Intégrateurs analogique et numérique

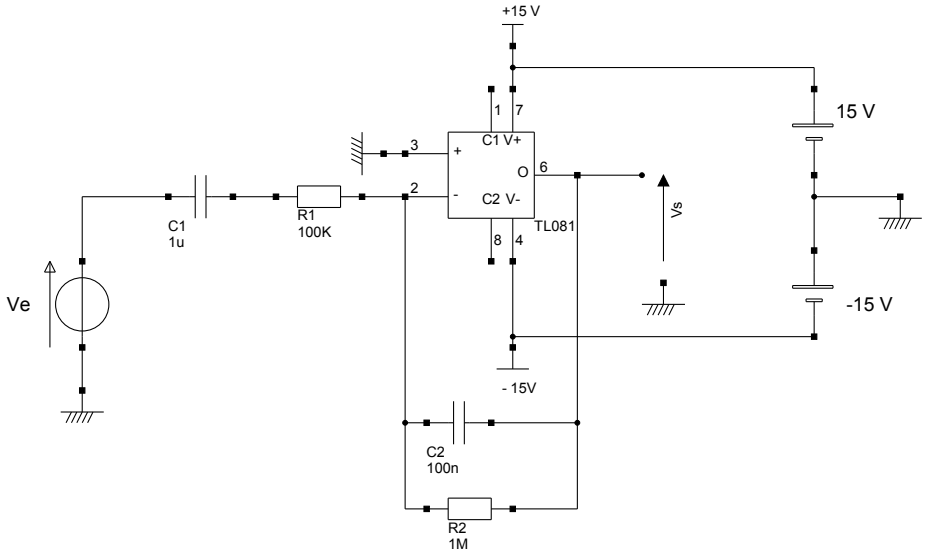
1. Introduction

On se propose de réaliser un filtre intégrateur, qui pourrait servir à intégrer le signal délivré par un capteur de vitesse ou par un accéléromètre. Il devra intégrer des signaux dont la fréquence d'étend d'environ 1 Hz à 100 Hz.

2. Intégrateur analogique

2.a. Circuit intégrateur

Voici le premier circuit utilisé (intégrateur 1), comportant un ALI TL081 à entrées JFET :



La fonction de transfert de ce filtre est :

$$H(\omega) = H_1(\omega)H_2(\omega) \tag{1}$$

$$H_1(\omega) = \frac{jR_1C_1\omega}{1 + jR_1C_1\omega} \tag{2}$$

$$H_2(\omega) = \frac{-R_2/R_1}{1 + jR_2C_2\omega} \tag{3}$$

Il s'agit donc du produit d'un filtre passe-haut H_1 du premier ordre par un filtre passe-bas H_2 du premier ordre. L'intégration est réalisée par le filtre passe-bas lorsque :

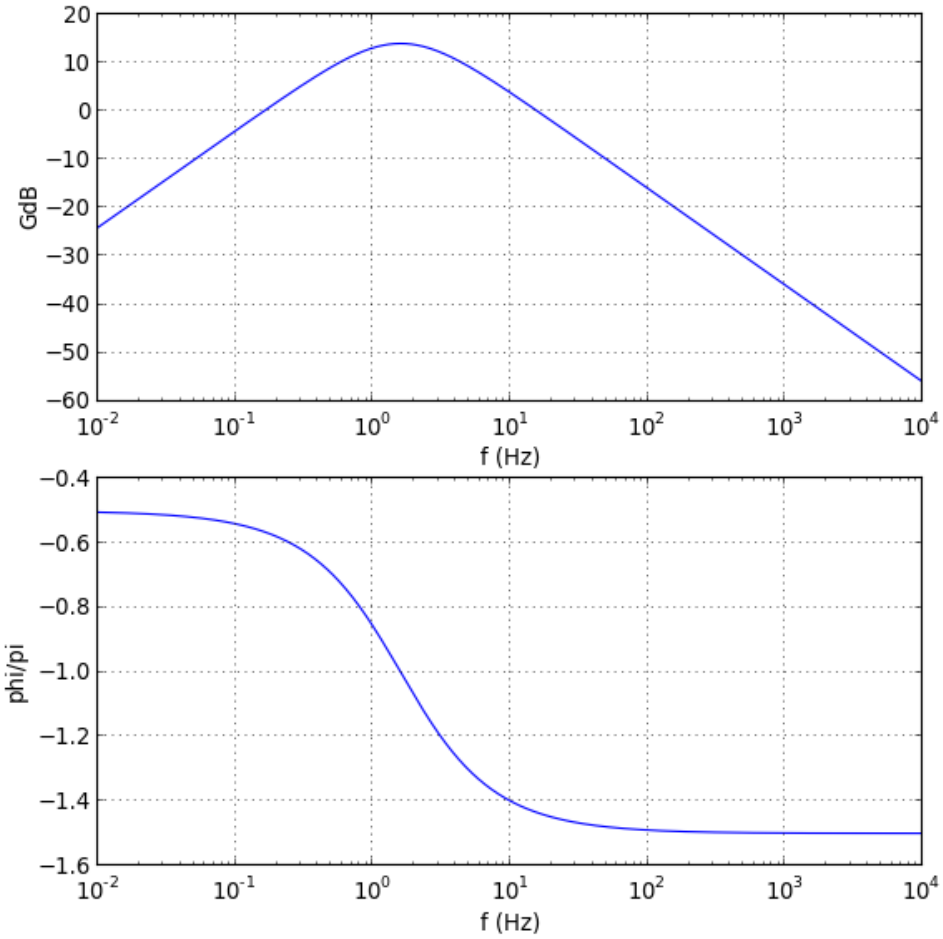
$$\omega \gg \frac{1}{R_2 C_2} \tag{4}$$

Le rôle du filtre passe-haut est d'enlever la composante de fréquence nulle (composante continue). Sa fréquence de coupure doit être inférieure à celle du filtre passe-bas :

$$R_2 C_2 < R_1 C_1 \tag{5}$$

Voici le diagramme de Bode de ce filtre :

```
import numpy
from matplotlib.pyplot import *

R1=100e3
C1=1e-6
R2=1e6
C2=100e-9

def H2(f):
    w=2*numpy.pi*f
    return (-R2/R1)/(1+1j*R2*C2*w)

def H1(f):
    w=2*numpy.pi*f
    return (1j*R1*C1*w)/(1+1j*R1*C1*w)

f = numpy.logspace(-2,4,1000)
H = H1(f)*H2(f)
GdB = 20*numpy.log10(numpy.absolute(H))
phi = numpy.unwrap(numpy.angle(H))

figure(figsize=(8,8))
subplot(211)
plot(f,GdB)
xscale('log')
xlabel("f (Hz)")
ylabel("GdB")
grid()

subplot(212)
plot(f,phi/numpy.pi)
xscale('log')
xlabel("f (Hz)")
ylabel("phi/pi")
grid()
```

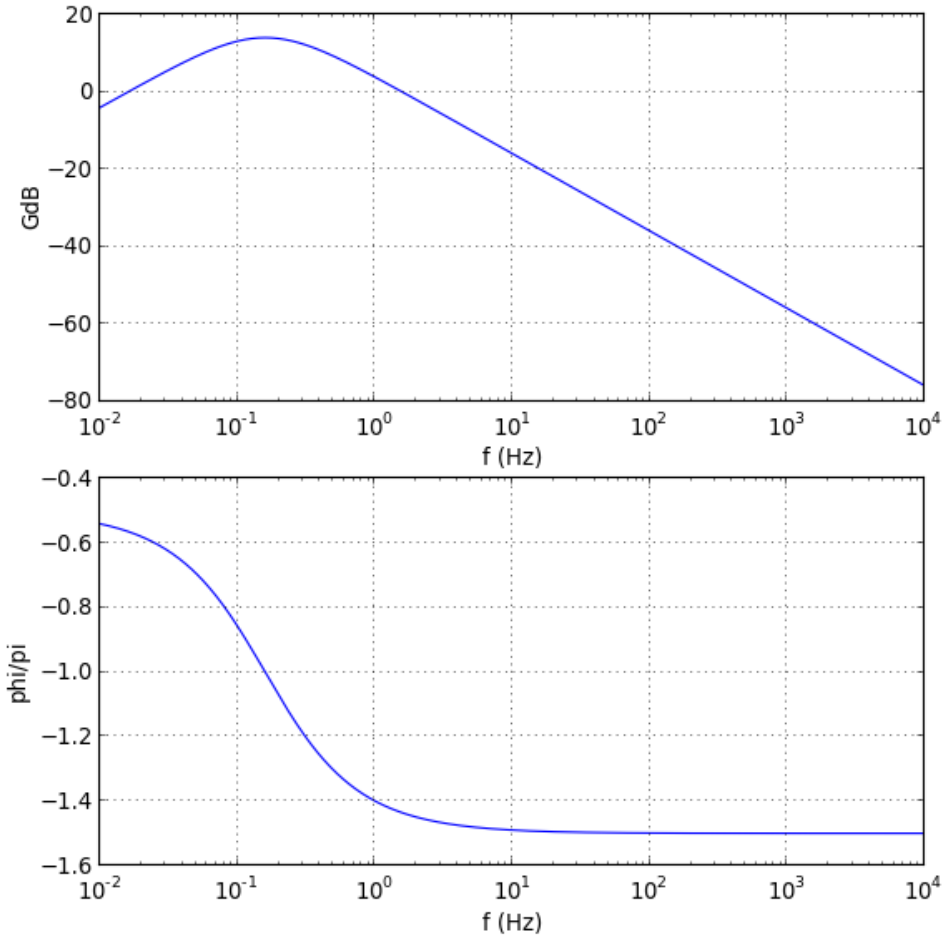


Le filtre est intégrateur lorsque le déphasage est égal à $-\pi/2$, environ à partir de 20 Hz. Le gain à cet fréquence est de 0 dB. Au délà, il décroît de 20 dB par décade.

On peut abaisser la fréquence minimale d'intégration en augmentant d'un facteur 10 les résistances (intégrateur 2) :

```
R1=1e6
C1=1e-6
R2=10e6
C2=100e-9

H = H1(f)*H2(f)
GdB = 20*numpy.log10(numpy.absolute(H))
phi = numpy.unwrap(numpy.angle(H))

figure(figsize=(8,8))
subplot(211)
plot(f,GdB)
xscale('log')
xlabel("f (Hz)")
ylabel("GdB")
```

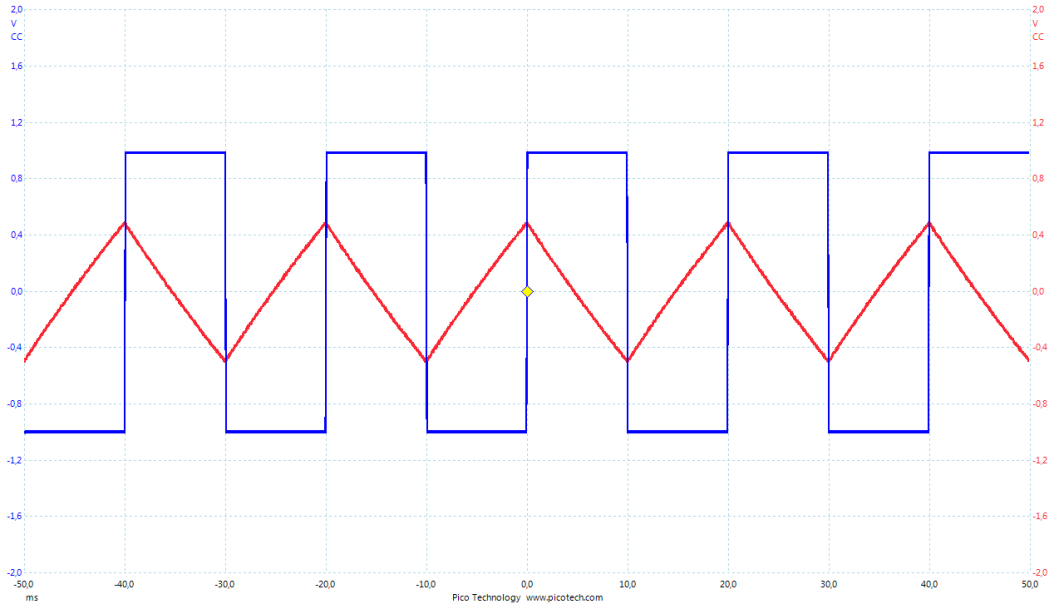
```
grid()

subplot(212)
plot(f,phi/numpy.pi)
xscale('log')
xlabel("f (Hz)")
ylabel("phi/pi")
grid()
```



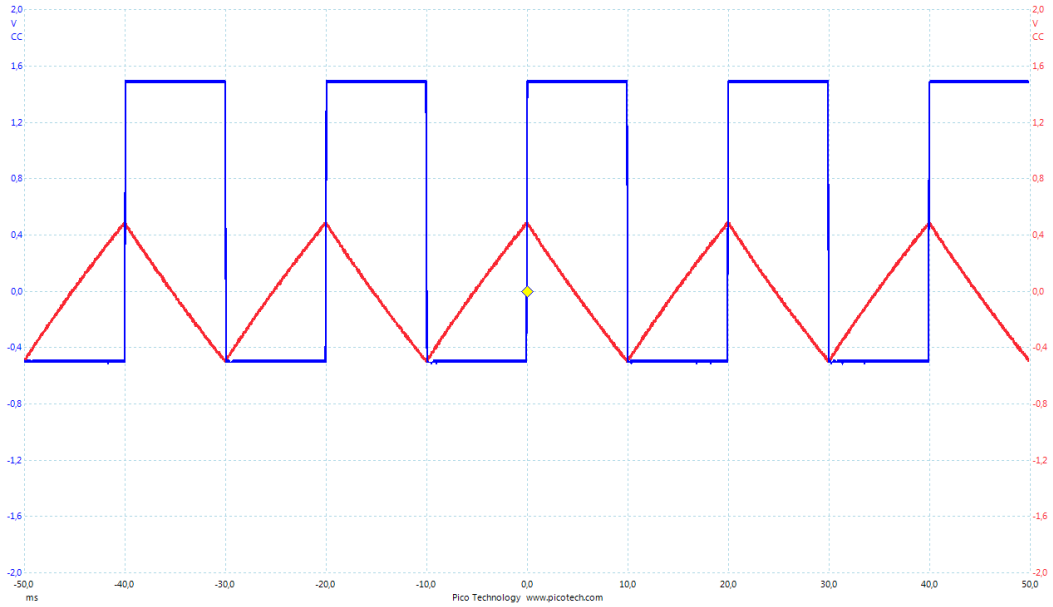
L'intégration est réalisée à partir de 2 Hz (environ).

2.b. Test expérimental de l'intégrateur 1

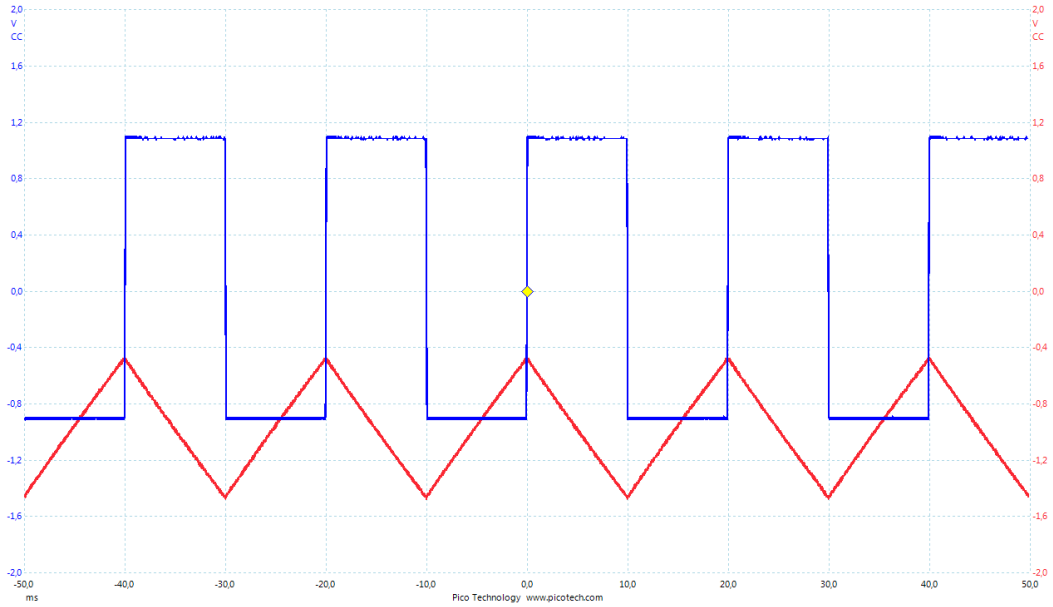
L'intégrateur est testé avec un signal en créneau délivré par un générateur de signaux. Voici l'oscillogramme obtenu pour une fréquence de 50 Hz avec l'intégrateur 1 :



On vérifie aussi que l'introduction d'un décalage de la tension en entrée ne conduit pas à une dérive en sortie :



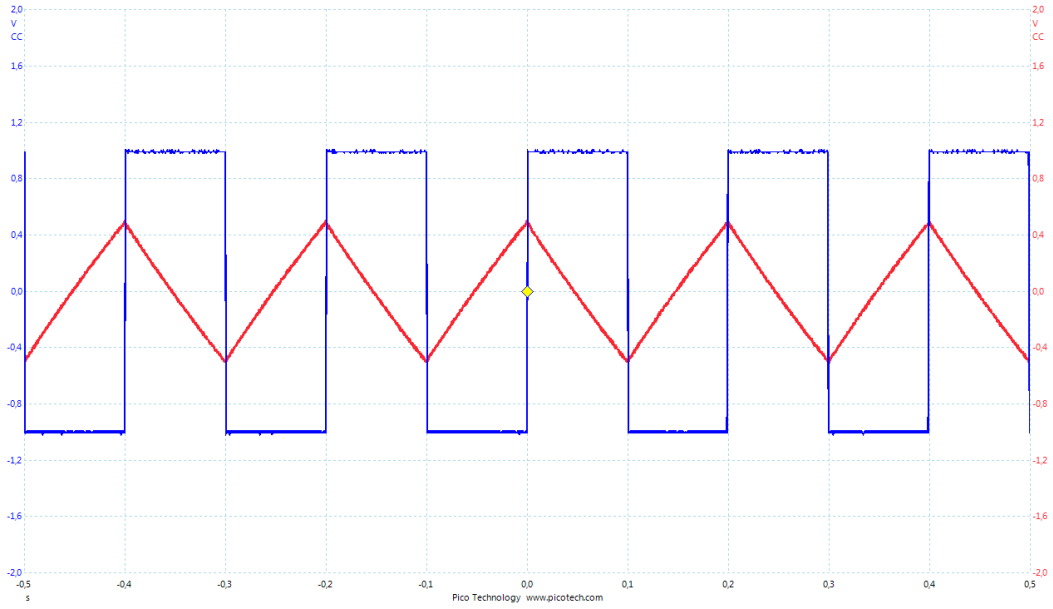
Voici ce qu'il arrive pour le même signal si on remplace le couplage AC en entrée par un couplage DC (en enlevant C_1) :



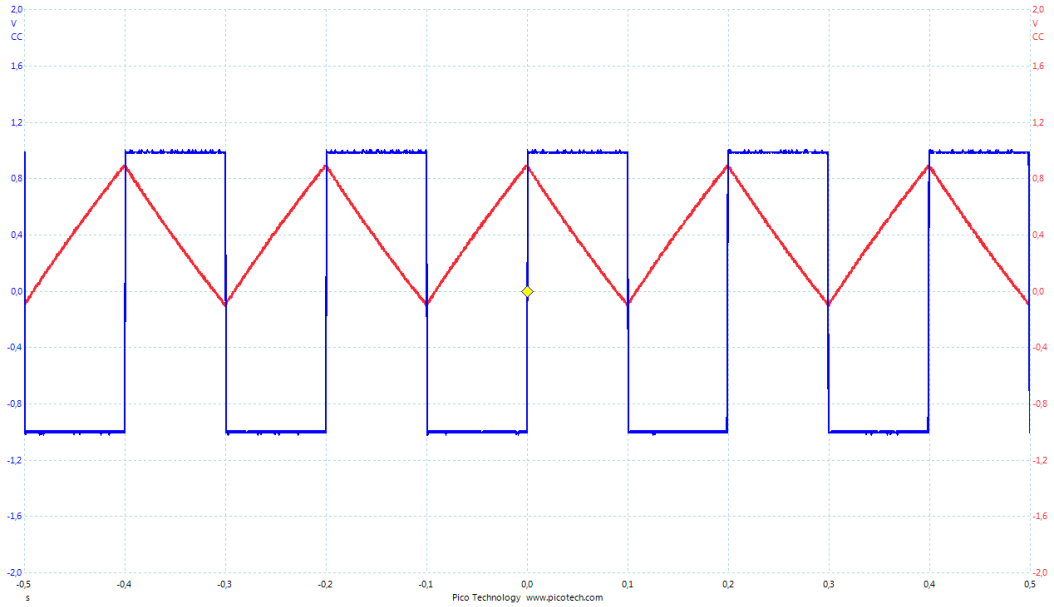
Le gain (négatif) de la composante continue introduit un décalage très important en sortie.

2.c. Test de l'intégrateur 2

Voici le résultat avec l'intégrateur 2 et un signal créneau de 5 Hz :



Voici ce qu'il arrive si on remplace l'ALI TL081 par un 741 (à transistors bipolaires) :



Il apparaît un décalage important en sortie, à cause du courant continue de l'entrée inverseuse qui passe dans la grande résistance de 10 $M\Omega$. Pour éviter cet inconvénient, il faut utiliser un ALI à entrées JFET comme le TL081, dont le courant d'entrée est beaucoup plus faible.

3. Intégrateur numérique

3.a. Numérisation d'un signal

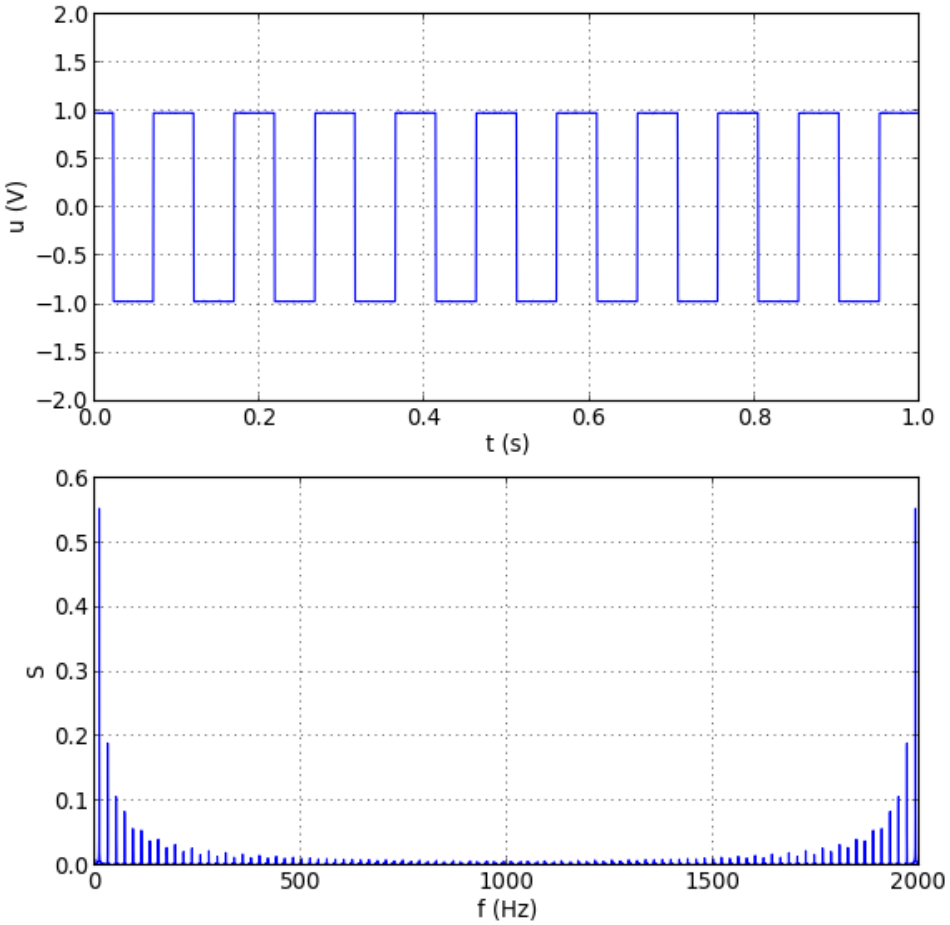
Pour tester l'intégrateur, on fait l'acquisition d'un signal créneau, avec une fréquence d'échantillonnage de 2 kHz.

acquisition.py

```
# -*- coding: utf-8 -*-
import pycan.main as pycan
import numpy
sys = pycan.Sysam("SP5")
fe = 2000.0 # fréquence d'échantillonnage
te = 1.0/fe
N=20000 # nombre d'échantillons
T=N*te
Umax = 2.0 # tension maximale
sys.config_entrees([0],[Umax])
sys.config_echantillon(te*1e6,N)
sys.acquerir()
temps = sys.temps()
entrees = sys.entrees()
t0 = temps[0]
u0 = entrees[0]
numpy.savetxt("creneau10.23Hz.txt",[t0,u0])
```

Voici par exemple le résultat de l'acquisition d'un signal créneau de $10,23\ Hz$ et son analyse spectrale :

```
import numpy.fft
[t0,u0] = numpy.loadtxt("creneau10.23Hz.txt")
te=t0[1]-t0[0]
N=t0.size
figure(figsize=(8,8))
subplot(211)
plot(t0,u0)
xlabel("t (s)")
ylabel("u (V)")
grid()
axis([0,1.0,-2,2])
subplot(212)
spectre = numpy.absolute(numpy.fft.fft(u0))/N
freq = numpy.arange(N)*1.0/(N*te)
plot(freq,spectre)
xlabel("f (Hz)")
ylabel("S")
grid()
```



3.b. Intégrateur élémentaire

L'intégrateur numérique élémentaire est défini par la relation de récurrence suivante :

$$y_n = y_{n-1} + \beta(x_n + x_{n-1}) \tag{6}$$

où x_n est l'entrée du filtre, y_n la sortie, et β un coefficient ajustable en fonction du gain souhaité.

Il s'agit d'un exemple de filtre récursif, défini de manière générale par la relation de récurrence :

$$y_n = \sum_{k=0}^{N-1} b_k x_{n-k} - \sum_{k=1}^{M-1} a_k y_{n-k} \tag{7}$$

Pour étudier ce type de filtre, on introduit la fonction de transfert en Z :

$$H(z) = \frac{b_0 + b_1 z^{-1} + b_2 z^{-2} + \cdots}{1 + a_1 z^{-1} + a_2 z^{-2} + \cdots} \tag{8}$$

La réponse fréquentielle est obtenue en posant :

$$z = e^{j2\pi \frac{f}{fe}} \tag{9}$$

où f_e est la fréquence d'échantillonnage.

Un filtre récursif peut être instable, car sa réponse impulsionnelle est en général infinie. L'étude de sa stabilité se fait en considérant les pôles de sa fonction de transfert, dont le module doit être inférieur à 1.

Pour le filtre intégrateur défini ci-dessus, la fonction de transfert en Z est :

$$H(z) = \beta \frac{1 + z^{-1}}{1 - z^{-1}} = \beta \frac{z + 1}{z - 1} \tag{10}$$

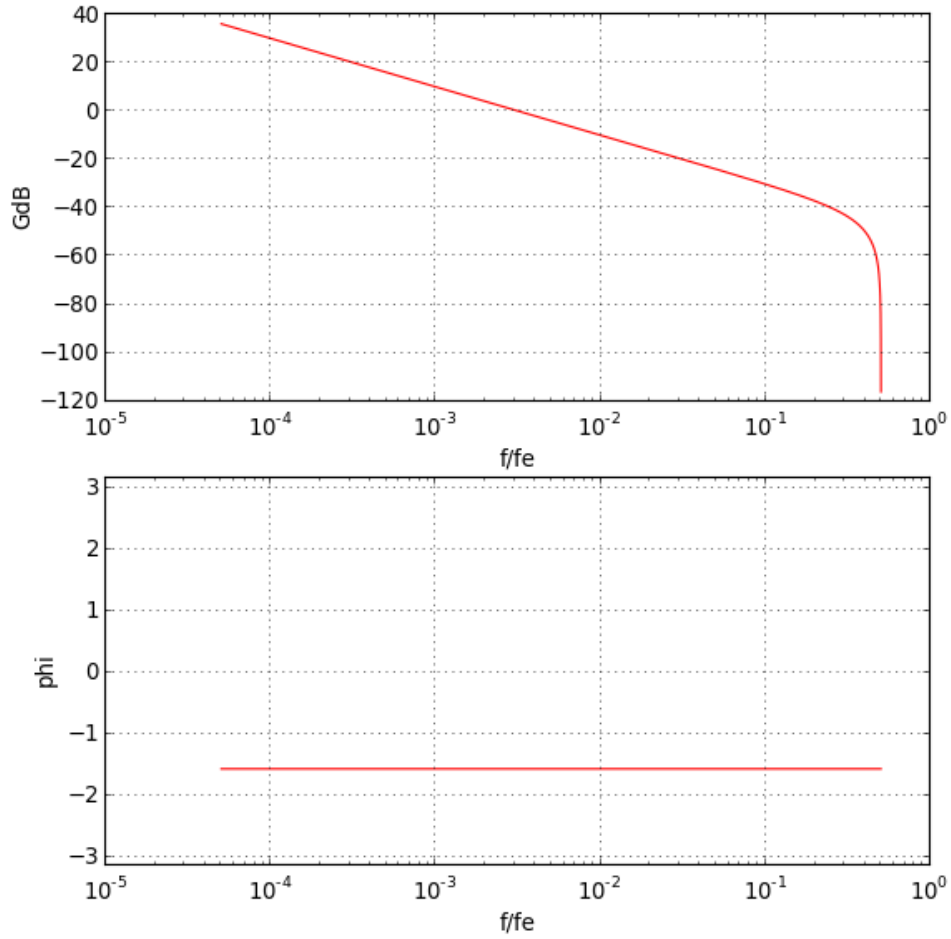
Le coefficient β n'a pas d'effet sur la forme de la réponse fréquentielle. Il sert à ajuster le gain du filtre.

Pour tracer la réponse fréquentielle (diagramme de Bode), on utilisera la fonction suivante, à laquelle on doit fournir les coefficients a_k et b_k sous forme de listes :

```
import scipy.signal

def trace_reponse_freq(b,a):
    [w,hf] = scipy.signal.freqz(b,a,worN=10000)
    subplot(211)
    plot(w/(2*numpy.pi),20*numpy.log10(numpy.abs(hf)),'r')
    xlabel('f/fe')
    ylabel('GdB')
    xscale('log')
    grid()
    subplot(212)
    plot(w/(2*numpy.pi),numpy.angle(hf),'r')
    xlabel('f/fe')
    ylabel('phi')
    xscale('log')
    axis([1e-5,1,-numpy.pi,numpy.pi])
    grid()
```

Voici la réponse fréquentielle du filtre intégrateur :

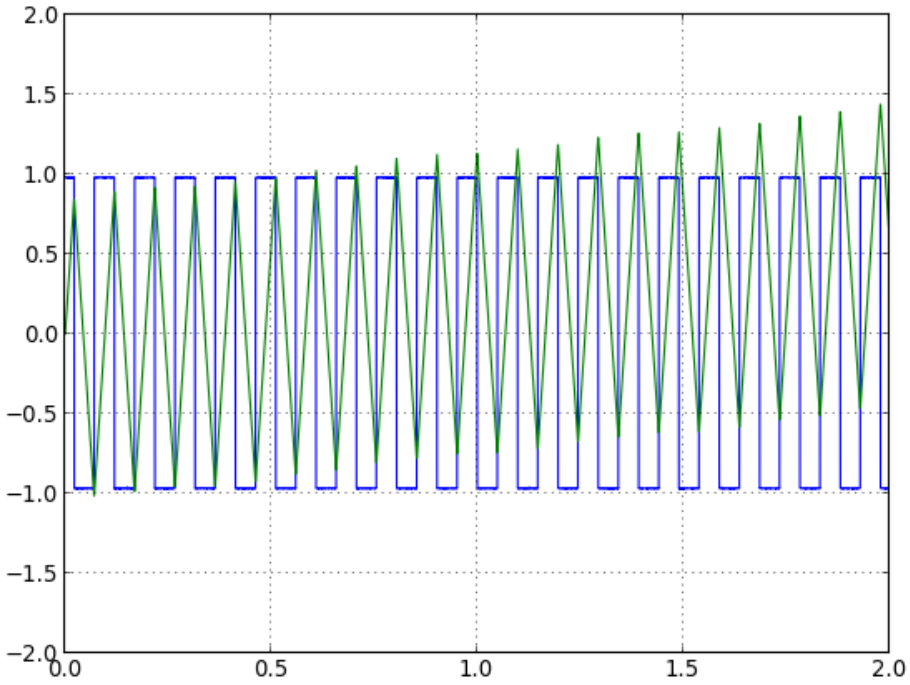
```
beta=0.01
b=[beta,beta]
a=[1.0,-1.0]
figure(figsize=(8,8))
trace_reponse_freq(b,a)
```



Ce filtre est parfaitement intégrateur, sauf à l'approche de la fréquence de Nyquist ($f_e/2$). Il a toutefois un inconvénient très important : son gain à fréquence nulle est infini, car il intègre aussi une composante de fréquence nulle. En conséquence, une composante continue dans le signal d'entrée (il y en a toujours une, aussi petite soit elle), conduit à une instabilité du filtre.

Pour appliquer ce filtre au signal numérisé ci-dessus, on peut appliquer directement la relation de récurrence, en posant $y_0 = 0$ pour le premier terme :

```
y = numpy.zeros(N)
x = u0.copy()
for n in range(2,N):
    y[n] = y[n-1]+beta*(x[n]+x[n-1])
figure()
plot(t0,x,label="x")
plot(t0,y,label="y")
grid()
axis([0,2,-2,2])
```



On observe la dérive due à la présence d'une petite composante continue dans le signal créneau.

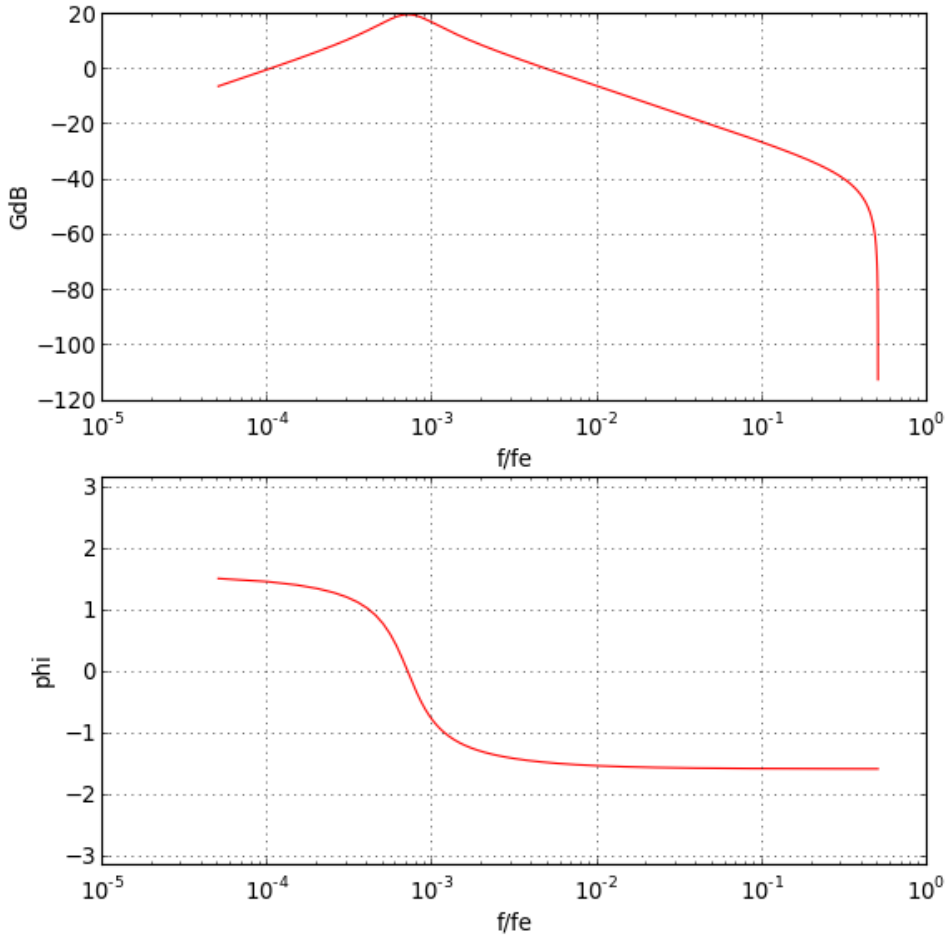
3.c. Filtrage passe-haut

Pour éliminer le phénomène de dérive, il faut combiner l'intégrateur élémentaire avec un filtre passe-haut, dont le rôle est d'éliminer la composante continue du signal d'entrée. Dans le filtre analogique étudié plus haut, cette fonction passe-haut est effectivement réalisée.

Le filtrage passe-haut ne peut être réalisé par un filtre à réponse impulsionnelle finie à phase linéaire, car ce type de filtre introduit un retard important. Une meilleure méthode consiste à construire un filtre semblable au filtre analogique considéré plus haut, en utilisant la méthode de la transformation bilinéaire, introduite dans Filtre à réponse impulsionnelle infinie.

Nous allons ici utiliser la fonction `scipy.signal.iirfilter`, qui effectue la transformation bilinéaire pour les filtres analogiques standard (Butterworth ou Chebyshev). Il faut définir un filtre passe-bande, qui effectuera l'intégration dans sa partie décroissante. Celle-ci doit donc être du premier ordre, ce qui nous conduit à définir un filtre de Butterworth du premier ordre. On définit pour cela une fréquence de coupure basse et une fréquence de coupure haute (relatives à la fréquence d'échantillonnage) :

```
fc1=0.0005
fc2=0.001
(b,a)=scipy.signal.iirfilter(1,[fc1*2,fc2*2],btype="bandpass",ftype="butter")
b=b*10
figure(figsize=(8,8))
trace_reponse_freq(b,a)
```



On a multiplié les coefficients b_k par 10 pour augmenter le gain. Le filtre obtenu supprime bien la composante continue (gain nul à fréquence nulle). La contrepartie est évidemment un domaine intégrateur plus limité, qui commence à environ $0,002 f_e$, soit 4 Hz pour la fréquence d'échantillonnage de 2 kHz utilisée ci-dessus.

Pour étudier la stabilité du filtre, on calcule les racines du dénominateur de sa fonction de transfert et leur norme :

```
from numpy.polynomial.polynomial import polyroots
poles = polyroots(a[::-1])


print(numpy.absolute(poles))
--> array([ 0.99843043,  0.99843043])
```

Les pôles ont un module strictement inférieur à 1, ce qui prouve la stabilité du filtre.

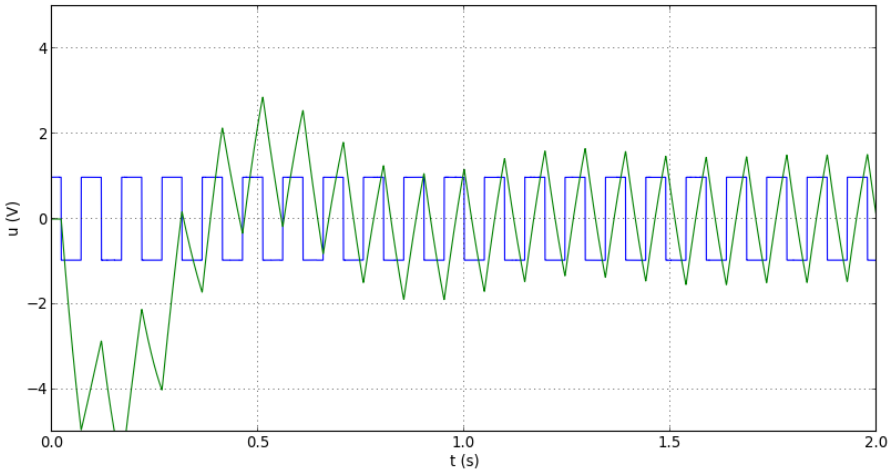
Voyons les coefficients du filtre :

```
print(a)
--> array([ 1.        , -1.99684362,  0.99686333])
```

```
print(b)
--> array([ 0.01568334,  0.        , -0.01568334])
```

Pour appliquer ce filtre au signal numérisé, on peut appliquer directement la relation de récurrence (7). Il faut pour cela attribuer une condition initiale à la sortie, c'est-à-dire les valeurs de y_0 et y_1. Voyons le résultat avec des valeurs initiales nulles :

```
y = numpy.zeros(N)
x = u0.copy()
for n in range(3,N):
    y[n] = b[0]*x[n]+b[1]*x[n-1]+b[2]*x[n-2]-a[1]*y[n-1]-a[2]*y[n-2]
figure(figsize=(12,6))
plot(t0,x,label="x")
plot(t0,y,label="y")
xlabel("t (s)")
ylabel("u (V)")
grid()
axis([0,2,-5,5])
```



Après un régime transitoire d'une dizaine de périodes, le filtre réalise bien une intégration insensible à la composante continue. Le temps caractéristique du régime transitoire est l'inverse de la fréquence du maximum du gain, ici environ $1,6\ Hz$. C'est le prix à payer pour ne pas avoir de dérive en régime permanent. Le filtre intégrateur élémentaire ne présente pas ce régime transitoire.

L'application directe de la relation (7) est appelée *réalisation directe de type 1*. Il existe une autre manière de procéder, consistant à définir un filtre intermédiaire qui réalise la partie récursive seulement, c'est-à-dire le dénominateur de la fonction de transfert :

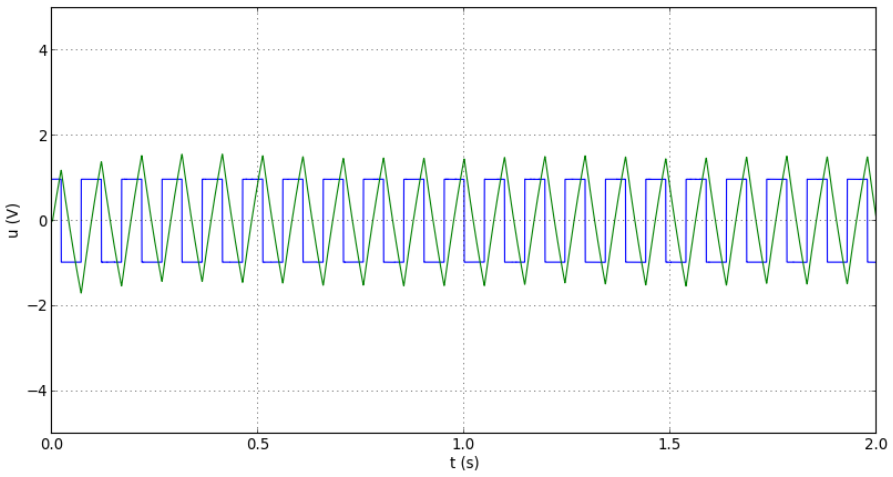
$$w_n = x_n - a_1 w_{n-1} - a_2 w_{n-2} \tag{11}$$

La sortie est alors calculée en appliquant la partie non récursive (le numérateur) au signal intermédiaire :

$$y_n = b_0 w_n + b_1 w_{n-1} + b_2 w_{n-2} \tag{12}$$

Ces deux relations définissent une *réalisation directe de type 2*. Un des avantages de cette méthode est de nécessiter seulement un tampon de mémoire pour le stockage de w_k. Voyons l'application au signal numérisé, avec $w_0 = w_1 = 0$:

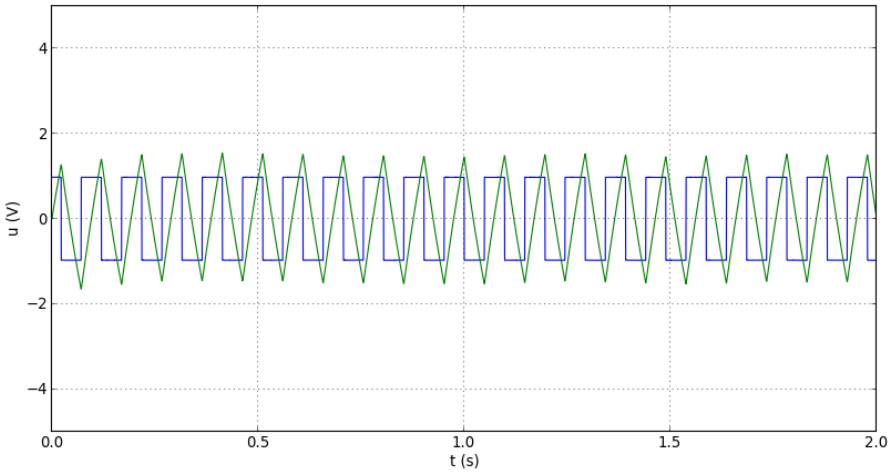
```
y = numpy.zeros(N)
x = u0.copy()
w = numpy.zeros(N)
for k in range(3,N):
    w[k] = x[k]-a[1]*w[k-1]-a[2]*w[k-2]
    y[k] = b[0]*w[k]+b[1]*w[k-1]+b[2]*w[k-2]
figure(figsize=(12,6))
plot(t0,x,label="x")
plot(t0,y,label="y")
xlabel("t (s)")
ylabel("u (V)")
grid()
axis([0,2,-5,5])
```



Le régime transitoire est beaucoup plus discret avec cette méthode.

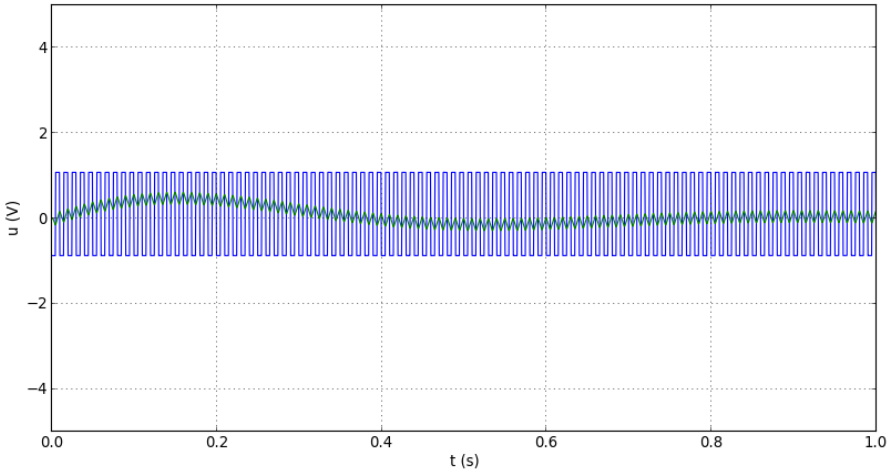
On peut aussi utiliser la fonction `scipy.signal.lfilter` :

```
zi = scipy.signal.lfiltic(b,a,y=[0],x=[0])
[y,zf] = scipy.signal.lfilter(b,a,u0,axis=-1,zi=zi)
figure(figsize=(12,6))
plot(t0,x,label="x")
plot(t0,y,label="y")
xlabel("t (s)")
ylabel("u (V)")
grid()
axis([0,2,-5,5])
```



Voici l'application du filtre à un créneau de 100 Hz :

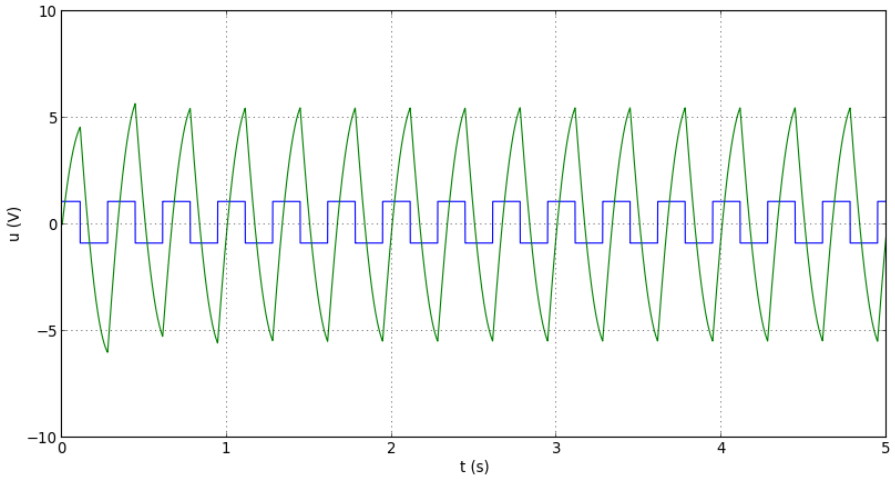
```
[t0,u0] = numpy.loadtxt("creneau-100Hz.txt")
te=t0[1]-t0[0]
N=t0.size
y = numpy.zeros(N)
x = u0.copy()
w = numpy.zeros(N)
for k in range(3,N):
    w[k] = x[k]-a[1]*w[k-1]-a[2]*w[k-2]
    y[k] = b[0]*w[k]+b[1]*w[k-1]+b[2]*w[k-2]
figure(figsize=(12,6))
plot(t0,x,label="x")
plot(t0,y,label="y")
xlabel("t (s)")
ylabel("u (V)")
grid()
axis([0,1.0,-5,5])
```



Comme prévu, l'amplitude en sortie est faible. On peut toujours multiplier les coefficients b_k par une constante plus grande pour augmenter le gain. Le régime transitoire a toujours la même durée absolue.

Enfin voici le résultat pour un créneau de 3 Hz :

```
[t0,u0] = numpy.loadtxt("creneau-3Hz.txt")
te=t0[1]-t0[0]
N=t0.size
y = numpy.zeros(N)
x = u0.copy()
w = numpy.zeros(N)
for k in range(3,N):
    w[k] = x[k]-a[1]*w[k-1]-a[2]*w[k-2]
    y[k] = b[0]*w[k]+b[1]*w[k-1]+b[2]*w[k-2]
figure(figsize=(12,6))
plot(t0,x,label="x")
plot(t0,y,label="y")
xlabel("t (s)")
ylabel("u (V)")
grid()
axis([0,5,-10,10])
```



Le rapport $f/f_e = 1,5\ 10^{-3}$ donne une intégration très approximative. Pour avoir toujours une bonne intégration à cette fréquence, il faut abaisser les fréquences de coupure du filtre passe-bande, ce qui n'est pas sans inconvénient sur le régime transitoire qui s'allongerait. Comme toujours en électronique, il faut trouver le compromis adapté aux signaux à traiter. De ce point de vue, l'avantage d'un filtre numérique est la possibilité de modifier facilement les paramètres pour s'adapter aux différentes situations. En revanche, le filtre numérique est limité par la fréquence d'échantillonnage. Dans le cas présent, le comportement intégrateur se fait jusqu'à environ 300 Hz pour une fréquence d'échantillonnage de 1 kHz.

3.d. Filtrage en mode permanent

Il est intéressant de tester le comportement du filtre intégrateur en faisant varier continûment la fréquence, l'amplitude, ou le décalage du signal. Le script suivant, qui fonctionne avec la version 4.0.2 de pyCAN, trace le signal de départ et le signal filtré dans une fenêtre actualisée en permanence (à la manière d'un oscilloscope). Le filtrage est effectué en interne par pyCAN. Les coefficients du filtre sont fournis par la commande `sys.config_filtre(a,b)`.

acquisitionSansFinFiltrage.py

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

import pycan.main as pycan
import math
from matplotlib.pyplot import *
import matplotlib.animation as animation
import numpy
import scipy.signal
import time


sys=pycan.Sysam("SP5")
Umax = 10.0
sys.config_entrees([0],[Umax])
fe=2000.0 # fréquence de la numérisation
te=1.0/fe
N = 2000 # nombre d'échantillons dans la liste circulaire (fenêtre d'analyse)
duree = N*te
print(u"Durée de la fenêtre = %f s"%(duree))
sys.config_echantillon_permanent(te*1e6,N)
longueur_bloc = N


#filtre intégrateur
fc1=0.0005
fc2=0.001
(b,a)=scipy.signal.iirfilter(1,[fc1*2,fc2*2],btype="bandpass",ftype="butter")
b=b*10
sys.config_filtre(a,b)

sys.lancer_permanent(repetition=1) # lancement d'une acquisition sans fin

fig0,ax0 = subplots()
t = numpy.arange(longueur_bloc)*te
u = numpy.zeros(longueur_bloc)
line0, = ax0.plot(t,u)
line1, = ax0.plot(t,u)
ax0.grid()
ax0.axis([0,duree,-Umax,Umax])
```

```
ax0.set_xlabel("t (s)")



def animate0(i): # tracé du signal
    global sys,line0,data,u
    data = sys.paquet_filtrees(-1)
    u = data[1]
    if u.size==longueur_bloc:
        line0.set_ydata(u)
    data = sys.paquet(-1)
    u = data[1]
    if u.size==longueur_bloc:
        line1.set_ydata(u)



ani0 = animation.FuncAnimation(fig0,animate0,frames=100,repeat=True,interval=duree*10
show()
sys.stopper_acquisition()
time.sleep(1)
sys.fermer()
```