Bruit de quantification

1. Introduction

On s'intéresse à la mise en évidence du bruit de quantification, observé lorsqu'on numérise un signal avec une profondeur variable (9,10 ou 12 bits). On fera une analyse spectrale du bruit de quantification puis on verra comment le réduire par une méthode de sur-échantillonnage avec filtrage, une opération importante pour la réalisation d'un filtre dérivateur.

2. Numérisation et spectre

On effectue la numérisation d'un signal sinusoïdal avec la carte d'acquisition SysamSP5. La sinusoïde est délivrée par un générateur de fonctions (GBF). Sa fréquence est d'environ 100 Hz.

La carte SysamSP5 a une profondeur de numérisation de 12 bits. Le module `pycanum` permet de réduire la profondeur de numérisation (par exemple à 8 bits), depuis la version 4.2.5.

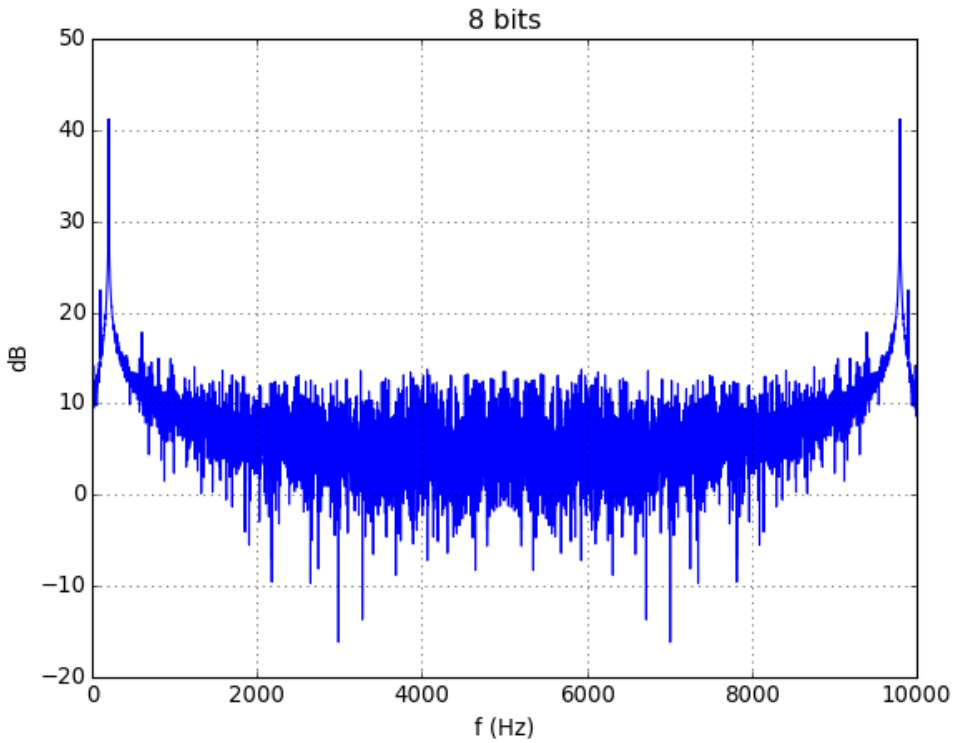
On se fixe comme objectif une bande passante de 4 kHz. On procède à deux fréquences d'échantillonnage, 10 et 100 kHz. La durée de l'acquisition est de 1 seconde, de manière à obtenir une précision de 1 Hz sur les spectres.

Voici le script utilisé pour l'acquisition :

```
# -*- coding: utf-8 -*-

import numpy
from matplotlib.pyplot import *
import pycanum.main as pycan

sys = pycan.Sysam("SP5")
sys.config_entrees([0],[5])
fe=10000.0
te=1.0/fe
duree=1.0
Ne=int(duree*fe)
sys.config_echantillon(te*1e6,Ne,quantification=10)
sys.acquerir()
tensions = sys.entrees()
temps = sys.temps()
u0 = tensions[0]
t = temps[0]
sys.fermer()
numpy.savetxt("sinus100Hz-fe10kHz-12bits.txt",[t,u0])
```

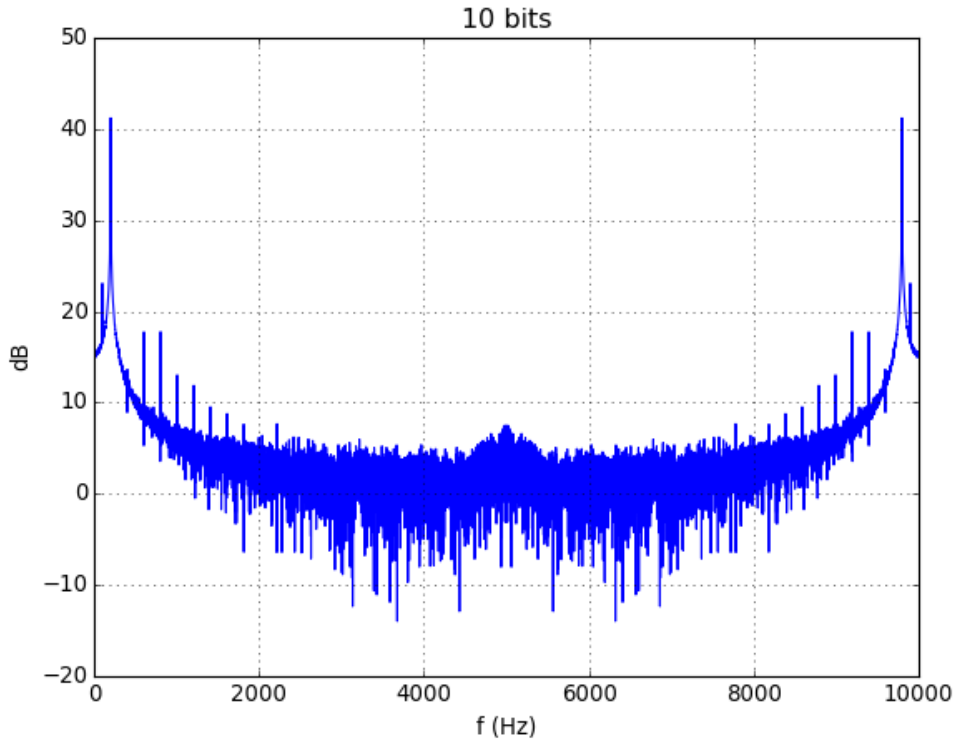
Voici, pour une fréquence d'échantillonnage de 10 kHz, les spectres de puissance pour les numérisations à 8,10 et 12 bits :

```
import numpy
from matplotlib.pyplot import *
import numpy.fft
import scipy.signal

[t,u] = numpy.loadtxt("sinus100Hz-fe10kHz-8bits.txt2")
Ne=len(t)
te=t[1]-t[0]
fe = 1.0/te
tfd = numpy.fft.fft(u*u)
freq = numpy.arange(Ne)*1.0/(Ne*te)
figure()
plot(freq,10*numpy.log10(numpy.absolute(tfd)))
xlabel("f (Hz)")
ylabel("dB")
title("8 bits")
grid()
```

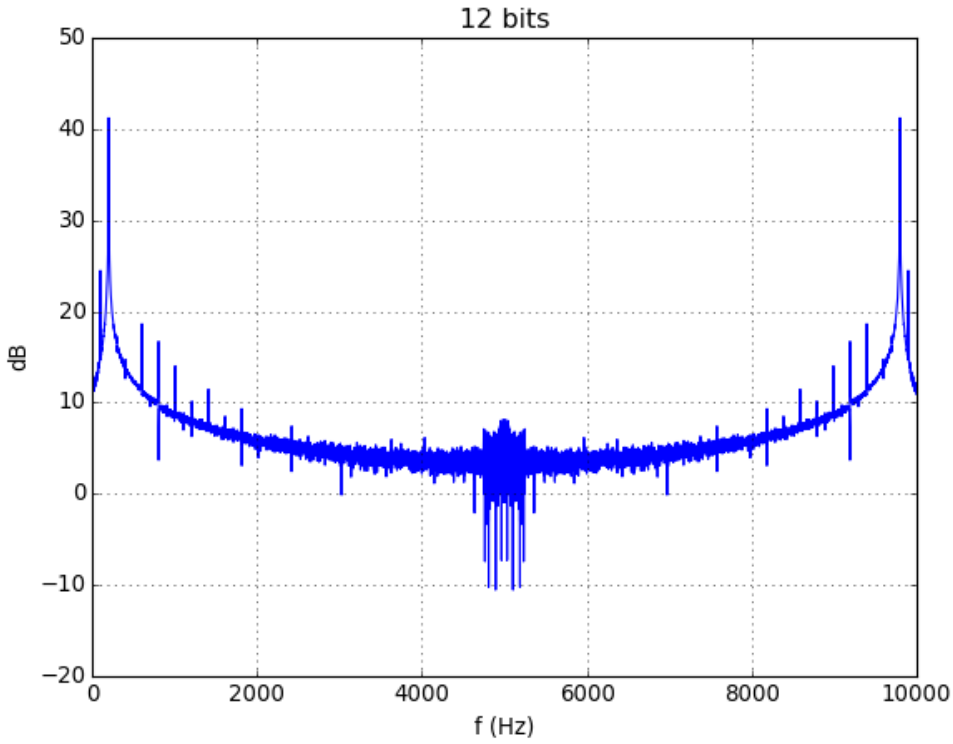



```
[t,u] = numpy.loadtxt("sinus100Hz-fe10kHz-10bits.txt2")
tfd = numpy.fft.fft(u*u)
freq = numpy.arange(Ne)*1.0/(Ne*te)
figure()
plot(freq,10*numpy.log10(numpy.absolute(tfd)))
xlabel("f (Hz)")
ylabel("dB")
```

```
title("10 bits")
grid()
```




```
[t,u] = numpy.loadtxt("sinus100Hz-fe10kHz-12bits.txt2")
tfd = numpy.fft.fft(u*u)
freq = numpy.arange(Ne)*1.0/(Ne*te)
figure()
plot(freq,10*numpy.log10(numpy.absolute(tfd)))
xlabel("f (Hz)")
ylabel("dB")
title("12 bits")
grid()
```

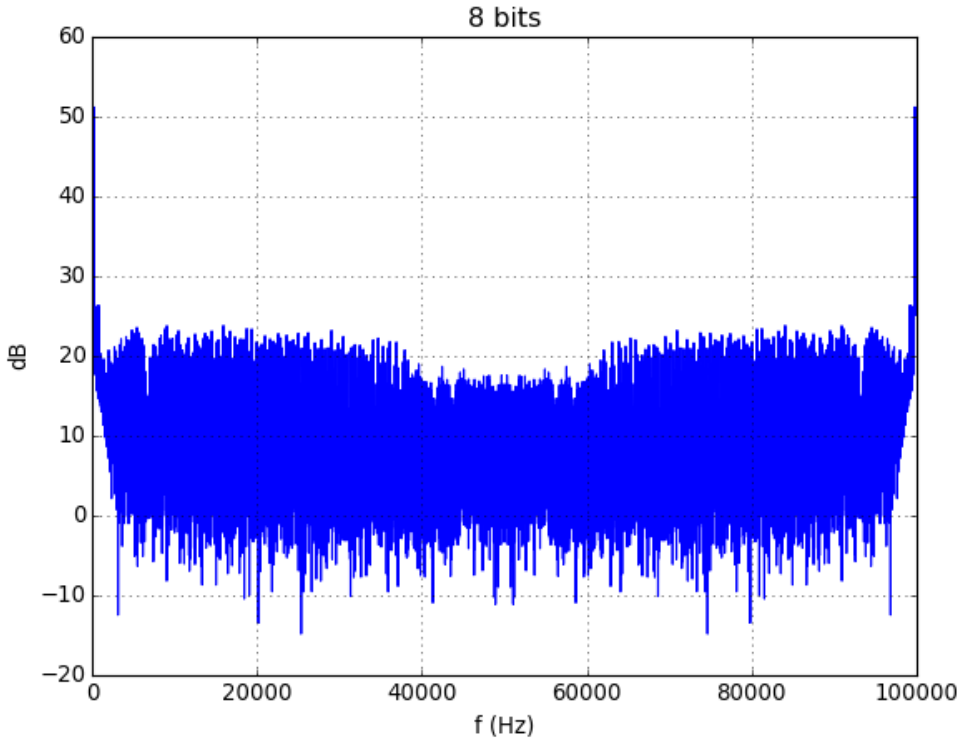


On observe la réduction du bruit de quantification lorsque la profondeur de numérisation augmente. Théoriquement, si l'on note q l'échelon de quantification, le bruit de quantification a une puissance égale à $q^2/12$. En passant de 8 bits à 10 bits, l'échelon est réduit d'un facteur 4, soit une réduction d'un facteur 16 de la puissance, ou bien -12 dB. Pour la numérisation à 12 bits, il est probable que le bruit résiduel soit essentiellement intrinsèque au signal analogique, et non pas seulement du bruit de quantification.

3. Sur-échantillonnage et filtrage passe-bas

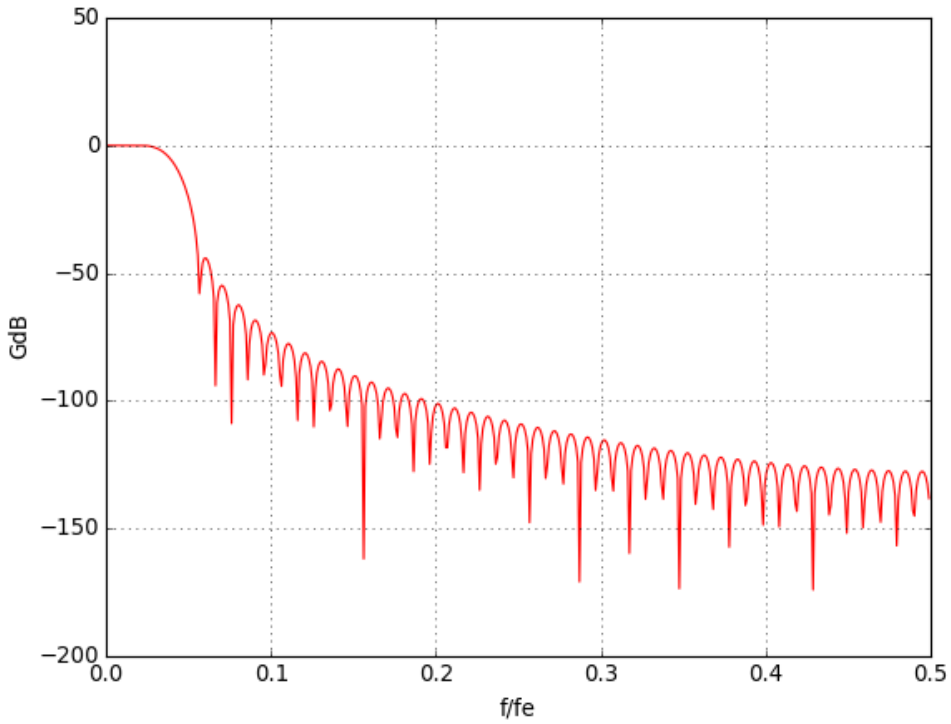
Une technique de réduction du bruit de quantification consiste à sur-échantillonner le signal avant de lui appliquer un filtrage passe-bas. Voici tout d'abord le spectre du signal échantillonné à 100 kHz en 8 bits :

```
[t,u] = numpy.loadtxt("sinus100Hz-fe100kHz-8bits.txt2")
Ne=len(t)
te=t[1]-t[0]
fe = 1.0/te
tfd = numpy.fft.fft(u*u)
freq = numpy.arange(Ne)*1.0/(Ne*te)
figure()
plot(freq,10*numpy.log10(numpy.absolute(tfd)))
xlabel("f (Hz)")
ylabel("dB")
title("8 bits")
grid()
```



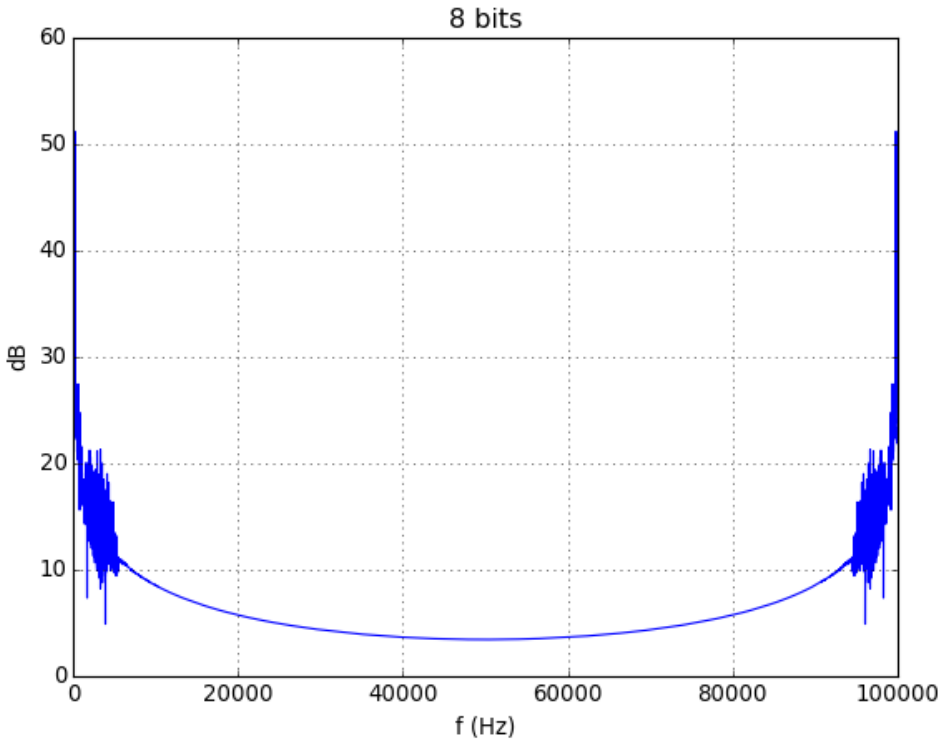
On calcule un filtre RIF passe-bas pour filtrer ce signal. La fréquence de coupure est choisie légèrement inférieure à la moitié de la fréquence d'échantillonnage finale ($10kHz/2$), de manière à éviter tout repliement lors de la réduction.

```
fc = 4.0e3
b = scipy.signal.firwin(numtaps=100,cutoff=[fc/fe],window='hann',nyq=0.5)
[w,h] = scipy.signal.freqz(b,[1])
figure()
plot(w/(2*numpy.pi),20*numpy.log10(numpy.abs(h)),'r')
xlabel('f/fe')
ylabel('GdB')
grid()
```



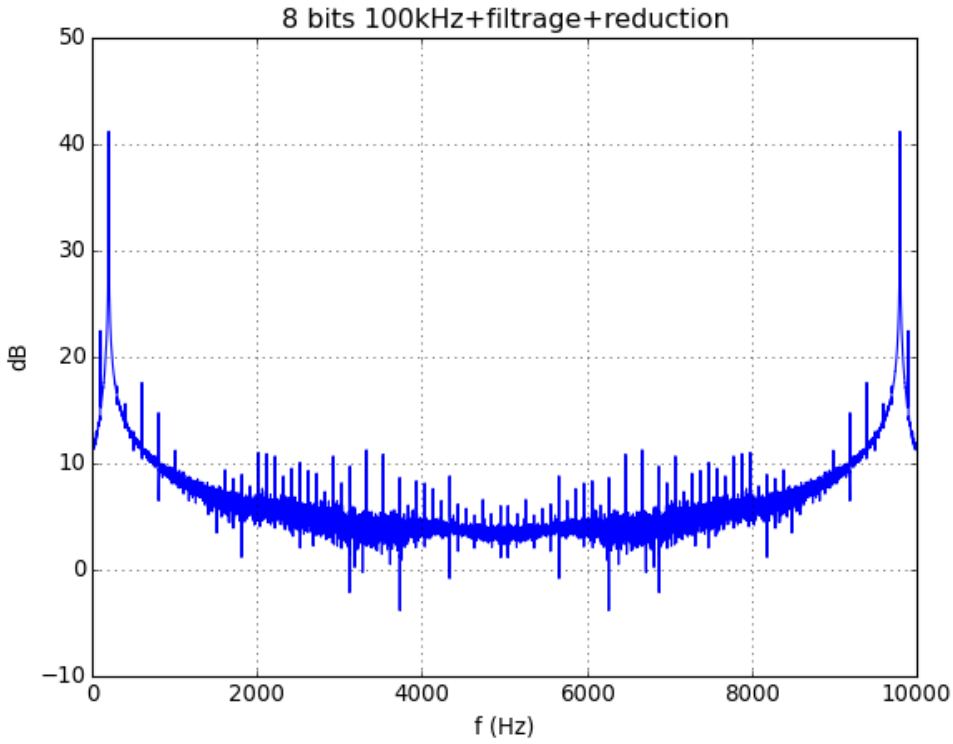
On filtre le signal et on trace le spectre du signal résultant :

```
u1 = scipy.signal.lfilter(b,[1],u)
tfd = numpy.fft.fft(u1*u1)
freq = numpy.arange(Ne)*1.0/(Ne*te)
figure()
plot(freq,10*numpy.log10(numpy.absolute(tfd)))
xlabel("f (Hz)")
ylabel("dB")
title("8 bits")
grid()
```

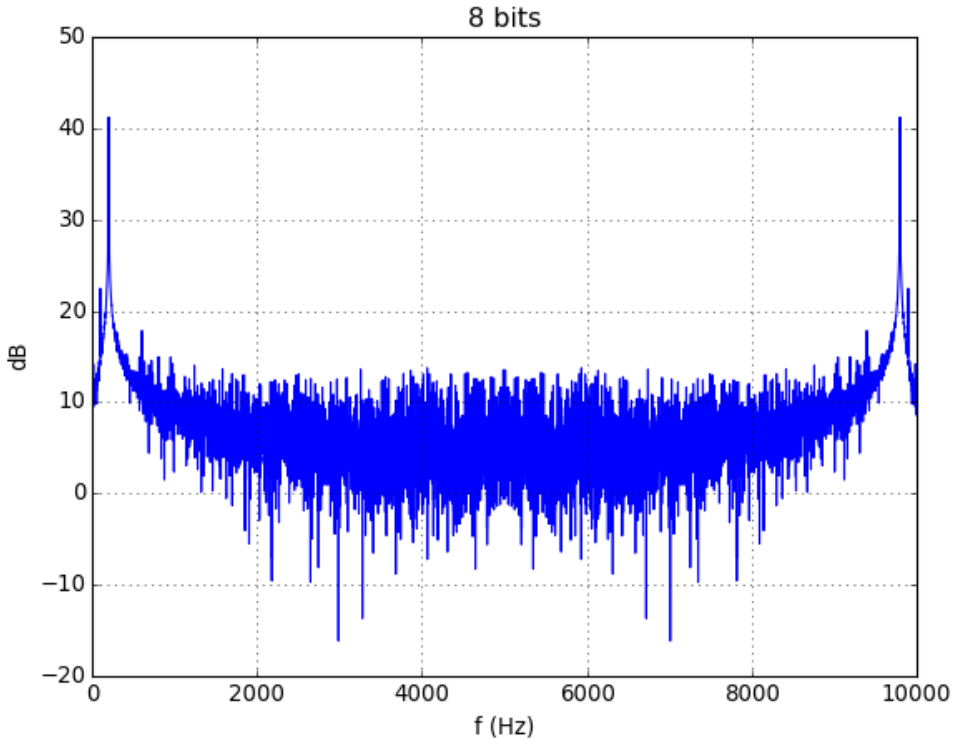


On peut à présent réduire la fréquence d'échantillonnage d'un facteur 10 :

```
u2 = u1[0::10]
fe2 = fe/10
Ne2 = len(u2)
tfd = numpy.fft.fft(u2*u2)
freq = numpy.arange(Ne2)*1.0/(Ne2/fe2)
figure()
plot(freq,10*numpy.log10(numpy.absolute(tfd)))
xlabel("f (Hz)")
ylabel("dB")
title("8 bits 100kHz+filtrage+reduction")
grid()
```



Pour comparaison, voici à nouveau le spectre du signal numérisé à 10 kHz :



4. Application : filtre dérivateur

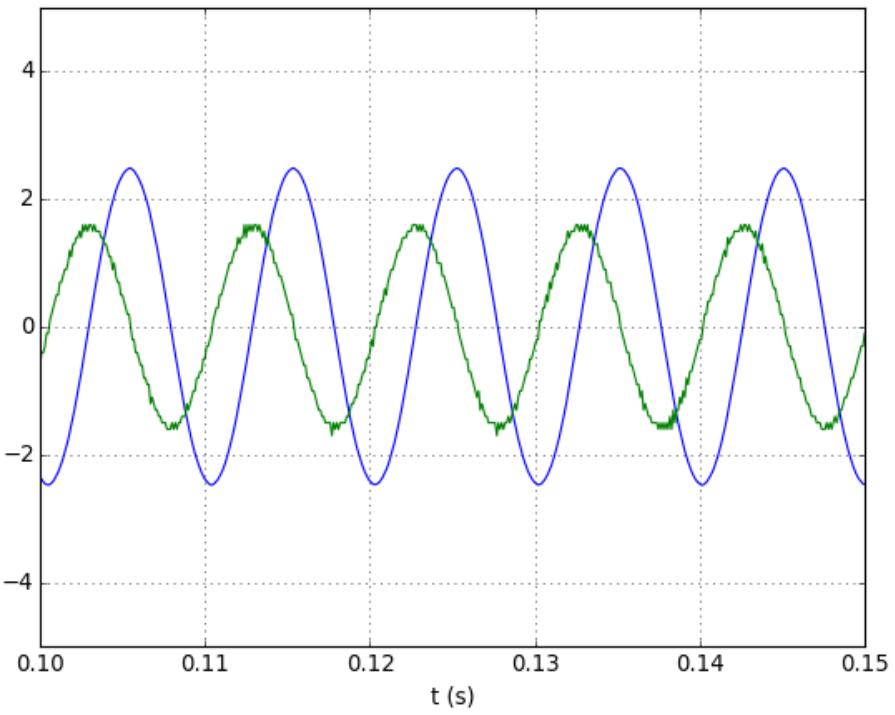
Pour observer l'effet du bruit dans le domaine temporel, un bon exemple est le filtre dérivateur, qui est très sensible au bruit. Pour la définition du filtre dérivateur, voir Filtres intégrateur et dérivateur.

Voici tout d'abord une fonction qui calcule la dérivée d'un signal :

```
def derive(x,te):
    a=[te]
    b=[1,-1]
    return scipy.signal.lfilter(b,a,x)
```

On dérive le signal numérisé à 10 kHz en 10 bits :

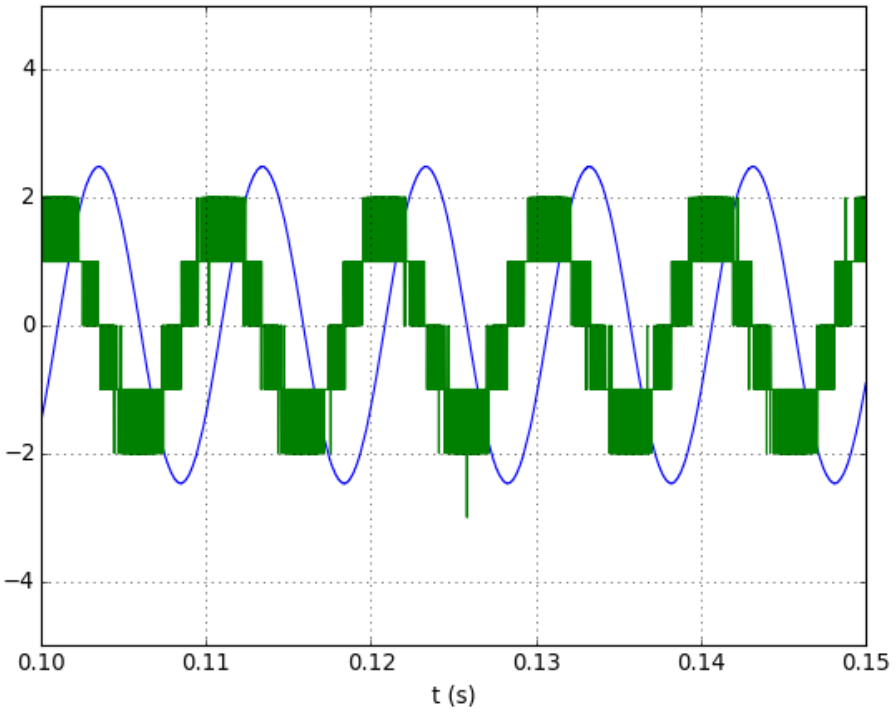
```
[t,x] =  numpy.loadtxt("sinus100Hz-fe10kHz-10bits.txt2")
y = derive(x,t[1]-t[0])
figure()
plot(t,x,label="x")
plot(t,y/1000,label="dx/dt")
xlabel("t (s)")
grid()
axis([0.1,0.15,-5,5])
```



Le bruit est imperceptible sur le signal de départ, mais bien visible sur la dérivée. Son aspect crénelé montre qu'il s'agit bien d'un bruit de quantification.

Voici le même calcul fait avec le signal numérisé à 100 kHz en 10 bits :

```
[t,x] =  numpy.loadtxt("sinus100Hz-fe100kHz-10bits.txt2")
y = derive(x,t[1]-t[0])
figure()
plot(t,x,label="x")
plot(t,y/1000,label="dx/dt")
xlabel("t (s)")
grid()
axis([0.1,0.15,-5,5])
```

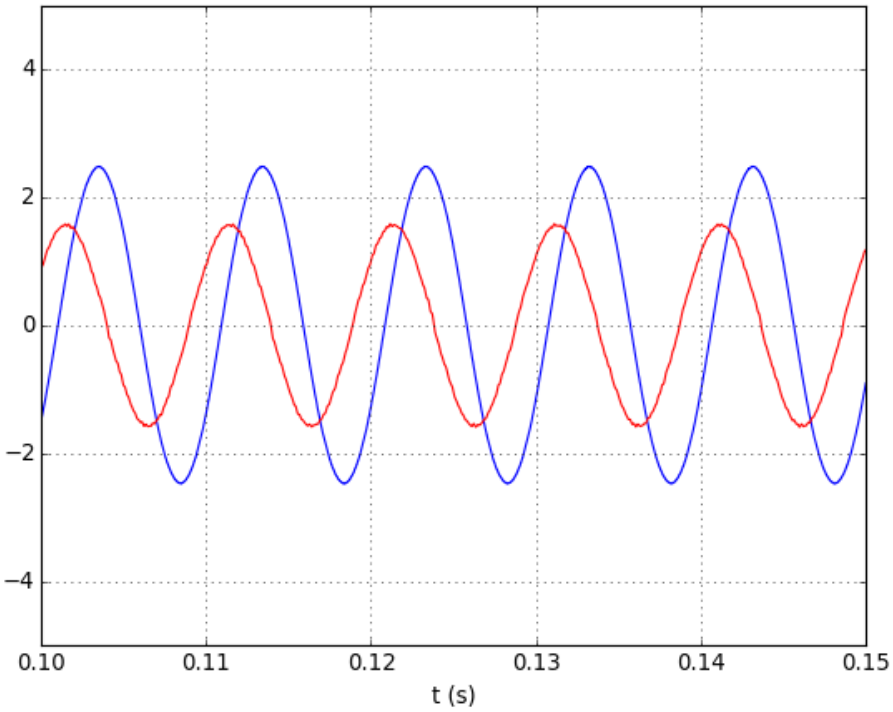


On voit que la dérivation du signal échantillonné à haute fréquence n'est pas une bonne idée. En effet, la fréquence du bruit augmente et l'opération de dérivation fait augmenter les composantes de haute fréquence au dépend des composantes de basse fréquence utiles (ici la sinusoïde). Le résultat est un bruit de quantification prépondérant.

La solution consiste à appliquer tout d'abord un filtre passe-bas au signal. On prévoit de ramener la fréquence d'échantillonnage à 10 kHz après le filtrage, donc la fréquence de coupure doit être inférieure à 5 kHz. On garde une marge car le filtre n'est pas parfait, par exemple 4 kHz :

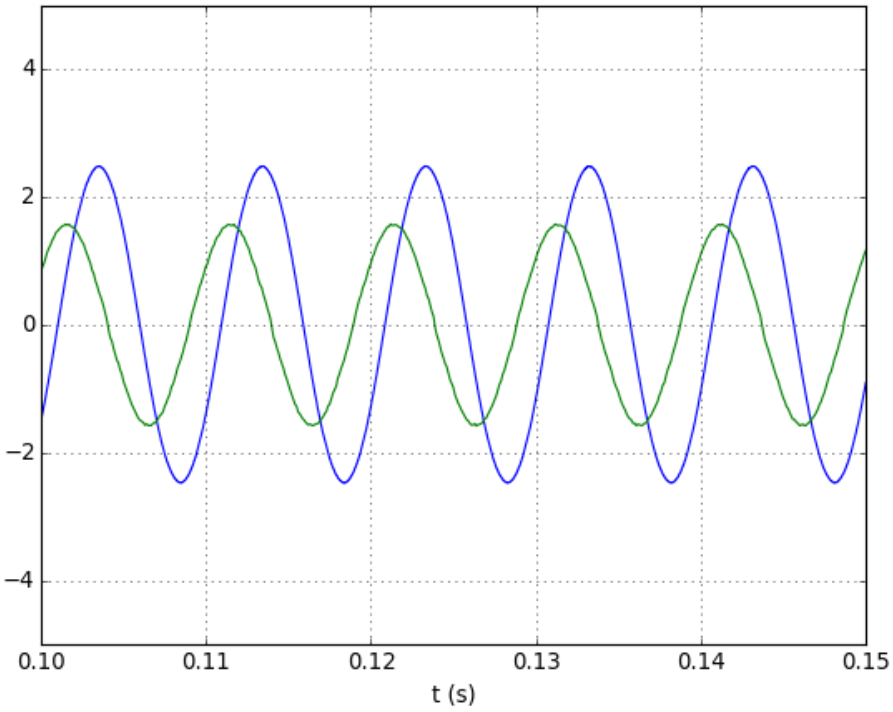
```
fc = 4.0e3
fe = 1e5
b = scipy.signal.firwin(numtaps=100,cutoff=[fc/fe],window='hann',nyq=0.5)
y = scipy.signal.lfilter(b,[1],x)
y1 = derive(y,t[1]-t[0])
figure()
plot(t,x,'b',label="x")
plot(t,y1/1000,'r',label="dx/dt")
```

```
xlabel("t (s)")
grid()
axis([0.1,0.15,-5,5])
```



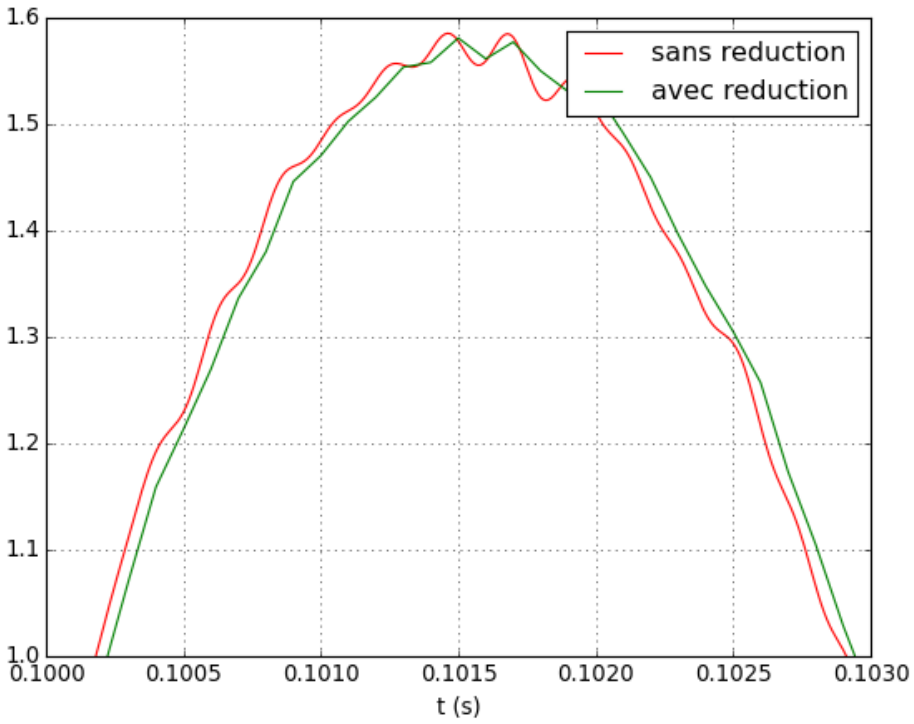
Pour stocker le signal, on a intérêt à réduire la fréquence d'échantillonnage après le filtrage, avant d'appliquer la dérivation :

```
y2 = y[0::10]
t2 = t[0::10]
y3 = derive(y2,t2[1]-t2[0])
figure()
plot(t,x,'b',label="x")
plot(t2,y3/1000,'g',label="dx/dt")
xlabel("t (s)")
grid()
axis([0.1,0.15,-5,5])
```



Il est intéressant de comparer les deux dérivées, celle obtenue sans la réduction avec celle obtenue avec réduction :

```
figure()
plot(t,y1/1000,'r',label="sans reduction")
plot(t2,y3/1000,'g',label="avec reduction")
xlabel("t (s)")
legend(loc="upper right")
grid()
axis([0.1,0.103,1,1.6])
```



On voit qu'il est préférable de dériver après avoir réduit la fréquence d'échantillonnage. En effet, le filtrage n'enlève pas complètement les composantes fréquentielles au dessus de 5000 Hz, et celles-ci sont amplifiées par la dérivation.

Qu'en est-t-il s'il n'est pas possible de sur-échantillonner ? On peut bien sûr appliquer un filtre passe-bas au signal échantillonné à 10 kHz, avec la même fréquence de coupure que ci-dessus (4 kHz) :

```
[t,x] =  numpy.loadtxt("sinus100Hz-fe10kHz-10bits.txt2")
fc = 4.0e3
fe = 1e4
b = scipy.signal.firwin(numtaps=100,cutoff=[fc/fe],window='hann',nyq=0.5)
y = scipy.signal.lfilter(b,[1],x)
y1 = derive(y,t[1]-t[0])
figure()
plot(t,y1/1000,'r',label="dx/dt")
xlabel("t (s)")
grid()
axis([0.106,0.11,1,1.6])
```

Pour la même bande passante finale (4 kHz), la méthode du sur-échantillonnage avec réduction puis dérivation fournit un bruit plus faible. Cela peut surprendre, puisque la dérivation appliquée au signal sur-échantillonné donne au contraire un très mauvais résultat. Pour le comprendre, il faut tenir compte du fait que le bruit (de quantification) a un spectre qui s'étend jusqu'à la fréquence d'échantillonnage, c'est-à-dire qu'il est sujet au repliement. En sur-échantillonnant, on réduit le repliement du bruit dans la bande utile.