Digital Signal Processing 2/ Advanced Digital Signal Processing
Lecture 12,
Wiener and Matched Filter, Prediction

Gerald Schuller, TU Ilmenau

These filters goal is to reduce the influence of **noise or distortions.**
Assume: x(n) is the original signal, y(n) is the distorted signal.
Example: y(n)=x(n)+v(n)
where v(n) is is assumed to be independent white noise.

-Wiener filter $h_W(n)\ :\ \ y(n)*h_W(n) \rightarrow x(n)$ **(signal fidelity**, the reconstruction is close to the original, for instance for de-noising an image or audio signal, where the audio signal does not need to be deterministic)

-Matched filter $h_M(n)\ :\ y(n)*h_M(n) \rightarrow x(n)*h_M(n)$
(no signal fidelity, just **high SNR for detection**, in communication applications, where you would like to detect a 0 or 1, or any given **known** signal, usually a **deterministic signal**; object recognition in images, face recognition).

Wiener Filter

The goal here is an approximation of the original signal x(n) in the **least mean squared** sense, meaning we would like to **minimize the mean quadratic error** between the filtered and the original signal, because it is mathematically convenient.

We have a filter system with Wiener Filter $h_W(n)$

$$y(n) * h_w(n) \rightarrow x(n)$$

meaning we filter our distorted signal y(n) with our still unknown filter $h_W(n)$.

The convolution of $h_W(n)$ (with filter length L) with y(n) is

$$\sum_{m=0}^{L-1} y(n-m) \cdot h_W(m) \rightarrow x(n)$$

A well known mathematical approach to obtain the minimum of a mean squared error in a matrix framework is the so-called Moore-Penrose **Pseudo Inverse** (http://en.wikipedia.org/wiki/Moore%E2%80%93Penrose_pseudoinverse). To be able to apply it we can reformulate our convolution equation as a matrix multiplication.

Let's define 2 vectors. The first is a vector of the the **past L samples of our noisy signal** y, with the past on the right (flipped), up to the present sample at time n, (bold face font to indicate that it it a vector)

$$\boldsymbol{y}(n) = [y(n), y(n-1), \ldots, y(n-L+1)]$$

The next vector contains the **impulse response**,

$$\boldsymbol{h_W} = [h_W(0), h_W(1), \ldots, h_W(L-1)]$$

Using those 2 vectors, we can rewrite our convolution equation above as a vector multiplication,

$$x(n) = \boldsymbol{y}(n) \cdot \boldsymbol{h_W^T}$$

Observe that $\boldsymbol{h_W}$ has no time index because it already contains all the samples of the time-reversed impulse response, and is constant.

We can now also put the output signal x(n) into the **row vector**,

$$\boldsymbol{x} = [x(0), x(1), \ldots]$$

To obtain this vector, we simply assemble all the row vectors of our noisy signal $\boldsymbol{y}(n)$ into a matrix $\boldsymbol{A}$,

$$\boldsymbol{A} = \begin{bmatrix} \boldsymbol{y}(0) \\ \boldsymbol{y}(1) \\ \vdots \end{bmatrix}$$

With this matrix, we obtain the result of our convolution at all time steps n to

$$\boldsymbol{A} \cdot \boldsymbol{h_W^T} \rightarrow \boldsymbol{x^T}$$

this is just another way of writing our convolution.

For the example of a filter length of h_W of L=2 hence we get,

$$\begin{bmatrix} y(1) & y(0) \\ y(2) & y(1) \\ y(3) & y(2) \\ \vdots & \vdots \end{bmatrix} . \begin{bmatrix} h_W(0) \\ h_W(1) \end{bmatrix} \rightarrow \begin{bmatrix} x(0) \\ x(1) \\ x(2) \\ \vdots \end{bmatrix}$$

This is now the **matrix multiplication** formulation of our

convolution.

We can now obtain the minimum mean squared error **solution** of this matrix multiplication using the Moore-Penrose pseudo inverse.
This pseudo-inverse finds the column vector $\boldsymbol{h^T}$ which minimizes the distance to a given $\boldsymbol{x}$ with the matrix $\boldsymbol{A}$ (which contains our signal y to be filtered):

$$\boldsymbol{A \cdot h_W^T \rightarrow x^T}$$

Matrix $\boldsymbol{A}$ and vector $\boldsymbol{x}$ are known (this is done in a **"trainings"-phase** to obtain the Wiener filter coefficients $\boldsymbol{h_W}$, from noisy signals in matrix **A** and the known clean signals in vector **x**). Vector $\boldsymbol{h_W}$ is unknown so far. After the trainings-phase the filter can also be applied to **similar signals**.

This problem can be solved exactly if the matrix $\boldsymbol{A}$ is **square and invertible**. Just multiplying the equation with $\boldsymbol{A^{-1}}$ from the left would give us the solution

$$\boldsymbol{h_W^T = A^{-1} \cdot x^T}$$

This cannot be done, if $\boldsymbol{A}$ is **non-square,** for instance if it has many more rows than columns. In this case we don't have an exact solution, but many solutions that come close to $\boldsymbol{x}$. We would like to obtain the solution which comes **closest** to $\boldsymbol{x}$ in a mean squared error distance sense (also called **Euclidean Distance**).

This solution is derived using the pseudo-inverse. First we

multiply both sides by A^T ,

$$A^T \cdot A \cdot h_W^T = A^T \cdot x^T$$

Here, $A^T \cdot A$ is now a square matrix, and the formulation is no longer over-determined, hence we can replace the right arrow by an equal sign. The square matrix usually invertible, such that we obtain our solution

$$h_W^T = \left(A^T \cdot A\right)^{-1} A^T \cdot x^T$$

This h_W is now the **solution** we where looking for. This solution has the minimum mean squared distance to the un-noisy version of all solutions.

Python Example for denoising speech:

```
ipython -pylab
from sound import *
from scipy import signal as sp
x, fs = wavread('fspeech.wav')
#make x a matrix and transpose it into a
column:
x=matrix(x).T
sound(array(x), fs)

#additive zero mean white noise (for -
2**15<x<+2**15):
y=x+0.1*(random.random(shape(x))-0.5)*2**15
```

```
sound(array(y), fs)
```

#we assume L=**10 coefficients** for our Wiener filter.
#10 to 12 is a good number for speech signals.

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

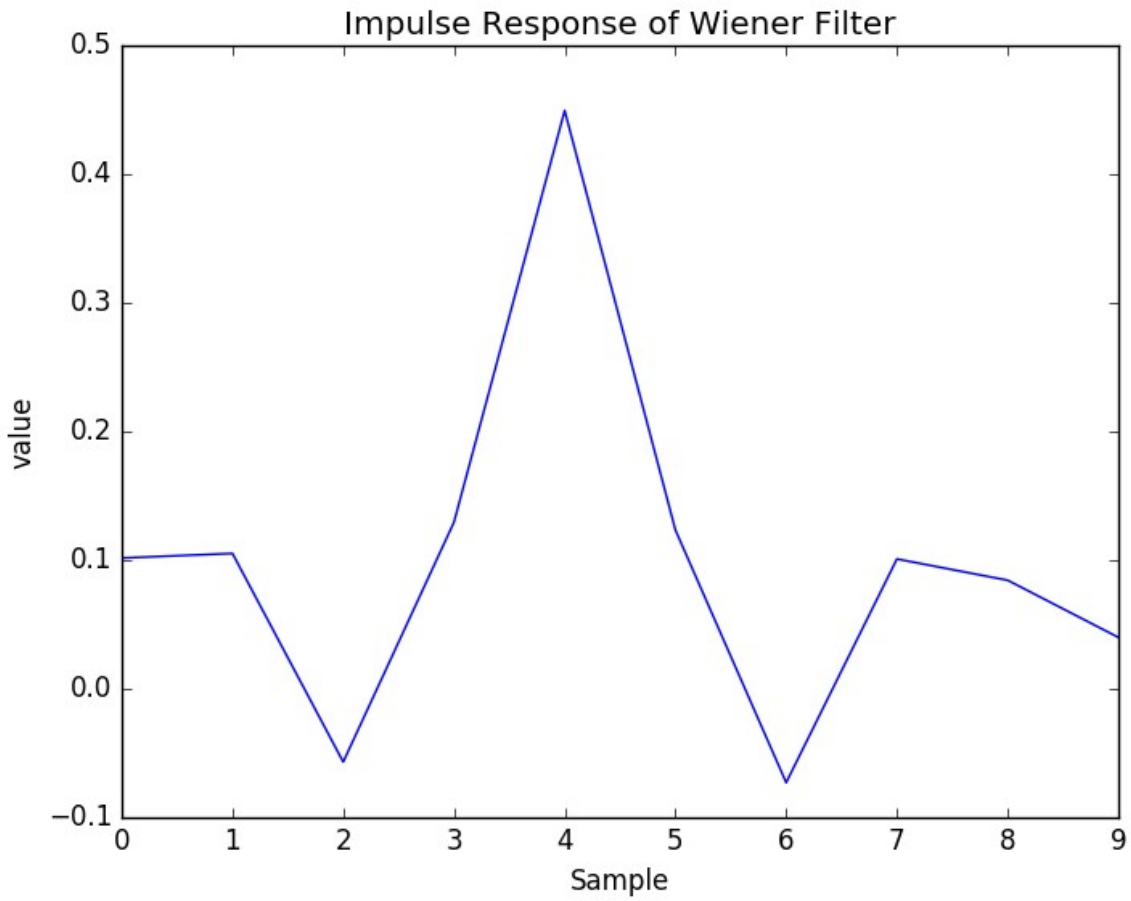
#Our matrix has 100000 rows and 10 colums:

```
print A.shape
# (100000, 10)
```

#Compute Wiener Filter:

#Trick: **allow filter delay of 5 samples**
#to get better working denoising.
#This corresponds to the center of our Wiener filter.
#The desired signal hence is x[5:100005].
#Observe: Since we have the matrix type, operator
„*“ is matrix multiplication!

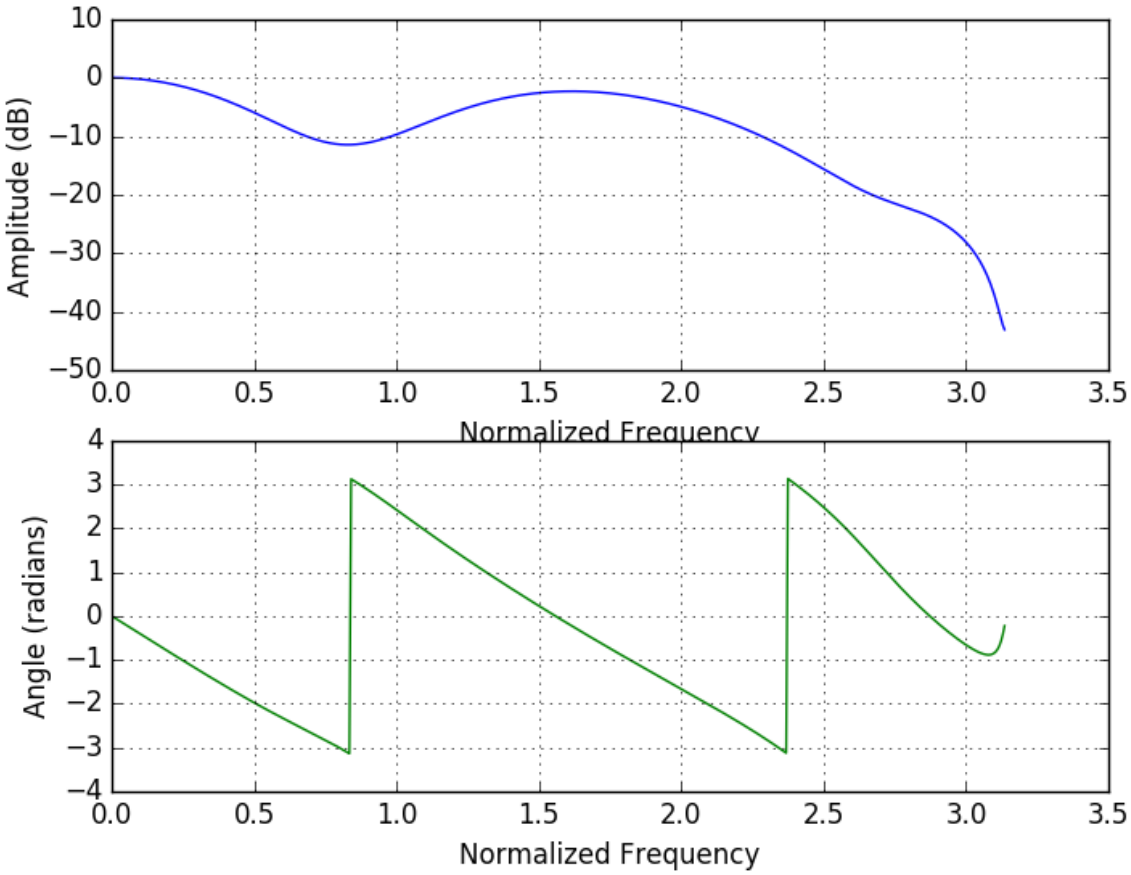
```
h=inv(A.T*A)*A.T*x[5:100005]
plot(h)
xlabel('Sample')
ylabel('value')
title('Impulse Response of Wiener Filter')
```



Observe that for this impulse response we see a delay of 4 samples (the peak is at sample number 4).

Its frequency response is

```
from freqz import *
freqz(flipud(h))
```

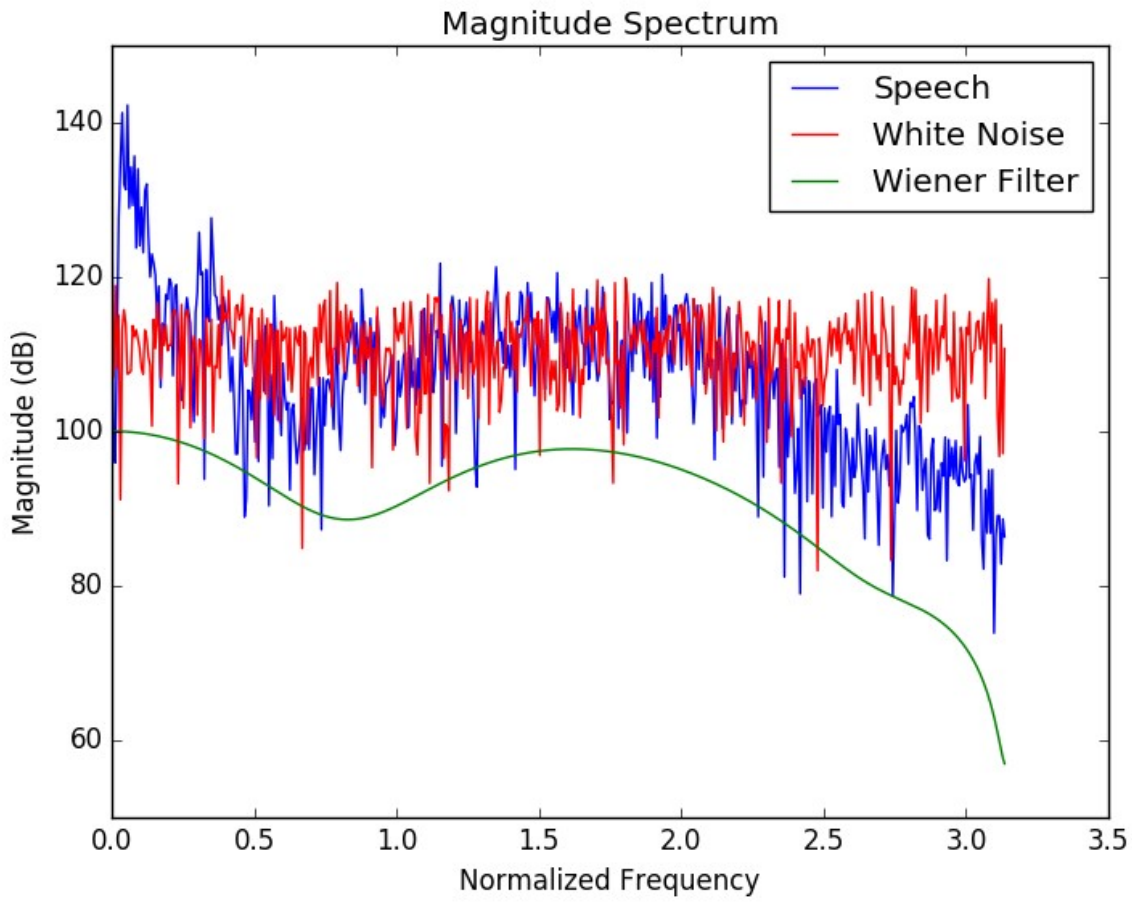


Here we can see that the resulting filter has a somewhat **low pass characteristic**, because our speech signal has energy mostly at low frequencies. At high frequencies we have mostly noise, hence it makes sense to have more attenuation there! This attenuation curve of this Wiener filter also has some similarity to the speech spectrum. If we compare it with the spectrum of our white noise, then we see that at low frequencies the speech is dominating, and at high frequencies noise is dominating. hence we need to remove or attenuate that latter, noisy, part of the spectrum.

We can plot the spectra of the speech and the noise together with (this time with the freqz from the signal

processing library, without the build in plotting):

```
w,Hspeech=sp.freqz(x);
w,Hnoise=sp.freqz(0.1*(random.random(shape(x))-
0.5)*2**15);
w,Hw=sp.freqz(h);
plot(w,20*log10(abs(Hspeech)));
hold
plot(w,20*log10(abs(Hnoise)),'r');
#plot and shift the filter into the vicinity of the
signal:
plot(w,20*log10(abs(Hw))+100,'g');
xlabel('Normalized Frequency')
ylabel('Magnitude (dB)')
legend(('Speech', 'White Noise', 'Wiener Filter'))
title('Magnitude Spectrum')
```



Here we see that **speech dominates the spectrum only at low and middle frequencies**, noise at the other frequencies, hence it makes sense to suppress those noisy frequencies.

Now we can filter it. For “lfilter” function argument we need to convert the matrix type into a 1 dimensional array type:

```
xw = sp.lfilter(array(h.T)[0],[1],array(y.T)[0])
```

#and listen and compare it:
original:

```
sound(array(x), fs)
```

noisy:

```
sound(array(y),fs)
```

Wiener Filtered:

```
sound(xw, fs)
```

We can hear that the signal now sounds more “muffled”, the higher frequencies are indeed attenuated, which reduces the influence of the noise. But it is still a question if it actually “sounds” better to the human ear, because the ear is not looking for the mean squared error solution.
This Wiener filter could now also be applied to **other speech signals**, with **similar frequency characteristics** for signal and noise.

Let's compare the mean (squared) quadratic error (**mse**), to see if it is indeed reduced, and by how much. For the noisy signal it is

```
print shape(x)
#(207612, 1)
#Compute the quadratic error for the first
200000 samples:
sum(power(y[:200000]-x[:200000],2))/200000
# 895724.70095581945
```

For the Wiener filtered signal it is (taking into account 4 samples delay from our filter, beginning to peak).

```
sum(power(xw[4:200004]-x[:200000].T,2))/
200000
# 373727.8735729566
```

We can see that the mean quadratic error is indeed less than half as much as for the noisy version y(n)!

Let's take a look at the matrix $A^T \cdot A$ which we used in the computation,

```
A.T*A
out:
matrix([[  1.08192901e+12,   9.19784413e+11,   8.64233389e+11,
           9.02427205e+11,   8.96487813e+11,   8.52517530e+11,
           8.28117032e+11,   7.98498157e+11,   7.63978129e+11,
           7.41600697e+11],
        [  9.19784413e+11,   1.08192831e+12,   9.19784781e+11,
           8.64234606e+11,   9.02426244e+11,   8.96488076e+11,
           8.52519489e+11,   8.28115493e+11,   7.98496774e+11,
           7.63979130e+11],
```

```
[  8.64233389e+11,   9.19784781e+11,   1.08192869e+12,
   9.19785555e+11,   8.64234241e+11,   9.02427089e+11,
   8.96489010e+11,   8.52519220e+11,   8.28114928e+11,
   7.98496806e+11],
[  9.02427205e+11,   8.64234606e+11,   9.19785555e+11,
   1.08193006e+12,   9.19785087e+11,   8.64236205e+11,
   9.02428512e+11,   8.96489037e+11,   8.52518449e+11,
   8.28114559e+11],...
```

We can see that it is a 10x10 matrix in our example for a Wiener filter with 10 filter taps. In this matrix, the next row looks almost like the previous line, but shifted by 1 sample to the right.

Observe that in general this matrix $\boldsymbol{A^T \cdot A}$ **converges** to the **autocorrelation matrix** of signal $y(n)$ if the length of the signal in the matrix goes to infinity!

$$\boldsymbol{A^T \cdot A \rightarrow R_{yy}} = \begin{bmatrix} r_{yy}(0) & r_{yy}(1) & r_{yy}(2) & \dots \\ r_{yy}(1) & r_{yy}(0) & r_{yy}(1) & \dots \\ \vdots & \vdots & \vdots & \vdots \end{bmatrix}$$

(The autocorrelation of signal y is $r_{yy}(m) = \sum_{n=-\infty}^{\infty} y(n) \cdot y(n+m)$)

Since one row of this matrix is the shifted by one sample version of the one above, it is called a "**Toeplitz Matrix**" (https://en.wikipedia.org/wiki/Toeplitz_matrix).

The expression $\boldsymbol{A^T \cdot x^T}$ in our formulation of the Wiener filter becomes the cross correlation vector

$$\boldsymbol{A}^{\boldsymbol{T}} \cdot \boldsymbol{x}^{\boldsymbol{T}} \rightarrow \boldsymbol{r}_{\boldsymbol{xy}} = \begin{bmatrix} r_{xy}(0) \\ r_{xy}(1) \\ \vdots \end{bmatrix}$$

(where the cross correlation function is $r_{xy}(m) = \sum_{n=-\infty}^{\infty} y(n) \cdot x(n+m)$)

Observe: In the receiver we usually don't have the un-noisy signal x(n), but we can **estimate** the above **cross correlation** function.

Hence our expression for the Wiener filter $\boldsymbol{h}^{\boldsymbol{T}} = \left(\boldsymbol{A}^{\boldsymbol{T}} \cdot \boldsymbol{A}\right)^{-1} \boldsymbol{A}^{\boldsymbol{T}} \cdot \boldsymbol{x}^{\boldsymbol{T}}$ becomes

$$\boldsymbol{h}^{\boldsymbol{T}} = \left(\boldsymbol{R}_{\boldsymbol{yy}}\right)^{-1} \boldsymbol{r}_{\boldsymbol{xy}}$$

This matrix form is also called the "**Yule-Walker** equation", and the general statistical formulation is called the "**Wiener-Hopf** equation".

*See also: *M.H. Hayes, "Statistical Signal Processing and Modelling", Wiley.*

This general statistical formulation now also has the advantage, that we can design a Wiener Filter by just **knowing the statistics**, the **auto-correlation function** of our noisy signal, and the **cross-correlation function** of our noisy and original signal. Observe that this auto-correlation and cross-correlation can also

be obtained from the **power-spectra** (cross-power spectra, the product of the 2 spectra) of the respective signals.
The power spectrum of the noisy signal can usually be measured, since it is the signal to filter, and the **spectrum of the original signal** x(n) usually has to be **estimated** (using assumptions about it).
For instance, we know typical speech spectra. If we want to adapt a Wiener filter in a receiver, we take this typical speech spectrum, and measure the noise level at the receiver. Then we can add the power spectrum of the noise to the power spectrum of the speech to obtain the power spectrum of the noisy speech (which is the power cross spectrum of the clean speech and noise, or the Fourier Transform of cross correlation, because we assume the speech and the noise uncorrelated).
To use Wiener-Hopf, we simply apply the inverse Fourier Transforms to the power spectra.
That is sufficient to compute the Wiener filter coefficients, using above formulations.