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.

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 échantillons d'entrée \n
     w : wiener filter to update \n
     mu : adaptation step
 Outputs:
 =======
     w : upated filter
     error : y-yest
     dest : prediction = :math:`u(n)^T w`
 """
```

1.1 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

```
from scipy.signal import lfilter
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=lfilter(htest,[1],x)+0.01*randn(N)
```

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)

- 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
    """
```

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(hew[:,n])**2 for n in range(N+1)]

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)
```

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)
```

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.

1.2 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
        """
(....)
# 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)
```

For loading the data, you have to do something like

```
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']
```

1.3 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 *`).

THE END.