Filtres intégrateur et dérivateur

1. Introduction

Ce document montre comment effectuer l'intégration et la dérivation d'un signal numérique.

L'opération de dérivation est particulièrement difficile à réaliser, en raison de sa très grande sensibilité au bruit présent dans le signal. On verra comment un sur-échantillonnage associé à un filtrage passe-bas permet d'améliorer considérablement le filtre dérivateur.

2. Filtre intégrateur

2.a. Définition du filtre

Un intégrateur analogique réalise l'opération suivante (à une constante multiplicative près) :

$$y(t) = \int_0^t x(u)\,du$$

Pour un signal discret avec une période d'échantillonnage T_e, la relation de récurrence la plus simple qui vienne à l'esprit est :

$$y_n = y_{n-1} + T_e x_n$$

En effectuant la transformée en Z de cette équation on obtient :

$$Y(z) = Y(z)z^{-1} + T_e X(z)$$

D'où on déduit la fonction de transfert en Z :

$$H(z) = \frac{Y(z)}{X(z)} = \frac{T_e}{1 - z^{-1}} = \frac{T_e z}{z - 1}$$

La réponse impulsionnelle est la transformée en Z inverse de la fonction de transfert. Dans le cas présent, la réponse impulsionnelle est obtenue en consultant un tableau des transformées en Z usuelles ([1]) : $h_n = T_e u_n$, où u_n est l'échelon unité. Il s'agit donc d'un filtre à réponse impulsionnelle infinie.

La réponse fréquentielle est obtenue en posant $z = \exp(j\omega T_e)$:

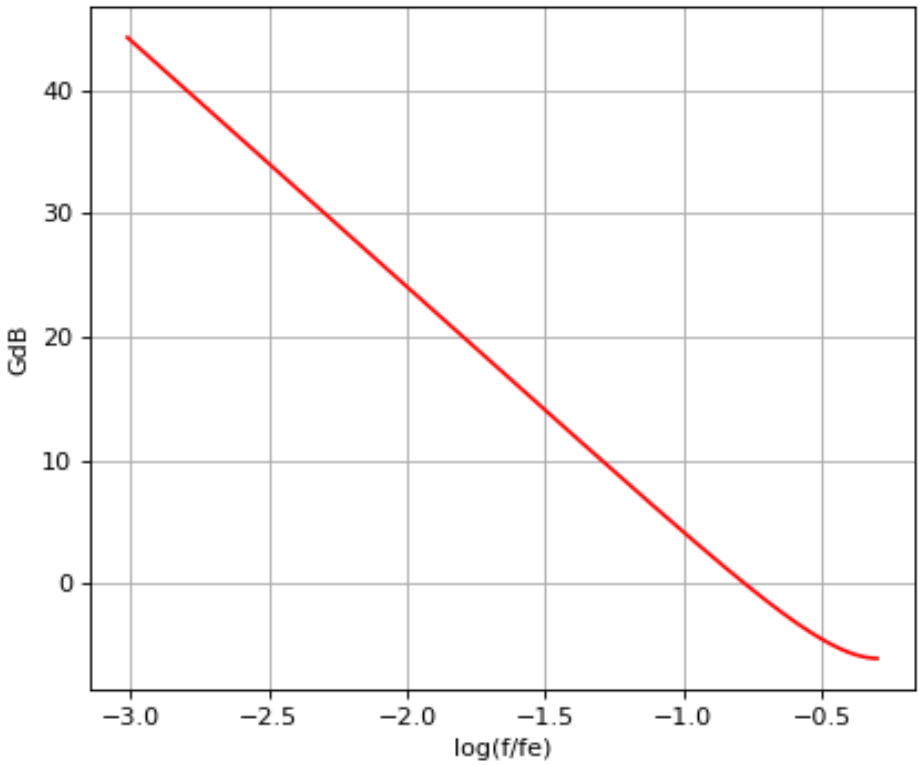
$$H(\omega) = \frac{T_e \exp(j\omega T_e)}{\exp(j\omega T_e) - 1}$$

La réponse fréquentielle peut être obtenue avec la fonction `scipy.signal.freqz`. Pour cela, il faut fournir la fonction de transfert en Z sous la forme :

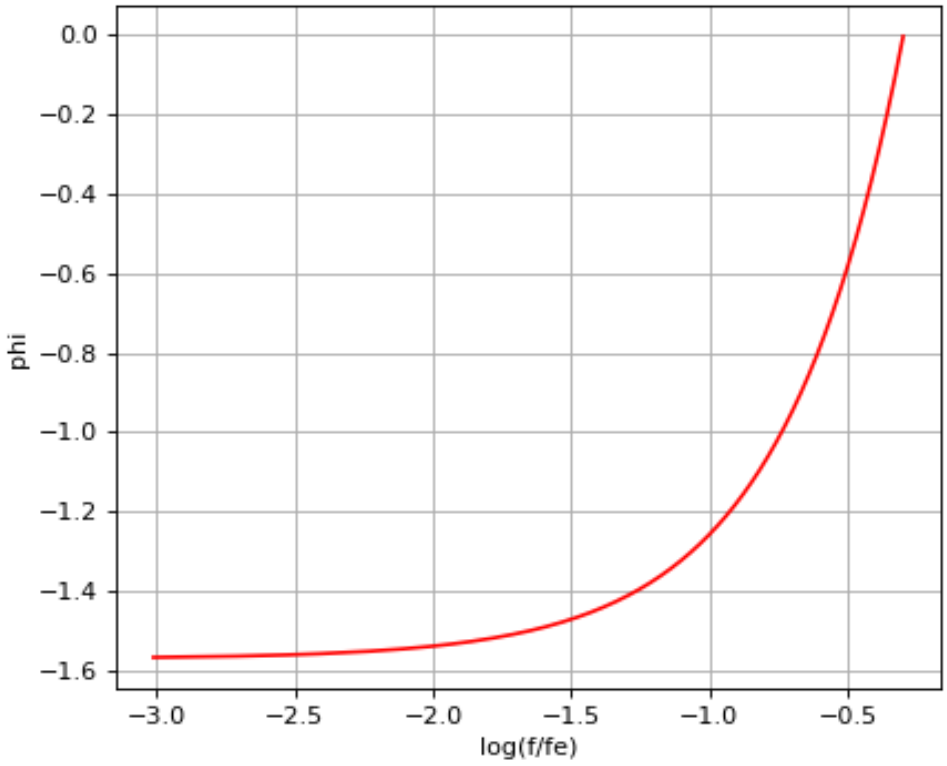
$$H(z) = \frac{b_0 + b_1 z^{-1} + b_2 z^{-2} + \cdots}{a_0 + a_1 z^{-1} + a_2 z^{-2} + \cdots}$$

```
import numpy as np
import math
from matplotlib.pyplot import *
```

```
import scipy.signal
b=[1]
a=[1,-1]
[w,h] = scipy.signal.freqz(b,a)
figure(figsize=(6,5))
plot(np.log10(w/(2*math.pi)),20*np.log10(np.abs(h)),'r')
xlabel('log(f/fe)')
ylabel('GdB')
grid()
```




```
figure(figsize=(6,5))
plot(np.log10(w/(2*math.pi)),np.angle(h),'r')
xlabel('log(f/fe)')
ylabel('phi')
grid()
```



On observe bien une pente de -20 dB par décade caractéristique d'un intégrateur. En revanche, la phase n'est égale à $-\pi/2$ que sur une plage de fréquence très limitée.

Un meilleur filtre est obtenu avec la relation de récurrence suivante ([1]) :

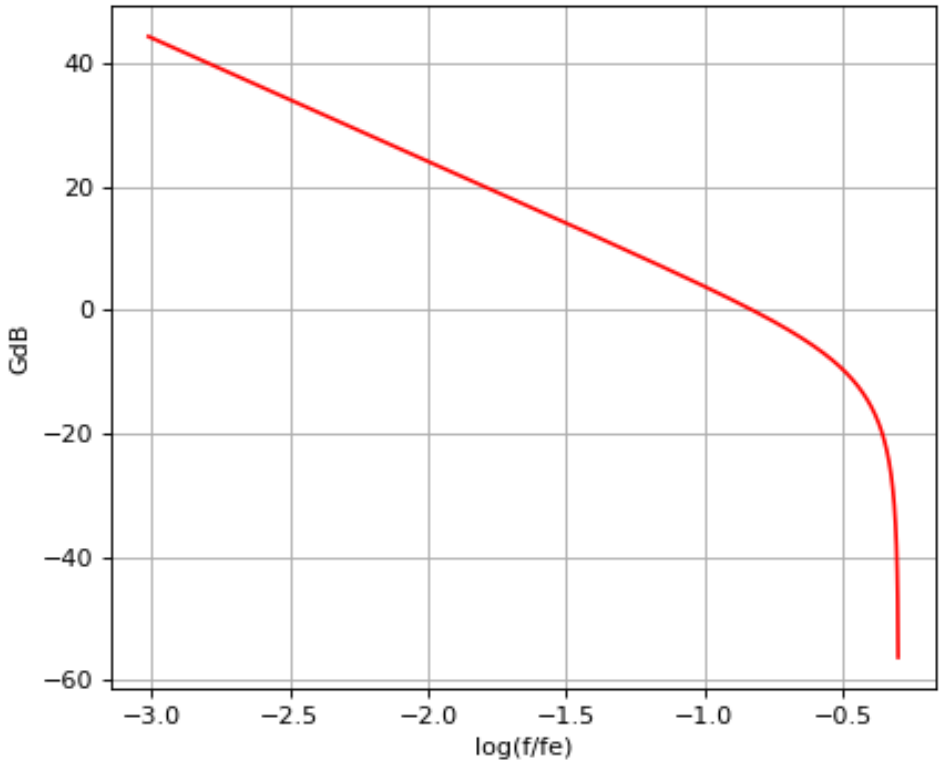
$$y_n = y_{n-1} + T_e \frac{x_n + x_{n-1}}{2}$$

La fonction de transfert en Z est :

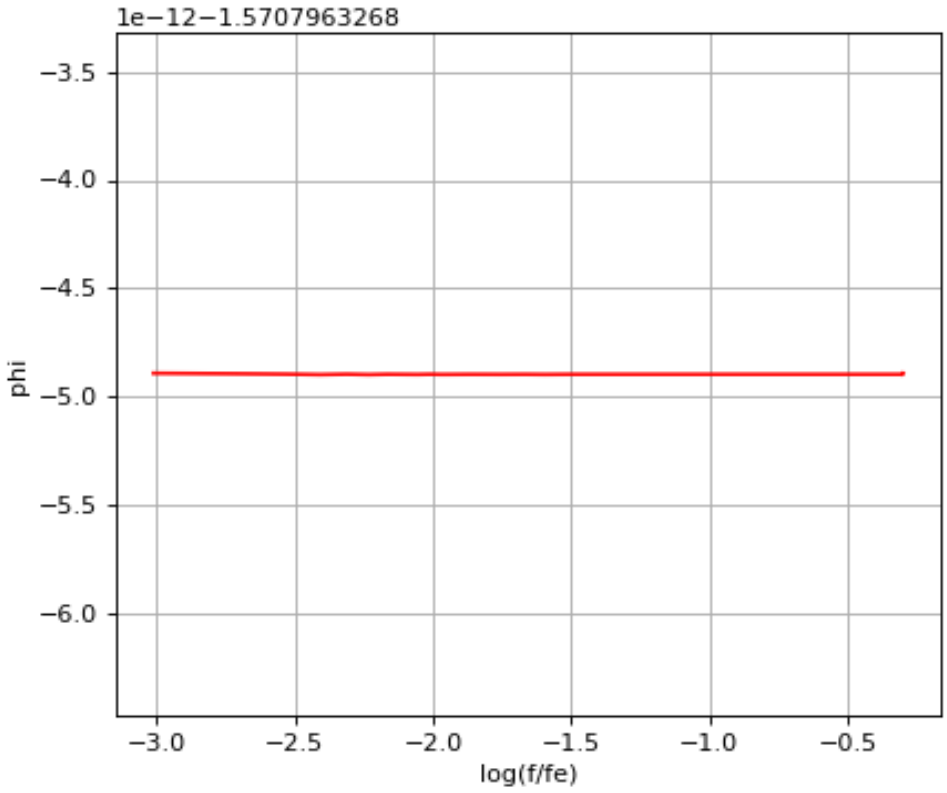
$$H(z) = \frac{T_e}{2} \frac{1 + z^{-1}}{1 - z^{-1}}$$

Voyons sa réponse fréquentielle :

```
b=[0.5,0.5]
a=[1,-1]
[w,h] = scipy.signal.freqz(b,a)
figure(figsize=(6,5))
plot(np.log10(w/(2*math.pi)),20*np.log10(np.abs(h)),'r')
xlabel('log(f/fe)')
ylabel('GdB')
grid()
```

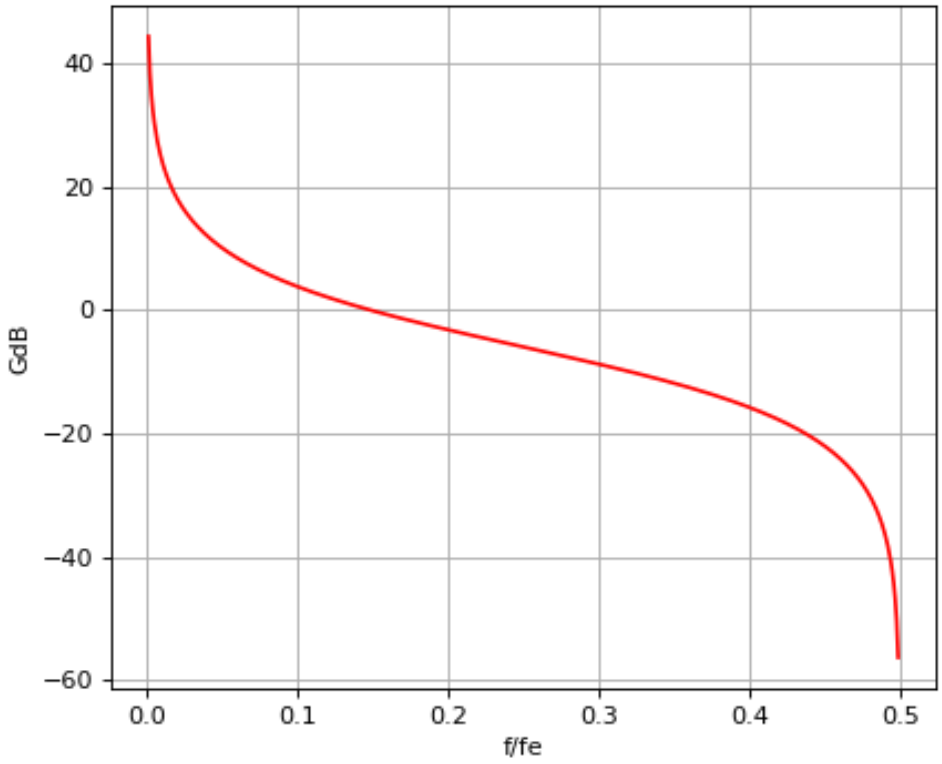



```
figure(figsize=(6,5))
plot(np.log10(w/(2*math.pi)),np.angle(h),'r')
xlabel('log(f/fe)')
ylabel('phi')
grid()
```



La phase est bien constante et égale à $-\pi/2$. Il y a en revanche une chute du gain à l'approche de la fréquence de Nyquist (fe/2). Voyons cela sur un tracé en échelle linéaire :

```
figure(figsize=(6,5))
plot(w/(2*math.pi),20*np.log10(np.abs(h)),'r')
xlabel('f/fe')
ylabel('GdB')
grid()
```



Le comportement intégrateur est valable jusqu'à environ un quart de la fréquence d'échantillonnage. Cela signifie qu'il faut échantillonner au moins à 4 fois la plus grande fréquence présente dans le signal. En pratique, on aura intérêt à effectuer un filtrage passe-bas analogique avant la numérisation pour respecter cette condition. On peut aussi effectuer un filtrage numérique passe-bas après une numérisation avec sur-échantillonnage.

Pour obtenir la réponse impulsionnelle du filtre, il faut tout d'abord décomposer la fonction de transfert en fractions rationnelles simples. La fonction `scipy.signal.residuez` fournit cette décomposition sous la forme suivante :

$$\frac{B(z)}{A(z)} = \frac{r_0}{1 - p_0 z^{-1}} + \frac{r_1}{1 - p_1 z^{-2}} + \cdots + k_0 + k_1 z^{-1} + \cdots$$

```
[r,p,k] = scipy.signal.residuez(b,a)


print(r)
--> array([ 1.])

print(p)
--> array([ 1.])

print(k)
--> array([-0.5])
```

On a donc :

$$H(z) = \frac{T_e z}{z-1} - \frac{T_e}{2}$$

la réponse impulsionnelle est :

$$h_n = T_e u_n - \frac{T_e}{2}\delta_n$$

où u_n est l'échelon unité et δ_n l'impulsion unité.

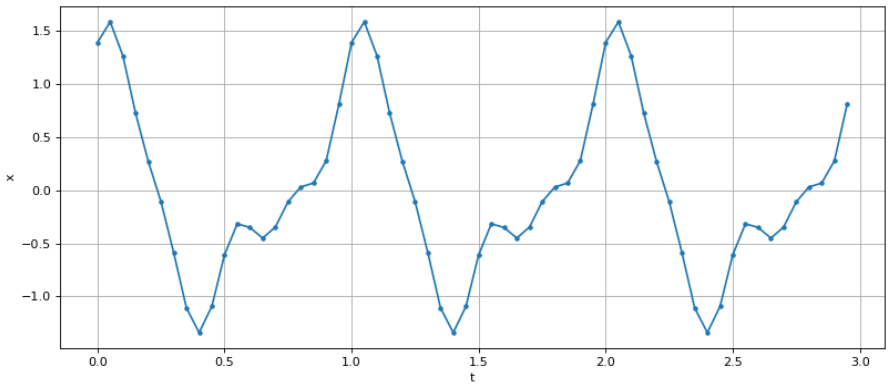
2.b. Exemple

Comme exemple de signal à intégrer, considérons le polynome trigonométrique de fréquence 1 suivant :

```
def signal(t):
    return np.cos(2*math.pi*t)\
    +0.5*np.cos(2*2*math.pi*t-math.pi/3)\
    +0.2*np.cos(4*2*math.pi*t-math.pi/4)
```

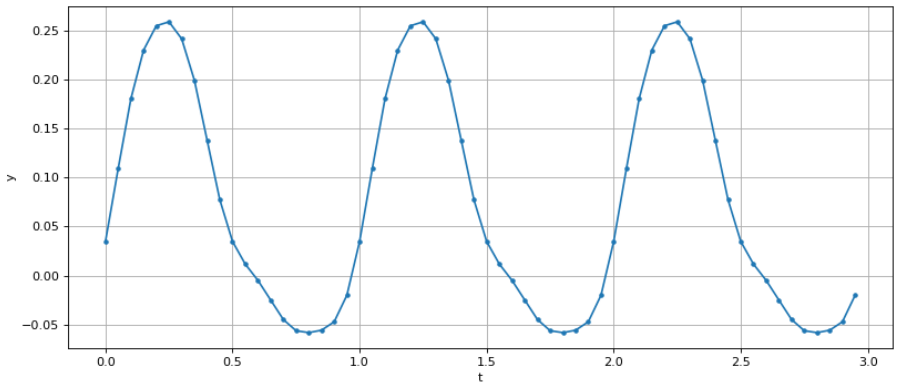
La fréquence maximale est 4. Il faut donc une fréquence d'échantillonnage supérieure à 8 pour éviter le repliement de bande, mais supérieure à 16 pour appliquer le filtre intégrateur :

```
fe = 20.0
te=1.0/fe
t = np.arange(start=0.0,stop=3.0,step=te)
x = signal(t)
figure(figsize=(12,5))
plot(t,x,".-")
xlabel('t')
ylabel('x')
grid()
```



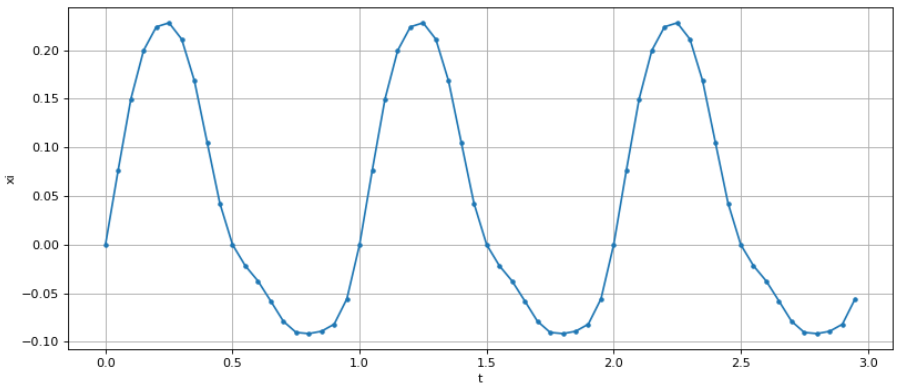
La fonction `scipy.signal.lfilter` permet d'effectuer le filtrage :

```
a=[1.0,-1.0]
b=[0.5*te,0.5*te]
y = scipy.signal.lfilter(b,a,x)
figure(figsize=(12,5))
plot(t,y,".-")
xlabel('t')
ylabel('y')
grid()
```



Pour comparaison, voici les échantillons de la fonction intégrée :

```
def integre(t):
    return np.sin(2*math.pi*t)/(2*math.pi)\
    +0.5*(np.sin(2*2*math.pi*t-math.pi/3)-np.sin(-math.pi/3))/(2*2*math.pi)\
    +0.2*(np.sin(4*2*math.pi*t-math.pi/4)-np.sin(-math.pi/4))/(4*2*math.pi)
xi = integre(t)
figure(figsize=(12,5))
plot(t,xi,".-")
xlabel('t')
ylabel('xi')
grid()
```



La signal fourni par la fonction `lfilter` est décalé (sa valeur moyenne est non nulle). Cela est dû au fait que la première valeur est :

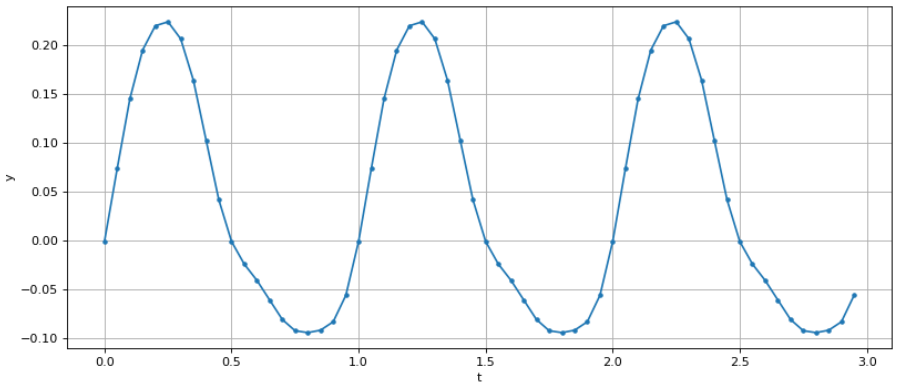
```
print(x[0])
--> 1.3914213562373094
```

alors que celle de l'intégrale est :

```
print(xi[0])
--> 0.0
```

La relation de récurrence nécessite en fait une condition initiale, qui peut être fournie dans la fonction `lfilter` :

```
zi = scipy.signal.lfiltic(b,a,y=[integre(-te)],x=[signal(-te)])
[y,zf] = scipy.signal.lfilter(b,a,x,axis=-1,zi=zi)
figure(figsize=(12,5))
plot(t,y,".-")
xlabel('t')
ylabel('y')
grid()
```



3. Filtre dérivateur

3.a. Définition du filtre

L'opération analogique est :

$$y(t) = \frac{dx(t)}{dt}$$

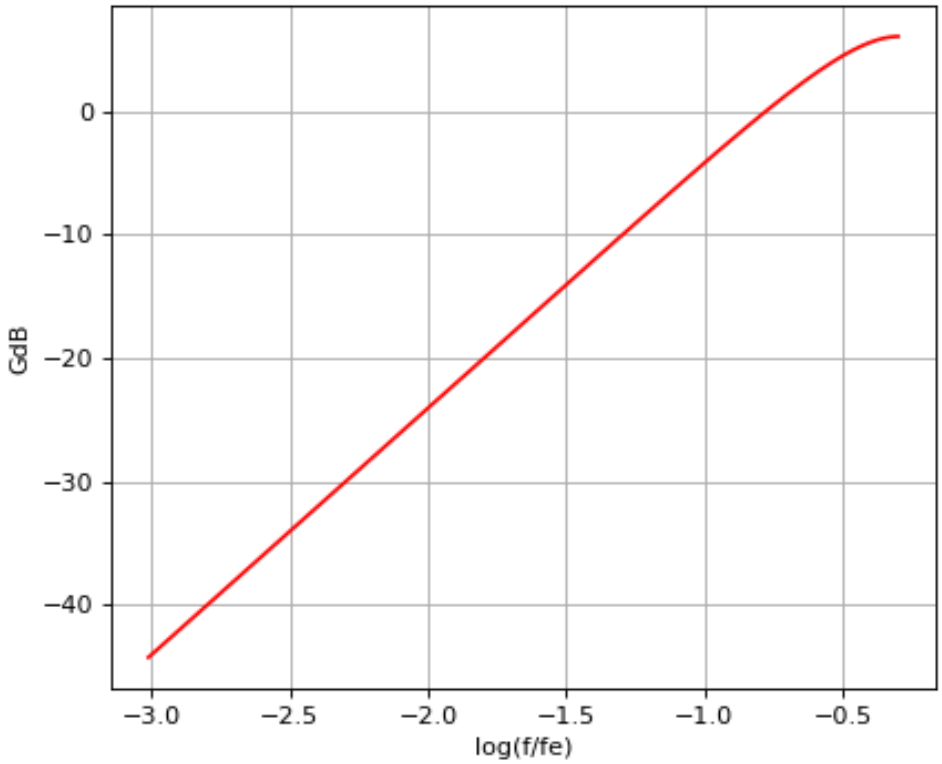
Une première idée consiste à utiliser la formule des accroissements finis suivante pour discrétiser la dérivée :

$$y_n = \frac{1}{T_e}(x_n - x_{n-1})$$

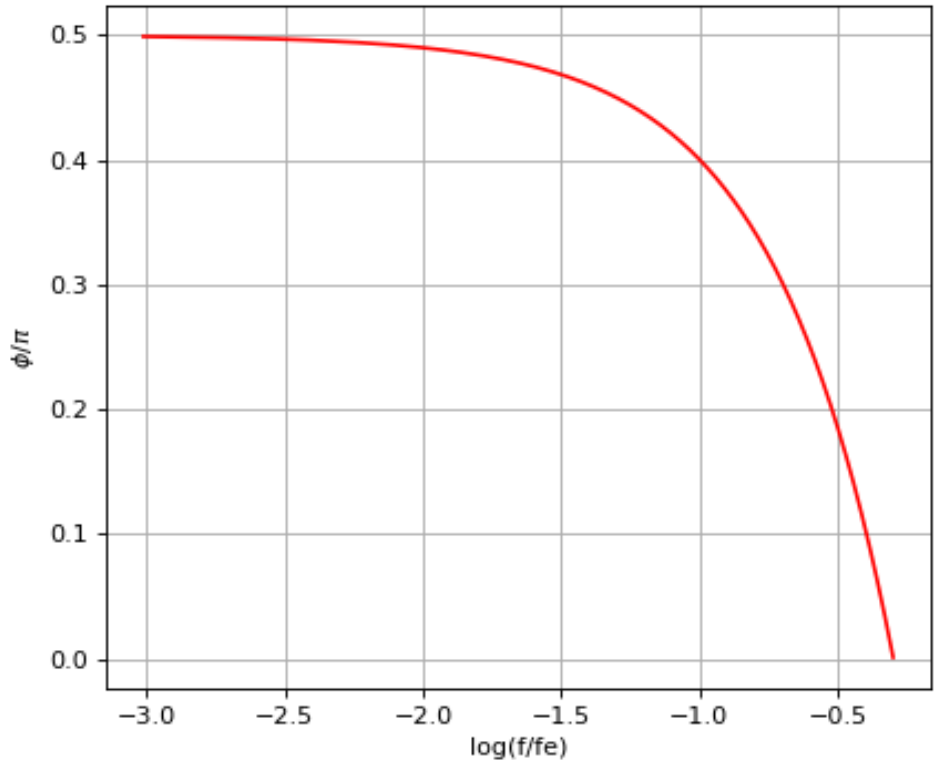
On obtient ainsi un filtre à préponse impulsionnelle finie dont la fonction de transfert est :

$$H(z) = \frac{1 - z^{-1}}{T_e}$$

```
a=[1]
b=[1,-1]
[w,h] = scipy.signal.freqz(b,a)
figure(figsize=(6,5))
plot(np.log10(w/(2*math.pi)),20*np.log10(np.abs(h)),'r')
xlabel('log(f/fe)')
ylabel('GdB')
grid()
```




```
figure(figsize=(6,5))
plot(np.log10(w/(2*math.pi)),np.angle(h)/np.pi,'r')
xlabel('log(f/fe)')
ylabel(r'$\phi/\pi$')
grid()
```



Le gain a bien une pente croissante de +20 dB par décade mais le déphasage n'est pas du tout constant.

Ce filtre n'est utilisable que si la fréquence d'échantillonnage est très grande devant la fréquence du signal. Dans ce cas, un développement limité à l'ordre 1 de la fonction de transfert donne en effet :

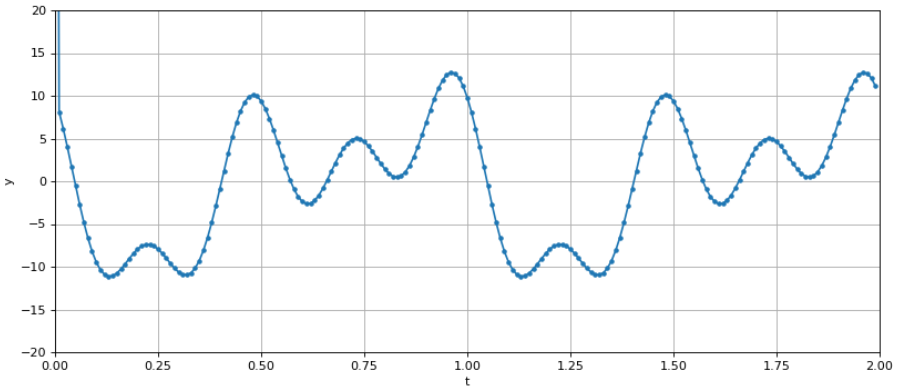
$$H(z) \approx \frac{1}{T_e}\left(1 - 1 + i2\pi T_e f\right) = i2\pi f \tag{1}$$

3.b. Exemple

On peut considérer que le filtre est dérivateur (très approximativement) aux fréquences inférieures au dixième de la fréquence d'échantillonnage. Celle-ci devra donc être au moins 10 fois la fréquence la plus haute du signal (ici 4)

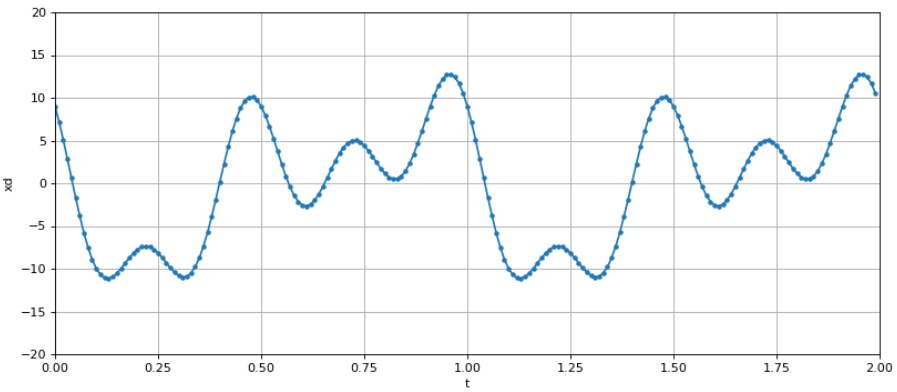
```
fe = 100.0
te=1.0/fe
t = np.arange(start=0.0,stop=2.0,step=te)
x = signal(t)
def derive(t):
    return -np.sin(2*math.pi*t)*2*math.pi\
    -0.5*np.sin(2*2*math.pi*t-math.pi/3)*2*2*math.pi\
    -0.2*np.sin(4*2*math.pi*t-math.pi/4)*4*2*math.pi
a=[te]
b=[1,-1]
y = scipy.signal.lfilter(b,a,x)
```

```
figure(figsize=(12,5))
plot(t,y,".-")
xlabel('t')
ylabel('y')
axis([0,2,-20,20])
grid()
```



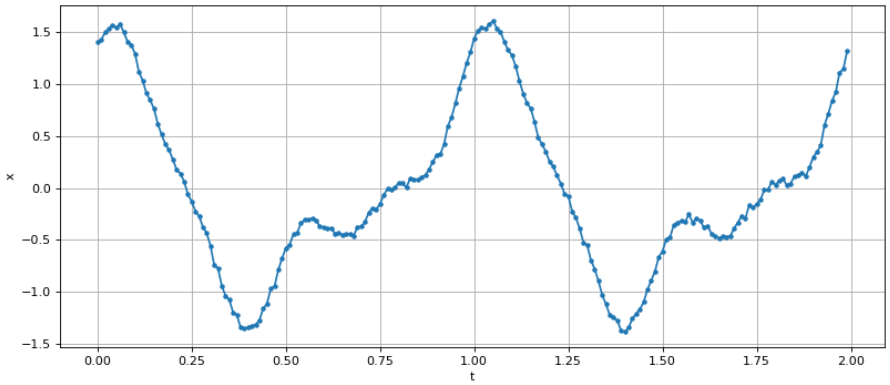
Pour comparaison, voici les échantillons de la fonction dérivée :

```
xd = derive(t)
figure(figsize=(12,5))
plot(t,xd,".-")
xlabel('t')
ylabel('xd')
axis([0,2,-20,20])
grid()
```



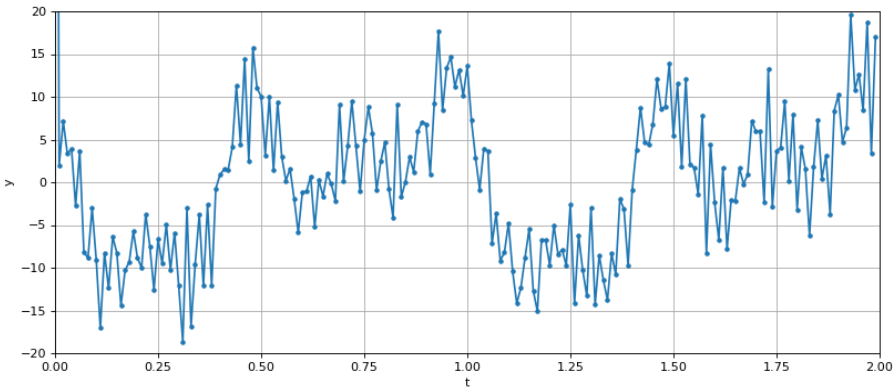
Le filtre dérivateur a un gain proportionnel à la fréquence. Il est donc très sensible au bruit présent dans le signal, particulièrement les parties hautes fréquences du bruit. Pour tester cela, on introduit un bruit aléatoire dans le signal. Dans ce cas, il faut échantillonner la fonction en décrivant explicitement une boucle.

```
import random
def signal(t):
    return np.cos(2*math.pi*t)\
    +0.5*np.cos(2*2*math.pi*t-math.pi/3)\
    +0.2*np.cos(4*2*math.pi*t-math.pi/4)\
    +0.05*random.uniform(-1.0,1.0)
fe = 100.0
te=1.0/fe
t = np.arange(start=0.0,stop=2.0,step=te)
n = t.size
x = np.zeros(n)
for k in range(n):
    x[k] = signal(te*k)
figure(figsize=(12,5))
plot(t,x,".-")
xlabel('t')
ylabel('x')
grid()
```



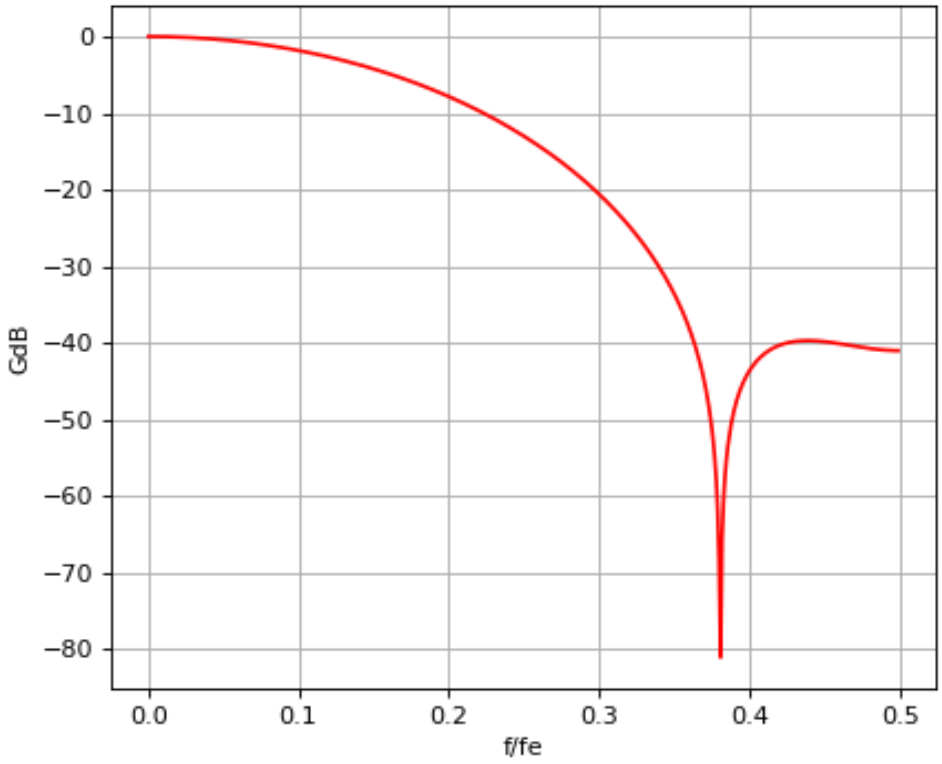
Voici le signal dérivé :

```
y = scipy.signal.lfilter(b,a,x)
figure(figsize=(12,5))
plot(t,y,".-")
xlabel('t')
ylabel('y')
axis([0,2,-20,20])
grid()
```

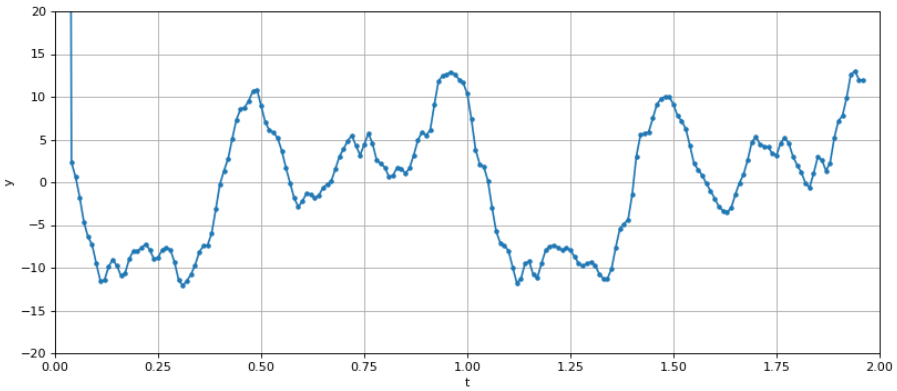


Le signal dérivé comporte un bruit très important, car le filtre dérivateur amplifie les composantes hautes fréquences du bruit au dépend des fréquences utiles. Pour y remédier, on peut appliquer un filtrage passe-bas avant la dérivation, par exemple avec un filtre RIF :

```
P=3
h = scipy.signal.firwin(numtaps=2*P+1,cutoff=[0.1],nyq=0.5,window='hann')
b_pb=h
a_pb=[1.0]
[w,hf] = scipy.signal.freqz(b_pb,a_pb)
figure(figsize=(6,5))
plot(w/(2*math.pi),20*np.log10(np.abs(hf)),'r')
xlabel('f/fe')
ylabel('GdB')
grid()
```



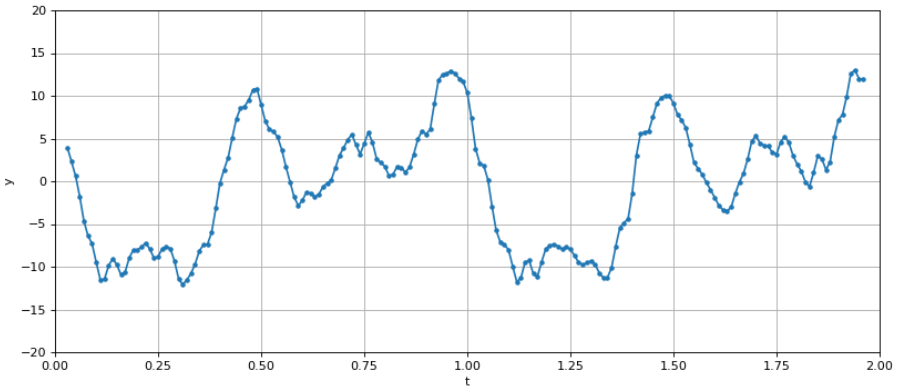

```
print(h)
--> array([ 0.        ,  0.06801965,  0.25223074,  0.35949921,  0.25223074,
        0.06801965,  0.        ])

x1 = scipy.signal.convolve(x,h,mode='valid')
n1 = x1.size
t1 = (np.arange(n1)+P)*te
y = scipy.signal.lfilter(b,a,x1)
figure(figsize=(12,5))
plot(t1,y,".-")
xlabel('t')
ylabel('y')
axis([0,2,-20,20])
grid()
```



Pour un filtre à 7 coefficients, on perd 3 points au début et 3 points à la fin.

Pour plus de rapidité d'exécution du filtrage, on peut composer les deux filtres RIF (filtrage et dérivation) :

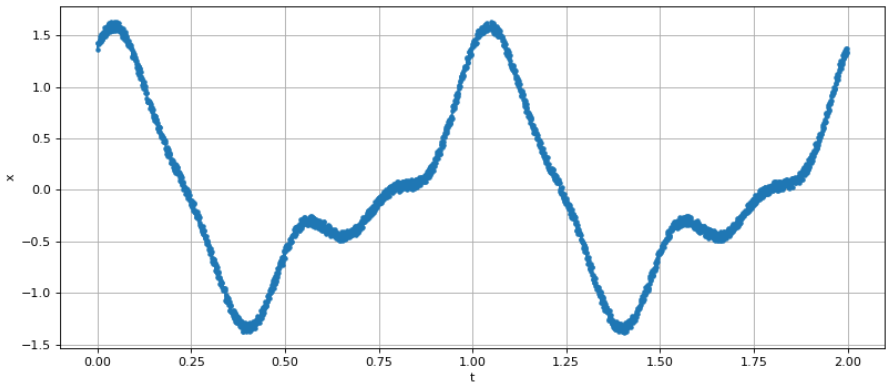
```
hc = np.zeros(h.size)
for k in range(1,h.size):
    hc[k] = (h[k]-h[k-1])/te
y = scipy.signal.convolve(x,hc,mode='valid')
figure(figsize=(12,5))
plot(t1,y,".-")
xlabel('t')
ylabel('y')
axis([0,2,-20,20])
grid()
```



La dérivée est à présent reconnaissable. Le résultat peut être encore amélioré en faisant la numérisation avec un sur-échantillonnage, par exemple avec une fréquence dix fois plus grande que précédemment :

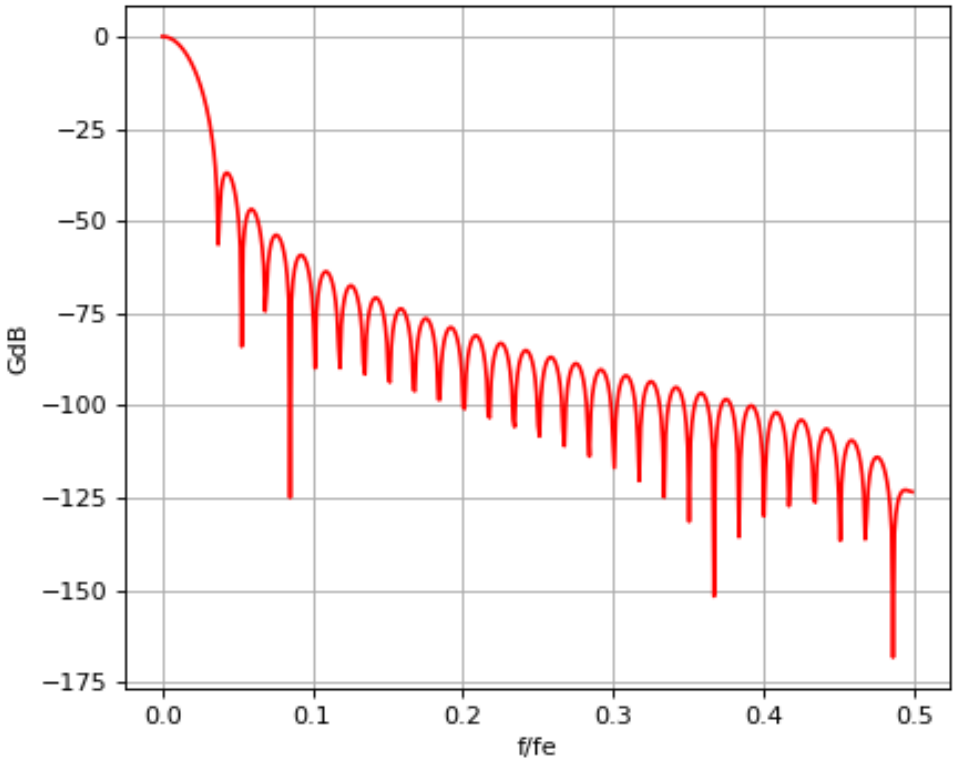
```
fe = 1000.0
te=1.0/fe
t = np.arange(start=0.0,stop=2.0,step=te)
n = t.size
```

```
x = np.zeros(n)
for k in range(n):
    x[k] = signal(te*k)
figure(figsize=(12,5))
plot(t,x,".-")
xlabel('t')
ylabel('x')
grid()
```



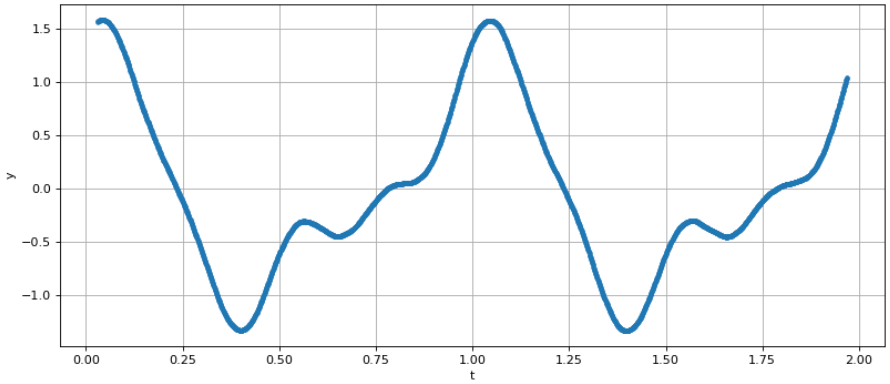
L'étape suivante consiste à appliquer un filtrage passe-bas très sélectif, avec un filtre RIF dont la réponse impulsionnelle a 10 fois plus de coefficients (puisqu'on a augmenté la fréquence d'un facteur 10) :

```
P=30
h = scipy.signal.firwin(numtaps=2*P+1,cutoff=[0.01],nyq=0.5,window='hann')
b_pb=h
a_pb=[1.0]
[w,hf] = scipy.signal.freqz(b_pb,a_pb)
figure(figsize=(6,5))
plot(w/(2*math.pi),20*np.log10(np.abs(hf)),'r')
xlabel('f/fe')
ylabel('GdB')
grid()
```



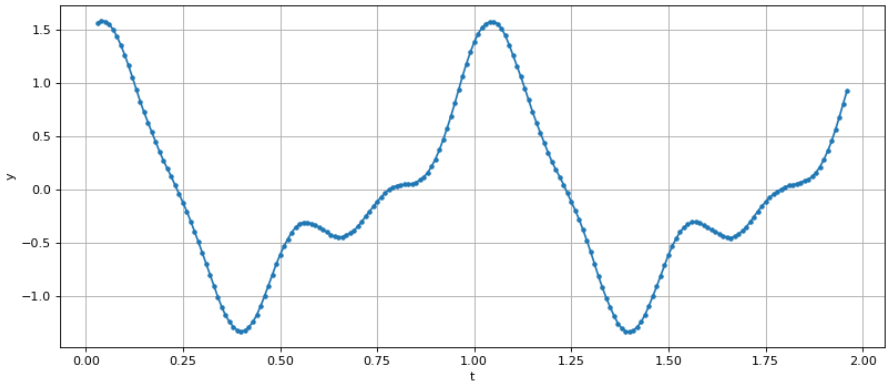
Voici le résultat du filtrage :

```
x1 = scipy.signal.convolve(x,h,mode='valid')
n1 = x1.size
t1 = (np.arange(n1)+P)*te
figure(figsize=(12,5))
plot(t1,x1,".-")
xlabel('t')
ylabel('y')
grid()
```



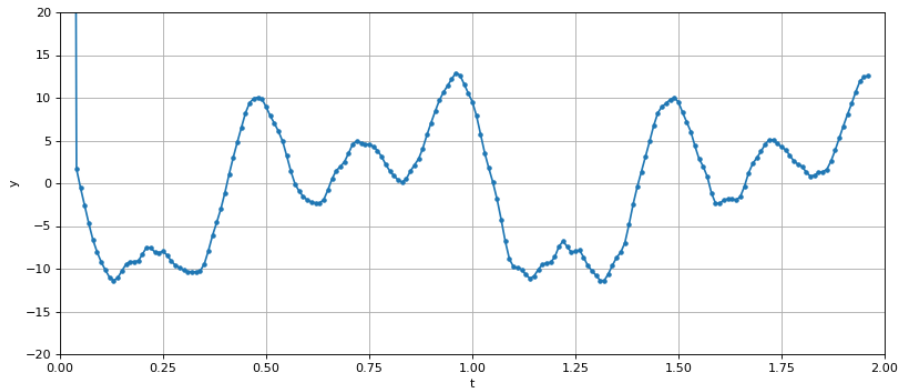
On peut alors réduire la fréquence d'échantillonnage d'un facteur 10 :

```
x2 = x1[0::10]
t2 = t1[0::10]
figure(figsize=(12,5))
plot(t2,x2,".-")
xlabel('t')
ylabel('y')
grid()
```



puis enfin appliquer la dérivation :

```
y2 = scipy.signal.lfilter(b,a,x2)
figure(figsize=(12,5))
plot(t2,y2,".-")
xlabel('t')
ylabel('y')
axis([0,2,-20,20])
grid()
```



Dans cette simulation, on a introduit volontairement du bruit. En pratique, les signaux numériques comportent toujours du bruit, au minimum le bruit de quantification.

Conclusion : pour faire un bon filtre dérivateur, il faut appliquer la technique du sur-échantillonnage : échantillonner 10 fois plus vite, appliquer un filtre passe-bas très sélectif,

puis réduire la fréquence d'échantillonnage avant de dériver. D'une manière générale, la méthode du sur-échantillonnage est un bon moyen de réduire le bruit dans un signal, en particulier le bruit de quantification.

Références

[1] Tan Li, Jiang Jean, *Digital signal processing : fundamentals and applications*, (Elsevier, 2013)