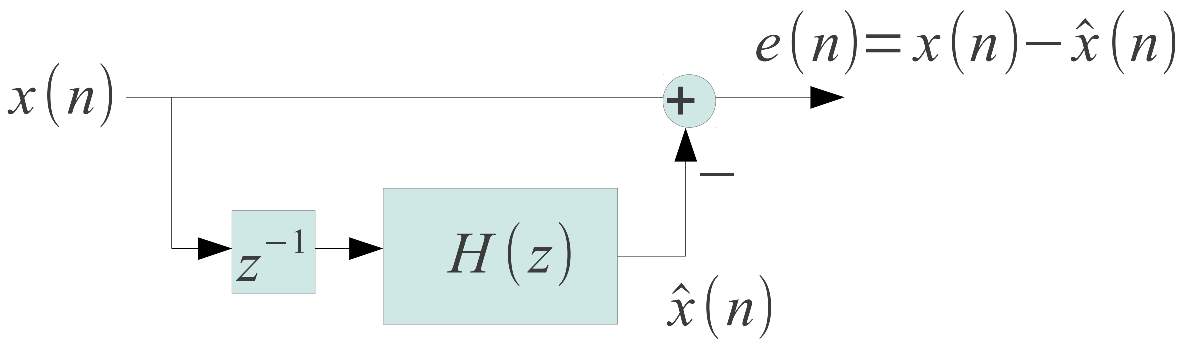
Digital Signal Processing 2/ Advanced Digital Signal Processing Lecture 14, Matched Filter, Prediction

Gerald Schuller, TU Ilmenau

Prediction

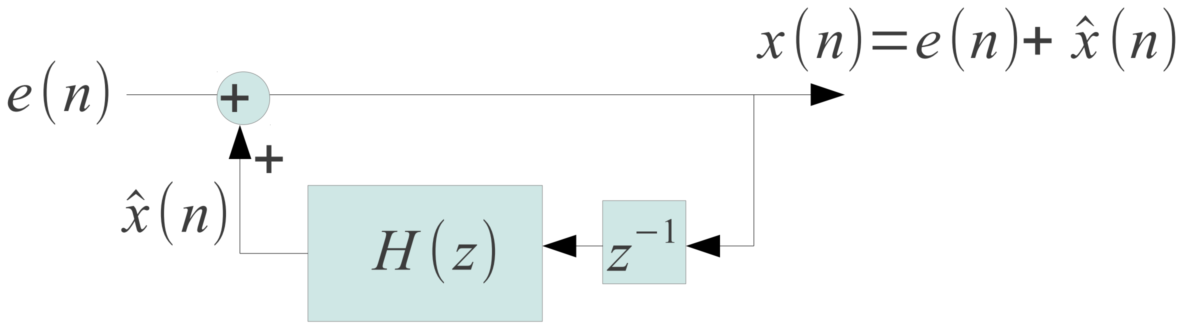
A system which produces the **prediction error** (for instance of part of an encoder) as an output is the following,



Here, $x(n)$ is the signal to be predicted, $H(z)$ is our prediction filter, whose **coeffiicents** 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

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

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

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

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

Octave/Matlab Example

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

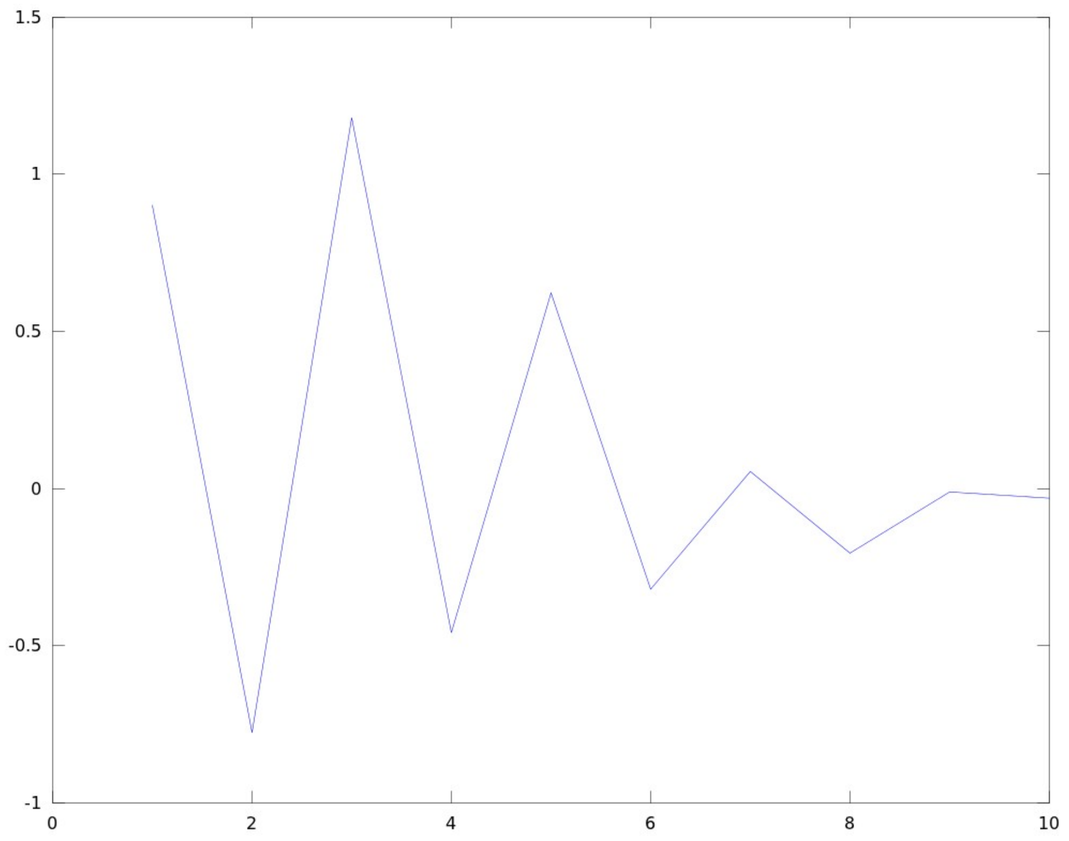
Read in the female speech sound:

```
x=wavread('fspeech.wav');

size(x)

%ans =
%207612 1
%listen to it:
sound(x,32000)
```

%Construct our Matrix A from x:

```
A=zeros(100000,10);

for m=1:100000,
  A(m,:)=x(m-1+(1:10))';
end;

%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(11:100010);

%Compute the prediction filter:
h=inv(A'*A) * A' * d;
flipud(h)
ans =
0.900784
-0.776478
1.179245
-0.458494
0.622308
-0.320261
0.054122
-0.205571
-0.011090
-0.030701
```

```
plot(flipud(h))
```

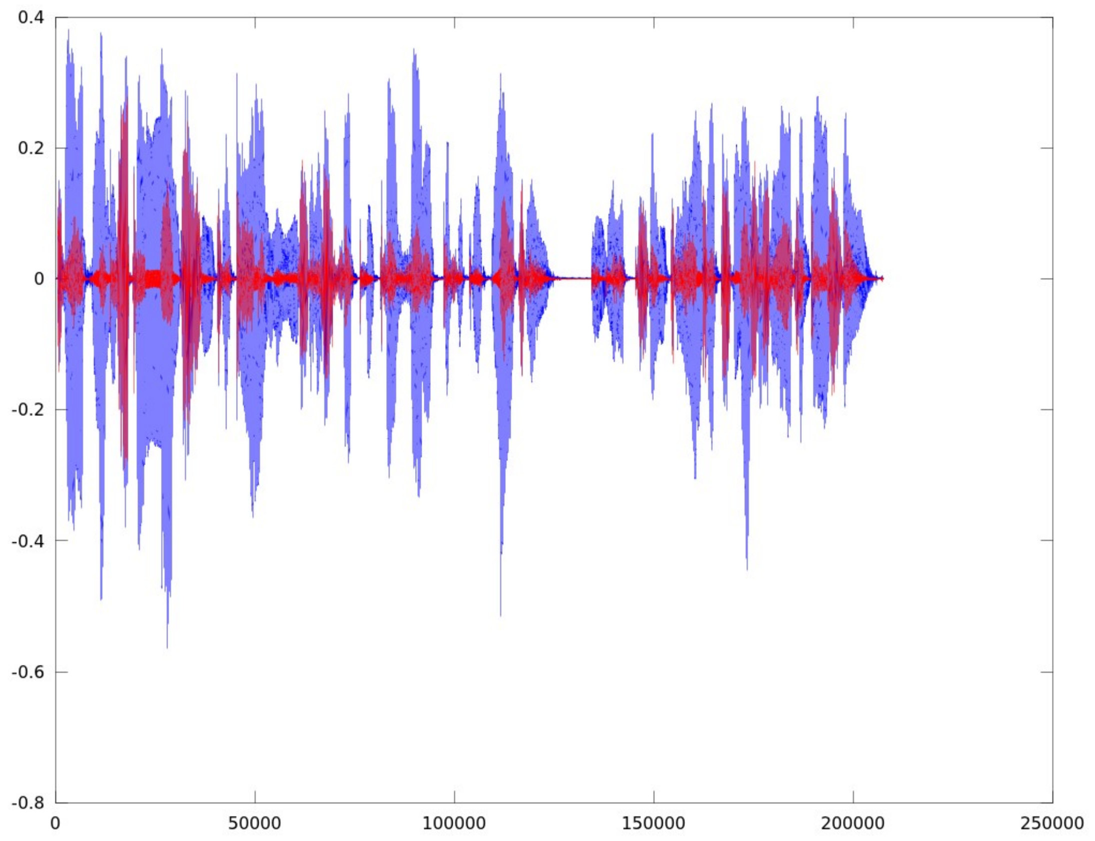


The impulse response of the resulting prediction filter.
Our corresponding prediction error filter is $H_{perr}(z)=1-z^{-1}\cdot H(z)$, in Octave/Matlab:

```
hperr=[1; -flipud(h)]
hperr =
1.000000
-0.900784
0.776478
-1.179245
0.458494
-0.622308
0.320261
-0.054122
0.205571
0.011090
0.030701
```

%The prediction error e(n) is

```
e=filter(hperr,1,x);
%The mean-squared error is:
e'*e/max(size(e))
%ans = 4.3287e-04

%Compare that with the mean squared signal power:

x'*x/max(size(x))
%ans = 0.0069761
%Which is more than 10 times as big as the prediction error!
%which shows that it works!


%Listen to it:
sound(e,32000)

%Take a look at the signal and it's prediction error:
plot(x)
hold
plot(e,'r')
```



Here we can see the original signal in blue, and it's prediction error in red.

Online Adaptation

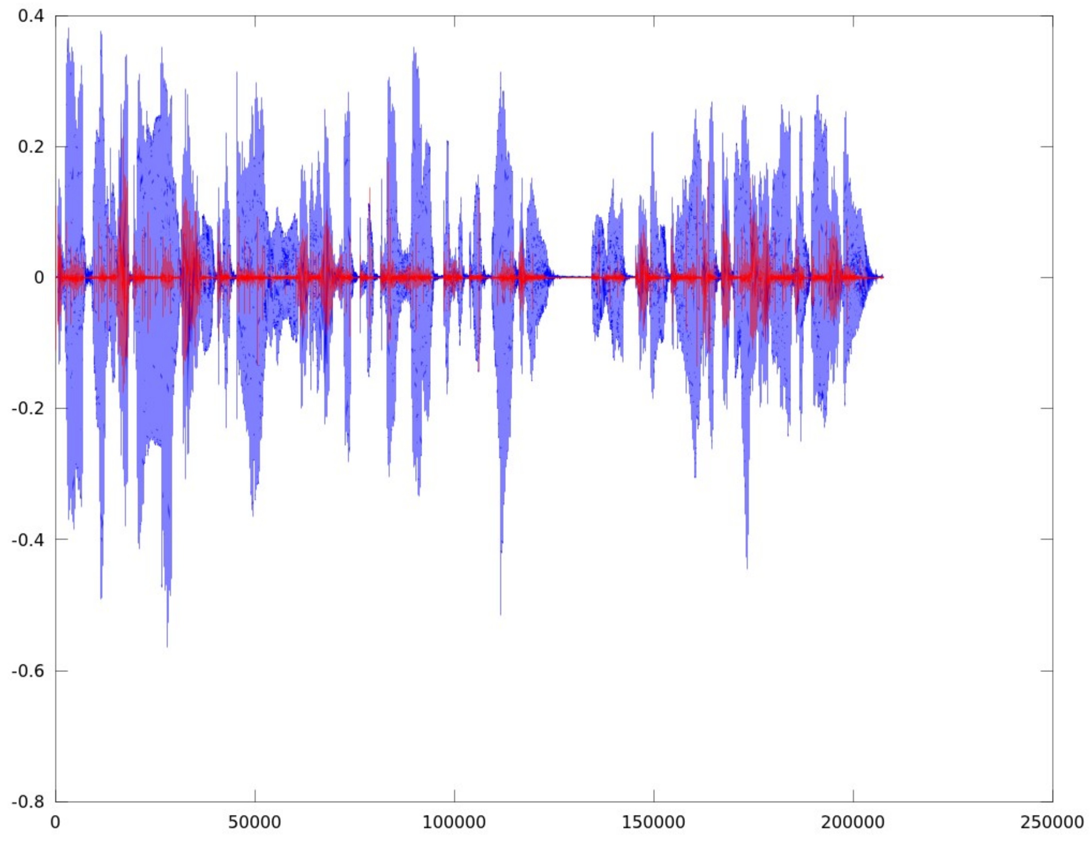
The previous example calculated the prediction coefficients for the entire speech file. But when we look at the signal waveform, we see that its characteristics and hence its statistics is changing, it is **not stationary.** Hence we can expect a prediction improvement if we divide the speech signal 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.

Matlab/Octave Example

Our speech signal is sampled at 32 kHz, hence a block of 20 ms has **640 samples**.

```
x=wavread('fspeech.wav');
size(x)
%ans =
%207612 1
len=max(size(x));
e=zeros(size(x)); %prediction error variable initialization
blocks=floor(len/640); %total number of blocks
state=zeros(1,10); %Memory state of prediction filter
%Building our Matrix A from blocks of length 640 samples and process:
for m=1:blocks,
  A=zeros(630,10); %trick: up to 630 to avoid zeros in the matrix
  for n=1:630,
    A(n,:)=x((m-1)*640+n-1+(1:10))';
  end;

  %Construct our desired target signal d, one sample into the future:
  d=x((m-1)*640+(11:640));

  %Compute the prediction filter:
  h=inv(A'*A) * A' * d;

  hperr=[1; -flipud(h)];
  [e((m-1)*640+(1:640)),state]=filter(hperr,1,x((m-1)*640+(1:640)),state);
end;


%The mean-squared error now is:
e'*e/max(size(e))
%ans = 1.1356e-04
%We can see that this is only about 1 / 4 of the previous pred. Error!
%Compare that with the mean squared signal power:

x'*x/max(size(x))
%ans = 0.0069761
x'*x/(e'*e)
ans = 61.429

%So our LPC pred err energy is more than a factor of 61 smaller than the
signal energy!

%Listen to the prediction error:
sound(e,32000)
```

```
%Take a look at the signal and it's prediction error:
plot(x)
hold
plot(e,'r')
```



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

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 samples 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.

To show the dependency on the time n , we now call the vector of prediction coefficients $\boldsymbol{h}(n)$, with

$$\boldsymbol{h}(n)=[h_L(n),\ldots,h_1(n)]$$

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

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

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,

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

with **optimization function** as our squared prediction error,

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

We have

$$\nabla f(n)=\left[\frac{\partial f(\boldsymbol{h}(n))}{\partial h_L(n)},\ldots,\frac{\partial f(\boldsymbol{h}(n))}{\partial h_1(n)}\right]$$

and we get

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

So together we obtain the **LMS algorithm** or **update rule** as

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

for $k=1,\dots L$, where μ is a tuning parameter (the factor 2 is incorporated into μ), with which we can trade off **convergence speed and convergence accuracy**.

In vector form, this LMS update rule is

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

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

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 μ there are different “recipes”, for instance the so-called normalized LMS (NLMS) uses the inverse signal power as μ . If the signal power is one, then μ can be one. But in general it is subject to “hand tuning”, trial and error.

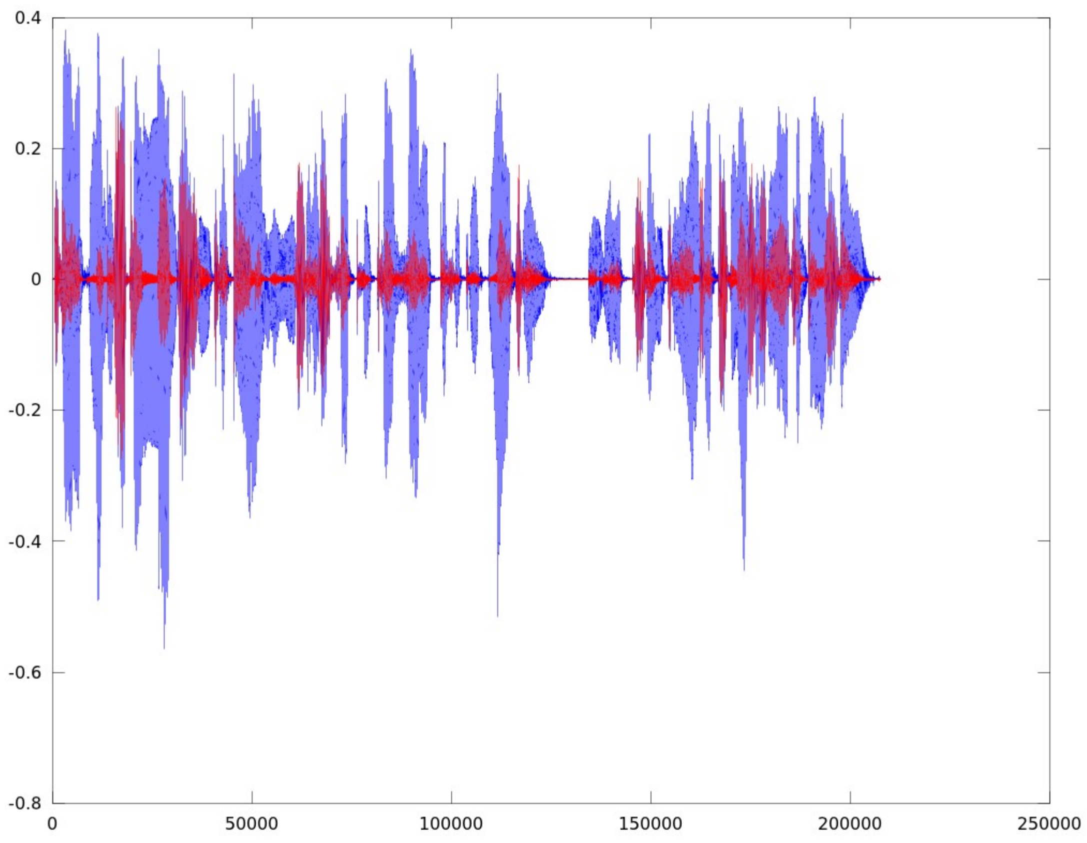
LMS Octave/Matlab Example

```
x=wavread('fspeech.wav');
size(x)
e=zeros(size(x));
%ans =
%207612 1
h=zeros(10,1);
for n=11:max(size(x)),
  %prediction error and filter, using the vector of the time reversed IR:
  e(n)=x(n) - x(n-10+(0:9))' * flipud(h);
  %LMS update rule, according to the definition above:
  h= h + 1.0*e(n)*flipud(x(n-10+(0:9)));
end


e'*e/max(size(e))
%ans = 2.1586e-04
%listen to it:
sound(e,32000);
```

%This is bigger than in the **LPC** case, but we also don't need to transmit the prediction coefficients.

Plot a comparison of the original to the prediction error,

```
plot(x);
hold
plot(e,'r')
```



Listen to the prediction error,

```
sound(e,32000)
```

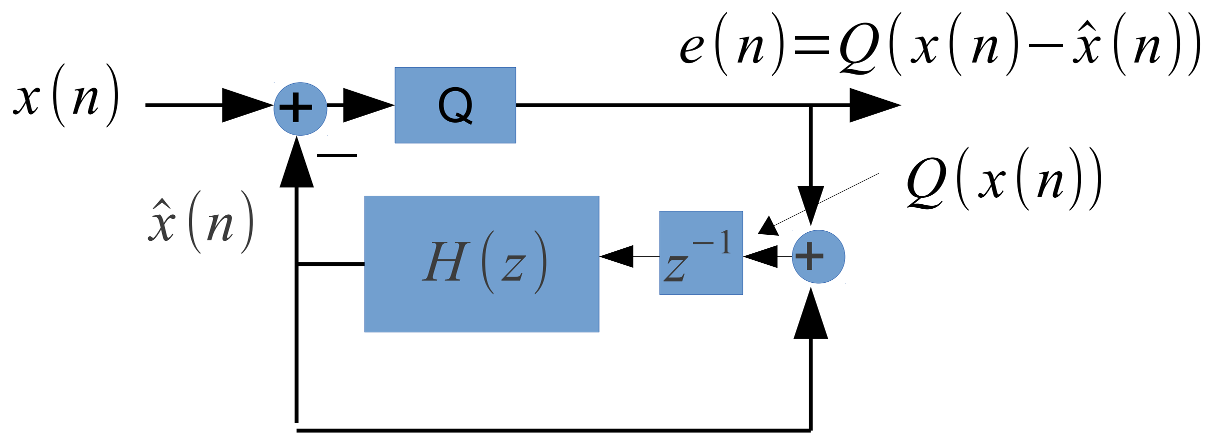
For the **decoder** we get the reconstruction

```
h=zeros(10,1);
xrek=zeros(size(x));

for n=11:max(size(x)),
  xrek(n)=e(n) + xrek(n-10+(0:9))' * h;
  h= h + 1*e(n)*(x(n-10+(0:9)));
end

%Listen to the reconstructed signal:
sound(xrek,32000)
```

Sensitivity of the decoder for transmission **errors**: In the code for the decoder in the LMS update for the predictor h, correctly we would need xrek instead of x (since x is not available in the decoder). But slight computation errors are sufficient to make the decoder diverge and stop working after a few syllables of the speech. Try it out.

This shows that **LMS is very sensitive to transmission errors**.
To avoid at least the comptation 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 decoder stays the same, except for the de-quantization of the prediction error in the beginning.

LMS with Quantizer Octave Example

```
x=wavread('fspeech.wav');
size(x)
e=zeros(size(x));
%ans =
%207612 1
h=zeros(10,1);
xrek=zeros(size(x));
P=0;
for n=11:max(size(x)),
  %prediction filter, using the vector of the time reversed IR:
  %predicted value from past reconstructed values:
  P=xrek(n-10+(0:9))' * flipud(h);

  %quantize and de-quantize e to step-size 0.05 (mid tread):
  quantstep=0.05
  e(n)=round((x(n) – P)/quantstep)*quantstep;
  %Decoder in encoder:
  %new reconstructed value:
  xrek(n)=e(n)+P;
  %LMS update rule, according to the definition above:
  h= h + 1.0*e(n)*flipud(xrek(n-10+(0:9)));
end


e'*e/max(size(e))
%ans = 4.8325e-04
%listen to it:
sound(e,32000);
```

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

Decoder:

```
h=zeros(10,1);
xrek=zeros(size(x));

for n=11:max(size(x)),
  P=xrek(n-10+(0:9))' * h;
  xrek(n)=e(n) + P;
  %compute/update the already flipped version of h:
  h= h + 1*e(n)*(xrek(n-10+(0:9)));
end

%Listen to the reconstructed signal:
sound(xrek,32000)
```

Observe: The signal is now **fully decoded**, even with the correct xrek in the update of h, although with a little noise as a result of the quantization, which was to be expected. This shows that small computation differences between encoder and decoder don't matter anymore, because they are dominated by the quantization, which can more easily be identical between encoder and decoder.

Observe that this structure for the **decoder in the encoder also applies** to the **other prediction methods**.