=env, dur=5, mul=.3)
>>> n = Noise(tenv).out()
```

out (*chnl=0*, *inc=1*, *dur=0*, *delay=0*)

Start processing and send samples to audio output beginning at *chnl*.

This method returns *self*, allowing it to be applied at the object creation.

Args

chnl: int, optional Physical output assigned to the first audio stream of the object. Defaults to 0.

inc: int, optional Output channel increment value. Defaults to 1.

dur: float, optional Duration, in seconds, of the object's activation. The default is 0 and means infinite duration.

delay: float, optional Delay, in seconds, before the object's activation. Defaults to 0.

If *chnl* >= 0, successive streams increment the output number by *inc* and wrap around the global number of channels.

If *chnl* is negative, streams begin at 0, increment the output number by *inc* and wrap around the global number of channels. Then, the list of streams is scrambled.

If *chnl* is a list, successive values in the list will be assigned to successive streams.

setMul (*x*)

Replace the *mul* attribute.

Args

x: float or PyoObject New *mul* attribute.

setAdd (*x*)

Replace the *add* attribute.

Args

x: float or PyoObject New *add* attribute.

setSub (*x*)

Replace and inverse the *add* attribute.

Args

x: float or PyoObject New inversed *add* attribute.

setDiv (*x*)

Replace and inverse the *mul* attribute.

Args

x: float or PyoObject New inversed *mul* attribute.

TrigBurst

class TrigBurst(*input*, *time=0.25*, *count=10*, *expand=1.0*, *ampfade=1.0*, *poly=1*)

Generates a time/amplitude expandable trigger pattern.

A trigger is an audio signal with a value of 1 surrounded by 0s.

When TrigBurst receives a trigger in its *input* argument, it starts to output *count* triggers with a variable delay between each trigger of the pattern. If *expand* is less than 1.0, the delay becomes shorter, if it is greater than 1.0, the delay becomes longer.

Parent *PyoObject*

Args

input: PyoObject Input signal sending triggers.

time: float or PyoObject, optional Base time, in seconds, between each trig of the serie. Defaults to 0.25.

count: int, optional Number of trigs generated (length of the serie). Defaults to 10.

expand: float, optional Timing power serie factor. Each delay before the next trig is the current delay (starting with *time*) times *expand* factor. Defaults to 1.0.

ampfade: float, optional Amplitude power serie factor. Each amplitude in the serie is the current amplitude (starting at 1) times *ampfade* factor. Defaults to 1.0.

poly: int, optional Voice polyphony. Denotes how many independent streams are generated by the object, allowing overlapping processes.

Available only at initialization. Defaults to 1.

Note: TrigBurst outputs many signals identified with a string between brackets:

obj['tap'] returns audio stream of the current tap of the serie.
obj['amp'] returns audio stream of the current beat amplitude.
obj['dur'] returns audio stream of the current beat duration in seconds.
obj['end'] returns audio stream with a trigger just before the end of the serie.

obj without brackets returns the generated trigger stream of the serie.

The out() method is bypassed. TrigBurst's signal can not be sent to audio outs.

TrigBurst has no *mul* and *add* attributes.

```
>>> s = Server().boot()
>>> s.start()
>>> env = CosTable([(0,0), (100,0.5), (500, 0.3), (4096,0.3), (8192,0)])
>>> m = Metro(2).play()
>>> tb = TrigBurst(m, time=0.15, count=[15,20], expand=[0.92,0.9], ampfade=0.85)
>>> amp = TrigEnv(tb, env, dur=tb["dur"], mul=tb["amp"]*0.3)
>>> a = Sine([800,600], mul=amp)
>>> rev = STRev(a, inpos=[0,1], revtime=1.5, cutoff=5000, bal=0.1).out()
```

get(*identifier='amp'*, *all=False*)

Return the first sample of the current buffer as a float.

Can be used to convert audio stream to usable Python data.

"tap", "amp" or "dur" must be given to *identifier* to specify which stream to get value from.

Args

identifier: string {"tap", "amp", "dur"} Address string parameter identifying audio stream. Defaults to "amp".

all: boolean, optional If True, the first value of each object's stream will be returned as a list.

If False, only the value of the first object's stream will be returned as a float.

setInput (*x*, *fadetime=0.05*)

Replace the *input* attribute.

Args

x: PyoObject New signal to process.

fadetime: float, optional Crossfade time between old and new input. Defaults to 0.05.

setTime (*x*)

Replace the *time* attribute.

Args

x: float or PyoObject New *time* attribute.

setCount (*x*)

Replace the *count* attribute.

Args

x: int New *count* attribute.

setExpand (*x*)

Replace the *expand* attribute.

Args

x: float New *expand* attribute.

setAmpfade (*x*)

Replace the *ampfade* attribute.

Args

x: float New *ampfade* attribute.

play (*dur=0*, *delay=0*)

Start processing without sending samples to output. This method is called automatically at the object creation.

This method returns *self*, allowing it to be applied at the object creation.

Args

dur: float, optional Duration, in seconds, of the object's activation. The default is 0 and means infinite duration.

delay: float, optional Delay, in seconds, before the object's activation. Defaults to 0.

stop (*wait=0*)

Stop processing.

This method returns *self*, allowing it to be applied at the object creation.

Args

wait: float, optional Delay, in seconds, before the process is actually stopped. If autoStartChildren is activated in the Server, this value is propagated to the children objects. Defaults to 0.

Note: if the method setStopDelay(x) was called before calling stop(wait) with a positive *wait* value, the *wait* value won't overwrite the value given to setStopDelay for the current object, but will be the one propagated to children objects. This allow to set a waiting time for a specific object with setStopDelay whithout changing the global delay time given to the stop method.

Fader and Adsr objects ignore waiting time given to the stop method because they already implement a delayed processing triggered by the stop call.

---

out (*chnl=0*, *inc=1*, *dur=0*, *delay=0*)

Start processing and send samples to audio output beginning at *chnl*.

This method returns *self*, allowing it to be applied at the object creation.

Args

chnl: int, optional Physical output assigned to the first audio stream of the object. Defaults to 0.

inc: int, optional Output channel increment value. Defaults to 1.

dur: float, optional Duration, in seconds, of the object's activation. The default is 0 and means infinite duration.

delay: float, optional Delay, in seconds, before the object's activation. Defaults to 0.

If *chnl* >= 0, successive streams increment the output number by *inc* and wrap around the global number of channels.

If *chnl* is negative, streams begin at 0, increment the output number by *inc* and wrap around the global number of channels. Then, the list of streams is scrambled.

If *chnl* is a list, successive values in the list will be assigned to successive streams.

setMul (*x*)

Replace the *mul* attribute.

Args

x: float or PyoObject New *mul* attribute.

setAdd (*x*)

Replace the *add* attribute.

Args

x: float or PyoObject New *add* attribute.

setSub (*x*)

Replace and inverse the *add* attribute.

Args

x: float or PyoObject New inversed *add* attribute.

setDiv (*x*)

Replace and inverse the *mul* attribute.

Args

x: float or PyoObject New inversed *mul* attribute.

ctrl(*map_list=None*, *title=None*, *wxnoserver=False*)
Opens a sliders window to control the parameters of the object. Only parameters that can be set to a PyoObject are allowed to be mapped on a slider.

If a list of values are given to a parameter, a multisliders will be used to control each stream independently.

Args

map_list: list of SLMap objects, optional Users defined set of parameters scaling. There is default scaling for each object that accept *ctrl* method.

title: string, optional Title of the window. If none is provided, the name of the class is used.

wxnoserver: boolean, optional With wxPython graphical toolkit, if True, tells the interpreter that there will be no server window.

If *wxnoserver* is set to True, the interpreter will not wait for the server GUI before showing the controller window.

input
PyoObject. Audio trigger signal.

time
float or PyoObject. Base time, in seconds, between each trig.

count
int. Number of triggers in the generated serie.

expand
float. Time's power expansion factor.

ampfade
float. Amplitude's power expansion factor.

TrigChoice

class TrigChoice(*input*, *choice*, *port=0.0*, *init=0.0*, *mul=1*, *add=0*)
Random generator from user's defined values.

TrigChoice chooses randomly a new value in list *choice* each time it receives a trigger in its *input* parameter. The value is kept until the next trigger.

Parent *PyoObject*

Args

input: PyoObject Audio signal sending triggers.

choice: list of floats Possible values for the random generation.

port: float, optional Portamento. Time to reach a new value. Defaults to 0.

init: float, optional Initial value. Available at initialization time only. Defaults to 0.

```
>>> s = Server().boot()
>>> s.start()
>>> t = CosTable([(0,0), (50,1), (250,.3), (8191,0)])
>>> met = Metro(.125, poly=2).play()
>>> freq = TrigChoice(met, [300, 350, 400, 450, 500, 550])
```

(continues on next page)

(continued from previous page)

```
>>> amp = TrigEnv(met, table=t, dur=.25, mul=.3)
>>> a = Sine(freq=freq, mul=amp).out()
```

setInput(*x*, *fadetime=0.05*)
Replace the *input* attribute.

Args

x: PyoObject New signal to process.

fadetime: float, optional Crossfade time between old and new input. Defaults to 0.05.

setChoice(*x*)
Replace the *choice* attribute.

Args

x: list of floats new *choice* attribute.

setPort(*x*)
Replace the *port* attribute.

Args

x: float new *port* attribute.

input
PyoObject. Audio trigger signal.

choice
list of floats. Possible values.

port
float. Ramp time.

TrigEnv

class TrigEnv(*input*, *table*, *dur=1*, *interp=2*, *mul=1*, *add=0*)
Envelope reader generator.

TrigEnv starts reading an envelope in *dur* seconds each time it receives a trigger in its *input* parameter.

Parent *PyoObject*

Args

input: PyoObject Audio signal sending triggers.

table: PyoTableObject Table containing the envelope.

dur: float or PyoObject, optional Duration in seconds of the envelope. Defaults to 1.

interp: int, optional

Choice of the interpolation method. Defaults to 2.

1. no interpolation
2. linear
3. cosinus
4. cubic

Note: TrigEnv will send a trigger signal at the end of the playback. User can retreive the trigger streams by calling obj['trig']. Useful to synchronize other processes.

```
>>> s = Server().boot()
>>> s.start()
>>> env = HannTable()
>>> m = Metro(.125, poly=2).play()
>>> tr = TrigRand(m, 400, 600)
>>> te = TrigEnv(m, table=env, dur=.25, mul=.2)
>>> a = Sine(tr, mul=te).out()
```

setInput(*x*, *fadetime=0.05*)
Replace the *input* attribute.

Args

x: PyoObject New signal to process.

fadetime: float, optional Crossfade time between old and new input. Defaults to 0.05.

setTable(*x*)
Replace the *table* attribute.

Args

x: PyoTableObject new *table* attribute.

setDur(*x*)
Replace the *dur* attribute.

Args

x: float or PyoObject new *dur* attribute.

setInterp(*x*)
Replace the *interp* attribute.

Args

x: int {1, 2, 3, 4} new *interp* attribute.

ctrl(*map_list=None*, *title=None*, *wxnoserver=False*)
Opens a sliders window to control the parameters of the object. Only parameters that can be set to a PyoObject are allowed to be mapped on a slider.

If a list of values are given to a parameter, a multisliders will be used to control each stream independently.

Args

map_list: list of SLMap objects, optional Users defined set of parameters scaling. There is default scaling for each object that accept *ctrl* method.

title: string, optional Title of the window. If none is provided, the name of the class is used.

wxnoserver: boolean, optional With wxPython graphical toolkit, if True, tells the interpreter that there will be no server window.

If *wxnoserver* is set to True, the interpreter will not wait for the server GUI before showing the controller window.

input
PyoObject. Audio trigger signal.

table
PyoTableObject. Envelope table.

dur
float or PyoObject. Duration in seconds.

interp
int {1, 2, 3, 4}, Interpolation method.

TrigExpseg

class TrigExpseg(*input*, *list*, *exp=10*, *inverse=True*, *mul=1*, *add=0*)
Exponential segments trigger.

TrigExpseg starts reading break-points exponential segments each time it receives a trigger in its *input* parameter.

Parent *PyoObject*

Args

input: PyoObject Audio signal sending triggers.

list: list of tuples Points used to construct the line segments. Each tuple is a new point in the form (time, value).

Times are given in seconds and must be in increasing order.

exp: float, optional Exponent factor. Used to control the slope of the curves. Defaults to 10.

inverse: boolean, optional If True, downward slope will be inversed. Useful to create biexponential curves. Defaults to True.

Note: TrigExpseg will send a trigger signal at the end of the playback. User can retreive the trigger streams by calling obj['trig']. Useful to synchronize other processes.

The out() method is bypassed. TrigExpseg's signal can not be sent to audio outs.

```
>>> s = Server().boot()
>>> s.start()
>>> m = Metro(time=0.5, poly=2).play()
>>> pit = TrigExpseg(m, [(0,1000),(.25,1300),(.5,1000),(1,1000)])
>>> a = Sine(pit, mul=.2).out()
```

out(*chnl=0*, *inc=1*, *dur=0*, *delay=0*)
Start processing and send samples to audio output beginning at *chnl*.

This method returns *self*, allowing it to be applied at the object creation.

Args

chnl: int, optional Physical output assigned to the first audio stream of the object. Defaults to 0.

inc: int, optional Output channel increment value. Defaults to 1.

dur: float, optional Duration, in seconds, of the object's activation. The default is 0 and means infinite duration.

delay: float, optional Delay, in seconds, before the object's activation. Defaults to 0.

If *chnl* >= 0, successive streams increment the output number by *inc* and wrap around the global number of channels.

If *chnl* is negative, streams begin at 0, increment the output number by *inc* and wrap around the global number of channels. Then, the list of streams is scrambled.

If *chnl* is a list, successive values in the list will be assigned to successive streams.

setInput(*x*, *fadetime=0.05*)
Replace the *input* attribute.

Args

x: PyoObject New signal to process.

fadetime: float, optional Crossfade time between old and new input. Defaults to 0.05.

setList(*x*)
Replace the *list* attribute.

Args

x: list of tuples new *list* attribute.

setExp(*x*)
Replace the *exp* attribute.

Args

x: float new *exp* attribute.

setInverse(*x*)
Replace the *inverse* attribute.

Args

x: boolean new *inverse* attribute.

replace(*x*)
Alias for *setList* method.

Args

x: list of tuples new *list* attribute.

ctrl(*map_list=None*, *title=None*, *wxnoserver=False*)
Opens a sliders window to control the parameters of the object. Only parameters that can be set to a PyoObject are allowed to be mapped on a slider.

If a list of values are given to a parameter, a multisliders will be used to control each stream independently.

Args

map_list: list of SLMap objects, optional Users defined set of parameters scaling. There is default scaling for each object that accept *ctrl* method.

title: string, optional Title of the window. If none is provided, the name of the class is used.

wxnoserver: boolean, optional With wxPython graphical toolkit, if True, tells the interpreter that there will be no server window.

If *wxnoserver* is set to True, the interpreter will not wait for the server GUI before showing the controller window.

graph(*xlen=None*, *yrange=None*, *title=None*, *wxnoserver=False*)
Opens a grapher window to control the shape of the envelope.

When editing the grapher with the mouse, the new set of points will be send to the object on mouse up.

Ctrl+C with focus on the grapher will copy the list of points to the clipboard, giving an easy way to insert the new shape in a script.

Args

xlen: float, optional Set the maximum value of the X axis of the graph. If None, the maximum value is retrieve from the current list of points. Defaults to None.

yrange: tuple, optional Set the min and max values of the Y axis of the graph. If None, min and max are retrieve from the current list of points. Defaults to None.

title: string, optional Title of the window. If none is provided, the name of the class is used.

wxnoserver: boolean, optional With wxPython graphical toolkit, if True, tells the interpreter that there will be no server window.

If *wxnoserver* is set to True, the interpreter will not wait for the server GUI before showing the controller window.

input
PyoObject. Audio trigger signal.

list
list of tuples. Points used to construct the line segments.

exp
float. Exponent factor.

inverse
boolean. Inversion of downward slope.

TrigFunc

class TrigFunc(*input*, *function*, *arg=None*)
Python function callback.

TrigFunc calls the function given at parameter *function* each time it receives a trigger in its *input* parameter.

Parent *PyoObject*

Args

input: PyoObject Audio signal sending triggers.

function: Python callable (function or method) Function to be called.

arg: anything, optional Argument sent to the function's call. If None, the function will be called without argument. Defaults to None.

Note: The out() method is bypassed. TrigFunc's signal can not be sent to audio outs.

TrigFunc has no *mul* and *add* attributes.

```
>>> s = Server().boot()
>>> s.start()
>>> f = Fader(fadein=.005, fadeout=.1, dur=.12, mul=.2)
>>> a = SineLoop(midiToHz([60,60]), feedback=0.05, mul=f).out()
>>> c = 0.0
>>> def count():
...     global c
...     freq = midiToHz(round(c) + 60)
...     a.freq = [freq, freq*0.995]
...     c += 1.77
...     if c > 13: c = 0
...     f.play()
>>> m = Metro(.125).play()
>>> tf = TrigFunc(m, count)
```

out (*chnl=0*, *inc=1*, *dur=0*, *delay=0*)

Start processing and send samples to audio output beginning at *chnl*.

This method returns *self*, allowing it to be applied at the object creation.

Args

chnl: int, optional Physical output assigned to the first audio stream of the object. Defaults to 0.

inc: int, optional Output channel increment value. Defaults to 1.

dur: float, optional Duration, in seconds, of the object's activation. The default is 0 and means infinite duration.

delay: float, optional Delay, in seconds, before the object's activation. Defaults to 0.

If *chnl* >= 0, successive streams increment the output number by *inc* and wrap around the global number of channels.

If *chnl* is negative, streams begin at 0, increment the output number by *inc* and wrap around the global number of channels. Then, the list of streams is scrambled.

If *chnl* is a list, successive values in the list will be assigned to successive streams.

setMul (*x*)

Replace the *mul* attribute.

Args

x: float or PyoObject New *mul* attribute.

setAdd (*x*)

Replace the *add* attribute.

Args

x: float or PyoObject New *add* attribute.

setInput (*x*, *fadetime=0.05*)

Replace the *input* attribute.

Args

x: PyoObject New signal to process.

fadetime: float, optional Crossfade time between old and new input. Defaults to 0.05.

setFunction(*x*)
Replace the *function* attribute.

Args

x: Python callable (function or method) new *function* attribute.

setArg(*x*)
Replace the *arg* attribute.

Args

x: Anything new *arg* attribute.

input
PyoObject. Audio trigger signal.

function
Python callable. Function to be called.

arg
Anything. Callable's argument.

TrigLinseg

class TrigLinseg(*input*, *list*, *mul=1*, *add=0*)
Line segments trigger.

TrigLinseg starts reading a break-points line segments each time it receives a trigger in its *input* parameter.

Parent *PyoObject*

Args

input: PyoObject Audio signal sending triggers.

list: list of tuples Points used to construct the line segments. Each tuple is a new point in the form (time, value).

Times are given in seconds and must be in increasing order.

Note: TrigLinseg will send a trigger signal at the end of the playback. User can retreive the trigger streams by calling obj['trig']. Useful to synchronize other processes.

The out() method is bypassed. TrigLinseg's signal can not be sent to audio outs.

```
>>> s = Server().boot()
>>> s.start()
>>> m = Metro(time=1, poly=2).play()
>>> pit = TrigLinseg(m, [(0,1000),(.1,1300),(.2,900),(.3,1000),(2,1000)])
>>> a = Sine(pit, mul=.2).out()
```

out(*chnl=0*, *inc=1*, *dur=0*, *delay=0*)
Start processing and send samples to audio output beginning at *chnl*.

This method returns *self*, allowing it to be applied at the object creation.

Args

chnl: int, optional Physical output assigned to the first audio stream of the object. Defaults to 0.

inc: int, optional Output channel increment value. Defaults to 1.

dur: float, optional Duration, in seconds, of the object's activation. The default is 0 and means infinite duration.

delay: float, optional Delay, in seconds, before the object's activation. Defaults to 0.

If *chnl* >= 0, successive streams increment the output number by *inc* and wrap around the global number of channels.

If *chnl* is negative, streams begin at 0, increment the output number by *inc* and wrap around the global number of channels. Then, the list of streams is scrambled.

If *chnl* is a list, successive values in the list will be assigned to successive streams.

setInput (*x*, *fadetime=0.05*)
Replace the *input* attribute.

Args

x: PyoObject New signal to process.

fadetime: float, optional Crossfade time between old and new input. Defaults to 0.05.

setList (*x*)
Replace the *list* attribute.

Args

x: list of tuples new *list* attribute.

replace (*x*)
Alias for *setList* method.

Args

x: list of tuples new *list* attribute.

ctrl (*map_list=None*, *title=None*, *wxnoserver=False*)
Opens a sliders window to control the parameters of the object. Only parameters that can be set to a PyoObject are allowed to be mapped on a slider.

If a list of values are given to a parameter, a multisliders will be used to control each stream independently.

Args

map_list: list of SLMap objects, optional Users defined set of parameters scaling. There is default scaling for each object that accept *ctrl* method.

title: string, optional Title of the window. If none is provided, the name of the class is used.

wxnoserver: boolean, optional With wxPython graphical toolkit, if True, tells the interpreter that there will be no server window.

If *wxnoserver* is set to True, the interpreter will not wait for the server GUI before showing the controller window.

graph (*xlen=None*, *yrange=None*, *title=None*, *wxnoserver=False*)
Opens a grapher window to control the shape of the envelope.

When editing the grapher with the mouse, the new set of points will be send to the object on mouse up.

Ctrl+C with focus on the grapher will copy the list of points to the clipboard, giving an easy way to insert the new shape in a script.

Args

xlen: float, optional Set the maximum value of the X axis of the graph. If None, the maximum value is retrieve from the current list of points. Defaults to None.

yrange: tuple, optional Set the min and max values of the Y axis of the graph. If None, min and max are retrieve from the current list of points. Defaults to None.

title: string, optional Title of the window. If none is provided, the name of the class is used.

wxnoserver: boolean, optional With wxPython graphical toolkit, if True, tells the interpreter that there will be no server window.

If *wxnoserver* is set to True, the interpreter will not wait for the server GUI before showing the controller window.

input
PyoObject. Audio trigger signal.

list
list of tuples. Points used to construct the line segments.

TrigRand

class TrigRand(*input*, *min=0.0*, *max=1.0*, *port=0.0*, *init=0.0*, *mul=1*, *add=0*)
Pseudo-random number generator.

TrigRand generates a pseudo-random number between *min* and *max* values each time it receives a trigger in its *input* parameter. The value is kept until the next trigger.

Parent *PyoObject*

Args

input: PyoObject Audio signal sending triggers.

min: float or PyoObject, optional Minimum value for the random generation. Defaults to 0.

max: float or PyoObject, optional Maximum value for the random generation. Defaults to 1.

port: float, optional Portamento. Time to reach a new value. Defaults to 0.

init: float, optional Initial value. Available at initialization time only. Defaults to 0.

```
>>> s = Server().boot()
>>> s.start()
>>> t = CosTable([(0,0), (50,1), (250,.3), (8191,0)])
>>> met = Metro(.125, poly=2).play()
>>> amp = TrigEnv(met, table=t, dur=.25, mul=.3)
>>> tr = TrigRand(met, 400, 600)
>>> a = Sine(tr, mul=amp).out()
```

setInput(*x*, *fadetime=0.05*)
Replace the *input* attribute.

Args

x: PyoObject New signal to process.

fadetime: float, optional Crossfade time between old and new input. Defaults to 0.05.

setMin(*x*)
Replace the *min* attribute.

Args

x: float or PyoObject new *min* attribute.

setMax(*x*)
Replace the *max* attribute.

Args

x: float or PyoObject new *max* attribute.

setPort(*x*)
Replace the *port* attribute.

Args

x: float new *port* attribute.

ctrl(*map_list=None*, *title=None*, *wxnoserver=False*)
Opens a sliders window to control the parameters of the object. Only parameters that can be set to a PyoObject are allowed to be mapped on a slider.

If a list of values are given to a parameter, a multisliders will be used to control each stream independently.

Args

map_list: list of SLMap objects, optional Users defined set of parameters scaling. There is default scaling for each object that accept *ctrl* method.

title: string, optional Title of the window. If none is provided, the name of the class is used.

wxnoserver: boolean, optional With wxPython graphical toolkit, if True, tells the interpreter that there will be no server window.

If *wxnoserver* is set to True, the interpreter will not wait for the server GUI before showing the controller window.

input
PyoObject. Audio trigger signal.

min
float or PyoObject. Minimum value.

max
float or PyoObject. Maximum value.

port
float. Ramp time.

TrigRandInt

class TrigRandInt(*input*, *max=100.0*, *mul=1*, *add=0*)
Pseudo-random integer generator.

TrigRandInt generates a pseudo-random number integer number between 0 and *max* values each time it receives a trigger in its *input* parameter. The value is kept until the next trigger.

Parent `PyoObject`

Args

input: PyoObject Audio signal sending triggers.

max: float or PyoObject, optional Maximum value for the random generation. Defaults to 100.

Note: The out() method is bypassed. TrigRandInt's signal can not be sent to audio outs.

```
>>> s = Server().boot()
>>> s.start()
>>> t = CosTable([(0,0), (50,1), (250,.3), (8191,0)])
>>> met = Metro(.125, poly=2).play()
>>> amp = TrigEnv(met, table=t, dur=.25, mul=.3)
>>> tr = TrigRandInt(met, max=10, mul=100, add=200)
>>> a = Sine(tr, mul=amp).out()
```

setInput (*x*, *fadetime=0.05*)

Replace the *input* attribute.

Args

x: PyoObject New signal to process.

fadetime: float, optional Crossfade time between old and new input. Defaults to 0.05.

setMax (*x*)

Replace the *max* attribute.

Args

x: float or PyoObject new *max* attribute.

out (*chnl=0*, *inc=1*, *dur=0*, *delay=0*)

Start processing and send samples to audio output beginning at *chnl*.

This method returns *self*, allowing it to be applied at the object creation.

Args

chnl: int, optional Physical output assigned to the first audio stream of the object. Defaults to 0.

inc: int, optional Output channel increment value. Defaults to 1.

dur: float, optional Duration, in seconds, of the object's activation. The default is 0 and means infinite duration.

delay: float, optional Delay, in seconds, before the object's activation. Defaults to 0.

If *chnl* >= 0, successive streams increment the output number by *inc* and wrap around the global number of channels.

If *chnl* is negative, streams begin at 0, increment the output number by *inc* and wrap around the global number of channels. Then, the list of streams is scrambled.

If *chnl* is a list, successive values in the list will be assigned to successive streams.

ctrl (*map_list=None*, *title=None*, *wxnoserver=False*)

Opens a sliders window to control the parameters of the object. Only parameters that can be set to a PyoObject are allowed to be mapped on a slider.

If a list of values are given to a parameter, a multisliders will be used to control each stream independently.

Args

map_list: list of SLMap objects, optional Users defined set of parameters scaling. There is default scaling for each object that accept *ctrl* method.

title: string, optional Title of the window. If none is provided, the name of the class is used.

wxnoserver: boolean, optional With wxPython graphical toolkit, if True, tells the interpreter that there will be no server window.

If *wxnoserver* is set to True, the interpreter will not wait for the server GUI before showing the controller window.

input
PyoObject. Audio trigger signal.

max
float or PyoObject. Maximum value.

TrigTableRec

class TrigTableRec(*input*, *trig*, *table*, *fadetime=0*)
TrigTableRec is for writing samples into a previously created NewTable.

See *NewTable* to create an empty table.

Each time a "trigger" is received in the *trig* input, TrigTableRec starts the recording into the table until the table is full.

Parent *PyoObject*

Args

input: PyoObject Audio signal to write in the table.

trig: PyoObject Audio signal sending triggers.

table: NewTable The table where to write samples.

fadetime: float, optional Fade time at the beginning and the end of the recording in seconds. Defaults to 0.

Note: The out() method is bypassed. TrigTableRec returns no signal.

TrigTableRec has no *mul* and *add* attributes.

TrigTableRec will send a trigger signal at the end of the recording. User can retrieve the trigger streams by calling obj['trig'].

obj['time'] outputs an audio stream of the current recording time, in samples.

See also:

NewTable, *TableRec*

```
>>> s = Server().boot()
>>> s.start()
>>> snd = SNDS_PATH + '/transparent.aif'
>>> dur = sndinfo(snd)[1]
>>> t = NewTable(length=dur)
>>> src = SfPlayer(snd, mul=.3).out()
```

(continues on next page)

(continued from previous page)

```
>>> trec = TrigTableRec(src, trig=Trig().play(), table=t)
>>> rep = TrigEnv(trec["trig"], table=t, dur=dur).out(1)
```

out (*chnl=0*, *inc=1*, *dur=0*, *delay=0*)

Start processing and send samples to audio output beginning at *chnl*.

This method returns *self*, allowing it to be applied at the object creation.

Args

chnl: int, optional Physical output assigned to the first audio stream of the object. Defaults to 0.

inc: int, optional Output channel increment value. Defaults to 1.

dur: float, optional Duration, in seconds, of the object's activation. The default is 0 and means infinite duration.

delay: float, optional Delay, in seconds, before the object's activation. Defaults to 0.

If *chnl* >= 0, successive streams increment the output number by *inc* and wrap around the global number of channels.

If *chnl* is negative, streams begin at 0, increment the output number by *inc* and wrap around the global number of channels. Then, the list of streams is scrambled.

If *chnl* is a list, successive values in the list will be assigned to successive streams.

setMul (*x*)

Replace the *mul* attribute.

Args

x: float or PyoObject New *mul* attribute.

setAdd (*x*)

Replace the *add* attribute.

Args

x: float or PyoObject New *add* attribute.

setInput (*x*, *fadetime=0.05*)

Replace the *input* attribute.

Args

x: PyoObject New signal to process.

fadetime: float, optional Crossfade time between old and new input. Defaults to 0.05.

setTrig (*x*, *fadetime=0.05*)

Replace the *trig* attribute.

Args

x: PyoObject New trigger signal.

fadetime: float, optional Crossfade time between old and new input. Defaults to 0.05.

setTable (*x*)

Replace the *table* attribute.

Args

x: NewTable new *table* attribute.

input
PyoObject. Audio signal to write in the table.

trig
PyoObject. Audio signal sending triggers.

table
NewTable. The table where to write samples.

TrigVal

class TrigVal(*input*, *value=0.0*, *init=0.0*, *mul=1*, *add=0*)
Outputs a previously defined value on a trigger signal.

Value defined at *value* argument is sent when a trigger signal is detected in input.

Parent *PyoObject*

Args

input: PyoObject Audio signal sending triggers.

value: float or PyoObject, optional Next value. Defaults to 0.

init: float, optional Initial value. Defaults to 0.

Note: The out() method is bypassed. TrigVal's signal can not be sent to audio outs.

```
>>> s = Server().boot()
>>> s.start()
>>> def newfreq():
...     val.value = (val.value + 50) % 500 + 100
>>> tr = Metro(1).play()
>>> val = TrigVal(tr, value=250)
>>> a = SineLoop(val, feedback=.1, mul=.3).out()
>>> trfunc = TrigFunc(tr, newfreq)
```

setInput(*x*, *fadetime=0.05*)
Replace the *input* attribute.

Args

x: PyoObject New signal to process.

fadetime: float, optional Crossfade time between old and new input. Defaults to 0.05.

setValue(*x*)
Replace the *value* attribute.

Args

x: float or PyoObject new *value* attribute.

out(*chnl=0*, *inc=1*, *dur=0*, *delay=0*)
Start processing and send samples to audio output beginning at *chnl*.

This method returns *self*, allowing it to be applied at the object creation.

Args

chnl: int, optional Physical output assigned to the first audio stream of the object. Defaults to 0.

inc: int, optional Output channel increment value. Defaults to 1.

dur: float, optional Duration, in seconds, of the object's activation. The default is 0 and means infinite duration.

delay: float, optional Delay, in seconds, before the object's activation. Defaults to 0.

If *chnl* >= 0, successive streams increment the output number by *inc* and wrap around the global number of channels.

If *chnl* is negative, streams begin at 0, increment the output number by *inc* and wrap around the global number of channels. Then, the list of streams is scrambled.

If *chnl* is a list, successive values in the list will be assigned to successive streams.

ctrl(*map_list=None*, *title=None*, *wxnoserver=False*)
Opens a sliders window to control the parameters of the object. Only parameters that can be set to a PyoObject are allowed to be mapped on a slider.

If a list of values are given to a parameter, a multisliders will be used to control each stream independently.

Args

map_list: list of SLMap objects, optional Users defined set of parameters scaling. There is default scaling for each object that accept *ctrl* method.

title: string, optional Title of the window. If none is provided, the name of the class is used.

wxnoserver: boolean, optional With wxPython graphical toolkit, if True, tells the interpreter that there will be no server window.

If *wxnoserver* is set to True, the interpreter will not wait for the server GUI before showing the controller window.

input
PyoObject. Audio trigger signal.

value
float or PyoObject. Next value.

TrigXnoise

class TrigXnoise(*input*, *dist=0*, *x1=0.5*, *x2=0.5*, *mul=1*, *add=0*)
Triggered X-class pseudo-random generator.

Xnoise implements a few of the most common noise distributions. A new value is generated each time the object receive a trigger in input. Each distribution generates values in the range 0 and 1.

Parent *PyoObject*

Args

input: PyoObject Audio signal sending triggers.

dist: string or int, optional Distribution type. Defaults to 0.

x1: float or PyoObject, optional First parameter. Defaults to 0.5.

x2: float or PyoObject, optional Second parameter. Defaults to 0.5.

Note:

Available distributions are:

0. uniform
1. linear minimum
2. linear maximum
3. triangular
4. exponential minimum
5. exponential maximum
6. double (bi)exponential
7. cauchy
8. weibull
9. gaussian
10. poisson
11. walker (drunk)
12. loopseg (drunk with looped segments)

Depending on the distribution, *x1* and *x2* parameters are applied as follow (names as string, or associated number can be used as *dist* parameter):

0. **uniform**
    - x1: not used
    - x2: not used
1. **linear_min**
    - x1: not used
    - x2: not used
2. **linear_max**
    - x1: not used
    - x2: not used
3. **triangle**
    - x1: not used
    - x2: not used
4. **expon_min**
    - x1: slope {0 = no slope -> 10 = sharp slope}
    - x2: not used
5. **expon_max**
    - x1: slope {0 = no slope -> 10 = sharp slope}
    - x2: not used
6. **biexpon**

- x1: bandwidth {0 = huge bandwidth -> 10 = narrow bandwidth}
- x2: not used

7. **cauchy**
   - x1: bandwidth {0 = narrow bandwidth -> 10 = huge bandwidth}
   - x2: not used
8. **weibull**
   - x1: mean location {0 -> 1}
   - x2: shape {0.5 = linear min, 1.5 = expon min, 3.5 = gaussian}
9. **gaussian**
   - x1: mean location {0 -> 1}
   - x2: bandwidth {0 = narrow bandwidth -> 10 = huge bandwidth}
10. **poisson**
    - x1: gravity center {0 = low values -> 10 = high values}
    - x2: compress/expand range {0.1 = full compress -> 4 full expand}
11. **walker**
    - x1: maximum value {0.1 -> 1}
    - x2: maximum step {0.1 -> 1}
12. **loopseg**
    - x1: maximum value {0.1 -> 1}
    - x2: maximum step {0.1 -> 1}

```
>>> s = Server().boot()
>>> s.start()
>>> wav = SquareTable()
>>> env = CosTable([(0,0), (100,1), (500,.3), (8191,0)])
>>> met = Metro(.125, 12).play()
>>> amp = TrigEnv(met, table=env, mul=.2)
>>> pit = TrigXnoise(met, dist=4, x1=10, mul=1000, add=200)
>>> a = Osc(table=wav, freq=pit, mul=amp).out()
```

setInput (*x*, *fadetime=0.05*)

Replace the *input* attribute.

Args

x: PyoObject New signal to process.

fadetime: float, optional Crossfade time between old and new input. Defaults to 0.05.

setDist (*x*)

Replace the *dist* attribute.

Args

x: int new *dist* attribute.

setX1 (*x*)

Replace the *x1* attribute.

Args

x: float or PyoObject new *x1* attribute.

setX2(*x*)
Replace the *x2* attribute.

Args

x: float or PyoObject new *x2* attribute.

ctrl(*map_list=None*, *title=None*, *wxnoserver=False*)
Opens a sliders window to control the parameters of the object. Only parameters that can be set to a PyoObject are allowed to be mapped on a slider.

If a list of values are given to a parameter, a multisliders will be used to control each stream independently.

Args

map_list: list of SLMap objects, optional Users defined set of parameters scaling. There is default scaling for each object that accept *ctrl* method.

title: string, optional Title of the window. If none is provided, the name of the class is used.

wxnoserver: boolean, optional With wxPython graphical toolkit, if True, tells the interpreter that there will be no server window.

If *wxnoserver* is set to True, the interpreter will not wait for the server GUI before showing the controller window.

input
PyoObject. Audio trigger signal.

dist
string or int. Distribution type.

x1
float or PyoObject. First parameter.

x2
float or PyoObject. Second parameter.

TrigXnoiseMidi

class TrigXnoiseMidi(*input*, *dist=0*, *x1=0.5*, *x2=0.5*, *scale=0*, *mrange=(0, 127)*, *mul=1*, *add=0*)
Triggered X-class midi notes pseudo-random generator.

Xnoise implements a few of the most common noise distributions. A new value is generated each time the object receive a trigger in input. Each distribution generates integer values in the range defined with *mrange* parameter and output can be scaled on midi notes, hertz or transposition factor.

Parent `PyoObject`

Args

input: PyoObject Audio signal sending triggers.

dist: string of int, optional Distribution type. Defaults to 0.

x1: float or PyoObject, optional First parameter. Defaults to 0.5.

x2: float or PyoObject, optional Second parameter. Defaults to 0.5.

scale: int {0, 1, 2}, optional

Output format. 0 = MIDI, 1 = Hertz, 2 = transposition factor. Defaults to 0.

In the transposition mode, the central key (the key where there is no transposition) is (*minrange* + *maxrange*) / 2.

mrange: tuple of int, optional Minimum and maximum possible values, in Midi notes. Available only at initialization time. Defaults to (0, 127).

Note:

Available distributions are:

0. uniform
1. linear minimum
2. linear maximum
3. triangular
4. exponential minimum
5. exponential maximum
6. double (bi)exponential
7. cauchy
8. weibull
9. gaussian
10. poisson
11. walker (drunk)
12. loopseg (drunk with looped segments)

Depending on the distribution, *x1* and *x2* parameters are applied as follow (names as string, or associated number can be used as *dist* parameter):

0. **uniform**
    - x1: not used
    - x2: not used
1. **linear_min**
    - x1: not used
    - x2: not used
2. **linear_max**
    - x1: not used
    - x2: not used
3. **triangle**
    - x1: not used
    - x2: not used
4. **expon_min**

- x1: slope {0 = no slope -> 10 = sharp slope}
- x2: not used

5. **expon_max**
   - x1: slope {0 = no slope -> 10 = sharp slope}
   - x2: not used
6. **biexpon**
   - x1: bandwidth {0 = huge bandwidth -> 10 = narrow bandwidth}
   - x2: not used
7. **cauchy**
   - x1: bandwidth {0 = narrow bandwidth -> 10 = huge bandwidth}
   - x2: not used
8. **weibull**
   - x1: mean location {0 -> 1}
   - x2: shape {0.5 = linear min, 1.5 = expon min, 3.5 = gaussian}
9. **gaussian**
   - x1: mean location {0 -> 1}
   - x2: bandwidth {0 = narrow bandwidth -> 10 = huge bandwidth}
10. **poisson**
    - x1: gravity center {0 = low values -> 10 = high values}
    - x2: compress/expand range {0.1 = full compress -> 4 full expand}
11. **walker**
    - x1: maximum value {0.1 -> 1}
    - x2: maximum step {0.1 -> 1}
12. **loopseg**
    - x1: maximum value {0.1 -> 1}
    - x2: maximum step {0.1 -> 1}

```
>>> s = Server().boot()
>>> s.start()
>>> wav = SquareTable()
>>> env = CosTable([(0,0), (100,1), (500,.3), (8191,0)])
>>> met = Metro(.125, 12).play()
>>> amp = TrigEnv(met, table=env, mul=.2)
>>> pit = TrigXnoiseMidi(met, dist=4, x1=10, scale=1, mrange=(48,84))
>>> a = Osc(table=wav, freq=pit, mul=amp).out()
```

setInput(*x*, *fadetime=0.05*)

Replace the *input* attribute.

Args

x: PyoObject New signal to process.

fadetime: float, optional Crossfade time between old and new input. Defaults to 0.05.

setDist(*x*)
Replace the *dist* attribute.

Args

x: int new *dist* attribute.

setScale(*x*)
Replace the *scale* attribute.

Possible values are:

0. Midi notes
1. Hertz
2. transposition factor (centralkey is (*minrange* + *maxrange*) / 2

Args

x: int {0, 1, 2} new *scale* attribute.

setRange(*mini*, *maxi*)
Replace the *mrange* attribute.

Args

mini: int minimum output midi range.

maxi: int maximum output midi range.

setX1(*x*)
Replace the *x1* attribute.

Args

x: float or PyoObject new *x1* attribute.

setX2(*x*)
Replace the *x2* attribute.

Args

x: float or PyoObject new *x2* attribute.

ctrl(*map_list=None*, *title=None*, *wxnoserver=False*)
Opens a sliders window to control the parameters of the object. Only parameters that can be set to a PyoObject are allowed to be mapped on a slider.

If a list of values are given to a parameter, a multisliders will be used to control each stream independently.

Args

map_list: list of SLMap objects, optional Users defined set of parameters scaling. There is default scaling for each object that accept *ctrl* method.

title: string, optional Title of the window. If none is provided, the name of the class is used.

wxnoserver: boolean, optional With wxPython graphical toolkit, if True, tells the interpreter that there will be no server window.

If *wxnoserver* is set to True, the interpreter will not wait for the server GUI before showing the controller window.

input
PyoObject. Audio trigger signal.

dist
string or int. Distribution type.

x1
float or PyoObject. First parameter.

x2
float or PyoObject. Second parameter.

scale
int. Output format.

Utilities

Miscellaneous objects.

AToDB

class AToDB(*input*, *mul=1*, *add=0*)
Returns the decibel equivalent of an amplitude value.

Returns the decibel equivalent of an amplitude value, 1 = 0 dB. The *input* values are internally clipped to 0.000001 so values less than or equal to 0.000001 return -120 dB.

Parent *PyoObject*

Args

input: PyoObject Input signal, amplitude value.

```
>>> s = Server().boot()
>>> s.start()
>>> # amplitude modulation of a notch around 1000 Hz, from -120 db to 0dB
>>> amp = Sine([1,1.5], mul=.5, add=.5)
>>> db = AToDB(amp)
>>> a = PinkNoise(.2)
>>> b = EQ(a, freq=1000, q=2, boost=db).out()
```

setInput(*x*, *fadetime=0.05*)
Replace the *input* attribute.

Args

x: PyoObject New signal to process.

fadetime: float, optional Crossfade time between old and new input. Default to 0.05.

input
PyoObject. Input signal to process.

Between

class Between(*input*, *min=-1.0*, *max=1.0*, *mul=1*, *add=0*)
Informs when an input signal is contained in a specified range.

Outputs a value of 1.0 if the input signal is greater or egal than *min* and less than *max*. Otherwise, outputs a value of 0.0.

Parent *PyoObject*

Args

input: PyoObject Input signal to process.

min: float or PyoObject, optional Minimum range value. Defaults to 0.

max: float or PyoObject, optional Maximum range value. Defaults to 1.

```
>>> s = Server().boot()
>>> s.start()
>>> ph = Phasor(freq=[7,8])
>>> tr = Between(ph, min=0, max=.25)
>>> amp = Port(tr, risetime=0.002, falltime=0.002, mul=.2)
>>> a = SineLoop(freq=[245,250], feedback=.1, mul=amp).out()
```

setInput(*x*, *fadetime=0.05*)
Replace the *input* attribute.

Args

x: PyoObject New signal to process.

fadetime: float, optional Crossfade time between old and new input. Defaults to 0.05.

setMin(*x*)
Replace the *min* attribute.

Args

x: float or PyoObject New *min* attribute.

setMax(*x*)
Replace the *max* attribute.

Args

x: float or PyoObject New *max* attribute.

ctrl(*map_list=None*, *title=None*, *wxnoserver=False*)
Opens a sliders window to control the parameters of the object. Only parameters that can be set to a PyoObject are allowed to be mapped on a slider.

If a list of values are given to a parameter, a multisliders will be used to control each stream independently.

Args

map_list: list of SLMap objects, optional Users defined set of parameters scaling. There is default scaling for each object that accept *ctrl* method.

title: string, optional Title of the window. If none is provided, the name of the class is used.

wxnoserver: boolean, optional With wxPython graphical toolkit, if True, tells the interpreter that there will be no server window.

If *wxnoserver* is set to True, the interpreter will not wait for the server GUI before showing the controller window.

input
PyoObject. Input signal to process.

min
float or PyoObject. Minimum range value.

max
float or PyoObject. Maximum range value.

CentsToTranspo

class CentsToTranspo(*input*, *mul=1*, *add=0*)
Returns the transposition factor equivalent of a given cents value.

Returns the transposition factor equivalent of a given cents value, 0 cents = 1.

Parent *PyoObject*

Args

input: PyoObject Input signal, cents value.

```
>>> s = Server().boot()
>>> s.start()
>>> met = Metro(.125, poly=2).play()
>>> cts = TrigRandInt(met, max=12, mul=100)
>>> trans = CentsToTranspo(cts)
>>> sf = SfPlayer(SNDS_PATH+"/transparent.aif", loop=True, speed=trans, mul=.25).
↪out()
```

setInput(*x*, *fadetime=0.05*)
Replace the *input* attribute.

Args

x: PyoObject New signal to process.

fadetime: float, optional Crossfade time between old and new input. Default to 0.05.

input
PyoObject. Input signal to process.

Clean_objects

class Clean_objects(*time*, **args*)
Stops and deletes PyoObjects after a given time.

The start() method starts the thread timer (must be called).

Args

time: float Time, in seconds, to wait before calling stop on the given objects and deleting them.

***args: PyoObject(s)** Objects to delete.

```
>>> s = Server().boot()
>>> s.start()
>>> a = Noise(mul=.5).mix(2)
>>> b = Fader(fadein=.5, fadeout=1, dur=5).play()
>>> c = Biquad(a, freq=500, q=2, mul=b).out()
>>> dump = Clean_objects(6, a, b, c)
>>> dump.start()
```

run()
Method representing the thread's activity.

You may override this method in a subclass. The standard run() method invokes the callable object passed to the object's constructor as the target argument, if any, with sequential and keyword arguments taken from the args and kwargs arguments, respectively.

Compare

class Compare(*input*, *comp*, *mode='<'*, *mul=1*, *add=0*)
Comparison object.

Compare evaluates a comparison between a PyoObject and a number or between two PyoObjects and outputs 1.0, as audio stream, if the comparison is true, otherwise outputs 0.0.

Parent *PyoObject*

Args

input: PyoObject Input signal.

comp: float or PyoObject comparison signal.

mode: string, optional Comparison operator as a string. Allowed operator are "<", "<=", ">", ">=", "==", "!=". Default to "<".

```
>>> s = Server().boot()
>>> s.start()
>>> a = SineLoop(freq=[199,200], feedback=.1, mul=.2)
>>> b = SineLoop(freq=[149,150], feedback=.1, mul=.2)
>>> ph = Phasor(freq=1)
>>> ch = Compare(input=ph, comp=0.5, mode="<=")
>>> out = Selector(inputs=[a,b], voice=Port(ch)).out()
```

out(*chnl=0*, *inc=1*, *dur=0*, *delay=0*)
Start processing and send samples to audio output beginning at *chnl*.

This method returns *self*, allowing it to be applied at the object creation.

Args

chnl: int, optional Physical output assigned to the first audio stream of the object. Defaults to 0.

inc: int, optional Output channel increment value. Defaults to 1.

dur: float, optional Duration, in seconds, of the object's activation. The default is 0 and means infinite duration.

delay: float, optional Delay, in seconds, before the object's activation. Defaults to 0.

If *chnl* >= 0, successive streams increment the output number by *inc* and wrap around the global number of channels.

If *chnl* is negative, streams begin at 0, increment the output number by *inc* and wrap around the global number of channels. Then, the list of streams is scrambled.

If *chnl* is a list, successive values in the list will be assigned to successive streams.

setInput(*x*, *fadetime=0.05*)
Replace the *input* attribute.

Args

x: PyoObject New signal to process.

fadetime: float, optional Crossfade time between old and new input. Default to 0.05.

setComp(*x*)
Replace the *comp* attribute.

Args

x: float or PyoObject New comparison signal.

setMode(*x*)
Replace the *mode* attribute.

Allowed operator are "<", "<=", ">", ">=", "==", "!=".

Args

x: string New *mode* attribute.

input
PyoObject. Input signal.

comp
PyoObject. Comparison signal.

mode
string. Comparison operator.

ControlRead

class ControlRead(*filename*, *rate=1000*, *loop=False*, *interp=2*, *mul=1*, *add=0*)
Reads control values previously stored in text files.

Read sampled sound from a table, with optional looping mode.

Parent *PyoObject*

Args

filename: string Full path (without extension) used to create the files.

Usually the same filename as the one given to a ControlRec object to record automation.

The directory will be scaned and all files named "filename_xxx" will add a new stream in the object.

rate: int, optional Rate at which the values are sampled. Defaults to 1000.

loop: boolean, optional Looping mode, False means off, True means on. Defaults to False.

interp: int, optional

Choice of the interpolation method.

1. no interpolation
2. linear (default)
3. cosinus
4. cubic

Note: ControlRead will send a trigger signal at the end of the playback if loop is off or any time it wraps around if loop is on. User can retreive the trigger streams by calling obj['trig']:

```
>>> rnds = ControlRead(home+"/freq_auto", loop=True)
>>> t = SndTable(SNDS_PATH+"/transparent.aif")
>>> loop = TrigEnv(rnds["trig"], t, dur=[.2,.3,.4,.5], mul=.5).out()
```

The out() method is bypassed. ControlRead's signal can not be sent to audio outs.

See also:

ControlRec

```
>>> s = Server().boot()
>>> s.start()
>>> home = os.path.expanduser('~')
>>> # assuming "test_xxx" exists in the user directory
>>> # run ControlRec's example to generate the files
>>> rnds = ControlRead(home+"/test", rate=100, loop=True)
>>> sines = SineLoop(freq=rnds, feedback=.05, mul=.15).out()
```

out (*chnl=0*, *inc=1*, *dur=0*, *delay=0*)

Start processing and send samples to audio output beginning at *chnl*.

This method returns *self*, allowing it to be applied at the object creation.

Args

chnl: int, optional Physical output assigned to the first audio stream of the object. Defaults to 0.

inc: int, optional Output channel increment value. Defaults to 1.

dur: float, optional Duration, in seconds, of the object's activation. The default is 0 and means infinite duration.

delay: float, optional Delay, in seconds, before the object's activation. Defaults to 0.

If *chnl* >= 0, successive streams increment the output number by *inc* and wrap around the global number of channels.

If *chnl* is negative, streams begin at 0, increment the output number by *inc* and wrap around the global number of channels. Then, the list of streams is scrambled.

If *chnl* is a list, successive values in the list will be assigned to successive streams.

setRate (*x*)

Replace the *rate* attribute.

Args

x: int new *rate* attribute.

setLoop (*x*)

Replace the *loop* attribute.

Args

x: boolean new *loop* attribute.

setInterp (*x*)

Replace the *interp* attribute.

Args

x: int {1, 2, 3, 4} new *interp* attribute.

rate
int. Sampling frequency in cycles per second.

loop
boolean. Looping mode.

interp
int {1, 2, 3, 4}. Interpolation method.

ControlRec

class ControlRec(*input*, *filename*, *rate=1000*, *dur=0.0*)
Records control values and writes them in a text file.

input parameter must be a valid PyoObject managing any number of streams, other parameters can't be in list format. The user must call the *write* method to create text files on the disk.

Each line in the text files contains two values, the absolute time in seconds and the sampled value.

The play() method starts the recording and is not called at the object creation time.

Parent *PyoObject*

Args

input: PyoObject Input signal to sample.

filename: string Full path (without extension) used to create the files.

"_000" will be added to file's names with increasing digits according to the number of streams in input.

The same filename can be passed to a ControlRead object to read all related files.

rate: int, optional Rate at which the input values are sampled. Defaults to 1000.

dur: float, optional Duration of the recording, in seconds. If 0.0, the recording won't stop until the end of the performance.

If greater than 0.0, the *stop* method is automatically called at the end of the recording.

Note: All parameters can only be set at intialization time.

The write() method must be called on the object to write the files on the disk.

The out() method is bypassed. ControlRec's signal can not be sent to audio outs.

ControlRec has no *mul* and *add* attributes.

See also:

ControlRead

```
>>> s = Server().boot()
>>> s.start()
>>> rnds = Randi(freq=[1,2], min=200, max=400)
>>> sines = SineLoop(freq=rnds, feedback=.05, mul=.2).out()
>>> home = os.path.expanduser('~')
>>> rec = ControlRec(rnds, home+"/test", rate=100, dur=4).play()
>>> # call rec.write() to save "test_000" and "test_001" in the home directory.
>>> def write_files():
```

(continues on next page)

(continued from previous page)

```
...     sines.mul = 0
...     rec.write()
>>> call = CallAfter(function=write_files, time=4.5)
```

out(*chnl=0*, *inc=1*, *dur=0*, *delay=0*)
Start processing and send samples to audio output beginning at *chnl*.

This method returns *self*, allowing it to be applied at the object creation.

Args

chnl: int, optional Physical output assigned to the first audio stream of the object. Defaults to 0.

inc: int, optional Output channel increment value. Defaults to 1.

dur: float, optional Duration, in seconds, of the object's activation. The default is 0 and means infinite duration.

delay: float, optional Delay, in seconds, before the object's activation. Defaults to 0.

If *chnl* >= 0, successive streams increment the output number by *inc* and wrap around the global number of channels.

If *chnl* is negative, streams begin at 0, increment the output number by *inc* and wrap around the global number of channels. Then, the list of streams is scrambled.

If *chnl* is a list, successive values in the list will be assigned to successive streams.

write()
Writes recorded values in text files on the disk.

DBToA

class DBToA(*input*, *mul=1*, *add=0*)
Returns the amplitude equivalent of a decibel value.

Returns the amplitude equivalent of a decibel value, 0 dB = 1. The *input* values are internally clipped to -120 dB so -120 dB returns 0.

Parent *PyoObject*

Args

input: PyoObject Input signal, decibel value.

```
>>> s = Server().boot()
>>> s.start()
>>> # amplitude modulation 6 dB around -18 dB
>>> db = Sine(freq=1, phase=[0,.5], mul=6, add=-18)
>>> amp = DBToA(db)
>>> b = SineLoop(freq=[350,400], feedback=.1, mul=amp).out()
```

setInput(*x*, *fadetime=0.05*)
Replace the *input* attribute.

Args

x: PyoObject New signal to process.

fadetime: float, optional Crossfade time between old and new input. Default to 0.05.

input
PyoObject. Input signal to process.

Denorm

class Denorm(*input*, *mul=1*, *add=0*)
Mixes low level noise to an input signal.

Mixes low level (~1e-24 for floats, and ~1e-60 for doubles) noise to a an input signal. Can be used before IIR filters and reverbs to avoid denormalized numbers which may otherwise result in significantly increased CPU usage.

Parent *PyoObject*

Args

input: PyoObject Input signal to process.

```
>>> s = Server().boot()
>>> s.start()
>>> amp = Linseg([(0,0),(2,1),(4,0)], loop=True).play()
>>> a = Sine(freq=[800,1000], mul=0.01*amp)
>>> den = Denorm(a)
>>> rev = Freeverb(den, size=.9).out()
```

setInput(*x*, *fadetime=0.05*)
Replace the *input* attribute.

Args

x: PyoObject New signal to process.

fadetime: float, optional Crossfade time between old and new input. Default to 0.05.

input
PyoObject. Input signal to filter.

FToM

class FToM(*input*, *mul=1*, *add=0*)
Returns the midi note equivalent to a frequency in Hz.

Returns the midi note equivalent to a frequency in Hz, 440.0 (hz) = 69.

Parent *PyoObject*

Args

input: PyoObject Input signal as frequency in Hz.

```
>>> s = Server().boot()
>>> s.start()
>>> lfo = Sine([0.2,0.25], mul=300, add=600)
>>> src = SineLoop(freq=lfo, feedback=0.05)
>>> hz = Yin(src, minfreq=100, maxfreq=1000, cutoff=500)
>>> mid = FToM(hz)
>>> fr = Snap(mid, choice=[0,2,5,7,9], scale=1)
>>> freq = Port(fr, risetime=0.01, falltime=0.01)
>>> syn = SineLoop(freq, feedback=0.05, mul=0.3).out()
```

setInput(*x*, *fadetime=0.05*)
Replace the *input* attribute.

Args

x: PyoObject New signal to process.

fadetime: float, optional Crossfade time between old and new input. Default to 0.05.

input
PyoObject. Input signal to process.

Interp

class Interp(*input*, *input2*, *interp=0.5*, *mul=1*, *add=0*)
Interpolates between two signals.

Parent *PyoObject*

Args

input: PyoObject First input signal.

input2: PyoObject Second input signal.

interp: float or PyoObject, optional Averaging value. 0 means only first signal, 1 means only second signal. Default to 0.5.

```
>>> s = Server().boot()
>>> s.start()
>>> sf = SfPlayer(SNDS_PATH + '/accord.aif', speed=[.99,1], loop=True, mul=.3)
>>> sf2 = SfPlayer(SNDS_PATH + '/transparent.aif', speed=[.99,1], loop=True, mul=.
↪3)
>>> lfo = Osc(table=SquareTable(20), freq=5, mul=.5, add=.5)
>>> a = Interp(sf, sf2, lfo).out()
```

setInput(*x*, *fadetime=0.05*)
Replace the *input* attribute.

Args

x: PyoObject New signal to process.

fadetime: float, optional Crossfade time between old and new input. Default to 0.05.

setInput2(*x*, *fadetime=0.05*)
Replace the *input2* attribute.

Args

x: PyoObject New signal to process.

fadetime: float, optional Crossfade time between old and new input. Default to 0.05.

setInterp(*x*)
Replace the *interp* attribute.

Args

x: float or PyoObject New *interp* attribute.

ctrl(*map_list=None*, *title=None*, *wxnoserver=False*)
Opens a sliders window to control the parameters of the object. Only parameters that can be set to a PyoObject are allowed to be mapped on a slider.

If a list of values are given to a parameter, a multisliders will be used to control each stream independently.

Args

map_list: list of SLMap objects, optional Users defined set of parameters scaling. There is default scaling for each object that accept *ctrl* method.

title: string, optional Title of the window. If none is provided, the name of the class is used.

wxnoserver: boolean, optional With wxPython graphical toolkit, if True, tells the interpreter that there will be no server window.

If *wxnoserver* is set to True, the interpreter will not wait for the server GUI before showing the controller window.

input
PyoObject. First input signal.

input2
PyoObject. Second input signal.

interp
float or PyoObject. Averaging value.

MToF

class MToF(*input*, *mul=1*, *add=0*)
Returns the frequency (Hz) equivalent to a midi note.

Returns the frequency (Hz) equivalent to a midi note, 60 = 261.62556530066814 Hz.

Parent *PyoObject*

Args

input: PyoObject Input signal as midi note.

```
>>> s = Server().boot()
>>> s.start()
>>> met = Metro(.125, poly=2).play()
>>> mid = TrigChoice(met, choice=[60, 63, 67, 70], port=.005)
>>> hz = MToF(mid)
>>> syn = SineLoop(freq=hz, feedback=.07, mul=.2).out()
```

setInput(*x*, *fadetime=0.05*)
Replace the *input* attribute.

Args

x: PyoObject New signal to process.

fadetime: float, optional Crossfade time between old and new input. Default to 0.05.

input
PyoObject. Input signal to process.

MToT

class MToT(*input*, *centralkey=60.0*, *mul=1*, *add=0*)
Returns the transposition factor equivalent to a midi note.

Returns the transposition factor equivalent to a midi note. If the midi note equal the *centralkey* argument, the output is 1.0.

Parent *PyoObject*

Args

input: PyoObject Input signal as midi note.

centralkey: float, optional The midi note that returns a transposition factor of 1, that is to say no transposition. Defaults to 60.

```
>>> s = Server().boot()
>>> s.start()
>>> tsnd = SndTable(SNDS_PATH+"/accord.aif")
>>> tenv = CosTable([(0,0), (100,1), (1000,.5), (8192,0)])
>>> met = Metro(.125, poly=2).play()
>>> amp = TrigEnv(met, table=tenv, dur=.25, mul=.7)
>>> mid = TrigChoice(met, choice=[43, 45, 60, 63], port=.0025)
>>> sp = MToT(mid)
>>> snd = Osc(tsnd, freq=tsnd.getRate()/sp, mul=amp).out()
```

setInput(*x*, *fadetime=0.05*)
Replace the *input* attribute.

Args

x: PyoObject New signal to process.

fadetime: float, optional Crossfade time between old and new input. Default to 0.05.

setCentralKey(*x*)
Replace the *centralkey* attribute.

Args

x: float New *centralkey* attribute.

input
PyoObject. Input signal to process.

centralkey
float. The midi note that returns no transposition.

NoteinRead

class NoteinRead(*filename*, *loop=False*, *mul=1*, *add=0*)
Reads Notein values previously stored in text files.

Parent *PyoObject*

Args

filename: string Full path (without extension) used to create the files.

Usually the same filename as the one given to a NoteinRec object to record automation.

The directory will be scaned and all files named "filename_xxx" will add a new stream in the object.

loop: boolean, optional Looping mode, False means off, True means on. Defaults to False.

Note: NoteinRead will send a trigger signal at the end of the playback if loop is off or any time it wraps around if loop is on. User can retreive the trigger streams by calling obj['trig']:

```
>>> notes = NoteinRead(home+"/notes_rec", loop=True)
>>> t = SndTable(SNDS_PATH+"/transparent.aif")
>>> loop = TrigEnv(notes["trig"], t, dur=[.2,.3,.4,.5], mul=.25).out()
```

The out() method is bypassed. NoteinRead's signal can not be sent to audio outs.

See also:

NoteinRec

```
>>> s = Server().boot()
>>> s.start()
>>> home = os.path.expanduser('~')
>>> # assuming "test_xxx" exists in the user directory
>>> notes = NoteinRead(home+"/test", loop=True)
>>> amps = Port(notes['velocity'], 0.001, 0.5, mul=.2)
>>> sines = SineLoop(freq=notes['pitch'], feedback=.05, mul=amps).out()
```

get(*identifier='pitch'*, *all=False*)

Return the first sample of the current buffer as a float.

Can be used to convert audio stream to usable Python data.

"pitch" or "velocity" must be given to *identifier* to specify which stream to get value from.

Args

identifier: string {"pitch", "velocity"} Address string parameter identifying audio stream. Defaults to "pitch".

all: boolean, optional If True, the first value of each object's stream will be returned as a list.

If False, only the value of the first object's stream will be returned as a float.

out(*chnl=0*, *inc=1*, *dur=0*, *delay=0*)

Start processing and send samples to audio output beginning at *chnl*.

This method returns *self*, allowing it to be applied at the object creation.

Args

chnl: int, optional Physical output assigned to the first audio stream of the object. Defaults to 0.

inc: int, optional Output channel increment value. Defaults to 1.

dur: float, optional Duration, in seconds, of the object's activation. The default is 0 and means infinite duration.

delay: float, optional Delay, in seconds, before the object's activation. Defaults to 0.

If *chnl* >= 0, successive streams increment the output number by *inc* and wrap around the global number of channels.

If *chnl* is negative, streams begin at 0, increment the output number by *inc* and wrap around the global number of channels. Then, the list of streams is scrambled.

If *chnl* is a list, successive values in the list will be assigned to successive streams.

setLoop(*x*)
Replace the *loop* attribute.

Args

x: boolean new *loop* attribute.

loop
boolean. Looping mode.

NoteinRec

class NoteinRec(*input*, *filename*)
Records Notein inputs and writes them in a text file.

input parameter must be a Notein object managing any number of streams, other parameters can't be in list format. The user must call the *write* method to create text files on the disk.

Each line in the text files contains three values, the absolute time in seconds, the Midi pitch and the normalized velocity.

The play() method starts the recording and is not called at the object creation time.

Parent *PyoObject*

Args

input: Notein Notein signal to sample.

filename: string Full path (without extension) used to create the files.

"_000" will be added to file's names with increasing digits according to the number of streams in input.

The same filename can be passed to a NoteinRead object to read all related files.

Note: All parameters can only be set at intialization time.

The *write* method must be called on the object to write the files on the disk.

The out() method is bypassed. NoteinRec's signal can not be sent to audio outs.

NoteinRec has no *mul* and *add* attributes.

See also:

NoteinRead

```
>>> s = Server().boot()
>>> s.start()
>>> notes = Notein(poly=2)
>>> home = os.path.expanduser('~')
>>> rec = NoteinRec(notes, home+"/test").play()
>>> # call rec.write() to save "test_000" and "test_001" in the home directory.
```

out(*chnl=0*, *inc=1*, *dur=0*, *delay=0*)
Start processing and send samples to audio output beginning at *chnl*.

This method returns *self*, allowing it to be applied at the object creation.

Args

chnl: int, optional Physical output assigned to the first audio stream of the object. Defaults to 0.

inc: int, optional Output channel increment value. Defaults to 1.

dur: float, optional Duration, in seconds, of the object's activation. The default is 0 and means infinite duration.

delay: float, optional Delay, in seconds, before the object's activation. Defaults to 0.

If *chnl* >= 0, successive streams increment the output number by *inc* and wrap around the global number of channels.

If *chnl* is negative, streams begin at 0, increment the output number by *inc* and wrap around the global number of channels. Then, the list of streams is scrambled.

If *chnl* is a list, successive values in the list will be assigned to successive streams.

write()
Writes recorded values in text files on the disk.

Print

class Print(*input*, *method=0*, *interval=0.25*, *message=''*)
Print PyoObject's current value.

Parent *PyoObject*

Args

input: PyoObject Input signal to filter.

method: int {0, 1}, optional There is two methods to set when a value is printed (Defaults to 0): 0. at a periodic interval. 1. everytime the value changed.

interval: float, optional Interval, in seconds, between each print. Used by method 0. Defaults to 0.25.

message: str, optional Message to print before the current value. Defaults to "".

Note: The out() method is bypassed. Print's signal can not be sent to audio outs.

Print has no *mul* and *add* attributes.

```
>>> s = Server().boot()
>>> s.start()
>>> a = SfPlayer(SNDS_PATH + '/transparent.aif', loop=True, mul=.3).out()
>>> b = Follower(a)
>>> p = Print(b, method=0, interval=.1, message="RMS")
```

setInput(*x*, *fadetime=0.05*)
Replace the *input* attribute.

Args

x: PyoObject New signal to process.

fadetime: float, optional Crossfade time between old and new input. Default to 0.05.

setMethod(*x*)
Replace the *method* attribute.

Args

x: int {0, 1} New *method* attribute.

setInterval(*x*)
Replace the *interval* attribute.

Args

x: float New *interval* attribute.

setMessage(*x*)
Replace the *message* attribute.

Args

x: str New *message* attribute.

out(*chnl=0*, *inc=1*, *dur=0*, *delay=0*)
Start processing and send samples to audio output beginning at *chnl*.

This method returns *self*, allowing it to be applied at the object creation.

Args

chnl: int, optional Physical output assigned to the first audio stream of the object. Defaults to 0.

inc: int, optional Output channel increment value. Defaults to 1.

dur: float, optional Duration, in seconds, of the object's activation. The default is 0 and means infinite duration.

delay: float, optional Delay, in seconds, before the object's activation. Defaults to 0.

If *chnl* >= 0, successive streams increment the output number by *inc* and wrap around the global number of channels.

If *chnl* is negative, streams begin at 0, increment the output number by *inc* and wrap around the global number of channels. Then, the list of streams is scrambled.

If *chnl* is a list, successive values in the list will be assigned to successive streams.

input
PyoObject. Input signal.

method
int. Controls when a value is printed.

interval
float. For method 0, interval, in seconds, between each print.

message
str. Message to print before the current value.

Record

class Record(*input*, *filename*, *chnls=2*, *fileformat=0*, *sampletype=0*, *buffering=4*, *quality=0.4*)
Writes input sound in an audio file on the disk.

input parameter must be a valid PyoObject or an addition of PyoObjects, parameters can't be in list format.

Parent `PyoObject`

Args

input: PyoObject Input signal to record.

filename: string Full path of the file to create.

chnls: int, optional Number of channels in the audio file. Defaults to 2.

fileformat: int, optional Format type of the audio file. Defaults to 0.

Record will first try to set the format from the filename extension.

If it's not possible, it uses the fileformat parameter. Supported formats are:

0. WAV - Microsoft WAV format (little endian) {.wav, .wave}
1. AIFF - Apple/SGI AIFF format (big endian) {.aif, .aiff}
2. AU - Sun/NeXT AU format (big endian) {.au}
3. RAW - RAW PCM data {no extension}
4. SD2 - Sound Designer 2 {.sd2}
5. FLAC - FLAC lossless file format {.flac}
6. CAF - Core Audio File format {.caf}
7. OGG - Xiph OGG container {.ogg}

sampletype: int, optional Bit depth encoding of the audio file.

SD2 and FLAC only support 16 or 24 bit int. Supported types are:

0. 16 bits int (default)
1. 24 bits int
2. 32 bits int
3. 32 bits float
4. 64 bits float
5. U-Law encoded
6. A-Law encoded

buffering: int, optional Number of bufferSize to wait before writing samples to disk.

High buffering uses more memory but improves performance. Defaults to 4.

quality: float, optional The encoding quality value, between 0.0 (lowest quality) and 1.0 (highest quality). This argument has an effect only with FLAC and OGG compressed formats. Defaults to 0.4.

Note: All parameters can only be set at intialization time.

The stop() method must be called on the object to close the file properly.

The out() method is bypassed. Record's signal can not be sent to audio outs.

Record has no *mul* and *add* attributes.

```
>>> s = Server().boot()
>>> s.start()
>>> from random import uniform
>>> import os
```

(continues on next page)

(continued from previous page)

```
>>> t = HarmTable([1, 0, 0, .2, 0, 0, 0, .1, 0, 0, .05])
>>> amp = Fader(fadein=.05, fadeout=2, dur=4, mul=.05).play()
>>> osc = Osc(t, freq=[uniform(350,360) for i in range(10)], mul=amp).out()
>>> home = os.path.expanduser('~')
>>> # Records an audio file called "example_synth.aif" in the home folder
>>> rec = Record(osc, filename=home+"/example_synth.aif", fileformat=1,␣
↪sampletype=1)
>>> clean = Clean_objects(4.5, rec)
>>> clean.start()
```

out(*chnl=0*, *inc=1*, *dur=0*, *delay=0*)

Start processing and send samples to audio output beginning at *chnl*.

This method returns *self*, allowing it to be applied at the object creation.

Args

chnl: int, optional Physical output assigned to the first audio stream of the object. Defaults to 0.

inc: int, optional Output channel increment value. Defaults to 1.

dur: float, optional Duration, in seconds, of the object's activation. The default is 0 and means infinite duration.

delay: float, optional Delay, in seconds, before the object's activation. Defaults to 0.

If *chnl* >= 0, successive streams increment the output number by *inc* and wrap around the global number of channels.

If *chnl* is negative, streams begin at 0, increment the output number by *inc* and wrap around the global number of channels. Then, the list of streams is scrambled.

If *chnl* is a list, successive values in the list will be assigned to successive streams.

setInput(*x*, *fadetime=0.05*)

Replace the *input* attribute.

Args

x: PyoObject New signal to process.

fadetime: float, optional Crossfade time between old and new input. Default to 0.05.

input

PyoObject. Input signal to filter.

SampHold

class SampHold(*input*, *controlsig*, *value=0.0*, *mul=1*, *add=0*)

Performs a sample-and-hold operation on its input.

SampHold performs a sample-and-hold operation on its input according to the value of *controlsig*. If *controlsig* equals *value*, the input is sampled and held until next sampling.

Parent *PyoObject*

Args

input: PyoObject Input signal.

controlsig: PyoObject Controls when to sample the signal.

value: float or PyoObject, optional Sampling target value. Default to 0.0.

```
>>> s = Server().boot()
>>> s.start()
>>> a = Noise(500,1000)
>>> b = Sine([3,4])
>>> c = SampHold(input=a, controlsig=b, value=0)
>>> d = Sine(c, mul=.2).out()
```

setInput(*x*, *fadetime=0.05*)
Replace the *input* attribute.

Args

x: PyoObject New signal to process.

fadetime: float, optional Crossfade time between old and new input. Default to 0.05.

setControlsig(*x*, *fadetime=0.05*)
Replace the *controlsig* attribute.

Args

x: PyoObject New control signal.

fadetime: float, optional Crossfade time between old and new input. Default to 0.05.

setValue(*x*)
Replace the *value* attribute.

Args

x: float or PyoObject New *value* attribute.

input
PyoObject. Input signal.

controlsig
PyoObject. Control signal.

value
float or PyoObject. Target value.

Scale

class Scale(*input*, *inmin=0*, *inmax=1*, *outmin=0*, *outmax=1*, *exp=1*, *mul=1*, *add=0*)
Maps an input range of audio values to an output range.

Scale maps an input range of audio values to an output range. The ranges can be specified with *min* and *max* reversed for invert-mapping. If specified, the mapping can also be exponential.

Parent *PyoObject*

Args

input: PyoObject Input signal to process.

inmin: float or PyoObject, optional Minimum input value. Defaults to 0.

inmax: float or PyoObject, optional Maximum input value. Defaults to 1.

outmin: float or PyoObject, optional Minimum output value. Defaults to 0.

outmax: float or PyoObject, optional Maximum output value. Defaults to 1.

exp: float, optional Exponent value, specifies the nature of the scaling curve. Values between 0 and 1 give a reversed curve. Defaults to 1.0.

```
>>> s = Server().boot()
>>> s.start()
>>> met = Metro(.125, poly=2).play()
>>> rnd = TrigRand(met, min=0, max=1, port=.005)
>>> omlf = Sine(.5, mul=700, add=1000)
>>> fr = Scale(rnd, inmin=0, inmax=1, outmin=250, outmax=omlf, exp=1)
>>> amp = TrigEnv(met, table=HannTable(), dur=.25, mul=.2)
>>> out = SineLoop(fr, feedback=.07, mul=amp).out()
```

setInput (*x*, *fadetime=0.05*)
Replace the *input* attribute.

Args

x: PyoObject New signal to process.

fadetime: float, optional Crossfade time between old and new input. Defaults to 0.05.

setInMin (*x*)
Replace the *inmin* attribute.

Args

x: float or PyoObject New *inmin* attribute.

setInMax (*x*)
Replace the *inmax* attribute.

Args

x: float or PyoObject New *inmax* attribute.

setOutMin (*x*)
Replace the *outmin* attribute.

Args

x: float or PyoObject New *outmin* attribute.

setOutMax (*x*)
Replace the *outmax* attribute.

Args

x: float or PyoObject New *outmax* attribute.

setExp (*x*)
Replace the *exp* attribute.

Args

x: float New *exp* attribute.

ctrl (*map_list=None*, *title=None*, *wxnoserver=False*)
Opens a sliders window to control the parameters of the object. Only parameters that can be set to a PyoObject are allowed to be mapped on a slider.

If a list of values are given to a parameter, a multisliders will be used to control each stream independently.

Args

map_list: list of SLMap objects, optional Users defined set of parameters scaling. There is default scaling for each object that accept *ctrl* method.

title: string, optional Title of the window. If none is provided, the name of the class is used.

wxnoserver: boolean, optional With wxPython graphical toolkit, if True, tells the interpreter that there will be no server window.

If *wxnoserver* is set to True, the interpreter will not wait for the server GUI before showing the controller window.

input
PyoObject. Input signal to process.

inmin
float or PyoObject. Minimum input value.

inmax
float or PyoObject. Maximum input value.

outmin
float or PyoObject. Minimum output value.

outmax
float or PyoObject. Maximum output value.

exp
float or PyoObject. Exponent value (nature of the scaling curve).

Snap

class Snap(*input*, *choice*, *scale=0*, *mul=1*, *add=0*)
Snap input values on a user's defined midi scale.

Snap takes an audio input of floating-point values from 0 to 127 and output the nearest value in the *choice* parameter. *choice* can be defined on any number of octaves and the real snapping values will be automatically expended. The object will take care of the input octave range. According to *scale* parameter, output can be in midi notes, hertz or transposition factor (centralkey = 60).

Parent *PyoObject*

Args

input: PyoObject Incoming Midi notes as an audio stream.

choice: list of floats Possible values, as midi notes, for output.

scale: int {0, 1, 2}, optional

Pitch output format.

0. MIDI (default)
1. Hertz
2. transposition factor

In the transpo mode, the central key (the key where there is no transposition) is 60.

```
>>> s = Server().boot()
>>> s.start()
>>> wav = SquareTable()
>>> env = CosTable([(0,0), (100,1), (500,.3), (8191,0)])
>>> met = Metro(.125, 8).play()
```

(continues on next page)

(continued from previous page)

```
>>> amp = TrigEnv(met, table=env, mul=.2)
>>> pit = TrigXnoiseMidi(met, dist=4, x1=20, mrange=(48,84))
>>> hertz = Snap(pit, choice=[0,2,3,5,7,8,10], scale=1)
>>> a = Osc(table=wav, freq=hertz, phase=0, mul=amp).out()
```

setInput(*x*, *fadetime=0.05*)
Replace the *input* attribute.

Args

x: PyoObject New signal to process.

fadetime: float, optional Crossfade time between old and new input. Defaults to 0.05.

setChoice(*x*)
Replace the *choice* attribute.

Args

x: list of floats new *choice* attribute.

setScale(*x*)
Replace the *scale* attribute.

Possible values are:

0. Midi notes
1. Hertz
2. transposition factor (centralkey = 60)

Args

x: int {0, 1, 2} new *scale* attribute.

input
PyoObject. Audio signal to transform.

choice
list of floats. Possible values.

scale
int. Output format.

TranspoToCents

class TranspoToCents(*input*, *mul=1*, *add=0*)
Returns the cents value equivalent of a transposition factor.

Returns the cents value equivalent of a transposition factor, 1 = 0 cents.

Parent *PyoObject*

Args

input: PyoObject Input signal, transposition factor.

```
>>> s = Server().boot()
>>> s.start()
>>> met = Metro(.125, poly=2).play()
>>> trans = TrigChoice(met, choice=[.25,.5,.5,.75,1,1.25,1.5])
>>> semi = TranspoToCents(trans, mul=0.01)
>>> sf = SfPlayer(SNDS_PATH+"/transparent.aif", loop=True, mul=.3).out()
>>> harm = Harmonizer(sf, transpo=semi).out()
```

setInput(*x*, *fadetime=0.05*)
Replace the *input* attribute.

Args

x: PyoObject New signal to process.

fadetime: float, optional Crossfade time between old and new input. Default to 0.05.

input
PyoObject. Input signal to process.

TrackHold

class TrackHold(*input*, *controlsig*, *value=0.0*, *mul=1*, *add=0*)
Performs a track-and-hold operation on its input.

TrackHold lets pass the signal in *input* without modification but hold a sample according to the value of *controlsig*. If *controlsig* equals *value*, the input is sampled and held, otherwise, it passes thru.

Parent *PyoObject*

Args

input: PyoObject Input signal.

controlsig: PyoObject Controls when to sample the signal.

value: float or PyoObject, optional Sampling target value. Default to 0.0.

```
>>> s = Server().boot()
>>> s.start()
>>> ph = Phasor([3,4])
>>> lf = Sine(.2, mul=.5, add=.5)
>>> th = TrackHold(lf, ph > 0.5, 1, mul=500, add=300)
>>> a = Sine(th, mul=.3).out()
```

setInput(*x*, *fadetime=0.05*)
Replace the *input* attribute.

Args

x: PyoObject New signal to process.

fadetime: float, optional Crossfade time between old and new input. Default to 0.05.

setControlsig(*x*, *fadetime=0.05*)
Replace the *controlsig* attribute.

Args

x: PyoObject New control signal.

fadetime: float, optional Crossfade time between old and new input. Default to 0.05.

setValue(*x*)
Replace the *value* attribute.

Args

x: float or PyoObject New *value* attribute.

input
PyoObject. Input signal.

controlsig
PyoObject. Control signal.

value
float or PyoObject. Target value.

Resample

class Resample(*input*, *mode=1*, *mul=1*, *add=0*)
Realtime upsampling or downsampling of an audio signal.

This object should be used in the context of a resampling block created with the Server's methods *beginResamplingBlock* and *EndResamplingBlock*.

If used inside the block, it will resample its input signal according to the resampling factor given to *beginResamplingFactor*. If the factor is a negative value, the new virtual sampling rate will be *current sr / abs(factor)*. If the factor is a postive value, the new virtual sampling rate will be *current sr * factor*.

If used after *endResamplingBlock*, it will resample its input signal to the current sampling rate of the server.

The *mode* argument specifies the interpolation/decimation mode used internally.

Parent *PyoObject*

Args

input: PyoObject Input signal to resample.

mode: int, optional The interpolation/decimation mode. Defaults to 1. For the upsampling process, possible values are:

0: zero-padding 1: sample-and-hold 2 or higher: the formula *mode * resampling factor* gives the FIR lowpass kernel length used to interpolate.

For the downsampling process, possible values are: 0 or 1: discard extra samples 2 or higher: the formula *mode * abs(resampling factor)* gives the FIR lowpass kernel length used for the decimation.

```
>>> s = Server().boot()
>>> s.start()
>>> drv = Sine(.5, phase=[0, 0.5], mul=0.49, add=0.5)
>>> sig = SfPlayer(SNDS_PATH+"/transparent.aif", loop=True)
>>> s.beginResamplingBlock(8)
>>> sigup = Resample(sig, mode=32)
>>> drvup = Resample(drv, mode=1)
>>> disto = Disto(sigup, drive=drvup, mul=0.5)
>>> s.endResamplingBlock()
>>> sigdown = Resample(disto, mode=32, mul=0.4).out()
```

setMode(*x*)
Replace the *mode* attribute.

Args

x: int New *mode* attribute.

mode
int. The interpolation/decimation mode.

Tables

Tables are one-dimension containers to keep samples (sounds, envelopes, algorithmic patterns, etc.) in memory and access them quickly.

ChebyTable

class ChebyTable(*list=[1.0, 0.0], size=8192*)
Chebyshev polynomials of the first kind.

Uses Chebyshev coefficients to generate stored polynomial functions which, under waveshaping, can be used to split a sinusoid into harmonic partials having a pre-definable spectrum.

Parent *PyoTableObject*

Args

list: list, optional Relative strengths of partials numbers 1,2,3, . . . , 12 that will result when a sinusoid of amplitude 1 is waveshaped using this function table. Up to 12 partials can be specified. Defaults to [1].

size: int, optional Table size in samples. Defaults to 8192.

```
>>> s = Server().boot()
>>> s.start()
>>> t = ChebyTable([1,0,.33,0,.2,0,.143,0,.111])
>>> lfo = Sine(freq=.25, mul=0.45, add=0.5)
>>> a = Sine(freq=[200,201], mul=lfo)
>>> b = Lookup(table=t, index=a, mul=1-lfo).out()
```

autoNormalize(*x*)
Activate/deactivate automatic normalization when harmonics changed.

Args

x: boolean True for activating automatic normalization, False for deactivating it.

replace(*list*)
Redraw the waveform according to a new set of harmonics relative strengths that will result when a sinusoid of amplitude 1 is waveshaped using this function table.

Args

list: list Relative strengths of the fixed harmonic partial numbers 1,2,3, . . . , 12. Up to 12 partials can be specified.

getNormTable()
Return a DataTable filled with the normalization function corresponding to the current polynomial.

graph (*yrange=(-1.0, 1.0), title=None, wxnoserver=False*)
Opens a multislider window to control the data values.

When editing the grapher with the mouse, the new values are sent to the object to replace the table content.

Args

yrange: tuple, optional Set the min and max values of the Y axis of the multislider. Defaults to (0.0, 1.0).

title: string, optional Title of the window. If none is provided, the name of the class is used.

wxnoserver: boolean, optional With wxPython graphical toolkit, if True, tells the interpreter that there will be no server window.

If *wxnoserver* is set to True, the interpreter will not wait for the server GUI before showing the controller window.

Note: The number of bars in the graph is initialized to the length of the list of relative strentghs at the time the graph is created.

---

list
list. Relative strengths of the fixed harmonic partial numbers.

CosLogTable

class CosLogTable (*list=[(0, 0.0), (8191, 1.0)], size=8192*)
Construct a table from logarithmic-cosine segments in breakpoint fashion.

Parent *PyoTableObject*

Args

list: list, optional List of tuples indicating location and value of each points in the table. The default, [(0,0.), (8191, 1.)], creates a logarithmic line from 0.0 at location 0 to 1.0 at the end of the table (size - 1). Location must be an integer.

size: int, optional Table size in samples. Defaults to 8192.

Note: Locations in the list must be in increasing order. If the last value is less than size, the rest of the table will be filled with zeros.

Values must be greater than 0.0.

```
>>> s = Server().boot()
>>> s.start()
>>> t = CosLogTable([(0,0), (4095,1), (8192,0)])
>>> a = Osc(table=t, freq=2, mul=.25)
>>> b = SineLoop(freq=[599,600], feedback=0.05, mul=a).out()
```

replace (*list*)
Draw a new envelope according to the new *list* parameter.

Args

list: list List of tuples indicating location and value of each points in the table. Location must be integer.

loadRecFile(*filename*, *tolerance=0.02*)
Import an automation recording file in the table.

loadRecFile takes a recording file, usually from a ControlRec object, as *filename* parameter, applies a filtering pre-processing to eliminate redundancies and loads the result in the table as a list of points. Filtering process can be controled with the *tolerance* parameter.

Args

filename: string Full path of an automation recording file.

tolerance: float, optional Tolerance of the filter. A higher value will eliminate more points. Defaults to 0.02.

getPoints()
Returns list of points of the current table.

graph(*yrange=(0.0, 1.0)*, *title=None*, *wxnoserver=False*)
Opens a grapher window to control the shape of the envelope.

When editing the grapher with the mouse, the new set of points will be send to the object on mouse up.

Ctrl+C with focus on the grapher will copy the list of points to the clipboard, giving an easy way to insert the new shape in a script.

Args

yrange: tuple, optional Set the min and max values of the Y axis of the graph. Defaults to (0.0, 1.0).

title: string, optional Title of the window. If none is provided, the name of the class is used.

wxnoserver: boolean, optional With wxPython graphical toolkit, if True, tells the interpreter that there will be no server window.

If *wxnoserver* is set to True, the interpreter will not wait for the server GUI before showing the controller window.

list
list. List of tuples indicating location and value of each points in the table.

CosTable

class CosTable(*list=[(0, 0.0), (8191, 1.0)]*, *size=8192*)
Construct a table from cosine interpolated segments.

Parent `PyoTableObject`

Args

list: list, optional List of tuples indicating location and value of each points in the table. The default, [(0,0.), (8191, 1.)], creates a cosine line from 0.0 at location 0 to 1.0 at the end of the table (size - 1). Location must be an integer.

size: int, optional Table size in samples. Defaults to 8192.

Note: Locations in the list must be in increasing order. If the last value is less than size, the rest of the table will be filled with zeros.

```
>>> s = Server().boot()
>>> s.start()
>>> # Sharp attack envelope
>>> t = CosTable([(0,0), (100,1), (1000,.25), (8191,0)])
>>> a = Osc(table=t, freq=2, mul=.25)
>>> b = SineLoop(freq=[299,300], feedback=0.05, mul=a).out()
```

replace (*list*)
Draw a new envelope according to the new *list* parameter.

Args

list: list List of tuples indicating location and value of each points in the table. Location must be integer.

loadRecFile (*filename*, *tolerance=0.02*)
Import an automation recording file in the table.

loadRecFile takes a recording file, usually from a ControlRec object, as *filename* parameter, applies a filtering pre-processing to eliminate redundancies and loads the result in the table as a list of points. Filtering process can be controled with the *tolerance* parameter.

Args

filename: string Full path of an automation recording file.

tolerance: float, optional Tolerance of the filter. A higher value will eliminate more points. Defaults to 0.02.

getPoints ()
Returns list of points of the current table.

graph (*yrange=(0.0, 1.0)*, *title=None*, *wxnoserver=False*)
Opens a grapher window to control the shape of the envelope.

When editing the grapher with the mouse, the new set of points will be send to the object on mouse up.

Ctrl+C with focus on the grapher will copy the list of points to the clipboard, giving an easy way to insert the new shape in a script.

Args

yrange: tuple, optional Set the min and max values of the Y axis of the graph. Defaults to (0.0, 1.0).

title: string, optional Title of the window. If none is provided, the name of the class is used.

wxnoserver: boolean, optional With wxPython graphical toolkit, if True, tells the interpreter that there will be no server window.

If *wxnoserver* is set to True, the interpreter will not wait for the server GUI before showing the controller window.

list
list. List of tuples indicating location and value of each points in the table.

CurveTable

class CurveTable (*list=[(0, 0.0), (8191, 1.0)]*, *tension=0*, *bias=0*, *size=8192*)
Construct a table from curve interpolated segments.

CurveTable uses Hermite interpolation (sort of cubic interpolation) to calculate each points of the curve. This algorithm allows tension and biasing controls. Tension can be used to tighten up the curvature at the known points. The bias is used to twist the curve about the known points.

Parent *PyoTableObject*

Args

list: list, optional List of tuples indicating location and value of each points in the table. The default, [(0,0.), (8191, 1.)], creates a curved line from 0.0 at location 0 to 1.0 at the end of the table (size - 1). Location must be an integer.

tension: float, optional Curvature at the known points. 1 is high, 0 normal, -1 is low. Defaults to 0.

bias: float, optional Curve attraction (for each segments) toward bundary points. 0 is even, positive is towards first point, negative is towards the second point. Defaults to 0.

size: int, optional Table size in samples. Defaults to 8192.

Note: Locations in the list must be in increasing order. If the last value is less than size, the rest of the table will be filled with zeros.

High tension or bias values can create unstable or very loud table, use normalize method to keep the curve between -1 and 1.

```
>>> s = Server().boot()
>>> s.start()
>>> t = CurveTable([(0,0),(2048,.5),(4096,.2),(6144,.5),(8192,0)], 0, 20)
>>> t.normalize()
>>> a = Osc(table=t, freq=2, mul=.25)
>>> b = SineLoop(freq=[299,300], feedback=0.05, mul=a).out()
```

setTension(*x*)

Replace the *tension* attribute.

1 is high, 0 normal, -1 is low.

Args

x: float New *tension* attribute.

setBias(*x*)

Replace the *bias* attribute.

0 is even, positive is towards first point, negative is towards the second point.

Args

x: float New *bias* attribute.

replace(*list*)

Draw a new envelope according to the new *list* parameter.

Args

list: list List of tuples indicating location and value of each points in the table. Location must be integer.

loadRecFile(*filename*, *tolerance=0.02*)

Import an automation recording file in the table.

loadRecFile takes a recording file, usually from a ControlRec object, as *filename* parameter, applies a filtering pre-processing to eliminate redundancies and loads the result in the table as a list of points. Filtering process can be controled with the *tolerance* parameter.

Args

filename: string Full path of an automation recording file.

tolerance: float, optional Tolerance of the filter. A higher value will eliminate more points. Defaults to 0.02.

getPoints()
Returns list of points of the current table.

graph(*yrange=(0.0, 1.0)*, *title=None*, *wxnoserver=False*)
Opens a grapher window to control the shape of the envelope.

When editing the grapher with the mouse, the new set of points will be send to the object on mouse up.

Ctrl+C with focus on the grapher will copy the list of points to the clipboard, giving an easy way to insert the new shape in a script.

Args

yrange: tuple, optional Set the min and max values of the Y axis of the graph. Defaults to (0.0, 1.0).

title: string, optional Title of the window. If none is provided, the name of the class is used.

wxnoserver: boolean, optional With wxPython graphical toolkit, if True, tells the interpreter that there will be no server window.

If *wxnoserver* is set to True, the interpreter will not wait for the server GUI before showing the controller window.

tension
float. Curvature tension.

bias
float. Curve Attraction.

list
list. List of tuples indicating location and value of each points in the table.

DataTable

class DataTable(*size*, *chnls=1*, *init=None*)
Create an empty table ready for data recording.

See *TableRec* to write samples in the table.

Parent *PyoTableObject*

Args

size: int Size of the table in samples.

chnls: int, optional Number of channels that will be handled by the table. Defaults to 1.

init: list of floats, optional Initial table. List of list can match the number of channels, otherwise, the list will be loaded in all tablestreams.

See also:

NewTable, *TableRec*

```
>>> s = Server().boot()
>>> s.start()
>>> import random
>>> notes = [midiToHz(random.randint(48,72)) for i in range(10)]
>>> tab = DataTable(size=10, init=notes)
>>> ind = RandInt(10, 8)
>>> pit = TableIndex(tab, ind)
>>> a = SineLoop(freq=[pit,pit*0.99], feedback = 0.07, mul=.2).out()
```

replace(*x*)
Replaces the actual table.

Args

x: list of floats New table. Must be of the same size as the actual table.

List of list can match the number of channels, otherwise, the list will be loaded in all tablestreams.

getRate()
Returns the frequency (cycle per second) to give to an oscillator to read the sound at its original pitch.

graph(*yrange=(0.0, 1.0)*, *title=None*, *wxnoserver=False*)
Opens a multislider window to control the data values.

When editing the grapher with the mouse, the new values are sent to the object to replace the table content.

Args

yrange: tuple, optional Set the min and max values of the Y axis of the multislider. Defaults to (0.0, 1.0).

title: string, optional Title of the window. If none is provided, the name of the class is used.

wxnoserver: boolean, optional With wxPython graphical toolkit, if True, tells the interpreter that there will be no server window.

If *wxnoserver* is set to True, the interpreter will not wait for the server GUI before showing the controller window.

size
int. Length of the table in samples.

chnls
int. Number of channels that will be handled by the table.

init
list of floats. Initial table.

ExpTable

class ExpTable(*list=[(0, 0.0), (8192, 1.0)]*, *exp=10*, *inverse=True*, *size=8192*)
Construct a table from exponential interpolated segments.

Parent *PyoTableObject*

Args

list: list, optional List of tuples indicating location and value of each points in the table. The default, [(0,0.), (8192, 1.)], creates a exponential line from 0.0 at location 0 to 1.0 at the end of the table. Location must be an integer.

exp: float, optional Exponent factor. Used to control the slope of the curve. Defaults to 10.

inverse: boolean, optional If True, downward slope will be inversed. Useful to create biexponential curves. Defaults to True.

size: int, optional Table size in samples. Defaults to 8192.

Note: Locations in the list must be in increasing order. If the last value is less than size, the rest of the table will be filled with zeros.

```
>>> s = Server().boot()
>>> s.start()
>>> t = ExpTable([(0,0),(4096,1),(8192,0)], exp=5, inverse=True)
>>> a = Osc(table=t, freq=2, mul=.3)
>>> b = Sine(freq=[299,300], mul=a).out()
```

setExp(*x*)

Replace the *exp* attribute.

Args

x: float New *exp* attribute.

setInverse(*x*)

Replace the *inverse* attribute.

Args

x: boolean New *inverse* attribute.

replace(*list*)

Draw a new envelope according to the new *list* parameter.

Args

list: list List of tuples indicating location and value of each points in the table. Location must be integer.

loadRecFile(*filename*, *tolerance=0.02*)

Import an automation recording file in the table.

loadRecFile takes a recording file, usually from a ControlRec object, as *filename* parameter, applies a filtering pre-processing to eliminate redundancies and loads the result in the table as a list of points. Filtering process can be controled with the *tolerance* parameter.

Args

filename: string Full path of an automation recording file.

tolerance: float, optional Tolerance of the filter. A higher value will eliminate more points. Defaults to 0.02.

getPoints()

Returns list of points of the current table.

graph(*yrange=(0.0, 1.0)*, *title=None*, *wxnoserver=False*)

Opens a grapher window to control the shape of the envelope.

When editing the grapher with the mouse, the new set of points will be send to the object on mouse up.

Ctrl+C with focus on the grapher will copy the list of points to the clipboard, giving an easy way to insert the new shape in a script.

Args

yrange: tuple, optional Set the min and max values of the Y axis of the graph. Defaults to (0.0, 1.0).

title: string, optional Title of the window. If none is provided, the name of the class is used.

wxnoserver: boolean, optional With wxPython graphical toolkit, if True, tells the interpreter that there will be no server window.

If *wxnoserver* is set to True, the interpreter will not wait for the server GUI before showing the controller window.

exp
float. Exponent factor.

inverse
boolean. Inverse factor.

list
list. List of tuples indicating location and value of each points in the table.

HannTable

class HannTable(*size=8192*)
Generates Hanning window function.

Parent *PyoTableObject*

Args

size: int, optional Table size in samples. Defaults to 8192.

```
>>> s = Server().boot()
>>> s.start()
>>> # Hanning envelope
>>> t = HannTable()
>>> a = Osc(table=t, freq=2, mul=.2)
>>> b = Sine(freq=[299,300], mul=a).out()
```

HarmTable

class HarmTable(*list=[1.0, 0.0], size=8192*)
Harmonic waveform generator.

Generates composite waveforms made up of weighted sums of simple sinusoids.

Parent *PyoTableObject*

Args

list: list, optional Relative strengths of the fixed harmonic partial numbers 1,2,3, etc. Defaults to [1].

size: int, optional Table size in samples. Defaults to 8192.

```
>>> s = Server().boot()
>>> s.start()
>>> # Square wave up to 9th harmonic
>>> t = HarmTable([1,0,.33,0,.2,0,.143,0,.111])
>>> a = Osc(table=t, freq=[199,200], mul=.2).out()
```

autoNormalize(*x*)
Activate/deactivate automatic normalization when harmonics changed.

Args

x: boolean True for activating automatic normalization, False for deactivating it.

replace(*list*)
Redraw the waveform according to a new set of harmonics relative strengths.

Args

list: list Relative strengths of the fixed harmonic partial numbers 1,2,3, etc.

graph(*yrange=(-1.0, 1.0)*, *title=None*, *wxnoserver=False*)
Opens a multislider window to control the data values.

When editing the grapher with the mouse, the new values are sent to the object to replace the table content.

Args

yrange: tuple, optional Set the min and max values of the Y axis of the multislider. Defaults to (0.0, 1.0).

title: string, optional Title of the window. If none is provided, the name of the class is used.

wxnoserver: boolean, optional With wxPython graphical toolkit, if True, tells the interpreter that there will be no server window.

If *wxnoserver* is set to True, the interpreter will not wait for the server GUI before showing the controller window.

Note: The number of bars in the graph is initialized to the length of the list of relative strentghs at the time the graph is created.

---

list
list. Relative strengths of the fixed harmonic partial numbers.

LinTable

class LinTable(*list=[(0, 0.0), (8191, 1.0)]*, *size=8192*)
Construct a table from segments of straight lines in breakpoint fashion.

Parent *PyoTableObject*

Args

list: list, optional List of tuples indicating location and value of each points in the table. The default, [(0,0.), (8191, 1.)], creates a straight line from 0.0 at location 0 to 1.0 at the end of the table (size - 1). Location must be an integer.

size: int, optional Table size in samples. Defaults to 8192.

Note: Locations in the list must be in increasing order. If the last value is less than size, the rest of the table will be filled with zeros.

```
>>> s = Server().boot()
>>> s.start()
>>> # Sharp attack envelope
>>> t = LinTable([(0,0), (100,1), (1000,.25), (8191,0)])
>>> a = Osc(table=t, freq=2, mul=.25)
>>> b = SineLoop(freq=[299,300], feedback=0.05, mul=a).out()
```

replace(*list*)

Draw a new envelope according to the new *list* parameter.

Args

list: list List of tuples indicating location and value of each points in the table. Location must be integer.

loadRecFile(*filename*, *tolerance=0.02*)

Import an automation recording file in the table.

loadRecFile takes a recording file, usually from a ControlRec object, as *filename* parameter, applies a filtering pre-processing to eliminate redundancies and loads the result in the table as a list of points. Filtering process can be controled with the *tolerance* parameter.

Args

filename: string Full path of an automation recording file.

tolerance: float, optional Tolerance of the filter. A higher value will eliminate more points. Defaults to 0.02.

getPoints()

Returns list of points of the current table.

graph(*yrange=(0.0, 1.0)*, *title=None*, *wxnoserver=False*)

Opens a grapher window to control the shape of the envelope.

When editing the grapher with the mouse, the new set of points will be send to the object on mouse up.

Ctrl+C with focus on the grapher will copy the list of points to the clipboard, giving an easy way to insert the new shape in a script.

Args

yrange: tuple, optional Set the min and max values of the Y axis of the graph. Defaults to (0.0, 1.0).

title: string, optional Title of the window. If none is provided, the name of the class is used.

wxnoserver: boolean, optional With wxPython graphical toolkit, if True, tells the interpreter that there will be no server window.

If *wxnoserver* is set to True, the interpreter will not wait for the server GUI before showing the controller window.

list

list. List of tuples indicating location and value of each points in the table.

LogTable

class LogTable(*list=[(0, 0.0), (8191, 1.0)], size=8192*)

Construct a table from logarithmic segments in breakpoint fashion.

Parent *PyoTableObject*

Args

list: list, optional List of tuples indicating location and value of each points in the table. The default, [(0,0.), (8191, 1.)], creates a logarithmic line from 0.0 at location 0 to 1.0 at the end of the table (size - 1). Location must be an integer.

size: int, optional Table size in samples. Defaults to 8192.

Note: Locations in the list must be in increasing order. If the last value is less than size, the rest of the table will be filled with zeros.

Values must be greater than 0.0.

```
>>> s = Server().boot()
>>> s.start()
>>> t = LogTable([(0,0), (4095,1), (8192,0)])
>>> a = Osc(table=t, freq=2, mul=.25)
>>> b = SineLoop(freq=[599,600], feedback=0.05, mul=a).out()
```

replace(*list*)

Draw a new envelope according to the new *list* parameter.

Args

list: list List of tuples indicating location and value of each points in the table. Location must be integer.

loadRecFile(*filename*, *tolerance=0.02*)

Import an automation recording file in the table.

loadRecFile takes a recording file, usually from a ControlRec object, as *filename* parameter, applies a filtering pre-processing to eliminate redundancies and loads the result in the table as a list of points. Filtering process can be controled with the *tolerance* parameter.

Args

filename: string Full path of an automation recording file.

tolerance: float, optional Tolerance of the filter. A higher value will eliminate more points. Defaults to 0.02.

getPoints()

Returns list of points of the current table.

graph(*yrange=(0.0, 1.0), title=None, wxnoserver=False*)

Opens a grapher window to control the shape of the envelope.

When editing the grapher with the mouse, the new set of points will be send to the object on mouse up.

Ctrl+C with focus on the grapher will copy the list of points to the clipboard, giving an easy way to insert the new shape in a script.

Args

yrange: tuple, optional Set the min and max values of the Y axis of the graph. Defaults to (0.0, 1.0).

title: string, optional Title of the window. If none is provided, the name of the class is used.

wxnoserver: boolean, optional With wxPython graphical toolkit, if True, tells the interpreter that there will be no server window.

If *wxnoserver* is set to True, the interpreter will not wait for the server GUI before showing the controller window.

list
list. List of tuples indicating location and value of each points in the table.

NewTable

class NewTable(*length*, *chnls=1*, *init=None*, *feedback=0.0*)
Create an empty table ready for recording.

See *TableRec* to write samples in the table.

Parent *PyoTableObject*

Args

length: float Length of the table in seconds.

chnls: int, optional Number of channels that will be handled by the table. Defaults to 1.

init: list of floats, optional Initial table. List of list can match the number of channels, otherwise, the list will be loaded in all tablestreams. Defaults to None.

feedback: float, optional Amount of old data to mix with a new recording. Defaults to 0.0.

See also:

DataTable, *TableRec*

```
>>> s = Server(duplex=1).boot()
>>> s.start()
>>> t = NewTable(length=2, chnls=1)
>>> a = Input(0)
>>> b = TableRec(a, t, .01)
>>> amp = Iter(b["trig"], [.5])
>>> freq = t.getRate()
>>> c = Osc(table=t, freq=[freq, freq*.99], mul=amp).out()
>>> # to record in the empty table, call:
>>> # b.play()
```

replace(*x*)
Replaces the actual table.

Args

x: list of floats New table. Must be of the same size as the actual table.

List of list can match the number of channels, otherwise, the list will be loaded in all tablestreams.

setFeedback(*x*)
Replaces the‘feedback‘ attribute.

Args

x: float New *feedback* value.

getLength()
Returns the length of the table in seconds.

getDur(*all=True*)
Returns the length of the table in seconds.

The *all* argument is there for compatibility with SndTable but is not used for now.

getRate()
Returns the frequency (cycle per second) to give to an oscillator to read the sound at its original pitch.

getViewTable(*size*, *begin=0*, *end=0*)
Return a list of points (in X, Y pixel values) for each channel in the table. These lists can be draw on a DC (WxPython) with a DrawLines method.

Args

size: tuple Size, (X, Y) pixel values, of the waveform container window.

begin: float, optional First position in the the table, in seconds, where to get samples. Defaults to 0.

end: float, optional Last position in the table, in seconds, where to get samples.

if this value is set to 0, that means the end of the table. Defaults to 0.

view(*title='Sound waveform'*, *wxnoserver=False*, *mouse_callback=None*)
Opens a window showing the contents of the table.

Args

title: string, optional Window title. Defaults to "Table waveform".

wxnoserver: boolean, optional With wxPython graphical toolkit, if True, tells the interpreter that there will be no server window.

mouse_callback: callable If provided, this function will be called with the mouse position, inside the frame, as argument. Defaults to None.

If *wxnoserver* is set to True, the interpreter will not wait for the server GUI before showing the controller window.

refreshView()
Updates the graphical display of the table, if applicable.

length
float. Length of the table in seconds.

chnls
int. Number of channels that will be handled by the table.

init
list of floats. Initial table.

feedback
float. Amount of old data to mix with a new recording.

size
int. Table size in samples.

ParaTable

class ParaTable(*size=8192*)

Generates parabola window function.

The parabola is a conic section, the intersection of a right circular conical surface and a plane parallel to a generating straight line of that surface.

Parent *PyoTableObject*

Args

size: int, optional Table size in samples. Defaults to 8192.

```
>>> s = Server().boot()
>>> s.start()
>>> # Parabola envelope
>>> t = ParaTable()
>>> a = Osc(table=t, freq=2, mul=.2)
>>> b = SineLoop(freq=[299,300], feedback=0.05, mul=a).out()
```

PartialTable

class PartialTable(*list=[(1, 1), (1.33, 0.5), (1.67, 0.3)], size=65536*)

Inharmonic waveform generator.

Generates waveforms made of inharmonic components. Partials are given as a list of 2-values tuple, where the first one is the partial number (can be float) and the second one is the strength of the partial.

The object uses the first two decimal values of each partial to compute a higher harmonic at a multiple of 100 (so each component is in reality truly harmonic). If the oscillator has a frequency divided by 100, the real desired partials will be restituted.

The list:

[(1, 1), (1.1, 0.7), (1.15, 0.5)] will draw a table with:

harmonic 100: amplitude = 1 harmonic 110: amplitude = 0.7 harmonic 115: amplitude = 0.5

To listen to a signal composed of 200, 220 and 230 Hz, one should declared an oscillator like this (frequency of 200Hz divided by 100):

a = Osc(t, freq=2, mul=0.5).out()

Parent *PyoTableObject*

Args

list: list of tuple, optional List of 2-values tuples. First value is the partial number (float up to two decimal values) and second value is its amplitude (relative to the other harmonics). Defaults to [(1,1), (1.33,0.5),(1.67,0.3)].

size: int, optional Table size in samples. Because computed harmonics are very high in frequency, the table size must be bigger than a classic HarmTable. Defaults to 65536.

```
>>> s = Server().boot()
>>> s.start()
>>> t = PartialTable([(1,1), (2.37, 0.5), (4.55, 0.3)]).normalize()
>>> # Play with fundamentals 199 and 200 Hz
>>> a = Osc(table=t, freq=[1.99,2], mul=.2).out()
```

replace (*list*)
Redraw the waveform according to a new set of harmonics relative strengths.

Args

list: list of tuples Each tuple contains the partial number, as a float, and its strength.

list
list. List of partial numbers and strength.

SawTable

class SawTable (*order=10*, *size=8192*)
Sawtooth waveform generator.

Generates sawtooth waveforms made up of fixed number of harmonics.

Parent *PyoTableObject*

Args

order: int, optional Number of harmonics sawtooth is made of. Defaults to 10.

size: int, optional Table size in samples. Defaults to 8192.

```
>>> s = Server().boot()
>>> s.start()
>>> t = SawTable(order=12).normalize()
>>> a = Osc(table=t, freq=[199,200], mul=.2).out()
```

setOrder (*x*)
Change the *order* attribute and redraw the waveform.

Args

x: int New number of harmonics

order
int. Number of harmonics sawtooth is made of.

SincTable

class SincTable (*freq=6.283185307179586*, *windowed=False*, *size=8192*)
Generates sinc window function.

Parent *PyoTableObject*

Args

freq: float, optional Frequency, in radians, of the sinc function. Defaults to pi*2.

windowed: boolean, optional If True, an hanning window is applied on the sinc function. Defaults to False.

size: int, optional Table size in samples. Defaults to 8192.

```
>>> import math
>>> s = Server().boot()
>>> s.start()
>>> t = SincTable(freq=math.pi*6, windowed=True)
>>> a = Osc(t, freq=[199,200], mul=.2).out()
```

setFreq(*x*)
Change the frequency of the sinc function. This will redraw the envelope.

Args

x: float New frequency in radians.

setWindowed(*x*)
Change the windowed flag. This will redraw the envelope.

Args

x: boolean New windowed flag.

freq
float. Frequency of the sinc function.

windowed
boolean. Windowed flag.

SndTable

class SndTable(*path=None*, *chnl=None*, *start=0*, *stop=None*, *initchnls=1*)
Transfers data from a soundfile into a function table.

If *chnl* is None, the table will contain as many table streams as necessary to read all channels of the loaded sound.

Parent *PyoTableObject*

Args

path: string, optional Full path name of the sound. The defaults, None, creates an empty table.

chnl: int, optional Channel number to read in. The count starts at 0 (first channel is is 0, second is 1 and so on). Available at initialization time only. The default (None) reads all channels.

start: float, optional Begins reading at *start* seconds into the file. Available at initialization time only. Defaults to 0.

stop: float, optional Stops reading at *stop* seconds into the file. Available at initialization time only. The default (None) means the end of the file.

```
>>> s = Server().boot()
>>> s.start()
>>> snd_path = SNDS_PATH + '/transparent.aif'
>>> t = SndTable(snd_path)
>>> freq = t.getRate()
>>> a = Osc(table=t, freq=[freq, freq*.995], mul=.3).out()
```

setSound(*path*, *start=0*, *stop=None*)
Load a new sound in the table.

Keeps the number of channels of the sound loaded at initialization. If the new sound has less channels, it will wrap around and load the same channels many times. If the new sound has more channels, the extra channels will be skipped.

Args

path: string Full path of the new sound.

start: float, optional Begins reading at *start* seconds into the file. Defaults to 0.

stop: float, optional Stops reading at *stop* seconds into the file. The default (None) means the end of the file.

append(*path*, *crossfade=0*, *start=0*, *stop=None*)
Append a sound to the one already in the table with crossfade.

Keeps the number of channels of the sound loaded at initialization. If the new sound has less channels, it will wrap around and load the same channels many times. If the new sound has more channels, the extra channels will be skipped.

Args

path: string Full path of the new sound.

crossfade: float, optional Crossfade time, in seconds, between the sound already in the table and the new one. Defaults to 0.

start: float, optional Begins reading at *start* seconds into the file. Defaults to 0.

stop: float, optional Stops reading at *stop* seconds into the file. The default, None, means the end of the file.

insert(*path*, *pos=0*, *crossfade=0*, *start=0*, *stop=None*)
Insert a sound into the one already in the table with crossfade.

Insert a sound at position *pos*, specified in seconds, with crossfading at the beginning and the end of the insertion.

Keeps the number of channels of the sound loaded at initialization. If the new sound has less channels, it will wrap around and load the same channels many times. If the new sound has more channels, the extra channels will be skipped.

Args

path: string Full path of the new sound.

pos: float, optional Position in the table, in seconds, where to insert the new sound. Defaults to 0.

crossfade: float, optional Crossfade time, in seconds, between the sound already in the table and the new one. Defaults to 0.

start: float, optional Begins reading at *start* seconds into the file. Defaults to 0.

stop: float, optional Stops reading at *stop* seconds into the file. The default, None, means the end of the file.

getRate(*all=True*)
Return the frequency in cps at which the sound will be read at its original pitch.

Args

all: boolean If the table contains more than one sound and *all* is True, returns a list of all durations. Otherwise, returns only the first duration as a float.

getDur(*all=True*)
Return the duration of the sound in seconds.

Args

all: boolean If the table contains more than one sound and *all* is True, returns a list of all durations. Otherwise, returns only the first duration as a float.

setSize(*x*)
Change the size of the table.

This will erase the previously drawn waveform.

Args

size: int New table size in samples.

getSize(*all=True*)
Return the size of the table in samples.

Args

all: boolean If the table contains more than one sound and *all* is True, returns a list of all sizes. Otherwise, returns only the first size as an int.

getViewTable(*size*, *begin=0*, *end=0*)
Return a list of points (in X, Y pixel values) for each channel in the table. These lists can be draw on a DC (WxPython) with a DrawLines method.

Args

size: tuple Size, (X, Y) pixel values, of the waveform container window.

begin: float, optional First position in the the table, in seconds, where to get samples. Defaults to 0.

end: float, optional Last position in the table, in seconds, where to get samples.

if this value is set to 0, that means the end of the table. Defaults to 0.

getEnvelope(*points*)
Return the amplitude envelope of the table.

Return a list, of length *chnl*, of lists of length *points* filled with the amplitude envelope of the table.

Args

points: int Number of points of the amplitude analysis.

view(*title='Sound waveform'*, *wxnoserver=False*, *mouse_callback=None*)
Opens a window showing the contents of the table.

Args

title: string, optional Window title. Defaults to "Table waveform".

wxnoserver: boolean, optional With wxPython graphical toolkit, if True, tells the interpreter that there will be no server window.

mouse_callback: callable If provided, this function will be called with the mouse position, inside the frame, as argument. Defaults to None.

If *wxnoserver* is set to True, the interpreter will not wait for the server GUI before showing the controller window.

refreshView()
Updates the graphical display of the table, if applicable.

sound
string. Full path of the sound.

path
string. Full path of the sound.

chnl
int. Channel to read in.

start
float. Start point, in seconds, to read into the file.

stop
float. Stop point, in seconds, to read into the file.

size
int. Table size in samples.

SquareTable

class SquareTable(*order=10*, *size=8192*)
Square waveform generator.

Generates square waveforms made up of fixed number of harmonics.

Parent *PyoTableObject*

Args

order: int, optional Number of harmonics square waveform is made of. The waveform will contains *order* odd harmonics. Defaults to 10.

size: int, optional Table size in samples. Defaults to 8192.

```
>>> s = Server().boot()
>>> s.start()
>>> t = SquareTable(order=15).normalize()
>>> a = Osc(table=t, freq=[199,200], mul=.2).out()
```

setOrder(*x*)
Change the *order* attribute and redraw the waveform.

Args

x: int New number of harmonics

order
int. Number of harmonics square waveform is made of.

WinTable

class WinTable(*type=2*, *size=8192*)
Generates different kind of windowing functions.

Parent *PyoTableObject*

Args

type: int, optional

Windowing function. Possible choices are:

0. Rectangular (no window)
1. Hamming
2. Hanning (default)

3. Bartlett (triangular)
4. Blackman 3-term
5. Blackman-Harris 4-term
6. Blackman-Harris 7-term
7. Tuckey (alpha = 0.66)
8. Sine (half-sine window)

size: int, optional Table size in samples. Defaults to 8192.

```
>>> s = Server().boot()
>>> s.start()
>>> # Triangular envelope
>>> t = WinTable(type=3)
>>> a = Osc(table=t, freq=2, mul=.2)
>>> b = SineLoop(freq=[199,200], feedback=0.05, mul=a).out()
```

setType(*type*)
Sets the windowing function.

Args

type: int {0 -> 8} Windowing function.

type
int. Windowing function.

AtanTable

class AtanTable(*slope=0.5*, *size=8192*)
Generates an arctangent transfert function.

This table allow the creation of the classic arctangent transfert function, useful in distortion design. See Lookup object for a simple table lookup process.

Parent *PyoTableObject*

Args

slope: float, optional Slope of the arctangent function, between 0 and 1. Defaults to 0.5.

size: int, optional Table size in samples. Defaults to 8192.

```
>>> import math
>>> s = Server().boot()
>>> s.start()
>>> t = AtanTable(slope=0.8)
>>> a = Sine(freq=[149,150])
>>> l = Lookup(table=t, index=a, mul=0.3).out()
```

setSlope(*x*)
Change the slope of the arctangent function. This will redraw the table.

Args

x: float New slope between 0 and 1.

slope
float. slope of the arctangent function.

PadSynthTable

class PadSynthTable(*basefreq=440*, *spread=1*, *bw=50*, *bwscl=1*, *nharms=64*, *damp=0.7*, *size=262144*)

Generates wavetable with the PadSynth algorithm from Nasca Octavian Paul.

This object generates a wavetable with the PadSynth algorithm describe here:

http://zynaddsubfx.sourceforge.net/doc/PADsynth/PADsynth.htm

This algorithm generates some large wavetables that can be played at different speeds to get the desired sound. This algorithm describes only how these wavetables are generated. The result is a perfectly looped wavetable.

To get the desired pitch from the table, the playback speed must be *sr / table size*. This speed can be transposed to obtain different pitches from a single wavetable.

Parent *PyoTableObject*

Args

basefreq: float, optional The base frequency of the algorithm in Hz. If the spreading factor is near 1.0, this frequency is the fundamental of the spectrum. Defaults to 440.

spread: float, optional The spreading factor for the harmonics. Each harmonic real frequency is computed as *basefreq * pow(n, spread)* where *n* is the harmonic order. Defaults to 1.

bw: float, optional The bandwidth of the first harmonic in cents. The bandwidth allows to control the harmonic profile using a gaussian distribution (bell shape). Defaults to 50.

bwscl: float, optional The bandswidth scale specifies how much the bandwidth of the harmonic increase according to its frequency. Defaults to 1.

nharms: int, optional The number of harmonics in the generated wavetable. Higher numbers of harmonics take more time to generate the wavetable. Defaults to 64.

damp: float, optional The amplitude damping factor specifies how much the amplitude of the harmonic decrease according to its order. It uses a simple power serie, *amp = pow(damp, n)* where *n* is the harmonic order. Defaults to 0.7.

size: int, optional Table size in samples. Must be a power-of-two, usually a big one! Defaults to 262144.

Note: Many thanks to Nasca Octavian Paul for making this beautiful algorithm and releasing it under Public Domain.

```
>>> s = Server().boot()
>>> s.start()
>>> f = s.getSamplingRate() / 262144
>>> t = PadSynthTable(basefreq=midiToHz(48), spread=1.205, bw=10, bwscl=1.5)
>>> a = Osc(table=t, freq=f, phase=[0, 0.5], mul=0.5).out()
```

setBaseFreq(*x*, *generate=True*)

Change the base frequency of the algorithm.

Args

x: float New base frequency in Hz.

generate: boolean, optional If True, a new table will be computed with changed value.

setSpread(*x*, *generate=True*)
Change the frequency spreading factor of the algorithm.

Args

x: float New spread factor.

generate: boolean, optional If True, a new table will be computed with changed value.

setBw(*x*, *generate=True*)
Change the bandwidth of the first harmonic.

Args

x: float New bandwidth in cents.

generate: boolean, optional If True, a new table will be computed with changed value.

setBwScl(*x*, *generate=True*)
Change the bandwidth scaling factor.

Args

x: float New bandwidth scaling factor.

generate: boolean, optional If True, a new table will be computed with changed value.

setNharms(*x*, *generate=True*)
Change the number of harmonics.

Args

x: int New number of harmonics.

generate: boolean, optional If True, a new table will be computed with changed value.

setDamp(*x*, *generate=True*)
Change the amplitude damping factor.

Args

x: float New amplitude damping factor.

generate: boolean, optional If True, a new table will be computed with changed value.

setSize(*size*, *generate=True*)
Change the size of the table.

This will erase the previously drawn waveform.

Args

size: int New table size in samples. Must be a power-of-two.

generate: boolean, optional If True, a new table will be computed with changed value.

basefreq
float. Base frequency in Hz.

spread
float. Frequency spreading factor.

bw
float. Bandwitdh of the first harmonic in cents.

bwscl
float. Bandwitdh scaling factor.

nharms
int. Number of harmonics.

damp
float. Amplitude damping factor.

SharedTable

class SharedTable(*name*, *create*, *size*)
Create an inter-process shared memory table.

This table uses the given name to open an internal shared memory object, used as the data memory of the table. Two or more tables from different processes, if they use the same name, can read and write to the same memory space.

Note: SharedTable is not implemented yet for Windows (unix only).

Parent *PyoTableObject*

Args

name: string Unique name in the system shared memory. Two or more tables created with the same name will shared the same memory space. The name must conform to the construction rules for a unix pathname (ie. it must begin with a slash). Available at initialization time only.

create: boolean If True, an entry will be create in the system shared memory. If False, the object will use an already created shared memory. Can't be a list. Available at initialization time only.

size: int Size of the table in samples. Can't be a list. Available at initialization time only.

```
>>> s = Server().boot()
>>> s.start()
>>> # Creating parent table.
>>> table = SharedTable(["/sharedl", "/sharedr"], True, s.getBufferSize())
>>> # Creating child table.
>>> shared = SharedTable(["/sharedl", "/sharedr"], False, s.getBufferSize())
>>> # Read and output the content of the parent table.
>>> tabread = TableRead(table, table.getRate(), True).out()
>>> # Record some signal signal in the child table.
>>> lfo = Sine(freq=[0.2, 0.25]).range(98, 102)
>>> wave = LFO(freq=lfo, type=4, sharp=0.7, mul=0.3)
>>> pos = Phasor(shared.getRate())
>>> record = TableWrite(wave, pos, shared)
```

getRate()
Returns the frequency (cycle per second) to give to an oscillator to read the sound at its original pitch.

size
int. Length of the table in samples.

Matrices

Matrices are two-dimensions containers to keep samples (sounds, envelopes, algorithmic patterns, images, etc.) in memory and access them quickly.

NewMatrix

class NewMatrix(*width*, *height*, *init=None*)

Create a new matrix ready for recording.

Optionally, the matrix can be filled with the contents of the *init* parameter.

See *MatrixRec* to write samples in the matrix.

Parent *PyoMatrixObject*

Args

width: int Desired matrix width in samples.

height: int Desired matrix height in samples.

init: list of list of floats, optional Initial matrix. Defaults to None.

See also:

MatrixRec

```
>>> s = Server().boot()
>>> s.start()
>>> SIZE = 256
>>> mm = NewMatrix(SIZE, SIZE)
>>> mm.genSineTerrain(freq=2, phase=16)
>>> lfw = Sine([.1,.11], 0, .124, .25)
>>> lfh = Sine([.15,.16], 0, .124, .25)
>>> w = Sine(100, 0, lfw, .5)
>>> h = Sine(10.5, 0, lfh, .5)
>>> c = MatrixPointer(mm, w, h, mul=.2).out()
```

replace(*x*)

Replaces the actual matrix.

Args

x: list of list of floats New matrix. Must be of the same size as the actual matrix.

getRate()

Returns the frequency (cycle per second) to give to an oscillator to read the sound at its original pitch.

genSineTerrain(*freq=1*, *phase=0.0625*)

Generates a modulated sinusoidal terrain.

Args

freq: float Frequency of sinusoids used to created the terrain. Defaults to 1.

phase: float Phase deviation between rows of the terrain. Should be in the range 0 -> 1. Defaults to 0.0625.

Value Converters (SLMap)

These objects are used to convert values between different ranges.

Map

class Map(*min*, *max*, *scale*)
Converts value between 0 and 1 on various scales.

Base class for Map objects.

Args

min: int or float Lowest value of the range.

max: int or float Highest value of the range.

scale: string {'lin', 'log'} Method used to scale the input value on the specified range.

```
>>> m = Map(20., 20000., 'log')
>>> print(m.get(.5))
632.455532034
>>> print(m.set(12000))
0.926050416795
```

get(*x*)
Takes *x* between 0 and 1 and returns scaled value.

set(*x*)
Takes *x* in the real range and returns value unscaled (between 0 and 1).

setMin(*x*)
Replace the *min* attribute.

Args

x: float New *min* attribute.

setMax(*x*)
Replace the *max* attribute.

Args

x: float New *max* attribute.

setScale(*x*)
Replace the *scale* attribute.

Args

x: string New *scale* attribute.

min
int or float. Lowest value of the range.

max
int or float. Highest value of the range.

scale
string. Method used to scale the input value.

SLMap

class SLMap(*min*, *max*, *scale*, *name*, *init*, *res='float'*, *ramp=0.025*, *dataOnly=False*)

Base Map class used to manage control sliders.

Derived from Map class, a few parameters are added for sliders initialization.

Parent *Map*

Args

min: int or float Smallest value of the range.

max: int or float Highest value of the range.

scale: string {'lin', 'log'} Method used to scale the input value on the specified range.

name: string Name of the attributes the slider is affected to.

init: int or float Initial value. Specified in the real range, not between 0 and 1. Use *set* method to retreive the normalized corresponding value.

res: string {'int', 'float'}, optional Sets the resolution of the slider. Defaults to 'float'.

ramp: float, optional Ramp time, in seconds, used to smooth the signal sent from slider to object's attribute. Defaults to 0.025.

dataOnly: boolean, optional Set this argument to True if the parameter does not accept audio signal as control but discreet values. If True, label will be marked with a star symbol (*). Defaults to False.

```
>>> s = Server().boot()
>>> s.start()
>>> ifs = [350,360,375,388]
>>> slmapfreq = SLMap(20., 2000., 'log', 'freq', ifs)
>>> slmapfeed = SLMap(0, 0.25, 'lin', 'feedback', 0)
>>> maps = [slmapfreq, slmapfeed, SLMapMul(.1)]
>>> a = SineLoop(freq=ifs, mul=.1).out()
>>> a.ctrl(maps)
```

name
string. Name of the parameter to control.

init
float. Initial value of the slider.

res
string. Slider resolution {int or float}.

ramp
float. Ramp time in seconds.

dataOnly
boolean. True if argument does not accept audio stream.

SLMapFreq

class SLMapFreq(*init=1000*)

SLMap with normalized values for a 'freq' slider.

Parent *SLMap*

Args

init: int or float, optional Initial value. Specified in the real range, not between 0 and 1. Defaults to 1000.

Note: SLMapFreq values are:

- min = 20.0
- max = 20000.0
- scale = 'log'
- name = 'freq'
- res = 'float'
- ramp = 0.025

SLMapMul

class SLMapMul(*init=1.0*)

SLMap with normalized values for a 'mul' slider.

Parent *SLMap*

Args

init: int or float, optional Initial value. Specified in the real range, not between 0 and 1. Defaults to 1.

Note: SLMapMul values are:

- min = 0.0
- max = 2.0
- scale = 'lin'
- name = 'mul'
- res = 'float'
- ramp = 0.025

SLMapPhase

class SLMapPhase(*init=0.0*)

SLMap with normalized values for a 'phase' slider.

Parent *SLMap*

Args

init: int or float, optional Initial value. Specified in the real range, not between 0 and 1. Defaults to 0.

Note: SLMapPhase values are:

- min = 0.0
- max = 1.0
- scale = 'lin'
- name = 'phase'
- res = 'float'
- ramp = 0.025

SLMapQ

class SLMapQ(*init=1.0*)

SLMap with normalized values for a 'q' slider.

Parent *SLMap*

Args

init: int or float, optional Initial value. Specified in the real range, not between 0 and 1. Defaults to 1.

Note: SLMapQ values are:

- min = 0.1
- max = 100.0
- scale = 'log'
- name = 'q'
- res = 'float'
- ramp = 0.025

SLMapDur

class SLMapDur(*init=1.0*)

SLMap with normalized values for a 'dur' slider.

Parent *SLMap*

Args

init: int or float, optional Initial value. Specified in the real range, not between 0 and 1. Defaults to 1.

Note: SLMapDur values are:

- min = 0.
- max = 60.0
- scale = 'lin'
- name = 'dur'

- res = 'float'
- ramp = 0.025

SLMapPan

class SLMapPan(*init=0.0*)

SLMap with normalized values for a 'pan' slider.

Parent *SLMap*

Args

init: int or float, optional Initial value. Specified in the real range, not between 0 and 1. Defaults to 0.

Note: SLMapPhase values are:

- min = 0.0
- max = 1.0
- scale = 'lin'
- name = 'pan'
- res = 'float'
- ramp = 0.025

Additional WxPython widgets

The classes in this module are based on internal classes that where originally designed to help the creation of graphical tools for the control and the visualization of audio signals. WxPython must be installed under the current Python distribution to access these classes.

PyoGuiControlSlider

class PyoGuiControlSlider(*parent*, *minvalue*, *maxvalue*, *init=None*, *pos=(0, 0)*, *size=(200, 16)*, *log=False*, *integer=False*, *powoftwo=False*, *orient=4*)

Floating-point control slider.

Parent wx.Panel

Events

EVT_PYO_GUI_CONTROL_SLIDER Sent after any change of the slider position. The current value of the slider can be retrieve with the *value* attribute of the generated event. The object itself can be retrieve with the *object* attribute of the event and the object's id with the *id* attrbute.

Args

parent: wx.Window The parent window.

minvalue: float The minimum value of the slider.

maxvalue: float The maximum value of the slider.

init: float, optional The initial value of the slider. If None, the slider inits to the minimum value. Defaults to None.

pos: tuple, optional The slider's position in pixel (x, y). Defaults to (0, 0).

size: tuple, optional The slider's size in pixel (x, y). Defaults to (200, 16).

log: boolean, optional If True, creates a logarithmic slider (minvalue must be greater than 0). Defaults to False.

integer: boolean, optional If True, creates an integer slider. Defaults to False.

powoftwo: boolean, optional If True, creates a power-of-two slider (log is automatically False and integer is True). If True, minvalue and maxvalue must be exponents to base 2 but init is a real power-of-two value. Defaults to False.

orient: {wx.HORIZONTAL or wx.VERTICAL}, optional The slider's orientation. Defaults to wx.HORIZONTAL.

enable()
Enable the slider for user input.

disable()
Disable the slider for user input.

setValue(*x*, *propagate=True*)
Sets a new value to the slider.

Args

x: int or float The controller number.

propagate: boolean, optional If True, an event will be sent after the call.

setMidiCtl(*x*, *propagate=True*)
Sets the midi controller number to show on the slider.

Args

x: int The controller number.

propagate: boolean, optional If True, an event will be sent after the call.

setRange(*minvalue*, *maxvalue*)
Sets new minimum and maximum values.

Args

minvalue: int or float The new minimum value.

maxvalue: int or float The new maximum value.

getValue()
Returns the current value of the slider.

getMidiCtl()
Returns the midi controller number, if any, assigned to the slider.

getMinValue()
Returns the current minimum value.

getMaxValue()
Returns the current maximum value.

getInit()
Returns the initial value.

getRange()
Returns minimum and maximum values as a list.

isInteger()
Returns True if the slider manage only integer, False otherwise.

isLog()
Returns True if the slider is logarithmic, False otherwise.

isPowOfTwo()
Returns True if the slider manage only power-of-two values, False otherwise.

PyoGuiVuMeter

class PyoGuiVuMeter(*parent*, *nchnls=2*, *pos=(0, 0)*, *size=(200, 11)*, *orient=4*, *style=0*)
Multi-channels Vu Meter.

When registered as the Server's meter, its internal method *setRms* will be called each buffer size with a list of normalized amplitudes as argument. The *setRms* method can also be registered as the function callback of a PeakAmp object.

Parent wx.Panel

Args

parent: wx.Window The parent window.

nchnls: int, optional The initial number of channels of the meter. Defaults to 2.

pos: wx.Point, optional Window position in pixels. Defaults to (0, 0).

size: tuple, optional The meter's size in pixels (x, y). Defaults to (200, 11).

orient: {wx.HORIZONTAL or wx.VERTICAL}, optional The meter's orientation. Defaults to wx.HORIZONTAL.

style: int, optional Window style (see wx.Window documentation). Defaults to 0.

setNchnls(*nchnls*)
Sets the number of channels of the meter.

Args

nchnls: int The number of channels.

PyoGuiGrapher

class PyoGuiGrapher(*parent, xlen=8192, yrange=(0, 1), init=[(0.0, 0.0), (1.0, 1.0)], mode=0, exp=10, inverse=True, tension=0, bias=0, pos=(0, 0), size=(300, 200), style=0*)
Multi-modes break-points function editor.

Parent wx.Panel

Events

EVT_PYO_GUI_GRAPHER Sent after any change of the grapher function. The current list of points of the grapher can be retrieve with the *value* attribute of the generated event. The object itself can be retrieve with the *object* attribute of the event and the object's id with the *id* attrbute.

Args

parent: wx.Window The parent window.

xlen: int, optional The length, in samples, of the grapher. Defaults to 8192.

yrange: two-values tuple, optional A tuple indicating the minimum and maximum values of the Y-axis. Defaults to (0, 1).

init: list of two-values tuples, optional The initial break-points function set as normalized values. A point is defined with its X and Y positions as a tuple. Defaults to [(0.0, 0.0), (1.0, 1.0)].

mode: int, optional The grapher mode definning how line segments will be draw. Possible modes are:

0. linear (default)
1. cosine
2. exponential (uses *exp* and *inverse* arguments)
3. curve (uses *tension* and *bias* arguments)
4. logarithmic
5. logarithmic cosine

exp: int or float, optional The exponent factor for an exponential graph. Defaults to 10.0.

inverse: boolean, optional If True, downward slope will be inversed. Useful to create biexponential curves. Defaults to True.

tension: int or float, optional Curvature at the known points. 1 is high, 0 normal, -1 is low. Defaults to 0.

bias: int or float, optional Curve attraction (for each segments) toward bundary points. 0 is even, positive is towards first point, negative is towards the second point. Defaults to 0.

pos: wx.Point, optional Window position in pixels. Defaults to (0, 0).

size: wx.Size, optional Window size in pixels. Defaults to (300, 200).

style: int, optional Window style (see wx.Window documentation). Defaults to 0.

reset()
Resets the points to the initial state.

getPoints()
Returns the current normalized points of the grapher.

getValues()
Returns the current points, according to Y-axis range, of the grapher.

setPoints(*pts*)
Sets a new group of normalized points in the grapher.

Args

pts: list of two-values tuples New normalized (between 0 and 1) points.

setValues(*vals*)
Sets a new group of points, according to Y-axis range, in the grapher.

Args

vals: list of two-values tuples New real points.

setYrange(*yrange*)
Sets a new Y-axis range to the grapher.

Args

yrange: two-values tuple New Y-axis range.

setInitPoints(*pts*)
Sets a new initial normalized points list to the grapher.

Args

pts: list of two-values tuples New normalized (between 0 and 1) initial points.

setMode(*x*)
Changes the grapher's mode.

Args

x: int

New mode. Possible modes are:

0. linear (default)
1. cosine
2. exponential (uses *exp* and *inverse* arguments)
3. curve (uses *tension* and *bias* arguments)
4. logarithmic
5. logarithmic cosine

setExp(*x*)
Changes the grapher's exponent factor for exponential graph.

Args

x: float New exponent factor.

setInverse(*x*)
Changes the grapher's inverse boolean for exponential graph.

Args

x: boolean New inverse factor.

setTension(*x*)
Changes the grapher's tension factor for curved graph.

Args

x: float New tension factor.

setBias(*x*)
Changes the grapher's bias factor for curved graph.

Args

x: float New bias factor.

PyoGuiMultiSlider

class PyoGuiMultiSlider(*parent*, *xlen=16*, *yrange=(0, 1)*, *init=None*, *pos=(0, 0)*, *size=(300, 200)*, *style=0*)

Data multi-sliders editor.

Parent wx.Panel

Events

EVT_PYO_GUI_MULTI_SLIDER Sent after any change of the multi-sliders values. The current list of values of the multi-sliders can be retrieve with the *value* attribute of the generated event. The object itself can be retrieve with the *object* attribute of the event and the object's id with the *id* attrbute.

Args

parent: wx.Window The parent window.

xlen: int, optional The number of sliders in the multi-sliders. Defaults to 16.

yrange: two-values tuple A tuple indicating the minimum and maximum values of the Y-axis. Defaults to (0, 1).

init: list values, optional The initial list of values of the multi-sliders. Defaults to None, meaning all sliders initialized to the minimum value.

pos: wx.Point, optional Window position in pixels. Defaults to (0, 0).

size: wx.Size, optional Window size in pixels. Defaults to (300, 200).

style: int, optional Window style (see wx.Window documentation). Defaults to 0.

reset()

Resets the sliders to their initial state.

getValues()

Returns the current values of the sliders.

setValues(*vals*)

Sets new values to the sliders.

Args

vals: list of values New values.

setYrange(*yrange*)

Sets a new Y-axis range to the multi-sliders.

Args

yrange: two-values tuple New Y-axis range.

PyoGuiSpectrum

class PyoGuiSpectrum(*parent*, *lowfreq=0*, *highfreq=22050*, *fscaling=0*, *mscaling=0*, *pos=(0, 0)*, *size=(300, 200)*, *style=0*)

Frequency spectrum display.

This widget should be used with the Spectrum object, which measures the magnitude of an input signal versus frequency within a user defined range. It can show both magnitude and frequency on linear or logarithmic scale.

To create the bridge between the analyzer and the display, the Spectrum object must be registered in the PyoGuiSpectrum object with the setAnalyzer(obj) method. The Spectrum object will automatically call the update(points) method to refresh the display.

Parent wx.Panel

Args

parent: wx.Window The parent window.

lowfreq: int or float, optional The lowest frequency, in Hz, to display on the X-axis. Defaults to 0.

highfreq: int or float, optional The highest frequency, in Hz, to display on the X-axis. Defaults to 22050.

fscaling: int, optional The frequency scaling on the X-axis. 0 means linear, 1 means logarithmic. Defaults to 0.

mscaling: int, optional The magnitude scaling on the Y-axis. 0 means linear, 1 means logarithmic. Defaults to 0.

pos: wx.Point, optional Window position in pixels. Defaults to (0, 0).

size: wx.Size, optional Window size in pixels. Defaults to (300, 200).

style: int, optional Window style (see wx.Window documentation). Defaults to 0.

update(*points*)

Display updating method.

This method is automatically called by the audio analyzer object (Spectrum) with points to draw as arguments. The points are already formatted for the current drawing surface to save CPU cycles.

The method setAnalyzer(obj) must be used to register the audio analyzer object.

Args

points: list of list of tuples A list containing n-channels list of tuples. A tuple is a point (X-Y coordinates) to draw.

setAnalyzer(*object*)

Register an audio analyzer object (Spectrum).

Args

object: Spectrum object The audio object performing the frequency analysis.

setLowFreq(*x*)

Changes the lowest frequency of the display.

This method propagates the value to the audio analyzer.

Args

x: int or float New lowest frequency.

setHighFreq(*x*)

Changes the highest frequency of the display.

This method propagates the value to the audio analyzer.

Args

x: int or float New highest frequency.

setFscaling(*x*)
Changes the frequency scaling (X-axis) of the display.

This method propagates the value to the audio analyzer.

Args

x: int 0 means linear scaling, 1 means logarithmic scaling.

setMscaling(*x*)
Changes the magnitude scaling (Y-axis) of the display.

This method propagates the value to the audio analyzer.

Args

x: int 0 means linear scaling, 1 means logarithmic scaling.

PyoGuiScope

class PyoGuiScope(*parent*, *length=0.05*, *gain=0.67*, *pos=(0, 0)*, *size=(300, 200)*, *style=0*)
Oscilloscope display.

This widget should be used with the Scope object, which computes the waveform of an input signal to display on a GUI.

To create the bridge between the analyzer and the display, the Scope object must be registered in the PyoGuiScope object with the setAnalyzer(obj) method. The Scope object will automatically call the update(points) method to refresh the display.

Parent wx.Panel

Args

parent: wx.Window The parent window.

length: float, optional Length, in seconds, of the waveform segment displayed on the window. Defaults to 0.05.

gain: float, optional Linear gain applied to the signal to be displayed. Defaults to 0.67.

pos: wx.Point, optional Window position in pixels. Defaults to (0, 0).

size: wx.Size, optional Window size in pixels. Defaults to (300, 200).

style: int, optional Window style (see wx.Window documentation). Defaults to 0.

update(*points*)
Display updating method.

This method is automatically called by the audio analyzer object (Scope) with points to draw as arguments. The points are already formatted for the current drawing surface to save CPU cycles.

The method setAnalyzer(obj) must be used to register the audio analyzer object.

Args

points: list of list of tuples A list containing n-channels list of tuples. A tuple is a point (X-Y coordinates) to draw.

setAnalyzer(*object*)
Register an audio analyzer object (Scope).

Args

object: Scope object The audio object performing the waveform analysis.

setLength(*x*)
Changes the length, in seconds, of the displayed audio segment.

This method propagates the value to the audio analyzer.

Args

x: float New segment length in seconds.

setGain(*x*)
Changes the gain applied to the input signal.

This method propagates the value to the audio analyzer.

Args

x: float New linear gain.

PyoGuiSndView

class PyoGuiSndView(*parent*, *pos=(0, 0)*, *size=(300, 200)*, *style=0*)
Soundfile display.

This widget should be used with the SndTable object, which keeps soundfile in memory and computes the waveform to display on the GUI.

To create the bridge between the audio memory and the display, the SndTable object must be registered in the PyoGuiSndView object with the setTable(object) method.

The SndTable object will automatically call the update() method to refresh the display when the table is modified.

Parent wx.Panel

Events

EVT_PYO_GUI_SNDVIEW_MOUSE_POSITION Sent when the mouse is moving on the panel with the left button pressed. The *value* attribute of the event will hold the normalized position of the mouse into the sound. For X-axis value, 0.0 is the beginning of the sound and 1.0 is the end of the sound. For the Y-axis, 0.0 is the bottom of the panel and 1.0 is the top. The object itself can be retrieve with the *object* attribute of the event and the object's id with the *id* attrbute.

EVT_PYO_GUI_SNDVIEW_SELECTION Sent when a new region is selected on the panel. A new selection is created with a Right-click and drag on the panel. The current selection can be moved with Shift+Right-click and drag. Ctrl+Right-click (Cmd on OSX) remove the selected region. The *value* attribute of the event will hold the normalized selection as a tuple (min, max). 0.0 means the beginning of the sound and 1.0 means the end of the sound. The object itself can be retrieve with the *object* attribute of the event and the object's id with the *id* attrbute.

Args

parent: wx.Window The parent window.

pos: wx.Point, optional Window position in pixels. Defaults to (0, 0).

size: wx.Size, optional Window size in pixels. Defaults to (300, 200).

style: int, optional Window style (see wx.Window documentation). Defaults to 0.

update()
Display updating method.

This method is automatically called by the audio memory object (SndTable) when the table is modified.

The method setTable(obj) must be used to register the audio memory object.

setTable(*object*)
Register an audio memory object (SndTable).

Args

object: SndTable object The audio table keeping the sound in memory.

setSelection(*start*, *stop*)
Changes the selected region.

This method will trigger a EVT_PYO_GUI_SNDVIEW_SELECTION event with a tuple (start, stop) as value.

Args

start: float The starting point of the selected region. This value must be normalized between 0 and 1 (0 is the beginning of the sound, 1 is the end).

stop: float The ending point of the selected region. This value must be normalized between 0 and 1 (0 is the beginning of the sound, 1 is the end).

resetSelection()
Removes the selected region.

This method will trigger a EVT_PYO_GUI_SNDVIEW_SELECTION event with a tuple (0.0, 1.0) as value.

PyoGuiKeyboard

class PyoGuiKeyboard(*parent*, *poly=64*, *pos=(0, 0)*, *size=(600, 100)*, *style=0*)
Virtual MIDI keyboard.

Parent wx.Panel

Events

EVT_PYO_GUI_KEYBOARD Sent whenever a note change on the keyboard. The *value* attribute of the event will hold a (pitch, velocity) tuple. The object itself can be retrieve with the *object* attribute of the event and the object's id with the *id* attrbute.

Args

parent: wx.Window The parent window.

poly: int, optional Maximum number of notes that can be held at the same time. Defaults to 64.

pos: wx.Point, optional Window position in pixels. Defaults to (0, 0).

size: wx.Size, optional Window size in pixels. Defaults to (300, 200).

style: int, optional Window style (see wx.Window documentation). Defaults to 0.

getCurrentNotes()
Returns a list of the current notes.

reset()
Resets the keyboard state.

setPoly(*poly*)
Sets the maximum number of notes that can be held at the same time.

Args

poly: int New maximum number of notes held at once.

CHAPTER

TWO

EXAMPLES

2.1 01-intro

2.1.1 01-audio-server.py - Booting the audio server.

A Server object needs to be created before any other audio object. It is the one that handles the communication with the audio and midi drivers and also the one that keeps track of the processing chain.

```
from pyo import *

# Creates a Server object with default arguments.
# See the manual about how to change the sampling rate, the buffer
# size, the number of channels or one of the other global settings.
s = Server()

# Boots the server. This step initializes audio and midi streams.
# Audio and midi configurations (if any) must be done before that call.
s.boot()

# Starts the server. This step activates the server processing loop.
s.start()

# Here comes the processing chain...

# The Server object provides a Graphical User Interface with the
# gui() method. One of its purpose is to keep the program alive
# while computing samples over time. If the locals dictionary is
# given as argument, the user can continue to send commands to the
# python interpreter from the GUI.
s.gui(locals())
```

2.1.2 02-sine-tone.py - The “hello world” of audio programming!

This script simply plays a 1000 Hz sine tone.

```
from pyo import *

# Creates and boots the server.
# The user should send the "start" command from the GUI.
s = Server().boot()
# Drops the gain by 20 dB.
```

(continues on next page)

(continued from previous page)

```
s.amp = 0.1

# Creates a sine wave player.
# The out() method starts the processing
# and sends the signal to the output.
a = Sine().out()

# Opens the server graphical interface.
s.gui(locals())
```

2.1.3 03-parallel-proc.py - Multiple processes on a single source.

This example shows how to play different audio objects side-by-side. Every processing object (ie the ones that modify an audio source) have a first argument called "input". This argument takes the audio object to process.

Note the input variable given to each processing object and the call to the out() method of each object that should send its samples to the output.

```
from pyo import *

s = Server().boot()
s.amp = 0.1

# Creates a sine wave as the source to process.
a = Sine()

# Passes the sine wave through an harmonizer.
hr = Harmonizer(a).out()

# Also through a chorus.
ch = Chorus(a).out()

# And through a frequency shifter.
sh = FreqShift(a).out()

s.gui(locals())
```

2.1.4 04-serial-proc.py - Chaining processes on a single source.

This example shows how to chain processes on a single source. Every processing object (ie the ones that modify an audio source) have a first argument called "input". This argument takes the audio object to process.

Note the input variable given to each Harmonizer.

```
from pyo import *

s = Server().boot()
s.amp = 0.1

# Creates a sine wave as the source to process.
a = Sine().out()

# Passes the sine wave through an harmonizer.
h1 = Harmonizer(a).out()
```

(continues on next page)

(continued from previous page)

```
# Then the harmonized sound through another harmonizer.
h2 = Harmonizer(h1).out()

# And again...
h3 = Harmonizer(h2).out()

# And again...
h4 = Harmonizer(h3).out()

s.gui(locals())
```

2.1.5 05-output-channels.py - Sending signals to different physical outputs.

The simplest way to choose the output channel where to send the sound is to give it as the first argument of the out() method. In fact, the signature of the out() method reads as:

.out(chnl=0, inc=1, dur=0, delay=0)

chnl is the output where to send the first audio channel (stream) of the object. *inc* is the output increment for other audio channels. *dur* is the living duration, in seconds, of the process and *delay* is a delay, in seconds, before activating the process. A duration of 0 means play forever.

```
from pyo import *

s = Server().boot()
s.amp = 0.1

# Creates a source (white noise)
n = Noise()

# Sends the bass frequencies (below 1000 Hz) to the left
lp = ButLP(n).out()

# Sends the high frequencies (above 1000 Hz) to the right
hp = ButHP(n).out(1)

s.gui(locals())
```

2.2 02-controls

2.2.1 01-fixed-control.py - Number as argument.

Audio objects behaviour can be controlled by passing value to their arguments at initialization time.

```
from pyo import *

s = Server().boot()
s.amp = 0.1

# Sets fundamental frequency
freq = 200
```

(continues on next page)

(continued from previous page)

```
# Approximates a triangle waveform by adding odd harmonics with
# amplitude proportional to the inverse square of the harmonic number.
h1 = Sine(freq=freq, mul=1).out()
h2 = Sine(freq=freq*3, phase=0.5, mul=1./pow(3,2)).out()
h3 = Sine(freq=freq*5, mul=1./pow(5,2)).out()
h4 = Sine(freq=freq*7, phase=0.5, mul=1./pow(7,2)).out()
h5 = Sine(freq=freq*9, mul=1./pow(9,2)).out()
h6 = Sine(freq=freq*11, phase=0.5, mul=1./pow(11,2)).out()

# Displays the final waveform
sp = Scope(h1+h2+h3+h4+h5+h6)

s.gui(locals())
```

2.2.2 02-dynamic-control.py - Graphical control for parameters.

With pyo, it's easy to quickly try some parameter combination with the controller window already configured for each object. To open the controller window, just call the ctrl() method on the object you want to control.

```
from pyo import *

s = Server().boot()
s.amp = 0.1

# Creates two objects with cool parameters, one per channel.
a = FM().out()
b = FM().out(1)

# Opens the controller windows.
a.ctrl(title="Frequency modulation left channel")
b.ctrl(title="Frequency modulation right channel")

s.gui(locals())
```

2.2.3 03-output-range.py - The *mul* and *add* attributes.

Almost all audio objects have a *mul* and *add* attributes. These are defined inside the PyoObject, which is the base class for all objects generating audio signal. The manual page of the PyoObject explains all behaviours common to audio objects.

An audio signal outputs samples as floating-point numbers in the range -1 to 1. The *mul* and *add* attributes can be used to change the output range. Common uses are for modulating the amplitude of a sound or for building control signals like low frequency oscillators.

A shortcut to automatically manipulate both *mul* and *add* attributes is to call the range(min, max) method of the PyoObject. This method sets *mul* and *add* attributes according to the desired *min* and *max* output values. It assumes that the generated signal is in the range -1 to 1.

```
from pyo import *

s = Server().boot().start()
```

(continues on next page)

(continued from previous page)

```
# The `mul` attribute multiplies each sample by its value.
a = Sine(freq=100, mul=0.1)

# The `add` attribute adds an offset to each sample.
# The multiplication is applied before the addition.
b = Sine(freq=100, mul=0.5, add=0.5)

# Using the range(min, max) method allows to automatically
# compute both `mul` and `add` attributes.
c = Sine(freq=100).range(-0.25, 0.5)

# Displays the waveforms
sc = Scope([a, b, c])

s.gui(locals())
```

2.2.4 04-building-lfo.py - Audio control of parameters.

One of the most important thing with computer music is the trajectories taken by parameters over time. This is what gives life to the synthesized sound.

One way to create moving values is by connecting a low frequency oscillator to an object's attribute. This script shows that process.

Other possibilities that will be covered later use random class objects or feature extraction from an audio signal.

```
from pyo import *

s = Server().boot()

# Creates a noise source
n = Noise()

# Creates an LFO oscillating +/- 500 around 1000 (filter's frequency)
lfo1 = Sine(freq=.1, mul=500, add=1000)
# Creates an LFO oscillating between 2 and 8 (filter's Q)
lfo2 = Sine(freq=.4).range(2, 8)
# Creates a dynamic bandpass filter applied to the noise source
bp1 = ButBP(n, freq=lfo1, q=lfo2).out()

# The LFO object provides more waveforms than just a sine wave

# Creates a ramp oscillating +/- 1000 around 12000 (filter's frequency)
lfo3 = LFO(freq=.25, type=1, mul=1000, add=1200)
# Creates a square oscillating between 4 and 12 (filter's Q)
lfo4 = LFO(freq=4, type=2).range(4, 12)
# Creates a second dynamic bandpass filter applied to the noise source
bp2 = ButBP(n, freq=lfo3, q=lfo4).out(1)

s.gui(locals())
```

2.2.5 05-math-ops.py - Audio objects and arithmetic expresssions.

This script shows how a PyoObject reacts when used inside an arithmetic expression.

Multiplication, addition, division and substraction can be applied between pyo objects or between pyo objects and numbers. Doing so returns a Dummy object that outputs the result of the operation.

A Dummy object is only a place holder to keep track of arithmetic operations on audio objects.

PyoObject can also be used in expression with the exponent (**), modulo (%) and unary negative (-) operators.

```
from __future__ import print_function
from pyo import *

s = Server().boot()
s.amp = 0.1

# Full scale sine wave
a = Sine()

# Creates a Dummy object `b` with `mul` attribute
# set to 0.5 and leaves `a` unchanged.
b = a * 0.5
b.out()

# Instance of Dummy class
print(b)

# Computes a ring modulation between two PyoObjects
# and scales the amplitude of the resulting signal.
c = Sine(300)
d = a * c * 0.3
d.out()

# PyoObject can be used with Exponent operator.
e = c ** 10 * 0.4
e.out(1)

# Displays the ringmod and the rectified signals.
sp = Spectrum([d, e])
sc = Scope([d, e])

s.gui(locals())
```

2.2.6 06-multichannel-expansion.py - What is a *Stream*? Polyphonic objects.

List expansion is a powerful technique for generating many audio streams at once.

What is a "stream"? A "stream" is a monophonic channel of samples. It is the basic structure over which all the library is built. Any PyoObject can handle as many streams as necessary to represent the defined process. When a polyphonic (ie more than one stream) object is asked to send its signals to the output, the server will use the arguments (*chnl* and *inc*) of the out() method to distribute the streams over the available output channels.

Almost all attributes of all objects of the library accept list of values instead of a single value. The object will create internally the same number of streams than the length of the largest list given to an attribute at the initialization time. Each value of the list is used to generate one stream. Shorter lists will wrap around when reaching the end of the list.

A PyoObject is considered by other object as a list. The function *len(obj)* returns the number of streams managed by the object. This feature is useful to create a polyphonic dsp chain.

```
from pyo import *

s = Server().boot()

### Using multichannel-expansion to create a square wave ###

# Sets fundamental frequency.
freq = 100
# Sets the highest harmonic.
high = 20

# Generates the list of harmonic frequencies (odd only).
harms = [freq * i for i in range(1, high) if i%2 == 1]
# Generates the list of harmonic amplitudes (1 / n).
amps = [0.33 / i for i in range(1, high) if i%2 == 1]

# Creates all sine waves at once.
a = Sine(freq=harms, mul=amps)
# Prints the number of streams managed by "a".
print(len(a))

# The mix(voices) method (defined in PyoObject) mixes
# the object streams into `voices` streams.
b = a.mix(voices=1).out()

# Displays the waveform.
sc = Scope(b)

s.gui(locals())
```

2.2.7 07-multichannel-expansion-2.py - Extended multichannel expansion.

When using multichannel expansion with lists of different lengths, the longer list is used to set the number of streams and smaller lists will be wrap-around to fill the holes.

This feature is very useful to create complex sonorities.

```
from pyo import *

s = Server().boot()

# 12 streams with different combinations of `freq` and `ratio`.
a = SumOsc(freq=[100, 150.2, 200.5, 250.7],
           ratio=[0.501, 0.753, 1.255],
           index=[.3, .4, .5, .6, .7, .4, .5, .3, .6, .7, .3, .5],
           mul=.05)

# Adds a stereo reverberation to the signal
rev = Freeverb(a.mix(2), size=0.80, damp=0.70, bal=0.30).out()

s.gui(locals())
```

2.2.8 08-handling-channels.py - Managing object's internal audio streams.

Because audio objects expand their number of streams according to lists given to their arguments and the fact that an audio object is considered as a list, if a multi-streams object is given as an argument to another audio object, the later will also be expanded in order to process all given streams. This is really powerful to create polyphonic processes without copying long chunks of code but it can be very CPU expensive.

In this example, we create a square from a sum of sine waves. After that, a chorus is applied to the resulting waveform. If we don't mix down the square wave, we got tens of Chorus objects in the processing chain (one per sine). This can easily overrun the CPU. The exact same result can be obtained with only one Chorus applied to the sum of the sine waves. The *mix(voices)* method of the PyoObject helps the handling of channels in order to save CPU cycles. Here, we down mix all streams to only two streams (to maintain the stereo) before processing the Chorus arguments.

```
from pyo import *

s = Server().boot()

# Sets fundamental frequency and highest harmonic.
freq = 100
high = 20

# Generates lists for frequencies and amplitudes
harms = [freq * i for i in range(1, high) if i%2 == 1]
amps = [0.33 / i for i in range(1, high) if i%2 == 1]

# Creates a square wave by additive synthesis.
a = Sine(freq=harms, mul=amps)
print("Number of Sine streams: %d" % len(a))

# Mix down the number of streams of "a" before computing the Chorus.
b = Chorus(a.mix(2), feedback=0.5).out()
print("Number of Chorus streams: %d" % len(b))

s.gui(locals())
```

2.2.9 09-handling-channels-2.py - The *out* method and the physical outputs.

In a multichannel environment, we can carefully choose which stream goes to which output channel. To achieve this, we use the *chnl* and *inc* arguments of the out method.

chnl : Physical output assigned to the first audio stream of the object. inc : Output channel increment value.

```
from pyo import *

# Creates a Server with 8 channels
s = Server(nchnls=8).boot()

# Generates a sine wave
a = Sine(freq=500, mul=0.3)

# Mixes it up to four streams
b = a.mix(4)

# Outputs to channels 0, 2, 4 and 6
b.out(chnl=0, inc=2)

s.gui(locals())
```

2.2.10 10-handling-channels-3.py - Random multichannel outputs.

If *chnl* is negative, streams begin at 0, increment the output number by inc and wrap around the global number of channels. Then, the list of streams is scrambled.

```
from pyo import *

# Creates a Server with 8 channels
s = Server(nchnls=8).boot()

amps = [.05,.1,.15,.2,.25,.3,.35,.4]

# Generates 8 sine waves with
# increasing amplitudes
a = Sine(freq=500, mul=amps)

# Shuffles physical output channels
a.out(chnl=-1)

s.gui(locals())
```

2.2.11 11-handling-channels-4.py - Explicit control of the physical outputs.

If *chnl* is a list, successive values in the list will be assigned to successive streams.

```
from pyo import *

# Creates a Server with 8 channels
s = Server(nchnls=8).boot()

amps = [.05,.1,.15,.2,.25,.3,.35,.4]

# Generates 8 sine waves with
# increasing amplitudes
a = Sine(freq=500, mul=amps)

# Sets the output channels ordering
a.out(chnl=[3,4,2,5,1,6,0,7])

s.gui(locals())
```

2.3 03-generators

2.3.1 01-complex-oscs.py - Complex spectrum oscillators.

This tutorial presents four objects of the library which are useful to generate complex spectrums by means of synthesis.

Blit: Impulse train generator with control over the number of harmonics.

RCOsc: Aproximation of a RC circuit (a capacitor and a resistor in series).

SineLoop: Sine wave oscillator with feedback.

SuperSaw: Roland JP-8000 Supersaw emulator.

Use the "voice" slider of the window "Input interpolator" to interpolate between the four waveforms. Each one have an LFO applied to the argument that change the tone of the sound.

```
from pyo import *

s = Server().boot()

# Sets fundamental frequency.
freq = 187.5

# Impulse train generator.
lfo1 = Sine(.1).range(1, 50)
osc1 = Blit(freq=freq, harms=lfo1, mul=0.3)

# RC circuit.
lfo2 = Sine(.1, mul=0.5, add=0.5)
osc2 = RCOsc(freq=freq, sharp=lfo2, mul=0.3)

# Sine wave oscillator with feedback.
lfo3 = Sine(.1).range(0, .18)
osc3 = SineLoop(freq=freq, feedback=lfo3, mul=0.3)

# Roland JP-8000 Supersaw emulator.
lfo4 = Sine(.1).range(0.1, 0.75)
osc4 = SuperSaw(freq=freq, detune=lfo4, mul=0.3)

# Interpolates between input objects to produce a single output
sel = Selector([osc1, osc2, osc3, osc4]).out()
sel.ctrl(title="Input interpolator (0=Blit, 1=RCOsc, 2=SineLoop, 3=SuperSaw)")

# Displays the waveform of the chosen source
sc = Scope(sel)

# Displays the spectrum contents of the chosen source
sp = Spectrum(sel)

s.gui(locals())
```

2.3.2 02-band-limited-oscs.py - Oscillators whose spectrum is kept under the Nyquist frequency.

This tutorial presents an object (misnamed LFO but it's too late to change its name!) that implements various band-limited waveforms. A band-limited signal is a signal that none of its partials exceeds the nyquist frequency (sr/2).

The LFO object, despite its name, can be use as a standard oscillator, with very high fundamental frequencies. At lower frequencies (below 20 Hz) this object will give a true LFO with various shapes.

The "type" slider in the controller window lets choose between these particular waveforms:

0. Saw up (default)
1. Saw down
2. Square
3. Triangle
4. Pulse

5. Bipolar pulse
6. Sample and hold
7. Modulated Sine

```
from pyo import *

s = Server().boot()

# Sets fundamental frequency.
freq = 187.5

# LFO applied to the `sharp` attribute
lfo = Sine(.2, mul=0.5, add=0.5)

# Various band-limited waveforms
osc = LFO(freq=freq, sharp=lfo, mul=0.4).out()
osc.ctrl()

# Displays the waveform
sc = Scope(osc)

# Displays the spectrum contents
sp = Spectrum(osc)

s.gui(locals())
```

2.3.3 03-fm-generators.py - Frequency modulation synthesis.

There two objects in the library that implement frequency modulation algorithms. These objects are very simple, although powerful. It is also relatively simple to build a custom FM algorithm, this will be covered in the tutorials on custom synthesis algorithms.

Use the "voice" slider of the window "Input interpolator" to interpolate between the two sources. Use the controller windows to change the parameters of the FM algorithms.

Note what happened in the controller window when we give a list of floats to an object's argument.

```
from pyo import *

s = Server().boot()

# FM implements the basic Chowning algorithm
fm1 = FM(carrier=250, ratio=[1.5,1.49], index=10, mul=0.3)
fm1.ctrl()

# CrossFM implements a frequency modulation synthesis where the
# output of both oscillators modulates the frequency of the other one.
fm2 = CrossFM(carrier=250, ratio=[1.5,1.49], ind1=10, ind2=2, mul=0.3)
fm2.ctrl()

# Interpolates between input objects to produce a single output
sel = Selector([fm1, fm2]).out()
sel.ctrl(title="Input interpolator (0=FM, 1=CrossFM)")

# Displays the spectrum contents of the chosen source
```

(continues on next page)

(continued from previous page)

```
sp = Spectrum(sel)

s.gui(locals())
```

2.3.4 04-noise-generators.py - Different pseudo-random noise generators.

There are three noise generators (beside random generators that will be covered later) in the library. These are the classic white noise, pink noise and brown noise.

Noise: White noise generator, flat spectrum.

PinkNoise: Pink noise generator, 3dB rolloff per octave.

BrownNoise: Brown noise generator, 6dB rolloff per octave.

Use the "voice" slider of the window "Input interpolator" to interpolate between the three sources.

```
from pyo import *

s = Server().boot()

# White noise
n1 = Noise(0.3)

# Pink noise
n2 = PinkNoise(0.3)

# Brown noise
n3 = BrownNoise(0.3)

# Interpolates between input objects to produce a single output
sel = Selector([n1, n2, n3]).out()
sel.ctrl(title="Input interpolator (0=White, 1=Pink, 2=Brown)")

# Displays the spectrum contents of the chosen source
sp = Spectrum(sel)

s.gui(locals())
```

2.3.5 05-strange-attractors.py - Non-linear ordinary differential equations.

Oscilloscope part of the tutorial

A strange attractor is a system of three non-linear ordinary differential equations. These differential equations define a continuous-time dynamical system that exhibits chaotic dynamics associated with the fractal properties of the attractor.

There is three strange attractors in the library, the Rossler, the Lorenz and the ChenLee objects. Each one can output stereo signal if the *stereo* argument is set to True.

Use the "voice" slider of the window "Input interpolator" to interpolate between the three sources.

Audio part of the tutorial

It's possible to create very interesting LFO with strange attractors. The last part of this tutorial shows the use of Lorenz's output to drive the frequency of two sine wave oscillators.

```
from pyo import *

s = Server().boot()

### Oscilloscope ###

# LFO applied to the `chaos` attribute
lfo = Sine(0.2).range(0, 1)

# Rossler attractor
n1 = Rossler(pitch=0.5, chaos=lfo, stereo=True)

# Lorenz attractor
n2 = Lorenz(pitch=0.5, chaos=lfo, stereo=True)

# ChenLee attractor
n3 = ChenLee(pitch=0.5, chaos=lfo, stereo=True)

# Interpolates between input objects to produce a single output
sel = Selector([n1, n2, n3])
sel.ctrl(title="Input interpolator (0=Rossler, 1=Lorenz, 2=ChenLee)")

# Displays the waveform of the chosen attractor
sc = Scope(sel)

### Audio ###

# Lorenz with very low pitch value that acts as a LFO
freq = Lorenz(0.005, chaos=0.7, stereo=True, mul=250, add=500)
a = Sine(freq, mul=0.3).out()

s.gui(locals())
```

2.3.6 06-random-generators.py - Overview of some random generators of pyo.

The pyo's random category contains many objects that can be used for different purposes. This category really deserves to be studied.

In this tutorial, we use three random objects (Choice, Randi, RandInt) to control the pitches, the amplitude and the tone of a simple synth.

We will come back to random generators when we will talk about musical algorithms.

```
from pyo import *

s = Server().boot()

# Two streams of midi pitches chosen randomly in a predefined list.
# The argument `choice` of Choice object can be a list of lists to
# list-expansion.
mid = Choice(choice=[60,62,63,65,67,69,71,72], freq=[2,3])
```

(continues on next page)

(continued from previous page)

```
# Two small jitters applied on frequency streams.
# Randi interpolates between old and new values.
jit = Randi(min=0.993, max=1.007, freq=[4.3,3.5])

# Converts midi pitches to frequencies and applies the jitters.
fr = MToF(mid, mul=jit)

# Chooses a new feedback value, between 0 and 0.15, every 4 seconds.
fd = Randi(min=0, max=0.15, freq=0.25)

# RandInt generates a pseudo-random integer number between 0 and `max`
# values at a frequency specified by `freq` parameter. It holds the
# value until the next generation.
# Generates an new LFO frequency once per second.
sp = RandInt(max=6, freq=1, add=8)
# Creates an LFO oscillating between 0 and 0.4.
amp = Sine(sp, mul=0.2, add=0.2)

# A simple synth...
a = SineLoop(freq=fr, feedback=fd, mul=amp).out()

s.gui(locals())
```

2.4 04-soundfiles

2.4.1 01-read-from-disk.py - Soundfile playback from disk.

SfPlayer and friends read samples from a file on disk with control over playback speed and looping mode.

Player family:

- **SfPlayer** : Reads many soundfile formats from disk.
- **SfMarkerLooper** : AIFF with markers soundfile looper.
- **SfMarkerShuffler** : AIFF with markers soundfile shuffler.

Reading sound file from disk can save a lot of RAM, especially if the soundfile is big, but it is more CPU expensive than loading the sound file in memory in a first pass.

```
from pyo import *

s = Server().boot()

path = SNDS_PATH + "/transparent.aif"

# stereo playback with a slight shift between the two channels.
sf = SfPlayer(path, speed=[1, 0.995], loop=True, mul=0.4).out()

s.gui(locals())
```

2.4.2 02-read-from-disk-2.py - Catching the *end-of-file* signal from the SfPlayer object.

This example demonstrates how to use the *end-of-file* signal of the SfPlayer object to trigger another playback (possibly with another sound, another speed, etc.).

When a SfPlayer reaches the end of the file, it sends a trigger (more on trigger later) that the user can retrieve with the syntax :

variable_name["trig"]

```
from pyo import *
import random

s = Server().boot()

# Sound bank
folder = "../snds/"
sounds = ["alum1.wav", "alum2.wav", "alum3.wav", "alum4.wav"]

# Creates the left and right players
sfL = SfPlayer(folder+sounds[0], speed=1, mul=0.5).out()
sfR = SfPlayer(folder+sounds[0], speed=1, mul=0.5).out(1)

# Function to choose a new sound and a new speed for the left player
def newL():
    sfL.path = folder + sounds[random.randint(0, 3)]
    sfL.speed = random.uniform(0.75, 1.5)
    sfL.out()

# The "end-of-file" signal triggers the function "newL"
tfL = TrigFunc(sfL["trig"], newL)

# Function to choose a new sound and a new speed for the right player
def newR():
    sfR.path = folder + sounds[random.randint(0, 3)]
    sfR.speed = random.uniform(0.75, 1.5)
    sfR.out(1)

# The "end-of-file" signal triggers the function "newR"
tfR = TrigFunc(sfR["trig"], newR)

s.gui(locals())
```

2.4.3 03-read-from-ram.py - Soundfile playback from RAM.

Reading a sound file from the RAM gives the advantage of a very fast access to every loaded samples. This is very useful for a lot of processes, such as granulation, looping, creating envelopes and waveforms and many others.

The simplest way of loading a sound in RAM is to use the SndTable object. This example loads a sound file and reads it in loop. We will see some more evolved processes later...

```
from pyo import *

s = Server().boot()

path = SNDS_PATH + "/transparent.aif"
```

(continues on next page)

(continued from previous page)

```
# Loads the sound file in RAM. Beginning and ending points
# can be controlled with "start" and "stop" arguments.
t = SndTable(path)

# Gets the frequency relative to the table length.
freq = t.getRate()

# Simple stereo looping playback (right channel is 180 degrees out-of-phase).
osc = Osc(table=t, freq=freq, phase=[0, 0.5], mul=0.4).out()

s.gui(locals())
```

2.4.4 04-record-perf.py - Recording the performance on disk.

The Server object allow the recording of the overall playback (that is exactly what your hear). The "Rec Start" button of the Server's window is doing that with default parameters. It'll record a file called "pyo_rec.wav" (16-bit, 44100 Hz) on the user's desktop.

You can control the recording with the Server's method called *recordOptions*, the arguments are:

- **dur** [The duration of the recording, a value of -1 means] record forever (recstop() must be called by the user).
- filename : Indicates the location of the recorded file.
- **fileformat** [The format of the audio file (see documentation] for available formats).
- **sampletype** [The sample type of the audio file (see documentation] for available types).

The recording can be triggered programmatically with the Server's methods *recstart()* and *recstop()*. In order to record multiple files from a unique performance, it is possible to set the filename with an argument to *recstart()*.

```
from pyo import *
import os

s = Server().boot()

# Path of the recorded sound file.
path = os.path.join(os.path.expanduser("~"), "Desktop", "synth.wav")
# Record for 10 seconds a 24-bit wav file.
s.recordOptions(dur=10, filename=path, fileformat=0, sampletype=1)

# Creates an amplitude envelope
amp = Fader(fadein=1, fadeout=1, dur=10, mul=0.3).play()

# A Simple synth
lfo = Sine(freq=[0.15, 0.16]).range(1.25, 1.5)
fm2 = CrossFM(carrier=200, ratio=lfo, ind1=10, ind2=2, mul=amp).out()

# Starts the recording for 10 seconds...
s.recstart()

s.gui(locals())
```

2.4.5 05-record-streams.py - Recording individual audio streams on disk.

The Record object can be used to record specific audio streams from a performance. It can be useful to record a sound in mutiple tracks to make post-processing on individual part easier. This example record the bass, the mid and the higher part in three separated files on the user's desktop.

The *fileformat* and *sampletype* arguments are the same as in the Server's *recordOptions* method.

```
from pyo import *
import os

s = Server().boot()

# Defines sound file paths.
path = os.path.join(os.path.expanduser("~"), "Desktop")
bname = os.path.join(path, "bass.wav")
mname = os.path.join(path, "mid.wav")
hname = os.path.join(path, "high.wav")

# Creates an amplitude envelope
amp = Fader(fadein=1, fadeout=1, dur=10, mul=0.3).play()

# Bass voice
blfo = Sine(freq=[0.15, 0.16]).range(78, 82)
bass = RCOsc(freq=blfo, mul=amp).out()

# Mid voice
mlfo = Sine(freq=[0.18, 0.19]).range(0.24, 0.26)
mid = FM(carrier=1600, ratio=mlfo, index=5, mul=amp*0.3).out()

# High voice
hlfo = Sine(freq=[0.1, 0.11, 0.12, 0.13]).range(7000, 8000)
high = Sine(freq=hlfo, mul=amp*0.1).out()

# Creates the recorders
brec = Record(bass, filename=bname, chnls=2, fileformat=0, sampletype=1)
mrec = Record(mid, filename=mname, chnls=2, fileformat=0, sampletype=1)
hrec = Record(high, filename=hname, chnls=2, fileformat=0, sampletype=1)

# After 10.1 seconds, recorder objects will be automatically deleted.
# This will trigger their stop method and properly close the sound files.
clean = Clean_objects(10.1, brec, mrec, hrec)

# Starts the internal timer of Clean_objects. Use its own thread.
clean.start()

# Starts the Server, in order to be sync with the cleanup process.
s.start()

s.gui(locals())
```

2.4.6 06-record-table.py - Recording live sound in RAM.

By recording a stream of sound in RAM, one can quickly re-use the samples in the current process. A combination NewTable - TableRec is all what one need to record any stream in a table.

The NewTable object has a *feedback* argument, allowing overdub.

The TableRec object starts a new recording (records until the table is full) every time its method *play()* is called.

```
from pyo import *
import os

# Audio inputs must be available.
s = Server(duplex=1).boot()

# Path of the recorded sound file.
path = os.path.join(os.path.expanduser("~"), "Desktop", "synth.wav")

# Creates a two seconds stereo empty table. The "feedback" argument
# is the amount of old data to mix with a new recording (overdub).
t = NewTable(length=2, chnls=2, feedback=0.5)

# Retrieves the stereo input
inp = Input([0,1])

# Table recorder. Call rec.play() to start a recording, it stops
# when the table is full. Call it multiple times to overdub.
rec = TableRec(inp, table=t, fadetime=0.05)

# Reads the content of the table in loop.
osc = Osc(table=t, freq=t.getRate(), mul=0.5).out()

s.gui(locals())
```

2.5 05-envelopes

2.5.1 01-data-signal-conversion.py - Conversion from number to audio stream.

The Stream object is a new type introduced by pyo to represent an audio signal as a vector of floats. It is sometimes useful to be able to convert simple numbers (python's floats or integers) to audio signal or to extract numbers from an audio stream.

The Sig object converts a number to an audio stream.

The PyoObject.get() method extracts a float from an audio stream.

```
from __future__ import print_function
from pyo import *

s = Server().boot()

# A python integer (or float).
anumber = 100

# Conversion from number to an audio stream (vector of floats).
astream = Sig(anumber)

# Use a Print (capital "P") object to print an audio stream.
pp = Print(astream, interval=0.1, message="Audio stream value")

# Use the get() method to extract a float from an audio stream.
print("Float from audio stream : ", astream.get())
```

(continues on next page)

(continued from previous page)

```
s.gui(locals())
```

2.5.2 02-linear-ramp.py - Portamento, glissando, ramping.

The SigTo object allows to create audio glissando between the current value and the target value, within a user-defined time. The target value can be a float or another PyoObject. A new ramp is created everytime the target value changes.

Also:

The VarPort object acts similarly but works only with float and can call a user-defined callback when the ramp reaches the target value.

The PyoObject.set() method is another way create a ramp for any given parameter that accept audio signal but is not already controlled with a PyoObject.

```
from pyo import *

s = Server().boot()

# 2 seconds linear ramp starting at 0.0 and ending at 0.3.
amp = SigTo(value=0.3, time=2.0, init=0.0)

# Pick a new value four times per second.
pick = Choice([200,250,300,350,400], freq=4)

# Print the chosen frequency
pp = Print(pick, method=1, message="Frequency")

# Add a little portamento on an audio target and detune a second frequency.
freq = SigTo(pick, time=0.01, mul=[1, 1.005])
# Play with portamento time.
freq.ctrl([SLMap(0, 0.25, "lin", "time", 0.01, dataOnly=True)])

# Play a simple wave.
sig = RCOsc(freq, sharp=0.7, mul=amp).out()

s.gui(locals())
```

2.5.3 03-exponential-ramp.py - Exponential portamento with rising and falling times.

The Port object is designed to lowpass filter an audio signal with different coefficients for rising and falling signals. A lowpass filter is a good and efficient way of creating an exponential ramp from a signal containing abrupt changes. The rising and falling coefficients are controlled in seconds.

```
from pyo import *

s = Server().boot()

# 2 seconds linear ramp starting at 0.0 and ending at 0.3.
amp = SigTo(value=0.3, time=2.0, init=0.0)

# Pick a new value four times per second.
```

(continues on next page)

(continued from previous page)

```
pick = Choice([200,250,300,350,400], freq=4)

# Print the chosen frequency
pp = Print(pick, method=1, message="Frequency")

# Add an exponential portamento on an audio target and detune a second frequency.
# Sharp attack for rising notes and long release for falling notes.
freq = Port(pick, risetime=0.001, falltime=0.25, mul=[1, 1.005])
# Play with portamento times.
freq.ctrl()

# Play a simple wave.
sig = RCOsc(freq, sharp=0.7, mul=amp).out()

s.gui(locals())
```

2.5.4 04-simple-envelopes.py - ASR and ADSR envelopes.

The Fader object is a simple way to setup an Attack/Sustain/Release envelope. This envelope allows to apply fadein and fadeout on audio streams.

If the *dur* argument of the Fader object is set to 0 (the default), the object waits for a stop() command before activating the release part of the envelope. Otherwise, the sum of *fadein* and *fadeout* must be less than or egal to *dur* and the envelope runs to the end on a play() command.

The Adsr object (Attack/Decay/Sustain/Release) acts exactly like the Fader object, with a more flexible (and so common) kind of envelope.

```
from pyo import *
import random

s = Server().boot()

# Infinite sustain for the global envelope.
globalamp = Fader(fadein=2, fadeout=2, dur=0).play()

# Envelope for discrete events, sharp attack, long release.
env = Adsr(attack=0.01, decay=0.1, sustain=0.5, release=1.5, dur=2, mul=0.5)
# setExp method can be used to create exponential or logarithmic envelope.
env.setExp(0.75)

# Initialize  a simple wave player and apply both envelopes.
sig = SuperSaw(freq=[100,101], detune=0.6, bal=0.8, mul=globalamp*env).out()

def play_note():
    "Play a new note with random frequency and jitterized envelope."
    freq = random.choice(midiToHz([36, 38, 41, 43, 45]))
    sig.freq = [freq, freq*1.005]
    env.attack = random.uniform(0.002, 0.01)
    env.decay = random.uniform(0.1, 0.5)
    env.sustain = random.uniform(0.3, 0.6)
    env.release = random.uniform(0.8, 1.4)
    # Start the envelope for the event.
    env.play()
```

(continues on next page)

(continued from previous page)

```
# Periodically call a function.
pat = Pattern(play_note, time=2).play()

s.gui(locals())
```

2.5.5 05-breakpoints-functions.py - Multi-segments envelopes.

Linseg and Expseg objects draw a series of line segments between specified break-points, either linear or exponential.

These objects wait for play() call to start reading the envelope.

They have methods to set loop mode, call pause/play without reset, and replace the breakpoints.

One can use the graph() method to open a graphical display of the current envelope, edit it, and copy the points (in the list format) to the clipboard (Menu "File" of the graph display => "Copy all Points ..."). This makes it easier to explore and paste the result into the python script when happy with the envelope!

```
from pyo import *
import random

s = Server().boot()

# Randomly built 10-points amplitude envelope.
t = 0
points = [(0.0, 0.0), (2.0, 0.0)]
for i in range(8):
    t += random.uniform(.1, .2)
    v = random.uniform(.1, .9)
    points.insert(-1, (t, v))

amp = Expseg(points, exp=3, mul=0.3)
amp.graph(title="Amplitude envelope")

sig = RCOsc(freq=[150,151], sharp=0.85, mul=amp)

# A simple linear function to vary the amount of frequency shifting.
sft = Linseg([(0.0, 0.0), (0.5, 20.0), (2, 0.0)])
sft.graph(yrange=(0.0, 20.0), title="Frequency shift")

fsg = FreqShift(sig, shift=sft).out()

rev = WGVerb(sig+fsg, feedback=0.9, cutoff=3500, bal=0.3).out()

def playnote():
    "Start the envelopes to play an event."
    amp.play()
    sft.play()

# Periodically call a function.
pat = Pattern(playnote, 2).play()

s.gui(locals())
```

2.6 06-filters

2.6.1 01-lowpass-filters.py - The effect of the order of a filter.

For this first example about filtering, we compare the frequency spectrum of three common lowpass filters.

- Tone : IIR first-order lowpass
- ButLP : IIR second-order lowpass (Butterworth)
- MoogLP : IIR fourth-order lowpass (+ resonance as an extra parameter)

Complementary highpass filters for the Tone and ButLP objects are Atone and ButHP. Another common highpass filter is the DCBlock object, which can be used to remove DC component from an audio signal.

The next example will present bandpass filters.

```
from pyo import *

s = Server().boot()

# White noise generator
n = Noise(.5)

# Common cutoff frequency control
freq = Sig(1000)
freq.ctrl([SLMap(50, 5000, "lin", "value", 1000)], title="Cutoff Frequency")

# Three different lowpass filters
tone = Tone(n, freq)
butlp = ButLP(n, freq)
mooglp = MoogLP(n, freq)

# Interpolates between input objects to produce a single output
sel = Selector([tone, butlp, mooglp]).out()
sel.ctrl(title="Filter selector (0=Tone, 1=ButLP, 2=MoogLP)")

# Displays the spectrum contents of the chosen source
sp = Spectrum(sel)

s.gui(locals())
```

2.6.2 02-bandpass-filters.py - Narrowing a bandpass filter bandwidth.

This example illustrates the difference between a simple IIR second-order bandpass filter and a cascade of second-order bandpass filters. A cascade of four bandpass filters with a high Q can be used as a efficient resonator on the signal.

```
from pyo import *

s = Server().boot()

# White noise generator
n = Noise(.5)

# Common cutoff frequency control
```

(continues on next page)

(continued from previous page)

```
freq = Sig(1000)
freq.ctrl([SLMap(50, 5000, "lin", "value", 1000)], title="Cutoff Frequency")

# Common filter's Q control
q = Sig(5)
q.ctrl([SLMap(0.7, 20, "log", "value", 5)], title="Filter's Q")

# Second-order bandpass filter
bp1 = Reson(n, freq, q=q)
# Cascade of second-order bandpass filters
bp2 = Resonx(n, freq, q=q, stages=4)

# Interpolates between input objects to produce a single output
sel = Selector([bp1, bp2]).out()
sel.ctrl(title="Filter selector (0=Reson, 1=Resonx)")

# Displays the spectrum contents of the chosen source
sp = Spectrum(sel)

s.gui(locals())
```

2.6.3 03-complex-resonator.py - Filtering by mean of a complex multiplication.

ComplexRes implements a resonator derived from a complex multiplication, which is very similar to a digital filter.

It is used here to create a rhythmic chime with varying resonance.

```
from pyo import *
import random

s = Server().boot()

# Six random frequencies.
freqs = [random.uniform(1000, 3000) for i in range(6)]

# Six different plucking speeds.
pluck = Metro([.9,.8,.6,.4,.3,.2]).play()

# LFO applied to the decay of the resonator.
decay = Sine(.1).range(.01, .15)

# Six ComplexRes filters.
rezos = ComplexRes(pluck, freqs, decay, mul=5).out()

# Change chime frequencies every 7.2 seconds
def new():
    freqs = [random.uniform(1000, 3000) for i in range(6)]
    rezos.freq = freqs
pat = Pattern(new, 7.2).play()

s.gui(locals())
```

2.6.4 04-phasing.py - The phasing effect.

The Phaser object implements a variable number of second-order allpass filters, allowing to quickly build complex phasing effects.

> A phaser is an electronic sound processor used to filter a signal by creating a series of peaks and troughs in the frequency spectrum. The position of the peaks and troughs of the waveform being affected is typically modulated so that they vary over time, creating a sweeping effect. For this purpose, phasers usually include a low-frequency oscillator. - https://en.wikipedia.org/wiki/Phaser_(effect)

A phase shifter unit can be built from scratch with the Allpass2 object, which implement a second-order allpass filter that create, when added to the original source, one notch in the spectrum.

```
from pyo import *

s = Server().boot()

# Simple fadein.
fade = Fader(fadein=.5, mul=.2).play()

# Noisy source.
a = PinkNoise(fade)

# These LFOs modulate the `freq`, `spread` and `q` arguments of
# the Phaser object. We give a list of two frequencies in order
# to create two-streams LFOs, therefore a stereo phasing effect.
lf1 = Sine(freq=[.1, .15], mul=100, add=250)
lf2 = Sine(freq=[.18, .13], mul=.4, add=1.5)
lf3 = Sine(freq=[.07, .09], mul=5, add=6)

# Apply the phasing effect with 20 notches.
b = Phaser(a, freq=lf1, spread=lf2, q=lf3, num=20, mul=.5).out()

s.gui(locals())
```

2.6.5 05-convolution-filters.py - Circular convolution.

A circular convolution is defined as the integral of the product of two functions after one is reversed and shifted.

Circular convolution allows to implement very complex FIR filters, at a CPU cost that is related to the filter impulse response (kernel) length.

Within pyo, there is a family of IR* filter objects using circular convolution with predefined kernel:

- IRAverage : moving average filter
- IRFM : FM-like filter
- IRPulse : comb-like filter
- RWinSinc : break wall filters (lp, hp, hp, br)

For general circular convolution, use the Convolve object with a PyoTableObject as the kernel, as in this example:

A white noise is filtered by four impulses taken from the input mic.

Call r1.play(), r2.play(), r3.play() or r4.play() in the Interpreter field while making some noise in the mic to fill the impulse response tables. The slider handles the morphing between the four kernels.

Call t1.view(), t2.view(), t3.view() or t4.view() to view impulse response tables.

Because circular convolution is very expensive, TLEN (in samples) should be keep small.

```
from pyo import *

# duplex=1 to tell the Server we need both input and output sounds.
s = Server(duplex=1).boot()

# Length of the impulse response in samples.
TLEN = 512

# Conversion to seconds for NewTable objects.
DUR = sampsToSec(TLEN)

# Excitation signal for the filters.
sf = Noise(.5)

# Signal from the mic to record the kernels.
inmic = Input()

# Four tables and recorders.
t1 = NewTable(length=DUR, chnls=1)
r1 = TableRec(inmic, table=t1, fadetime=.001)

t2 = NewTable(length=DUR, chnls=1)
r2 = TableRec(inmic, table=t2, fadetime=.001)

t3 = NewTable(length=DUR, chnls=1)
r3 = TableRec(inmic, table=t3, fadetime=.001)

t4 = NewTable(length=DUR, chnls=1)
r4 = TableRec(inmic, table=t4, fadetime=.001)

# Interpolation control between the tables.
pha = Sig(0)
pha.ctrl(title="Impulse responses morphing")

# Morphing between the four impulse responses.
res = NewTable(length=DUR, chnls=1)
morp = TableMorph(pha, res, [t1,t2,t3,t4])

# Circular convolution between the excitation and the morphed kernel.
a = Convolve(sf, table=res, size=res.getSize(), mul=.1).mix(2).out()

s.gui(locals())
```

2.6.6 06-vocoder.py - Analysis/resynthesis vocoder effect.

A vocoder is an analysis/resynthesis process that uses the spectral envelope of a first sound to shape the spectrum of a second sound. Usually (for the best results) the first sound should present a dynamic spectrum (for both frequencies and amplitudes) and the second sound should contain a rich and stable spectrum.

In this example, LFOs are applied to every dynamic argument of the Vocoder object to show the range of sound effects the user can get with a vocoder.

```
from pyo import *
from random import random
```

(continues on next page)

(continued from previous page)

```
s = Server().boot()

# First sound - dynamic spectrum.
spktrm = SfPlayer("../snds/baseballmajeur_m.aif", speed=[1,1.001], loop=True)

# Second sound - rich and stable spectrum.
excite = Noise(0.2)

# LFOs to modulate every parameters of the Vocoder object.
lf1 = Sine(freq=0.1, phase=random()).range(60, 100)
lf2 = Sine(freq=0.11, phase=random()).range(1.05, 1.5)
lf3 = Sine(freq=0.07, phase=random()).range(1, 20)
lf4 = Sine(freq=0.06, phase=random()).range(0.01, 0.99)

voc = Vocoder(spktrm, excite, freq=lf1, spread=lf2, q=lf3, slope=lf4, stages=32).out()

s.gui(locals())
```

2.6.7 07-hilbert-transform.py - Barberpole-like phasing effect.

This example uses two frequency shifters (based on complex modulation) linearly shifting the frequency content of a sound.

Frequency shifting is similar to ring modulation, except the upper and lower sidebands are separated into individual outputs.

```
from pyo import *

class ComplexMod:
    """
    Complex modulation used to shift the frequency
    spectrum of the input sound.
    """
    def __init__(self, hilb, freq):
        # Quadrature oscillator (sine, cosine).
        self._quad = Sine(freq, [0, 0.25])
        # real * cosine.
        self._mod1 = hilb['real'] * self._quad[1]
        # imaginary * sine.
        self._mod2 = hilb['imag'] * self._quad[0]
        # Up shift corresponds to the sum frequencies.
        self._up = (self._mod1 + self._mod2) * 0.7

    def out(self, chnl=0):
        self._up.out(chnl)
        return self

s = Server().boot()

# Large spectrum source.
src = PinkNoise(.2)

# Apply the Hilbert transform.
hilb = Hilbert(src)
```

(continues on next page)

(continued from previous page)

```
# LFOs controlling the amount of frequency shifting.
lf1 = Sine(.03, mul=6)
lf2 = Sine(.05, mul=6)

# Stereo Single-Sideband Modulation.
wetl = ComplexMod(hilb, lf1).out()
wetr = ComplexMod(hilb, lf2).out(1)

# Mixed with the dry sound.
dry = src.mix(2).out()

s.gui(locals())
```

2.7 07-effects

2.7.1 01-flanger.py - Hand-made simple flanger.

A flanger is an audio effect produced by mixing two identical signals together, one signal delayed by a small and gradually changing period. This produces a swept comb filter effect: peaks and notches are produced in the resulting frequency spectrum, related to each other in a linear harmonic series. Varying the time delay causes these to sweep up and down the frequency spectrum.

```
from pyo import *

s = Server().boot()

# Rich frequency spectrum as stereo input source.
amp = Fader(fadein=0.25, mul=0.5).play()
src = PinkNoise(amp).mix(2)

# Flanger parameters                        == unit ==
middelay = 0.005                            # seconds

depth = Sig(0.99)                           # 0 --> 1
depth.ctrl(title="Modulation Depth")
lfospeed = Sig(0.2)                         # Hertz
lfospeed.ctrl(title="LFO Frequency in Hz")
feedback = Sig(0.5, mul=0.95)               # 0 --> 1
feedback.ctrl(title="Feedback")

# LFO with adjusted output range to control the delay time in seconds.
lfo = Sine(freq=lfospeed, mul=middelay*depth, add=middelay)

# Dynamically delayed signal. The source passes through a DCBlock
# to ensure there is no DC offset in the signal (with feedback, DC
# offset can be fatal!).
flg = Delay(DCBlock(src), delay=lfo, feedback=feedback)

# Mix the original source with its delayed version.
# Compress the mix to normalize the output signal.
cmp = Compress(src+flg, thresh=-20, ratio=4).out()

s.gui(locals())
```

2.7.2 02-schroeder-reverb.py - Simple reverberator based on Schroeder's algorithms.

An artificial reverberation based on the work of Manfred Schroeder.

This reverberator takes a monophonic input and outputs two uncorrelated reverberated signals.

This algorithm presents four parallel comb filters fedding two serial allpass filters. An additional lowpass filter is used at the end to control the brightness of the reverberator.

The manual example for the Allpass object presents an other Schroeder's reverberator.

If you are interested in builtin reverberation objects, see:

Freeverb, WGVerb, STRev, CvlVerb

Not really reverbs, but you can build some cool resonant effects with:

Waveguide, AllpassWG

```
from pyo import *

s = Server(duplex=0).boot()

soundfile = SndTable(SNDS_PATH + "/transparent.aif")

src = Looper(soundfile, dur=2, xfade=0, mul=0.3)
src2 = src.mix(2).out()

# Four parallel stereo comb filters. The delay times are chosen
# to be as uncorrelated as possible. Prime numbers are a good
# choice for delay lengths in samples.
comb1 = Delay(src, delay=[0.0297, 0.0277], feedback=0.65)
comb2 = Delay(src, delay=[0.0371, 0.0393], feedback=0.51)
comb3 = Delay(src, delay=[0.0411, 0.0409], feedback=0.5)
comb4 = Delay(src, delay=[0.0137, 0.0155], feedback=0.73)

combsum = src + comb1 + comb2 + comb3 + comb4

# The sum of the original signal and the comb filters
# feeds two serial allpass filters.
all1 = Allpass(combsum, delay=[.005, .00507], feedback=0.75)
all2 = Allpass(all1, delay=[.0117, .0123], feedback=0.61)

# Brightness control.
lowp = Tone(all2, freq=3500, mul=.25).out()

s.gui(locals())
```

2.7.3 03-fuzz-disto.py - Hand-written asymmetrical tranfert function.

This example implements a fuzz distortion. Bandpass filtered signal is eavily boosted, then feeded to an asymmetrical transfert function, and finally lowpass filtered to smooth out the sound. Balance between the dry and the distorted sound is applied before sending the signal to the outputs.

Builtin objects to distort an audio signal:

Disto, Clip, Mirror, Wrap

Degrade object applies some kind of distortion to a signal by changing its sampling rate and bit depth.

```
from pyo import *

s = Server(duplex=0).boot()

# The audio source (try with your own sounds).
SOURCE = '../snds/flute.aif'

# Distortion parameters
BP_CENTER_FREQ = 400        # Bandpass filter center frequency.
BP_Q = 3                    # Bandpass Q (center_freq / Q = bandwidth).
BOOST = 25                  # Pre-boost (linear gain).
LP_CUTOFF_FREQ = 3000       # Lowpass filter cutoff frequency.
BALANCE = 0.7               # Balance dry - wet.

src = SfPlayer(SOURCE, loop=True).mix(2)

# The transfert function is build in two phases.

# 1. Transfert function for signal lower than 0.
table = ExpTable([(0,-.25),(4096,0),(8192,0)], exp=30)

# 2. Transfert function for signal higher than 0.
# First, create an exponential function from 1 (at the beginning of the table)
# to 0 (in the middle of the table).
high_table = ExpTable([(0,1),(2000,1),(4096,0),(4598,0),(8192,0)],
                      exp=5, inverse=False)
# Then, reverse the table's data in time, to put the shape in the second
# part of the table.
high_table.reverse()

# Finally, add the second table to the first, point by point.
table.add(high_table)

# Show the transfert function.
table.view(title="Transfert function")

# Bandpass filter and boost gain applied on input signal.
bp = ButBP(src, freq=BP_CENTER_FREQ, q=BP_Q)
boost = Sig(bp, mul=BOOST)

# Apply the transfert function.
sig = Lookup(table, boost)

# Lowpass filter on the distorted signal.
lp = ButLP(sig, freq=LP_CUTOFF_FREQ, mul=.7)

# Balance between dry and wet signals.
mixed = Interp(src, lp, interp=BALANCE)

# Send the signal to the outputs.
out = (mixed * 0.3).out()

# Show the resulting waveform.
sc = Scope(mixed)

s.gui(locals())
```

2.7.4 04-ping-pong-delay.py - Stereo ping-pong delay.

A stereo ping-pong delay is when you use two delays, one hard left and the other hard right, and the delays alternate.

This exemple illustrates how we can pass the signal of a first object to the input of a second even if the first object does not exist yet when the second is created.

```
from pyo import *
import random

s = Server(duplex=0).boot()

# Compute the duration, in seconds, of one buffer size.
buftime = s.getBufferSize() / s.getSamplingRate()

# Delay parameters
delay_time_l = Sig(0.125)    # Delay time for the left channel delay.
delay_time_l.ctrl()
delay_feed = Sig(0.75)       # Feedback value for both delays.
delay_feed.ctrl()

# Because the right delay gets its input sound from the left delay, while
# it is computed before (to send its output sound to the left delay), it
# will be one buffer size late. To compensate this additional delay on the
# right, we substract one buffer size from the real delay time.
delay_time_r = Sig(delay_time_l, add=-buftime)

# Setup up a soundfile player.
sf = SfPlayer('../snds/alum1.wav').stop()

# Send the original sound to both speakers.
sfout = sf.mix(2).out()

# Initialize the right delay with zeros as input because the left delay
# does not exist yet.
right = Delay(Sig(0), delay=delay_time_r).out(1)

# Initialize the left delay with the original mono source and the right
# delay signal (multiplied by the feedback value) as input.
left = Delay(sf + right * delay_feed, delay=delay_time_l).out()

# One issue with recursive cross-delay is if we set the feedback to
# 0, the right delay never gets any signal. To resolve this, we add a
# non-recursive delay, with a gain that is the inverse of the feedback,
# to the right delay input.
original_delayed = Delay(sf, delay_time_l, mul=1-delay_feed)

# Change the right delay input (now that the left delay exists).
right.setInput(original_delayed + left * delay_feed)

def playit():
    "Assign a sound to the player and start playback."
    which = random.randint(1, 4)
    path = "../snds/alum%d.wav" % which
    sf.path = path
    sf.play()

# Call the function "playit" every second.
```

(continues on next page)

(continued from previous page)

```
pat = Pattern(playit, 1).play()

s.gui(locals())
```

2.7.5 06-hand-made-chorus.py - Hand-written 8 delay lines stereo chorus.

A chorus (or ensemble) is a modulation effect used to create a richer, thicker sound and add subtle movement. The effect roughly simulates the slight variations in pitch and timing that occur when multiple performers sing or play the same part.

A single voice chorus uses a single delay that creates a single modulated duplicate of the incoming audio. Basic chorus effects and inexpensive guitar pedals are often single-voice.

A multiple voice chorus uses multiple modulated delays to create a richer sound with more movement than a single voice chorus.

The Chorus object (from pyo) implements an 8 delay lines chorus and should use less CPU than the hand-written version. this example's purpose is only to show how it works or to be used as a starting point to build an extended version.

```
from pyo import *

s = Server(duplex=0).boot()

# Start a source sound.
sf = SfPlayer('../snds/baseballmajeur_m.aif', speed=1, loop=True, mul=.3)
# Mix the source in stereo and send the signal to the output.
sf2 = sf.mix(2).out()

# Sets values for 8 LFO'ed delay lines (you can add more if you want!).
# LFO frequencies.
freqs = [.254, .465, .657, .879, 1.23, 1.342, 1.654, 1.879]
# Center delays in seconds.
cdelay = [.0087, .0102, .0111, .01254, .0134, .01501, .01707, .0178]
# Modulation depths in seconds.
adelay = [.001, .0012, .0013, .0014, .0015, .0016, .002, .0023]

# Create 8 sinusoidal LFOs with center delays "cdelay" and depths "adelay".
lfos = Sine(freqs, mul=adelay, add=cdelay)

# Create 8 modulated delay lines with a little feedback and send the signals
# to the output. Streams 1, 3, 5, 7 to the left and streams 2, 4, 6, 8 to the
# right (default behaviour of the out() method).
delays = Delay(sf, lfos, feedback=.5, mul=.5).out()

s.gui(locals())
```

2.7.6 07-hand-made-harmonizer.py - Hand-written harmonizer effect.

A harmonizer is a type of pitch shifter that combines the "shifted" pitch with the original pitch to create a two or more note harmony.

The implementation consists of two overlapping delay lines for which the reading head speed is tuned to transpose the signal by an amount specified in semitones.

The Harmonizer object (from pyo) implements an pitch shifter and should use less CPU than the hand-written version. this example's purpose is only to show how it works or to be used as a starting point to build an extended version.

```
from pyo import *

s = Server(duplex=0).boot()

# Play a melodic sound and send its signal to the left speaker.
sf = SfPlayer('../snds/flute.aif', speed=1, loop=True, mul=.5).out()

# Half-sine window used as the amplitude envelope of the overlaps.
env = WinTable(8)

# Length of the window in seconds.
wsize = .1

# Amount of transposition in semitones.
trans = -7

# Compute the transposition ratio.
ratio = pow(2., trans/12.)

# Compute the reading head speed.
rate = -(ratio-1) / wsize

# Two reading heads out-of-phase.
ind = Phasor(freq=rate, phase=[0,0.5])

# Each head reads the amplitude envelope...
win = Pointer(table=env, index=ind, mul=.7)

# ... and modulates the delay time (scaled by the window size) of a delay line.
# mix(1) is used to mix the two overlaps on a single audio stream.
snd = Delay(sf, delay=ind*wsize, mul=win).mix(1)

# The transposed signal is sent to the right speaker.
snd.out(1)

s.gui(locals())
```

2.8 08-dynamics

2.8.1 01-dynamic-range.py - Adjust the dynamic range of the signal.

Comparison of three objects used to adjust the dynamic range of the signal.

- Compress : Reduces the dynamic range of an audio signal.
- Expand : Increases the dynamic range of an audio signal.
- Gate : Allows a signal to pass only when its amplitude is above a threshold.

These three objects, by default, process independently each audio stream relatively to its own RMS value. This can be a problem if they are passed a stereo signal (or any multiphonic signals) where both channels should be processed in the same way.

An alternative usage to the one illustrated below is to mix a multi-channel signal prior to the dynamic processor and to tell the object to output the amplitude value that should be applied to all streams at once. Something like this:

```
>>> sf = SfPlayer("your/stereo.sound.wav")
>>> cmp = Compress(sf.mix(1), thresh=-18, ratio=3, outputAmp=True)
>>> # Compress both signal with the common amplitude curve.
>>> compressed = sf * cmp
```

```
from pyo import *

s = Server().boot()

# The original source.
src = SfPlayer("../snds/drumloop.wav", loop=True)

# The three dynamic processing.
cmp = Compress(src, thresh=-18, ratio=3, risetime=0.005, falltime=0.05, knee=0.5)
exp = Expand(src, downthresh=-32, upthresh=-12, ratio=3, risetime=0.005, falltime=0.
↪05)
gat = Gate(src, thresh=-40, risetime=0.005, falltime=0.05)

# These are labels that are shown as the scope window title.
labels =    ["Original", "Compressed", "Expanded", "Gated"]

# List of signals to choose from.
signals =   [src, cmp, exp, gat]

# Selector is used here to choose which signal to listen to.
output = Selector(signals)

# Converts the signal from mono to stereo.
stout = output.mix(2).out()

# Live oscilloscope of the alternated signals.
sc = Scope(output, wintitle="=== Original ===")

# The endOfLoop function cycles through the different signals
# and change the selection of the Selector object.
num_of_sigs = len(signals)
c = 1
def endOfLoop():
    global c
    output.voice = c
    if sc.viewFrame is not None:
        sc.viewFrame.SetTitle("=== %s ===" % labels[c])
    c = (c + 1) % num_of_sigs

# endOfLoop is called every time the SfPlayer reaches the end of the sound.
tf = TrigFunc(src["trig"], endOfLoop)

s.gui(locals())
```

2.8.2 02-ducking.py - Adjust the gain of a signal based on the presence of another one.

Ducking is an audio effect commonly used in radio. In ducking, the level of one audio signal is reduced by the presence of another signal.

Here we use a Follower object to track the RMS envelope of the voice signal. Then we use an audio conditional to create a switch, whose value is 1 when the voice is talking and 0 when it is silent. This signal is finally used to change the amplitude of the music whenever the voice is talking.

```
from pyo import *

s = Server().boot()

# Alternate voice and silence.
table = SndTable(SNDS_PATH + "/transparent.aif")
metro = SDelay(Metro(3).play(), 1)
voice = TrigEnv(metro, table, dur=table.getDur(), mul=0.7)
stvoice = voice.mix(2).out()

# Play some music-box style tune!
freqs = midiToHz([60, 62, 64, 65, 67, 69, 71, 72])
choice = Choice(choice=freqs, freq=[1,2,3,4])
port = Port(choice, risetime=.001, falltime=.001)
sines = SineLoop(port, feedback=0.05)
music = SPan(sines, pan=[0, 1, .2, .8, .5], mul=.1).mix(2)

# Follow voice RMS amplitude.
follow = Follower(voice, freq=10)
# talk = 1 if voice is playing and 0 if not.
talk = follow > 0.005

# Smooth the on/off signal (rising is faster than falling)...
amp = Port(talk, risetime=0.05, falltime=0.1)
# ... then rescale it (1 when no voice and 0.1 when voice is playing).
ampscl = Scale(amp, outmin=1, outmax=0.1)

# Display the gain factor.
sc = Scope(ampscl)

# Apply gain factor and output music.
outsynth = (music * ampscl).out()

s.gui(locals())
```

2.8.3 03-gated-verb.py - Gated reverb applied to a drum loop.

The gated reverb effect, which was most popular in the 1980s, is made using a combination of strong reverb and a noise gate. The drum sound passes through a strong reverb, which is rapidly cut off with a gate driven by the dry signal.

```
from pyo import *

s = Server().boot()

# Play the drum lopp..
```

(continues on next page)

(continued from previous page)

```
sf = SfPlayer('../snds/drumloop.wav', loop=True)

# Use a gate to generate the gain curve that will be applied to the reverb.
gate = Gate(sf, thresh=-50, risetime=0.005, falltime=0.04, lookahead=4,
            outputAmp=True)

# Strong reverb.
rev = Freeverb(sf.mix(2), size=0.95, damp=0.3, bal=1.0)

# Compress the reverb signal and control its amplitude with the gating signal.
cmp = Compress(rev, thresh=-12, ratio=3, risetime=0.005, falltime=0.05,
               lookahead=4, knee=0.5, mul=gate)

# Balance between the dry and wet (gated-reverb) signals.
output = Interp(sf.mix(2), cmp, interp=0.2).out()

s.gui(locals())
```

2.8.4 04-rms-tracing.py - Auto-wah effect.

The auto-wah effect (also know as "envelope following filter") is like a wah-wah effect, but instead of being controlled by a pedal, it is the RMS amplitude of the input sound which control it. The envelope follower (RMS) is rescaled and used to change the frequency of a bandpass filter applied to the source.

```
from pyo import *

s = Server().boot()

MINFREQ = 250
MAXFREQ = 5000

# Play the drum lopp.
sf = SfPlayer('../snds/drumloop.wav', loop=True)

# Follow the amplitude envelope of the input sound.
follow = Follower(sf)

# Scale the amplitude envelope (0 -> 1) to the desired frequency
# range (MINFREQ -> MAXFREQ).
freq = Scale(follow, outmin=MINFREQ, outmax=MAXFREQ)

# Filter the signal with a band pass. Play with the Q to make the
# effect more or less present.
filter = ButBP(sf.mix(2), freq=freq, q=2).out()

s.gui(locals())
```

2.9 09-callbacks

2.9.1 01-periodic-calls.py - Periodic event generation.

When an audio thread is running, the user must take care of not freezing the interpreter lock by using blocking function like sleep(). It is better to use objects from the library that are designed to call a function without blocking the interpreter. Because these objects are aware of the server's timeline, they are well suited for generating rhythmic sequences of functions calls.

Pyo offers three objects for calling function:

- Pattern: Periodically calls a Python function.
- Score: Calls functions by incrementation of a preformatted name.
- CallAfter: Calls a Python function after a given time.

This example shows the usage of the Pattern object to create a sequence of audio events.

```
from pyo import *
import random

s = Server().boot()

# A small instrument to play the events emitted by the function call.
amp = Fader(fadein=0.005, fadeout=0.05, mul=.15)
osc = RCOsc(freq=[100, 100], mul=amp).out()
dly = Delay(osc, delay=1.0, feedback=0.5).out()

def new_event():
    # Choose a duration for this event.
    dur = random.choice([.125, .125, .125, .25, .25, .5, 1])

    # Assigns the new duration to the envelope.
    amp.dur = dur
    # Assigns the new duration to the caller, thus the next function call
    # will be only after the current event has finished.
    pat.time = dur

    # Choose a new frequency.
    freq = random.choice(midiToHz([60, 62, 63, 65, 67, 68, 71, 72]))

    # Replace oscillator's frequencies.
    osc.freq = [freq, freq * 1.003]

    # Start the envelope.
    amp.play()

# A Pattern object periodically call the referred function given as
# argument. The "time" argument is the delay between successive calls.
# The play() method must be explicitly called for a Pattern object
# to start its processing loop.
pat = Pattern(function=new_event, time=0.25).play()

s.gui(locals())
```

2.9.2 02-score-calls.py - Sequencing the function calls.

The Score object takes in input an audio stream containing integers and any time the integer changes, it calls a function with a generic name to which the integer is added. This allows the user to build a sequence of functions and to control how and when each one is called.

This example uses a metronome and a counter to generate a stream of integers at a specific rate. The called functions change the oscillator frequencies to produce a chords sequence.

```
from pyo import *

s = Server().boot()

# A four-streams oscillator to produce a chord.
osc = SineLoop(freq=[0, 0, 0, 0], feedback=0.05, mul=.2)
rev = WGVerb(osc.mix(2), feedback=0.8, cutoff=4000, bal=0.2).out()

def set_osc_freqs(notes):
    # PyoObject.set() method allow to change the value of an attribute
    # with an audio ramp to smooth out the change.
    osc.set(attr="freq", value=midiToHz(notes), port=0.005)

# The sequence of functions (some call set_osc_freqs to change the notes).
def event_0():
    set_osc_freqs([60, 64, 67, 72])

def event_1():
    pass

def event_2():
    set_osc_freqs([60, 64, 67, 69])

def event_3():
    pass

def event_4():
    set_osc_freqs([60, 65, 69, 76])

def event_5():
    pass

def event_6():
    set_osc_freqs([62, 65, 69, 74])

def event_7():
    set_osc_freqs([59, 65, 67, 74])

# Integer generator (more about triggers in section 12-triggers)
metro = Metro(time=0.5).play()
count = Counter(metro, min=0, max=8)

# Score calls the function named "event_" + count. (if count is 3,
# function named "event_3" is called without argument.
score = Score(count, fname="event_")

s.gui(locals())
```

2.9.3 03-delayed-calls.py - Calling a function once, after a given delay.

If you want to setup a callback once in the future, the CallAfter object is very easy to use. You just give it the function name, the time to wait before making the call and an optional argument.

```
from pyo import *

s = Server().boot()

# A four-streams oscillator to produce a chord.
amp = Fader(fadein=0.005, fadeout=0.05, mul=0.2).play()
osc = SineLoop(freq=[0, 0, 0, 0], feedback=0.05, mul=amp)
rev = WGVerb(osc.mix(2), feedback=0.8, cutoff=4000, bal=0.2).out()

# A function to change the oscillator's frequencies and start the envelope.
def set_osc_freqs(notes):
    print(notes)
    osc.set(attr="freq", value=midiToHz(list(notes)), port=0.005)
    amp.play()

# Initial chord.
set_osc_freqs([60, 64, 67, 72])

# We must be sure that our CallAfter object stays alive as long as
# it waits to call its function. If we don't keep a reference to it,
# it will be garbage-collected before doing its job.
call = None
def new_notes(notes):
    global call     # Use a global variable.
    amp.stop()      # Start the fadeout of the current notes...
    # ... then, 50 ms later, call the function that change the frequencies.
    call = CallAfter(set_osc_freqs, time=0.05, arg=notes)

# The sequence of events. We use a tuple for the list of frequencies
# because PyoObjects spread lists as argument over all their internal
# streams. This means that with a list of frequencies, only the first
# frequency would be passed to the callback of the first (and single)
# stream (a list of functions at first argument would create a
# multi-stream object). A tuple is treated as a single argument.
c1 = CallAfter(new_notes, time=0.95, arg=(60, 64, 67, 69))
c2 = CallAfter(new_notes, time=1.95, arg=(60, 65, 69, 76))
c3 = CallAfter(new_notes, time=2.95, arg=(62, 65, 69, 74))
c4 = CallAfter(new_notes, time=3.45, arg=(59, 65, 67, 74))
c5 = CallAfter(new_notes, time=3.95, arg=(60, 64, 67, 72))
# The last event activates the fadeout of the amplitude envelope.
c6 = CallAfter(amp.stop, time=5.95, arg=None)

s.gui(locals())
```

2.10 10-tables

2.10.1 01-envelopes.py - Using break-point function to control an FM synthesis.

This example shows how break-point tables can be used to control synthesis/effect parameters at audio rate. Pyo offers many objects to generate break-point function:

- LinTable: Construct a table from segments of straight lines.
- CosTable: Construct a table from cosine interpolated segments.
- ExpTable: Construct a table from exponential interpolated segments.
- CurveTable: Construct a table from curve, with tension and bias, interpolated segments.
- LogTable: Construct a table from logarithmic segments.
- CosLogTable: Construct a table from logarithmic-cosine segments.

These objects implement a *graph()* method (as well as Linseg and Expseg) which show a graph window with which the user can set the shape of the trajectory.

With the focus on the graph window, the copy menu item (Ctrl+C) saves to the clipboard the list of points in a format well suited to be paste in the code. Useful to experiment graphically and then copy/paste the result in the script.

To play more notes, in the Interpreter field of the Server GUI, call the *note(freq, dur)* function with the desired frequency and duration.

Note: The wxPython Phoenix graphical library must be installed to use the graph. Infos at:

http://www.wxpython.org/

```
from pyo import *

s = Server().boot()

# Defines tables for the amplitude, the ratio and the modulation index.
amp_table = CosTable([(0,0), (100,1), (1024,.5), (7000,.5), (8192,0)])
rat_table = ExpTable([(0, .5), (1500, .5), (2000, .25), (3500, .25),
                      (4000, 1), (5500, 1), (6000, .5), (8192, .5)])
ind_table = LinTable([(0, 20), (512, 10), (8192, 0)])

# call their graph() method. Use the "yrange" argument to set the minimum
# and maximum bundaries of the graph (defaults to 0 and 1).
amp_table.graph(title="Amplitude envelope")
rat_table.graph(title="Ratio envelope")
ind_table.graph(yrange=(0, 20), title="Modulation index envelope")

# Initialize the table readers (TableRead.play() must be called explicitly).
amp = TableRead(table=amp_table, freq=1, loop=False, mul=.3)
rat = TableRead(table=rat_table, freq=1, loop=False)
ind = TableRead(table=ind_table, freq=1, loop=False)

# Use the signals from the table readers to control an FM synthesis.
fm = FM(carrier=[100,100], ratio=rat, index=ind, mul=amp).out()

# Call the "note" function to generate an event.
def note(freq=100, dur=1):
    fm.carrier = [freq, freq*1.005]
    amp.freq = 1.0 / dur
    rat.freq = 1.0 / dur
    ind.freq = 1.0 / dur
    amp.play()
    rat.play()
    ind.play()

# Play one note, carrier = 100 Hz, duration = 2 seconds.
note(200, 2)
```

(continues on next page)

(continued from previous page)

```
s.gui(locals())
```

2.10.2 02-scrubbing.py - Navigate through a sound table.

The SndTable object allows to transfer the data from a soundfile into a function table for a fast access to every sample.

This example illustrates how to do scrubbing with the mouse in a sound window. The mouse position is then used to control the position and the balance of a simple stereo granulator.

Give the focus to the Scrubbing window then click and move the mouse...

```
from pyo import *

s = Server().boot()

# The callback given to the SndTable.view() method.
def mouse(mpos):
    print("X = %.2f, Y = %.2f" % tuple(mpos))
    # X value controls the granulator pointer position.
    position.value = mpos[0]
    # Y value controls the balance between left and right channels.
    l, r = 1. - mpos[1], mpos[1]
    leftRightAmp.value = [l, r]

# Load and normalize the sound in the table.
snd = SndTable('../snds/ounkmaster.aif').normalize()
# Open the waveform view with a mouse position callback.
snd.view(title="Scrubbing window", mouse_callback=mouse)

# Left and right channel gain values.
leftRightAmp = SigTo([1,1], time=0.1, init=[1,1], mul=.1)
# Position, in samples, in the SndTable.
position = SigTo(0.5, time=0.1, init=0.5, mul=snd.getSize(), add=Noise(5))

# Simple sound granulator.
gran = Granulator(table=snd,             # the sound table.
                  env=HannTable(),       # the grain envelope.
                  pitch=[.999, 1.0011], # global pitch (change every grain).
                  pos=position,          # position in the table where to start a new␣
↪grain.
                  dur=Noise(.002, .1),   # duration of the grain (can be used to␣
↪transpose per grain).
                  grains=64,             # the number of grains.
                  basedur=.1,            # duration for which the grain is not␣
↪transposed.
                  mul=leftRightAmp       # stereo gain.
                 ).out()

s.gui(locals())
```

2.10.3 03-looper.py - High quality crossfading multimode sound looper.

The Looper object reads audio from a PyoTableObject and plays it back in a loop with user-defined pitch, start time, duration and crossfade time.

Looper will send a trigger signal every time a new playback starts, which means at the object activation and at the beginning of the crossfade when looping. User can retrieve the trigger streams with obj['trig'].

Looper also outputs a time stream, given the current position of the reading pointer, normalized between 0.0 and 1.0 (1.0 means the beginning of the crossfade), inside the loop. User can retrieve the trigger stream with obj['time'].

Some methods let change the behaviour of the loop:

- **Looper.appendFadeTime(boolean):** If True, the crossfade starts after the loop duration.
- **Looper.fadeInSeconds(boolean):** If True, the crossfade duration (*xfade* attribute) is set in seconds.

```
from pyo import *

s = Server().boot()

table = SndTable('../snds/baseballmajeur_m.aif')
table.view()

looper = Looper(table = table,          # The table to read.
                pitch = 1.0,            # Speed factor, 0.5 means an octave lower,
                                        # 2 means an octave higher.
                start = 0,              # Start time of the loop, in seconds.
                dur = table.getDur(),   # Duration of the loop, in seconds.
                xfade = 25,             # Duration of the crossfade, in % of the loop␣
↪length.
                mode = 1,               # Looping mode: 0 = no loop, 1 = forward,
                                        #               2 = backward, 3 = back-and-
↪forth.
                xfadeshape = 0,         # Shape of the crossfade envelope: 0 = linear
                                        #   1 = equal power, 2 = sigmoid.
                startfromloop = False,  # If True, the playback starts from the loop␣
↪start
                                        # point. If False, the playback starts from␣
↪the
                                        # beginning and enters the loop mode when␣
↪crossing
                                        # the loop start point.
                interp = 4,             # Interpolation method (used when speed != 1):
                                        # 1 = none, 2 = linear, 3 = cosine, 4 = cubic.
                autosmooth = True,      # If True, a lowpass filter, following the␣
↪pitch,
                                        # is applied on the output signal to reduce␣
↪the
                                        # quantization noise produced by very low␣
↪transpositions.
                                        # interp = 4 and autosmooth = True give a␣
↪very high
                                        # quality reader for playing sound at low␣
↪rates.
                mul = 0.5
               )
looper.ctrl()

stlooper = looper.mix(2).out()

s.gui(locals())
```

2.10.4 04-granulation.py - Full control granular synthesis.

The output of a Granular Synthesis is composed of many individual grains of sound. A grain is a short chunk of sound, typically between 10 and 100 ms (but can also vary outside this range), with an amplitude envelope in the shape of a bell curve. The sonic quality of a granular texture is a result of the distribution of grains in time and of the parameters selected for the synthesis of each grain.

This example shows the usage of the most featured granulation object of the library!

Available granulation objects, in order of complexity, are:

- Granulator
- Granule
- Particle
- Particle2

```
from pyo import *

s = Server().boot()

snd = SndTable('../snds/baseballmajeur_m.aif')
snd.view()

# Remove sr/4 samples to the size of the table, just to be sure
# that the reading pointer never exceeds the end of the table.
end = snd.getSize() - s.getSamplingRate() * 0.25

# A Tuckey envelope for the grains (also known as flat-top envelope).
env = WinTable(7)
env.view(title="Grain envelope")

# The grain pitch has a default value of 1, to which we can add a
# randomness factor by raising the "mul" value of the Noise.
pit = Noise(0, add=1)
pit.ctrl([SLMap(0, 1, "lin", "mul", 0)], title="Pitch Randomness")

# The grain position oscillates slowly between the beginning and
# the end of the table. We add a little jitter to the position to
# attenuate phasing artifacts when overlapping the grains.
pososc = Sine(0.05).range(0, end)
posrnd = Noise(mul=0.01, add=1)
pos = pososc * posrnd

# The grain panoramisation is completely random.
pan = Noise(mul=0.5, add=0.5)

# The grain filter center frequency choices are the first 40
# harmonics of a base frequency oscillating between 75 and 125 Hz.
cf = Sine(freq=0.07).range(75, 125)
fcf = Choice(list(range(1, 40)), freq=150, mul=cf)

grn = Particle2(table = snd,        # The table to read.
                env = env,          # The grain envelope.
                dens = 128,         # The density of grains per second.
                # The next arguments are sampled at the beginning of the grain
                # and hold their until the end of the grain.
                pitch = pit,        # The pitch of the grain.
```

(continues on next page)

(continued from previous page)

```
                pos = pos,          # The position of the grain in the table.
                dur = 0.2,          # The duration of the grain in seconds.
                dev = 0.005,        # The maximum deviation of the start time,
                                    # synchronous versus asynchronous granulation.
                pan = pan,          # The pan value of the grain.
                filterfreq = fcf,   # The filter frequency of the grain.
                filterq = 20,       # The filter Q of the grain.
                filtertype = 2,     # The filter type of the grain.
                # End of sampled arguments.
                chnls = 2,          # The output number of streams of the granulator.
                mul = .2
               )
grn.ctrl()

# Some grains can be surprisingly loud so we compress the output of the granulator.
comp = Compress(grn, thresh=-20, ratio=4, risetime=0.005, falltime=0.10, knee=0.5,␣
↪mul=2).out()

s.gui(locals())
```

2.10.5 05-micro-montage.py - Create table from random chunks of a soundfile.

This example creates a new sound table from random chunks of a soundfile.

The SndTable object has some methods to help mixing different sounds or parts of sounds into a single table:

- **setSound(path, start=0, stop=None)** Replace the table content with the new sound.
- **insert(path, pos=0, crossfade=0, start=0, stop=None)** Insert samples at a given position in the table, with crossfades.
- **append(path, crossfade=0, start=0, stop=None)** Append samples at the end of the table, with crossfade.

To generate a new mix in the sound table, call the *gen()* function in the Interpreter field of the Server GUI.

```
from pyo import *
import random

s = Server().boot()

# Path and duration of the choosen soundfile.
path = "../snds/baseballmajeur_m.aif"
snddur = sndinfo(path)[1]

# Initialize an empty sound table.
table = SndTable()

# Before generating a new table mix, we activate the fadeout of the
# envelope to ensure that the table is modified in the silence. As
# soon as the table is generated, we call the envelope fadein.
fade = Fader(fadein=0.005, fadeout=0.005, dur=0, mul=0.7)

# Reads the table with forward looping.
loop = Looper(table, dur=table.getDur(), xfade=5, mul=fade)

# Adds some reverb and send the signal to the output.
rvrb = STRev(loop, inpos=0.50, revtime=1.5, bal=0.15).out()
```

(continues on next page)

(continued from previous page)

```
def addsnd():
    # Randomly choose a new starting point and a new duration.
    start = random.uniform(0, snddur * 0.7)
    duration = random.uniform(.1, .3)

    # Randomly choose an insert point in the sound table and a croosfade time.
    pos = random.uniform(0.05, table.getDur()-0.5)
    cross = random.uniform(0.04, duration/2)

    # Insert the new chunk in the current sound table.
    table.insert(path, pos=pos, crossfade=cross, start=start, stop=start+duration)

def delgen():
    # Randomly choose a new starting point and a new duration.
    start = random.uniform(0, snddur * 0.7)
    duration = random.uniform(.1, .3)

    # Load the chosen segment in the sound table.
    table.setSound(path, start=start, stop=start+duration)

    # Add 10 more chunks.
    for i in range(10):
        addsnd()

    # Set the new table duration to the Looper and reset it.
    loop.dur = table.getDur()
    loop.reset()

    # Activate the envelope fadein.
    fade.play()

# CallAfter calls a function after a given delay time.
caller = CallAfter(function=delgen, time=0.005).stop()

def gen():
    "Create a new mix in the sound table."
    fade.stop()     # Launch the fadeout...
    caller.play()   # ... then call the delayed generation.

# Generate the intial table.
gen()

s.gui(locals())
```

2.10.6 06-table-stutter.py - Variable length table reading.

This little program use a function and a time-variable function caller to stutter a sound loaded in a table. The starting point of the table playback is set initially to near the end of the table, and move toward the beginning each time a new playback is called.

For the playback, we use a Pointer object, which is a table reader with control on the pointer position (normalized between 0 and 1).

```
from pyo import *
import random

s = Server().boot()

STUTTER = 0.025 # Delta time added each time the playback restart.
FADETIME = 0.01 # Fadein-fadeout time to avoid clicks.

# Load a sound in the table and get its duration.
table = SndTable(SNDS_PATH+"/transparent.aif")
tabdur = table.getDur()

# Intialize the line used to read the table.
line = Linseg([(0, 0), (1, 1)])

# Amplitude envelope, to avoid clicks.
amp = Fader(FADETIME, FADETIME, dur=STUTTER, mul=0.5)
amp.setExp(2)

# Read the sound and mix it to stereo.
read = Pointer(table=table, index=line, mul=amp).mix(2).out()

# Global variables (start position and playback duration).
start = tabdur - STUTTER
dur = STUTTER

def go():
    "Read a segment of a sound table and set the duration before the next iteration."
    global start, dur

    # Create the pointer segment (from normalized start position to the end of the␣
↪table)
    line.list = [(0, start/tabdur), (dur, 1)]

    # Assign duration to the envelope and the function caller (Pattern).
    amp.dur = pat.time = dur

    # Decrement start position and increment duration by STUTTER seconds.
    start -= STUTTER
    dur += STUTTER

    # Reset start and dur variables when reaching the beginning of the sound.
    if start < 0:
        start = tabdur - STUTTER
        dur = STUTTER

    # Activate the pointer's index line and the amplitude envelope.
    line.play()
    amp.play()

# Call go() each time the playback reaches the end of the file
pat = Pattern(function=go, time=STUTTER).play()

s.gui(locals())
```

2.10.7 07-moving-points.py - Periodically rewrite a break-point function table.

This example shows how we can dynamically modify a function table to create a curve that varies over time.

Note: Rewriting a large table can produce xruns in the audio output. Usually, for this kind of processes, we want to keep the table size relatively small.

```
from pyo import *

s = Server().boot()

# Initialize an empty table.
table = LinTable([(0,0), (255,0)], size=256)
table.view()

# Two LFOs whose values will change the center points in the table.
lfo1 = Sine(0.1, phase=0.75, mul=0.5, add=0.5)
lfo2 = Sine(0.15, phase=0.75, mul=0.5, add=0.5)

def create_line():
    "Function to create a new line."
    lst = [(0, 0)]                  # First point of the table at value 0.
    lst.append((8, lfo1.get()))     # Second point, from first LFO.
    lst.append((128, lfo2.get()))   # Third point, from second LFO.
    lst.append((255, 0))            # Last point of the table at value 0.

    # Replace the table content with the new list of points.
    table.replace(lst)

# Call the function "create_line" every 50 ms.
pat = Pattern(function=create_line, time=0.05).play()

# Little test case...
amp = Osc(table, freq=4, mul=0.4)
synth = RCOsc(freq=[99.5, 100], sharp=0.3, mul=amp).out()

s.gui(locals())
```

2.10.8 08-table-lookup.py - Table as transfer function.

In computer science, a lookup table is an array that replaces runtime computation with a simpler array indexing operation. The savings in terms of processing time can be significant, since retrieving a value from memory is often faster than undergoing an "expensive" computation.

This technic is also used a lot in audio synthesis, where the waveform can be pre-computed and stored in a table, then simply readed with a ramp at the desired frequency.

```
from pyo import *

s = Server().boot()

src = SfPlayer("../snds/flute.aif", loop=True)

# Arctangent transfer function table.
table = AtanTable(slope=0.5, size=512)
table.view()
```

(continues on next page)

(continued from previous page)

```
# A signal to dynamically control the drive of the transfer function.
drive = Sig(0.5)

# We give it True to the dataonly argument when opening the sliders window,
# otherwise the table would be rewrite way too often.
drive.ctrl([SLMap(0, 1, "lin", "value", 0.5, dataOnly=True)])

# Lookup reads a table given an audio index lying between -1 and 1.
# It is especially designed to scan a transfer function with an audio signal.
look = Lookup(table, index=src, mul=0.5).out()

# Function called to redraw the transfer function.
def redraw():
    table.slope = drive.get()

# We call the "redraw" function every time the "drive" value changes.
trig = TrigFunc(Change(drive), function=redraw)

sc = Scope(look)

s.gui(locals())
```

2.11 18-multicore

2.11.1 01-processes-spawning.py - Simple processes spawning, no synchronization.

Need at least 4 cores to be really effective.

Usage: python3 -i 01-processes-spawning.py

```
import sys, time, random, multiprocessing
from pyo import *

if sys.platform.startswith("linux"):
    audio = "jack"
elif sys.platform.startswith("darwin"):
    audio = "portaudio"
else:
    print("Multicore examples don't run under Windows... Sorry!")
    exit()

class Proc(multiprocessing.Process):
    def __init__(self, pitch):
        super(Proc, self).__init__()
        self.daemon = True
        self.pitch = pitch

    def run(self):
        self.server = Server(audio=audio)
        self.server.deactivateMidi()
        self.server.boot().start()
```

(continues on next page)

(continued from previous page)

```
        # 200 randomized band-limited square wave oscillators.
        self.amp = Fader(fadein=5, mul=0.01).play()
        lo, hi = midiToHz((self.pitch - 0.1, self.pitch + 0.1))
        self.fr = Randi(lo, hi, [random.uniform(.2, .4) for i in range(50)])
        self.sh = Randi(0.1, 0.9, [random.uniform(.2, .4) for i in range(50)])
        self.osc = LFO(self.fr, sharp=self.sh, type=2, mul=self.amp).out()

        time.sleep(30) # Play for 30 seconds.
        self.server.stop()

if __name__ == '__main__':
    # C major chord (one note per process).
    p1, p2, p3, p4 = Proc(48), Proc(52), Proc(55), Proc(60)
    p1.start(); p2.start(); p3.start(); p4.start()
    time.sleep(35)
```

2.11.2 02-sharing-audio.py - Sharing audio signals between processes.

Usage: python3 -i 02-sharing-audio.py

```
import sys, time, random, multiprocessing
from pyo import *

if sys.platform.startswith("linux"):
    audio = "jack"
elif sys.platform.startswith("darwin"):
    audio = "portaudio"
    print("SharedTable does not behave correctly under MacOS... This example doesn't␣
↪work.")
else:
    print("Multicore examples don't run under Windows... Sorry!")
    exit()

class Proc(multiprocessing.Process):
    def __init__(self, create):
        super(Proc, self).__init__()
        self.daemon = True
        self.create = create

    def run(self):
        self.server = Server(audio=audio)
        self.server.deactivateMidi()
        self.server.boot().start()
        bufsize = self.server.getBufferSize()

        nbands = 20
        names = ["/f%02d" % i for i in range(nbands)]

        if self.create: # 50 bands frequency splitter.
            freq = [20 * 1.1487 ** i for i in range(nbands)]
            amp = [pow(10, (i*-1)*0.05) * 8 for i in range(nbands)]
            self.input = SfPlayer(SNDS_PATH+"/transparent.aif", loop=True)
            self.filts = IRWinSinc(self.input, freq, freq, 3, 128, amp)
            self.table = SharedTable(names, create=True, size=bufsize)
            self.recrd = TableFill(self.filts, self.table)
```

(continues on next page)

(continued from previous page)

```
        else: # Independent transposition per band.
            self.table = SharedTable(names, create=False, size=bufsize)
            self.tscan = TableScan(self.table)
            transpofac = [random.uniform(0.98,1.02) for i in range(nbands)]
            self.pvana = PVAnal(self.tscan, size=1024, overlaps=4)
            self.pvtra = PVTranspose(self.pvana, transpo=transpofac)
            self.pvsyn = PVSynth(self.pvtra).out()

        time.sleep(30)
        self.server.stop()

if __name__ == '__main__':
    analysis = Proc(create=True)
    synthesis = Proc(create=False)
    analysis.start()
    synthesis.start()
```

2.11.3 03-synchronization.py - Synchronizing multiple processes.

Usage: python3 -i 03-synchronization.py

```
import sys, time, random, multiprocessing
from pyo import *

RECORD = False

if sys.platform.startswith("linux"):
    audio = "jack"
elif sys.platform.startswith("darwin"):
    audio = "portaudio"
    print("SharedTable does not behave correctly under MacOS... This example doesn't␣
↪work.")
else:
    print("Multicore examples don't run under Windows... Sorry!")
    exit()

class Main(multiprocessing.Process):
    def __init__(self):
        super(Main, self).__init__()
        self.daemon = True

    def run(self):
        self.server = Server(audio=audio)
        self.server.deactivateMidi()
        self.server.boot().start()
        bufsize = self.server.getBufferSize()
        if RECORD:
            self.server.recstart("synchronization.wav")

        self.tab1 = SharedTable("/audio-1", create=False, size=bufsize)
        self.tab2 = SharedTable("/audio-2", create=False, size=bufsize)
        self.out1 = TableScan(self.tab1).out()
        self.out2 = TableScan(self.tab2).out(1)

        time.sleep(30)
```

(continues on next page)

(continued from previous page)

```
        self.server.stop()

class Proc(multiprocessing.Process):
    def __init__(self, voice, conn):
        super(Proc, self).__init__()
        self.voice = voice
        self.connection = conn
        self.daemon = True

    def run(self):
        self.server = Server(audio=audio)
        self.server.deactivateMidi()
        self.server.boot()
        bufsize = self.server.getBufferSize()

        name = "/audio-%d" % self.voice
        self.audiotable = SharedTable(name, create=True, size=bufsize)

        onsets = random.sample([5,6,7,8,9], 2)
        self.tab = CosTable([(0,0), (32,1), (512,0.5), (4096,0.5), (8191,0)])
        self.ryt = Euclide(time=.125, taps=16, onsets=onsets, poly=1).play()
        self.mid = TrigXnoiseMidi(self.ryt, dist=12, mrange=(60, 96))
        self.frq = Snap(self.mid, choice=[0,2,3,5,7,8,10], scale=1)
        self.amp = TrigEnv(self.ryt, table=self.tab, dur=self.ryt['dur'],
                           mul=self.ryt['amp'])
        self.sig = SineLoop(freq=self.frq, feedback=0.08, mul=self.amp*0.3)
        self.fil = TableFill(self.sig, self.audiotable)

        # Wait for an incoming signal before starting the server.
        while not self.connection.poll():
            pass
        self.server.start()

        time.sleep(30)
        self.server.stop()

if __name__ == '__main__':
    signal, child = multiprocessing.Pipe()
    p1, p2 = Proc(1, child), Proc(2, child)
    main = Main()
    p1.start(); p2.start();
    time.sleep(.1)
    main.start()
    time.sleep(.1)
    signal.send(1)
```

2.11.4 04-data-control.py - Multicore midi synthesizer.

Need at least 4 cores to be really effective.

Usage: python3 -i 04-data-control.py

```
import sys, time, multiprocessing
from random import uniform
from pyo import *
```

(continues on next page)

(continued from previous page)

```
VOICES_PER_CORE = 4

if sys.platform.startswith("linux"):
    audio = "jack"
elif sys.platform.startswith("darwin"):
    audio = "portaudio"
else:
    print("Multicore examples don't run under Windows... Sorry!")
    exit()

class Proc(multiprocessing.Process):
    def __init__(self, pipe):
        super(Proc, self).__init__()
        self.daemon = True
        self.pipe = pipe

    def run(self):
        self.server = Server(audio=audio)
        self.server.deactivateMidi()
        self.server.boot().start()

        self.mid = Notein(poly=VOICES_PER_CORE, scale=1, first=0, last=127)
        self.amp = MidiAdsr(self.mid['velocity'], 0.005, .1, .7, 0.5, mul=.01)
        self.pit = self.mid['pitch'] * [uniform(.99, 1.01) for i in range(40)]
        self.rc1 = RCOsc(self.pit, sharp=0.8, mul=self.amp).mix(1)
        self.rc2 = RCOsc(self.pit*0.99, sharp=0.8, mul=self.amp).mix(1)
        self.mix = Mix([self.rc1, self.rc2], voices=2)
        self.rev = STRev(Denorm(self.mix), [.1, .9], 2, bal=0.30).out()

        while True:
            if self.pipe.poll():
                data = self.pipe.recv()
                self.server.addMidiEvent(*data)
            time.sleep(0.001)

        self.server.stop()

if __name__ == '__main__':
    main1, child1 = multiprocessing.Pipe()
    main2, child2 = multiprocessing.Pipe()
    main3, child3 = multiprocessing.Pipe()
    main4, child4 = multiprocessing.Pipe()
    mains = [main1, main2, main3, main4]
    p1, p2, p3, p4 = Proc(child1), Proc(child2), Proc(child3), Proc(child4)
    p1.start(); p2.start(); p3.start(); p4.start()

    playing = {0: [], 1: [], 2: [], 3: []}
    currentcore = 0
    def callback(status, data1, data2):
        global currentcore
        if status == 0x80 or status == 0x90 and data2 == 0:
            for i in range(4):
                if data1 in playing[i]:
                    playing[i].remove(data1)
                    mains[i].send([status, data1, data2])
                    break
```

(continues on next page)

(continued from previous page)

```
        elif status == 0x90:
            for i in range(4):
                currentcore = (currentcore + 1) % 4
                if len(playing[currentcore]) < VOICES_PER_CORE:
                    playing[currentcore].append(data1)
                    mains[currentcore].send([status, data1, data2])
                    break

    s = Server()
    s.setMidiInputDevice(99) # Open all devices.
    s.boot().start()
    raw = RawMidi(callback)
```

2.12 19-multirate

2.12.1 01-multi-rate-processing.py - Doing processing at very high sampling rate.

In numerical audio computing, it is sometimes useful to be able to process a signal with much more timing precision than what the usual sampling rates offer. A typical case is when the effect applied to the sound adds a lot of harmonics. Higher harmonics will quickly wrap around the Nyquist frequency, producing aliasing in the output signal. The solution is to increase the sampling rate, so the nyquist frequency, and to use anti-aliasing filters when converting from one rate to another.

Pyo allows to compute chunks of code at different sampling rates than the one with which the server was started. You should do this only for the objects you need to process with a higher sampling rate, without changing the server's sampling rate, otherwise the program will be very CPU consuming.

You start a new resampling block with the method:

Server.beginResamplingBlock(x)

where *x*, a power-of-two, is the resampling factor. A negative power-of-two will start a downsampling block of code.

To close the block, simply call:

Server.endResamplingBlock()

Everything between the two calls will be computed with the new sampling rate.

Audio signals must be resampled before used with a different sampling rate. The Resample object does this. Inside a resampling block, it will convert the signal to the new sampling rate, and outside the resampling block, it will convert the signal back to the original sampling rate. Its *mode* argument lets choose the quality of the interpolation/decimation filter used to resample the signal.

```
from pyo import *

# We create a new class for our upsampled distortion effect.
class UpSampDisto:
    """
    Upsampled distortion effect.

    :Args:

        input: PyoObject
            The signal to process.
        drive: float or PyoObject, optional
```

(continues on next page)

(continued from previous page)

```
            Amount of distortion applied to the signal, between 0 and 1.
        upfactor: int (power-of-two), optional
            Resampling factor.
        filtmode: int, optional
            The interpolation/decimation mode. See Resample's man page
            for details.

    """
    def __init__(self, input, drive=0.5, upfactor=8, filtmode=32):
        # The InputFader object lets change its input signal without clicks.
        self.input = InputFader(input)

        # Convert the drive argument to audio signal.
        self.drive = Sig(drive)

        # Get a reference to the audio server.
        server = self.drive.getServer()

        # Start an upsampled block of code.
        server.beginResamplingBlock(upfactor)

        # Resample the audio signals. Because the drive signal is only a
        # control signal, a linear interpolation is enough. The input
        # signal uses a much better filter to eliminate aliasing artifacts.
        self.inputUp = Resample(self.input, mode=filtmode)
        self.driveUp = Resample(self.drive, mode=1)

        # Apply the distortion effect.
        self.disto = Disto(self.inputUp, drive=self.driveUp)

        # Close the upsampled block.
        server.endResamplingBlock()

        # Convert back the distorted signal to the current sampling rate.
        # Again, we use a good decimination filter to eliminate aliasing.
        self.output = Resample(self.disto, mode=filtmode, mul=0.5)

    # Define some useful methods.
    def setInput(self, x, fadetime=0.05):
        self.input.setInput(x, fadetime)

    def setDrive(self, x):
        self.drive.value = x

    def out(self, chnl=0):
        self.output.out(chnl)
        return self

    def sig(self):
        return self.output

### Usage example ###
s = Server().boot()

# Two different sources for testing, a sine wave and a flute melody.
src1 = Sine(freq=722, mul=0.7)
src1.ctrl([SLMapFreq(722)], title="Sine frequency")
```

(continues on next page)

(continued from previous page)

```
src2 = SfPlayer("../snds/flute.aif", loop=True)
# Input source interpolation.
src = Interp(src1, src2, 0)
src.ctrl([SLMap(0, 1, "lin", "interp", 0)],
         title="Source: sine <=> flute")

# Control for the drive parameter of the distortion.
drv = Sig(0)
drv.ctrl(title="Drive")

# Distortion at current sampling rate.
dist = Disto(src, drive=drv, mul=0.5)

# Distortion with increased sampling rate.
updist = UpSampDisto(src, drv)

# Interpolator to compare the two processes.
output = Interp(dist, updist.sig(), 0, mul=0.5).out()
output.ctrl([SLMap(0, 1, "lin", "interp", 0)],
            title="Up Sampling: without <=> with")

sp = Spectrum(output)

s.gui(locals())
```

2.12.2 01-multi-rate-synthesis.py - Doing synthesis at very high sampling rate.

In numerical audio computing, it is sometimes useful to be able to process a signal with much more timing precision than what the usual sampling rates offer. A typical case is when the synthesis algorithm generates aliasing in the output signal. The solution is to increase the sampling rate, so the nyquist frequency, and to use anti-aliasing filters when converting from one rate to another.

Pyo allows to compute chunks of code at different sampling rates than the one with which the server was started. You should do this only for the objects you need to process with a higher sampling rate, without changing the server's sampling rate, otherwise the program will be very CPU consuming.

You start a new resampling block with the method:

Server.beginResamplingBlock(x)

where *x*, a power-of-two, is the resampling factor. A negative power-of-two will start a downsampling block of code.

To close the block, simply call:

Server.endResamplingBlock()

Everything between the two calls will be computed with the new sampling rate.

Audio signals must be resampled before used with a different sampling rate. The Resample object does this. Inside a resampling block, it will convert the signal to the new sampling rate, and outside the resampling block, it will convert the signal back to the original sampling rate. Its *mode* argument lets choose the quality of the interpolation/decimation filter used to resample the signal.

```
from pyo import *

s = Server().boot()
```

(continues on next page)

(continued from previous page)

```
# We create a new class for our upsampled Frequency modulation synthesis. We
# use only the modulation index as parameter in order to simplify the code.
class UpSampFM:
    """
    Frequency modulation synthesis that can be computed a higher sampling rate.

    :Args:

        fmindex: float or PyoObject, optional
            The modulation index of the FM synthesis.
        upfactor: int (power-of-two), optional
            Resampling factor.
        filtmode: int, optional
            The interpolation/decimation mode. See Resample's man page
            for details.

    """
    def __init__(self, fmindex=20, upfactor=8, filtmode=32):

        # Convert the modulation index argument to audio signal.
        self.fmindex = Sig(fmindex)

        # Get a reference to the audio server.
        server = self.fmindex.getServer()

        # Start an upsampled block of code.
        server.beginResamplingBlock(upfactor)

        # Resample the audio signals. Because the drive signal is only a
        # control signal, a linear interpolation is enough.
        self.fmindexUp = Resample(self.fmindex, mode=1)

        # Generate the FM synthesis.
        self.fmUp = FM(carrier=492, ratio=2, index=self.fmindexUp)

        # Close the upsampled block.
        server.endResamplingBlock()

        # Convert back the synthesized signal to the current sampling rate.
        # We use a good decimation filter to eliminate aliasing.
        self.output = Resample(self.fmUp, mode=filtmode)

    # Define some useful methods.
    def out(self):
        self.output.out()
        return self

    def sig(self):
        return self.output

# Control for the modulation index parameter of the synthesis.
index = Sig(20)
index.ctrl([SLMap(5, 50, "lin", "value", 20)],
           title="Modulation Index")

# FM synthesis at current sampling rate.
fm1 = FM(carrier=492, ratio=2, index=index)
```

(continues on next page)

(continued from previous page)

```
# FM synthesis with increased sampling rate.
fm2 = UpSampFM(index)

# Interpolator to compare the two processes.
output = Interp(fm1, fm2.sig(), 0, mul=0.5).out()
output.ctrl([SLMap(0, 1, "lin", "interp", 0)],
            title="Up Sampling: without <=> with")

sp = Spectrum(output)

s.gui(locals())
```

Much more to come… Stay tuned!

CHAPTER

THREE

ADVANCED TUTORIALS

3.1 Tutorial about creating a custom PyoObject (RingMod)

There are few steps we need to take care of in order to create a class with all of the PyoObject behaviors.

Things to consider:

- The parent class must be PyoObject, that means the PyoObject's __init__ method must be called inside the object's __init__ method to properly initialize PyoObject's basic attributes.
- When a PyoObject receives another PyoObject, it looks for a list of objects called "self._base_objs". This list must contain the C implementation of the audio objects generating the output sound of the process.
- Adding "mul" and "add" arguments (they act on objects in self._base_objs).
- All PyoObjects support "list expansion".
- All PyoObjects with sound in input support cross-fading between old and new sources.
- We will probably want to override the play(), out() and stop() methods.
- There is an attribute for any function that modify a parameter.
- We should override the ctrl() method to allow a GUI to control parameters.

In this tutorial, we will define a RingMod object with this definition:

```
RingMod(input, freq=100, mul=1, add=0)
```

First of all, we need to import the pyo module

```
from pyo import *
```

3.1.1 Step 1 - Declaring the class

We will create a new class called RingMod with PyoObject as its parent class. Another good habit is to put a __doc__ string at the beginning of our classes. Doing so will allow other users to retrieve the object's documentation with the standard python help() function.

```
class RingMod(PyoObject):
    """
    Ring modulator.

    Ring modulation is a signal-processing effect in electronics
    performed by multiplying two signals, where one is typically
```

(continues on next page)

(continued from previous page)

```
    a sine-wave or another simple waveform.

    :Parent: :py:class:`PyoObject`

    :Args:

        input : PyoObject
            Input signal to process.
        freq : float or PyoObject, optional
            Frequency, in cycles per second, of the modulator.
            Defaults to 100.

    >>> s = Server().boot()
    >>> s.start()
    >>> src = SfPlayer(SNDS_PATH+"/transparent.aif", loop=True, mul=.3)
    >>> lfo = Sine(.25, phase=[0,.5], mul=.5, add=.5)
    >>> ring = RingMod(src, freq=[800,1000], mul=lfo).out()

    """
```

3.1.2 Step 2 - The __init__ method

This is the place where we have to take care of some of pyo's generic behaviours. The most important thing to remember is that when a PyoObject receives another PyoObject in input, it looks for an attribute called self._base_objs. This attribute is a list of the object's base classes and is considered the audio output signal of the object (the Sine object uses internally an object called Sine_base). The getBaseObjects() method returns the list of base classes for a given PyoObject. We will call the getBaseObjects() method on the objects generating the output signal of our process. .play(), .out(), .stop() and .mix() methods act on this list.

We also need to add two arguments to the definition of the object: "mul" and "add". The attributes "self._mul" and "self._add" are handled by the parent class and are automatically applied to the objects stored in the list "self._base_objs".

Finally, we have to consider the "multi-channel expansion" feature, allowing lists given as arguments to create multiple instances of our object and managing multiple audio streams. Two functions help us to accomplish this:

convertArgsToLists(*args) : Return arguments converted to lists and the maximum list size. wrap(list,i) : Return value at position "i" in "list" with wrap around len(list).

```
def __init__(self, input, freq=100, mul=1, add=0):
    # Properly initialize PyoObject's basic attributes
    PyoObject.__init__(self, mul, add)

    # Keep references of all raw arguments
    self._input = input
    self._freq = freq

    # Using InputFader to manage input sound allows cross-fade when changing sources
    self._in_fader = InputFader(input)

    # Convert all arguments to lists for "multi-channel expansion"
    in_fader,freq,mul,add,lmax = convertArgsToLists(self._in_fader,freq,mul,add)

    # Apply processing
    self._mod = Sine(freq=freq, mul=in_fader)
```

(continues on next page)

(continued from previous page)

```
        # Use Sig object as a through to prevent modifying "mul" attribute of self._mod
        self._ring = Sig(self._mod, mul=mul, add=add)

        # self._base_objs is the audio output seen by the outside world!
        self._base_objs = self._ring.getBaseObjects()
```

3.1.3 Step 3 - setXXX methods and attributes

Now, we will add methods and attributes getter and setter for all controllable parameters. It should be noted that we use the setInput() method of the InputFader object to change an input source. This object implements a cross-fade between the old source and the new one with a cross-fade duration argument. Here, we need to keep references of raw argument in order to get the real current state when we call the dump() method.

```
def setInput(self, x, fadetime=0.05):
    """
    Replace the `input` attribute.

    :Args:

        x : PyoObject
            New signal to process.
        fadetime : float, optional
            Crossfade time between old and new input. Defaults to 0.05.

    """
    self._input = x
    self._in_fader.setInput(x, fadetime)

def setFreq(self, x):
    """
    Replace the `freq` attribute.

    :Args:

        x : float or PyoObject
            New `freq` attribute.

    """
    self._freq = x
    self._mod.freq = x

@property # getter
def input(self):
    """PyoObject. Input signal to process."""
    return self._input
@input.setter # setter
def input(self, x):
    self.setInput(x)

@property
def freq(self):
    """float or PyoObject. Frequency of the modulator."""
    return self._freq
@freq.setter
```

(continues on next page)

(continued from previous page)

```
def freq(self, x):
    self.setFreq(x)
```

3.1.4 Step 4 - The ctrl() method

The ctrl() method of a PyoObject is used to pop-up a GUI to control the parameters of the object. The initialization of sliders is done with a list of SLMap objects where we can set the range of the slider, the type of scaling, the name of the attribute linked to the slider and the initial value. We will define a default "self._map_list" that will be used if the user doesn't provide one to the parameter "map_list". If the object doesn't have any parameter to control with a GUI, this

```
def ctrl(self, map_list=None, title=None, wxnoserver=False):
    self._map_list = [SLMap(10, 2000, "log", "freq", self._freq),
                      SLMapMul(self._mul)]
    PyoObject.ctrl(self, map_list, title, wxnoserver)
```

3.1.5 Step 5 - Overriding the .play(), .stop() and .out() methods

Finally, we might want to override .play(), .stop() and .out() methods to be sure all our internal PyoObjects are consequently managed instead of only objects in self._base_obj, as it is in built-in objects. To handle properly the process for self._base_objs, we still need to call the method that belongs to PyoObject. We return the returned value (self) of these methods in order to possibly append the method to the object's creation. See the definition of these methods in the PyoObject man page to understand the meaning of arguments.

```
def play(self, dur=0, delay=0):
    self._mod.play(dur, delay)
    return PyoObject.play(self, dur, delay)

def stop(self, wait=0):
    self._mod.stop(wait)
    return PyoObject.stop(self, wait)

def out(self, chnl=0, inc=1, dur=0, delay=0):
    self._mod.play(dur, delay)
    return PyoObject.out(self, chnl, inc, dur, delay)
```

Here we are, we've just created our first custom pyo object!

3.1.6 Complete class definition and test

```
from pyo import *

class RingMod(PyoObject):
    """
    Ring modulator.

    Ring modulation is a signal-processing effect in electronics
    performed by multiplying two signals, where one is typically
    a sine-wave or another simple waveform.

    :Parent: :py:class:`PyoObject`
```

(continues on next page)

(continued from previous page)

```
    :Args:

        input : PyoObject
            Input signal to process.
        freq : float or PyoObject, optional
            Frequency, in cycles per second, of the modulator.
            Defaults to 100.

    >>> s = Server().boot()
    >>> s.start()
    >>> src = SfPlayer(SNDS_PATH+"/transparent.aif", loop=True, mul=.3)
    >>> lfo = Sine(.25, phase=[0,.5], mul=.5, add=.5)
    >>> ring = RingMod(src, freq=[800,1000], mul=lfo).out()

    """
    def __init__(self, input, freq=100, mul=1, add=0):
        PyoObject.__init__(self, mul, add)
        self._input = input
        self._freq = freq
        self._in_fader = InputFader(input)
        in_fader,freq,mul,add,lmax = convertArgsToLists(self._in_fader,freq,mul,add)
        self._mod = Sine(freq=freq, mul=in_fader)
        self._ring = Sig(self._mod, mul=mul, add=add)
        self._base_objs = self._ring.getBaseObjects()

    def setInput(self, x, fadetime=0.05):
        """
        Replace the `input` attribute.

        :Args:

            x : PyoObject
                New signal to process.
            fadetime : float, optional
                Crossfade time between old and new input. Defaults to 0.05.

        """
        self._input = x
        self._in_fader.setInput(x, fadetime)

    def setFreq(self, x):
        """
        Replace the `freq` attribute.

        :Args:

            x : float or PyoObject
                New `freq` attribute.

        """
        self._freq = x
        self._mod.freq = x

    def play(self, dur=0, delay=0):
        self._mod.play(dur, delay)
        return PyoObject.play(self, dur, delay)
```

(continues on next page)

(continued from previous page)

```
    def stop(self, wait=0):
        self._mod.stop(wait)
        return PyoObject.stop(self, wait)

    def out(self, chnl=0, inc=1, dur=0, delay=0):
        self._mod.play(dur, delay)
        return PyoObject.out(self, chnl, inc, dur, delay)

    def ctrl(self, map_list=None, title=None, wxnoserver=False):
        self._map_list = [SLMap(10, 2000, "log", "freq", self._freq),
                          SLMapMul(self._mul)]
        PyoObject.ctrl(self, map_list, title, wxnoserver)

    @property # getter
    def input(self):
        """PyoObject. Input signal to process."""
        return self._input
    @input.setter # setter
    def input(self, x):
        self.setInput(x)

    @property
    def freq(self):
        """float or PyoObject. Frequency of the modulator."""
        return self._freq
    @freq.setter
    def freq(self, x):
        self.setFreq(x)

# Run the script to test the RingMod object.
if __name__ == "__main__":
    s = Server().boot()
    src = SfPlayer(SNDS_PATH+"/transparent.aif", loop=True, mul=.3)
    lfo = Sine(.25, phase=[0,.5], mul=.5, add=.5)
    ring = RingMod(src, freq=[800,1000], mul=lfo).out()
    s.gui(locals())
```

3.2 Another tutorial about creating a custom PyoObject (Flanger)

There are few steps we need to take care of in order to create a class with all of the PyoObject behaviors.

Things to consider:

- The parent class must be PyoObject, that means the PyoObject's __init__ method must be called inside the object's __init__ method to properly initialize PyoObject's basic attributes.
- When a PyoObject receives another PyoObject, it looks for a list of objects called "self._base_objs". This list must contain the C implementation of the audio objects generating the output sound of the process.
- Adding "mul" and "add" arguments (they act on objects in self._base_objs).
- All PyoObjects support "list expansion".
- All PyoObjects with sound in input support cross-fading between old and new sources.
- We will probably want to override the play(), out() and stop() methods.

- There is an attribute for any function that modify a parameter.
- We should override the ctrl() method to allow a GUI to control parameters.

In this tutorial, we will define a Flanger object with this definition:

```
Flanger(input, depth=0.75, lfofreq=0.2, feedback=0.25, mul=1, add=0)
```

First of all, we need to import the pyo module

```
from pyo import *
```

3.2.1 Step 1 - Declaring the class

We will create a new class called Flanger with PyoObject as its parent class. Another good habit is to put a __doc__ string at the beginning of our classes. Doing so will allow other users to retrieve the object's documentation with the standard python help() function.

```
class Flanger(PyoObject):
    """
    Flanging effect.

    A flanging is an audio effect produced by mixing two identical signals together,
    with one signal delayed by a small and gradually changing period, usually smaller
    than 20 milliseconds. This produces a swept comb filter effect: peaks and notches
    are produced in the resultant frequency spectrum, related to each other in a␣
↪linear
    harmonic series. Varying the time delay causes these to sweep up and down the
    frequency spectrum.


    :Parent: :py:class:`PyoObject`

    :Args:

        input : PyoObject
            Input signal to process.
        depth : float or PyoObject, optional
            Amplitude of the delay line modulation, between 0 and 1.
            Defaults to 0.75.
        lfofreq : float or PyoObject, optional
            Frequency of the delay line modulation, in Hertz.
            Defaults to 0.2.
        feedback : float or PyoObject, optional
            Amount of output signal reinjected into the delay line.
            Defaults to 0.25.

    >>> s = Server().boot()
    >>> s.start()
    >>> inp = SfPlayer(SNDS_PATH + "/transparent.aif", loop=True)
    >>> lf = Sine(0.005, mul=0.25, add=0.5)
    >>> flg = Flanger(input=inp, depth=0.9, lfofreq=0.1, feedback=lf).out()

    """
```

3.2.2 Step 2 - The __init__ method

This is the place where we have to take care of some of pyo's generic behaviours. The most important thing to remember is that when a PyoObject receives another PyoObject in input, it looks for an attribute called self._base_objs. This attribute is a list of the object's base classes and is considered the audio output signal of the object (the Sine object uses internally an object called Sine_base). The getBaseObjects() method returns the list of base classes for a given PyoObject. We will call the getBaseObjects() method on the objects generating the output signal of our process. .play(), .out(), .stop() and .mix() methods act on this list.

We also need to add two arguments to the definition of the object: "mul" and "add". The attributes "self._mul" and "self._add" are handled by the parent class and are automatically applied to the objects stored in the list "self._base_objs".

Finally, we have to consider the "multi-channel expansion" feature, allowing lists given as arguments to create multiple instances of our object and managing multiple audio streams. Two functions help us to accomplish this:

convertArgsToLists(*args) : Return arguments converted to lists and the maximum list size. wrap(list,i) : Return value at position "i" in "list" with wrap around len(list).

```
def __init__(self, input, depth=0.75, lfofreq=0.2, feedback=0.5, mul=1, add=0):
    # Properly initialize PyoObject's basic attributes
    PyoObject.__init__(self)

    # Keep references of all raw arguments
    self._input = input
    self._depth = depth
    self._lfofreq = lfofreq
    self._feedback = feedback

    # Using InputFader to manage input sound allows cross-fade when changing sources
    self._in_fader = InputFader(input)

    # Convert all arguments to lists for "multi-channel expansion"
    in_fader, depth, lfofreq, feedback, mul, add, lmax = convertArgsToLists(
                            self._in_fader, depth, lfofreq, feedback, mul, add)

    # Apply processing
    self._modamp = Sig(depth, mul=0.005)
    self._mod = Sine(freq=lfofreq, mul=self._modamp, add=0.005)
    self._dls = Delay(in_fader, delay=self._mod, feedback=feedback)
    self._flange = Interp(in_fader, self._dls, mul=mul, add=add)

    # self._base_objs is the audio output seen by the outside world!
    self._base_objs = self._flange.getBaseObjects()
```

3.2.3 Step 3 - setXXX methods and attributes

Now, we will add methods and attributes getter and setter for all controllable parameters. It should be noted that we use the setInput() method of the InputFader object to change an input source. This object implements a cross-fade between the old source and the new one with a cross-fade duration argument. Here, we need to keep references of raw argument in order to get the real current state when we call the dump() method.

```
def setInput(self, x, fadetime=0.05):
    """
    Replace the `input` attribute.

```

(continues on next page)

(continued from previous page)

```
    :Args:

        x : PyoObject
            New signal to process.
        fadetime : float, optional
            Crossfade time between old and new input. Defaults to 0.05.

    """
    self._input = x
    self._in_fader.setInput(x, fadetime)

def setDepth(self, x):
    """
    Replace the `depth` attribute.

    :Args:

        x : float or PyoObject
            New `depth` attribute.

    """
    self._depth = x
    self._modamp.value = x

def setLfoFreq(self, x):
    """
    Replace the `lfofreq` attribute.

    :Args:

        x : float or PyoObject
            New `lfofreq` attribute.

    """
    self._lfofreq = x
    self._mod.freq = x

def setFeedback(self, x):
    """
    Replace the `feedback` attribute.

    :Args:

        x : float or PyoObject
            New `feedback` attribute.

    """
    self._feedback = x
    self._dls.feedback = x

@property
def input(self):
    """PyoObject. Input signal to process."""
    return self._input
@input.setter
def input(self, x):
    self.setInput(x)
```

(continues on next page)

(continued from previous page)

```
@property
def depth(self):
    """float or PyoObject. Amplitude of the delay line modulation."""
    return self._depth
@depth.setter
def depth(self, x):
    self.setDepth(x)

@property
def lfofreq(self):
    """float or PyoObject. Frequency of the delay line modulation."""
    return self._lfofreq
@lfofreq.setter
def lfofreq(self, x):
    self.setLfoFreq(x)

@property
def feedback(self):
    """float or PyoObject. Amount of out sig sent back in delay line."""
    return self._feedback
@feedback.setter
def feedback(self, x):
    self.setFeedback(x)
```

3.2.4 Step 4 - The ctrl() method

The ctrl() method of a PyoObject is used to pop-up a GUI to control the parameters of the object. The initialization of sliders is done with a list of SLMap objects where we can set the range of the slider, the type of scaling, the name of the attribute linked to the slider and the initial value. We will define a default "self._map_list" that will be used if the user doesn't provide one to the parameter "map_list". If the object doesn't have any parameter to control with a GUI, this

```
def ctrl(self, map_list=None, title=None, wxnoserver=False):
    self._map_list = [SLMap(0., 1., "lin", "depth", self._depth),
                      SLMap(0.001, 20., "log", "lfofreq", self._lfofreq),
                      SLMap(0., 1., "lin", "feedback", self._feedback),
                      SLMapMul(self._mul)]
    PyoObject.ctrl(self, map_list, title, wxnoserver)
```

3.2.5 Step 5 - Overriding the .play(), .stop() and .out() methods

Finally, we might want to override .play(), .stop() and .out() methods to be sure all our internal PyoObjects are consequently managed instead of only objects in self._base_obj, as it is in built-in objects. To handle properly the process for self._base_objs, we still need to call the method that belongs to PyoObject. We return the returned value (self) of these methods in order to possibly append the method to the object's creation. See the definition of these methods in the PyoObject man page to understand the meaning of arguments.

```
def play(self, dur=0, delay=0):
    self._modamp.play(dur, delay)
    self._mod.play(dur, delay)
    self._dls.play(dur, delay)
```

(continues on next page)

(continued from previous page)

```
    return PyoObject.play(self, dur, delay)

def stop(self, wait=0):
    self._modamp.stop(wait)
    self._mod.stop(wait)
    self._dls.stop(wait)
    return PyoObject.stop(self, wait)

def out(self, chnl=0, inc=1, dur=0, delay=0):
    self._modamp.play(dur, delay)
    self._mod.play(dur, delay)
    self._dls.play(dur, delay)
    return PyoObject.out(self, chnl, inc, dur, delay)
```

Here we are, we've just created our second custom pyo object!

3.2.6 Complete class definition and test

```
from pyo import *

class Flanger(PyoObject):
    """
    Flanging effect.

    A flanging is an audio effect produced by mixing two identical signals together,
    with one signal delayed by a small and gradually changing period, usually smaller
    than 20 milliseconds. This produces a swept comb filter effect: peaks and notches
    are produced in the resultant frequency spectrum, related to each other in a␣
↪linear
    harmonic series. Varying the time delay causes these to sweep up and down the
    frequency spectrum.


    :Parent: :py:class:`PyoObject`

    :Args:

        input : PyoObject
            Input signal to process.
        depth : float or PyoObject, optional
            Amplitude of the delay line modulation, between 0 and 1.
            Defaults to 0.75.
        lfofreq : float or PyoObject, optional
            Frequency of the delay line modulation, in Hertz.
            Defaults to 0.2.
        feedback : float or PyoObject, optional
            Amount of output signal reinjected into the delay line.
            Defaults to 0.25.

    >>> s = Server().boot()
    >>> s.start()
    >>> inp = SfPlayer(SNDS_PATH + "/transparent.aif", loop=True)
    >>> lf = Sine(0.005, mul=0.25, add=0.5)
    >>> flg = Flanger(input=inp, depth=0.9, lfofreq=0.1, feedback=lf).out()
```

(continues on next page)

(continued from previous page)

```
    """
    def __init__(self, input, depth=0.75, lfofreq=0.2, feedback=0.5, mul=1, add=0):
        PyoObject.__init__(self)
        self._input = input
        self._depth = depth
        self._lfofreq = lfofreq
        self._feedback = feedback
        self._in_fader = InputFader(input)
        in_fader, depth, lfofreq, feedback, mul, add, lmax = convertArgsToLists(
                                self._in_fader, depth, lfofreq, feedback, mul, add)

        self._modamp = Sig(depth, mul=0.005)
        self._mod = Sine(freq=lfofreq, mul=self._modamp, add=0.005)
        self._dls = Delay(in_fader, delay=self._mod, feedback=feedback)
        self._flange = Interp(in_fader, self._dls, mul=mul, add=add)

        self._base_objs = self._flange.getBaseObjects()

    def setInput(self, x, fadetime=0.05):
        """
        Replace the `input` attribute.

        :Args:

            x : PyoObject
                New signal to process.
            fadetime : float, optional
                Crossfade time between old and new input. Defaults to 0.05.

        """
        self._input = x
        self._in_fader.setInput(x, fadetime)

    def setDepth(self, x):
        """
        Replace the `depth` attribute.

        :Args:

            x : float or PyoObject
                New `depth` attribute.

        """
        self._depth = x
        self._modamp.value = x

    def setLfoFreq(self, x):
        """
        Replace the `lfofreq` attribute.

        :Args:

            x : float or PyoObject
                New `lfofreq` attribute.

        """
        self._lfofreq = x
```

(continues on next page)

(continued from previous page)

```
        self._mod.freq = x

    def setFeedback(self, x):
        """
        Replace the `feedback` attribute.

        :Args:

            x : float or PyoObject
                New `feedback` attribute.

        """
        self._feedback = x
        self._dls.feedback = x

    def play(self, dur=0, delay=0):
        self._modamp.play(dur, delay)
        self._mod.play(dur, delay)
        self._dls.play(dur, delay)
        return PyoObject.play(self, dur, delay)

    def stop(self, wait=0):
        self._modamp.stop(wait)
        self._mod.stop(wait)
        self._dls.stop(wait)
        return PyoObject.stop(self, wait)

    def out(self, chnl=0, inc=1, dur=0, delay=0):
        self._modamp.play(dur, delay)
        self._mod.play(dur, delay)
        self._dls.play(dur, delay)
        return PyoObject.out(self, chnl, inc, dur, delay)

    def ctrl(self, map_list=None, title=None, wxnoserver=False):
        self._map_list = [SLMap(0., 1., "lin", "depth", self._depth),
                          SLMap(0.001, 20., "log", "lfofreq", self._lfofreq),
                          SLMap(0., 1., "lin", "feedback", self._feedback),
                          SLMapMul(self._mul)]
        PyoObject.ctrl(self, map_list, title, wxnoserver)

    @property
    def input(self):
        """PyoObject. Input signal to process."""
        return self._input
    @input.setter
    def input(self, x):
        self.setInput(x)

    @property
    def depth(self):
        """float or PyoObject. Amplitude of the delay line modulation."""
        return self._depth
    @depth.setter
    def depth(self, x):
        self.setDepth(x)

    @property
```

(continues on next page)

(continued from previous page)

```
    def lfofreq(self):
        """float or PyoObject. Frequency of the delay line modulation."""
        return self._lfofreq
    @lfofreq.setter
    def lfofreq(self, x):
        self.setLfoFreq(x)

    @property
    def feedback(self):
        """float or PyoObject. Amount of out sig sent back in delay line."""
        return self._feedback
    @feedback.setter
    def feedback(self, x):
        self.setFeedback(x)

# Run the script to test the Flanger object.
if __name__ == "__main__":
    s = Server().boot()
    src = BrownNoise([.2,.2]).out()
    fl = Flanger(src, depth=.9, lfofreq=.1, feedback=.5, mul=.5).out()
    s.gui(locals())
```

3.3 Tutorial about creating a custom PyoTableObject (TriTable)

```
from pyo import *

class TriTable(PyoTableObject):
    """
    Square waveform generator.

    Generates square waveforms made up of fixed number of harmonics.

    :Parent: :py:class:`PyoTableObject`

    :Args:

        order : int, optional
            Number of harmonics square waveform is made of. The waveform will
            contains `order` odd harmonics. Defaults to 10.
        size : int, optional
            Table size in samples. Defaults to 8192.

    >>> s = Server().boot()
    >>> s.start()
    >>> t = TriTable(order=15).normalize()
    >>> a = Osc(table=t, freq=[199,200], mul=.2).out()

    """
    def __init__(self, order=10, size=8192):
        PyoTableObject.__init__(self, size)
        self._order = order
        self._tri_table = HarmTable(self._create_list(order), size)
        self._base_objs = self._tri_table.getBaseObjects()
```

(continues on next page)

(continued from previous page)

```
    def _create_list(self, order):
        # internal method used to compute the harmonics's weight
        l = []
        ph = 1.0
        for i in range(1,order*2):
            if i % 2 == 0:
                l.append(0)
            else:
                l.append(ph / (i*i))
                ph *= -1
        return l

    def setOrder(self, x):
        """
        Change the `order` attribute and redraw the waveform.

        :Args:

            x : int
                New number of harmonics

        """
        self._order = x
        self._tri_table.replace(self._create_list(x))
        self.normalize()
        self.refreshView()

    @property
    def order(self):
        """int. Number of harmonics triangular waveform is made of."""
        return self._order
    @order.setter
    def order(self, x): self.setOrder(x)

# Run the script to test the TriTable object.
if __name__ == "__main__":
    s = Server().boot()
    t = TriTable(10, 8192)
    t.normalize()
    t.view()
    a = Osc(t, 500, mul=.3).out()
    s.gui(locals())
```

CHAPTER

FOUR

INDICES AND TABLES

- genindex
- search