Audio /Video Signal Processing
Lecture 7,
Noble Identities, Filters

Gerald Schuller, TU-Ilmenau

Multirate Noble Identities

For multirate systems, the so-called Noble Identities play an important role:

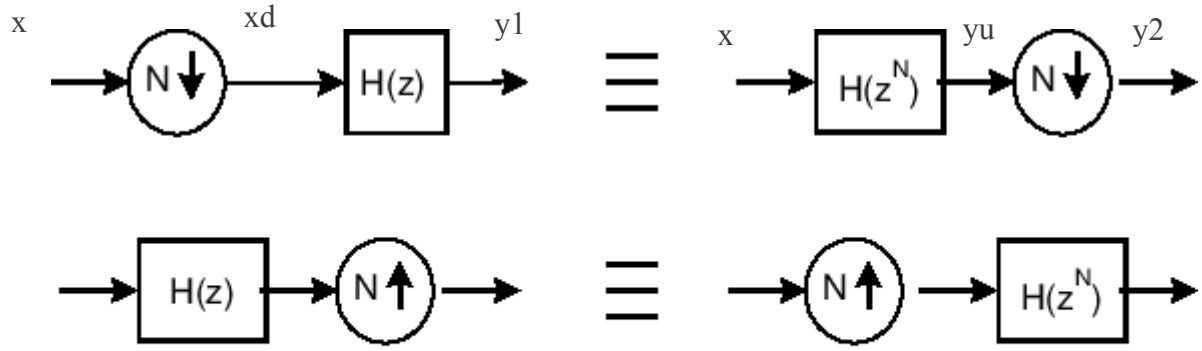
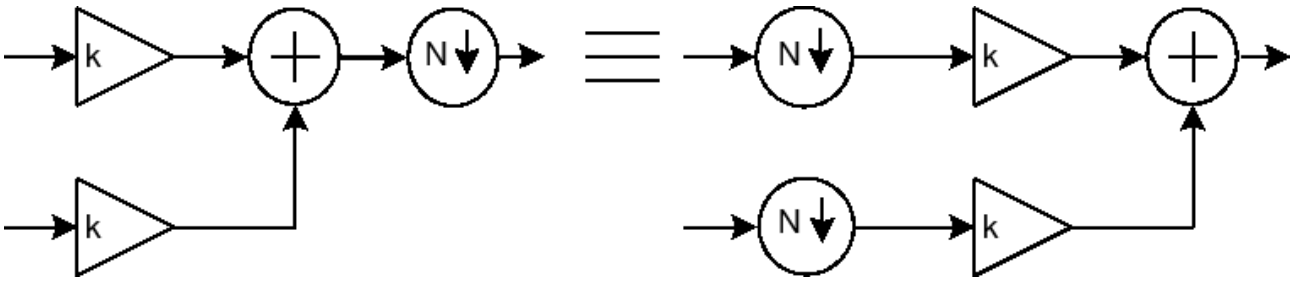


Figure 24.3: Multirate noble identities



from :
https://ccrma.stanford.edu/~jos/sasp/Multirate_Noble_Identities.html
This tells us, in which cases we can exchange down or up-sampling with filtering. This can be done in the case of sparse impulse responses, as can be seen above.

Observe that $H(z^N)$ is the upsampled version of $H(z)$. Remember that the upsampled impulse response has N-1 zeros inserted after each sample of the original impulse response

Example: Take a simple filter, in Matlab or Octave notation: B=[1,1]; (a running average filter), an input signal x=[1,2,3,4,...] or x=1:10, and N=2.

Now we would like to implement the first block diagram of the Noble Identities (the pair on the first line, with outputs y1 and y2). First, for y1, the down-sampling by a factor of N=2:

xd=x(1:N:end)

This yields

xd=1,3,5,7,9

The apply the filter B=[1,1],

y1=filter(B,1,xd)

This yields the sum of each pair in xd:

y1= 1,4,8,12,16

Now we would like to implement the corresponding **right-hand side** block diagram of the noble identity. Our filter is now up-sampled by N=2:

Bu(1:N:4)=B;

This yields

Bu= 1,0,1

Now filter the signal before down-sampling:

yu=filter(Bu,1,x)

This yields

yu= 1, 2, 4, 6, 8, 10, 12, 14, 16, 18

Now down-sample it:

y2=yu(1:N:end)

This yields

y2= 1,4,8,12,16

Here we can now see that indeed y1=y2!

These Noble identities can be used to create efficient systems for sampling rate conversions. In this way we can have filters, which always run on the lower sampling rate, which makes the implementation easier.

It is always possible to rewrite a filter as a sum of up-sampled versions of its phase shifted and down-sampled impulse response, as can be seen in the following decomposition of a filter $H_T(z)$:

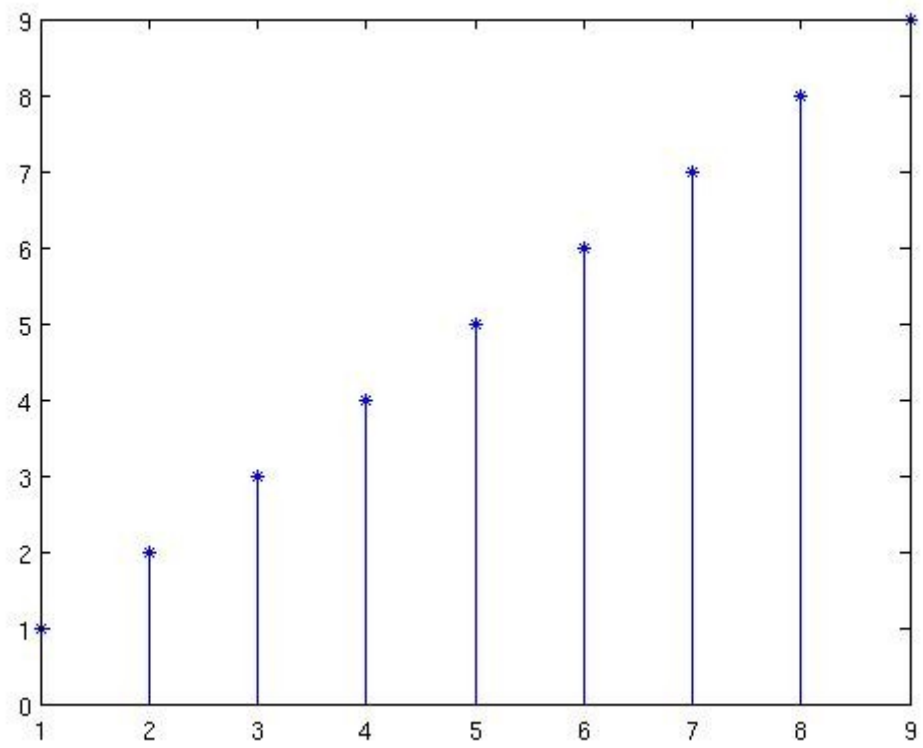
$$H_T(z)=H_0(z^N)+H_1(z^N)\cdot z^{-1}+\ldots+H_{N-1}(z^N)\cdot z^{N-1}$$

Here, $H_0(z^N)$ has all coefficients of positions at integer multiples of N, mN,

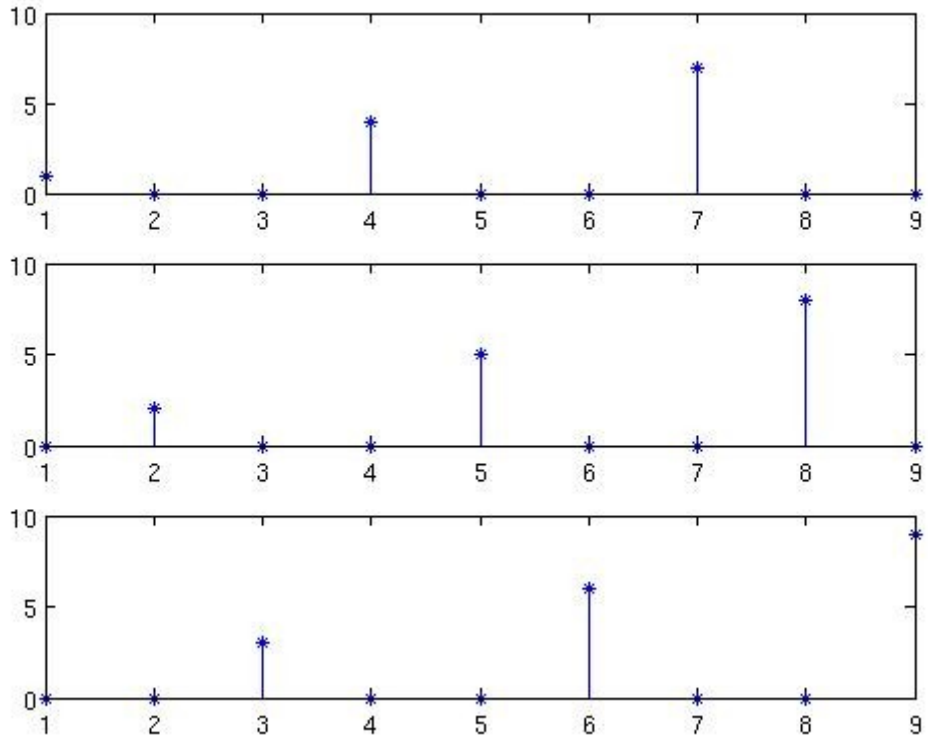
$H_1(z^N)$ contains the coefficients at positions mN+1,

and in general $H_i(z^N)$ contains the coefficients at positions mN+i. Since i can be seen as different “phases” of our impulse response, the components $H_i(z)$ are also called “polyphase

components" or "polyphase filters" for $H_i(z)$. This is illustrated in the following pictures. First is a simple time sequence,



This sequence, for a sampling factor of N=3, can then be decomposed in the following 3 up-sampled polyphase components:



The upper plot corresponds to $H_0(z^N)$, the middle plot is $z^{-1} \cdot H_1(z^N)$, and the lower plot is $z^{-2} \cdot H_2(z^N)$.

Example:

The impulse response or our filter is

$$h_T = [1,2,3,4,...]$$

then its z-Transform is

$$H_T(z) = 1 + 2z^{-1} + 3z^{-2} + 4z^{-3} + ...$$

Assume N=2. Then we obtain its (up-sampled) polyphase components as

$$H_0(z^2) = 1 + 3z^{-2} + 5z^{-4} + ...$$

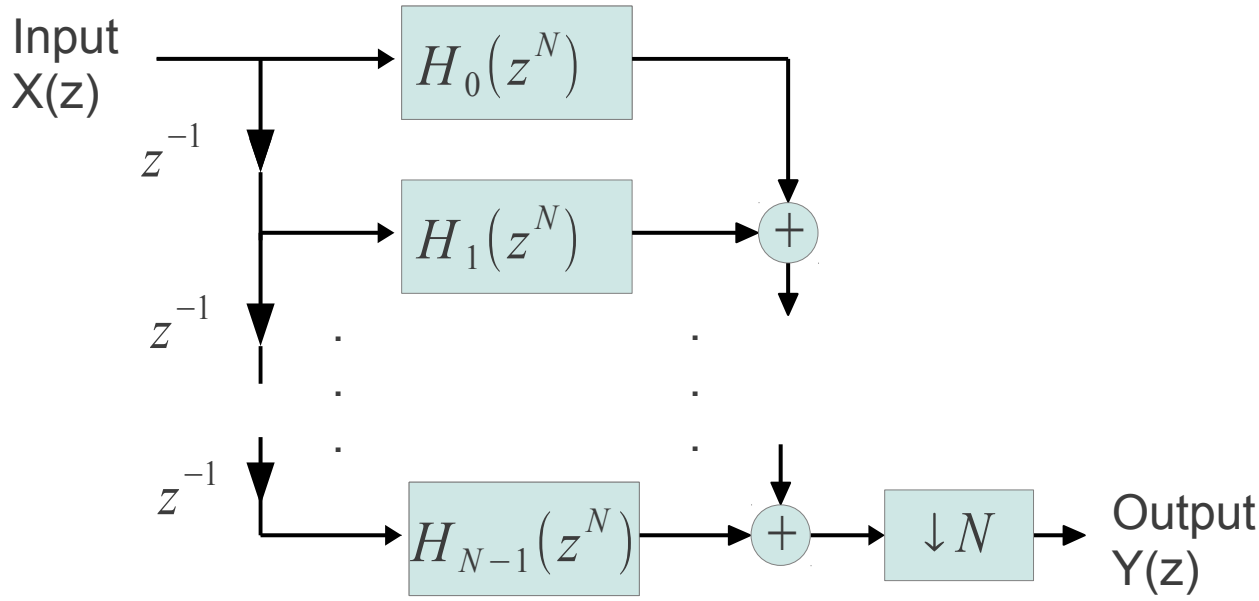
$$H_1(z^2) = 2 + 4z^{-2} + 6z^{-4} + ...$$

Hence we can combine the total filter from its

polyphase components,

$$H_T(z) = H_0(z^2) + z^{-1} H_1(z^2)$$

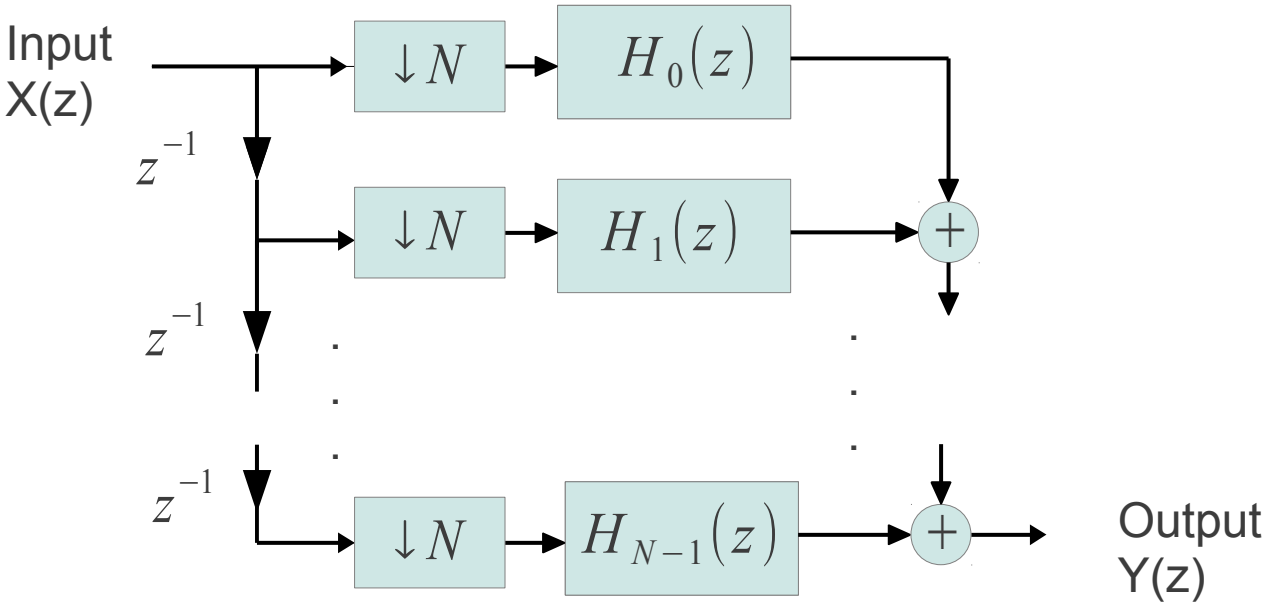
The general case is illustrated in the following block diagram, which consists of a delay chain on the left to implement the different delays z^{-i}, and the polyphase components $H_i(z^N)$ of the filter:



This could be a system for down-sampling rate conversion, where we first low-pass filter the signal x to avoid aliasing, and then down-sample it.

In this way we can decompose our filter in N polyphase components, where i is the “phase” index.
Now we can simplify this system by using the Noble Identities.
Because in this sum we have transfer functions of the form $H_i(z^N)$, we can use the Noble

identities to simplify our sampling rate conversion, to shift the down-samplers before the sum and before the filters (but not before the delay chain on the left side), with replacing the polyphase filters arguments z^N with z :



Looking at the delay chain and the following down-samplers, we see that this corresponds to "blocking" the signal x into consecutive blocks of size N. Hence we now have a block-wise processing with our filter, and the filtering is now completely done at the lower sampling rate, which reduces speed requirements for the hardware. We obtained a parallel processing at the lower sampling rate.

Example:
Down-sample an audio signal. First read in the audio signal into the variable x,

x=wavread('speech8kHz.wav');
Listen to it as a comparison:
sound(x,8000);
(If you are using Octave, you might first have to install the program 'sox' on your system for the sound output).
Take a low pass FIR filter with impulse response h=[0.5 1 1 0.5] and a down-sampling factor N=2. Hence we get the z-transform or the impulse response as
$H(z)=0.5+1\cdot z^{-1}+1\cdot z^{-2}+0.5\cdot z^{-3}$ and its polyphase components as
$H_0(z)=0.5+1\cdot z^{-1}$, $H_1(z)=1+0.5\,z^{-1}$
in the time domain (in Matlab or Octave)
h0=[0.5 1] and h1=[1 0.5]
Produce the 2 phases of a down-sampled input signal x:
x0=x(1:2:end); x1=x(2:2:end);
then the filtered and down-sampled output y is
y=filter(h0,1,x0)+filter(h1,1,x1);
Observe that each of these 2 filters now works on a down-sampled signal, but the result is identical to first filtering and then down-sampling.
Now listen to the resulting down-sampled signal:
sound(y,4000);

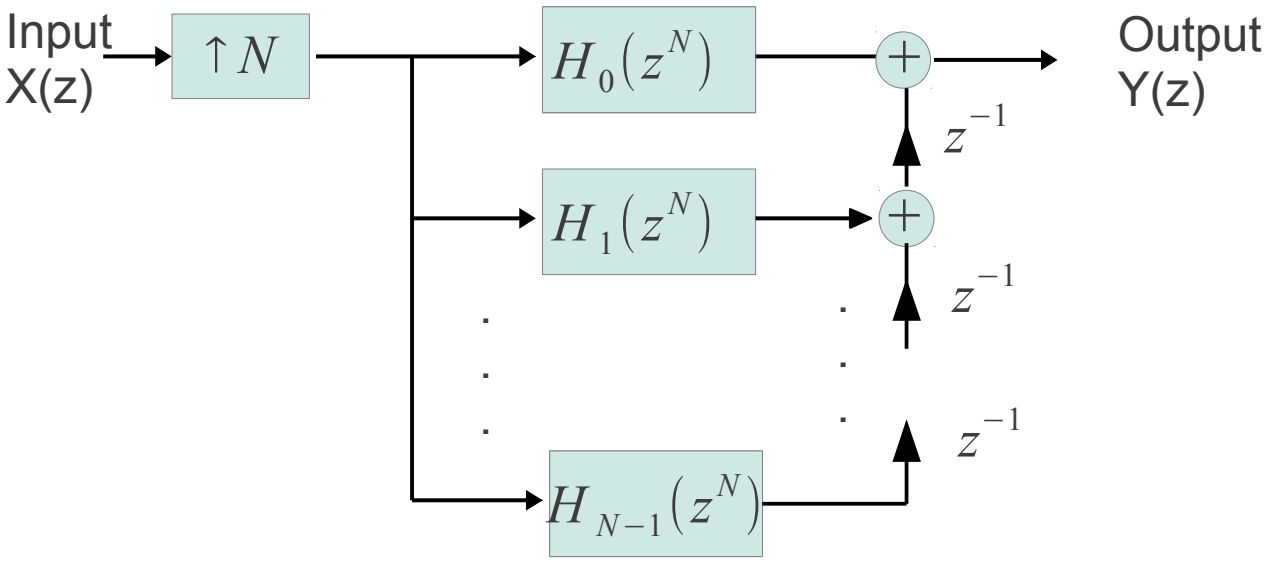
Correspondingly, **up-samplers** can be obtained with filters operating on the lower

sampling rate.

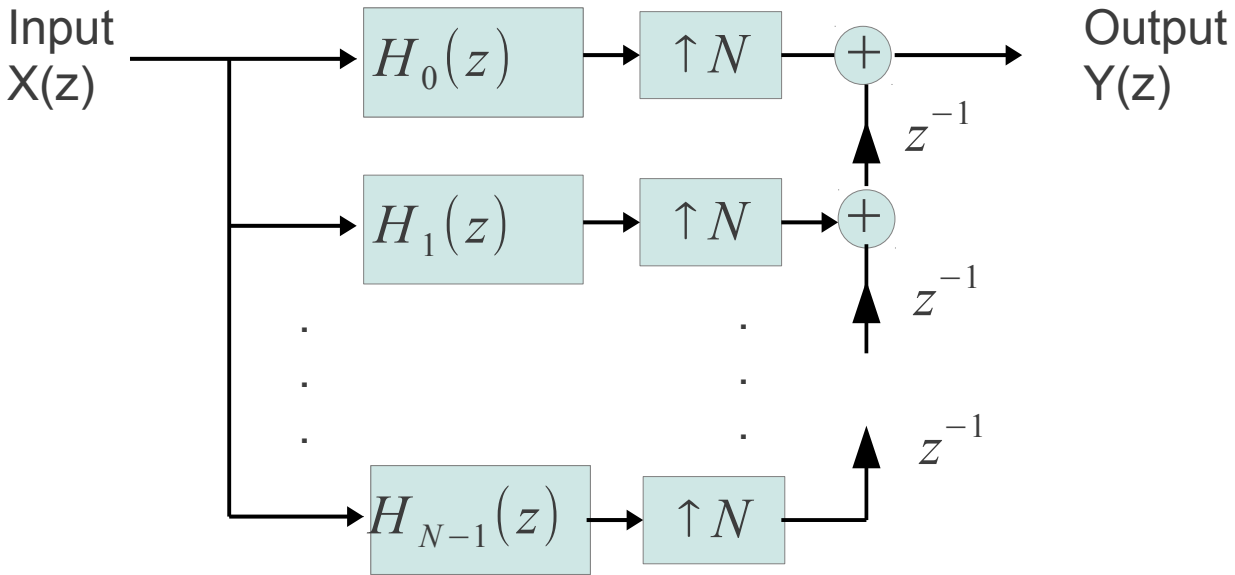
Since $H_i(z^N)$ and z^{-i} are linear time-invariant systems, we can exchange their ordering,

$$H_i(z^N)\cdot z^{-i} = z^{-i}\cdot H_i(z^N)$$

Hence we can redraw the polyphase decomposition for an up-sampler followed by a (e.g. low pass) filter as follows,



Using the Noble Identities, we can now shift the up-sampler to the right, behind the polyphase filters (again with changing their arguments from z^N to z) and before the delay chain,



Again, this leads to a parallel processing, with N filters working in parallel at the lower sampling rate. The structure on the right with the up-sampler and the delay chain can be seen as a de-blocking operation. Each time the up-sampler let a complete block through, it is given to the delay chain. In the next time-steps the up-samplers stop letting through, and the block is shifted through the delay chain as a sequence of samples.

Example (Matlab or Octave):
up-sample the signal x by a factor of N=2 and low-pass filter it with the filter h=[0.5 1 1 0.5]; as in the previous example. Again we obtain the filters polyphase components as h0=[0.5 1] and h1=[1 0.5]
Now we can use these polyphase components to filter at the lower sampling rate to obtain the polyphase components of the filtered and upsampled signal y0 and y1,
y0=filter(h0,1,x), y1=filter(h1,1,x);
The complete up-sampled signal is then

obtained from its 2 polyphase components,
L=max(size(x));
y(1:2:(2*L))=y0; y(2:2:(2*L))=y1;
Where now the signal y is the same as if we had first up-sampled and then filtered the signal!
Now listen to the up-sampled signal:
sound(y,16000);

Filter Design

How do we design filters, such that they have desired properties? How could we improve, for example, our filters for the down- and up-sampling application?
First we should know how the ideal or desired filter should look like. In general, we specify the frequency responses with magnitude and phase,

$$H(\Omega)=e^{j\phi(\Omega)}\cdot A(\Omega)$$

where $\phi(\Omega)$ is our phase (in dependence of our normalized frequency Ω), and $A(\Omega)$ is our magnitude. If we want to design our filter, we need to specify both, phase and magnitude.

The phase of our system is connected to the delay of our system. Imagine, we have a pure delay of d samples, resulting in a transfer

function $H(z)=z^{-d}$, then we get the frequency response $H(e^{j\Omega})=e^{-j\Omega\cdot d}$. Hence our phase response is $\phi(\Omega)=-\Omega\cdot d$. This is also what we call a “linear phase”, because the phase is linearly dependent on the frequency. We can now also see that the (negative) slope of the phase curve corresponds to the delay d. Since the slope in this case is the same as the derivative of the phase to frequency, we can also say that this is the delay. In case this derivative is different for different frequencies, we call it the “group delay”, which in general is dependent on the frequency Ω . If we call the group delay $d_g(\Omega)$, then it is defined as

$$d_g(\Omega)=\frac{-d\,\phi(\Omega)}{d\,\Omega}$$

This means that the phase tells you, how much delay each frequency group has through the system. When we design a system, this is what we need to keep in mind.
If delay is not important, often some **constant delay** d is chosen, which then leads to a system with a linear phase

$$\phi(\Omega)=-\Omega\cdot d$$

This then leads to linear phase filters.

Now we also need to think about the desired magnitude $A(\Omega)$ of our filter. Often, one or multiple pass bands are desirable, together

with one or multiple stop bands. To do that we need to decide where the band edge of the are, in normalized frequencies. Since we have no ideal filters, we need to give the system transition bands between the pass bands and the stop bands, to give the filter space to come from one state (passing a signal) to another state (stopping a signal).

Example: We would like to have a low pass filter. We would like to have a pass band from frequency 0 up to frequency $\pi/2$, a half band filter. If we want to use it for sampling rate conversion with a factor of 2, we actually need to make sure that the stop band starts at $\pi/2$. We also need to create a transition band, for instance going from $\pi/2-0.1$ to $\pi/2$. This transition bandwidth is where we don't care about that value of our frequency response, and it is something that we can use to fine-tune the resulting filter. For instance if we see we don't get enough attenuation, we can increase the bandwidth of our transition band.
This also means that our passband is given as between 0 and $\pi/2-0.1$.

This is now our ideal, but what we can obtain is never exactly this ideal. We can only come close to it in some sense. That is why we need to define an error measure or an error function, which measures how close we come to the

ideal, and which we can use to obtain a design which minimizes this error function.

Often used error functions are the mean squared error, the weighted mean squared error, or the minimax error function (which seeks to minimize the maximum deviation to the ideal).

The weighted mean squared error uses weights to give errors in different frequency regions different importance. For instance, the error in the stop-band is often more important than in the pass-band, to obtain a high attenuation in the stop-band. An error of 0.1 in the pass-band might not be so bad, but an error of 0.1 in the stop-band leads to only 20 dB attenuation, which is not very much. So in this case, we might assign a weight of 1 to the pass bands, and a weight of 1000 to the stop bands, to obtain higher stop band attenuations.

Given this information, we can now use a numerical optimization routine to minimize a given error function and a given filter structure, such that the filter coefficients minimize this error.
In Matlab, the function “fminunc” or in Octave the function “sqp” can be used.
To compute the frequency response of a given filter, we can again use “freqz”, which also has

an argument for the number of frequency samples, and which can also be used to make an assignment to a variable, for instance for an impulse response h and 1024 samples in the frequency domain we get:
H=freqz(h,1,1024);
where we now have the complex frequency response in H, a vector of 1024 samples.

Example:
We would like to design a half band low pass filter, with band edges as above, and 1024 samples in the frequency domain, ranging from normalized frequency 0 to π . First we define a vector with the ideal frequency response and for the weights. In this vector, the number of frequency samples that belong to the pass band is (in Matlab or Octave)
pb=round((pi/2-0.1)/pi*1024);
The number of samples for the transition band is
tb=round(0.1/pi*1024);
and for the stop band we obtain
sb=round(pi/2/pi*1024);
The vector with the desired magnitude of the frequency response then becomes
Ades=[ones(1,pb),zeros(1,tb+sb)];
The weight vector becomes
W=[ones(1,pb),zeros(1,tb),1000*ones(1,sb)];
Assume we would like to design an FIR filter with 8 taps (coefficients) and with linear phase.

This should lead to a symmetric filter, with a constant delay corresponding to the center of the filter at d=(8-1)/2=3.5;
Hence we can set the desired phase of our frequency response to
phi=-d*(0:1023)/1024*pi;
Then we obtain our desired frequency response as
Hdes=exp(i*phi).*Ades;
We can now start our optimization with a random set of coefficients for the filter:
h=rand(1,8);
and our error function becomes the Matlab/Octave function:

```
function err=optimfunc(h);
%produces the weighted squared error  for an FIR filter with coefficients in column vector h

%make it a row vector:
h=h';
pb=round((pi/2-0.1)/pi*1024);
tb=round(0.1/pi*1024);
sb=round(pi/2/pi*1024);
Ades=[ones(1,pb),zeros(1,tb+sb)];
W=[ones(1,pb),zeros(1,tb),100*ones(1,sb)];
d=3.5;
phi=-d*(0:1023)/1024*pi;
%Desired frequency response with magnitude and phase:
Hdes=exp(i*phi).*Ades;
%Compute frequency response for h, transpose to make it
```

```
a row vector:
H=freqz(h,1,1024)';
%Compute weighted squared error, abs instead of using
conjugate complex for quadratic distance:
err= sum(abs(H-Hdes).^2 .* W);
```

This is stored in the file “optimfunc.m”.
In Matlab the optimization is started with
hmin=fminunc('optimfunc', rand(8,1));
in Octave it is
hmin=sqp(rand(1,8)','optimfunc');

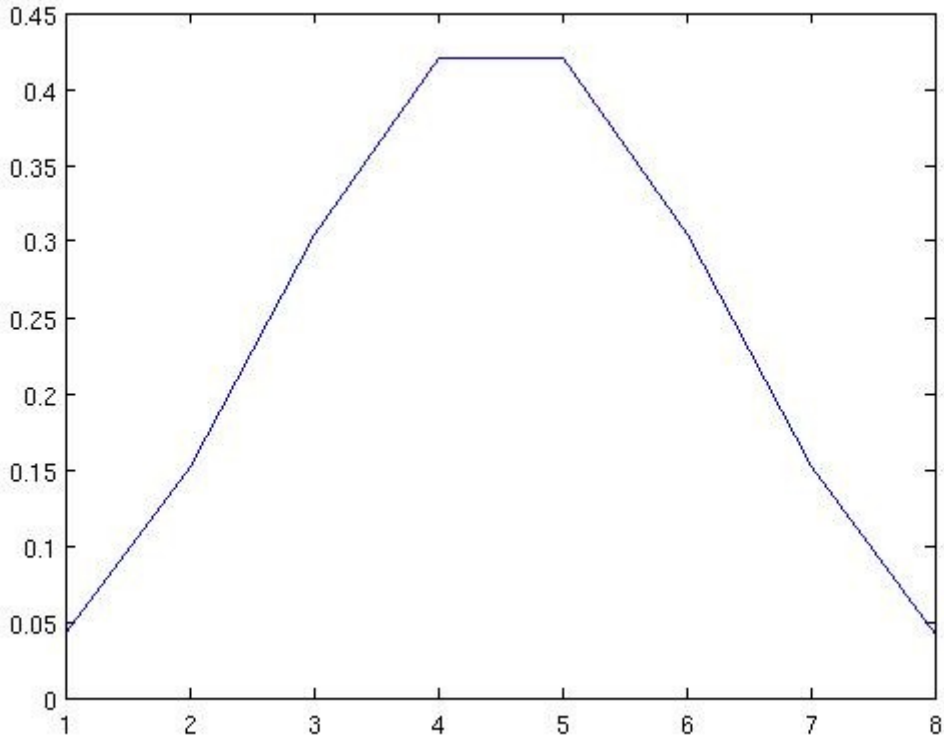
This approach has the advantage, that all kind of filters can be optimized, FIR and also IIR, with any kind of magnitude and phase behaviour, so that it is most universal.

But since these are general purpose optimization functions, they might not lead to the optimum solution. For linear phase filters, Matlab and Octave has a specialized optimization in the function “firpm”, or “remez” which implements the so-called Parks-McLellan algorithm (see also the Book: Oppenheim, Schafer: “Discrete-Time Signal Processing”, Prentice Hall) .
This is now also an example of the **minimax error function**. The algorithm minimizes the maximum error in the pass band and the stop band (weighted in comparison between the two), which leads to a so-called equi-ripple

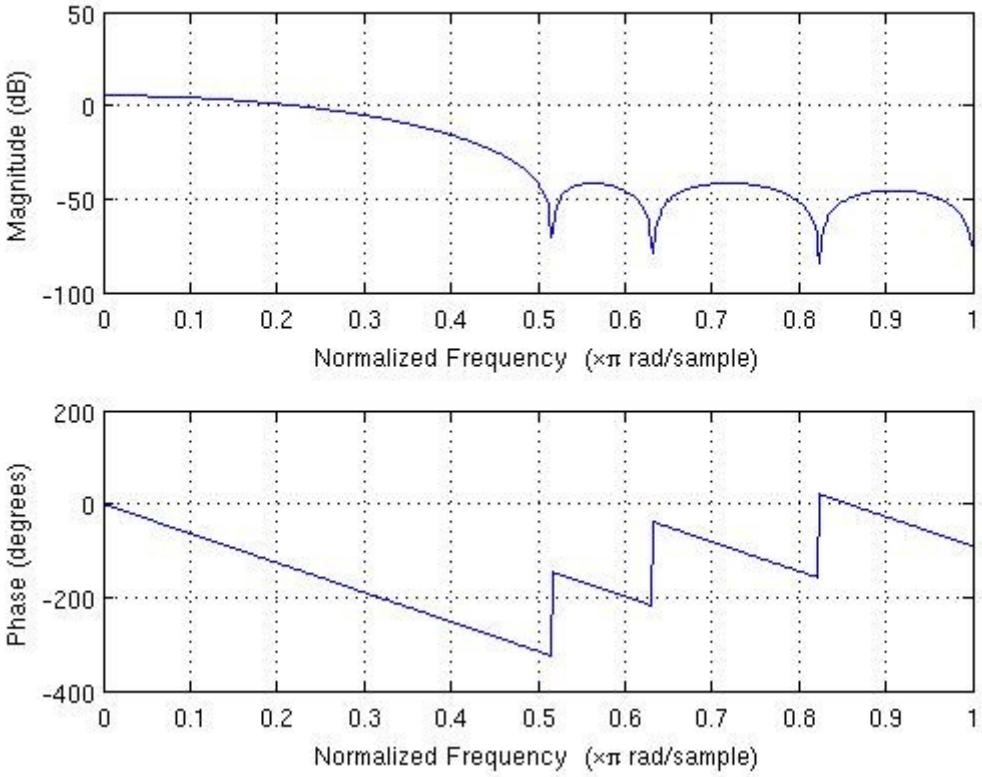
behaviour (all ripples have the same hight) of the filter in the frequency domain.

It is called in the form
 hmin=firpm(N,F,A,W);
or
 hmin=remez(N,F,A,W);
where N is the length of the filter minus 1, F is the vector containing the band edges (now normalized to the Nyquist frequency, between 0 and 1) of the pass band, transition band, and stop band,
A is the desired amplitude vector at the specified band edges, and W is the weight vector for the bands.
In our example we get:
N=7;
F=[0 1/2-0.1 1/2 1];
A=[1 1 0 0];
W= [1 100];
and we call:
hmin=firpm(N,F,A,W);
or
hmin=remez(N,F,A,W);

Now we obtain a nice impulse response or set of coefficients hmin:



and its frequencies response freqz(hmin) is



Here we see that we obtain about 40 dB of stop band attenuation, which roughly corresponds to our weight of 100 for the stop band.

Remark for Linux and Octave:

The filter design tools, in this case the remez function, are in the “Signal” toolbox. If the installation in qtOctave is not working properly, using “config-Install Octave packages”, then in Linux install “octave3.2-headers” from the Software Manager, and edit (with a terminal window):

```
sudo gedit /etc/octave3.x.conf
```

insert a # at the beginning of the line with "pkg prefix" to comment out this line, and save the file. Restart qtOctave. Then all new packages will be automatically placed in the directory ~/octave.

In Octave, click on “Config - Install Octave packages”, to download a package.

Then, in the Octave command window, type

```
pkg install pkg_name
```

(replace *pkg_name* with the name of the package). For instance

```
pkg install /tmp/qopm_packages/miscellaneous-1.0.10.tar.gz
```

In this way, first install the toolboxes “Miscellaneous”, “Struct”, “Specfun”, and “Optim”, on which the Signal toolbox depends. Next, the “Signal” toolbox can be successfully installed.

For the playback of sound (with the function “sound”), the Octave package “Audio” needs to be installed, and also the software package “sox” from the software package manager

under Linux.