OMDE/pmask

Maurizio Umberto Puxeddu

Edited by
Someone Else

OMDE/pmask

by Maurizio Umberto Puxeddu

Edited by Someone Else

Table of Contents

Chapter 1. Introduction

What's OMDE/pmask?

`pmask` is a set of `Python` (www.python.org[1]) modules for numeric score generation. Its first releases had basically the same set of algorithm present in `Cmask` by Andre Bartezky.

The main difference between `Cmask` and `pmask` is that `Cmask` is a dedicated language while `pmask` is extension to a generic programming language.

A dedicated language like `Cmask` is quite simple to learn but not very flexyble. You can't much more than the Andre programmed it for.

Switching to a generic programming language is a big step that has both pros and cons. The main con is that you have to learn Python in order to use `OMDE/pmask`.

Portability. Extendibility.

I think that writing in `Python` (using `pmask`) the equivalent of a `Cmask` program is not more complicated that the original, it is only that you have to type more.

Flexibility means that:

- you can add more basic components (generators)
- you can mix `Cmask`-like generation algorithms with other totally different score generation or processing algorithms
- you can apply the same tool to similar kind of problems (3d graphics and probably 3d graphic animations)

In successive releases `pmask` in fact added more generators (not present in `Cmask`) and a additional step, very simple (like everything in `OMDE/pmask`) but very important: events were no more stored as strings but in dedicated `Python` objects.

Of course you can print one of such object and get the standard numeric score notation for event, but you can also operate easily on that object in order to post-process them, aggregate it with other objects and operate on such groups of objects.

At this point I wanted to make a distinction between the way you produce the score events (`pmask`) and the way you represent and process them (`OMDE`). This is way I refer to the current development version of "`pmask`" as `OMDE/pmask`.

For example: the `Event` and `Aggregate` classes are in `omde.score`, the `cloud()` or `sequence()` algorithm are in `pmask.algorithm` (or in `pmask`) and generator and modifiers (the difference in `pmask` is less important) are in places like `pmask.rng` (pseudo random number generators), `pmask.chaos` (non linear generators), `pmask.miscellaneous` (`List`, `Mask`, `Quantizer`, `Attractor`, `Choice` etc). Someone else could use something completely different to produce `OMDE` objects than `pmask` and the continue the processing using `OMDE` or other `OMDE`-enabled tools. `OMDE` has also modules for pitch (`omde.pitch`, integrating the `SCALA` database of scales) and rhthm (`omde.rhythm`), both of which have room for being greatly extended and improved.

The CVS version `OMDE/pmask` includes contributions by prof. Zipfel in the `omde.pitch` and `omde.rythm` and a new `omde.plot` module to plot PDF sketchs out of your composition. Additional work is required in order to really integrate these features.

This `OMDE` representation of events as objects relies on the `Python` language object model. Going a step further, it opens the door to more abstract kind of musical objects, always espressed as `Python` objects. These objects are not directly mapped into score events but into whole groups of events and their properties are not the list of properties of the single events but properties of the set as a whole. This is a different kind of thing from the "aggregate" above and is not implemented at the moment in `OMDE/pmask` even if I'm already using a less formalized (and less functional) implementation of this idiom.

Since many people potentially interested in `OMDE/pmask` come from other algorithm or computer assisted composition experiences (`Cmask`, `SCORE11` for example), I'm starting to think that it could be useful to add "native languages" parsers and converters inside `OMDE`. Such languages are very simple in general so it is feasible. I already studied `SCORE11` and implemented the parser and part of the convertion of `SCORE11` in `pmask` objects.

Installing OMDE/pmask

Requirements

In order to use `OMDE/pmask` you need Python 2.0 or higher.

In order to use `omde.plot` you need `PIDDLE`.

Installation

To install `OMDE/pmask` on a console based environment, just run

```
python setup.py install
```

or simply

```
python setup.py
```

in the `omde` directory.

Windows and Macintosh users should be able to install `OMDE/pmask` by doubleclicking on the `setup.py` icon.

OMDE/pmask and Csound

There is quite a bit of overlap between a tool like `OMDE/pmask` and `Csound`.

The line between sound synthesis and algorithm or computer assisted composition is quite thin.

Notes

1. http://www.python.org

Chapter 2. First steps with `OMDE/pmask`

Here you can learn how to face simple problems with `OMDE/pmask`.

Some of the examples in this chapter are so simple that I probably would't use `OMDE/pmask` myself in real life if that was the only problem. Their aim is just to illustrate some basic operations and concepts.

A trivial cloud

Althought the term "cloud" is never used in the original Cmask documentation, using Cmask you just make clouds of sound events. This mean that you create a group of sound events controlling their parameters globally. You don't even control exactly the number of events in the cloud but only the cloud temporal extension.

Traditional configuration of events can be considered as special cases of stocastic clouds, where the parameters of single events are exactly specified. If fact we are going to build a sequence of identical events.

Let's say we have a simple Csound orchestra like this:

```
; trivialcloud.orc

sr=44100
kr=4410
ksmps=10
nchnls=1

giSinusoid ftgen 0, 0, 8192, 10, 1

instr 1

iDuration = p3
iAmplitude = ampdb(p4)
iFrequency = p5

kAmplitude linen iAmplitude, 0.1, iDuration, 0.1
aSignal oscil iAmplitude, iFrequency, giSinusoid

out aSignal

endin
```

then, this is the python code (using `OMDE/pmask`) needed to generate a 10 second long sequence of beeps at 1 second interval. Each sound is a 261 Hz, 0.4 seconds, 60 dB beep.

```
import pmask, omde.csound

score = pmask.cloud(0, 10, 1, 1.0, 0.4, 60, 261)

omde.csound.save('trivialcloud.sco', score)
```

The most interesting line of this simple snippet is the second one

```
score = cloud(0, 10, 1, 1.0, 0.4, 60, 261)
```

The first two parameters passed to `pmask.cloud()` are the start time and the end time of the cloud: our cloud starts at zero seconds and ends at 10 seconds.

The third parameter is the instrument number: instr 1. `cloud()` is not limited to generate csound I-statements but if the third parameter is a integer, it is exacly what it does.

The fourth parameter is the dispacement of successive events on the time axis (1 sec in our example). Pmask's documentation refer to this parameter as "density" but it is not a density at all. A temporal density is the frequence which is dimensionally the inverse of a time. This parameters is a time interval and the `OMDE/pmask` documentation refers to it as `dt`.

From the fifth parameter on we have the same parameters as we would pass them to instr 1: duration (0.4 seconds), level (60 dB), frequency (261 Hz).

With this line we created our trivial cloud and stored it in the object `score`. We can use this object to operate further transformations on the cloud but now we just want to get a standard numeric score file so that we can render it with Csound. The third line in the script does it:

```
omde.csound.save('trivialcloud.sco', score)
```

By running our `OMDE/pmask` script we'll get a `trivialcloud.sco` file, containing the Csound scorefile, in the current directory. This file will be something like:

```
i 1 0 0.4 60 261
i 1 1.0 0.4 60 261
i 1 2.0 0.4 60 261
i 1 3.0 0.4 60 261
i 1 4.0 0.4 60 261
i 1 5.0 0.4 60 261
i 1 6.0 0.4 60 261
i 1 7.0 0.4 60 261
i 1 8.0 0.4 60 261
i 1 9.0 0.4 60 261
```

Note: Even if you are accustomed to writing Csound score using an auxiliary language, it's worth to remember that if you change something in the script you have to run it to affect the real score file. This may sound like a pedantic warning but it's too much easy to forget it. At least, I do it sometimes.

Before we leave the trivial cloud example, let's say something about the first line in our script:

```
import pmask, omde.csound
```

it says to Python that we'll soon need the specified external components `pmask` and `omde.csound`. In the Python's lingo they are called "modules" and are containers of classes, objects and functions that are not part of the core of the language. A module (`omde` or `pmask`) may also contain other sub-modules: in this case it is called "package".

In general each module is dedicated to a specific service. Here we are loading `pmask`, which is the package dedicated to the algorithmic composition à la Cmask, and `omde.csound`, including Csound specific services in `OMDE`.

If you want to read more about Python module and packages go to the appendix A or directly to the Python language documentation.

Writing scores to standard output

`OMDE/pmask` users operating in GUI-based system will find this section of little or no interest.

Linux user or in general people using `OMDE/pmask` in console-based environments, may want to write the resulting numeric scores directly to the standard output.

This is done using the `stdout` object from the `omde.csound` module. You may want to re-write the trivial cloud example this way:

```
import pmask
from omde.csound import stdout

score = pmask.cloud(0, 10, 1, 1.0, 0.4, 60, 261)

stdout.write(score)
```

Supposing you called this script and its relate orchestra respectively `trivialcloud.py` and `trivialcloud.orc`, you will be able to write things like

```
trivialcloud.py | csound -d -o devaudio trivialcloud.orc /dev/stdin
```

Emulating Csound score parser's ramping feature

Now we want to add a little movement to the trivial cloud and we'll see how to imitate the behaviour of Csound score parser in presence of the ramping characters (**<** and **>**).

In particular we want our instrument to perform a crescendo from 40 dB to 80 db and sweeping frequencies an octave up from 261 Hz. We want the both variations to be linear, even if we know that only the intensity level scale is perceived as linear by humans.

Here follows a script (let's name it `trivialcloud2.py`) which shows how such a thing can be done using `OMDE/pmask`:

```
# trivialcloud2

import pmask, omde.csound
from pmask.bpf import LinearSegment

level = LinearSegment((0, 40), (10, 80))
frequency = LinearSegment((0, 261), (10, 261*2))
score = pmask.cloud(0, 10, 1, 1.0, 0.4, level, frequency)

omde.csound.save('trivialcloud2.sco', score)
```

When you make repeated use of the same values and especially when they have a important meaning, you may want to give them an explicit name. This has two advantages: the meaning of that value is evident and you can change globally that value without search and replace.

Using explicit names for the start and end time of the cloud, we can write the same program like this:

```
# trivialcloud2

import pmask, omde.csound
from pmask.bpf import LinearSegment

begin, end = 0, 10
level = LinearSegment((begin, 40), (end, 80))
frequency = LinearSegment((begin, 261), (end, 261*2))
score = pmask.cloud(begin, end, 1, 1.0, 0.4, level, frequency)

omde.csound.save('trivialcloud2.sco', score)
```

Using one style or the other is often matter of taste and habit.

In both cases, you end with a `trivialcloud2.sco` file containing the following score

```
i 1 0 0.4 40.0 261.0
i 1 1.0 0.4 44.0 287.1
i 1 2.0 0.4 48.0 313.2
i 1 3.0 0.4 52.0 339.3
i 1 4.0 0.4 56.0 365.4
i 1 5.0 0.4 60.0 391.5
i 1 6.0 0.4 64.0 417.6
i 1 7.0 0.4 68.0 443.7
i 1 8.0 0.4 72.0 469.8
i 1 9.0 0.4 76.0 495.9
```

In this script we introduced a new pmask symbol: `LinearSegment`. This is the first example of the so-called «generators».

At this point we may want to consider generators as objects that can be used in placed of fixed values in the `pmask.cloud()` call, representing values that change with time.

The line

```
level = LinearSegment((begin, 40), (end, 80))
```

says that `level` goes from 40 at the `begin` of this cloud to 80 at the `end` of the cloud. We know that `level` is expressed in dB.

As seen in the previous version of this script we can directly type numbers instead of `begin` and `end`, but this way we can easily and quickly change the begin time and/or the end time of the cloud.

Similarly the line

```
frequency = LinearSegment((begin, 261), (end, 261*2))
```

says that `frequency` goes from 261 Hz at the `begin` of this cloud to the higher octave at the `end` of the cloud.

`LinearSegment` belongs to a special group of generators, the "break point function" (BPF) generators. Such generators allow the user to indicate the value of the function in a finite set of points and interpolate the value of the functions in the other points. Different kind of BPF generators use a different interpolating functions.

`LinearSegment` uses linear interpolation, just like the simple ramping operators in Csound.

You can find `LinearSegment` and other BPF generators in the `pmask.bpf` module. As usual you have to import `LinearSegment` as in the following line, taken from the script above:

```
from pmask.bpf import LinearSegment
```

Emulating Csound score parser’s time warping feature

Given an `Aggregate` (or a `ScoreSection`, if you prefer) we can perform time warping using the `timewarp()` method.

Let’s say we have a file (`trivialcloud.sco`) containing a score fragment and we want to apply some tempo change. Suppose you can’t use the `t` statement for some reason. We can do it using `OMDE` even from an interactive Python shell.

```
Python 2.0 (#2, May 22 2001, 11:42:51)
[GCC 2.96 20000731 (Red Hat Linux 7.1 2.96-81)] on linux2
Type "copyright", "credits" or "license" for more information.
> > > from omde.all import *
> > > score = read_score(’trivialcloud.sco’)
> > > stdout.write(score)
i 1.0 0.0 0.4 60.0 261.0
i 1.0 1.0 0.4 60.0 261.0
i 1.0 2.0 0.4 60.0 261.0
i 1.0 3.0 0.4 60.0 261.0
i 1.0 4.0 0.4 60.0 261.0
i 1.0 5.0 0.4 60.0 261.0
i 1.0 6.0 0.4 60.0 261.0
i 1.0 7.0 0.4 60.0 261.0
i 1.0 8.0 0.4 60.0 261.0
i 1.0 9.0 0.4 60.0 261.0
> > > f = Tempo((0, 60), (5, 60), (5, 320), (10, 120))
> > > score.timewarp(f)
> > > stdout.write(score)
i 1.0 0.0 0.4 60.0 261.0
i 1.0 1.0 0.4 60.0 261.0
i 1.0 2.0 0.4 60.0 261.0
i 1.0 3.0 0.4 60.0 261.0
i 1.0 4.0 0.4 60.0 261.0
i 1.0 5.0 0.4 60.0 261.0
i 1.0 5.21875 0.4 60.0 261.0
i 1.0 5.5 0.4 60.0 261.0
i 1.0 5.84375 0.4 60.0 261.0
i 1.0 6.25 0.4 60.0 261.0
> > >
```

With the first line

```
from omde.all import *
```

you import everything from the `omde` package. It is a shortcut that I don’t like very much but can be useful sometimes like when doing an interactive Python session for a quick and dirty job.

Then we load our score excerpt from a file using the `read_score()` function.

```
score = read_score(’trivialcloud.sco’)
```

`read_score()` returns a `ScoreSection` containing the score except. At the moment only `i` and `f` statements are imported. Everything else is ignored. Special features (carry, ramping, warping) are not supported.

> **Note:** In this section I use the two terms `ScoreSection` and `Aggregate` as the were the same thing. Tecnically speaking `ScoreSection` is a subclass of `Aggregate`. This means that `ScoreSection` has every property and ability of an `Aggregate`: for example, just like all possible kinds (or subclasses) of `Aggregate` you can `timewarp()` it. But `ScoreSection` is/has something more that a simple `Aggregate`: it knows how to transform in a Csound standard numeric score.

After priting the score we create a `Tempo` object in a way very similar to Csound's `t` statements, indicating pairs (time, tempo) where the time is expressed in unwarped units name "beats".

```
f = Tempo((0, 60), (5, 60), (5, 320), (10, 120))
```

And finally we perform the time warping for real:

```
score.timewarp(f)
```

If you prefer, feel free to combine the last two commands in a single line like this:

```
score.timewarp(Tempo((0, 60), (5, 60), (5, 320), (10, 120)))
```

Speaking more generally the `timewarp()` method remaps all events in an `Aggregate` using the specified function to map time to time.

```
score.timewarp(f)
```

`f(t)` must be a function-like object returning the new onset of an event which currently begins a time `t`. The `Tempo` is just an easy way to build such a function in terms of tempo and tempo variations.

If you plan to do this operation several times you may prefer to write a script, something like the following one:

```
from omde.csound import read_score, save, Tempo

score = read_score('trivialcloud2.sco')
score.timewarp(Tempo((0, 60), (5, 60), (5, 320), (10, 120)))
save('timewarped.sco', score)
```

That's all.

The first bell example from the

Chapter 3. `pmask`

Pushing `pmask` beyond `CMask`

If you want to use `OMDE/pmask` you are probably interested in the content of the `omde.pmask` module and in particular in the `cloud()` function in this module.

In fact the `cloud()` function toghether with `OMDE`'s functions and generators can be used to replace the Cmask program in generating clouds. `OMDE`'s clouds re what Cmask names «fields».

But with `OMDE/pmask` you can the same algorithm to generate not only Csound score events but also more complex events.

The full syntax for `cloud()` is

```
cloud(begin, global_duration, CTOR, dt, duration, ...)
```

This function call return a cloud of events as an `Aggregate`.

The first two arguments are the begin time and the global (total) duration of the cloud as a whole.

Note: Here is a first difference between `OMDE/pmask` and `Cmask`: for reasons of coherence a cloud's temporal extention is defined by the start time and the duration instead of the start time and the end time as `Cmask` fields.

The third parameter is the class name of the event or the name of an event (or more often Aggregate) factory.

The fourth parameter is the time interval between two successive events.

The fifth parameter is the event duration.

Chapter 4. `OMDE`

`OMDE` signals

In `OMDE` OMDE signals come in two flavors: function (subclasses of `Function`) and generators (subclasses of `Generator`). The concept of `FunctionModel` may also make your life easier sometimes.

Functions

Functions derive from the `Function` class in the `omde.functional` module and are function of time. Thus they can be evaluated at a specific instant `t0` by calling them. For example if `f` is an instance of some kind of `Function`, then you could write

```
value = f(t0)
```

A well behavied function should always return the same value when evaluated a the same time, but there is no way for `OMDE` to enforce this condition and there could be case where this statement is not true.

Note: Functions are the most similar concept to `Cmask`'s generators and to early versions of `pmask`'s generators.

Generators

Generators derive from the `Generator` class in the `omde.functional` module. They just return a new value each time they are called. For example if `g` is an instance of some kind of `Function`, then you could write.

```
value = g()
```

There is not explicit time dependence here and a generator (for example a pseudo random number generator) is free to return whatever result at each call.

Function models

Function models are subclasses of `FunctionModel` and can be regarded as functions «out of time». They store a function shape but not the time interval within it develop-opts. They are very similar to `Csound` function tables read by an oscillator.

If `m` is a function model you can get a real function (an instance) in the `[t0, t1]` time range by using the `instance()` method.

```
f = m.instance(t0, t1)
```

Each `Function` is also a `FunctionModel` and return itself when instanced.

Converting between generators and functions

Of course you can convert back and forth between generators and functions but most of the time this kind of conversion is performed automatically behind the scene (see below).

Converting a generator into a function

Converting a generator into a function is done by using a `TimeDependenceAdaptor` object from the `omde.functional` module. If `g` is a generator you could write something like this:

```
f = TimeDependenceAdaptor(g)
```

Now `f` can be evaluated as a function of time even if it is not:

```
value = f(t0)
```

Anyway you should be aware that `f` still behaves like a generator: if `g` is, say, a RNG evaluating twice the `f` at the same time is likely to return two different values!

Converting a function to a generator

Converting a function into a generator is done by using a `Freezer` object from the `omde.functional` module. If `f` is a function and `t` is a generator that returns real numbers representing points in the time axis, you could write something like this:

```
g = Freezer(f, t)
```

Now `g` can called as generator to get the next value:

```
value = g()
```

As a special case `t` could be a real number and in this case the function would be evaluated always at the same time. The default value for the `t` argument in `Freezer` is zero.

Automatic conversion

There are a pair of functions from the `omde.functional` module that help you to be sure of the kind of signal you are handling. They are used by the `omde.pmask` algorithms and make life easier to both algorithm developer and user.

Making a function out of almost anything

If you have a not well defined object and you want to turn it into a function you could use the `make_function()` function from the `omde.functional` module.

The syntax is as follows:

```
make_function(object, begin = None, end = None)
```

If object is a function, make_function() returns object.

If object is a `Generator` `make_function()` performs adaptation and return the resulting `Function`.

If object is a `FunctionModel` it is instanced and `make_function()` returns the resulting function. But if both begin and end are None, `make_function()` expects only ready-to-use generators and not models.

Everything else is turned into a `ConstantFunction`.

Making a generator out of almost anything

If you have a not well defined object and you want to turn it into a generator you could use the `make_generator()` function from the `omde.functional` module.

The syntax is as follows:

```
make_generator(object [, t0=0.0])
```

If object is a `Generator` `make_generator()` returns object.

A `Function` be converted to a `Generator` by freezing it at `t0` (default value for `t0` is `0.0`).

Everything else is turned into a `ConstantGenerator`.

Appendix A. Python basics

modules, packages and the `import` command