Le projet `pythoneon`

Composer de la musique électronique en Python

L'association CARMEN – René Bastian

2003-2012

Composer de la musique électronique :

- ▷ synthétiser des sons par algorithmes,
- ▷ placer les sons dans le temps,
- ▷ placer les sons dans l'espace (n haut-parleurs).

- ▷ ne pas utiliser le système MIDI,
- ▷ ne pas utiliser des sons physiques enregistrés.

Mon premier son en Python (version originale)

```
import math
import array
t = 0.0; duree = 5.0; sr = 44100; dt = 1.0/sr
R = []; freq = 440.0
while t < duree:
     y = math.sin(2*math.pi*freq*t)
     R.append(y)
     t = t+dt
r = array.array('h')
for i in range(len(R)):
     y = int(R[i]*32767)
     r.append(y)
f = open('premierson.raw','wb')
r.tofile(f)
f.close()
```

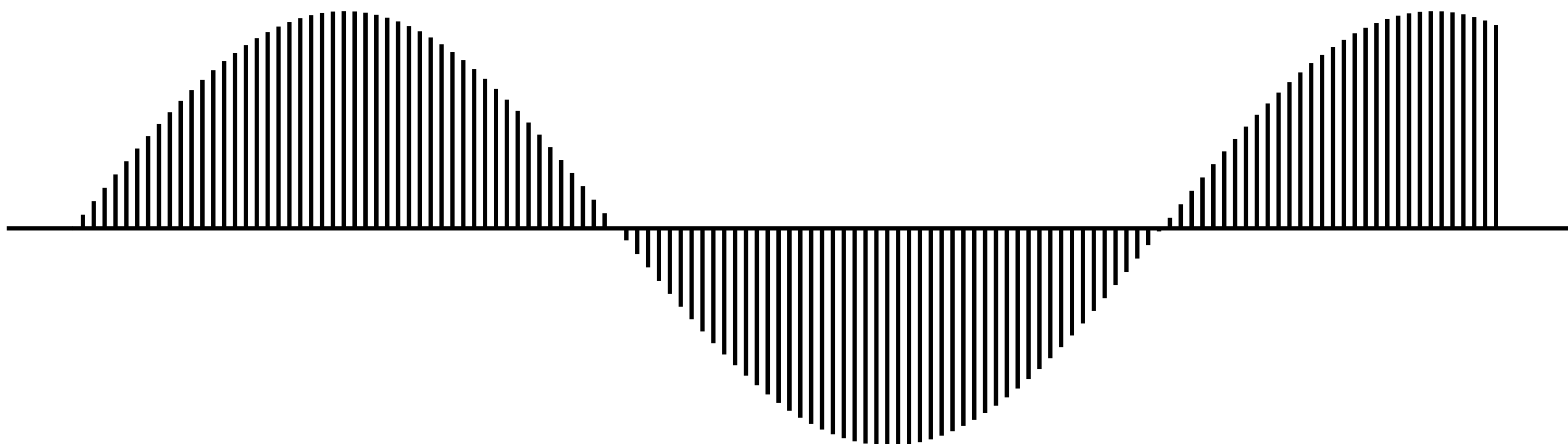
Agencement de deux sons dans le temps (reconstitution historique)

```
import numpy as np
sr = 44100
freq1, duree1 = 440.0, 3.0
freq2, duree2 = 770.0, 4.0
decalage = 1.5
dureeTotale = decalage + max(duree1, duree2)
dt = 1.0 / sr
t1 = np.arange(0.0, duree1, dt)
t2 = np.arange(0.0, duree2, dt)
w1 = np.sin(2 * np.pi * freq1 * t1)
w2 = np.sin(2 * np.pi * freq2 * t2)
nTotal = int(dureeTotale * sr)
wr = np.zeros(nTotal, np.float64)
wr[ : w1.size] = w1
n2 = int(decalage * sr)
wr[n2 : n2 + w2.size] += w2
#---------------------------
from audiodirect import audio
audio.play(wr, wr)
audio.rec(wr, wr, "wpreuve", "wpreuve")
```

Ingrédients nécessaires

1. des sinus,
2. des bruits,

Image de sinus :



L'algorithme des sinus :

```
y[i] = k * y[i-1] - y[i-2]
```

Établissement d'une récurrence pour le calcul des valeurs d'un sinus

$$\sin(\omega t + \omega dt) = \cos(\omega dt)\sin(\omega t) + \sin(\omega dt)\cos(\omega t) \tag{1}$$

$$\cos(\omega t + \omega dt) = -\sin(\omega dt)\sin(\omega t) + \cos(\omega dt)\cos(\omega t) \tag{2}$$

$$dt = 1.0 \,/\, \texttt{sr} \tag{3}$$

$$q = \sin(\omega dt) \tag{4}$$

$$p = \cos(\omega dt) \tag{5}$$

$$x_i = p\, x_{i-1} + q\, y_{i-1} \tag{6}$$

$$y_i = -q\, x_{i-1} + p\, y_{i-1} \tag{7}$$

$$x_{i-1} = p\, x_{i-2} + q\, y_{i-2} \tag{8}$$

$$y_{i-1} = -q\,x_{i-2} + p\,y_{i-2} \tag{9}$$

$$y_{i-2} = \frac{1}{q}(x_{i-1} - p\,x_{i-2}) \tag{10}$$

$$x_i = p\,x_{i-1} + q\,(-q x_{i-2} + p\,y_{i-2}) \tag{11}$$

$$x_i = p\,x_{i-1} + q\,(-q x_{i-2} + p\,\frac{1}{q}(x_{i-1} - p\,x_{i-2})) \tag{12}$$

$$x_i = p\,x_{i-1} + -q^2 x_{i-2} + p\,x_{i-1} - p^2\,x_{i-2} \tag{13}$$

$$x_i = p\,x_{i-1} + p\,x_{i-1} - (q^2 + p^2)\,x_{i-2} \tag{14}$$

$$x_i = 2\,p\,x_{i-1} - (q^2 + p^2)\,x_{i-2} \tag{15}$$

$$x_i = 2\cos(\omega dt)\,x_{i-1} - x_{i-2} \tag{16}$$

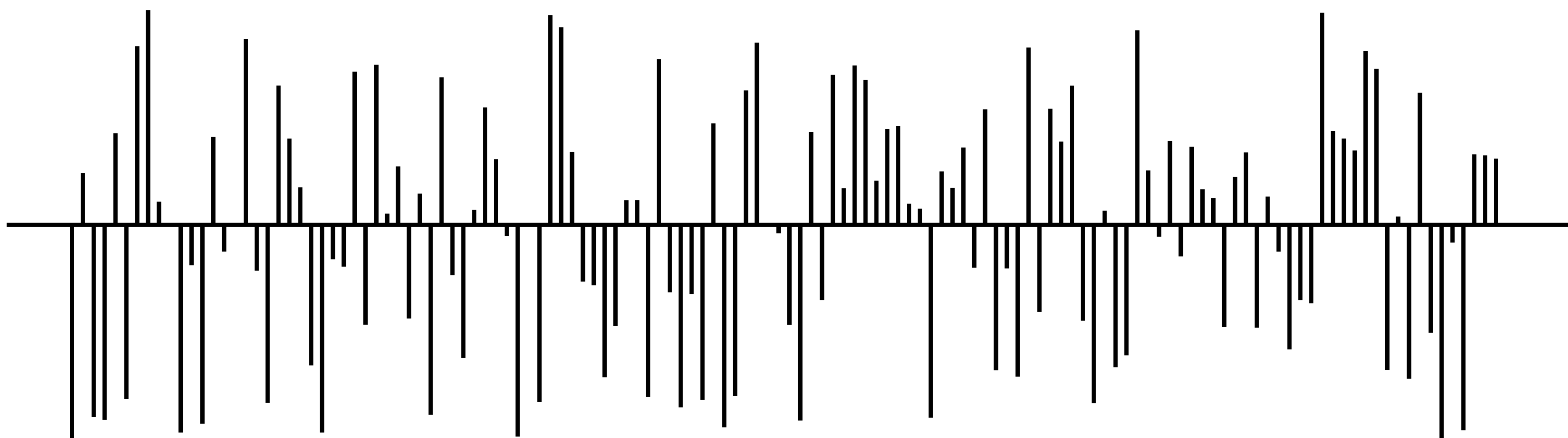
```
def sinrecursif(k, n, xini=1.0):
    w = np.zeros(n)
    w[0] = 0.5
    w[1] = xini
    for i in range(2, n):
        w[i] = k*w[i-1] - w[i-2]
    return w

def sinusrecursif(freq, duree, xini=1.0):
    n = long(duree*sr)
    k = 2*np.cos(2*np.pi*freq/sr)
    return sinrecursif(k, n, xini)
```

Conversion fréquence $\rightarrow$ k

Le principe d'une

```
def freq2ksin(freq):
    return 2 * np.cos( 2 * np.pi * freq / sr)
```

Image du bruit :



Un algorithme pour faire du bruit :

```
y[i] = a * y[i-1] + b % m
```

C'est une récurrence classique, proposée par D. H. Lehmer en 1949.

En raison de leur limitation par `% m`, ces récurrences fournissent des cycles. Ce que Schönberg avait appelée des *séries*[*].

[*]. d'où ma dénomination de *générateurs sériels*

Pythoneon

Principes de `pythoneon` (1)

- ▷ **largeur du mot de calcul** : 64 bits (au lieu de 16, 24 ou 32 bits ; l'erreur de calcul de 32 bits par rapport à 64 bits est de l'ordre de 0.5% soit de l'ordre d'un demit-ton chromatique !)
- ▷ **fréquence de samplage** : elle est utilisée sous forme symbolique et sa valeur n'est appliquée que lors de la compilation finale ;

Principes de `pythoneon` (2)

- **spécifications de durée** : absolue en sec. mais symbolique possible pour gérer le tempo ;
- **spécifications de hauteur** : deux possibilités :
  - **Hz** : fréquence en cycles par seconde (absolue),
  - **Prony** : échelle logarithmique ayant le *demi-ton chromatique* comme unité, avec possibilité de redéfinition de l'échelle : 69.0 $\rightarrow$ 440, 435, 442, 445, . . . Hz.
- **spécifications des macro-filtres** en sec. au lieu de samples

Principes de `pythoneon` (3)

- la boucle `écriture` $\rightarrow$ `écoute` $\rightarrow$ `écriture` $\rightarrow$ ... s'arrête quand l'écoute fournit le résultat recherché : les rectifications ultérieures ne doivent pas affecter le résultat,
- concept de **programme-partition**

Programme-partition

Mais ceci ne veut pas dire que le programme ne comporte pas d'*erreurs* (logiques, théoriques, stylistiques etc.).

Mais toute amélioration du code doit préserver le résultat initial.

Attention aux erreurs . . .

Pythoneon utilise :

- ▷ le Python standard
- ▷ l'extension Numpy

Pythoneon inclut :

1. Collecteurs
2. Profils
3. Ondes, oscillateurs, générateurs
4. Filtres
5. Outils (journal, pyls, evald, jt, etc.)

Collecteurs

Collecteurs

1. `CanalMono` collectage et mixage de micro-structures (samples, ondes, trains d'ondes) sur un canal,

2. `CanalMulti` idem sur 2 ou plusieurs canaux,

3. `Partition` assemblage et mixage d'événements sonores.

La classe CanalMono

Cette classe utilise la notion de *point temporel* qui correspond à l'intervalle de temps en sec. par rapport à l'origine 0.0 fixée au moment de la création d'une instance.

La classe CanalMono

Principales méthodes :

`s.addT(pt, signal)` ajoute le **`signal`** à partir du point temporel **`pt`** ; comme le point **`pt`** est indiqué en unités temporelles indépendantes, on est indépendant de la fréquence d'échantillonnage sr.

Cela commence en général par `c = CanalMono()` et se termine par un `return c.a`.

`s.rec(snom, numeroCanal` écrit un fichier en format `float64` utilisé par la classe `Partition` lors du montage final.

Exemple :

```
def evt(panos, durees, amplis, pronys, ecarts):
    cg, cd = CanalMono(), CanalMono()
    xpt = 0.0
    for pano, duree, ampli, hprony, ecart in /
            zip(panos, durees, amplis, pronys, ecarts):
        w = corde(hprony, duree)
        w *= ampli
        wg, wd = w * pano, w * (1. - pano)
        cg.addT(xpt, wg)
        cd.addT(xpt, wd)
        xpt += ecart
    return cg.a, cd.a
```

La classe Partition

Lors de la création d'une instance on peut inclure une valeur *unité de temps* : `Partition.u`

Il est possible de changer cette unité :

- ▷ abruptement, ou
- ▷ continûment par variation de *tempo*.

La classe `Partition`

Les principales méthodes :

`s.pt(t)` le point temporel actuel prend la valeur `t`, d'où `Partition.t = t * Partition.u`

`s.dr(ecart)` fixe le prochain point temporel en additionnant `ecart` au point temporel actuel, donc on aura `Partition.t += ecart * Partition.u`.

On dispose donc d'une méthode absolue et d'une méthode relative pour fixer le point temporel.

`s.evt(*signals)` **`signals`** est une liste de signaux, chaque chaque signal correspond à un canal (ou haut-parleur) ; le signal est ajouté au point temporel actuel.

`s.montage(instructions)` insère les fichiers f64 désig
par **`souche`** aux points temporels donnés avec l'amplitude donnée.

`s.montage_ecart(instructions)` les points temporels sont espacés par des écarts.

```
scoreprinc = Partition("noctua", 2)
ventA()
ventB()
ventC()
ventD()
guimbeE()
guimbeF()
fluteM()
fluteN()
fluteO()

listemontagefinale = [
    ("ventD",      0.0, 1.0),
    ("guimbeF",   20.0, 1.25),
    ("fluteO",    51.0, 0.0625),
    ("fluteN",    62.0, 0.0625),
```

```
    ("ventA",   106.0, 2.0),
    ("ventB",   170.0, 0.5),
    ("guimbeE", 170.0, 4.0),
    ("ventC",   234.0, 1.0),
    ("fluteM",  234.0, 0.125)
    ]
scoreprinc.montage(listemontagefinale)
scoreprinc.close()
```

Profils

Les événements musicaux sont rarement figés. Il nous faut donc un moyen pour décrire ces mouvements, c'est-à-dire des lignes qui les guident : **les profils**

Avec les `profils` on décrit :

- ▷ les enveloppes (variations d'intensité),
- ▷ les variations de hauteur,
- ▷ la fréquence d'accord des filtres,
- ▷ la largeur de bande des filtres,
- ▷ les écarts de temps entre des trains d'événements,
- ▷ les durées d'une suite d'événements,
- ▷ des ondes,
- ▷ etc.

Construction de profils :

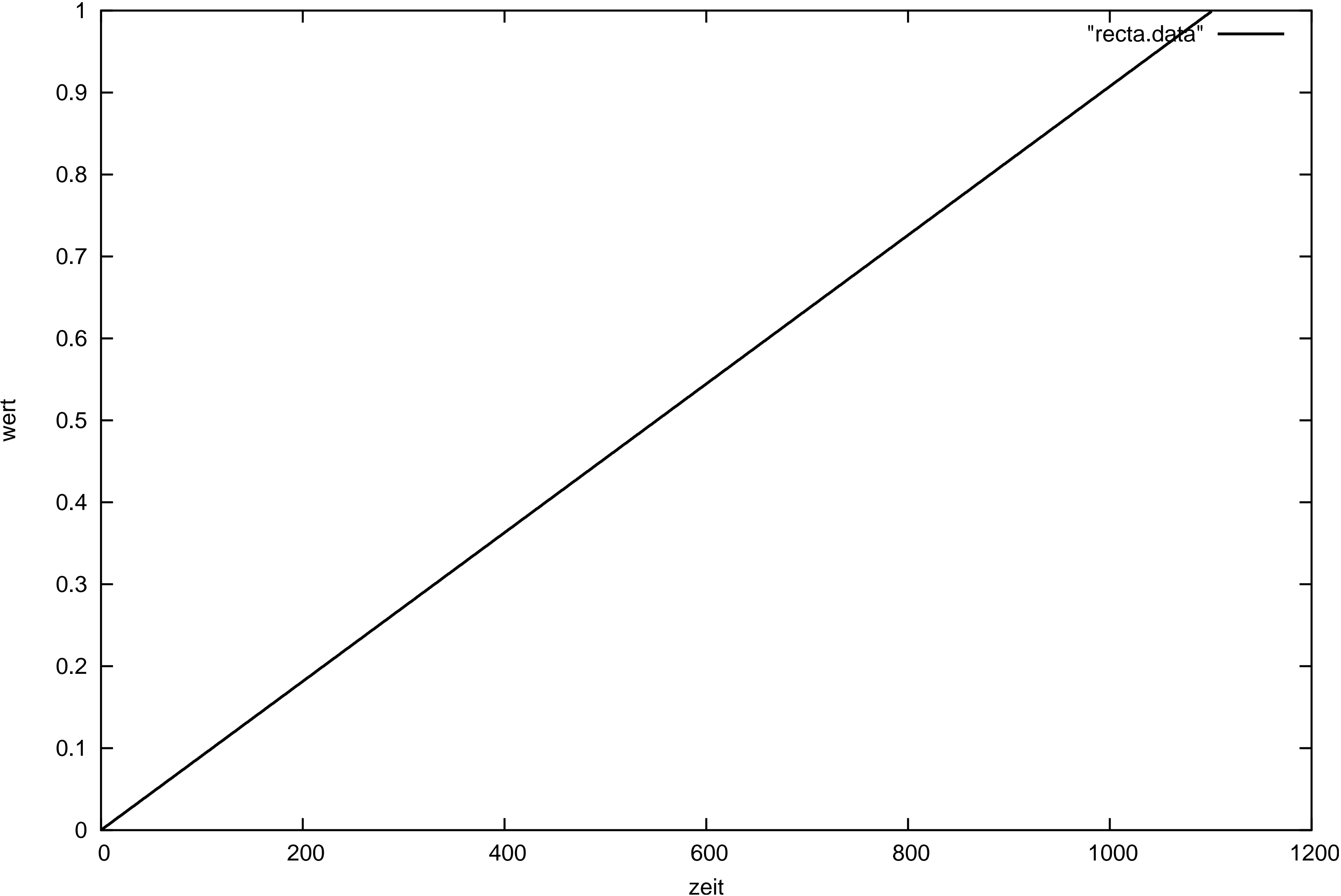
Les constituants élémentaires :

1. la droite rectiligne ou tordue : **`recta`**
2. les cubiques : **`cubica`**
3. . . .
4. des récurrences à la Gumowski & Mira
5. des fonctions mathématiques,
6. des courbes de Bézier.

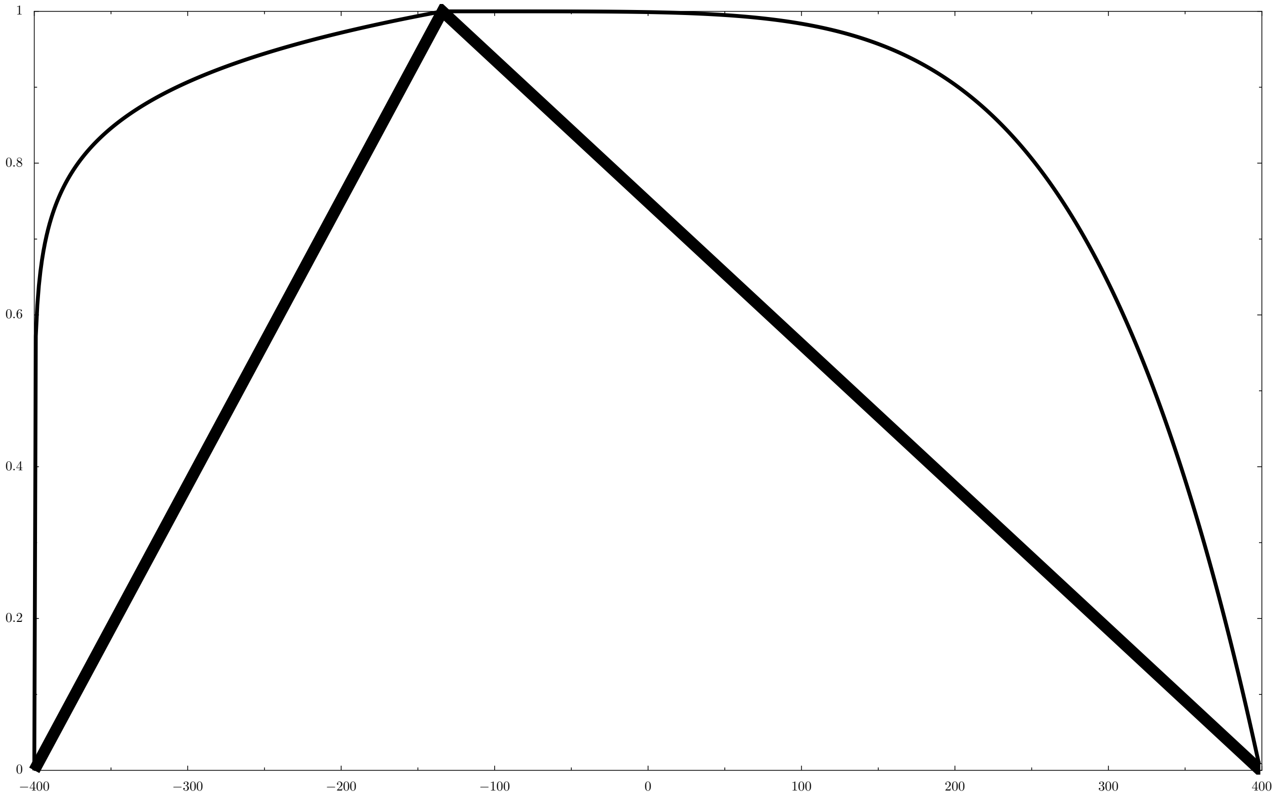
Lignes droites plus ou moins tordues

Syntaxes de recta :

```
[recta, duree, y0, y1]

[recta, duree, y0, y1, courbure]
```


Profil: recta
"recta.data"
wert
zeit
1
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
0
200
400
600
800
1000
1200

Un profil droit et le même tordu



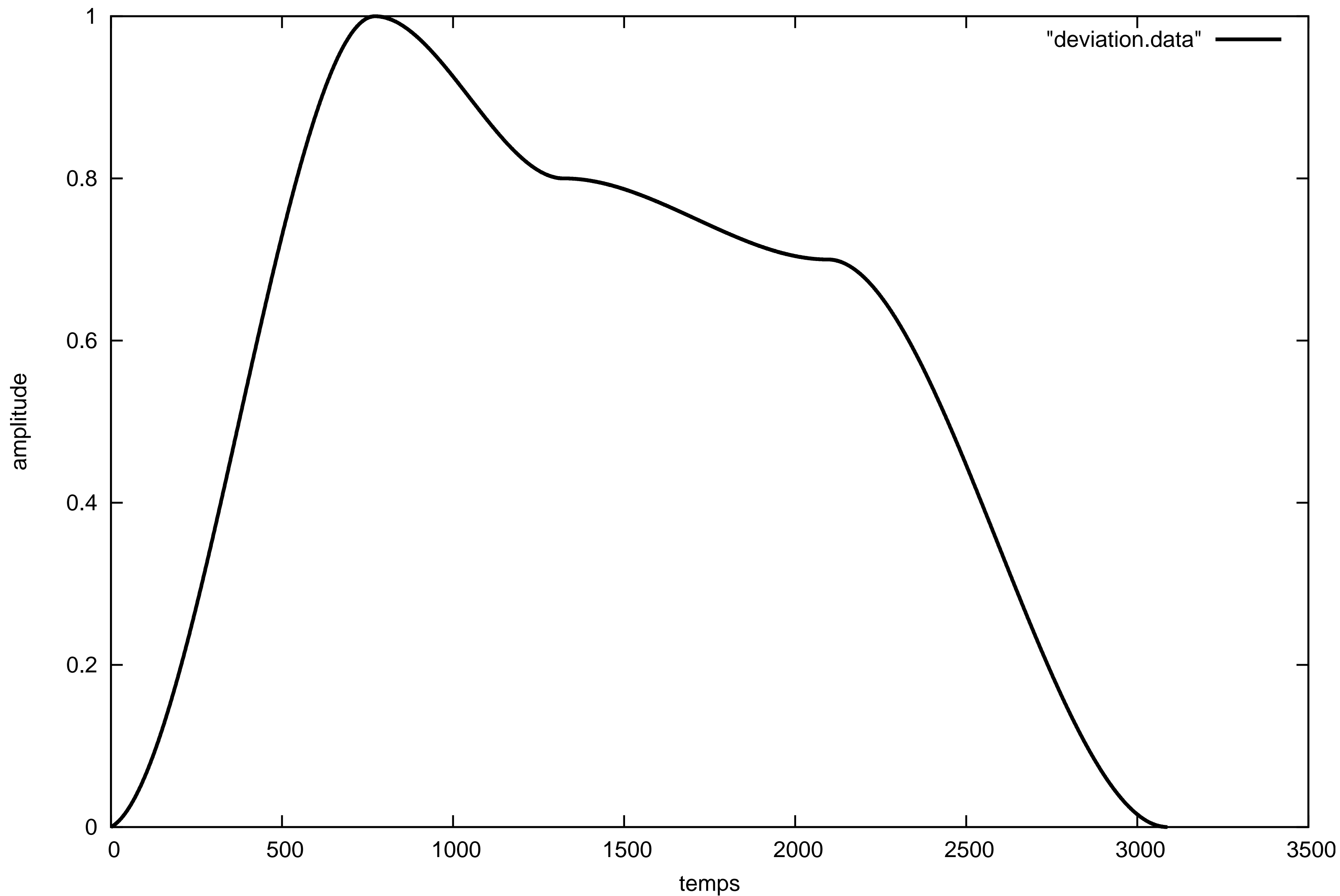
Construction de courbes avec cubica :

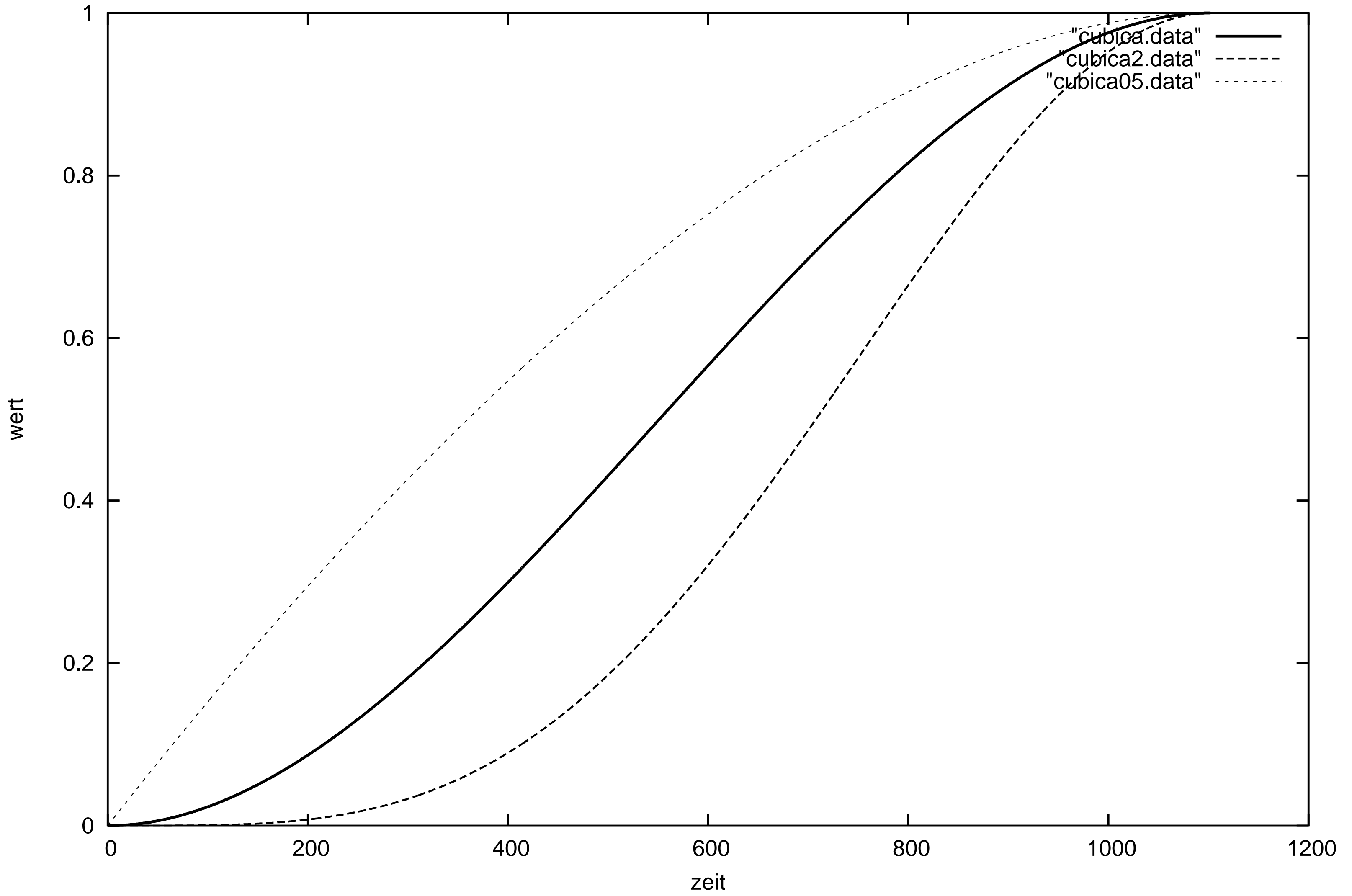
Syntaxe :

```
[cubica, duree, y0, pente0, y1, pente1]
```

Construction d'un profil avec quatre cubicas :

```
def calculdeviation():
    E=[[cubica, 3.5, 0.0, 10.0, 1.0, 0.0],
       [cubica, 2.5, 1.0, 0.0,  0.8, 0.0],
       [cubica, 3.5, 0.8, 0.0, 0.7, 0.0],
       [cubica, 4.5, 0.7, 0.0, 0.0, 0.0]]
    return profil(E)
```


Profil
"deviation.data"
amplitude
temps
0
0.2
0.4
0.6
0.8
1
0
500
1000
1500
2000
2500
3000
3500

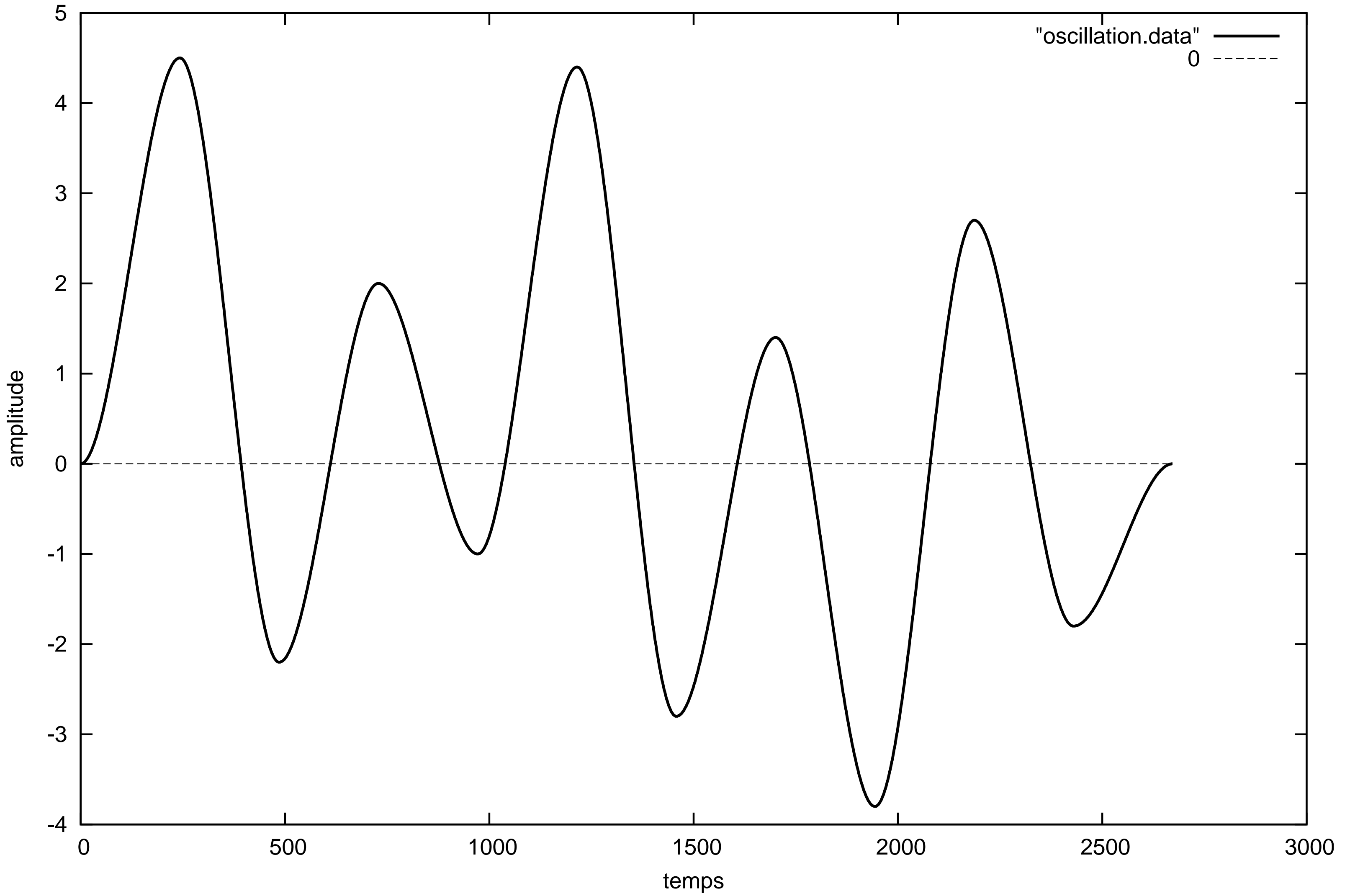

Profil: cubica
"cubica.data"
"cubica2.data"
"cubica05.data"
wert
zeit
0
0.2
0.4
0.6
0.8
1
0
200
400
600
800
1000
1200

Torsion d'une cubica par des puissances :

```
def torsions():
    D = [[cubica, 0.025, 0.0, 0.0, 1.0, 0.0]]
    d = profil(D)
    ecrire("cubica.data", d)

    d2 = d*d
    ecrire("cubica2.data", d2)

    d05 = d**0.5
    ecrire("cubica05.data", d05)
```

Construction d’une oscillation :

```
def calculOscillation():
    E = [[cubica, 1.0, 0.0, 0.0, 4.5, 0.0],
        [cubica, 1.0, 4.5, 0.0,  -2.2, 0.0],
        [cubica, 1.0, -2.2, 0.0, 2.0, 0.0],
        [cubica, 1.0, 2.0, 0., -1.0, 0.0],
        [cubica, 1.0, -1.0, 0., 4.4, 0.0],
        [cubica, 1.0, 4.40, 0., -2.8, 0.0],
        [cubica, 1.0, -2.8, 0., 1.4, 0.0],
        [cubica, 1.0, 1.4, 0., -3.8, 0.0],
        [cubica, 1.0, -3.8, 0., 2.7, 0.0],
        [cubica, 1.0, 2.7, 0., -1.8, 0.0],
        [cubica, 1.0, -1.8, 0., 0, 0.0]]
    return profil(E)

e = calculOscillation()
ecrire("Oscillation.data", e)
```

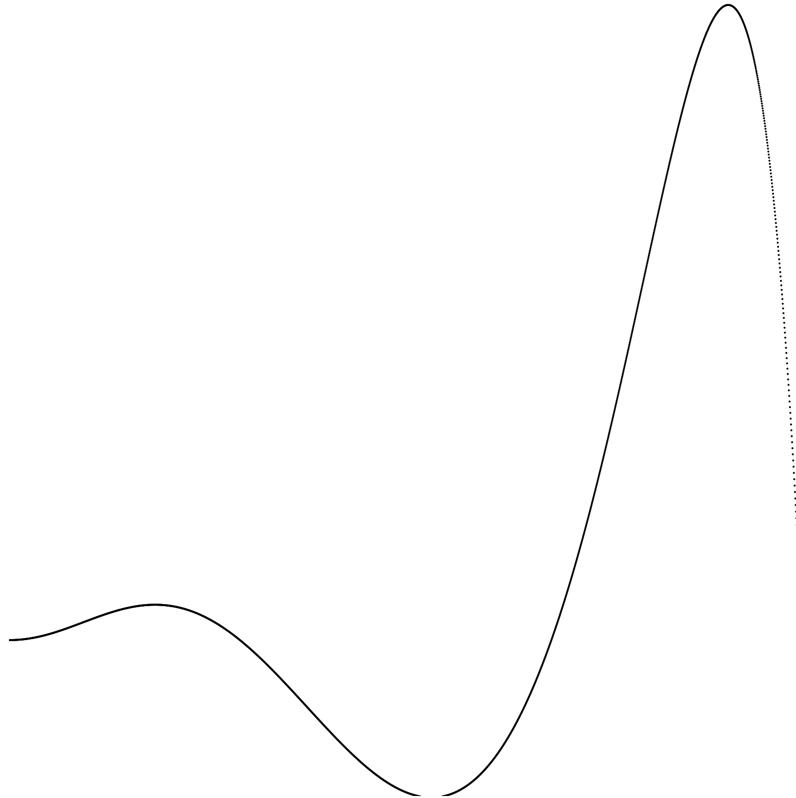

Profil
"oscillation.data"
0
amplitude
temps
5
4
3
2
1
0
-1
-2
-3
-4
0
500
1000
1500
2000
2500
3000

Adaptation de fonctions mathématiques

La fonction 'f' fournit les valeurs de f(x0) à f(x1) qui sont adaptées à l'intervalle (y0, y1). duree : durée y0, y1 : débattements f : une fonction écrite en Python x0, x1 : valeurs initiale et finale pour f(x)

```
def funcprofil(duree, y0, y1, f, x0, x1):
    n = int(duree * sr)
    vx = np.linspace(x0, x1, n)
    p = f(vx)
    dp = p[-1] - p[0]
    p -= p[0]
    dy = y1 - y0
    c = dy / dp
    p *= c
    p += y0
    return p
```

Courbes de Bézier



```
p0 =  [0.5, 0.0, (0.1, 1.0), (0.23, 4.0), (0.1, -11.0),
       (0.5, 5.0), (0.1, -15.0), (0.1, 25.), (0.2, 0.0)]
```

Ondes

Méthode de calcul d'ondes de bas niveau

1. balayage d'une forme d'onde,
2. calcul mathématique (numpy, $y = f(t)$),
3. récurrences (p. ex. sinus par récurrence).

Méthodes de création d'ondes complexes

1. procédés mathématiques : modulations de fréquence enchâssées,
2. superposition d'ondes,
3. construction de train d'ondes avec ou sans tuilage (CanalMono),
4. . . .
5. processus combinatoires.

Modulation de fréquence

La modulation de fréquence a été inventée par les oiseaux il y a tellement longtemps que leur brevet est tombé dans le domaine public.

Mais certains Bureaux de brevetage ne le savent pas.

Filtres

Filtres

1. **Filtre moyenneur** agit comme un filtre passe-bas brutal et n'a peut-être qu'intérêt pédagogique
2. **Filtre en peigne** accentue ou déprime des bandes de fréquences régulièrement espacées d'un intervalle donné.
3. le **Filtre fixe** → Filtres passe-bas, passe-haut, bandes, éliminateurs *
4. **Filtres récursifs** → Filtres de bandes

*. Avec Python, les filtres fixes deviennent flexibles.

Outils

Outils

- journal : rédactions depuis n jours
- pyls : un `ls` spécialisé pour `.py, .f64, .raw, ...`
- evald : calcul des durées de fichiers audio,

Musique concrète

Et la musique concrète ?

- capter les sons dans la meilleure définition possible,
- recoder en 64 bits,
- utiliser toutes les ficelles : filtrage, découpage, imposition d'enveloppes, modulations, montage . . . ,
- finir comme d'habitude.

Avantages des langages généralistes

On peut :

- ▷ explorer les mécanismes et algorithmes de synthèse, donc **expérimenter**,
- ▷ reproduire l’acquis à volonté,
- ▷ transmettre la partition.

Avantages du langage Python

- langage lisible,
- langage puissant et rapide,
- langage indéfiniment extensible,
- non compilable, donc ouvert et coopératif ;)
- apte à gérer tout le domaine informatique.

(accès au disque, accès au web, etc.)

F I N

Ce texte a été écrit en Latex avec **emacs** sur un système Debian/Ubuntu 12.04, compilé par **pdflatex** et projeté par **evince**.