

Digital Signal Processing 2/ Advanced Digital Signal Processing, Audio/Video Signal Processing

Lecture 8, Noble Identities, Filters

Gerald Schuller, TU Ilmenau

Filter Design

How do we design filters, such that they have desired properties?

First we should know how the **ideal or desired filter** should look like. In general, we specify the **frequency responses** with magnitude and phase,

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

where $\varphi(\Omega)$ 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 $\varphi(\Omega) = -\Omega \cdot d$. This is a **linear function** in Ω , with a **slope of $-d$** , and hence we also call it "**linear phase**", because the

phase is linearly dependent on the frequency. We have a **fixed delay** d for all frequencies. 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. We call the negative derivative to Ω 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

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

which is an important definition.

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

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

This then leads to **linear phase filters**.

This is important for instance for image processing. Edges contain many different frequencies, and if we had different group delays for different frequencies, edges would “dissolve” and appear unsharp.

Now we also need to think about the desired **magnitude** $A(\Omega)$ of our filter. Often, one or multiple **pass bands** are desirable, where the signal is passed and hence has close to 0 dB attenuation, together with one or multiple **stop bands**, at which frequencies the signal is “stopped”, meaning it has strong attenuation. To do that we need to decide where the band edges 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). This means, there need to be gaps between the pass bands and the stop bands.

Example: We would like to have a half band **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 downsampling 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, how “good” the filter is, and which we can use to obtain a design which minimizes this error function.

Often used error functions are the mean squared error, the mean absolute 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. (we optimize in the

linear domain and not in the dB domain). 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. The optimization usually results in the same or similar **weighted error** for all frequencies. For instance we get:

$\text{error}_{\text{stopband}} \cdot 1000 = \text{error}_{\text{passband}} \cdot 1$, and hence $\text{error}_{\text{stopband}} = \text{error}_{\text{passband}} / 1000$.

For **FIR linear phase filters**, Matlab and Octave have a specialized **optimization** in the function **"firpm"**, or **"remez"** which implements the so-called **Parks-McLellan algorithm**, using the **Chebyshev algorithm** (see also the Book: Oppenheim, Schaffer: "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 height in the same band, e.g. stop band or pass band) **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];
```

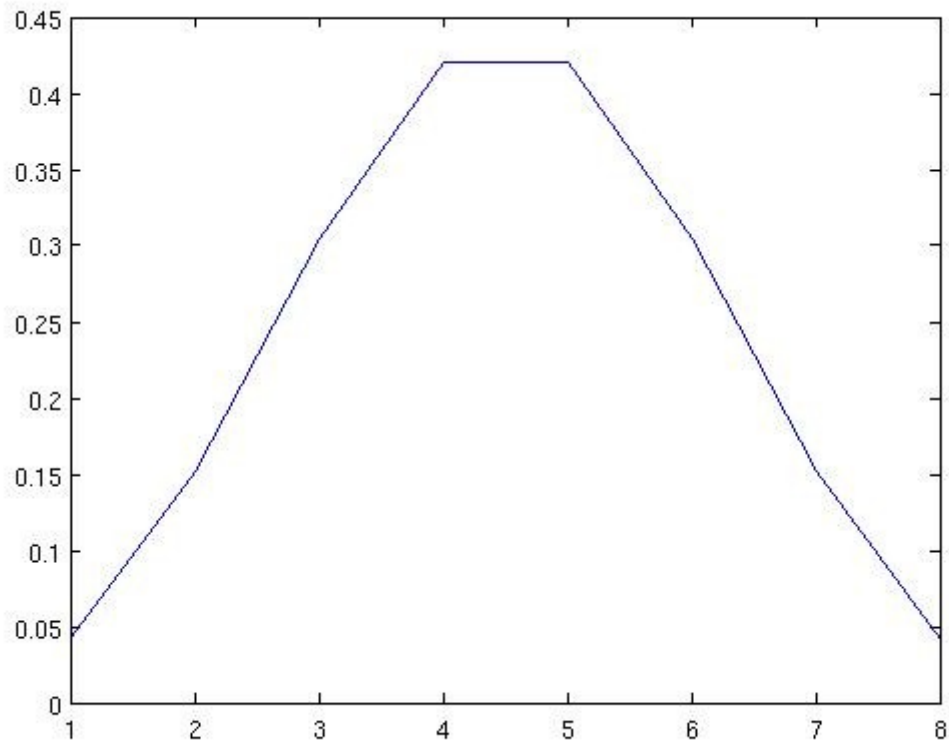
%and we call:

```
hmin=firpm(N,F,A,W);
```

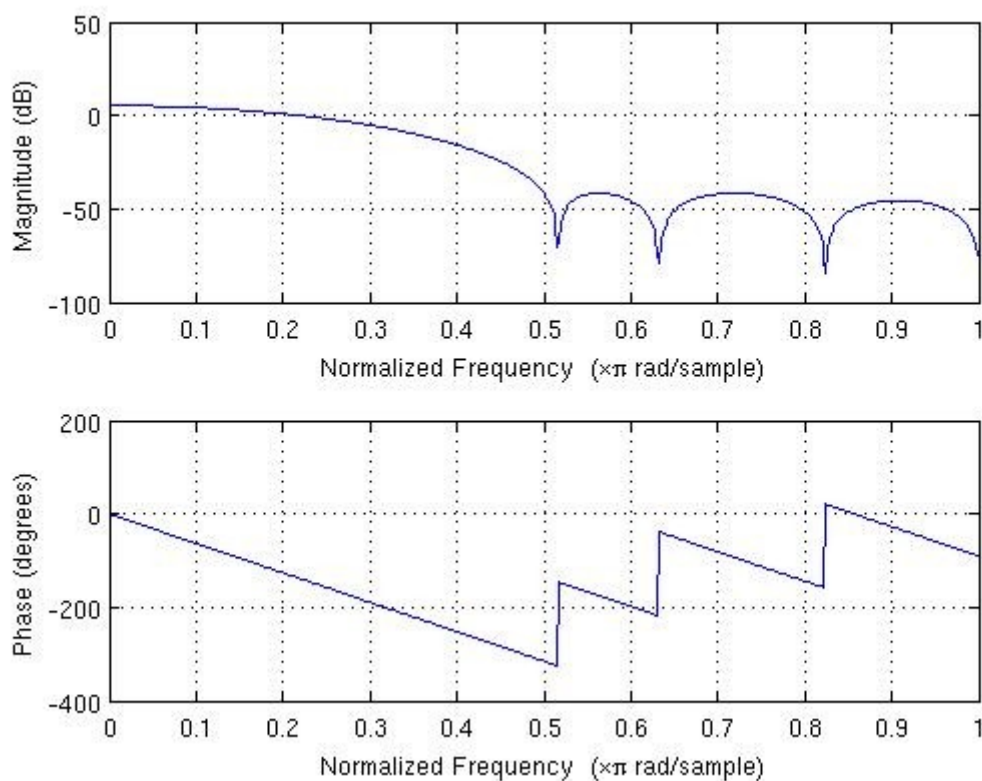
or

```
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 or package. You simply install “`octave-signal`” from the software manager.

For the playback of sound (with the function “`sound`”), you also need to install “`octave-audio`” and “`sox`” from the software manager.

Python Example

In Python we have similar functions in the library “`scipy.signal`”. For instance `remez`: `scipy.signal.remez(numtaps, bands, desired, weight=None)`.

Observe: in `scipy.signal.remez`, by default the Nyquist frequency is 0.5 and the sampling frequency is 1!

Example: We have **8000 Hz sampling rate**, and want to build a band pass filter. Our low stop band is between 0 and 0.05, our pass band between 0.1 and 0.2, and high stop band between 0.3 and 0.5. Since here, 1 corresponds

to the sampling frequency, our **pass band** will be between $0.1 \times 8000 = \mathbf{800\text{Hz}}$ and $0.2 \times 8000 = \mathbf{1600\text{ Hz}}$.

Hence our vector *bands* is
[0.0, 0.05, 0.1, 0.2, 0.3, 0.5]

The vector desired contains the desired output *per band* (not per band edge as in Octave/Matlab). Hence here for our bandpass filter it is:

[0.0, 1.0, 0.0]

We choose our weights all to 1:

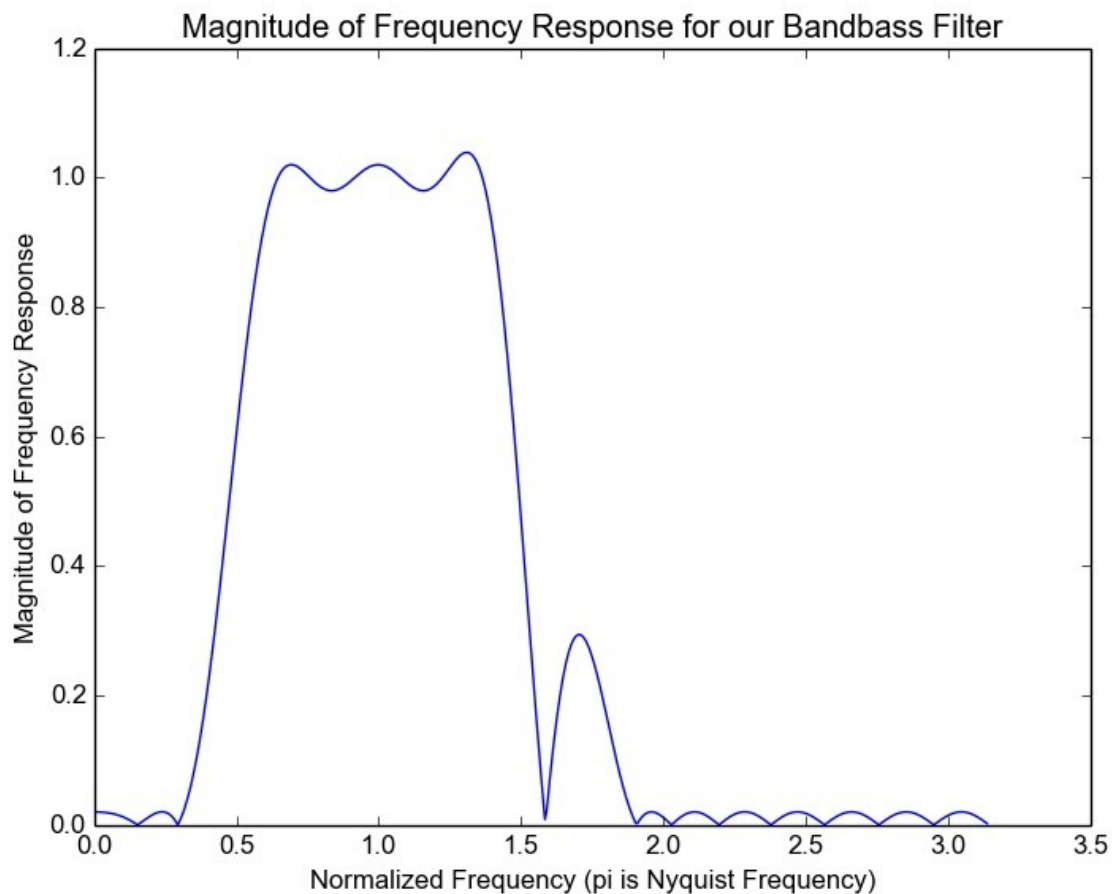
weight=[1.0, 1.0, 1.0]

and our *numtaps* to be 32.

Hence our design function in Python is:

```
import numpy as np
import scipy.signal
import matplotlib.pyplot as plt
N=32
bpass=scipy.signal.remez(N, [0.0, 0.05, 0.1,
0.2, 0.3, 0.5] , [0.0, 1.0, 0.0],
weight=[1.0, 1.0, 1.0])
#Plot the magnitude of the frequency
response:
fig = plt.figure()
[freq, response] = scipy.signal.freqz(bpass)
plt.plot(freq, np.abs(response))
plt.xlabel('Normalized Frequency (pi is
Nyquist Frequency)')
plt.ylabel("Magnitude of Frequency
Response")
```

```
plt.title("Magnitude of Frequency Response  
for our Bandpass Filter")  
fig.show()
```



Observe: The equi-ripple behaviour inside each band is clearly visible, and we see our pass band a little left of the center. The side lobe to its right is from the transition band there.

Now try it on life audio with out python script:

```
python pyrecplay_filterblock.py
```

Observe: Speech sounds like through a very cheap telephone, since only a small band is left of it.

Multirate Noble Identities

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

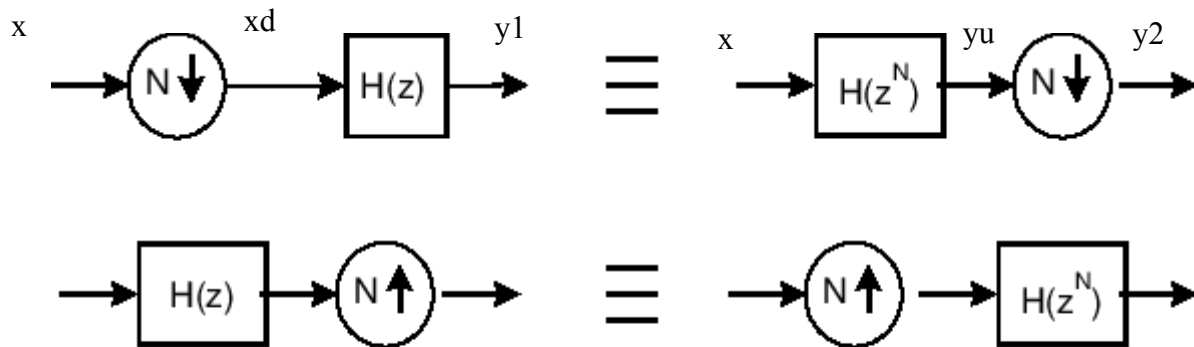
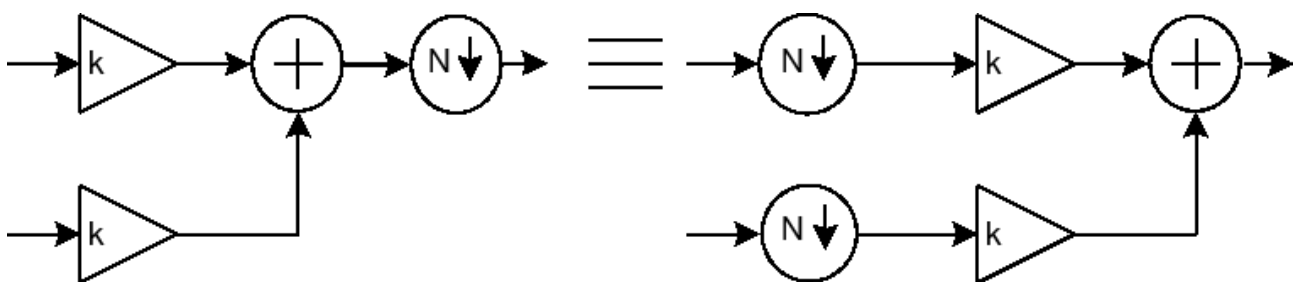


Figure 24.3: Multirate noble identities



from :

https://ccrma.stanford.edu/~jos/sasp/Multirate_Noble_Identities.html

The symbol $\downarrow N$ means: Downsampling by a factor of N by keeping only every N 'th sample.

The symbol $\uparrow N$ means: upsampling the sequence by inserting $N-1$ zeros after each sample.

Example: The filter impulse response shall be

$h=[1,2,3]$, hence its z-transform is

$$H(z)=1+z^{-1}\cdot 2+z^{-2}\cdot 3 .$$

For $N=2$ the upsampled version is

$H(z^2)=1+z^{-2}\cdot 2+z^{-4}\cdot 3$, which corresponds to the upsampled, (“sparse” because of the zeros in it), impulse response

$$h_u=[1,0,2,0,3] .$$

The Noble Identities tell 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.

Observe: This upsampled filter $H(z^N)$ is in most applications a **useless** filter, because we not only have 1 passband, but also get $N-1$ aliased versions for a total of N passbands! But in most applications we want to have only 1 passband. We will make it useful later.

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

Now we would like to implement the first block diagram of the Noble Identities, the down-sampling (the pair on the first line, with outputs y_1 and y_2). First, for y_1 , the down-sampling by a factor of $N=2$:

$$x_d = x(1:N:end)$$

This yields

$$x_d = 1, 3, 5, 7, 9$$

Then apply the filter $B=[1,1]$,

$$y_1 = \text{filter}(B, 1, x_d)$$

This yields the sum of each pair in x_d :

$$y_1 = 1, 4, 8, 12, 16$$

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

$$B_u(1:N:4) = B;$$

This yields

$$B_u = 1, 0, 1$$

Now filter the signal before down-sampling:

$$y_u = \text{filter}(B_u, 1, x)$$

This yields

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

Now down-sample it:

$$y_2 = y_u(1:N:end)$$

This yields

$$y_2 = 1, 4, 8, 12, 16$$

Here we can now see that indeed $y_1 = y_2$!

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 (Remember: so far we always did the filtering at the **higher** sampling rate).

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 $H_T(z)$. In this way we can make out of our previously **useless** filter **useful filters**, by **combining** several of our upsampled versions into a new useful filter $H_T(z)$:

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

Here, $H_0(z^N)$ has all coefficients of positions at integer multiples of N , mN , at phase 0,

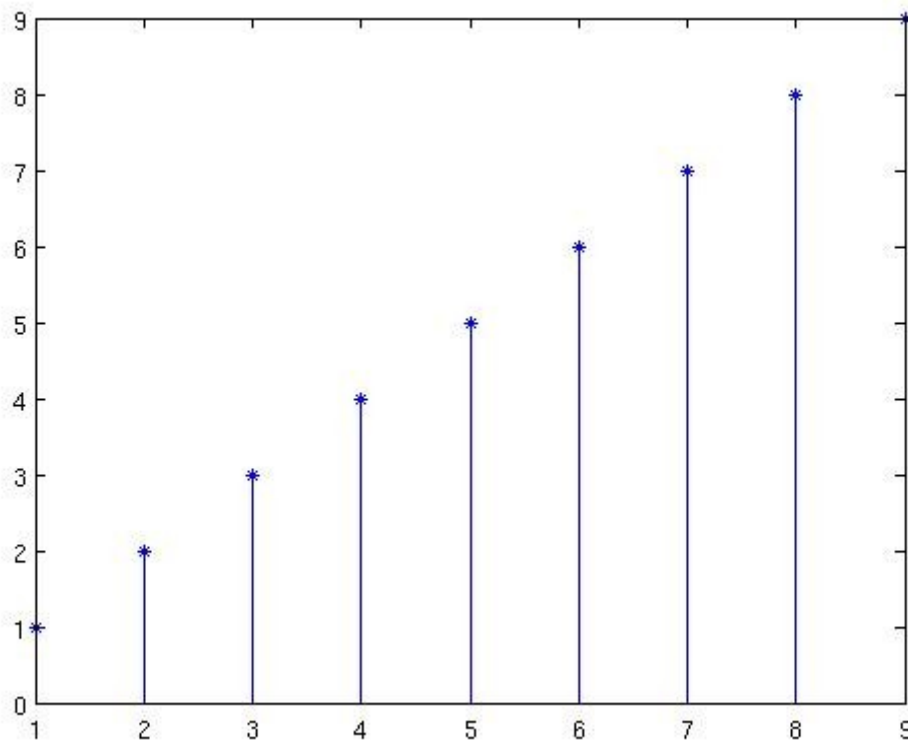
$H_1(z^N)$ contains the coefficients at positions $mN+1$, at phase 1, and in general $H_i(z^N)$ contains the coefficients at positions $mN+i$, at phase i . This means:

$H_i(z)$ is the z-transform of $h(mN+i)$

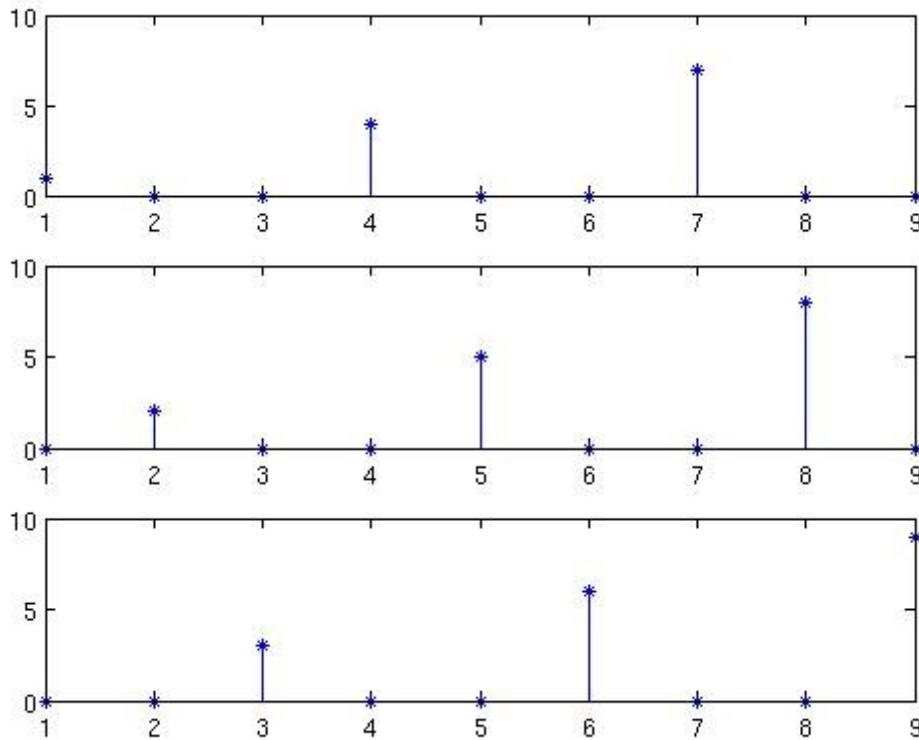
Since i can be seen as many different “phases” of our impulse response, the components

$H_i(z)$ are also called “**polyphase components**” or “**polyphase elements**”.

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 $z^{-2} \cdot H_2(z^N)$.

The same can be done for our signal $x(n)$. Our **polyphase component** $X_i(z)$ is the z-transform of $x(mN+i)$.

Example:

The impulse response of our filter is

$$h_T = [1, 2, 3, 4, \dots]$$

then its z-Transform is

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

Assume $N=2$. Then we obtain its (up-sampled)

polyphase components as

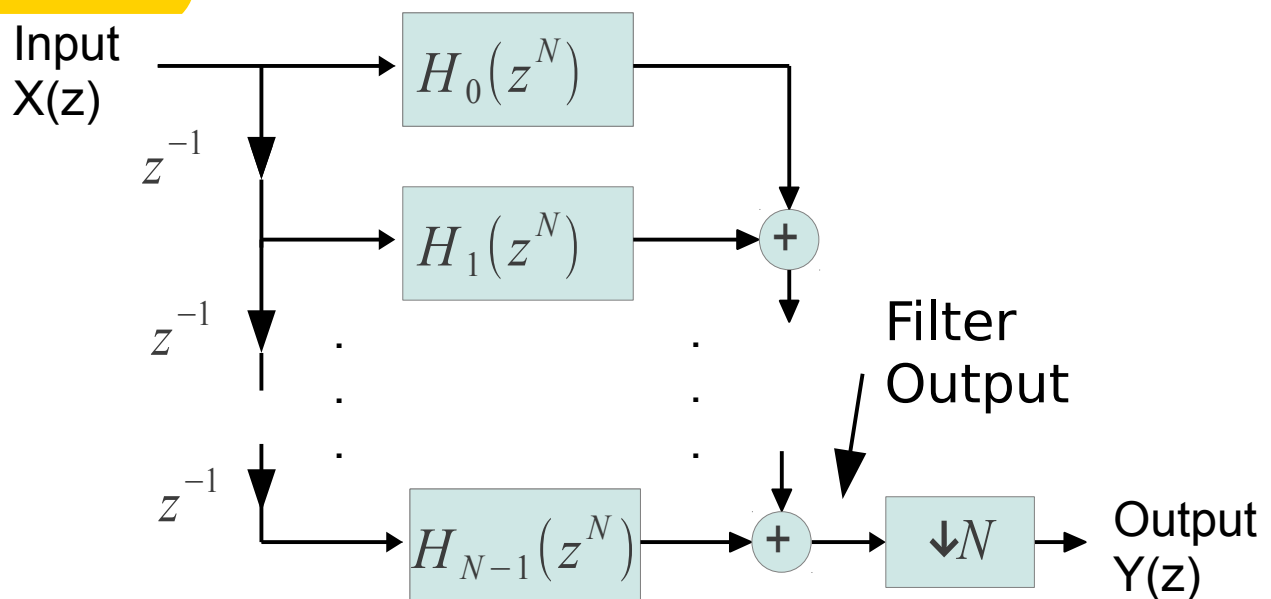
$$H_0(z^2) = 1 + 3z^{-2} + 5z^{-4} + \dots$$

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

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:



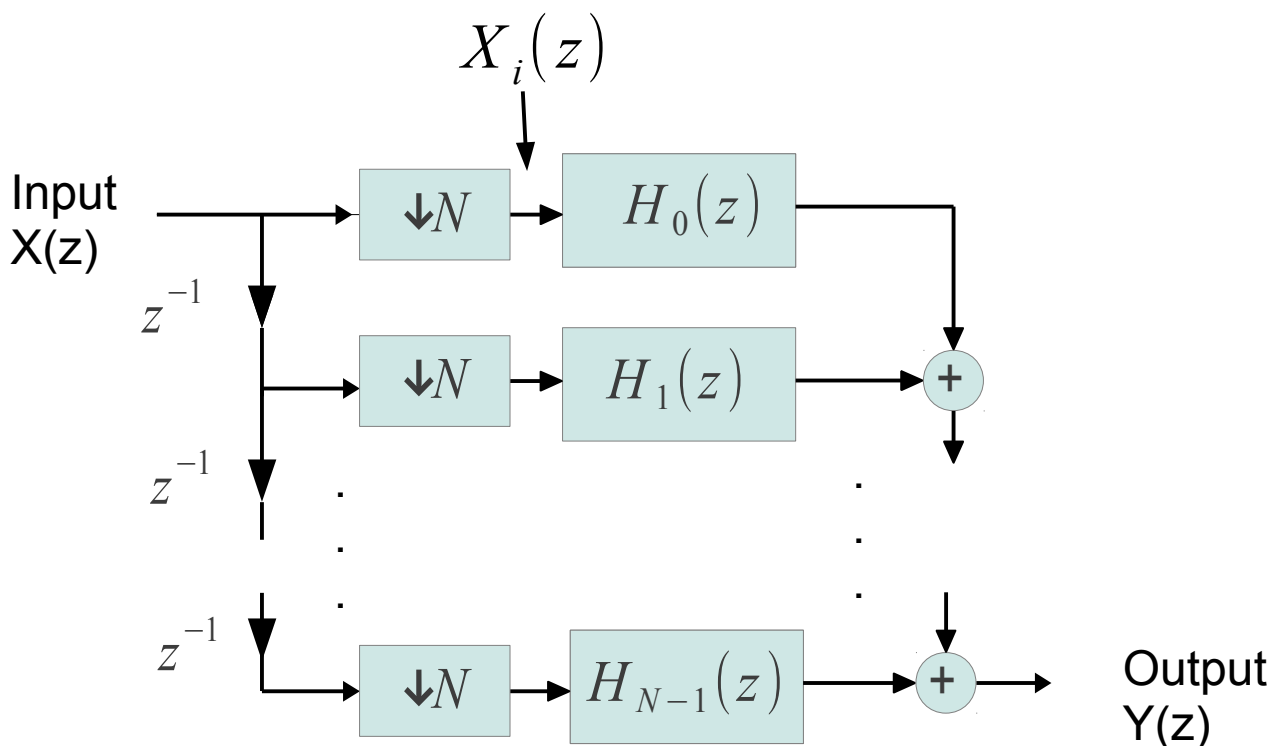
This could be a system for down-sampling rate conversion, where we first low-pass filter the signal x to avoid aliasing, and then down-sample it.

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 . This can also be seen as a **serial to**

parallel conversion for each N samples.

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.

Since we have N polyphase components in parallel, we can also represent them as **polyphase vectors**, and obtain a vector multiplication for the filtering at the lower sampling rate,

$$\sum_{i=0}^{N-1} X_i(z) \cdot H_i(z) = Y(z)$$

$$[X_0(z), \dots, X_{N-1}(z)] \cdot [H_0(z), \dots, H_{N-1}(z)]^T = Y(z)$$

Observe: If we have more than 1 filter, we can collect their polyphase vectors into “**polyphase matrices**”.

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 $h=[0.5 \ 1 \ 1.1 \ 0.6]$ and a down-sampling factor $N=2$. Hence we get the z-transform or the impulse response as

$H(z)=0.5+1\cdot z^{-1}+1.1\cdot z^{-2}+0.6\cdot z^{-3}$ and its polyphase components as

$$H_0(z)=0.5+1.1\cdot z^{-1}, \quad H_1(z)=1+0.6z^{-1}$$

in the time domain (in Matlab or Octave)

$$h0=[0.5 \ 1.1] ; \quad h1=[1 \ 0.6] ;$$

Produce the 2 phases of a down-sampled input signal x:

$$x0=x(1:2:end) ; \quad x1=x(2:2:end) ;$$

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:

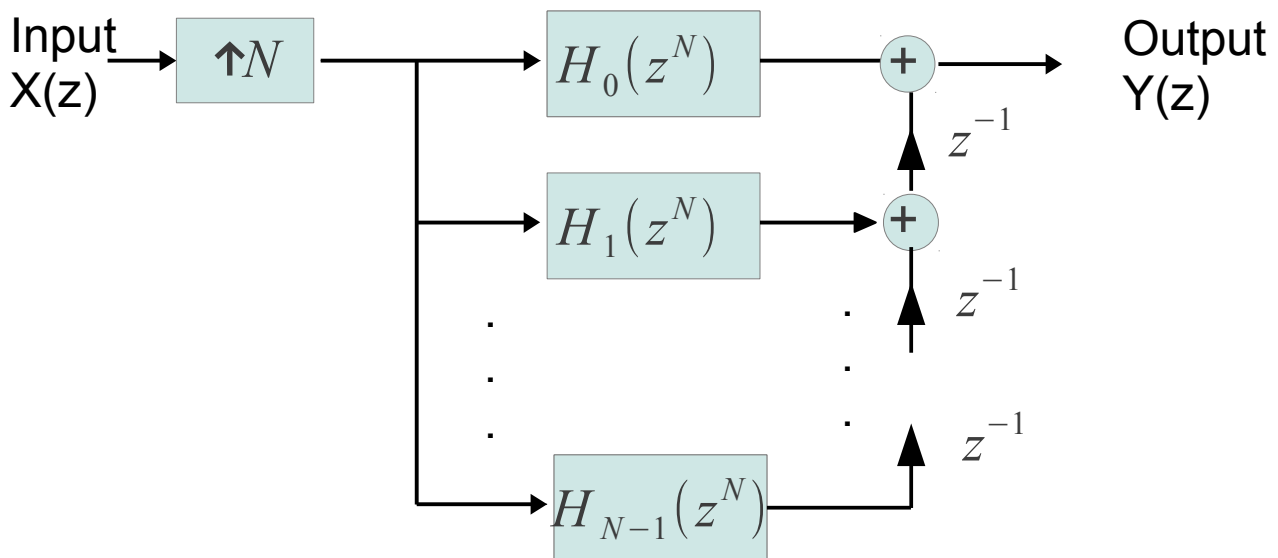
$$sound(y, 4000) ;$$

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

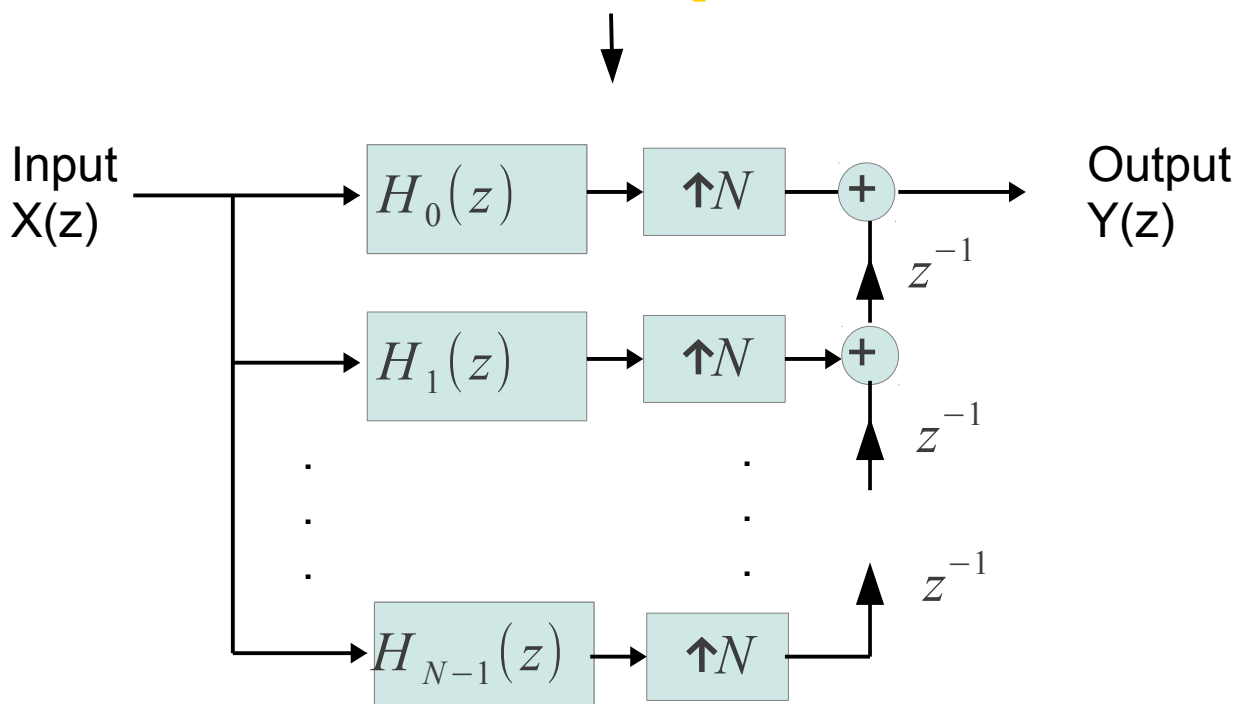
Since $H_i(z^N)$ and z^{-i} are linear time-invariant systems, we can exchange their ordering,

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

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



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



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. This can also be seen as a **parallel to serial conversion**.

With the polyphase elements $Y_i(z)$ the processing at the lower sampling rate can also be written in terms of **polyphase vectors**

$$X(z) \cdot [H_0(z), \dots, H_{N-1}(z)] = [Y_0(z), \dots, Y_{N-1}(z)]$$

Observe: If we have more than 1 filter, we can collect their polyphase vectors into **polyphase matrices**.

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

$h_0=[0.5 \ 1]$ and $h_1=[1 \ 0.5]$

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 y_0 and y_1 ,

$y_0 = \text{filter}(h_0, 1, y)$; $y_1 = \text{filter}(h_1, 1, y)$;

The complete up-sampling the signal is then obtained from its 2 polyphase components, performing our de-blocking

$L = \max(\text{size}(y))$;

$y_u(1:2:(2*L)) = y_0$; $y_u(2:2:(2*L)) = y_1$;

Where now the signal y_u is the same as if we had first up-sampled and then filtered the signal!

Now listen to the up-sampled signal:

$\text{sound}(y_u, 16000)$;