1 Table of Contents

In [1]:

```
%run nbinit.ipy
```

```
... Configuring matplotlib formats
... Configuring matplotlib with inline figures
... Importing numpy as np, scipy as sp, pyplot as plt, scipy.stats as stats
   ... scipy.signal as sig
... Importing widgets, display, HTML, Image, Javascript
... Some LaTeX definitions
```

(1)

```
... Defining figures captions

... Loading customized Javascript for interactive solutions (show/hide)
... Redefining interactive from ipywidgets
```

Contents

In [2]:

```
import mpld3
mpld3.enable_notebook()
import warnings
warnings.simplefilter('default')
```

2 Versions Adaptatives

L'algorithme du gradient utilise le gradient de l'erreur quadratique moyenne pour rechercher les coefficients du filtre de Wiener. L inconvénients sont que

- cela repose sur la connaissance des vraies statistiques de second ordre (corrélations), alors qu'elles sont évidemment non disponibles;
- le filtre résultant n'est pas adapté à un environnement non stationnaire, puisque les équations normales ont été dérivées da contexte stationnaire.

Afin de prendre en compte ces deux inconvénients, nous devons définir des «estimations des fonctions de corrélation» capables non-stationnarités des signaux. Disposant de telles estimations, nous aurons juste à les insérer dans les équations normales pou version adaptative...

Considérons l'exemple simple où nous devons estimer la puissance d'un signal non stationnaire :

$$\sigma(n)^2 = \mathbb{E}\left[X(n)^2\right].$$

Une solution simple est d'approximer la moyenne d'ensemble comme une moyenne temporelle au voisinage du point n :

$$\sigma_L(n)^2 = \frac{1}{2L+1}\sum_{l=-L}^{L} x(n-l)^2.$$

Ceci correspond au filtrage avec une fenêtre de longueur (rectangulaire) de longueur $2L+1$. Notons qu'il est possible de calcu récursivement

$$\sigma_L(n)^2 = \sigma_L(n-1)^2 + \frac{1}{2L+1}\left(x(n+L)^2 - x(n-L-1)^2\right).$$

Une autre solution consiste à introduire un facteur d'oubli λ qui permet de donner plus de poids aux échantillons les plus récents
les plus anciens. La formule correspondante est

$$\sigma_\lambda(n)^2 = K_n \sum_{l=0}^{n} \lambda^{n-l} x(l)^2,$$

où K_n est un facteur qui assure que l'estimation soit non-biaisée, à savoir $\mathbb{E}\left[\sigma_\lambda(n)^2\right] = \sigma(n)^2$. À titre d'exercice, vous devrie
$K_n = (1-\lambda^{n+1})/(1-\lambda)$. Pour $\lambda < 1$, K_n converge rapidement et nous pouvons le prendre comme une constante. Dans

$$s_\lambda(n)^2 = \sigma_\lambda(n)^2 / K,$$

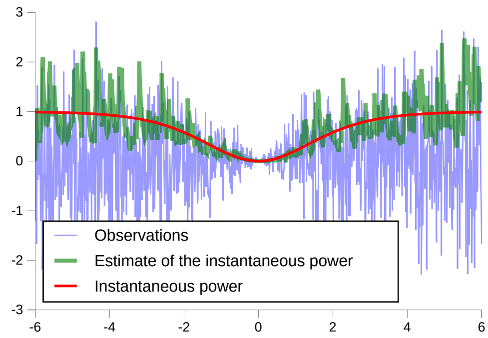
Nous obtenons alors une formule récursive très simple:

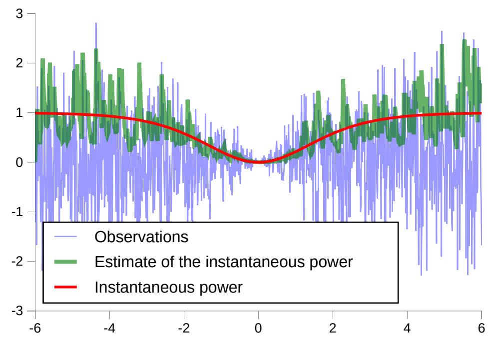
$$s_\lambda(n)^2 = \lambda s_\lambda(n-1)^2 + x(n)^2.$$

Les lignes suivantes simulent un signal non stationnaire avec une puissance variable dans le temps. On applique une moyenne
pour estimer la puissance. Expérimentez avec les valeurs de λ.

In [3]:

```
N = 1000
from scipy.special import expit  # logistic function
x = np.random.normal(size=N)
t = np.linspace(-6, 6, N)
z = x * (2 * expit(t) - 1)


def plt_vs_lambda(lamb):
    plt.plot(t, z, alpha=0.4, label='Observations')
    #We implement $s_\lambda(n)^2 = \lambda s_\lambda(n-1)^2 + x(n)^2.$
    slambda = np.zeros(N)
    for n in np.arange(1, N):
        slambda[n] = lamb * slambda[n - 1] + z[n]**2
    plt.plot(
        t,
        slambda * (1 - lamb),
        lw=3,
        alpha=0.6,
        label='Estimate of the instantaneous power')
    plt.plot(t, (2 * expit(t) - 1)**2, lw=2, label='Instantaneous power')
    plt.legend(loc='best')


lamb = widgets.FloatSlider(min=0, max=1, value=0.8, step=0.01)
_ = interact(plt_vs_lambda, lamb=lamb)
```





In [4]:

```
%matplotlib inline
```

Contents

Revenons à l'équation normale (???):

$$\overset{\triangle}{\mathbf{w}} = \mathbf{R}_{uu}^{-1}\mathbf{R}_{du}$$

Et à sa formulation sous forme itérative (???):

$$\begin{aligned}\mathbf{w}(n+1) &= \mathbf{w}(n) - \mu\mathbb{E}\left[\mathbf{u}(n)e(n)\right] \\ &= \mathbf{w}(n) - \mu\left(\mathbf{R}_{uu}\mathbf{w}(n) - \mathbf{R}_{du}\right)\end{aligned}$$

On va chercher à substituer les valeurs exactes par des valeurs estimées. Une remarque importante est que la solution de l'équa est insensible à un facteur d'échelle sur les estimations. Il est ainsi possible d'estimer la matrice de corrélation et le vecteur en ut moyenne glissante \begin{equation}

$$\begin{cases}\hat{\mathbf{R}}_{uu}(n) = \sum_{l=-L}^{L}\mathbf{u}(n-l)\mathbf{u}(n-l)^H \\ \hat{\mathbf{R}}_{du}(n) = \sum_{l=-L}^{L}\mathrm{d}(n-l)\mathbf{u}(n-l)\end{cases}$$

\end{equation} ou par une moyenne exponentielle \begin{equation}

\hat\Rb*{uu}(n)* = \sum{l=0}^n \lambda^{l-n} \ub(l)\ub(l)^H = \lambda \hat\Rb_{uu}(n-1) + \ub(n)\ub(n)^H \end{equation} ce qui four \begin{equation}

$$\begin{cases}\hat{\mathbf{R}}_{uu}(n) = \sum_{l=0}^{n}\lambda^{l-n}\mathbf{u}(l)\mathbf{u}(l)^H = \lambda\hat{\mathbf{R}}_{uu}(n-1) + \mathbf{u}(n)\mathbf{u}(n)^H \\ \hat{\mathbf{R}}_{du}(n) = \lambda\hat{\mathbf{R}}_{du}(n-1) + d(n)\mathbf{u}(n).\end{cases}$$

\end{equation}

2.1 L'Algorithme Least Mean Squares (LMS)

L'estimateur le plus simple que l'on peut définir est le cas limite où nous ne faisons pas la moyenne du tout... C'est-à-dire que no $L = 0$ ou $\lambda = 0$ dans les formules précédentes, pour obtenir les `estimations instantanées` \begin{equation}

$$\begin{cases}\hat{\mathbf{R}}_{uu}(n) = \mathbf{u}(n)\mathbf{u}(n)^H \\ \hat{\mathbf{R}}_{du}(n) = d(n)\mathbf{u}(n).\end{cases}$$

\end{equation} Cela consiste simplement à supprimer les espérances mathématiques dans les formules théoriques. Ainsi, on ob formules qui dépendent directement des données, sans avoir besoin de connaître quelque chose sur les statistiques théoriques, dépendent également du temps, conférant ainsi une adaptabilité à l'algorithme. En insérant ces estimations dans les itérations d du gradient, nous obtenons

$$\begin{aligned}\mathbf{w}(n+1) &= \mathbf{w}(n) - \mu\mathbf{u}(n)e(n) \\ &= \mathbf{w}(n) - \mu\mathbf{u}(n)\left(\mathbf{u}(n)^H\mathbf{w}(n) - d(n)\right)\end{aligned}$$

Substituer $\mathbf{u}(n)\mathbf{u}(n)^H$ $\mathbb{E}\left[\mathbf{u}(n)\mathbf{u}(n)^H\right]$ ou $\mathbf{u}(n)e(n)$ $\mathbb{E}\left[\mathbf{u}(n)e(n)\right]$ est une approximation absolument grossière. Néanmoir se fait via le processus d'itération de sorte que ce type de méthode finalemnt fonctionne. L'algorithme LMS est de loin l'algorithm adaptatif le plus couramment utilisé, car il est extrêmement simple à mettre en œuvre, a une charge de calcul très faible, fonctior relativement bien et possède des capacités de poursuite.

Afin d'illustrer le comportement de l'algorithme LMS, poursuivons l'exemple de l'identification d'un système inconnu. Commençor par recréer les données:

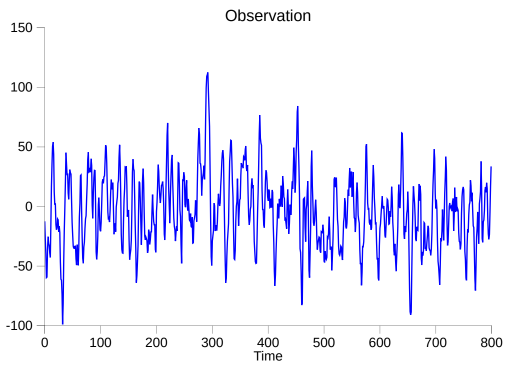
2.2 Illustation du LMS dans un problème d'identification

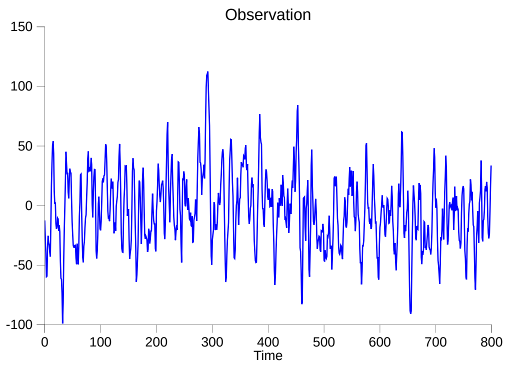
Contents

In [5]:

```
from scipy.signal import lfilter
# test
figplot = True
N = 800
x = lfilter([1, 1], [1], np.random.randn(N))
htest = 10 * np.array([1, 0.7, 0.7, 0.7, 0.3, 0])
y0 = lfilter(htest, [1], x)
y = y0 + 0.1 * randn(N)
if figplot:
    plt.plot(y)
    plt.xlabel("Time")
    plt.title("Observation")
    figcaption("System output in an identification problem")
```

Caption: System output in an identification problem





Contents

Maintenant, comme chacun devrait faire ce type d'exercice au moins une fois, essayez de mettre en oeuvre un algorithme LMS. une fonction avec la syntaxe suivante:

In [6]:

```
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 : desired sequence at time n
        u : input of length p
        w : wiener filter to update
        mu : adaptation step

    Returns:
    =======
        w : upated filter
        err : d-dest
        dest : prediction = :math:`u(n)^T w`
    """
    dest = 0
    err = d - dest
    #
    # DO IT YOURSELF!
    #
    return (w, err, dest)
```

You may test your function using the following validation:

In [7]:

```
np.random.seed(327)
wout, errout, destout = lms(
    np.random.normal(1), np.random.normal(6), np.zeros(6), 0.05)
wtest = np.array(
    [0.76063565, 0.76063565, 0.76063565, 0.76063565, 0.76063565, 0.76063565])
#Test
if np.shape(wout) == np.shape(wtest):
    if np.sum(np.abs(wout - wtest)) < 1e-8:
        print("Test validated")
    else:
        print("There was an error in implementation")
else:
    print("Error in dimensions")
```

```
There was an error in implementation
```

In [8]:

```
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 : desired sequence at time n
        u : input of length p
        w : wiener filter to update
        mu : adaptation step

    Returns:
    =======
        w : upated filter
        err : d-dest
        dest : prediction = :math:`u(n)^T w`
    """
    dest = u.dot(w)
    err = d - dest
    w = w + mu * u * err
    return (w, err, dest)
```

2.2.1 Procédure d'identification

- Commencez par quelques commandes directes (initialisations et une boucle `for` sur la variable temps) pour identifier le filtr que cela fonctionnera, vous collecterez les commandes sous la forme d'une fonction `ident`
- Si nécessaire, la fonction `squeeze()` permet de supprimer des entrées unidimensionnelles de d'un tableau n-D (*par exemp* transforme un tableau (3,1,1) en un vecteur de dimension 3)

Afin d'évaluer le comportement de l'algorithme, on trace l'erreur d'estimation, l'évolution des coefficients du filtre identifié au cour de l'algorithme, et enfin l'erreur quadratique entre le filtre exact et le filtre identifié. Cela doit être fait pour plusieurs ordres p (l'orc inconnu ...) et pour différentes valeurs de l'étape d'adaptation μ.

- L'erreur quadratique peut être évaluée simplement grâce à une *liste de compréhension* selon

```
Errh = [somme (he-w [:, n]) ** 2 pour n dans la plage (N + 1)]
```

Étudiez le code ci-dessous et implantez les lignes manquantes.

Contents

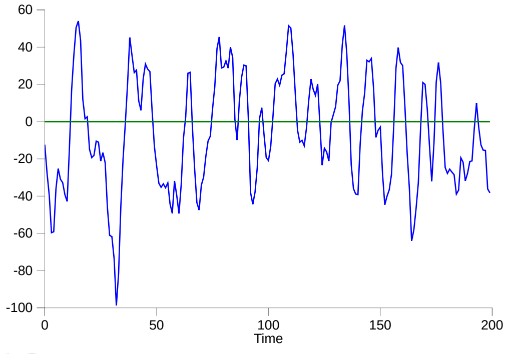
In [9]:

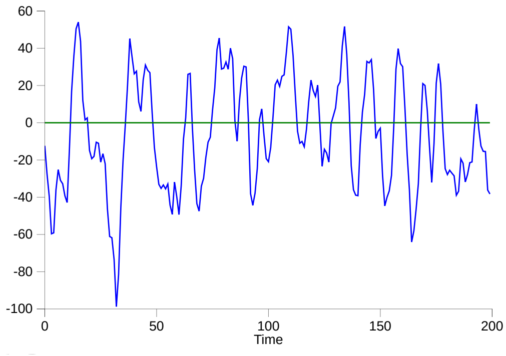
```
mu = 0.1  # an initial value for mu
L = 6  # size of identified filter (true size is p)
NN = 200  #number of iterations
err = np.zeros(NN)
w = zeros((L, NN + 1))
yest = np.zeros(NN)

# The key lines are here: you have to iterate over time and compute
# the output of the LMS at each iteration.You may save all outputs in the matrix
# w initialized above -- column k contains the solution at time k. You must
# also save the succession of errors, and the estimated output
#
# You have two lines to implement here.
# DO IT YOURSELF!
#
# After these lines, (w[:,t+1],err[t],yest[t]) are defined

# This is used to define the "true" impulse response  vector with the same size as w:
# a shorter (truncated) one if L<p, and a larger one (zero-padded) if L>p.
newhtest = np.zeros(L)
if np.size(htest) < L:
    newhtest = htest[:L]
else:
    newhtest[:np.size(htest)] = htest

# Results:
plt.figure(1)
tt = np.arange(NN)
plt.plot(tt, y0[:NN], label='Initial Noiseless Output')
plt.plot(tt, yest[:NN], label="Estimated Output")
plt.xlabel('Time')
figcaption(
    "Comparison of true output and estimated one after identification",
    label="fig:ident_lms_compareoutputs")

plt.figure(2)
errh = [sum((newhtest - w[:, t])**2) for t in range(NN)]
plt.plot(tt, errh, label='Quadratic error on h')
plt.legend()
plt.xlabel('Time')
figcaption(
    "Quadratic error between true and estimated filter",
    label="fig:ident_lms_eqonh")
```

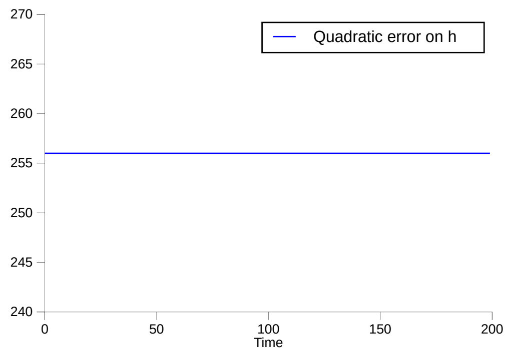
Contents

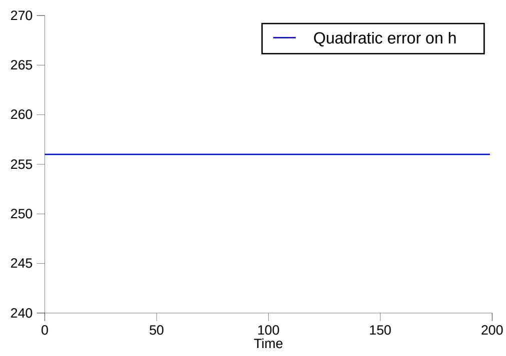
Caption: Comparison of true output and estimated one after identification

Caption: Quadratic error between true and estimated filter









The solution is given below:

Contents

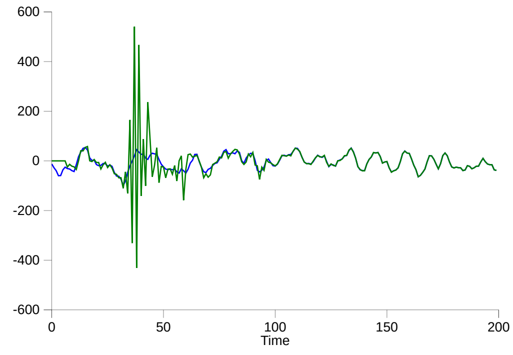
In [10]:

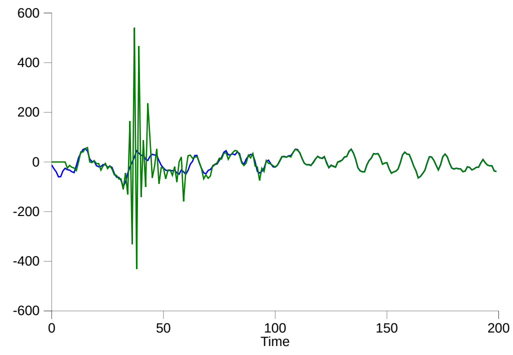
```
mu = 0.1  # an initial value for mu
L = 6  # size of identified filter (true size is p)
NN = 200  #number of iterations
err = np.zeros(NN)
w = zeros((L, NN + 1))
yest = np.zeros(NN)

# The key lines are here: you have to iterate over time and compute
# the output of the LMS at each iteration.You may save all outputs in the matrix
# w initialized above -- column k contains the solution at time k. You must
# also save the succession of errors, and the estimated output

for t in np.arange(L, NN):
    (w[:, t + 1], err[t], yest[t]) = lms(y[t], x[t:t - L:-1], w[:, t], mu)

# This is used to define the "true" impulse response  vector with the same size as w:
# a shorter (truncated) one if L<p, and a larger one (zero-padded) if L>p.
newhtest = np.zeros(L)
LL = np.min([np.size(htest), L])
newhtest[:LL] = htest[:LL]

# Results:
plt.figure(1)
tt = np.arange(NN)
plt.plot(tt, y0[:NN], label='Initial Noiseless Output')
plt.plot(tt, yest[:NN], label="Estimated Output")
plt.xlabel('Time')
figcaption(
    "Comparison of true output and estimated one after identification",
    label="fig:ident_lms_compareoutputs")

plt.figure(2)
errh = [sum((newhtest - w[:, t])**2) for t in range(NN)]
plt.plot(tt, errh, label='Quadratic error on h')
plt.legend()
plt.xlabel('Time')
figcaption(
    "Quadratic error between true and estimated filter",
    label="fig:ident_lms_eqonh")
```

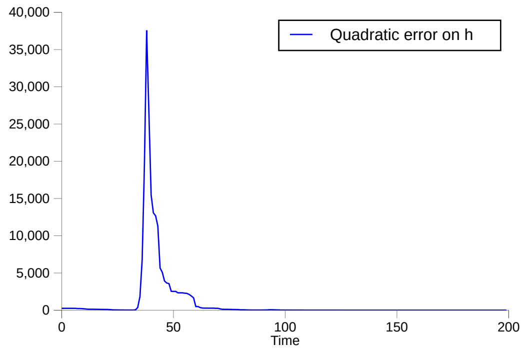
Contents

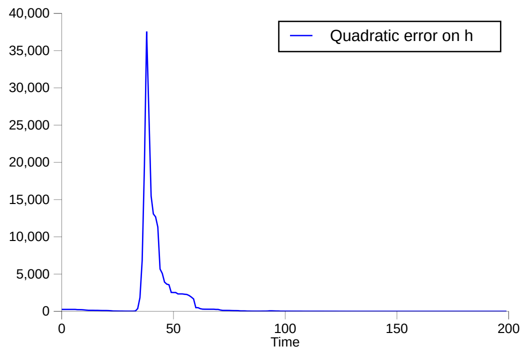
Caption: Comparison of true output and estimated one after identification

Caption: Quadratic error between true and estimated filter









Contents

Nous pouvons maintenant implémenter l'identification en tant que fonction par elle-même, soit finalement une fonction qui réalise initialisations et utilise une boucle sur le LMS. Implémentez cette fonction selon la syntaxe suivante.

In [11]:

```
def ident(observation, input_data, mu, p=20, h_initial=zeros(20)):
    """ Identification of an impulse response from an observation
    `observation` of its output, and from its input `input_data`
    `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
    normalized: boolean (default False)
        compute the normalized LMS instead of the standard one

    Outputs:
    ========
    w: array
        identified impulse response
    err: array
        estimation error
    yest: array
        estimated output
    """
    N = np.size(input_data)
    err = np.zeros(N)
    w = np.zeros((p, N + 1))
    yest = np.zeros(N)

    #
    # DO IT YOURSELF!
    #

    return (w, err, yest)
```

Contents

In [12]:

```
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 = np.size(input_data)
    input_data = squeeze(input_data)  #reshape(input_data,(N))
    observation = squeeze(observation)
    err = np.zeros(N)
    w = np.zeros((p, N + 1))
    yest = np.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], err[t], yest[t]) = lms(
            observation[t], input_data[t:t - p:-1], w[:, t], mun)

    return (w, err, yest)
```

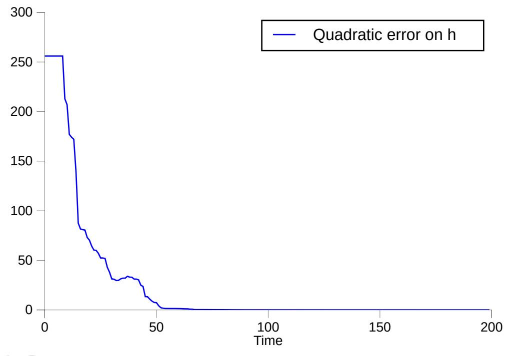
Votre implantation pourra être testée avec

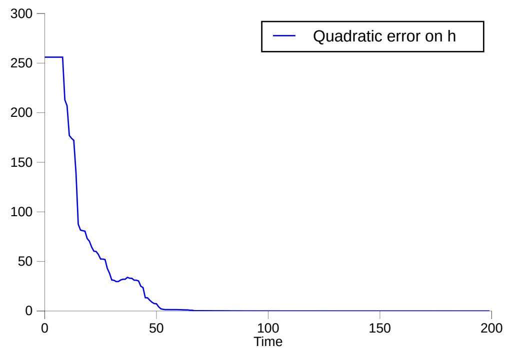
Contents

In [13]:

```
L = 8
(w, err, yest) = ident(y, x, mu=0.05, p=L, h_initial=zeros(L))

newhtest = np.zeros(L)
LL = np.min([np.size(htest), L])
newhtest[:LL] = htest[:LL]

NN = np.min([np.size(yest), 200])
errh = [sum((newhtest - w[:, t])**2) for t in range(NN)]
plt.plot(tt, errh, label='Quadratic error on h')
plt.legend()
_ = plt.xlabel('Time')
print("Identified filter: ", w[:, -1])
```

```
Identified filter:  [ 9.92784885  7.28007219  6.65473706  7.3350914   2.74130909  0.22159139
 -0.08345302  0.02815564]
```





Contents

In [14]:

```
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`
    `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
    normalized: boolean (default False)
        compute the normalized LMS instead of the standard one

    Outputs:
    ========
    w: array
        identified impulse response
    err: array
        estimation error
    yest: array
        estimated output
    """
    N = np.size(input_data)
    err = np.zeros(N)
    w = np.zeros((p, N + 1))
    yest = np.zeros(N)

    w[:, p] = h_initial
    for t in np.arange(p, N):
        if normalized:
            assert mu < 2, "In the normalized case, mu must be less than 2"
            mun = mu / (
                np.dot(input_data[t:t - p:-1], input_data[t:t - p:-1]) + 1e-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)
```

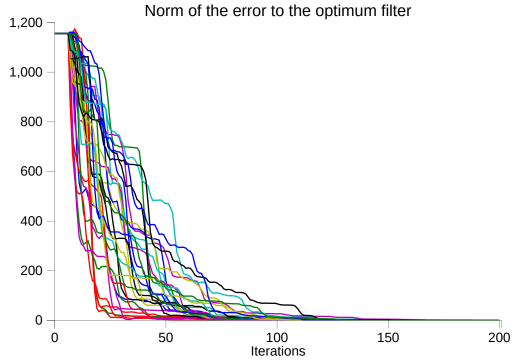
2.2.2 Stabilité des résultats

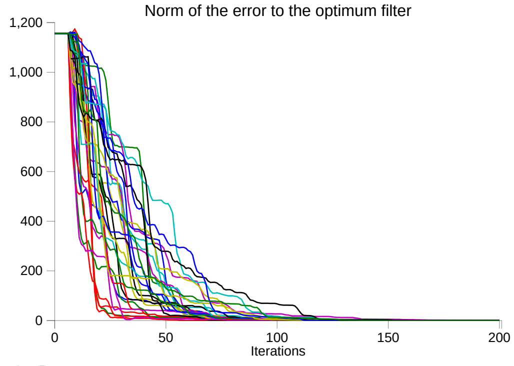
Il est très instructif d'examiner la reproductibilité des résultats lorsque les données changent. Laissez μ fixé et générez de nouve
Appliquez ensuite la procédure d'identification et tracez la courbe d'apprentissage.

Contents

In [15]:

```
p = 6  #<-- actual length of the filter
for ndata in range(30):
    ## Generate new datas
    N = 200
    x = lfilter([1, 1], [1], np.random.randn(N))
    htest = 10 * np.array([1, 0.7, 0.7, 0.7, 0.3, 0])
    y0 = lfilter(htest, [1], x)
    y = y0 + 0.1 * randn(N)
    iterations = np.arange(NN + 1)
    # --------------------

    for mu in [0.01]:
        (w, erreur, yest) = ident(y, x, mu, p=p, h_initial=zeros(p))
        Errh = [sum(htest - w[:, n])**2 for n in range(NN + 1)]
        plt.plot(iterations, Errh, label="$\mu={}$".format(mu))
        plt.xlim([0, NN + 1])

plt.title("Norm of the error to the optimum filter")
_ = plt.xlabel("Iterations")
```





Les données sont aléatoires; L'algorithme est stochastique et la courbe d'apprentissage l'est alors aussi ! Heureusement, on vér algorithmes convergent ... puisque l'erreur tend vers zéro. Donc, ça marche...

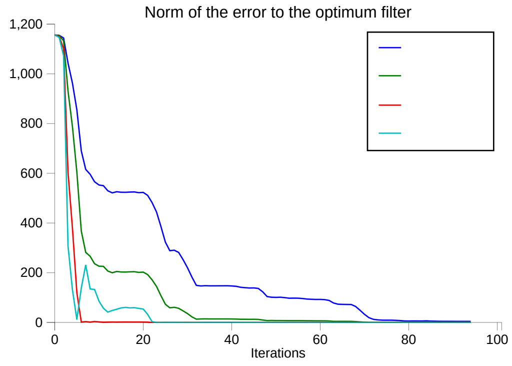
2.2.3 Etude en fonction de μ

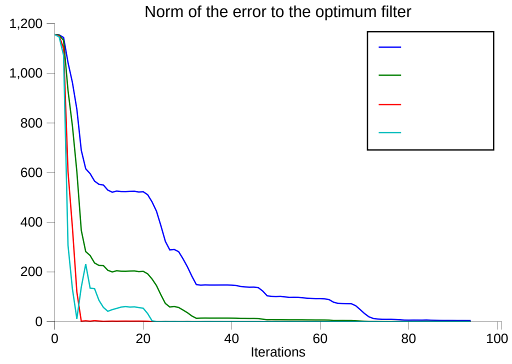
Il est assez aisé d'étudier le comportement de l'algorithme en fonction du pas μ. Il suffit de faire une boucle sur les valeurs possi d'appeler la procédure d'identification et d'afficher les résultats.

Contents

In [16]:

```
# Study with respect to $\mu$
p = 6
NN = 100
iter = np.arange(NN + 1) - p

## Generate new datas
N = 200
x = lfilter([1, 1], [1], np.random.randn(N))
htest = 10 * np.array([1, 0.7, 0.7, 0.7, 0.3, 0])
y0 = lfilter(htest, [1], x)
y = y0 + 0.1 * randn(N)
# --------------------

for mu in [0.01, 0.02, 0.05, 0.081]:
    (w, erreur, yest) = ident(y, x, mu, p=p, h_initial=zeros(p))
    Errh = [sum(htest - w[:, n])**2 for n in range(NN + 1)]
    plt.plot(iter, Errh, label="$\mu={}$".format(mu))
    plt.xlim([0, NN + 1])

plt.legend()
plt.title("Norm of the error to the optimum filter")
_ = plt.xlabel("Iterations")
```





Contents

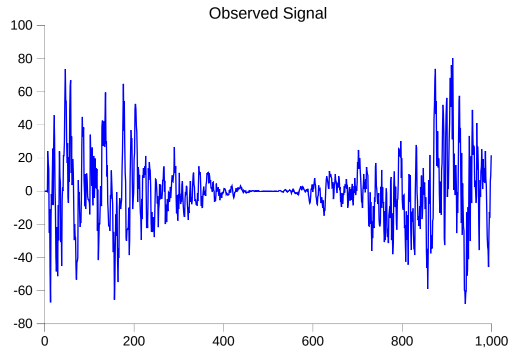
2.2.4 Capacités de poursuite

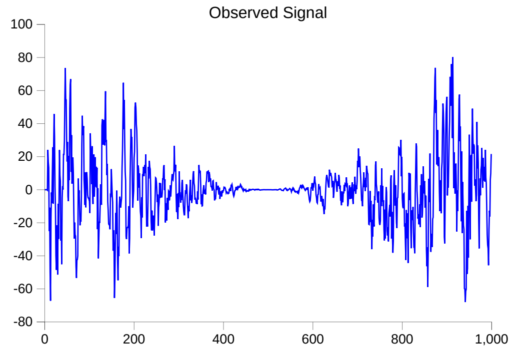
Avec un pas constant, le LMS ne converge **jamais**, car tant qu'une erreur existe, le filtre est toujours mis à jour. Une conséquen que le LMS maintient des capacités de poursuite, d'adaptation, qui sont particulièrement utiles dans un contexte non stationnaire problème d'identification, il est possible que le filtre à identifier varie dans le temps. Dans ce cas, l'algorithme doit pouvoir suivre modifications. Un tel exemple est simulé ci-dessous, où la réponse impulsionnelle est modulée par un $\cos()$, selon

$$h(t,\tau) = (1+\cos(2\pi f_0 t))\, h_{\text{test}}(\tau).$$

In [17]:

```
### Slow non-stationarity

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





Alors, on peut tester la procédure d'identification avec ce signal non stationnaire. On vérifie que l'erreur est nulle et que le filtre ic effectivement modulé avec un cosinus...

Contents

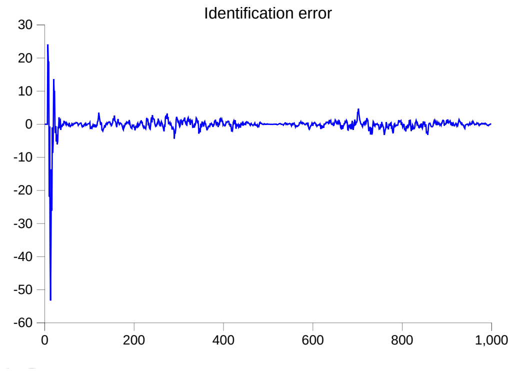
In [18]:

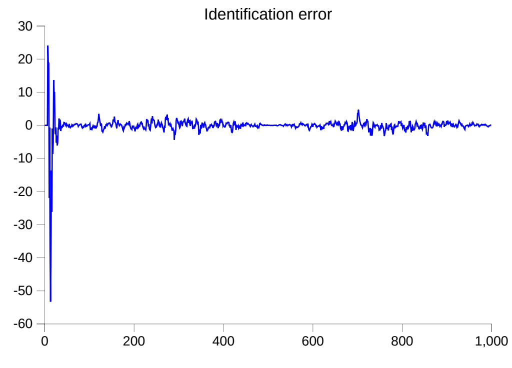
```
p = 7
(w, err, yest) = ident(y, u, mu=0.1, p=p, h_initial=zeros(p))
#(w,err,yest)=ident(y,u,mu=1,p=p,h_initial=zeros(p),normalized=True)
plt.figure(1)
clf()
plt.plot(err)
plt.title('Identification error')
figcaption(
    "Identification error in the nonstationary case",
    label="fig:error_ns_case")
plt.figure(2)
plt.clf()
t = np.arange(0, N + 1)
true_ns_h = np.outer((1 + cos(2 * pi * t / N)), htest)
plt.plot(t, w.T, lw=1)
plt.plot(t, true_ns_h, lw=2, label="True values", alpha=0.4)
plt.title("Evolution of filter's coefficients")
figcaption("Evolution of filter's coefficients", label="fig:coeff_ns_case")
```

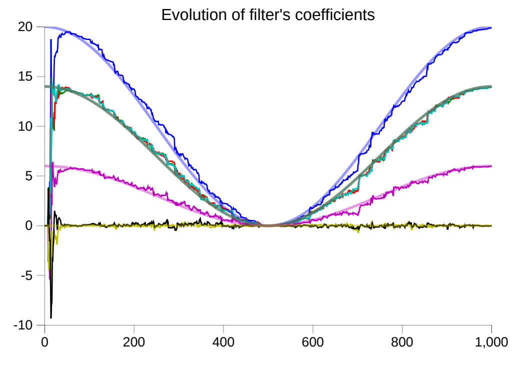
Contents

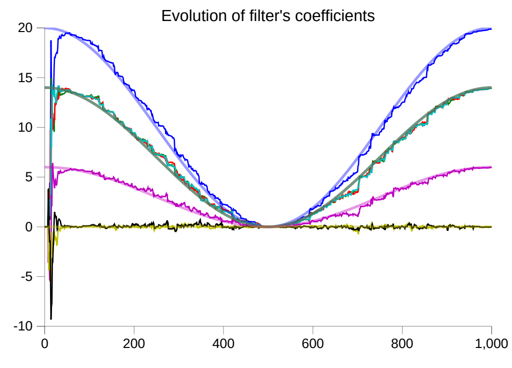
Caption: Identification error in the nonstationary case

Caption: Evolution of filter's coefficients









Contents

2.3 Propriétés de convergence du LMS

Comme nous le réalisons avec ces expériences numériques, puisque le LMS dépend directement des données, l'algorithme lui-i stochastique; Les courbes d'apprentissage ont un caractère aléatoire mais les trajectoires moyennes convergent toujours vers la correcte. La caractérisation correcte des algorithmes stochastiques est difficile - en fait, la première analyse correcte est due à E Macchi (1983). L'analyse traditionnelle repose sur une hypothèse fausse, l'*hypothèse d'indépendance*, qui donne cependant une ce qui se passe.

L'idée est simplement que l'algorithme moyen

$$\begin{aligned}\mathbb{E}\left[\mathbf{w}(n+1)\right] &= \mathbb{E}\left[\mathbf{w}(n) - \mu\mathbf{u}(n)\left(\mathbf{u}(n)^T\mathbf{w}(n) - d(n)\right)\right] \\ &= \mathbb{E}\left[\mathbf{w}(n)\right] - \mu\mathbb{E}\left[\mathbf{u}(n)\left(\mathbf{u}(n)^T\mathbf{w}(n) - d(n)\right)\right] \\ &\approx \mathbb{E}\left[\mathbf{w}(n)\right] - \mu\left(\mathbb{E}\left[\mathbf{u}(n)\mathbf{u}(n)^T\right]\mathbb{E}\left[\mathbf{w}(n)\right] - \mathbb{E}\left[\mathbf{u}(n)d(n)\right]\right)\end{aligned}$$

est exactement l'algorithme de gradient (le vrai). Ainsi, on retrouvera exactement les mêmes conditions de convergence que pou de gradient. Cependant, ce n'est qu'une approximation, et une approximation grossière ! En effet, dans la troisième ligne l'égalite $\mathbb{E}\left[\mathbf{u}(n)\mathbf{u}(n)^T\mathbf{w}(n)\right] = \mathbb{E}\left[\mathbf{u}(n)\mathbf{u}(n)^T\right]\mathbb{E}\left[\mathbf{w}(n)\right]$ est incorrecte car bien évidemment $\mathbf{w}(n)$ dépend de $\mathbf{u}(n)$ puisqu'il dép composantes aux instants $n-1, n-2,$ etc.

En outre, il faut souligner que les courbes d'apprentissage sont maintenant **aléatoires**. Ainsi, nous pouvons comprendre que les convergence sont plus strictes que pour l'algorithme de gradient. Une règle pratique pour le choix de μ est

$$\mu = \frac{2}{\alpha \mathrm{Tr}\left[\mathbf{R}_{uu}\right]} = \frac{2}{\alpha p R_{uu}(0)},$$

où α est un scalaire compris entre 2 et 3, $R_{uu}(0) = \mathbb{E}\left[|u(n)|^2\right]$ et p est la dimension de la matrice de corrélation.

... à suivre...

Eweda, E., and Macchi, O.. "Quadratic mean and almost-sure convergence of unbounded stochastic approximation algorithms wit observations." Annales de l'institut Henri Poincaré (B) Probabilités et Statistiques 19.3 (1983): 235-255. . @article{Eweda1983, au E., Macchi, O.}, journal = {Annales de l'institut Henri Poincaré (B) Probabilités et Statistiques}, keywords = {almost-sure convergen observations; quadratic mean convergence; stochastic gradient algorithm; finite memory; finite moments}, language = {eng}, numb {235-255}, publisher = {Gauthier-Villars}, title = {Quadratic mean and almost-sure convergence of unbounded stochastic approxima with correlated observations}, url = {http://eudml.org/doc/77211}, volume = {19}, year = {1983}, }

2.4 Le LMS normalisé

Une variante simple du LMS repose sur l'idée d'introduire un pas non constant μ_n et de déterminer une valeur optimale pour ce itération. Une manière simple d'obtenir le résultat est la suivante :

- L'erreur standard, avant de mettre à jour le LMS de $\mathbf{w}(n)$ en $\mathbf{w}(n+1)$, est
$$e(n|n) = \mathbf{w}(n)^T\mathbf{u}(n) - d(n)$$
- Après avoir mis à jour le filtre, on peut recalculer l'erreur, comme
$$e(n|n+1) = \mathbf{w}(n+1)^T\mathbf{u}(n) - d(n).$$
Cette erreur est appelée erreur *a posteriori*, car elle est calculée avec le filtre mis à jour. Ceci est également indiqué par la n . $|n+1$ qui signifie "calculée à l'aide du filtre au temps $n+1$". L'erreur standard est donc qualifiée d'erreur *a priori*.

À partir de $\mathbf{w}(n+1) = \mathbf{w}(n) - \mu\mathbf{u}(n)e(n|n)$, nous obtenons immédiatement que

$$\begin{aligned}e(n|n+1) &= \mathbf{w}(n+1)^T\mathbf{u}(n) - d(n) \\ &= \left(\mathbf{w}(n) - \mu_n\mathbf{u}(n)e(n|n)\right)^T\mathbf{u}(n) - d(n) \\ &= e(n|n) - \mu_n\mathbf{u}(n)^T\mathbf{u}(n)e(n|n) \\ &= \left(1 - \mu_n\mathbf{u}(n)^T\mathbf{u}(n)\right)e(n|n)\end{aligned}$$

Bien évidemment, la mise à jour doit diminuer l'erreur. Ainsi, on doit avoir

$$|e(n|n+1)| \leq |e(n|n)|,$$

soit

$$\left|\left(1 - \mu_n\mathbf{u}(n)^T\mathbf{u}(n)\right)\right| \leq 1.$$

Cela entraîne la condition

$$\boxed{0 \leq \mu_n \leq \frac{2}{\mathbf{u}(n)^T\mathbf{u}(n)}}.$$

La valeur optimale du pas correspond au minimum de $|e(n|n+1)|$, qui est simplement donné par

$$\boxed{\mu_n = \frac{1}{\mathbf{u}(n)^T\mathbf{u}(n)}}.$$

Cependant, l'**algorithme LMS normalisé** est souvent donné avec un facteur supplémantaire, par exemple $\tilde{\mu}$, qui ajoute un para réglage à l'algorithme selon

$$\boxed{\mathbf{w}(n+1) = \mathbf{w}(n) - \frac{\tilde{\mu}}{\mathbf{u}(n)^T\mathbf{u}(n)}\mathbf{u}(n)\left(\mathbf{w}(n)^T\mathbf{u}(n) - d(n)\right)}$$

La condition (26) donne directement

$$\boxed{0 \leq \tilde{\mu} \leq 2},$$

qui est une règle vraiment très simple pour ajuster le pas.

Contents

L'implantation du LMS normalisé requiert une simple modification du LMS standard. Notons cependant qu'il est utile d'introduire constante positive dans la définition du pas d'adaptation :

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

afin d'éviter une division par zéro erreur.

In [19]:

```
def normalized_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 : desired sequence at time n
        u : input of length p
        w : wiener filter to update
        mu : adaptation step for the NLMS; mu <2

    Returns:
    =======
        w : upated filter
        err : d-dest
        dest : prediction = :math:`u(n)^T w`
    """
    assert mu < 2, "In the normalized case, mu must be less than 2"
    u = squeeze(
        u)  #Remove single-dimensional entries from the shape of an array.
    w = squeeze(w)
    dest = u.dot(w)
    err = d - dest
    mun = mu / (dot(u, u) + 1e-10)
    w = w + mun * u * err
    return (w, err, dest)
```

Contents

2.5 Autres variantes du LMS

L'algorithme de gradient stochastique est obtenu à partir de l'algorithme de gradient théorique en approximant les quantités stati par leurs valeurs instantanées. Cette approche peut être étendue à des fonctions de coût arbitraires. En effet, si l'on considère u coût $J(\mathbf{w}) = \mathbb{E}\left[f(e(n))\right]$, avec f une fonction paire et positive, alors l'algorithme du gradient conduit à

$$\begin{aligned}\mathbf{w}(n+1) &= \mathbf{w}(n) - \mu\frac{\mathrm{d}\mathbb{E}\left[f(e(n))\right]}{\mathrm{d}\mathbf{w}(n)} \\ &= \mathbf{w}(n) - \mu\mathbb{E}\left[\mathbf{u}(n)\frac{\mathrm{d}f(e(n))}{\mathrm{d}\mathbf{w}(n)}\right],\end{aligned}$$

où on a utilisé la règle de dérivation pour les fonctions composées.

L'algorithme de gradient stochastique correspondant est alors immédiatement donné par

$$\mathbf{w}(n+1) = \mathbf{w}(n) - \mu\mathbf{u}(n)\frac{\mathrm{d}f(e(n))}{\mathrm{d}e(n)}.$$

Voyons quelques exemples:

- si $f(e) = |e|$, puis $f'(e) = \mathrm{sign}(e)$ et nous obtenons l'**algorithme du signe**:
  $$\mathbf{w}(n+1) = \mathbf{w}(n) - \mu\mathbf{u}(n)\mathrm{sign}(e(n)).$$
  Il s'agit d'un algorithme de très faible complexité, qui peut être implémenté sans multiplication (si μ est une puissance de 2, multiplication par la pas peut être implémentée comme un changement de bit).
- pour $f(e) = |e|^k$, puis $f'(e) = k|e|^{k-1}\mathrm{sign}(e)$, et l'algorithme de gradient stochastique a la forme
  $$\mathbf{w}(n+1) = \mathbf{w}(n) - \mu\mathbf{u}(n)|e(n)|^{k-1}\mathrm{sign}(e(n)).$$

Voir Mathews, ece6550-chapitre 4 (http://www.ece.utah.edu/~mathews/ece6550/chapter4.pdf), page 22, pour un exemple de for linéaire par morceaux, qui mène à une quantification de l'erreur .

2.6 Les moindres carrés récursifs

Au lieu de prendre une estimation instantanée de la matrice de corrélation et du vecteur de corrélation, il est possible d'utiliser le exponentielles \begin{equation}

$$\begin{cases} \hat{\mathbf{R}}_{uu}(n+1) = \sum_{l=0}^{n+1} \lambda^{l-n-1}\mathbf{u}(l)\mathbf{u}(l)^H = \lambda\hat{\mathbf{R}}_{uu}(n) + \mathbf{u}(n+1)\mathbf{u}(n+1)^H \\ \hat{\mathbf{R}}_{du}(n+1) = \lambda\hat{\mathbf{R}}_{du}(n) + d(n+1)\mathbf{u}(n+1). \end{cases}$$

\end{equation} Il reste alors à calculer la solution

$$\hat{\mathbf{w}}(n+1) = \left[\hat{\mathbf{R}}_{uu}(n+1)\right]^{-1}\hat{\mathbf{R}}_{du}(n+1).$$

Le problème principal est l'inversion, pour chaque n, de la matrice de corrélation. Heureusement, il est possible d'obtenir une so qui n'a pas besoin d'une inversion matricielle du tout ... La clé ici est d'invoquer le lemme d'inversion matricielle (http://en.wikipedia.org/wiki/Woodbury_matrix_identity)

$$[\mathbf{A} + \mathbf{BD}]^{-1} = \mathbf{A}^{-1} - \mathbf{A}^{-1}\mathbf{B}[\mathbf{I} + \mathbf{DA}^{-1}\mathbf{B}]^{-1}\mathbf{DA}^{-1}.$$

En appliquant ceci à $\mathbf{A} = \lambda\hat{\mathbf{R}}_{uu}(n-1)$, $\mathbf{B} = \mathbf{u}(n)$ et $\mathbf{C} = \mathbf{u}(n)^H$, et en dénotant

$$\mathbf{K}_{n+1} = \left[\hat{\mathbf{R}}_{uu}(n+1)\right]^{-1},$$

on obtient facilement

$$\mathbf{K}(n+1) = \frac{1}{\lambda}\mathbf{K}(n) - \frac{1}{\lambda^2}\frac{\mathbf{K}(n)\mathbf{u}(n+1)\mathbf{u}(n+1)^H\mathbf{K}(n)}{1 + \frac{1}{\lambda}\mathbf{u}(k+1)^H\mathbf{K}(n)\mathbf{u}(k+1)},$$

Après plusieurs lignes de calculs, on arrive à la formule de mise à jour

$$\hat{\mathbf{w}}(n+1) = \hat{\mathbf{w}}(n) + \mathbf{K}(n+1)\mathbf{u}(n+1)[d(n+1) - \mathbf{w}(n)^H\mathbf{u}(n+1)].$$

Notez qu'il existe quelques différences de notation entre le LMS et le RLS. Pour le LMS, le filtre $\mathbf{w}(n+1)$ est calculé sur la bas disponibles au moment n. Pour le RLS, $\mathbf{w}(n+1)$ est calculé en utilisant les données disponibles au moment $(n+1)$. C'est ju différence de notation - nous pourrions facilement renommer $\mathbf{w}(n+1)$ en $\mathbf{v}(n)$ et obtenir des index similaires. Cependant, ce traditionnelles, et nous les indiquons donc ici. Il est cependant important de noter que les deux filtres sont calculés à l'aide de l'er c'est-à-dire l'erreur utilisant les données à l'instant n, et le filtre calculé à l'aide des données à l'instant $n-1$.

Initialisation -

L'initialisation de l'algorithme nécessite la spécification d'un $\mathbf{w}(0)$ initial, qui est habituellement pris comme un vecteur nul. Il fau spécifier $\mathbf{K}(0)$. Puisque $\mathbf{K}(0)$ est l'inverse de la matrice de corrélation *avant* le début des itérations, on choisit en général $\mathbf{R}_{uu}$ avec δ très petit. Ainsi l'inverse est $\mathbf{K}(0) = \delta^{-1}\mathbf{I}$, une grande valeur qui disparaît pendant les itérations de l'algorithme.

Une implémentation de l'algorithme RLS est proposée ci-dessous, en utilisant d'abord le type standard `numpy array` puis le ty conversion d'un type à l'autre est effectuée par les mots-clés `np.matrix` ou `np.array` (qui font une copie) ou en utilisant `np.a` `np.asarray`.

Contents

In [20]:

```
# 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)
```

À ce stade, il est utile de mener à nouveau les expérimentations précédentes -- identification avec des données non stationnaire l'algorithme RLS. Ensuite, comparer et conclure.

Contents

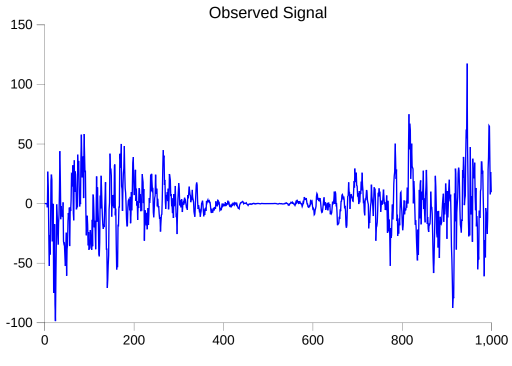
In [21]:

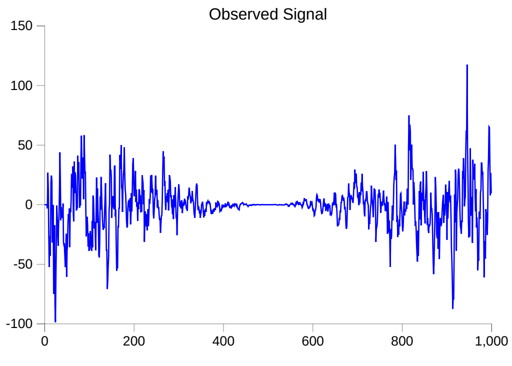
```
def ident_rls(observation, input_data, factor_lambda=0.95, p=20):
    """ Identification of an impulse response from an observation
    `observation` of its output, and from its input `input_data` \n
    `mu` is the adaptation step\n
    Inputs:
    =======
    observation: array
        output of 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 = np.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)
```

Contents

In [22]:

```
### Slow non-stationarity

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





Contents

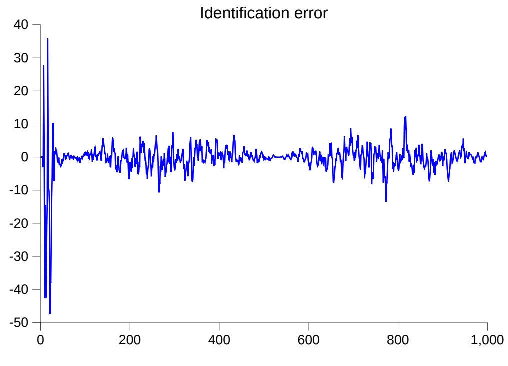
In [23]:

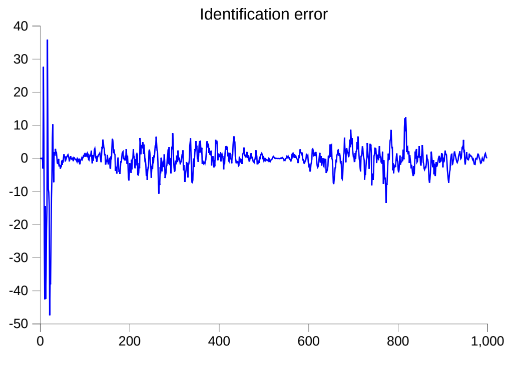
```
p = 7
lamb = 0.97
(w, err) = ident_rls(y, u, factor_lambda=lamb, p=10)
plt.figure(1)
clf()
plt.plot(err)
plt.title('Identification error')
figcaption(
    "Identification error in the nonstationary case",
    label="fig:error_ns_case")
plt.figure(2)
plt.clf()
t = np.arange(0, N + 1)
true_ns_h = np.outer((1 + cos(2 * pi * t / N)), htest)
plt.plot(t, w.T, lw=1)
plt.plot(t, true_ns_h, lw=2, label="True values", alpha=0.4)
plt.title("Evolution of filter's coefficients")
figcaption("Evolution of filter's coefficients", label="fig:coeff_ns_case")
```

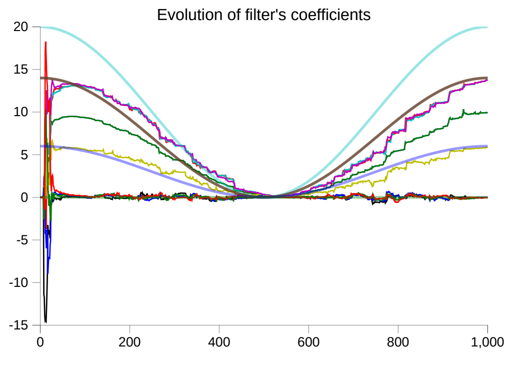
Contents

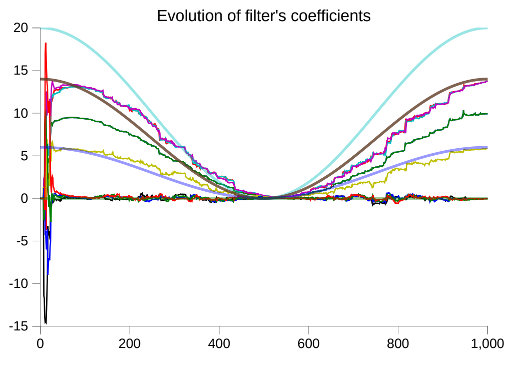
Caption: Identification error in the nonstationary case

Caption: Evolution of filter's coefficients









Contents

Quelques références:

- http://www.ece.utah.edu/~mathews/ece6550/chapter10.pdf (http://www.ece.utah.edu/~mathews/ece6550/chapter10.pdf)
- http://www.cs.tut.fi/~tabus/course/ASP/LectureNew10.pdf (http://www.cs.tut.fi/~tabus/course/ASP/LectureNew10.pdf)
- Recursive Least Squares at wikipedia (http://en.wikipedia.org/wiki/Recursive_least_squares_filter)
- Adaptive Filtering Applications (http://www.intechopen.com/books/adaptive-filtering-applications) (livre d'accès ouvert à intec

---