TP_Adaptatif

J.-F. Bercher

December 11, 2013

1 Adaptatif

Le but de ce TP est d'illustrer et consolider quelques concepts sur les problèmes de filtrage adaptatif. Dans un premier temps, on étudie le problème de manière analytique, puis dans un second temps, on expérimente et étudie le problème par simulation. En particulier, et il faut bien le faire une fois, on programmera et on mettra en oeuvre un algorithme LMS ; on étudiera les propriétés de convergence, le rôle du pas d'adaptation, etc. On considèrera un problème d'identification, puis différents problèmes de soustraction de bruit.

1.1 Algorithme LMS

Programmez un algorithme LMS. La syntaxe d'appel sera la suivante :

```
def lms(d,u,w,mu):
"""
Calcule une itération de l'algorithme du gradient stochastique (LMS)\n
$w(n+1)=w(n)+\mu u(n)\left(d(n)-w(n)^Tu(n)\right)$$ \n
Entrées :
    d : séquence désirée à l'instant n \n
    u : vecteur de longueur p des échantillons d'entrée \n
    w : filtre de wiener à mettre à jour \n
    mu : pas d'adaptation
Sorties :
    w : filtre mis à jour
    erreur : y-yest
    dest : prédiction = $u(n)^T w$
"""
```

In [123]:
```
def lms(d,u,w,mu):
    """
    Calcule une itération de l'algorithme du gradient stochastique (LMS)\n
    :math:`w(n+1)=w(n)+\\mu u(n)\\left(d(n)-w(n)^T u(n)\\right)`


    Entrées :
    ========
        d : séquence désirée à l'instant n \n
        u : vecteur de longueur p des échantillons d'entrée \n
        w : filtre de wiener à mettre à jour \n
        mu : pas d'adaptation

    Sorties :
    ========
        w : filtre mis à jour
        erreur : y-yest
        dest : prédiction = :math:`u(n)^T w`
    """
    u=squeeze(u)
```

```
    w=squeeze(w)
    dest=u.dot(w)
    erreur=d-dest
    w=w+mu*u*erreur
    return (w,erreur,dest)
```

Identification d'un filtre

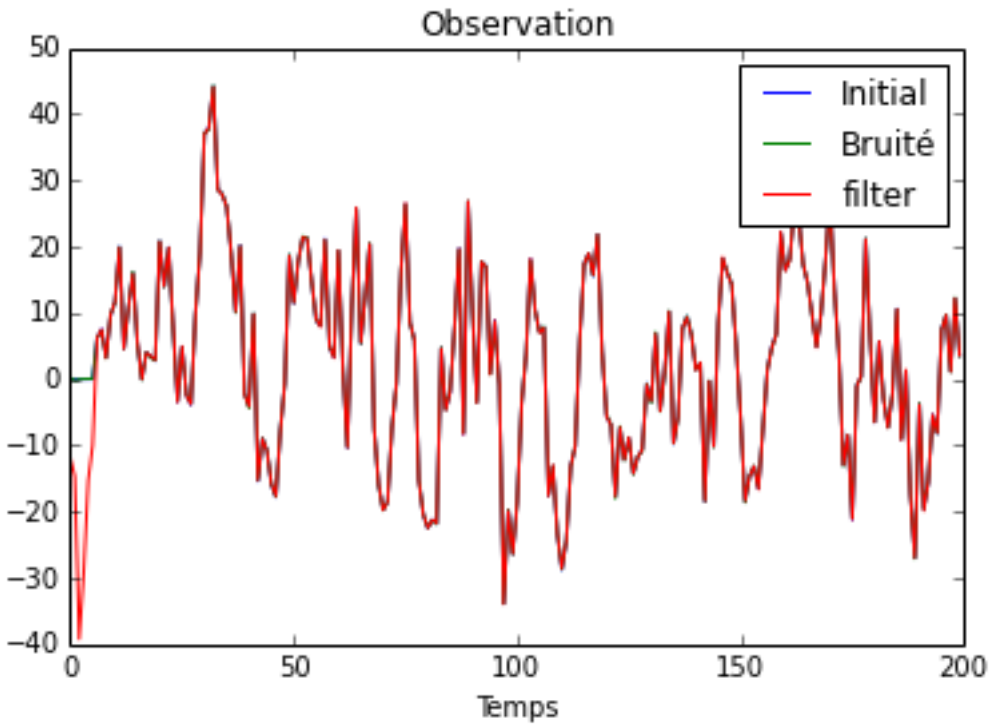
Vous testerez cet algorithme sur **un problème d'identification**. Vous utiliserez comme signal test la sortie d'un filtre excité par un bruit blanc, selon, par exemple # test N=200 x=randn(N) htest=10*array([1, 0.7, 0.7, 0.7, 0.3, 0]) L=size(htest) z=zeros(N) for t in range(L,200) : z[t]=htest.dot(x[t :t-L :-1]) z+= 0.01*randn(N) z2=scipy.signal.lfilter(htest,[1],x)+0.01*randn(N)

In [124]:
```
from scipy.signal import lfilter
# test
N=200
x=randn(N)
htest=10*array([1, 0.7, 0.7, 0.7, 0.3, 0 ])
L=size(htest)
yo=zeros(N)
for t in range(L,200):
    yo[t]=htest.dot(x[t:t-L:-1])
y=yo+ 0.1*randn(N)
y2=lfilter(htest,[1],x)+0.1*randn(N)
```

In [125]:
```
figure()
t=arange(size(y))
plot(t,yo, label='Initial')
plot(t,y, label='Bruité')
plot(t,y2, label='filter')
title('Observation')
xlabel('Temps')
legend()
```

```
<matplotlib.legend.Legend at 0x7f605fe48290>
```

Out [125]:



Pour mettre en oeuvre la procédure d'identification, il suffit de voir que l'on recherche un filtre, qui excité par $x(n)$ présente une sortie $z(n)$ la plus proche possible de $y_0(n)$. On prend donc ainsi `u=x`, et `d=y`(la séquence désirée est $y_0(n)$, que l'on doit remplacer par $y(n) - y_0$ n'est pas connu).

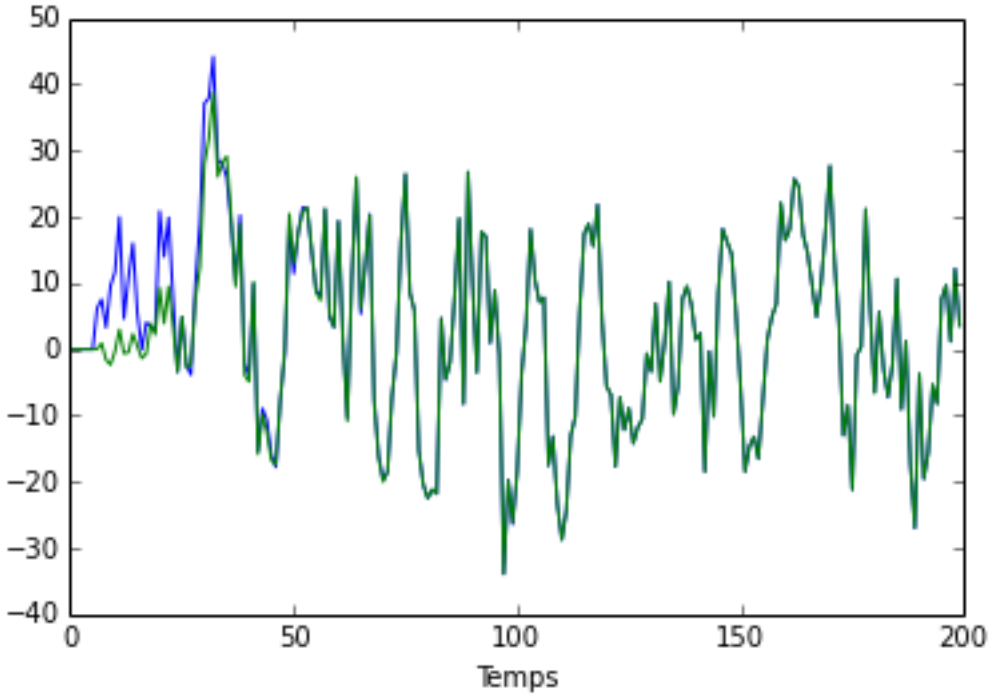
Mise en oeuvre :

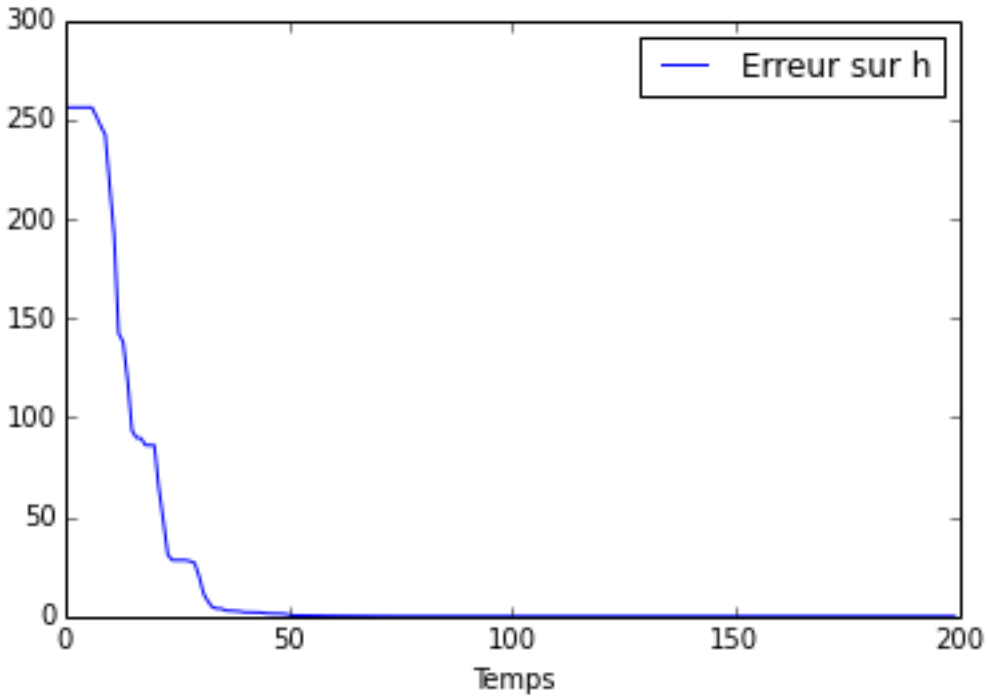
— Débutez par quelques commandes directes (initialisation et une boucle `for` sur le temps) pour effectuer cette identification

— Si nécessaire, la fonction `squeeze()` permet de supprimer les entrées monodimensionnelles du tableau n-D (*e.g.* transforme un tableau (3,1,1) en vecteur de dimension 3)

In [126]:
```
mu=0.1
erreur=zeros(200)
w=zeros((L,201))
yest=zeros(200)
for t in range(L,200):
    (w[:,t+1],erreur[t],yest[t])=lms(y[t],x[t:t-L:-1],w[:,t],mu)


figure(1)
tt=range(200)
plot(tt,yo,label='Signal initial non bruité')
plot(tt,yest, label="Estimée")
xlabel('Temps')

figure(2)
errh=[sum((htest-w[:,t])**2) for t in range(200)]
plot(tt,errh,label='Erreur sur h')
legend()
xlabel('Temps')
```

Out [126]:
```
<matplotlib.text.Text at 0x7f6060913cd0>
```





— Mettez ensuite en oeuvre une procédure d'identification adaptative, en écrivant une fonction `ident` utilisant une boucle sur l'algo `lms` appelé avec les paramètres pertinents

In [127]:
```
def ident(observation,input_data,mu,p=20,h_initial=zeros(20),normalized=False):
    """ Identification d'une réponse impulsionnelle à partir de l'observation
    'observation' de sa sortie et de son entrée 'input_data' \n
    'mu' est le pas d'adaptation \n
    Par défaut l'ordre est pris à p=20 \n
    et la condition initiale est prise, par défaut, à h_initial=zeros(20)
    """
    N=size(input_data)
    input_data=squeeze(input_data) #reshape(input_data,(N))
    observation=squeeze(observation)
    erreur=zeros(N)
    w=zeros((p,N+1))
    yest=zeros(N)

    w[:,p]=h_initial
    for t in range(p,N):
        if normalized:
            mun=mu/(dot(input_data[t:t-p:-1],input_data[t:t-p:-1])+1e-10)
        else:
            mun=mu
        (w[:,t+1],erreur[t],yest[t])=lms(observation[t],input_data[t:t-p:-1],w[:,t],

    return (w,erreur,yest)
```

Pour évaluer le comportement de l'algorithme, vous pourrez tracer l'erreur d'estimation, les coefficients du filtre identifié en fonction des itérations, ainsi que l'erreur quadratique entre le filtre identifié et le filtre exact. Ceci pourra être effectué pour différents ordres p (l'ordre du filtre n'est pas connu...) et pour différentes valeurs du pas d'adaptation μ.

— L'erreur quadratique pourra être évaluée simplement par une *liste en intention* selon

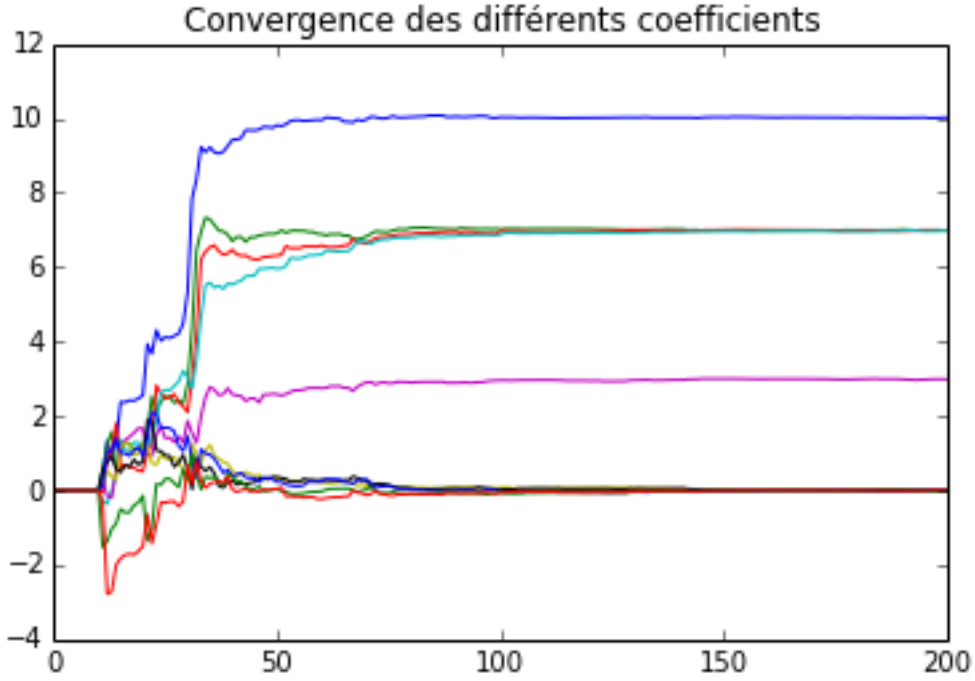
`Errh=[sum(he-w[:,n])**2 for n in range(N+1)]`

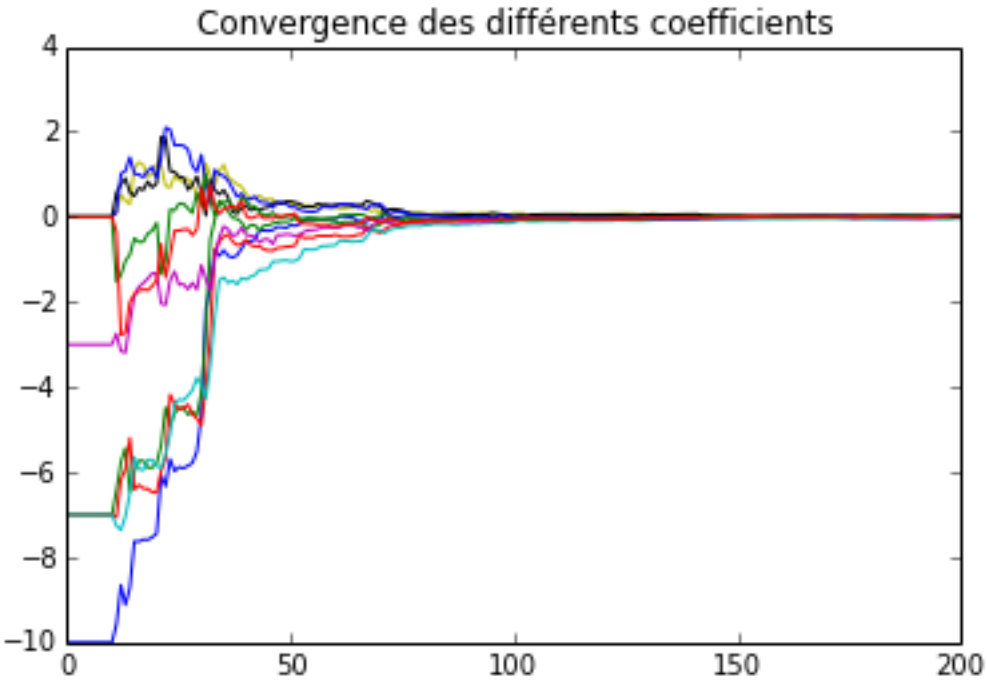
In [128]:
```
p=10
(w,erreur,yest)=ident(y,x,mu=0.05,p=p,h_initial=zeros(p))
figure(2)
plot(w.T)
title("Convergence des différents coefficients")
```

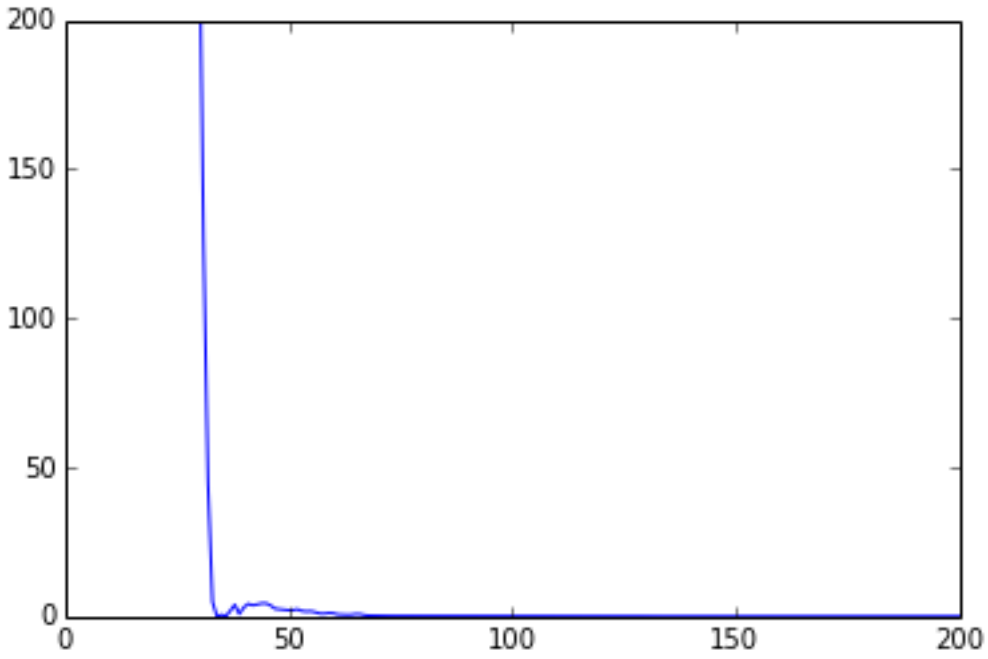
```
he=(np.concatenate((htest,zeros(4)),axis=0))
he_rep=outer(ones(N+1),he).T
figure(3)
plot((w-he_rep).T)
title("Convergence des différents coefficients")
# ou alors
figure(4)
Errh=[sum(he-w[:,n])**2 for n in range(N+1)]
plot(Errh)
ylim([0, 200])
```

```
(0, 200)
```

Out [128]:



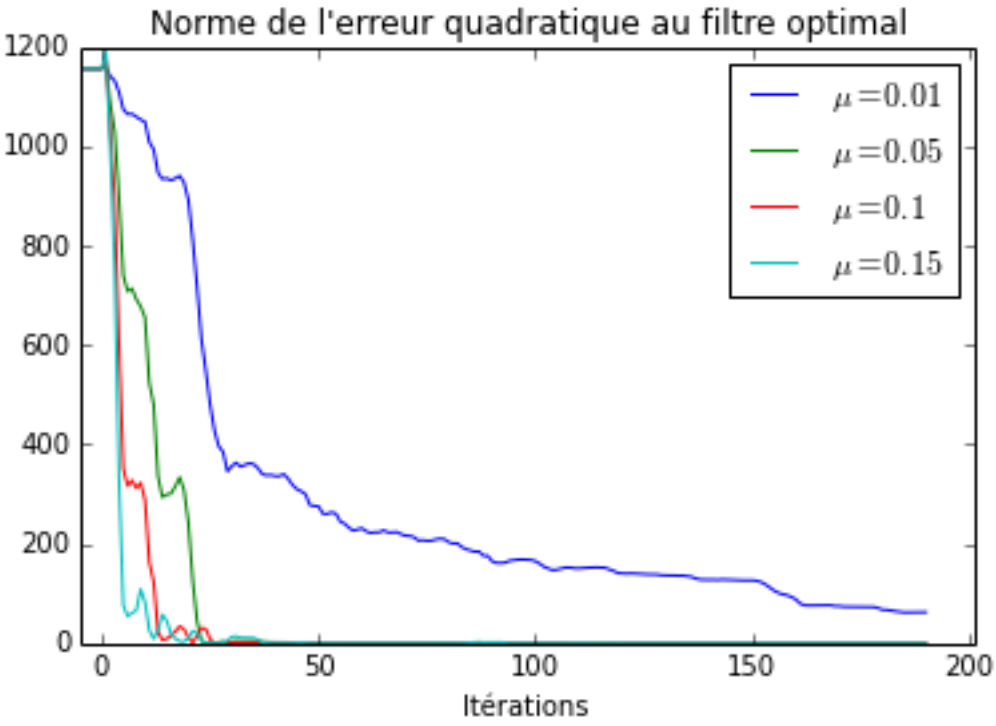




Etude en fonction de μ

In [129]:
```
 # Etude en fonction de mu
figure(4)
iter=arange(N+1)-p
for mu in [0.01, 0.05, 0.1, 0.15]:
    (w,erreur,yest)=ident(y,x,mu,p=p,h_initial=zeros(p))
    Errh=[sum(he-w[:,n])**2 for n in range(N+1)]
    plot(iter,Errh, label="$\mu={}$".format(mu))
    xlim([-5, N+1])

legend()
title("Norme de l'erreur quadratique au filtre optimal")
xlabel("Itérations")
```

Out [129]:
```
<matplotlib.text.Text at 0x7f605fdbd850>
```



LMS normalisé

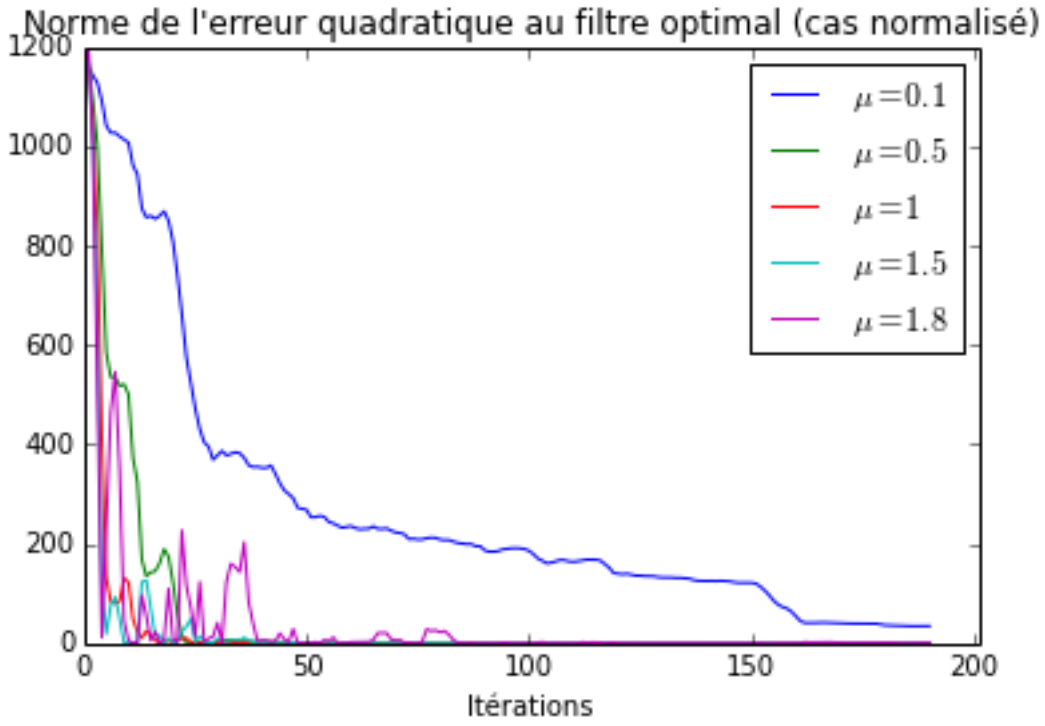
Le pas est ici calculé automatiquement comme

$$\mu = \frac{1}{u(n)^T u(n) + \epsilon}$$

```
mun=mu/(dot(input_data[t:t-p:-1],input_data[t:t-p:-1])+1e-10)
```

In [130]:
```
# LMS normalisé
figure()
for mu in [0.1, 0.5, 1, 1.5, 1.8 ]:
    (w,erreur,yest)=ident(y,x,mu=mu,p=p,h_initial=zeros(p),normalized=True)
    Errh=[sum(he-w[:,n])**2 for n in range(N+1)]
    plot(iter,Errh, label="$\mu={}$".format(mu))
xlim([0, N+1])
legend()
title("Norme de l'erreur quadratique au filtre optimal (cas normalisé)")
xlabel("Itérations")
```

Out [130]:
```
<matplotlib.text.Text at 0x7f605fcd80d0>
```

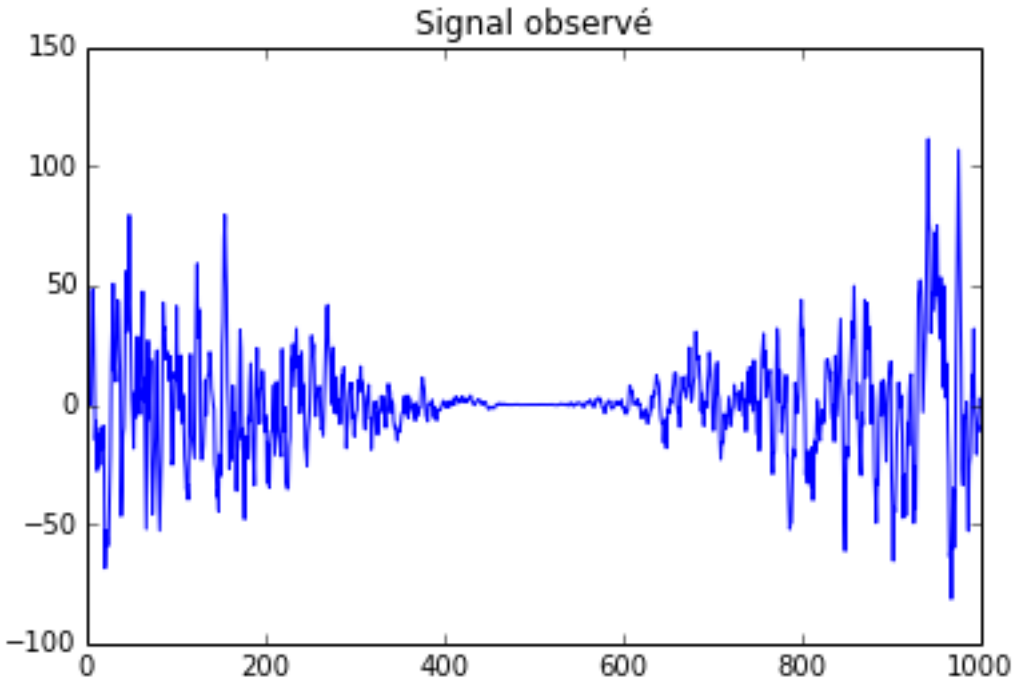


Examinez la vitesse de convergence en fonction du pas d'adaptation μ, ainsi que les capacités d'adaptation, en introduisant une non-stationnarité lente dans le signal, par exemple selon ### Non stationnarité lente

```
N=1000
u=randn(N)
y=zeros(N)
htest=10*array([1, 0.7, 0.7, 0.7, 0.3, 0 ])
L=size(htest)
for t in range(L,N):
    y[t]=dot((1+cos(2*pi*t/N))*htest,u[t:t-L:-1])
y+=0.01*randn(N)
```

In [131]:
```
### Non stationnarité lente

N=1000
u=randn(N)
y=zeros(N)
htest=10*array([1, 0.7, 0.7, 0.7, 0.3, 0 ])
L=size(htest)
for t in range(L,N):
    y[t]=dot((1+cos(2*pi*t/N))*htest,u[t:t-L:-1])
y+=0.01*randn(N)
figure()
plot(y)
title("Signal observé")
```

Out [131]:
```
<matplotlib.text.Text at 0x7f605fcd6b90>
```

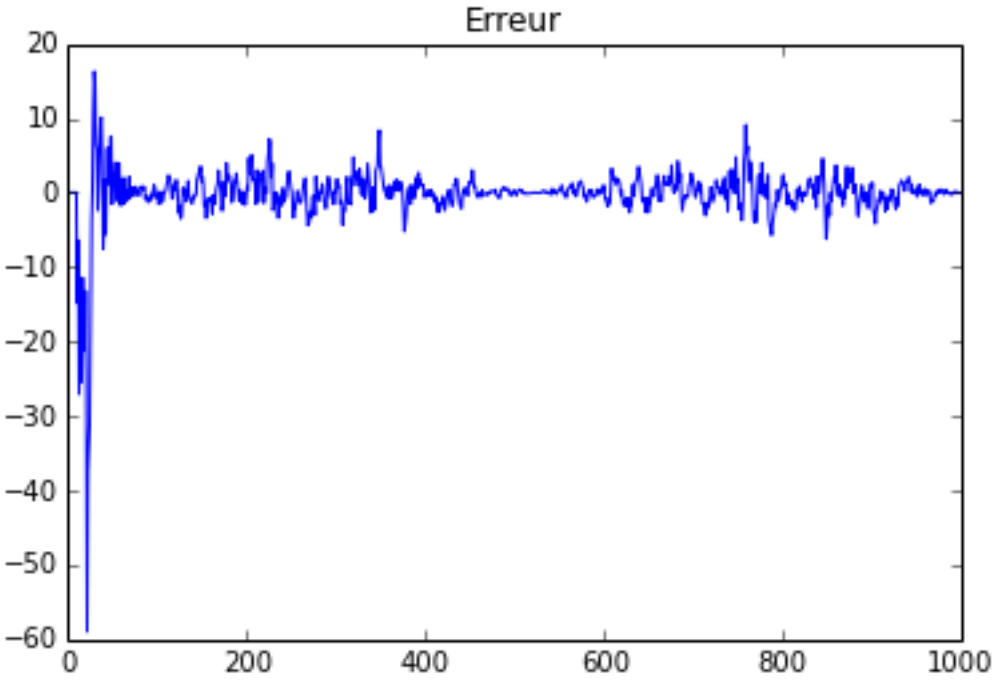


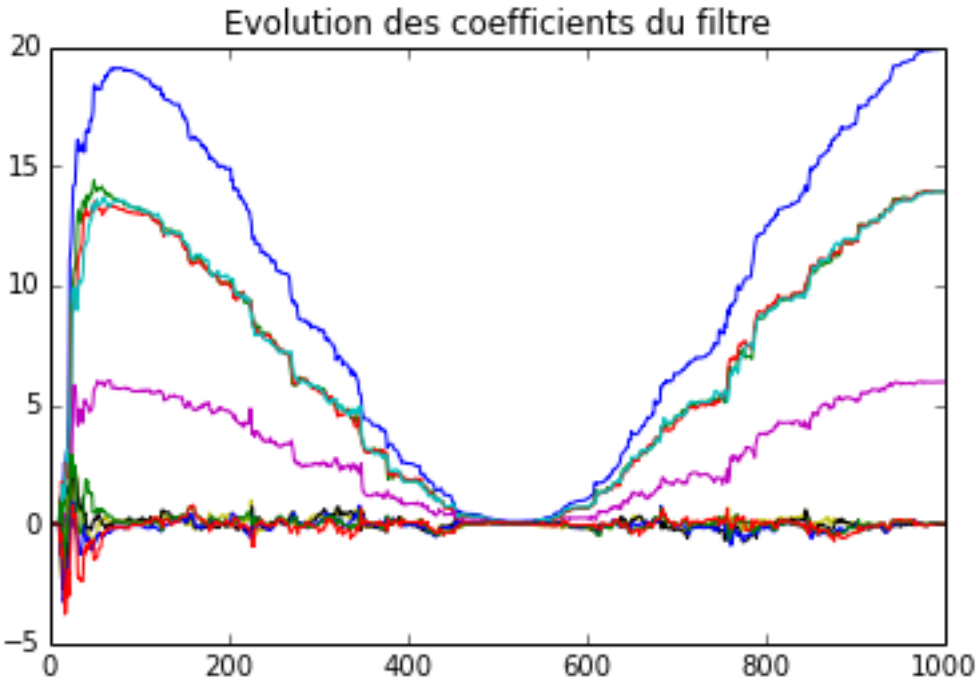
On utilise l'algo LMS sur ce signal non stationnaire.

In [132]:
```
p=10
(w,erreur,yest)=ident(y,u,mu=0.05,p=p,h_initial=zeros(p))
figure(1)
clf()
plot(erreur)
title('Erreur')
figure(2)
clf()
plot(w.T)
title("Evolution des coefficients du filtre")
```

```
<matplotlib.text.Text at 0x7f605fb97e10>
```

Out [132]:





Algorithme des MCR

Fort aimablement, on vous fournit une implantation de l'algorithme des mondres carrés récursifs. Reprendre alors les expérimentations précédentes. Comparez et concluez.

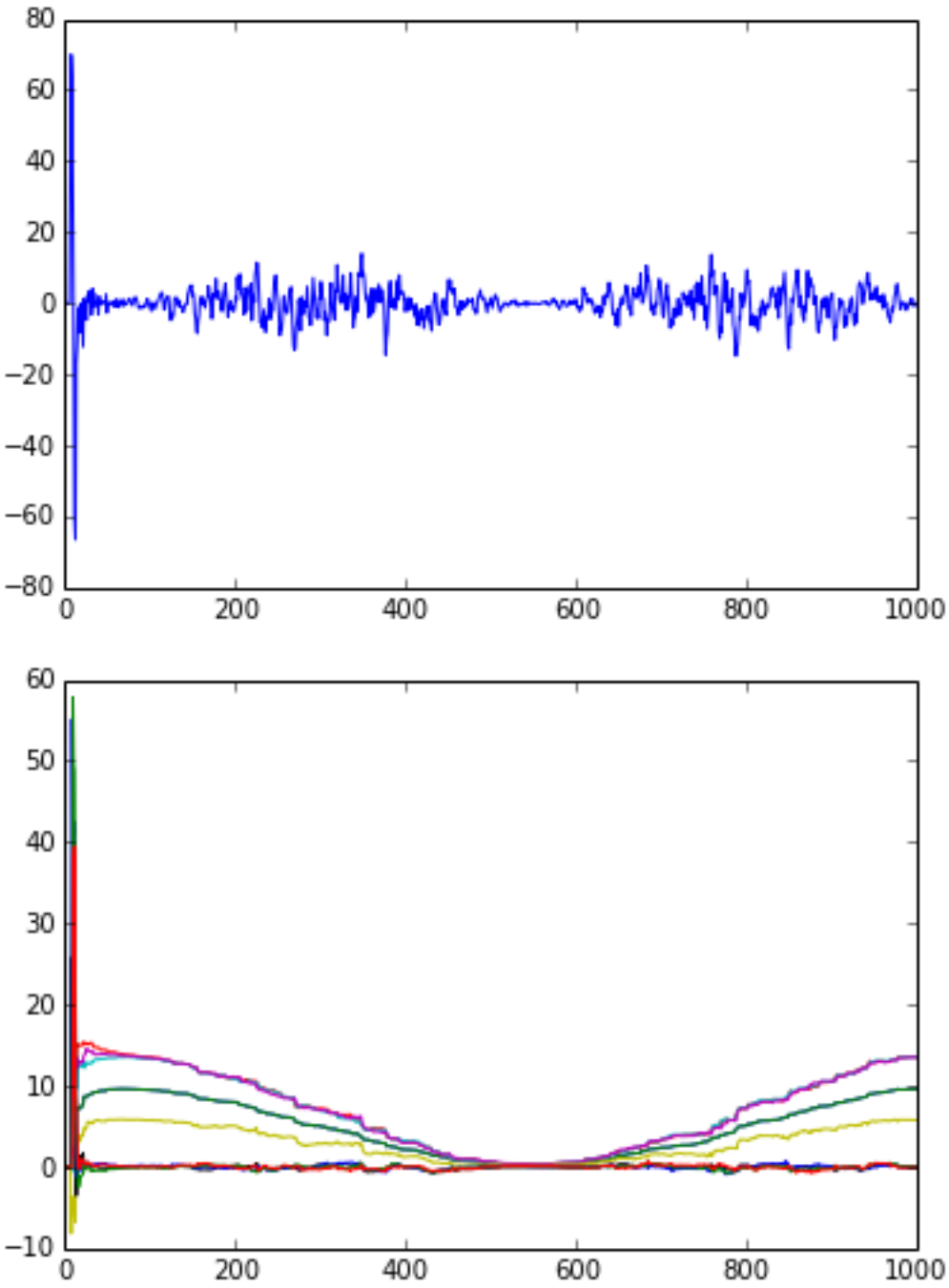
Pour importer les définitions, faire un `from algomcr import *`, ou copiez collez les définitions.

In [133]:
```
# Implantation utilisant le type array
def algo_mcr(u,d,M,plambda):
    N=size(u)
# initialisation
    e=zeros(N)
    wmcr=zeros((M,N+1))
    Kmcr=100*eye(M)
    u_v=zeros(M)
    for n in range(N):
        u_v[0]=u[n]
        u_v[1:M]=u_v[0:M-1]#concatenate((u[n], u_v[1:M]), axis=0)
        e[n]=conj(d[n])-dot(conj(u_v),wmcr[:,n])
        # print("n={}, Erreur de {}".format(n,e[n]))
        Kn=Kmcr/plambda
        Kmcr=Kn-dot(Kn,dot(outer(u_v,conj(u_v)),Kn))/(1+dot(conj(u_v),dot(Kn,u_v)))
        wmcr[:,n+1]=wmcr[:,n]+dot(Kmcr,u_v)*conj(e[n])
    return (wmcr,e)

## MCR, version matricielle

def col(v):
     """ transforme un array en vecteur colonne \n
     l'équivalent du x=x(:) sous Matlab"""
     v=asmatrix(v.flatten())
     return reshape(v,(size(v),1))

def algo_mcr_m(u,d,M,plambda):
    """
    Implantation avec le type matrix plutot que array
    """
    N=size(u)
# initialisation
    e=zeros(N)
    wmcr=matrix(zeros((M,N+1)))
```

```
    Kmcr=100*matrix(eye(M))
    u=col(u)
    u_v=matrix(col(zeros(M)))

    for n in range(N):
        u_v[0]=u[n]
        u_v[1:M]=u_v[0:M-1]
#u_v=concatenate(u[n], u_v[:M], axis=0)
        e[n]=conj(d[n])-u_v.H*wmcr[:,n]
        Kn=Kmcr/plambda
        Kmcr=Kn-Kn*(u_v*u_v.H*Kn)/(1+u_v.H*Kn*u_v)
        wmcr[:,n+1]=wmcr[:,n]+Kmcr*u_v*conj(e[n])

    return (wmcr,e)
```

In [134]:
```
# Implantation matricielle, c'est plus light
def ident_mcr(observation,input_data,facteur_lambda=0.95,p=20):
    """ Identification d'une réponse impulsionnelle à partir de l'observation
    'observation' de sa sortie et de son entrée 'input_data' \n
    'mu' est le pas d'adaptation \n
    Par défaut l'ordre est pris à p=20 \n
    et la condition initiale est prise, par défaut, à h_initial=zeros(20)
    """
    N=size(input_data)
    input_data=squeeze(input_data) #reshape(input_data,(N))
    observation=squeeze(observation)
    (wmcr,e)= algo_mcr(input_data,observation,p,facteur_lambda)
#    (w[:,t+1],erreur[t],yest[t])=lms(input_data[t:t-p:-1],w[:,t],mun)
    return (wmcr,e)

def ident_mcr_m(observation,input_data,facteur_lambda=0.95,p=20):
    """ Identification d'une réponse impulsionnelle à partir de l'observation
    'observation' de sa sortie et de son entrée 'input_data' \n
    'mu' est le pas d'adaptation \n
    Par défaut l'ordre est pris à p=20 \n
    et la condition initiale est prise, par défaut, à h_initial=zeros(20)
    """
    N=size(input_data)
    input_data=squeeze(input_data) #reshape(input_data,(N))
    observation=squeeze(observation)
    (wmcr,e)= algo_mcr_m(input_data,observation,p,facteur_lambda)
#    (w[:,t+1],erreur[t],yest[t])=lms(input_data[t:t-p:-1],w[:,t],mun)
    return (wmcr,e)

lamb=0.98
(w,e)=ident_mcr_m(y,u,facteur_lambda=lamb,p=10)
figure(1)
clf()
plot(e,label="$\lambda={}$".format(lamb))
figure(2)
clf()
plot(array(w).T)
```

Out [134]:
```
[<matplotlib.lines.Line2D at 0x7f605fb33cd0>,
 <matplotlib.lines.Line2D at 0x7f605fb380d0>,
 <matplotlib.lines.Line2D at 0x7f605fb38390>,
 <matplotlib.lines.Line2D at 0x7f605fb385d0>,
 <matplotlib.lines.Line2D at 0x7f605fb38810>,
 <matplotlib.lines.Line2D at 0x7f605fb38a50>,
 <matplotlib.lines.Line2D at 0x7f605fb38c90>,
 <matplotlib.lines.Line2D at 0x7f605fbec850>,
 <matplotlib.lines.Line2D at 0x7f605fb0ab90>,
 <matplotlib.lines.Line2D at 0x7f605fb0a4d0>]
```



Soustraction de bruit.

Le signal que l'on cherche à estimer est s, à partir du mélange $x = s + b$, et de la référence bruit u. Vous cherchez à retrouver le signal s. Vous appliquerez donc une structure de soustraction de bruit, où le filtre sera identifié de manière adaptative à l'aide d'un algorithme LMS. Vous disposez pour vos expérimentations, de signaux tests contenus dans les fichiers `sb1.npz` et `sb2.npz`. Le premier fichier contient deux références bruit, l'une stationnaire, l'autre non stationnaire, que vous considérerez successivement. Vous prendrez des valeurs de pas comprises entre 0.01 et 1. Pour le second signal test, contenu dans le fichier `sb2.npz`, on a une non-stationnarité beaucoup plus importante et un rapport signal-à-bruit très défavorable. Il pourra être utile de traiter le problème en << deux passes >>, afin d'obtenir une première estimée de la réponse impulsionnelle du filtre, qui sera utilisée lors de la seconde << passe >>.

Implantation :

Vous implanterez une fonction avec la syntaxe d'appel suivante :

```
def sousb(ref,signal,h_ini,mu):
#
# Algorithme de soustraction de bruit :
# Entrées :
#   ref : reference bruit seul
#   signal : voie signal (signal composite s +b, ou b est correllé avec b
#   h_ini : réponse impulsionnelle initiale
#   mu : pas d'adaptation
# Sortie :
#   s : signal identifié par soustraction de bruit
#   h : réponse impulsionnelle identifiée
```

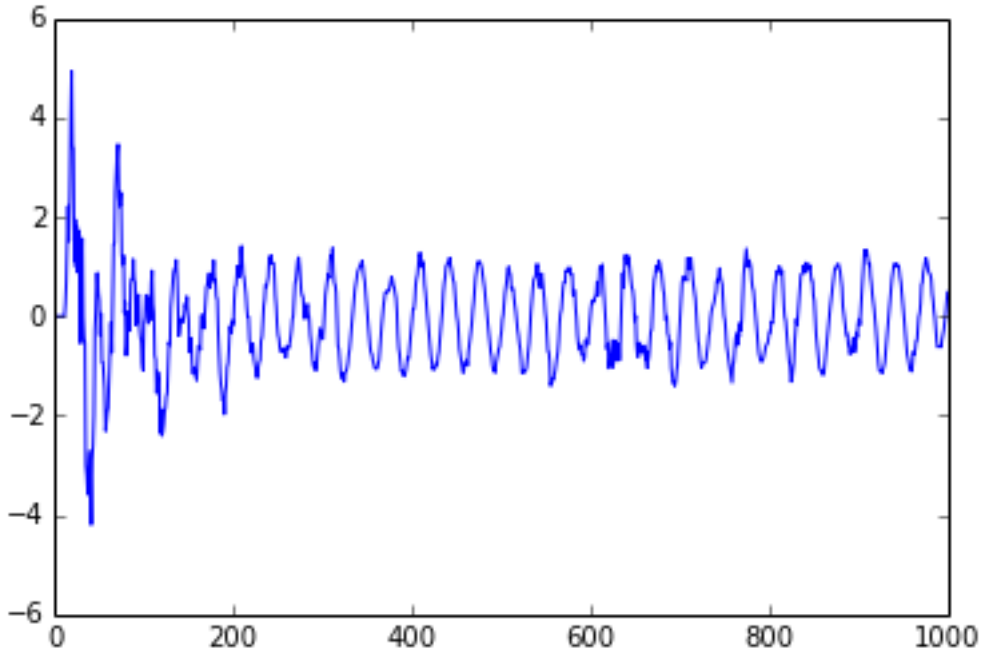
et qui revoie le tuple

```
return (s,h)
```

In [135]:
```
def sousb(ref,signal,h_ini,mu,normalized=False):
#
# Algorithme de soustraction de bruit :
# Entrées :
#  ref : reference bruit seul
# signal : voie signal (signal composite s +b, ou b est corrélé avec b
# h_ini : réponse impulsionnelle initiale
#  mu : pas d'adaptation
# Sortie :
# s : signal identifié par soustraction de bruit
# h : réponse impulsionnelle identifiée
#

    ref=squeeze(ref)
    signal=squeeze(signal)
    h_ini=squeeze(h_ini)

    N=size(signal)
    L=size(h_ini)
    h=h_ini
    s=zeros(N)
    s_est=zeros(N)

    for t in range(L,N):
         u=ref[t:t-L:-1]
         if normalized:
            mun=mu/(dot(u,u)+1e-10)
         else:
            mun=mu
         (h, s[t],s_est[t]) = lms(signal[t], u, h, mun)

    return (h,s,s_est)
```

In [136]:
```
# Test: Simulation d'un problème
N=1000
u=randn(N)
b=lfilter(ones(10),1,u)
s=sin(2*pi*0.03*arange(N))
x=s+b
(h,s,s_est)=sousb(u,x,zeros(12),0.2,normalized=True)
figure()
plot(s)
print(h)
```

```
[ 1.08691022  1.09314388  1.11139925  1.11195871  1.11222574
1.09621258
  1.09233792  1.07112179  1.0478783   1.02922039  0.01959866
0.02322366]
```



Il faut commencer par charger le problème, par :

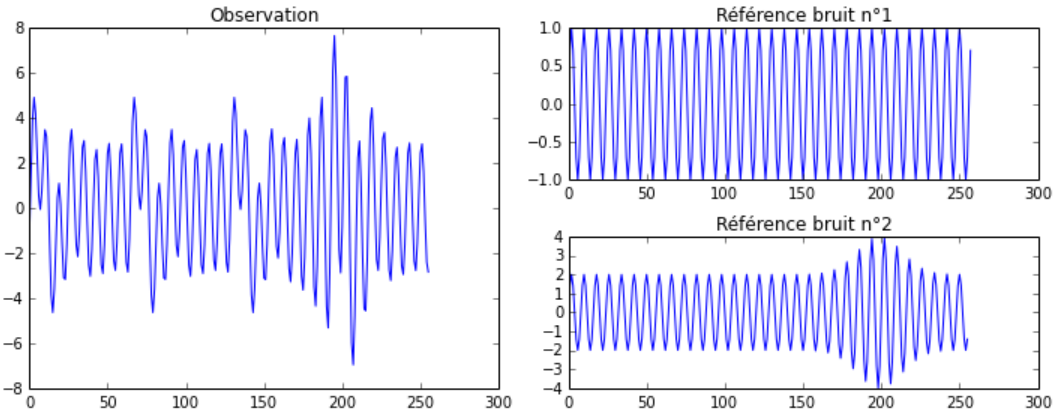
```
f=numpy.load('sb1.npz')
```

In [137]:
```
f=numpy.load('sb1.npz')
# le contenu est donné par
f.keys()
```

Out [137]:
```
['ref2', 'ref1', 'obs']
```

In [138]:
```
# On affecte à des variables locales
obs=f['obs']
ref1=f['ref1']
ref2=f['ref2']
```

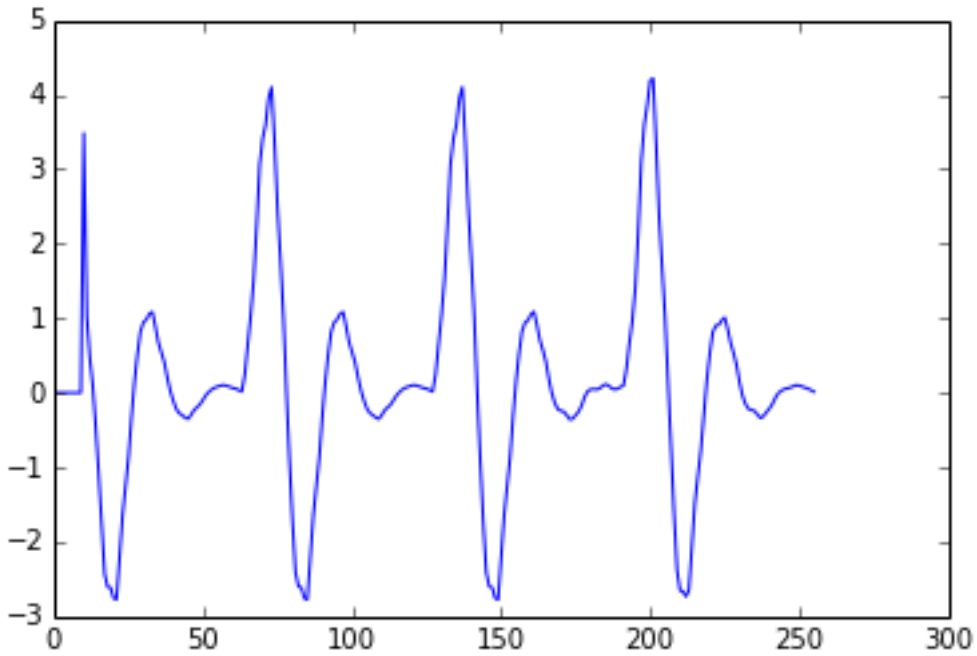
In [139]:
```
import matplotlib.gridspec as gridspec
G = gridspec.GridSpec(2, 2)

fig=figure(figsize=(10,4))
ax1 = subplot(G[0:, 0])
ax1.plot(obs)
title("Observation")
ax2 = subplot(G[0, 1])
ax2.plot(ref1)
title("Référence bruit n°1")

ax3 = subplot(G[1, 1])
ax3.plot(ref2)
title("Référence bruit n°2")
fig.tight_layout()    # évite le recouvrement des titles et labels
```

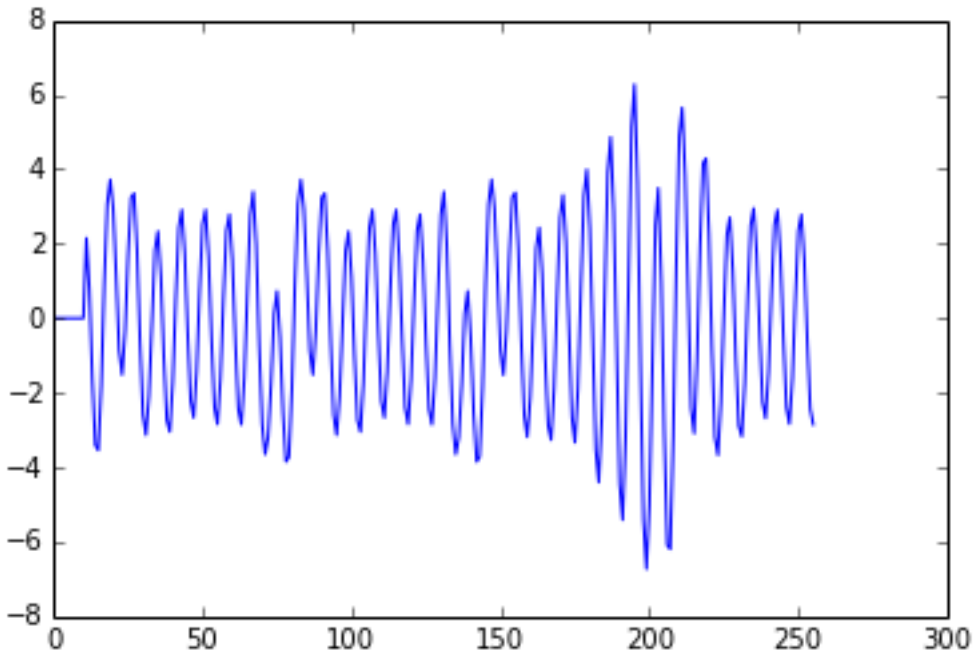


In [140]:
```
# Avec la référence bruit non stationnaire
(h,s,sest)=sousb(ref2,obs,zeros(10),0.8,normalized=True)
plot(s)
figure()
plot(sest)
```

```
[<matplotlib.lines.Line2D at 0x7f606014a210>]
```

Out [140]:

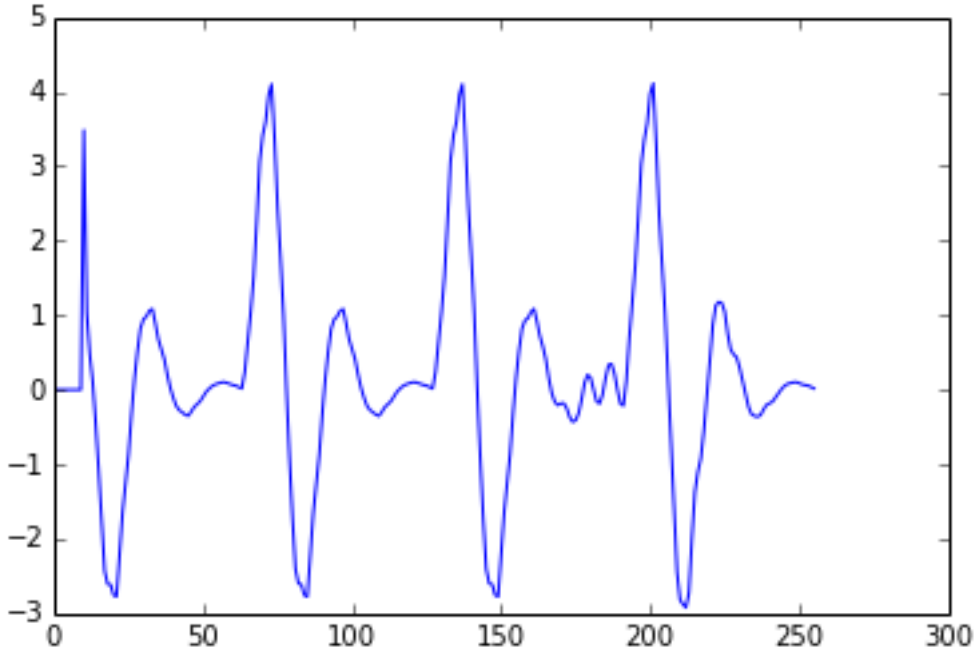


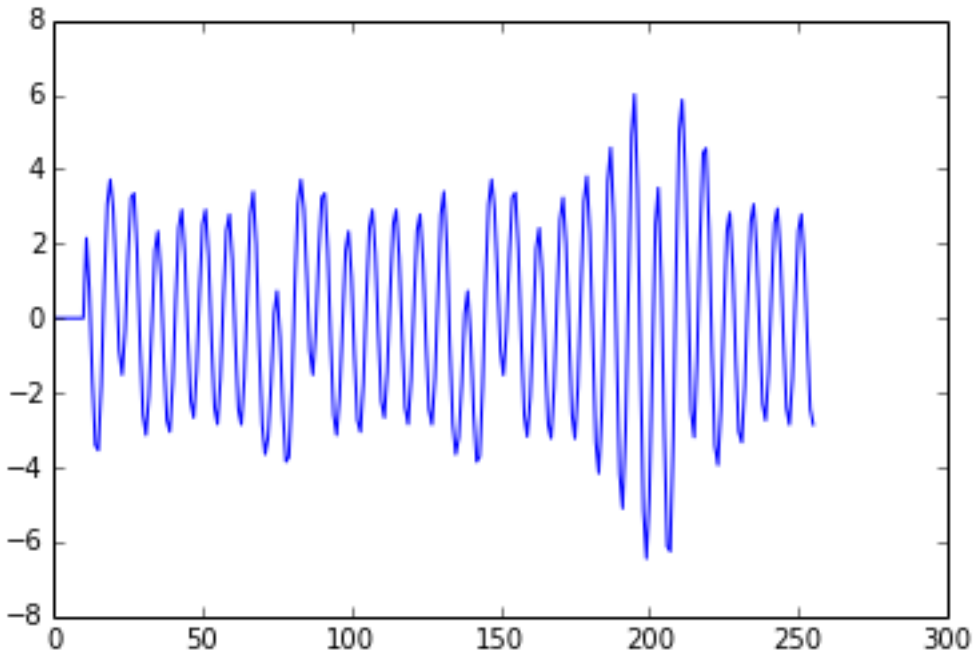


In [141]:
```
# Avec la ref bruit stationnaire -- problème un peu plus difficile
(h,s,sest)=sousb(ref1,obs,zeros(10),0.8,normalized=True)
plot(s)
figure()
plot(sest)
```

```
[<matplotlib.lines.Line2D at 0x7f60605764d0>]
```

Out [141]:





In [142]:
```
f=numpy.load('sb2.npz')
# le contenu est donné par
f.keys()
```

```
['Texte', 'x', 'y']
```

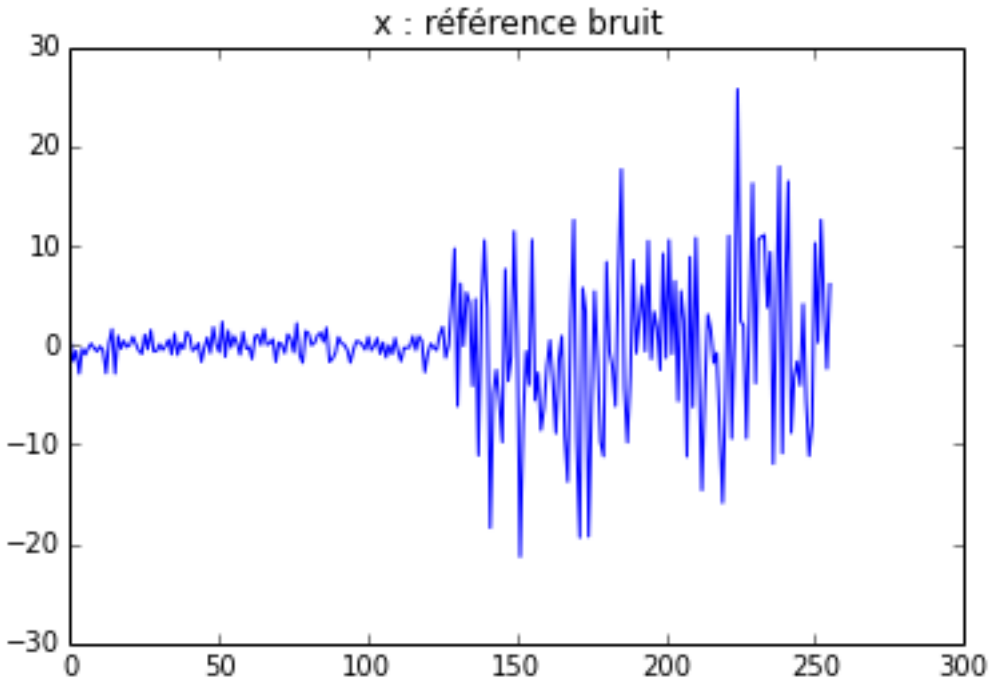
Out [142]:

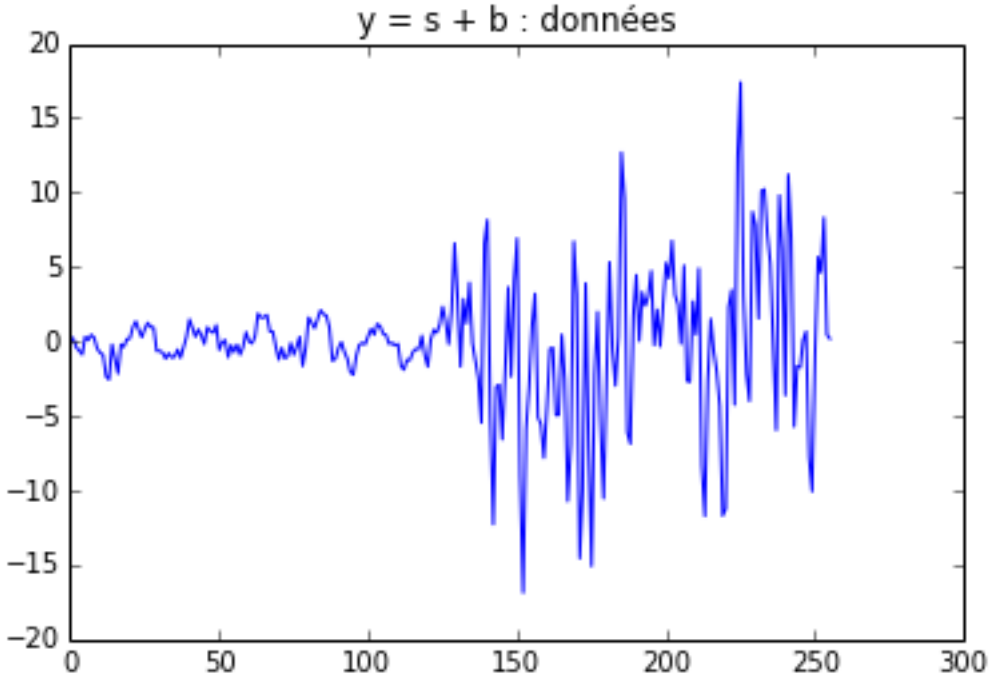
In [143]:
```
# On affecte à des variables locales
Texte=f['Texte']
x=f['x']
y=f['y']
```

In [144]:
```
print(Texte)
plot(x)
title("x : référence bruit")
figure()
plot(y)
title("y = s + b : données")
```

```
['x : reference bruit    ' 'y = s + b : donnees    ' 'b = h*x
'
 'sujet :  h ? s ? b ?  ']
<matplotlib.text.Text at 0x7f605f9e5110>
```

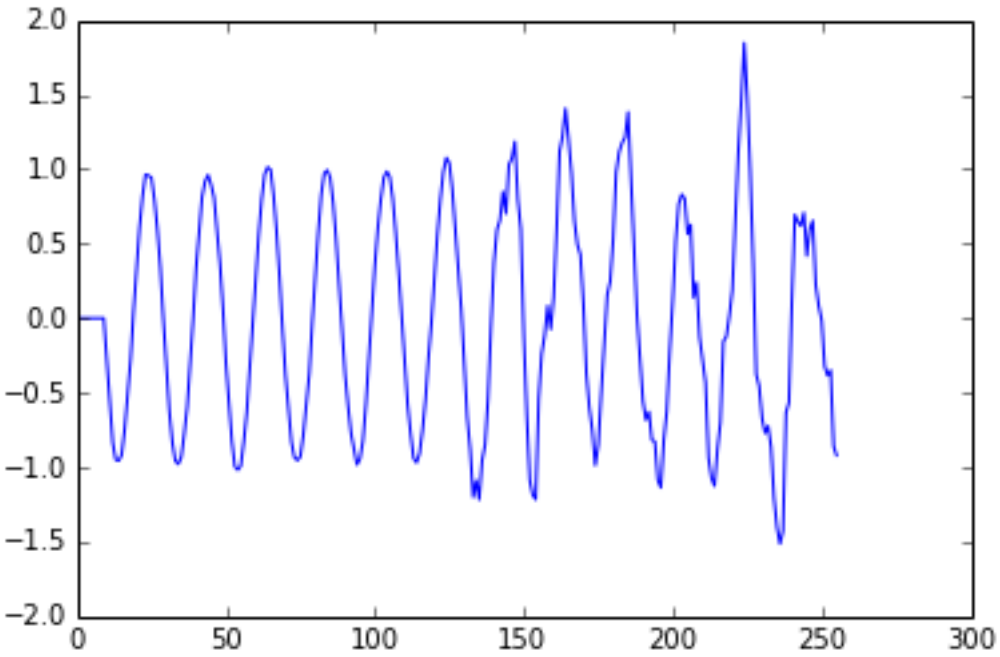
Out [144]:

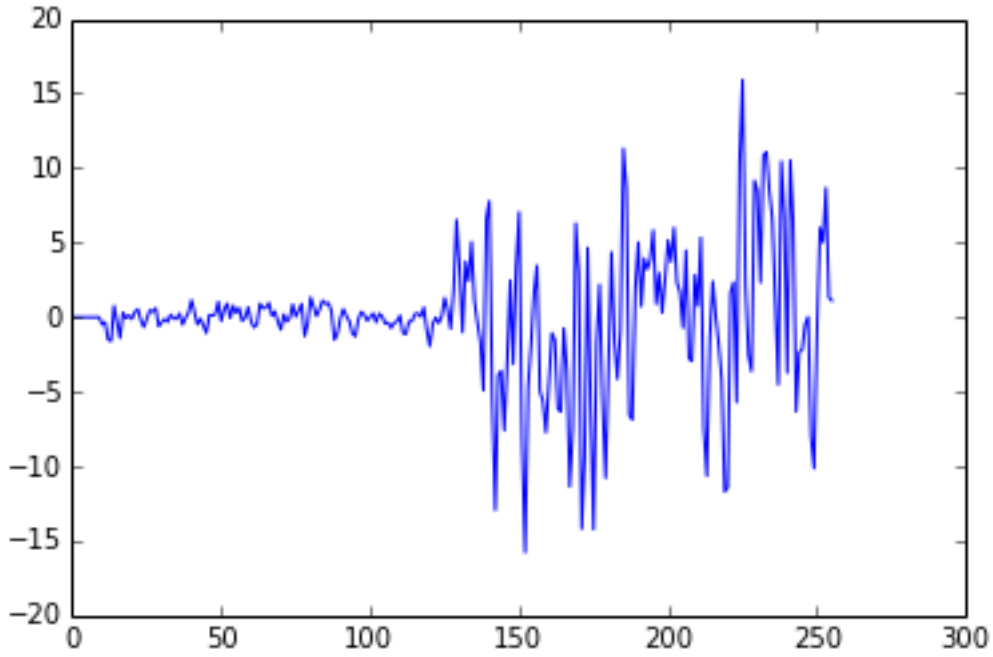




In [145]:
```
(h,s,sest)=sousb(x,y,zeros(10),0.8,normalized=True)
(h,s,sest)=sousb(x,y,h,0.1,normalized=True)
# On imagine que le filtre lui même est stationaire, et on refiltre avec un filtre d
(h,s,sest)=sousb(x,y,h,0.0001,normalized=True)
plot(s)
figure()
plot(sest)
print(h)
```

```
[ 0.41178786  0.59118808  0.00341047  0.00528776  0.0229309
0.02170033
  0.02053261  0.00828565  0.00401322 -0.00740649]
```





Parole bruitée

On termine avec la parole bruitée par un grillon

1. On charge les données par

```
f=numpy.load('parole_bruitee.npz')
g=numpy.load('decticelle.npz')
```

puis les variables sont affectées par

```
d=f['d']
u=g['u']
```

2. Effectuer la soustraction de bruit. Vous pourrez ensuite écouter le résultat avec la fonction `sound` qui a été bricolée par votre serviteur.

In [146]:
```
f=numpy.load('parole_bruitee.npz')
# le contenu est donné par
print(f.keys())
g=numpy.load('decticelle.npz')
# le contenu est donné par
g.keys()
```

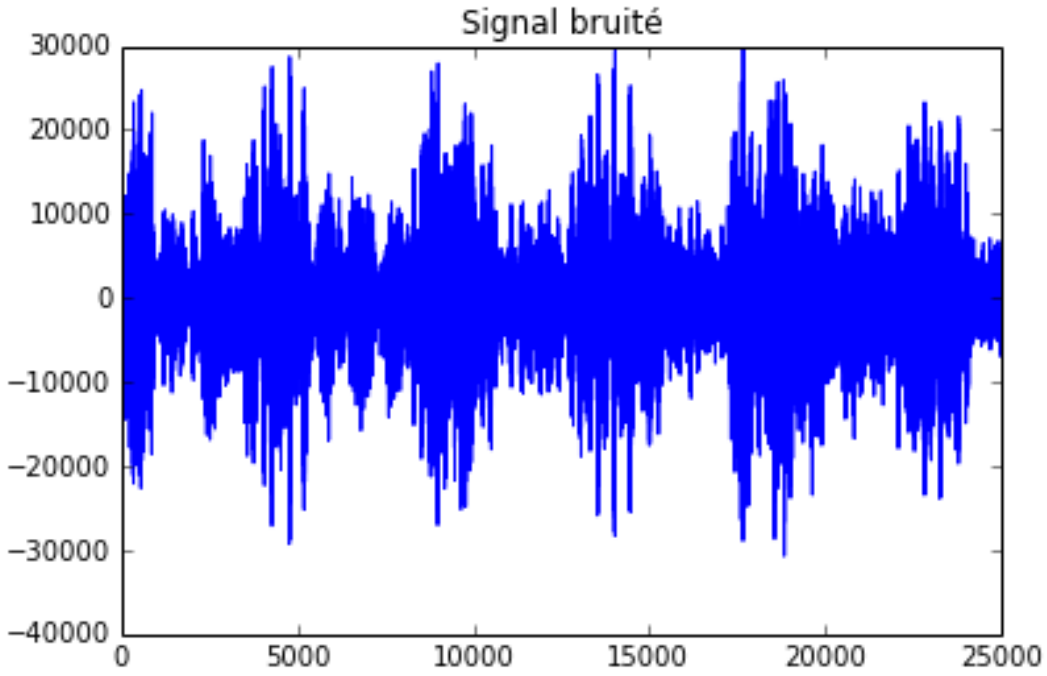
```
['d']
['u']
```

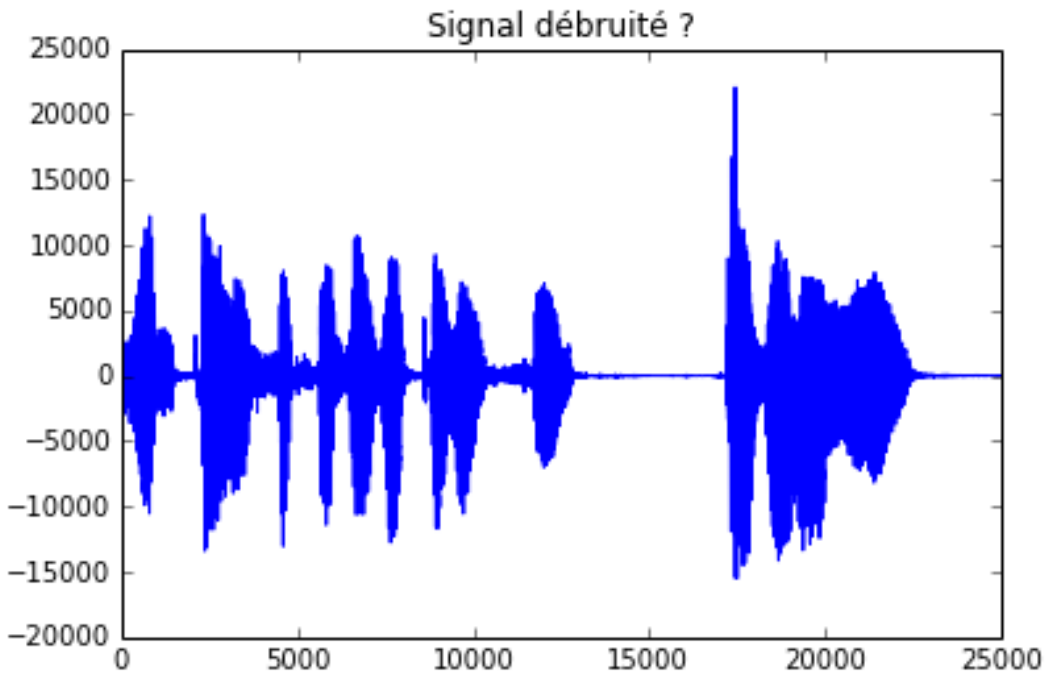
Out [146]:

In [147]:
```
d=f['d']
u=g['u']
(h,s,sest)=sousb(u,d,zeros(5),0.1,normalized=True)
plot(d)
title("Signal bruité")
figure()
plot(s)
title("Signal débruité ?")
```

```
<matplotlib.text.Text at 0x7f60601d7e10>
```

Out [147]:



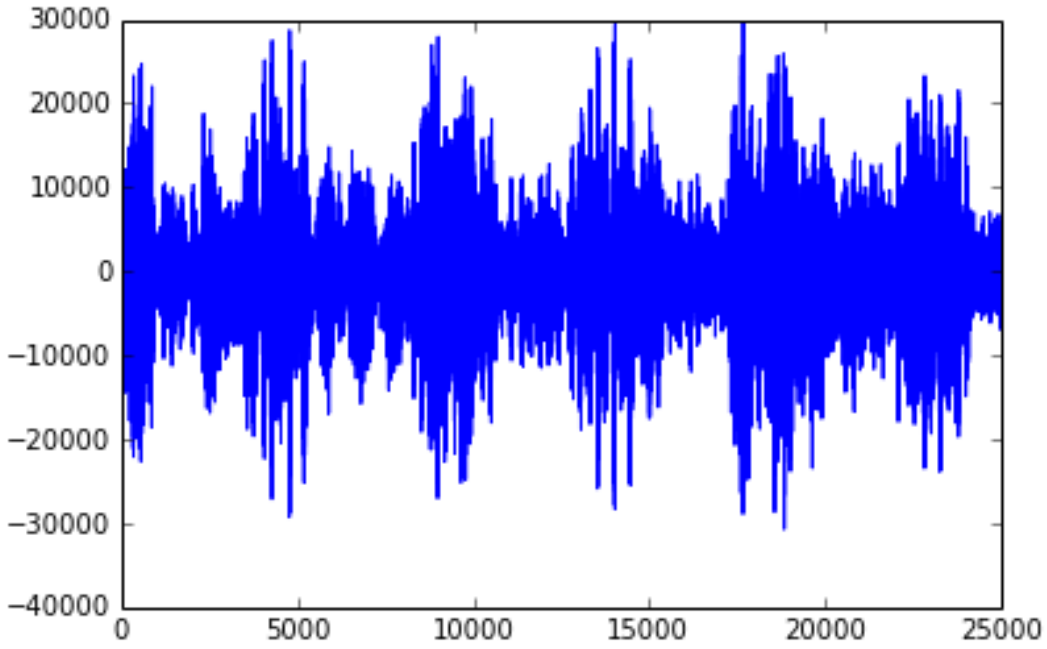


In [148]:
```
def sysfileopen(filepath):
    import subprocess, os
    if sys.platform.startswith('darwin'):
        subprocess.call(('open', filepath))
    elif os.name == 'nt':
            os.startfile(filepath)
    elif os.name == 'posix':
#            subprocess.call(('xdg-open', filepath))
            subprocess.Popen(["xdg-open", filepath])

# Import useful read-write for wav files
from scipy.io.wavfile import read as wavread, write as wavwrite

def sound(var):
    from scipy.io.wavfile import write as wavwrite
    scaled = np.int16(var/np.max(np.abs(var)) * 32767)
    stmp=asarray(scaled,dtype=np.int16)
    wavwrite('stmp.wav',8820,stmp)
    sysfileopen("stmp.wav")
```

In [149]:
```
sound(d)
```

In [150]:
```
plot(d)
```

Out [150]:
```
[<matplotlib.lines.Line2D at 0x7f60601c1350>]
```



In [151]:
```
sound(s)
```

FIN.