Elements on Adaptive Filters

J.-F. Bercher

March 15, 2014

Contents

Author: J.-F. Bercher
06 décembre 2013
English version: february 20, 2014
Last update: march 07, 2014

The goal of this lab is to illustrate and consolidate some concepts on adaptive filtering. First, we study the problem anatycally; then we will experiment and study the probleme in simulation. In particular –since one should do it at least one time, you will implement and use a LMS algorithm. You will study the convergence properties, the role of the adaptation step, etc. We will consider an identification problem and then several noise cancellation problems.

```
%config InlineBackend.figure_format = 'svg'
```

1 LMS Algorithm

Implement a LMS algorithme. The call syntax should be:

```
def lms(d,u,w,mu):
 """
 Implements a single iteration of the stochastic gradient (LMS)\n
 :math:`w(n+1)=w(n)+\\mu u(n)\\left(d(n)-w(n)^T u(n)\\right)`
 Input:
 ======
     d : wanted sequence at time n \n
     u : vecteur de longueur p des chantillons d'entre \n
     w : wiener filter to update \n
     mu : adaptation step
 Outputs:
 =======
     w : upated filter
     error : y-yest
     dest : prediction = :math:`u(n)^T w`
 """
```

```
def lms(d,u,w,mu):
    """
    Implements a single iteration of the stochastic gradient (LMS)\n
```

```
    :math:`w(n+1)=w(n)+\\mu u(n)\\left(d(n)-w(n)^T u(n)\\right)  `


    Input:
    ======
        d : wanted sequence at time n \n
        u : vecteur de longueur p des échantillons d'entrée \n
        w : wiener filter to update \n
        mu : adaptation step

    Outputs:
    ========
        w : upated filter
        err : y-yest
        dest : prediction = :math:`u(n)^T w`
    """
    u=squeeze(u) #Remove single-dimensional entries from the shape
        of an array.
    w=squeeze(w)
    dest=u.dot(w)
    err=d-dest
    w=w+mu*u*err
    return (w,err,dest)
```

2 Filter Identification

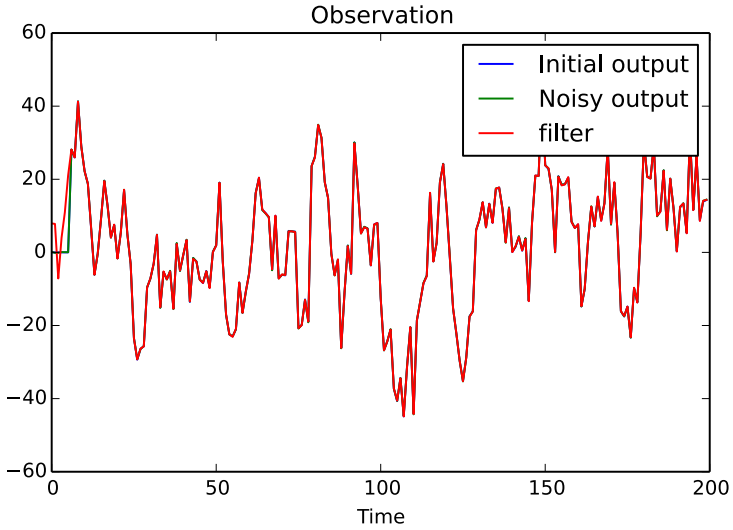
You will test this algorithm on an **identification problem**. You will use as a test signal the output of a filter excited by a white noise, according to

```
# 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)
```

```
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)
```

```
#Figures of the test system
figure()
t=arange(size(y))
plot(t,yo, label='Initial output')
plot(t,y, label='Noisy output')
plot(t,y2, label='filter')
title('Observation')
xlabel('Time')
legend()
```

```
<matplotlib.legend.Legend at 0x7fe298cc6cd0>
```



In order to implement the identification procedure, one should simply observe that we look for a filter, which excited by the same $x(n)$ should yield an output $z(n)$ as similar as $y_0(n)$ as possible. One thus take `u=x`, and `d=y`(the wanted sequence is $y_0(n)$, which shall be substituted by $y(n)$ – since y_0 is unknown).

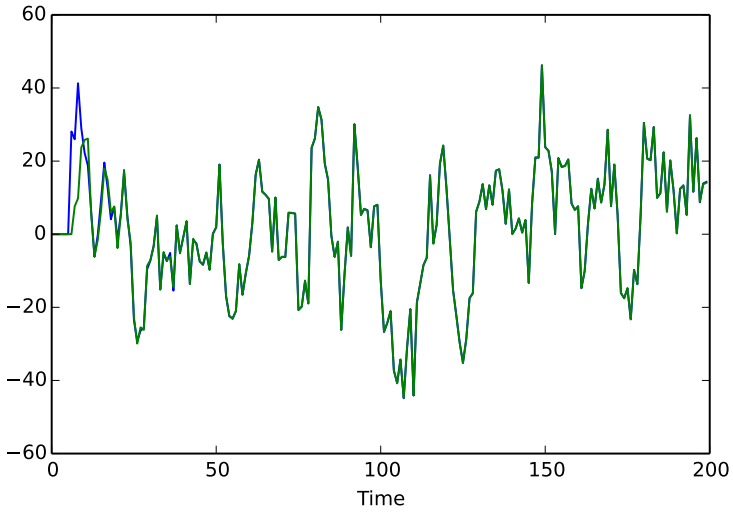
Implementation:

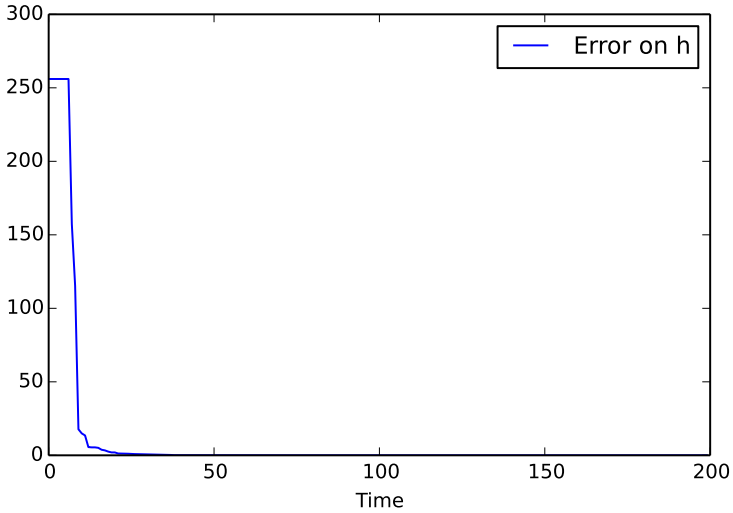
- Begin by some direct commands (initializations and a `for` loop on the time variable) for identifying the filter; once this works you will implement th commands as a function `ident`
  - If necessary, the function `squeeze()` enable to remove single-dimensional entries from the shape of an n-D array (*e.g.* transforms an array (3,1,1) into a vector of demension 3)

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

```
figure(1)
tt=range(200)
plot(tt,yo,label='Initial Noiseless Output')
plot(tt,yest, label="Estimated Output")
xlabel('Time')

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

```
<matplotlib.text.Text at 0x7fe298e8c1d0>
```





- Implement now the identication procedure as a function `ident` which uses a loop on the `lms` algorithm, called with the correct parameters. Syntax:

```
def ident(observation,input_data,mu,p=20,h_initial=zeros(20),normalized=False):
    """
    Identification of an impulse response from an observation
    'observation' of its output, and from its input 'input_data' \n
    'mu' is the adaptation step\n
    Inputs:
    =======
    observation: array
        output of the filter to identify
    input_data: array
        input of the filter to identify
    mu: real
        adaptation step
    p: int (default =20)
        order of the filter
    h_initial: array (default h_initial=zeros(20))
        initial guess for the filter
    Outputs:
    ========
    w: array
        identified impulse response
    err: array
        estimation error
    yest: array
        estimated output
    """
```

```
def ident(observation,input_data,mu,p=20,h_initial=zeros(20),
    normalized=False):
    """ Identification of an impulse response from an observation
    'observation' of its output, and from its input 'input_data' \n
    'mu' is the adaptation step\n
    Inputs:
    =======
    observation: array
        output of the filter to identify
    input_data: array
        input of the filter to identify
    mu: real
        adaptation step
    p: int (default =20)
        order of the filter
    h_initial: array (default h_initial=zeros(20))
        initial guess for the filter
    Outputs:
    ========
    w: array
        identified impulse response
```

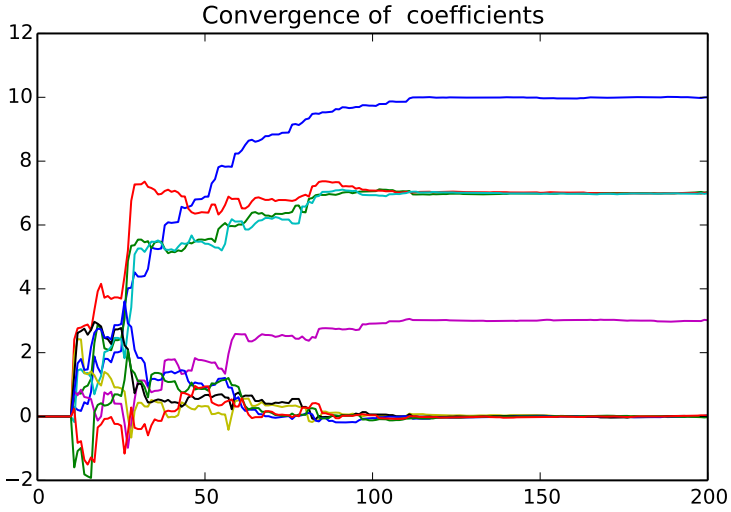
```
    err: array
        estimation error
    yest: array
        estimated output
    """
    N=size(input_data)
    input_data=squeeze(input_data) #reshape(input_data,(N))
    observation=squeeze(observation)
    err=zeros(N)
    w=zeros((p,N+1))
    yest=zeros(N)

    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])+1
                e-10)
        else:
            mun=mu
        (w[:,t+1],err[t],yest[t])=lms(observation[t],input_data[t:t-
            p:-1],w[:,t],mun)

    return (w,err,yest)
```

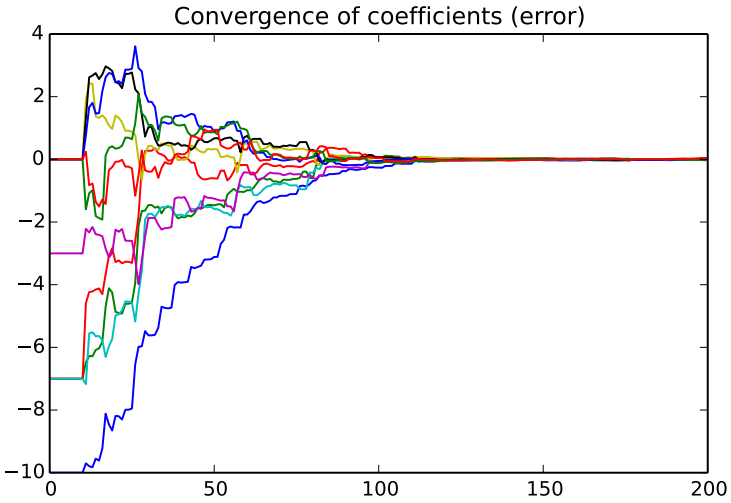
In order to evaluate the algorithm behaviour, you will plot the estimation error, the evolution of the coefficients of the identified filter during the iterations of the algorithm; and finally the quadratic error between the true filter and the identified one. This should be done for several orders p (the exact order is unknown...) and for different values of the adaptation step μ.

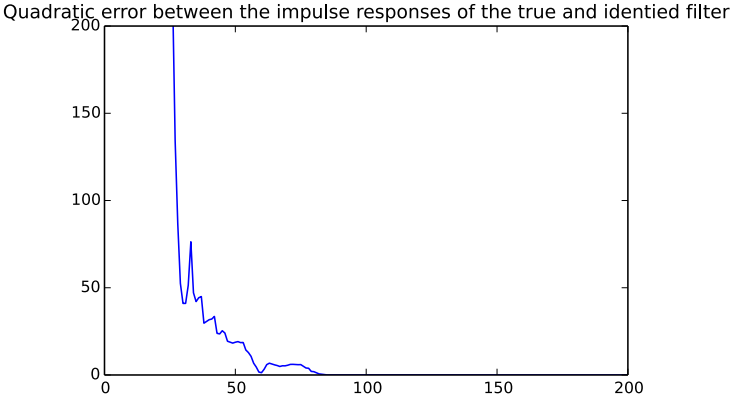
- The quadratic error can be evaluated simply thanks to a *comprehension list* according to Errh=[sum(he-w[:,n])**2 for n in range(N+1)]

```
p=10
(w,err,yest)=ident(y,x,mu=0.05,p=p, h_initial=zeros(p))
figure(2)
plot(w.T)
title("Convergence of  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 of coefficients (error)")
# or
figure(4)
Errh=[sum(he-w[:,n])**2 for n in range(N+1)]
plot(Errh)
title("Quadratic error between the impulse responses of the true and
    identied filter")
ylim([0, 200])
```

(0, 200)


Convergence of coefficients
12
10
8
6
4
2
0
−2
0
50
100
150
200

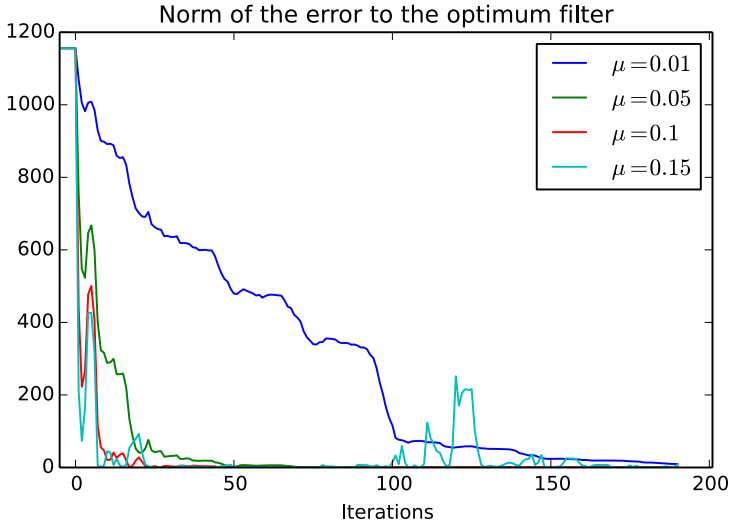

Convergence of coefficients (error)
4
2
0
−2
−4
−6
−8
−10
0
50
100
150
200


Quadratic error between the impulse responses of the true and identied filter
200
150
100
50
0
0
50
100
150
200

Study with respect to μ

```
# Study with respect to $\mu$
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("Norm of the error to the optimum filter")
xlabel("Iterations")
```

```
<matplotlib.text.Text at 0x7fe2988a1650>
```



Normalized LMS In the normalized LMS, the adaptation step can be computed automatically according to

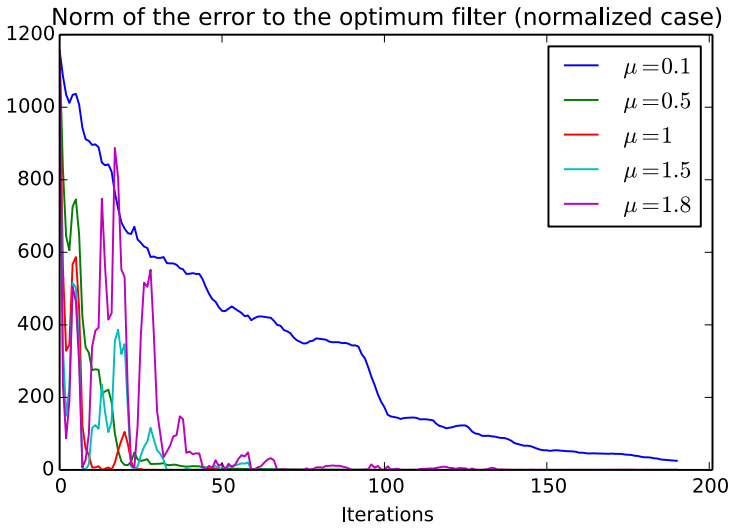
$$\mu = \frac{1}{u(n)^T u(n) + \epsilon}$$

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

```
# Normalized LMS
figure()
for mu in [0.1, 0.5, 1, 1.5, 1.8 ]:
    (w,err,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("Norm of the error to the optimum filter (normalized case)")
xlabel("Iterations")
```

```
<matplotlib.text.Text at 0x7fe298cb77d0>
```



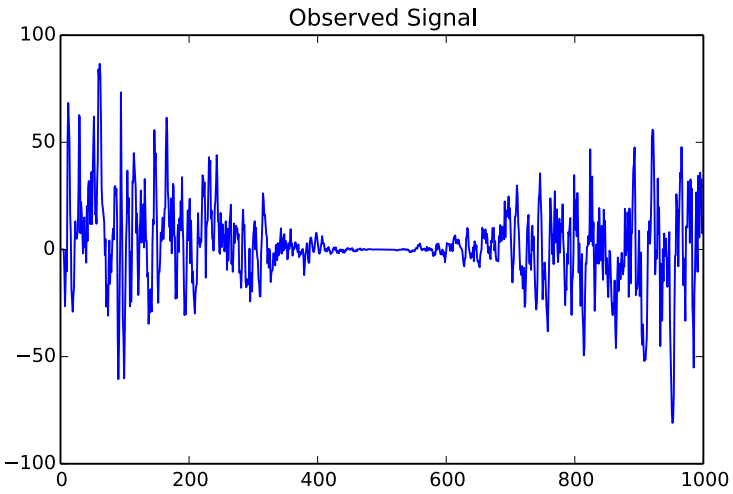
Study the convergence speed with respect to the adaptation step, by introducing a slow non-stationarity in the signal, for instance according to

```
### Slow non-stationarity
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)
```

```
### Slow non-stationarity

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("Observed Signal")
```

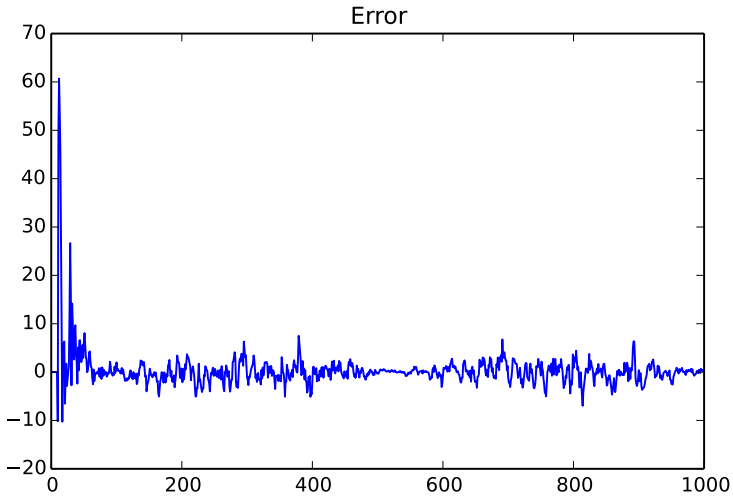
```
<matplotlib.text.Text at 0x7fe2984e4610>
```

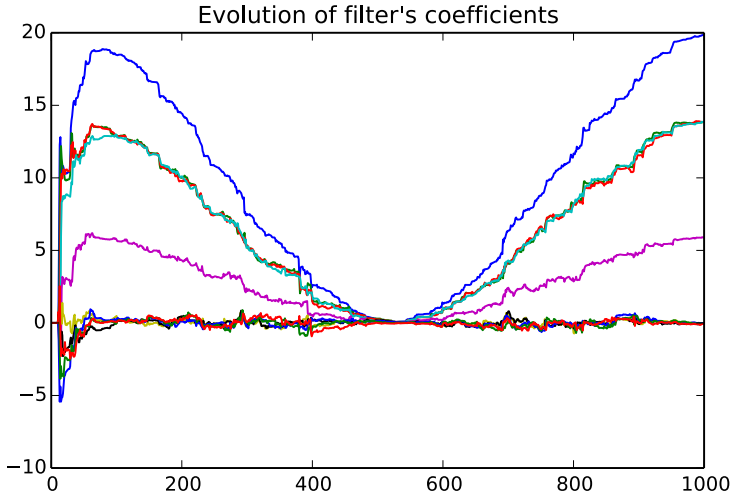


We use the LMS algorithm on the non-stationary signal.

```
p=10
(w,err,yest)=ident(y,u,mu=0.05,p=p,h_initial=zeros(p))
figure(1)
clf()
plot(err)
title('Error')
figure(2)
clf()
plot(w.T)
title("Evolution of filter's coefficients")
```

```
<matplotlib.text.Text at 0x7fe298367390>
```





RLS Algorithm Kindly, we offer you an implementation of the Recursive Least Squares algorithm. Restart the previous experimentations (identification with non stationary data) with the RLS algorithm. Compare and conclude.

In order to import the definitions, include a `from algorls import *`, or cut and paste these definitions.

```
# Implementation using the array type
def algo_rls(u,d,M,plambda):
    N=size(u)
# initialization
    e=zeros(N)
    wrls=zeros((M,N+1))
```

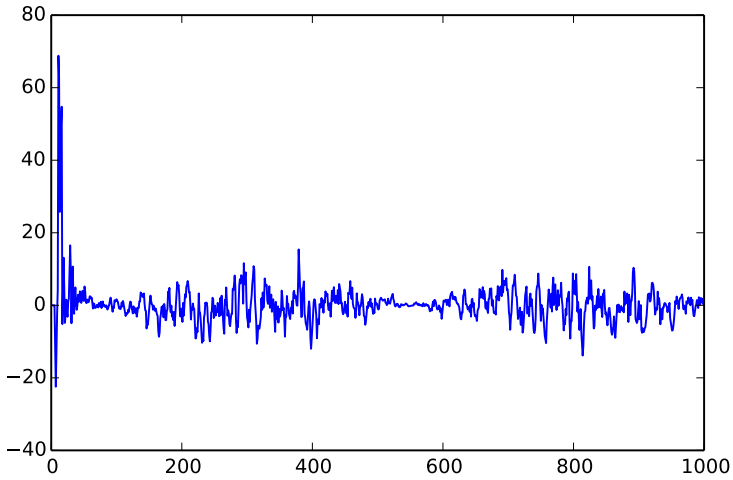
```
    Krls=100*eye(M)
    u_v=zeros(M)
    for n in range(N):
        u_v[0]=u[n]
        u_v[1:M]=u_v[0:M-1]#concatenate((u[n], u_v[1:M]), axis=0)
        e[n]=conj(d[n])-dot(conj(u_v),wrls[:,n])
        # print("n={}, Erreur de {}".format(n,e[n]))
        Kn=Krls/plambda
        Krls=Kn-dot(Kn,dot(outer(u_v,conj(u_v)),Kn))/(1+dot(conj(u_v
            ),dot(Kn,u_v)))
        wrls[:,n+1]=wrls[:,n]+dot(Krls,u_v)*conj(e[n])
    return (wrls,e)

## RLS, matrix version

def col(v):
    """ transforms an array into a column vector \n
    This is the equivalent of  x=x(:) under Matlab"""
    v=asmatrix(v.flatten())
    return reshape(v,(size(v),1))

def algo_rls_m(u,d,M,plambda):
    """
    Implementation with the matrix type instead of the array type
    """
    N=size(u)
        # initialization
    e=zeros(N)
    wrls=matrix(zeros((M,N+1)))
    Krls=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*wrls[:,n]
        Kn=Krls/plambda
        Krls=Kn-Kn*(u_v*u_v.H*Kn)/(1+u_v.H*Kn*u_v)
        wrls[:,n+1]=wrls[:,n]+Krls*u_v*conj(e[n])

    return (wrls,e)
```

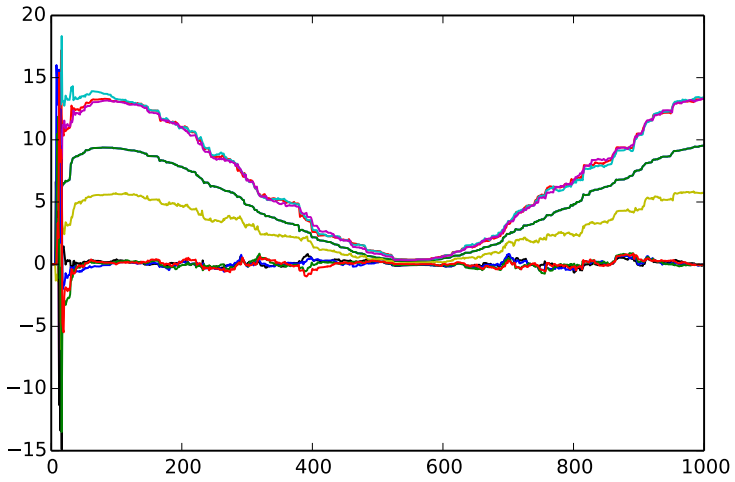
```
# Implementation of the identification procedure
# First using an array type, then using matrix type
def ident_rls(observation,input_data,factor_lambda=0.95,p=20):
    """ Identification of an impulse response from an observation
    'observation' of its output, and from its input 'input_data' \n
    'mu' is the adaptation step\n
    Inputs:
    =======
```

```
    observation: array
        output of the filter to identify
    input_data: array
        input of the filter to identify
    factor_lambda: real (defaut value=0.95)
        forguetting factor in the RLS algorithm
    p: int (default =20)
        order of the filter
    Outputs:
    ========
    w: array
        identified impulse response
    err: array
        estimation error
    yest: array
        estimated output
    """
    N=size(input_data)
    input_data=squeeze(input_data) #reshape(input_data,(N))
    observation=squeeze(observation)
    (wrls,e)= algo_rls(input_data,observation,p,factor_lambda)
#   (w[:,t+1],erreur[t],yest[t])=lms(input_data[t:t-p:-1],w[:,t],mun
    )
    return (wrls,e)

def ident_rls_m(observation,input_data,factor_lambda=0.95,p=20):
    """ Identification of an impulse response from an observation
    'observation' of its output, and from its input 'input_data' \n
    'mu' is the adaptation step\n
    Inputs:
    =======
    observation: array
        output of the filter to identify
    input_data: array
        input of the filter to identify
    factor_lambda: real (defaut value=0.95)
        forguetting factor in the RLS algorithm
    p: int (default =20)
        order of the filter
    Outputs:
    ========
    w: array
        identified impulse response
    err: array
        estimation error
    yest: array
        estimated output
    """
    N=size(input_data)
    input_data=squeeze(input_data) #reshape(input_data,(N))
    observation=squeeze(observation)
    (wrls,e)= algo_rls_m(input_data,observation,p,factor_lambda)
#   (w[:,t+1],erreur[t],yest[t])=lms(input_data[t:t-p:-1],w[:,t],mun
    )
```

```
    return (wrls,e)

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

```
[<matplotlib.lines.Line2D at 0x7fe297481dd0>,
 <matplotlib.lines.Line2D at 0x7fe2974a0210>,
 <matplotlib.lines.Line2D at 0x7fe2974a04d0>,
 <matplotlib.lines.Line2D at 0x7fe2974a0710>,
 <matplotlib.lines.Line2D at 0x7fe2974a0950>,
 <matplotlib.lines.Line2D at 0x7fe2974a0b90>,
 <matplotlib.lines.Line2D at 0x7fe2974a0dd0>,
 <matplotlib.lines.Line2D at 0x7fe2983974d0>,
 <matplotlib.lines.Line2D at 0x7fe2976afdd0>,
 <matplotlib.lines.Line2D at 0x7fe2976af250>]
```





3 Noise cancellation

Let s be the signal we want to estimate from the mixture $x = s+b$, and from the noise reference u. You will apply a noise cancellation procedure, wher the filter will be identified in an adaptive way, using a LMS algorithm. For your experiments, you have to consider the tests signals included in the files `sb1.npz` and `sb2.npz`. The first file contains two noise references, one is stationary, the second non stationary. You will consider these two references successively. You will take values between 0.01 and 1 for the adaptation step. For the second signal in `sb2.npz`, we have a much more important non stationarity and a lower signal-to-noise ratio. It might be useful to consider the problem in two phases, so as to obtain a frst estimate of the impulse response, which will be used as an initial condition in the second phase.

Implementation : You will implement a function following the call syntax

```
def noisecancel(ref,signal,h_ini,mu,normalized=False):
    """
    Noise cancellation algorithm
    Inputs:
    =======
    ref: array
        noise reference
    signal:  array
        signal path (signal mixture s +b, where b is correlated with ref
    h_ini: array
        initial impulse response
    mu: adaptation step
    Outputs:
    ========
    h: array
        identified impulse response
    s: array
        signal identified by noise cancellation
    b_est: array
```

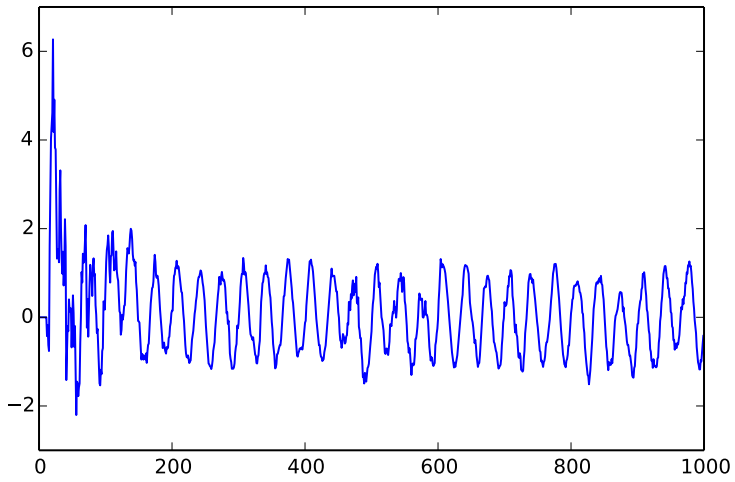
```
        noise on the signal path estimated from ref
    """
```

```
def noisecancel(ref,signal,h_ini,mu,normalized=False):
    """
    Noise cancellation algorithm
    Inputs:
    =======
    ref: array
        noise reference
        signal:  array
        signal path (signal mixture s +b, where b is correlated with
            ref
    h_ini: array
        initial impulse response
    mu: adaptation step
    Outputs:
    ========
    h: array
        identified impulse response
    s: array
        signal identified by noise cancellation
        b_est: array
        noise on the signal path estimated from ref
    """


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

    N=size(signal)
    L=size(h_ini)
    h=h_ini
    s=zeros(N)
    b_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],b_est[t]) = lms(signal[t], u, h, mun)

    return (h,s,b_est)
```

```
# Test: Problem Simulation
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)=noisecancel(u,x,zeros(12),0.2,normalized=True)
figure()
plot(s)
print(h)
```

```
[ 1.01007675  1.02532588  1.023375    1.04335271  1.03992215  1.05137742
  1.03441274  1.04257964  1.02475112  1.00860837 -0.01031707 -0.02301002]
```



We must first load the data:

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

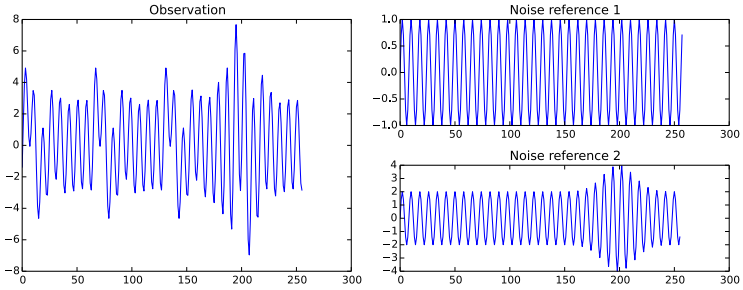
```
f=numpy.load('sb1.npz')
# f is a dictionary
# its keys are given by
f.keys()
```

```
['ref2', 'ref1', 'obs']
```

```
# Then the contents are affected to local variables
obs=f['obs']
ref1=f['ref1']
ref2=f['ref2']
```

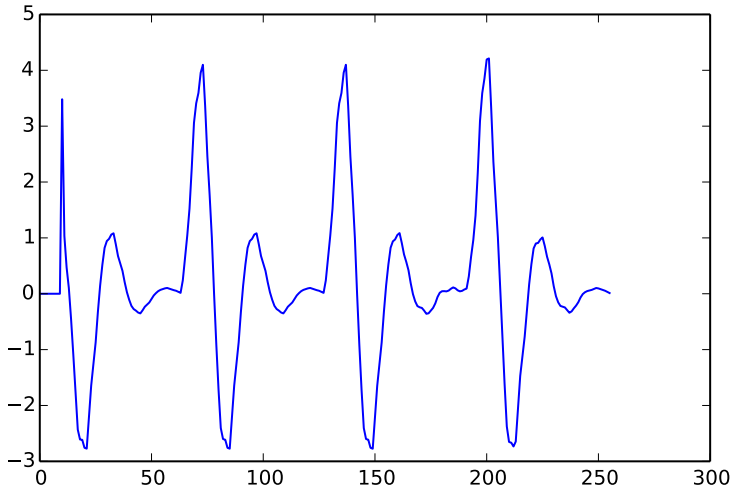
```
import matplotlib.gridspec as gridspec
G = gridspec.GridSpec(2, 2)

fig=figure(figsize=(10,4))
```

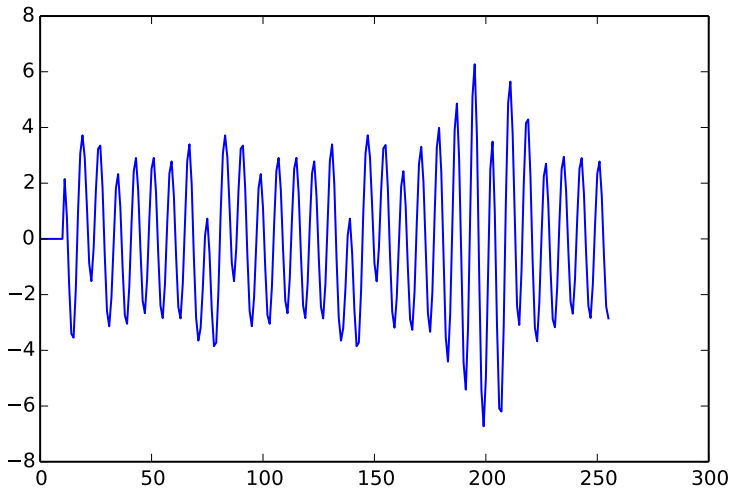
```
ax1 = subplot(G[0:, 0])
ax1.plot(obs)
title("Observation")
ax2 = subplot(G[0, 1])
ax2.plot(ref1)
title("Noise reference 1")

ax3 = subplot(G[1, 1])
ax3.plot(ref2)
title("Noise reference 2")
fig.tight_layout()    #avoid covering of titles and labels
```




```
# With the non stationary reference
(h,s,sest)=noisecancel(ref2,obs,zeros(10),0.8,normalized=True)
plot(s)
figure()
plot(sest)
```

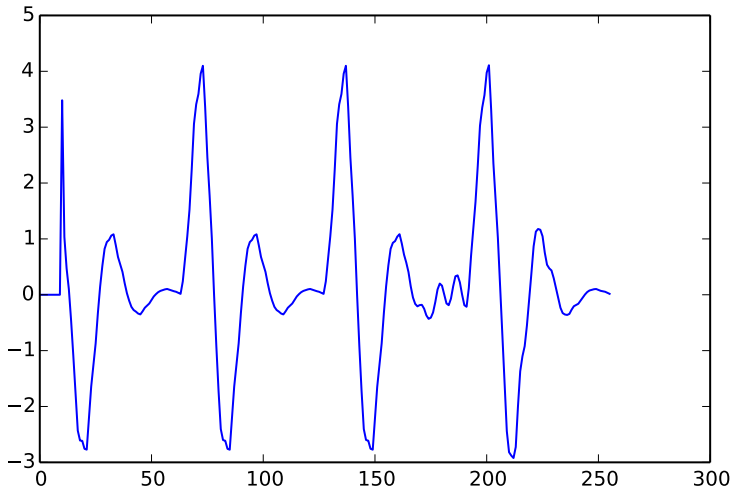
```
[<matplotlib.lines.Line2D at 0x7fe297669dd0>]
```

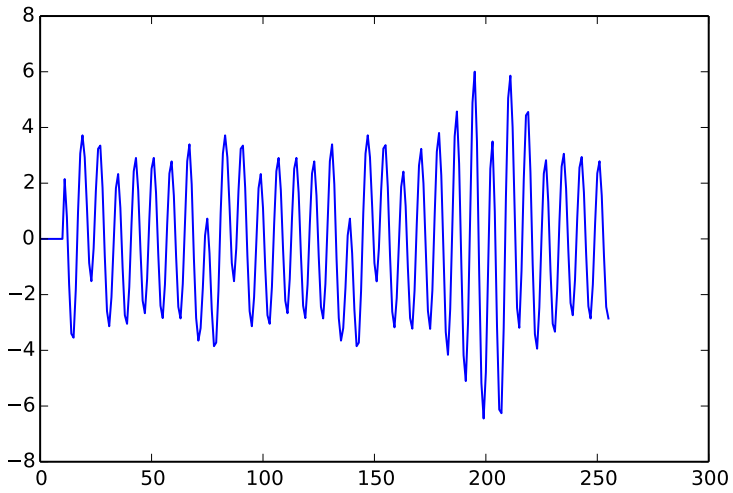





```
# With the stationary noise reference -- the problem is a bit more
    difficult
(h,s,sest)=noisecancel(ref1,obs,zeros(10),0.8,normalized=True)
plot(s)
figure()
plot(sest)
```

```
[<matplotlib.lines.Line2D at 0x7fe297297c90>]
```






```
f=numpy.load('sb2.npz')
# keys are given by
f.keys()
```

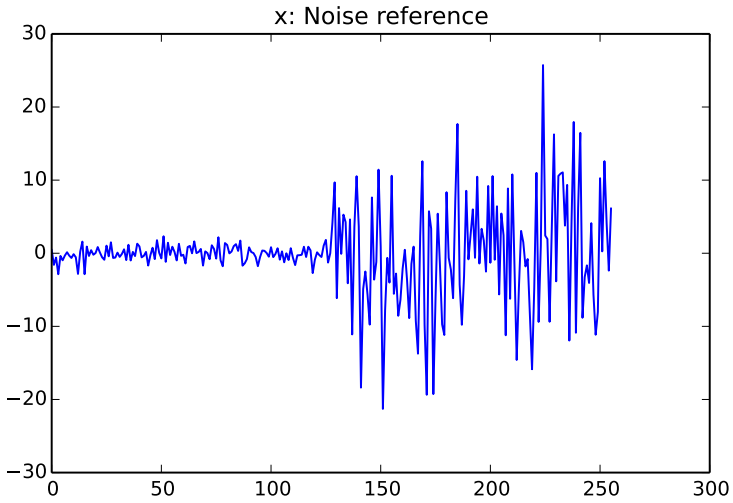
```
['Texte', 'x', 'y']
```

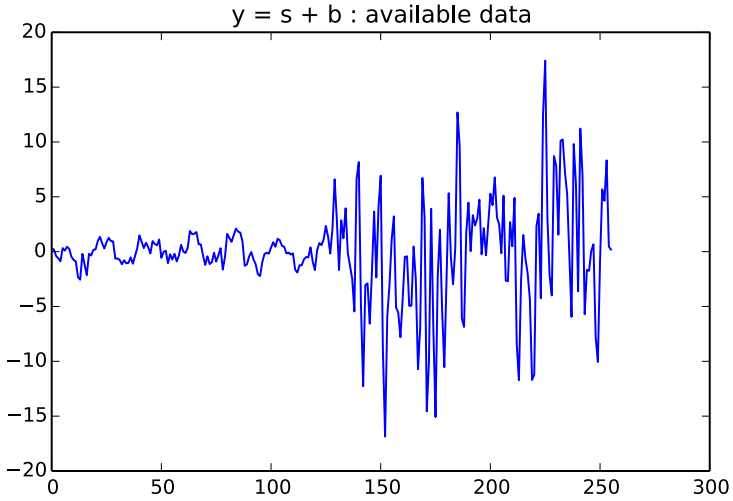
```
#  Then the contents are affected to local variables
Texte=f['Texte']
x=f['x']
y=f['y']
```

```
print(Texte) # in french, sorry
plot(x)
title("x: Noise reference")
figure()
plot(y)
title("y = s + b : available data")
```

```
['x : reference bruit   ' 'y = s + b : donnees   ' 'b = h*x                '
 'sujet :  h ? s ? b ?  ']
```

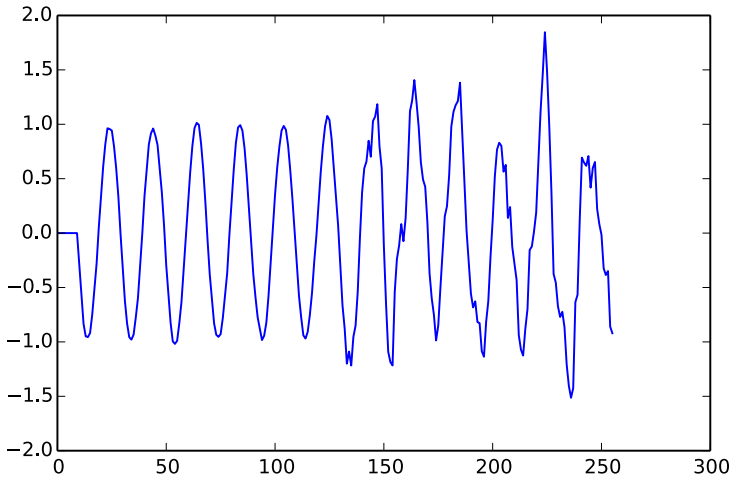
```
<matplotlib.text.Text at 0x7fe297278a90>
```

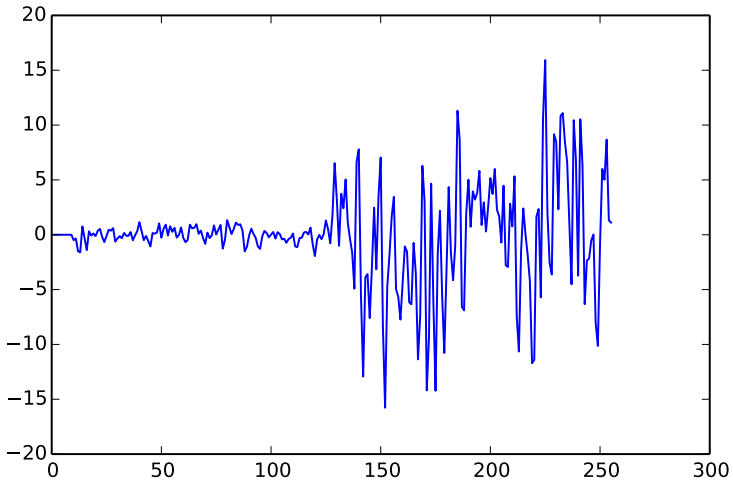





```
(h,s,sest)=noisecancel(x,y,zeros(10),0.8,normalized=True)
(h,s,sest)=noisecancel(x,y,h,0.1,normalized=True)
# We assume that the filter is stationary and filter again
# with this filter as initial condition
(h,s,sest)=noisecancel(x,y,h,0.0001,normalized=True)
plot(s)
figure()
plot(sest)
print(h)
```

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





4 Corrupted speech

We end with a speech sound, corrupted by a cricket chirping

1. We load data by
   f=numpy.load('parole_bruitee.npz')
   g=numpy.load('decticelle.npz')

then the contents are affected to local variables

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

2. Perform the noise cancellation. Then you will be able to listen the result thanks to the `sound` function which has been thrown together by your servant (`from syssound import *`).

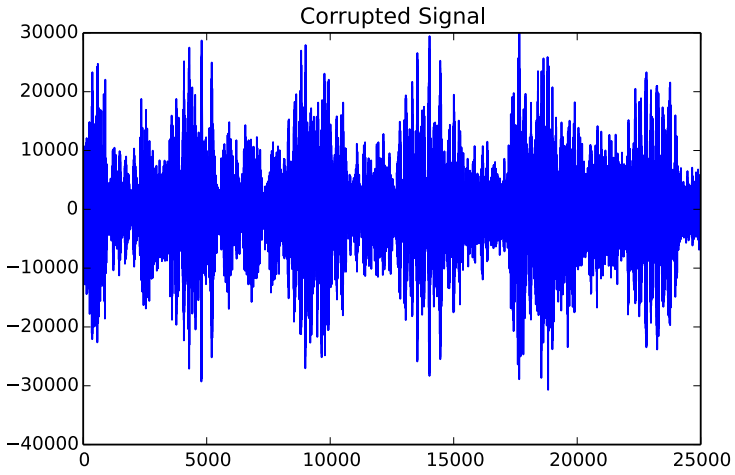
```
f=numpy.load('parole_bruitee.npz')
print(f.keys())
g=numpy.load('decticelle.npz')
g.keys()
```

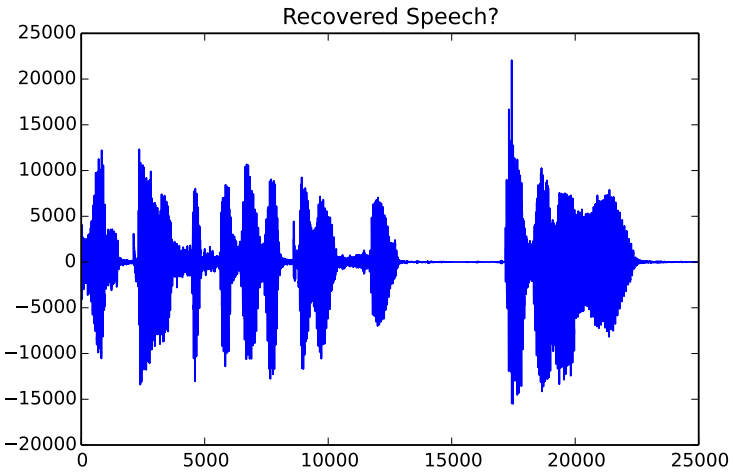
```
['d']
```

```
['u']
```

```
d=f['d']
u=g['u']
(h,s,sest)=noisecancel(u,d,zeros(5),0.1,normalized=True)
plot(d)
title("Corrupted Signal")
figure()
plot(s)
title("Recovered Speech?")
```

```
<matplotlib.text.Text at 0x7fe2972f24d0>
```



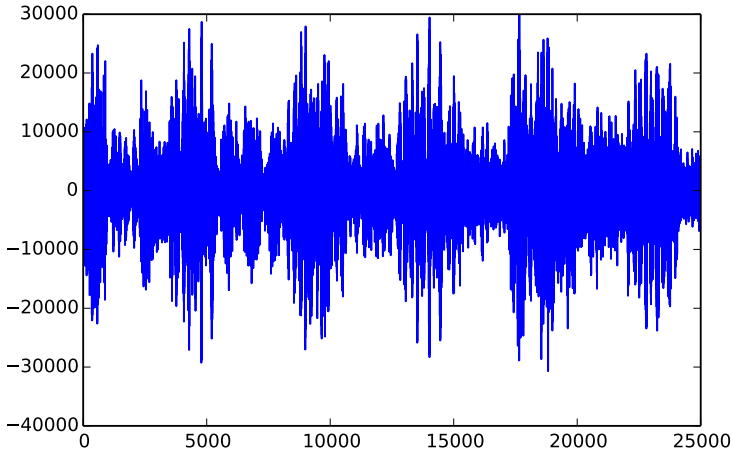



```
def sysfileopen(filepath):
    import subprocess, os
    if sys.platform.startswith('darwin'):
        subprocess.call(('open', filepath))
    elif os.name == 'nt':
            os.startfile(filepath)
    elif os.name == 'posix':
#            subprocess.call(('xdg-open', filepath))
            subprocess.Popen(["xdg-open",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

```
sound(d)
```

```
plot(d)
```

```
[<matplotlib.lines.Line2D at 0x7fe297211910>]
```




```
sound(s)
```

THE END.