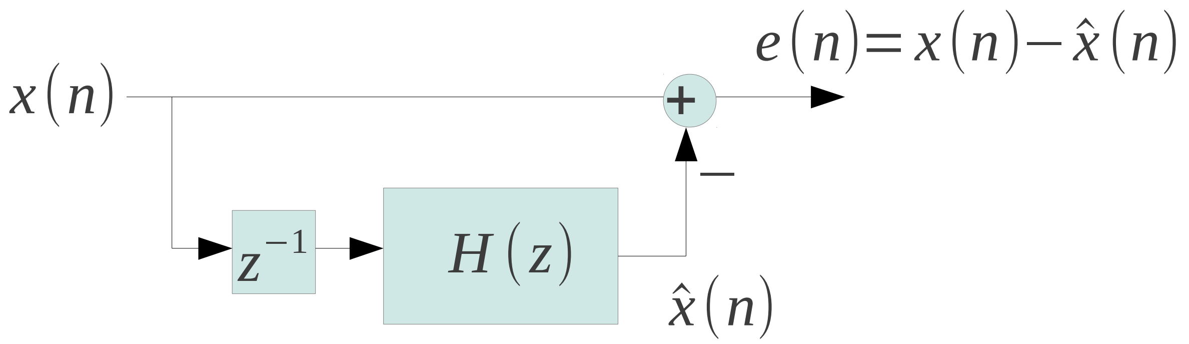
Digital Signal Processing 2/ Advanced Digital Signal Processing
Lecture 14,
Matched Filter, Prediction

Gerald Schuller, TU Ilmenau

Prediction

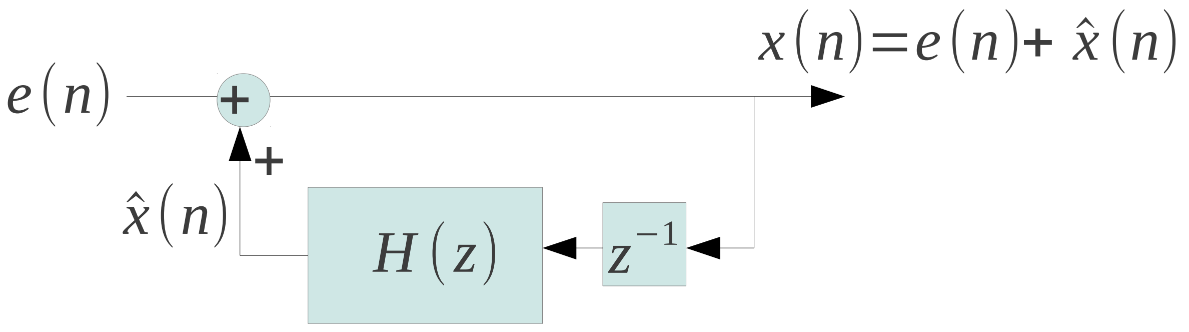
A system which produces the **prediction error** (for instance of part of an encoder) as an output is the following,



Here, $x(n)$ is the signal to be predicted, $H(z)$ is our prediction filter, whose **coefficents** are obtained with our **Wiener approach** in the last lecture slides, $\boldsymbol{h}=(\boldsymbol{R}_{xx})^{-1}\boldsymbol{r}_{xx}$, where $H(z)$ is simply its z-transform. It works on only past samples (that's why we have the delay element by one sample, z^{-1} , before it), $\hat{x}(n)$ is our **predicted** signal, and $e(n)=x(n)-\hat{x}(n)$ is our **prediction error** signal. Hence, our system which produces the prediction error has the z-domain transfer function of

$$H_{perr}(z)=1-z^{-1}\cdot H(z)$$

This can be an **encoder**. Observe that we can **reconstruct** the original signal $x(n)$ in a **decoder** from the prediction error $e(n)$, with the following system,



Remember that encoder computed $e(n)=x(n)-\hat{x}(n)$.
The feedback loop in this system is causal because it only uses **past**, already **reconstructed samples**!
Observe that this decoders transfer function is

$$H_{rec}(z)=\frac{1}{1-z^{-1}\cdot H(z)}=\frac{1}{H_{perr}(z)}$$

which is exactly the inverse of the encoder, which was to be expected.

Python Example

Goal: Construct a prediction filter for our female speech signal of **order L=10**, which minimizes the mean-squared prediction error.

Read in the female speech sound:

```
ipython3 –pylab
from sound import *
x,fs=wavread('fspeech.wav')
shape(x)
#Out: (207612,)
#make x a matrix of float type and transpose
it into a column, normalize to –1<x<1:
x=matrix(x,dtype=float).T/2**15
#listen to it, turning x into a 1-
dimensional array type for the argument:
sound(array(x.T)[0] *2**15,fs)
```

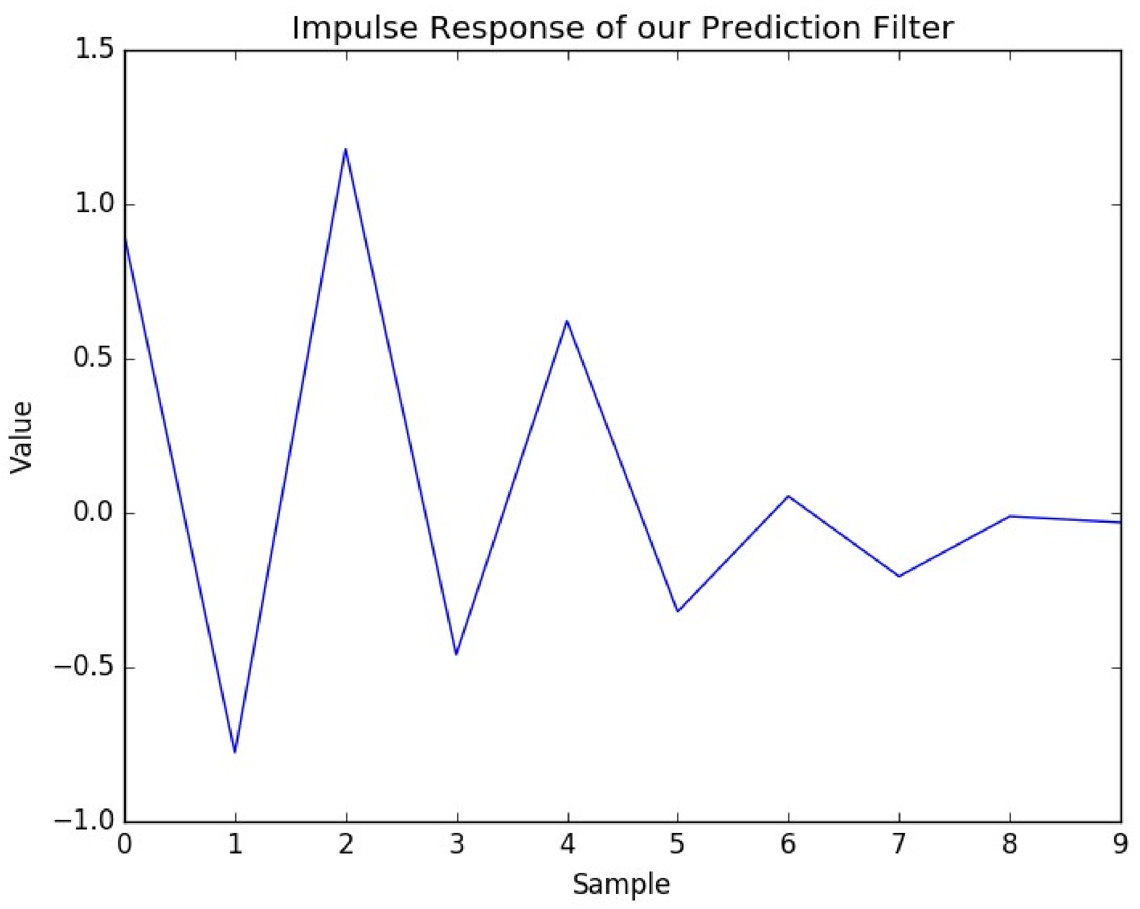
#Construct our Matrix A from x:

```
A=matrix(zeros((100000,10)));
for m in range(0,100000):
    A[m, :]=flipud(x[m+arange(10)]).T


#Construct our desired target signal d,
#one sample into the future, we
#start with the first 10 samples already in the
#prediction filter, then the 11th sample is
#the first to be predicted:
```

```
d = x[arange(10,100010)]

#Compute the prediction filter:
h=inv(A.T*A) * A.T * d;
h
```

```
matrix([[ 0.90078449],
        [-0.7764783 ],
        [ 1.17924513],
        [-0.45849443],
        [ 0.62230755],
        [-0.32026094],
        [ 0.05412175],
        [-0.20557095],
        [-0.01108994],
        [-0.03070101]])
```

```
plot(h)
xlabel('Sample')
ylabel('Value')
title('Impulse Response of our Prediction Filter')
```



Then our prediction filter, with the delay in the encoder becomes (to compare it with the original signal):

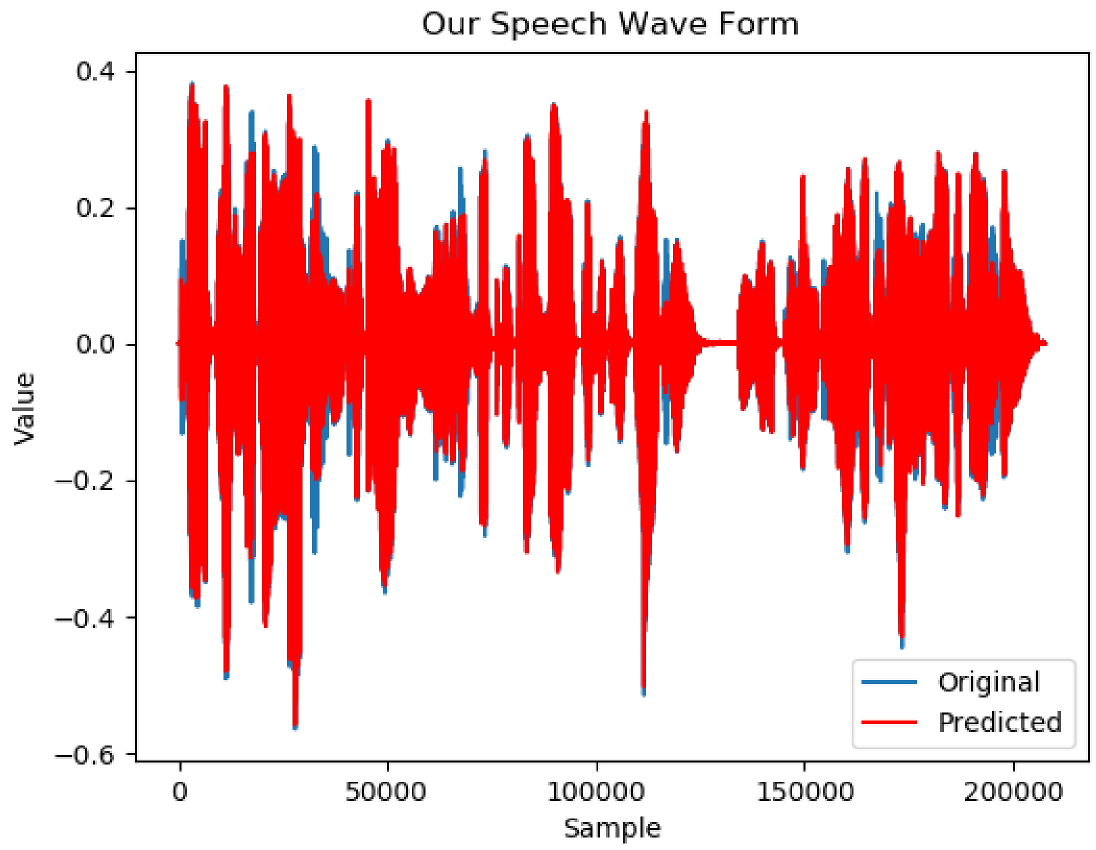
```
hpred = vstack([0, h])
```

The predicted values are now obtained by applying these coefficients as an FIR filter:

```
import scipy.signal as sp
xpred = sp.lfilter(array(hpred.T)[0],1,array(x.T)
[0])
```

Now we can plot the predicted values on top of the actual original signal values, to see how accurate our prediction is:

```
plot(x);
plot(xpred,'red')
legend(('Original','Predicted'))
xlabel('Sample')
ylabel('Value')
title('Our Speech Wave Form')
```



Our corresponding **prediction error** filter (which is in the **encoder**) is $H_{perr}(z)=1-z^{-1}\cdot H(z)$, in Python:

```
hperr = vstack([1, -h])
hperr
matrix([[ 1.        ],
        [-0.90078449],
        [ 0.7764783 ],
        [-1.17924513],
        [ 0.45849443],
        [-0.62230755],
        [ 0.32026094],
        [-0.05412175],
        [ 0.20557095],
        [ 0.01108994],
        [ 0.03070101]])
```

#The prediction error e(n) is obtained using our prediction error filter:

```
e = sp.lfilter(array(hperr.T)[0],1,array(x.T)[0]);
#make a matrix type out of it (row matrix):
e=matrix(e)
```

#error power per sample:

```
e*e.T/max(shape(e))
#Out: matrix([[ 0.00043284]])
```

#Compare that with the mean squared signal power per sample:

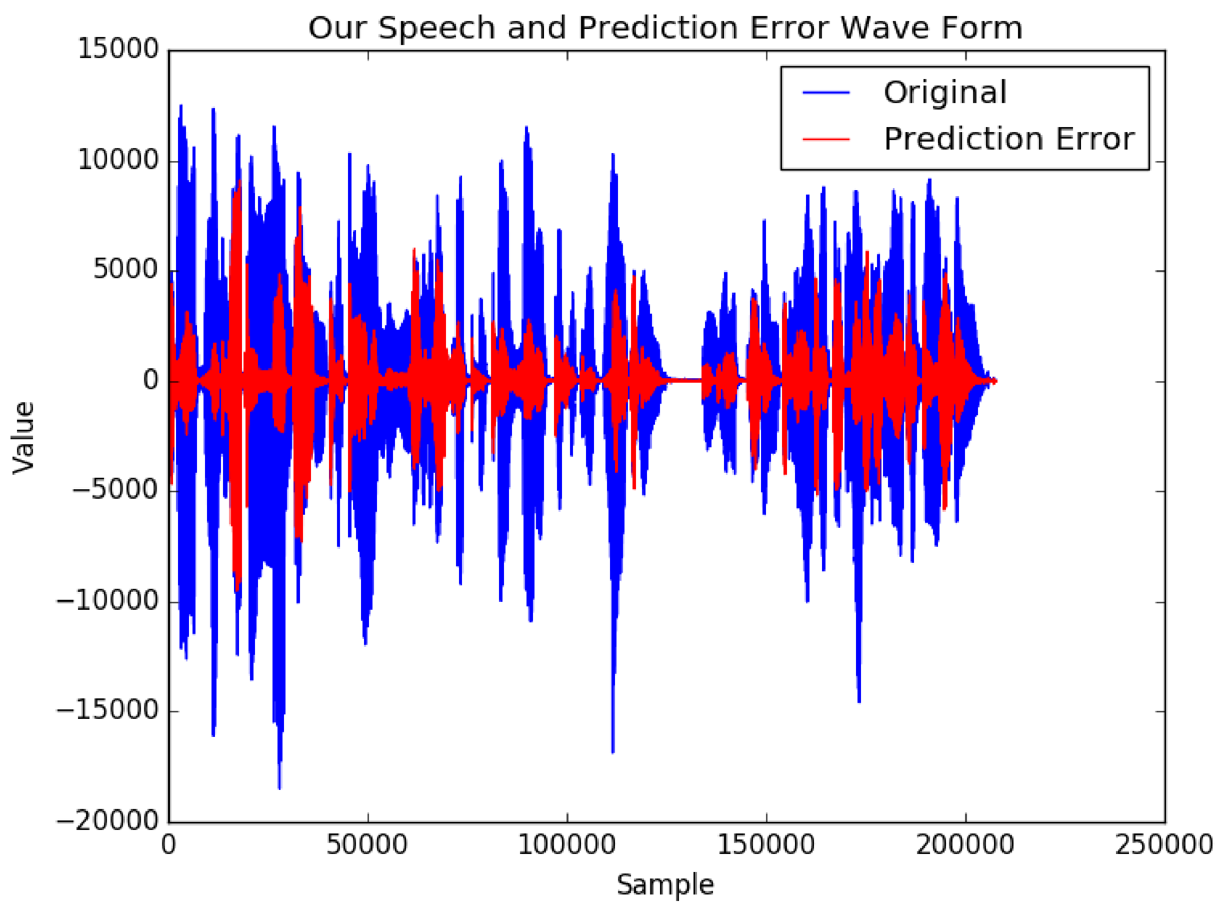
```
x.T*x/max(shape(x))
Out[26]: matrix([[ 0.00697569]])
```

#Which is more than 10 times as big as the prediction error! Which shows that it works!

#Listen to the error signal:

```
sound(2**15*array(e)[0],fs)
```

```
#Take a look at the signal and it's
prediction error:
plot(2**15*x)
plot(2**15*e.T,'r')
xlabel('Sample')
ylabel('Value')
title('Our Speech and Prediction Error Wave
Forms')
legend(('Original', 'Prediction Error'))
```



The **decoder** uses the reverse filter structure $H_{rec} = \frac{1}{1 - z^{-1} \cdot H(z)} = \frac{1}{H_{perr}(z)}$,
hence we use the following filter command to generate the reconstructed signal,

```
xrec = sp.lfilter([1],array(hperr.T)[0],
array(e)[0]);
#plot original for comparison:
plot(x)
#Plot decoded reconstructed on top in red:
plot(xrec,'r')
#We can listen to it with:
sound(2**15*array(xrec),fs)
```

Observe: The decoded, reconstructed signal looks and sound **identical** to the original, as expected. This means we can indeed use it in an encoder-decoder setting.

Neural Network Implementaion

Again we can also use the numerical optimization of Pytorch instead of our closed for solution from Wiener-Hopf, and a `conv1d` layer. We also use the **mean squared error** as minimization criterium or “loss function”. In this case this does not differ from the target of the closed form formulation, and hence we obtain almost the same solution. Here, the desired target signal Y is the audio input signal X, but 1 sample in the future,

```
X=audio[:-L])  #remove last samples (conv makes it longer
again)
Y=audio[1:]) #remove first sample, for the signal to predict, 1
sample in the future
```

We can let it run and see it with

```
python3 pytorch_linpred_inputs.py
```

Online Adaptation

The previous example calculated the prediction coefficients for the entire speech file (or the first 100000 samples). But when we look at the signal waveform, we see that its characteristics and hence its statistics is changing, it is **not stationary.** Hence we can expect a prediction improvement if we divide the speech signal into **small pieces** for the computation of the prediction coefficients, pieces which are small enough to show roughly **constant** statistics. In speech coding, those pieces are usually of length 20 ms, and this approach is called **Linear Predictive Coding (LPC)**. Here, the prediction coefficients are calculated usually every 20 ms, and then transmitted alongside the prediction error, from the encoder to the decoder. This also has the advantage that it needs **no “training set”**, and computes the coefficients from the **actual samples in the current block**.
Observe that this also need a **very fast optimization**, hence the Pytorch approach

with the “Adam” optimizer would not be suitable. We use our faster closed form solution instead.

Python Example

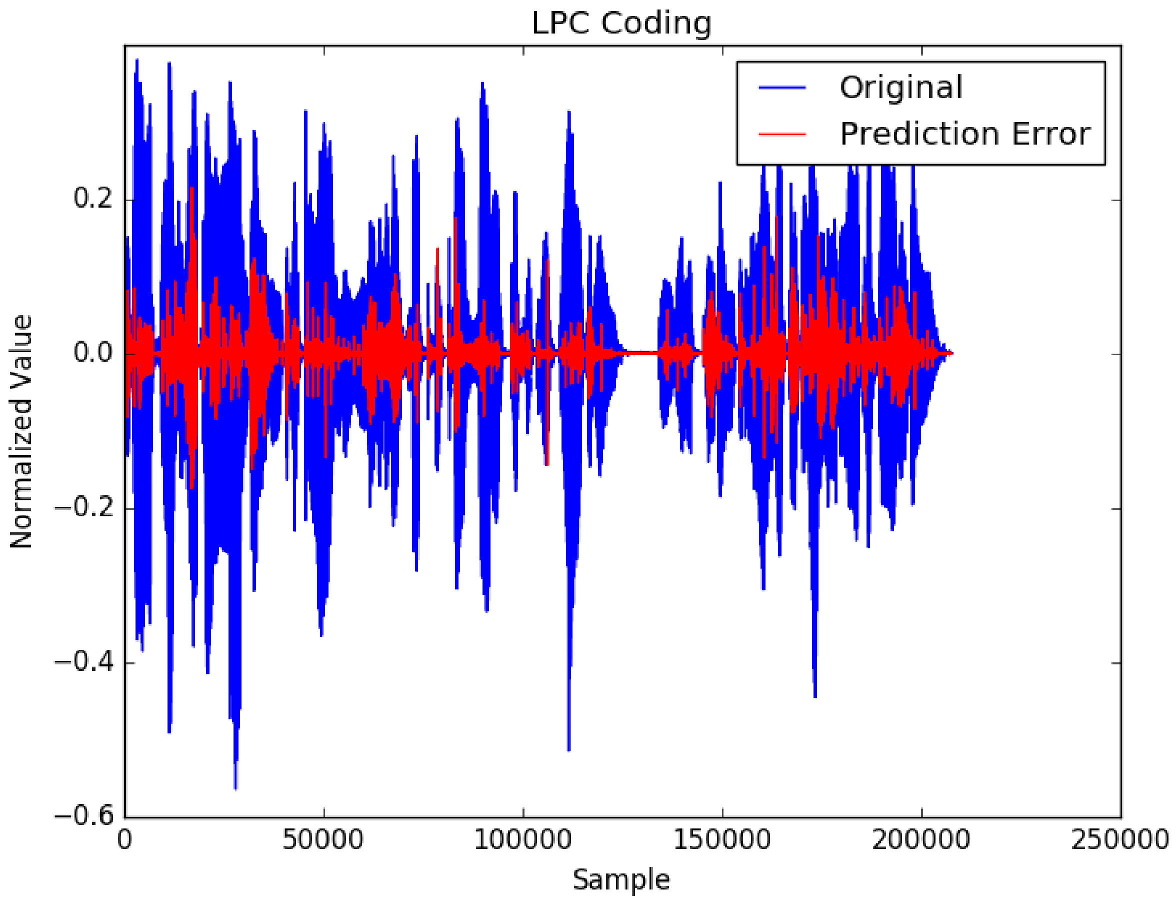
Our speech signal is sampled at 32 kHz, hence a block of 20 ms has **640 samples**. We write a python file with name “`lpcexample.py`”, with the following content,

```
import numpy as np
from sound import *
import matplotlib.pyplot as plt
import scipy.signal as sp

x, fs = wavread('fspeech.wav');
#convert to float array type, normalize to -1<x<1:
x = np.array(x,dtype=float)/2**15
print np.size(x)
sound(2**15*x,fs)

L=10 #predictor lenth
len0 = np.max(np.size(x))
e = np.zeros(np.size(x)) #prediction error variable initialization
blocks = np.int(np.floor(len0/640)) #total number of blocks
state = np.zeros(L) #Memory state of prediction filter
#Building our Matrix A from blocks of length 640 samples and
process:
```

```
for m in range(0,blocks):
    A = np.zeros((640-L,L)) #trick: up to 630 to avoid zeros in
the matrix
    for n in range(0,640-L):
        A[n,:] = np.flipud(x[m*640+n+np.arange(L)])

    #Construct our desired target signal d, one sample into the
future:
    d=x[m*640+np.arange(L,640)];
    #Compute the prediction filter:
    h = np.dot(np.dot(np.linalg.inv(np.dot(A.transpose(),A)),
A.transpose()), d)
    hperr = np.hstack([1, -h])
    e[m*640+np.arange(0,640)], state= sp.lfilter(hperr,
[1],x[m*640+np.arange(0,640)], zi=state)


#The mean-squared error now is:
print "The average squared error is:",
np.dot(e.transpose(),e)/np.max(np.size(e))
#The average squared error is: 0.000113347859337
#We can see that this is only about 1 / 4 of the previous pred.
Error!
print "Compare that with the mean squared signal power:",
np.dot(x.transpose(),x)/np.max(np.size(x))
#0.00697569381701
print "The Signal to Error ratio is:",
np.dot(x.transpose(),x)/np.dot(e.transpose(),e)
#61.5423516403
#So our LPC pred err energy is more than a factor of 61 smaller
than the
#signal energy!
#Listen to the prediction error:
sound(2**15*e,fs)
#Take a look at the signal and it's prediction error:
plt.figure()
plt.plot(x)
#plt.hold(True)
plt.plot(e,'r')
plt.xlabel('Sample')
plt.ylabel('Normalized Value')
plt.legend(('Original','Prediction Error'))
plt.title('LPC Coding')
plt.show()

#Decoder:
xrek=np.zeros(x.shape) #initialize reconstructed signal memory
state = np.zeros(L) #Initialize Memory state of prediction filter
for m in range(0,blocks):
    hperr = np.hstack([1, -h[m,:]])
    #predictive reconstruction filter: hperr from numerator to
denominator:
    xrek[m*640+np.arange(0,640)] , state = sp.lfilter([1], hperr,
e[m*640+np.arange(0,640)], zi=state)

#Listen to the reconstructed signal:
sound(2**15*xrek,fs)
```

Now execute the program with

```
python lpcexample.py
```



Here it can be seen that the prediction error is even smaller than before.

The **decoder** works the way as shown in the previous example, and the reconstructed speech can be heard in the end.

LPC type **coders** are for instance speech coders, where usually 12 coefficients are used for the prediction, and **transmitted as side information** every 20 ms. The prediction error is parameterized and transmitted as parameters with a very low bit rate. This kind of system is used for instance in most digital **cell phones** systems.

Least Mean Squares (LMS) Algorithm

Unlike the LPC algorithm above, which computes prediction coefficients for a block of samples and transmits these coefficients alongside the prediction error to the receiver, the LMS algorithm updates the prediction coefficients after **each** sample, but based only on **past** samples. Hence here we need the assumption that the signal statistics does not change much from the past to the present. Since it is based on the past samples, which are also available as decoded samples at the decoder, we do not need to transmit the coefficients to the decoder. Instead, the decoder carries out the same computations in **synchrony with the encoder**.

Instead of a matrix formulation, we use an iterative algorithm to come up with a solution for the prediction coefficients $\boldsymbol{h}$ the vector which contains the time-reversed impulse response of our predictor. To show the dependency on the time n , we now call the vector of prediction coefficients $\boldsymbol{h}(n)$, with

$$\boldsymbol{h}(n)=[h_0(n),\ldots,h_{L-1}(n)]$$

Again we would like to **minimize** the **mean quadratic prediction error** (with the prediction error $e(n)=x(n)-\hat{x}(n)$),

$$E\left[\left(x(n)-\hat{x}(n)\right)^2\right]=E\left[\left(x(n)-\sum_{k=0}^{L-1}h_k(n)\,x(n-1-k)\right)^2\right]$$

Instead of using the closed form solution, which lead to the Wiener-Hopf Solution, we now take an **iterative** approach to approach the minimum of this optimization function.

We use the algorithm of **Steepest Descent** (also called **Gradient Descent**), see also http://en.wikipedia.org/wiki/Gradient_descent, to iterate towards the minimum,

$$\boldsymbol{h}(n+1)=\boldsymbol{h}(n)-\alpha\cdot\nabla f(\boldsymbol{h}(n))$$

with **optimization (objective or error) function** as our squared prediction error,

$$f(\boldsymbol{h}(n))=\left(x(n)-\sum_{k=0}^{L-1} h_k(n)\,x(n-1-k)\right)^2$$

Observe that we omitted the Expectation operator “E” for simplicity; we expect that after several update steps there will be inherently some averaging. This is also called “**Stochastic Gradient Descent**”

We have the Gradient as the row vector

$$\nabla f(n)=\left[\frac{\partial f(\boldsymbol{h}(n))}{\partial h_0(n)},\ldots,\frac{\partial f(\boldsymbol{h}(n))}{\partial h_{L-1}(n)}\right]$$

and we get the individual derivatives as

$$\frac{\partial f(\boldsymbol{h}(n))}{\partial h_k(n)}=2\cdot e(n)\cdot(-x(n-1-k))$$

(with k=0,..,L-1). So together we obtain the **LMS algorithm** or **update rule** as

$$h_k(n+1)=h_k(n)+\alpha\cdot e(n)\cdot x(n-1-k)$$

for $k=0,\ldots,L-1$, where α is a tuning parameter (the factor 2 is incorporated into α), with which we can trade off

convergence speed and convergence accuracy. A derivation or computation of the α value can be found in the lecture **slide set 15** of our lecture “**Multirate Signal Processing**”.

In vector form, this LMS update rule is

$$\boldsymbol{h}(n+1)=\boldsymbol{h}(n)+\alpha\cdot e(n)\cdot\boldsymbol{x}(n)$$

where $\boldsymbol{x}(n)=[x(n-1),x(n-2),\dots,x(n-L)]$.

Observe that we need no matrices or matrix inverses in this case, just this simple update rule, and it still works! It still converges to the “correct” coefficients!
For the prediction coefficients h we have something like a “**sliding window**” of the past L samples $\boldsymbol{x}(n)$ of our signal.
For α there are different “recipes”, for instance the so-called normalized LMS (NLMS) uses the inverse signal power as α . If the signal power i α s one, then α can be one. But in general it is subject to “hand tuning”, trial and error.

LMS Python Example

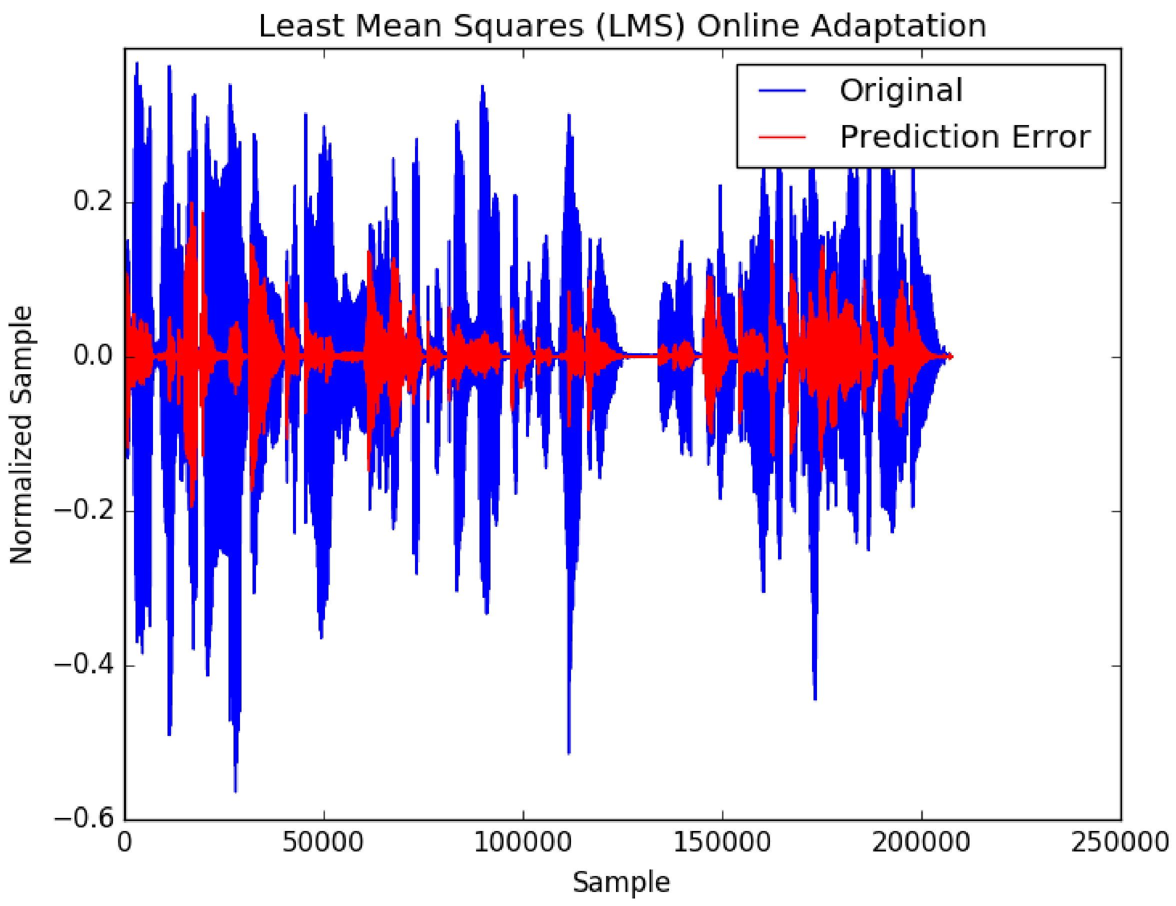
```
import numpy as np
from sound import *
import matplotlib.pyplot as plt

x, fs = wavread('fspeech.wav')
#normalized float, -1<x<1
x = np.array(x,dtype=float)/2**15
print np.size(x)
e = np.zeros(np.size(x))

h = np.zeros(10)

for n in range(10, len(x)):
    #prediction error and filter, using the vector of the time reversed
IR:
    e[n] = x[n] - np.dot(np.flipud(x[n-10+np.arange(0,10)]), h)
    #LMS update rule, according to the definition above:
    h = h + 1.0* e[n]*np.flipud(x[n-10+np.arange(0,10)])

print "Mean squared prediction error:", np.dot(e, e) /np.max(np.size(e))
#0.000215852452838
print "Compare that with the mean squared signal power:",
np.dot(x.transpose(),x)/np.max(np.size(x))
#0.00697569381701
print "The Signal to Error ratio is:",
np.dot(x.transpose(),x)/np.dot(e.transpose(),e)
#32.316954129056604, half as much as for LPC.

#listen to to the prediction error:
sound(2**15*e, fs)

plt.figure()
plt.plot(x)
#plt.hold(True)
plt.plot(e,'r')
plt.xlabel('Sample')
plt.ylabel('Normalized Sample')
plt.title('Least Mean Squares (LMS) Online Adaptation')
plt.legend(('Original','Prediction Error'))
plt.show()
```

Execute the program with

`python lmsexample.py`

Observe: its prediction error is bigger than in the **LPC** case, but we also don't need to transmit the prediction coefficients as side information.

The comparison plot of the original to the prediction error,



For the **decoder** we get the reconstruction

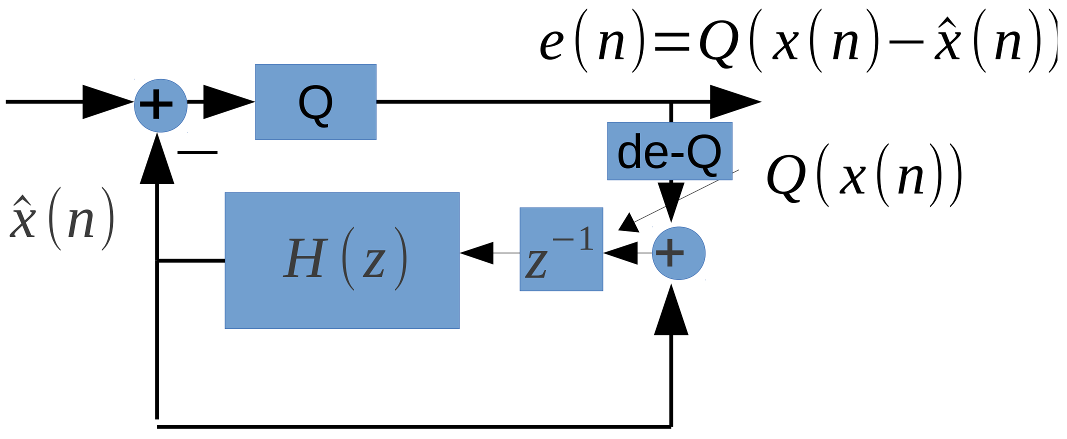
```
# Decoder
h = np.zeros(10);
xrek = np.zeros(np.size(x));
for n in range(10, len(x)):
    xrek[n] = e[n] + np.dot(np.flipud(xrek[n-10+np.arange(10)]), h)
    #LMS update:
    h = h + 1.0 * e[n]*np.flipud(xrek[n-10+np.arange(10)]);
```

```
plt.plot(xrek)
plt.show()
#Listen to the reconstructed signal:
sound(2**15*xrek,fs)
```

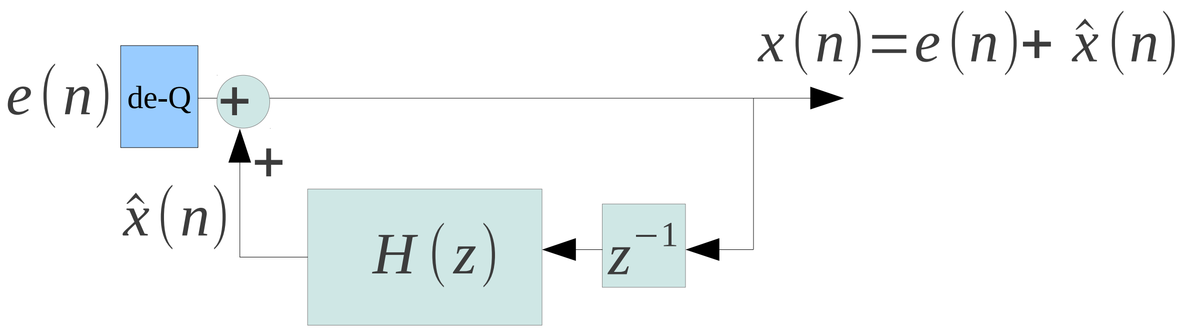
Sensitivity of the decoder for transmission **errors**: In the code for the decoder in the LMS update for the predictor h, correctly we need xrek instead of x (since x is not available in the decoder). The slightest computation errors, for instance **rounding errors**, are sufficient to make the decoder diverge and stop working after a few syllables of the speech. Try it out. We see that the computed **prediction coefficients differ** in the last digits between encoder and decoder, which is enough for **increasing divergence** between encoder and decoder, until the decoded signal “explodes” (becomes huge from instability).

This shows that **LMS is very sensitive to transmission errors**.
To avoid at least the computation errors, we need to include quantization in the process.

Predictive Encoder with Quantizer



Here we can see a predictive encoder with **quantization** of the prediction error. In order to make sure the encoder predictor works on the quantized values, like in the decoder, it uses a **decoder in the encoder** structure, which produces the quantized reconstructed value $Q(x(n))$ (the function $Q()$ here include both, quantizer and de-quantizer).
The decoder stays the same, except for the de-quantization of the prediction error in the beginning:



Observe: The reconstructed signal x(n) is the same as for the encoder, plus the quantization error from the quantizer:

$$Q(x(n)) = e(n) + \hat{x}(n)$$

LMS with Quantizer Python Example

Write a Python program file with

```
import numpy as np
from sound import *
import matplotlib.pyplot as plt

x, fs = wavread('fspeech.wav')
#normalized float, -1<x<1
x = np.array(x,dtype=float)/2**15
print np.size(x)
e = np.zeros(np.size(x))
xrek=np.zeros(np.size(x));
P=0;
L=10
h = np.zeros(L)
#have same 0 starting values as in decoder:
x[0:L]=0.0
quantstep=0.01;
for n in range(L, len(x)):
    if n> 4000 and n< 4010:
      print( "encoder h: ", h, "e=", e)
    #prediction error and filter, using the vector of the time reversed
IR:
    #predicted value from past reconstructed values:
    P=np.dot(np.flipud(xrek[n-L+np.arange(L)]), h)
    #quantize and de-quantize e to step-size 0.05 (mid tread):
    e[n]=np.round((x[n]-P)/quantstep)*quantstep;
    #Decoder in encoder:
    #new reconstructed value:
    xrek[n]=e[n]+P;
    #LMS update rule:
    h = h + 1.0* e[n]*np.flipud(xrek[n-L+np.arange(L)])

print "Mean squared prediction error:", np.dot(e, e) /np.max(np.size(e))
#without quant.: 0.000215852452838
#with quant. with 0.01 : 0.000244936708861
#0.00046094397241
#quant with 0.0005: 0.000215872774695
print "Compare that with the mean squared signal power:",
np.dot(x.transpose(),x)/np.max(np.size(x))
print "The Signal to Error ratio is:",
np.dot(x.transpose(),x)/np.dot(e.transpose(),e)
#The Signal to Error ratio is: 28.479576824, a little less than without
quant.
#listen to it:
sound(2**15*e, fs)

plt.figure()
plt.plot(x)
```

```
#plt.hold(True)
plt.plot(e,'r')
plt.xlabel('Sample')
plt.ylabel('Normalized Sample')
plt.title('Least Mean Squares (LMS) Online Adaptation')
plt.legend(('Original','Prediction Error'))
plt.show()

# Decoder
h = np.zeros(L);
xrek = np.zeros(np.size(x));
for n in range(L, len(x)):
    if n> 4000 and n< 4010:
       print "decoder h: ", h
    P=np.dot(np.flipud(xrek[n-L+np.arange(L)]), h)
    xrek[n] = e[n] + P
    #LMS update:
    h = h + 1.0 * e[n]*np.flipud(xrek[n-L+np.arange(L)]);

plt.plot(xrek)
plt.xlabel('Sample')
plt.ylabel('Normalized Sample')
plt.title('The Reconstructed Signal')
plt.show()

#Listen to the reconstructed signal:
sound(2**15*xrek,fs)
```

Execute it with

```
python lmsquantexample.py
```

Observe: Because of the quantization, the prediction error now clearly increased.

Observe: The signal is now **fully decoded**, even with quantization, although with a little noise, which was to be expected. But we can avoid the noise by reducing the quantization step size.

Observe that this structure for the

decoder in the encoder also applies to the **other prediction methods**.