Audio- / Videosignalverarbeitung
Advanced Digital Signal Processing
Digital Signal Processing 2

Lecture 1

WS 2014/2015

Alina Rubina

Helmholtzbau, H3522

Organizational information (1)

- Lectures every week
  - Tuesdays at 9-10.30 in Sr Oe 109
- Seminars every two weeks
  - Wednesdays (**even** weeks) at 13-14.30 in K 2003B
  - Tuesdays (**even** weeks) at 11-12.30 in HU 210
  - Thursdays (**odd** weeks) at 9-10.30 in K 2002A
- Quizzes every week

Organizational information (2)

- This week
  - **NO** lecture on Tuesday
  - **Lecture 1** is postponed to:
    - Wednesday (15.10) at 13-14.30 in K-Hs2
    - Thursday (16.10) at 09-10.30 in K-Hs2
  - NO seminars and quizzes
    - Seminars will start **next week**
    - Quizzes will also start **next week after the second lecture**

Organizational information (3)

- Signing in for seminars
- Follow the link: http://doodle.com/3gexvbmwt6f8yrgh
- Choose one available option
- Sign in with the full name and surname

Course organization (1)

- The homework points account for **30% of the final grade**. The **exam** accounts for the other **70%**.
- Every homework is worth 3 points. Every quiz is worth 1 point. Quizzes count as 25% and seminars as 75% for the homework points.
- Gained points will only be added after passing the exam. When a student fails the exam the points stay valid until the lecture is held again and there are new homework assignments (in the following winter semester).

Course organization (2)

- We have 3-5 LP (“Leistungspunkte”) for lecture and seminar, 1 LP means about 2-3 hours per week
- ADSP has 7 LP, so you will get an extra Homework assignment in the end
  - **NO** ‘Praktika’ on Thursdays

Contact information

- We are always available via e-mail:
- alina.rubina@tu-ilmenau.de
- gerald.schuller@tu-ilmenau.de
- Course website (course description):
- http://www.tu-ilmenau.de/it-dsv/lv/it-dsvlehre/
- **Moodle platform** (all announcements and materials):
- https://moodle2.tu-ilmenau.de/

Moodle (1)

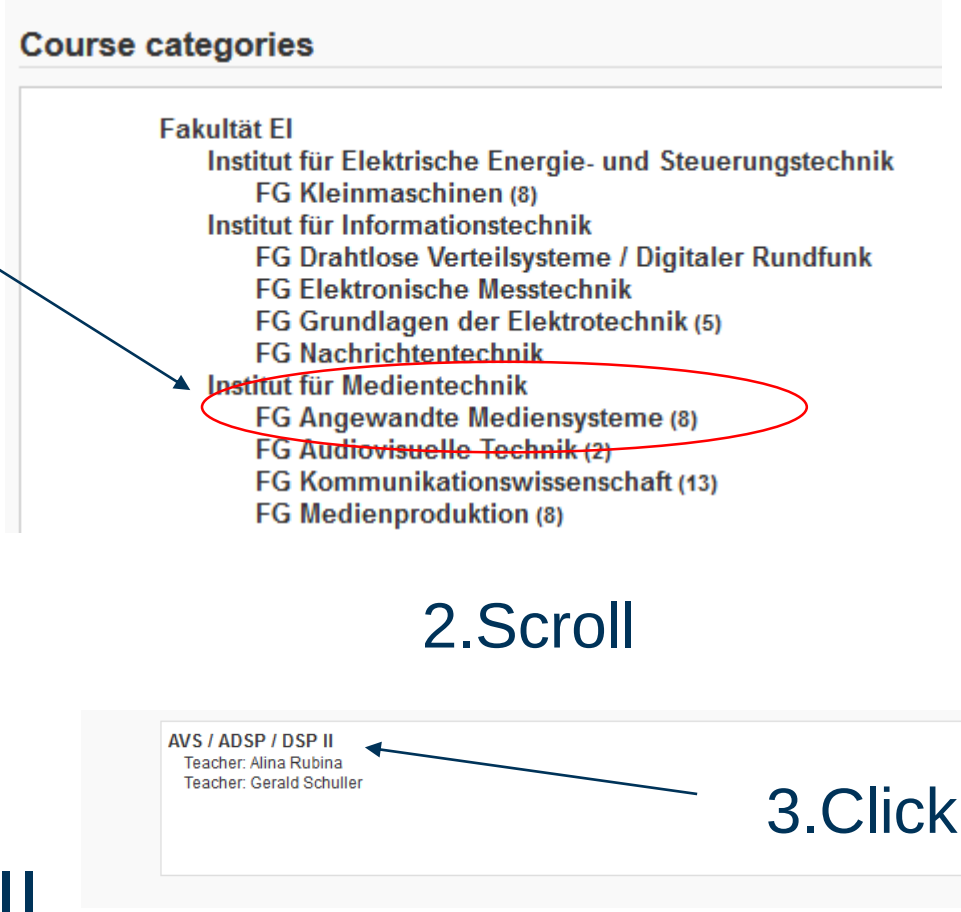
- E-learning platform
- Quizzes, slides, assignments and announcements
- Follow the link: https://moodle2.tu-ilmenau.de/

Change of language

Sign in with your university name and password

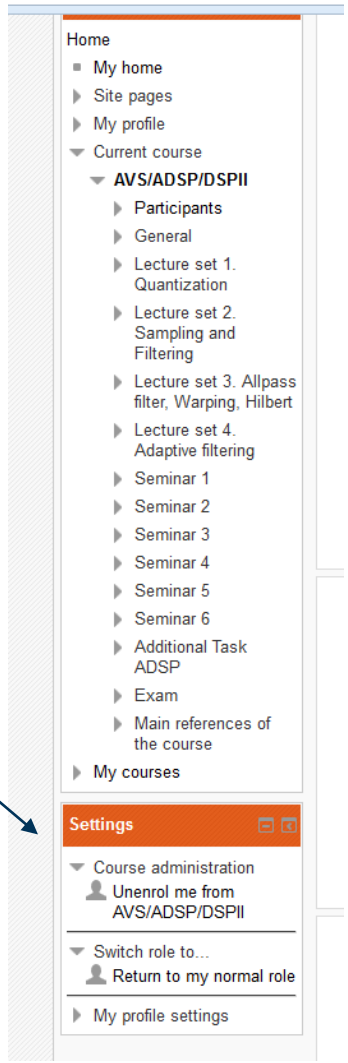
Moodle (2)

- Proceed as following:
- Fakultät EI --> Institut für Medientechnik --> FG Angewandte Mediensysteme --> AVS/ADSP/DSPII



Moodle (3)

- Enroll in course
- Follow the Moodle for new materials, announcements and quizzes
- Participate in Forums



Quizzes

- Test related to the latest lecture material
- Quizzes every week after the lecture
- Help in preparing for the **exam**
- For every quiz there is **one week** time to complete it
- **Deadlines** will be announced

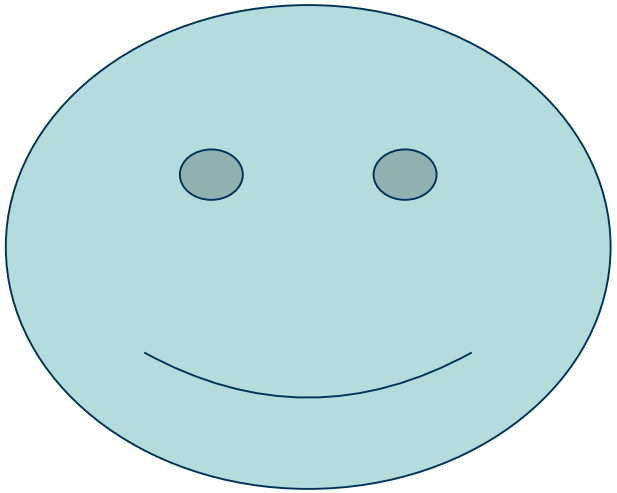
Seminars

- Seminars every two weeks
- **First part** of the seminar (15-30 min):
  - Answers to questions
  - Solving moodle tasks
  - Explaining the next homework assignment
- **Second part**
  - Check of the previous homework task
  - Principle: **First in first out**

Matlab / Octave / Python

- **QtOctave**
  - http://www.malinc.se/math/octave/mainen.php (Windows and Mac)
  - Graphical user interface for Octave
  - Freeware
  - Includes a tutorial
  - Apart from some syntax differences almost the same as Matlab
- **Matlab**
  - for those who have it
  - or use it at the computers at the UniRZ
  - Tutorial will be available on moodle2
- **Python Pylab**
  - Open source
  - http://ipython.org/install.html
- **Helpful links**
  - http://www.mathworks.de/help/techdoc/
  - http://www.gomatlab.de/ (german)

Thank you for your attention!



Digital Signal Processing 2/ Advanced Digital Signal Processing/ Audio-Video Signalverarbeitung
Lecture 1,
Organisation,
A/D conversion, Sampling

Gerald Schuller, TU Ilmenau

Gerald Schuller
gerald.schuller@tu-ilmenau.de

Organisation:
-Lecture each week, 2SWS,
-Seminar every other week, 1 SWS
-Homework assignments, they will receive points over the course of the semester, and the sum of points will count towards the final grade. The homework is divided into programming assignments and quizzes in the online learning platform “Moodle 2”, in which you need to register.
There will also be a final written exam at the end of the semester.
-Homework will count for 30%, and final exam for 70% of the final grade. For the homework, quizzes will count for 25%. You still need to pass the exam to pass the course.
-Web site: There is a web site for the lecture, under “Fachgebiet DSV (Digitale Signalverarbeitung)”, which will contain the current lecture slides, and current announcements:
http://www.tu-ilmenau.de/it-dsv/lv/

or clicking through in http://www.tu-ilmenau.de/mt
-Lehrveranstaltungen Master/Bacelor
-DSP2/ADSP, or Audio-Videoverabreitung.

We have 3-5 LP (“Leistungspunkte”) for lecture and seminar, 1 LP means about 2-3 hours per week, together 8-12 hour/week.
ADSP will get an extra Homework assignment in the end.

Will will use **Matlab** or its Open Source cousin **Octave** for programming. The Open Source system **Python Pylab** is also quite similar and can be faster and closer to the hardware, so we may also use it. Octave and Python Pylab can be downloaded freely from the Internet for all operating systems. We assume some familiarity with Matlab, Octave or Python Pylab for this course.
For instance, from a Terminal, you start Python Pylab with the command: “**ipython –pylab**”

Since we use Matlab or Octave or Python, we will also allow **programmable scientific pocket calculators** in the final exams.

Observe: The lecture slides are not a replacement of **books** on the subject, but something like a help to explain a topic, and something like a protocol for topics and questions treated. References to books are given throughout the lecture.

The slides are kept in **editing mode** on purpose. In this way they work like a **blackboard**, and answers to questions can be included immediately.
The main **purpose** of the lecture is to “**tailor**” the lecture content to your background and to **answer questions.** I will read parts of the slides, so that you can then **ask questions** about it, or such that I can further explain a topic according to my guess of your background.
Hence it is best to read the **slides before** the lecture (the previous version is online) and to think about possible questions for the lecture, whose answer I then include in the updated version.

Definitions: (Advanced) Digital Signal Processing 2:

-**Digital**: Quantization, sampling, encoding, digital filters, Nyquist
-**Signal**: Discrete Time, continuous time, TDMA, CDMA, OFDM, Voice, electrical current, white noise, colored noise, Sound, audio signals, electromagnetic waves, video signals, X-Ray, EEG (brain waves), Heart beat waves (ECG), Seismic waves.
Processing: Fourier Transform, Fourier Series, Cosine/Sine Transform, Filters, z-Transform, Discrete Fourier Transform (DFT), Fast Fourier Transform (FFT), upsampling, downsampling, encoding/decoding, quantization, modulation, denoising,

Part 2/Advanced: Previously learned? Which to start from? Your background? Media Technology: Multrate Signal Processing, CSP: Statistics, DFT, sampling quatization, Fourier Series, analog signal processing.

Application Example

ITU G.711 speech coding.
This is the first standard for ISDN speech coding, for speech transmission at 64 kb/s, with telephone speech quality. It uses a sampling rate of 8 kHz, and 8 bits/sample (hence we get 8*8=64 kbits/s). To obtain the 8 bits/sample, it uses companding, A-law or mu-law companding, to obtain 8 bits/sample from originally 16 bits/sample, with still acceptable speech quality. (ITU means International Telecommunications Unions, it standardizes speech communications)

The procedure of G.711 can be seen as the following

-Microphone input
-Analog-to-Digital converter (A/D converter), including sampling at 8 kHz sampling rate. It usually generates 14-16 bits/sample. Before the sampling there is an (analog) low pass filter with cut-off frequency of 3.4 kHz.
-Companding (compressing) with an A-Law or u-Law function before quantization, like

$$y=\frac{\ln\left(1+\mu x\right)}{\ln\left(1+\mu\right)}$$ with $\mu=255$ for the North American standard, which generates 8 bits/sample from the original 16 bits/sample (see eg. http://www.dspguide.com/ch22/5.htm or the

Book: Jayant, Noll: “Digital Coding of Waveforms”, Prentice Hall)
-Transmission to the receiver
-Expanding the signal using the inverse A-Law or u-Law function after inverse quantization, to obtain the original 16 bits/sample range
-Digital to Analog conversion, including low pass filtering (about 3.4 kHz).

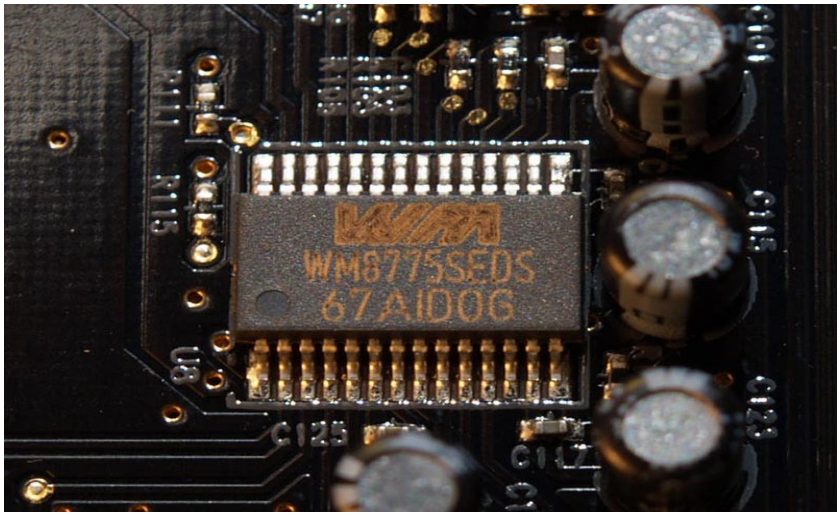
This system allow us to have a telephone conversation over a digital ISDN line at 64 kb/s. This system was standardized in the 1970's, and at that time the hardware did not allow for much more than companding. Today we have much more powerful hardware (Moore's Law), and hence we can devise much more powerful compression algorithms. Today, at 64 kb/s, we can get very high quality speech, instead of just telephone quality speech. Current speech coders allow for audio bandwidths of above 10 kHz at that bit rate, which means speech sounds like if we are right there with the speaker. Also, at 64 kb/s, we can transmit high quality audio/music signals, using for instance, the MPEG-AAC standard, which is also used in iTunes. This was actually the original motivation for the development of MPEG audio compression, to transmit high quality music over an ISDN line. ISDN lines are outdated nowadays, but compression remains useful, for instance for wireless connections (downloading or streaming to your wireless phone).
Observe: ISDN has a fixed bit rate (64kb/s), and has a fixed connection. You dial to the person you

would like to talk to, and have a fixed connection, and each bit you generate is send off to the receiver right away without delay.
In contrast: The internet has no fixed bit rate, it depends on the individual connection, and there is also no fixed connection. There, first we assemble a certain number of bits into packets, these packets get headers with the receiver address, and then the switches in the internet rout the packet to the receiver. Observe that this also leads to delay, first for the assembling of the packet, and then for the switching in the internet. This also shows that originally, the internet was not made for real time communications, but just for data traffic. But the internet is becoming faster, with less delay, because it is constantly expanding, for higher bit rates.

A/D and D/A Conversion of a signal

Example:

4-channel stereo multiplexed analog-to-digital converter WM8775SEDS made by Wolfson Microelectronics placed on an X-Fi Fatal1ty Pro sound card. From: Wikipedia, Analog-to-digital converter.
This is on the sound card of your computer, where you connect your microphone and speakers to.



This A/D converter measures the voltage at its input and assigns it to an index or codeword. A Matlab/Octave example is: Assume our A/D converter has an input range of -1V to 1V, 4 bit accuracy (meaning we have a total of 2^4 codewords or indices), and the A/D converter has **0.2 V** at its **input**. One possibility to obtain the **quantization stepsize** or **quantization interval** (in **Matlab/Octave**) is:

```
stepsize=(1-(-1))/(2^4);
stepsize = 0.12500
```

In **Pylab** this is: stepsize=(1.0-(-1.0))/(pow(2,4)); Next we get **quantization index** which is then encoded as a **codeword**:

```
index=  round(0.2/stepsize);
```

which results in:

```
index = 2
```

Observe: If the quantization **stepsize is constant**, independent of the signal, we call it a "**uniform quantizer**"

The index then is **coded** using the 4 bits and sent to a decoder, for instance using the 4 bit binary **codeword** “0010”. The first bit usually is the sign bit. The **decoder reconstructs** the voltage for instance be **multiplying the index** with the **stepsize**:

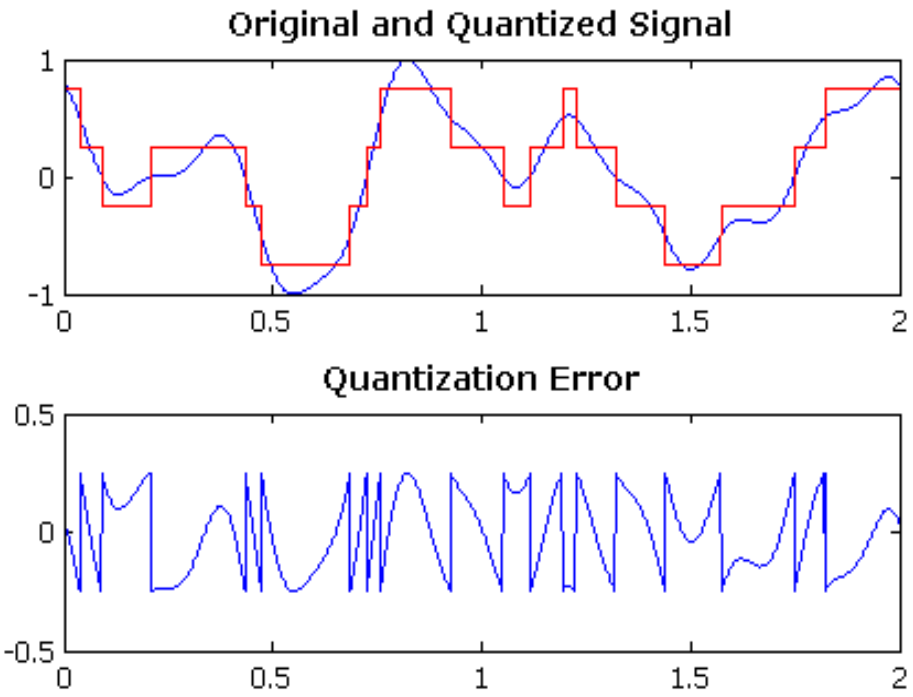
```
reconstr=stepsize*index
```

resulting in:

```
reconstr = 0.25000
```

This is also called the **quantized signal**, and its difference to the original value or signal is called the **quantization error**. In our example the quantization error is 0.25-0.2=**0.05**

Observe: There is always a range of voltages which is mapped to the same codeword. We call this range Δ , or **stepsize**. These steps represent the quantization in the A/D process, and they lead to quantization errors.
The output is a linear “**Pulse Code Modulation**” (**PCM**) signal. It is linear in the sense that the code values are proportional to the input signal values.



Quantization error for a full range signal.
(From: Wikipedia, quantization error, the red line shows the quantized signal, the blue line in the above image is the original analog signal)

Let's now call our quantization error "e". Then the **quantization error power** is the expectation value of the squared quantization error e:

$$E(e^2) = \int_{-\Delta/2}^{\Delta/2} e^2 \cdot p(e)\, de$$

where $p(e)$ is the probability of error value e. Here we compute the power of each possible error value e by squaring it, and multiply it with its probability to obtain the **average power**.

This number will give us some impression of the signal quality after quantization, if we set it in **relation to the signal energy**. Then we get a

Signal to Noise Ratio (SNR) for our quantizer and A/D converter.
Assume the quantization error e is uniformly distributed (all possible values of the quantization error e appear with equal probability), which is usually the case if the signal is much larger than the quantization step size Δ (large signal condition). Since the integral over the probabilities of all possible values of e must be 1, and the possible values of e are between $-\Delta/2$ and $+\Delta/2$, we have

$$p(e)=1/\Delta$$

which yields

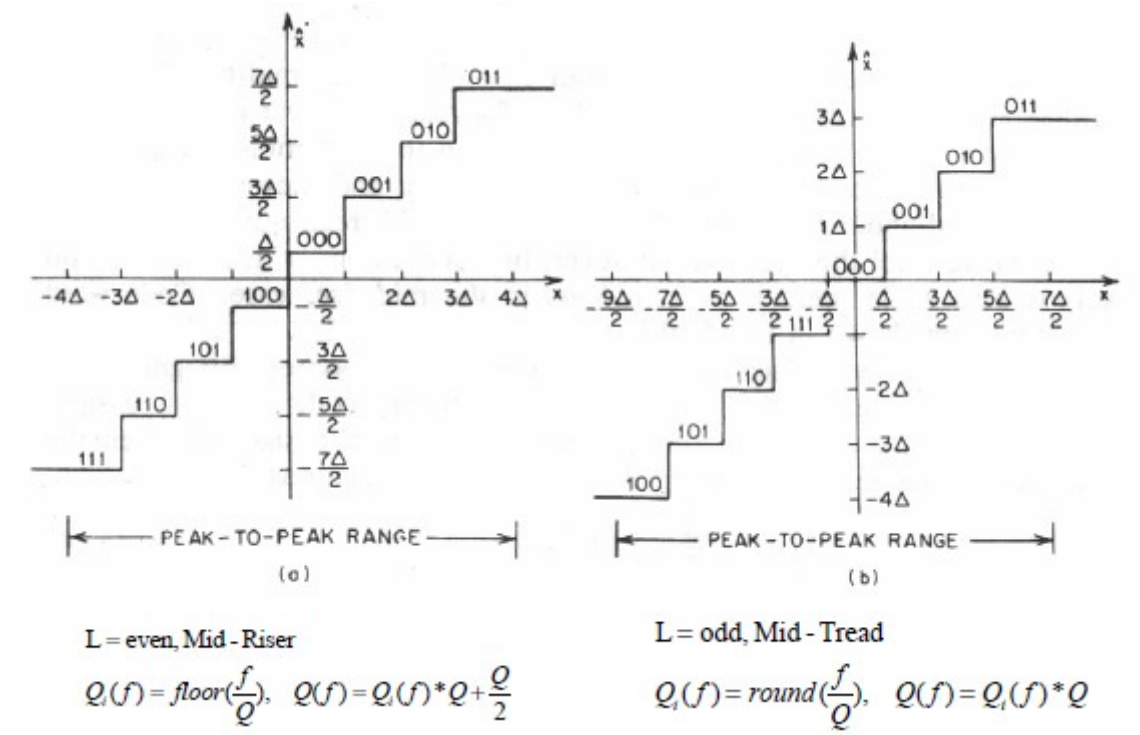
$$E(e^2)=\frac{1}{\Delta}\cdot\int_{-\Delta/2}^{\Delta/2} e^2\,de=\frac{1}{\Delta}\left(\frac{(\Delta/2)^3}{3}-\frac{(-\Delta/2)^3}{3}\right)=\frac{\Delta^2}{12}$$

Hence the **quantization error power for a uniform quantizer** with stepsize Δ and with a large signal is:

$$E(e^2)=\frac{\Delta^2}{12}$$

Mid-rise and Mid-tread quantization

Depending on if the quantizer has the input voltage 0 at the center of a quantization interval or on the boundary of that interval, we call the quantizer a mid-tread or a mid rise quantiser, as illustrated in the following picture:



L = even, Mid-Riser

$$Q_i(f) = floor(\frac{f}{Q}), \quad Q(f) = Q_i(f) * Q + \frac{Q}{2}$$

L = odd, Mid-Tread

$$Q_i(f) = round(\frac{f}{Q}), \quad Q(f) = Q_i(f) * Q$$

(From: http://eeweb.poly.edu/~yao/EE3414/quantization.pdf)

Here, $Q_i(f)$ is the index after quantization (which is then encoded and send to the receiver), and Q(f) is the inverse quantization, which produces the quantized reconstructed value at the receiver.

This makes mainly a difference at **very small input values**. For the mid-rise quantiser, very small values are always quantized to +/- half the quantization interval ($+/-\Delta/2$), whereas for the mid-tread quantizer, very small input values are always rounded to zero. You can also think about the **mid-rise** quantizer as **not having a**

zero as a reconstruction value, but only very small positive and negative values.
So the mid-rise can be seen as more accurate, because it also reacts to very small input values, and the mid tread can be seen as saving bit-rate because it always quantizes very small values to zero.
Observe that the expectation of the quantization error power for large signals stays the same for both types.

Octave/Matlab Example:

```
q=0.1; %Stepsize
x=[0.012,-1.234, 2.456, -3.789]
x =

   0.012000  -1.234000   2.456000  -3.789000
```

Mid-Tread quantizer (Encoder):

```
index=round(x/q)
index =

    0  -12   25  -38
```

De-quantization (Decoder):

```
reconstr=index*q
reconstr =

   0.00000  -1.20000   2.50000  -3.80000
```

Mid-Rise quantizer (Encoder):

```
index=floor(x/q)
index =
```

0 -13 24 -38

De-quantization (Decoder):

reconstr=index*q +q/2

reconstr =

0.050000 -1.250000 2.450000 -3.750000

In **Python**, you type the same

index=floor(x/q)

*reconstr=index*q +q/2*

to obtain the same result.

For Python, be sure to have the libraries python-scilab and python-matplotlib installed.

Real-time audio python example: The audio signal is between -32000 and +32000, and the quantization step size is q=5000.

Observe:

the Mid-Tread quantizer “swallows” small signal levels, since they are all rounded to zero.

The Mid-Rise quantizer still captures small levels, but distorted.

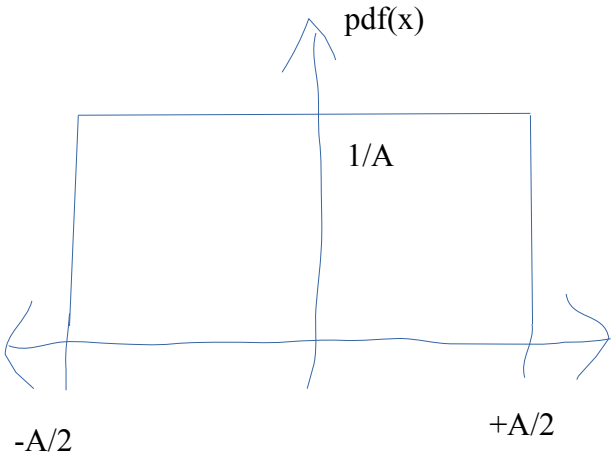
Digital Signal Processing 2/ Advanced Digital Signal Processing
Lecture 2,
Quantization, SNR

Gerald Schuller, TU Ilmenau

Quantization Signal to Noise Ratio (SNR).

Assume we have a A/D converter with a quantizer with a certain number of bits (say N bits), what is the resulting Signal to Noise Ratio (SNR) of this quantizer? The SNR is defined as the ratio of the expectation of the signal power to the expectation of the noise power. In our case, the expectation of the noise power is the expectation of the quantization error power. We already have the expectation of the quantization error power as $\Delta^2/12$.
So what we still need for the SNR is the average or **expectation of the signal power**. How do we obtain this?
Basically we can take the same approach as we did for the expectation of the power of the quantization error (which is basically the second moment of the distribution of the quantization error). So what we need to know from our signal is its **probability distribution**. For the quantization error it was a uniform distribution between $-\Delta/2$ and $+\Delta/2$.

A very **simple case** would be a **uniformly distributed signal** with amplitude A/2, which has values between -A/2 up to +A/2.



So we could again use our formula for the average power, but now for our signal x:

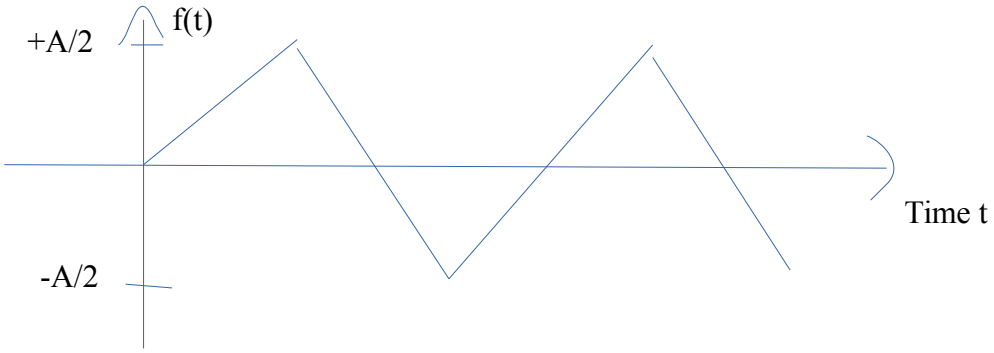
$$E(x^2) = \int_{-A/2}^{A/2} x^2 \cdot p(x)\,dx$$

So here we have the same type of signal, and the resulting expectation of the power (its second moment, assumed we have a zero mean signal) is obtained by using our previous formula, and replace Δ by A. The resulting power is: $\frac{A^2}{12}$

Which signals have this property? One example is uniformly distributed random values (basically like our quantization error). Matlab or Octave produces this kind of signal if we use the command: rand()-0.5 (the 0.5 is to make the distribution centered around 0). In Python, the command for a random number is: *scipy.rand()*.

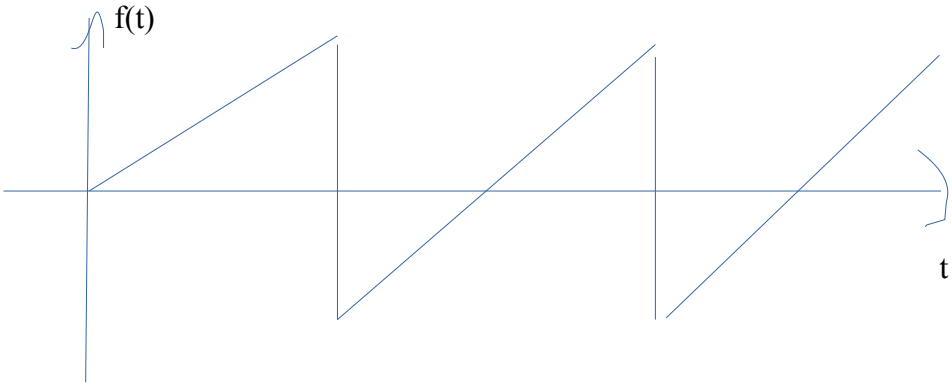
Observe: Speech has a non-uniform pdf, it is usually modeled by a Laplacian distribution or a gausssian mixture model, so it doesn't apply to this case!

An example for a uniform pdf: a **triangular wave**:



How do we obtain its pdf? One can imagine the vertical axis (the function value) covered by small intervals, and each interval is then passed in the same time-span. This means that the resulting pdf is also uniform!

A further example: A **sawtooth wave**:



Again we can make the same argument, each small interval of our function value is covered in the same time-span, hence we obtain a uniform distribution.

We now have seen a few examples which fulfil our assumption of a uniform distribution (realistic examples), and we know: their expectation of their power is $A^2/12$. So what does this then mean for the SNR? The **SNR** is just the ratio

$$SNR = \frac{A^2/12}{\Delta^2/12} = \frac{A^2}{\Delta^2}$$

If we assume our signal is full range, meaning the maximum values of our A/D converter is -A/2 and +A/2 (the signal goes to the maximum), we can compute the step size Δ if we know the **number of bits** of converter, and if we assume uniform quantization step sizes. Assume we have **N bits** in our converter. This means we have 2^N

quantization intervals. We obtain Δ by dividing the full range by this number,

$$\Delta = \frac{A}{2^N}$$

Plug this in the SNR equation, and we obtain:

$$SNR = \frac{A^2}{\Delta^2} = \frac{A^2}{(A/2^N)^2} = 2^{2N}$$

This is now quite a simple result! But usually, the SNR is given in dB, so lets convert it into dB:

$$SNR_{dB} = 10 \cdot \log_{10}(2^{2N}) = 10 \cdot 2N \cdot \log_{10}(2) \approx$$

$$\approx 10 \cdot 2N \cdot 0.301\, dB = N \cdot 6.02\, dB$$

This is now our famous **rule of thumb**, that **each bit** more gives you about **6 dB more SNR**. But observe that the above formula only holds for uniformly distributed full range signals! (the signal is between -A/2 and +A/2, using all possible values of our converter)

What happens if the signal is not full range? What is the SNR if we have a signal with reduced range? Assume our signal has an amplitude of A/c, with a factor c>1.
We can then simply plug this into our equation:

$$SNR = \frac{(A/c)^2}{\Delta^2} = \frac{(A/c)^2}{(A/2^N)^2} = \frac{2^{2N}}{c^2}$$

in dB:

$$SNR_{dB} = 10 \cdot \log_{10}\left(\frac{2^{2N}}{c^2}\right) = 10 \cdot 2N \cdot \log_{10}(2) - 20 \cdot \log_{10}(c) \approx$$

$$\approx 10 \cdot 2N \cdot 0.301\,dB - 20 \cdot \log_{10}(c) =$$

$$= N \cdot 6.02\,dB - 20 \cdot \log_{10}(c)$$

The last term, the $20 \cdot \log 10(c)$, is the number of dB which we are below our full range. This means we **reduce our SNR** by this number of **dB** which we are **below full range**!

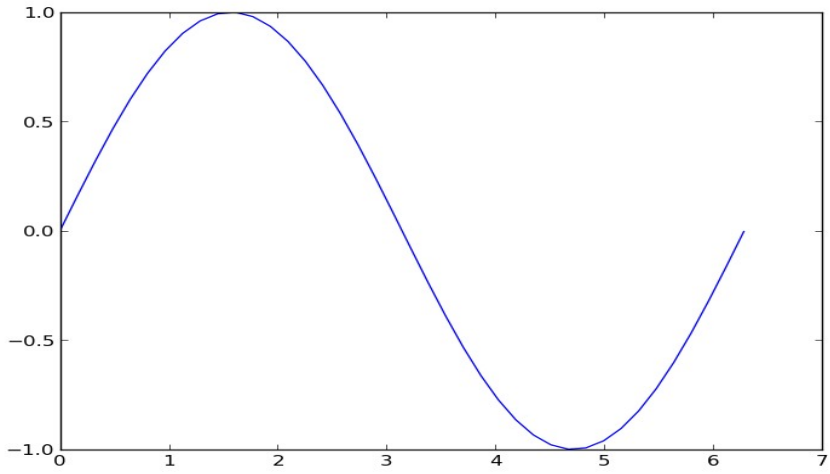
Example: We have a 16 bit quantiser, then the SNR for uniformly distributed full range signals would be

$$SNR = 6.02 \cdot 16\,dB = 96.32\,dB$$

Now assume we have the same signal, but 20dB below full range (meaning only 1/10th of the full range). Then the resulting SNR would be only 96.32-20=76.32 dB! This is considerably less. This also shows why it is important not to make the safety margin to full range too big! So for instance our sound engineer should keep the signal as big as possible, without ever reaching full range to avoid clipping the signal.

The other assumption we made concerned the type of signal we quantize. What if we don't have a uniformly distributed signal? As we saw, speech signals are best modeled by a Laplacian distribution or a gaussian mixture model, and similar for audio signals. Even a simple sine wave does not fulfill this assumption of a uniform

distribution. What is the pdf of a simple sine wave?



Observe: If a sinusoid represents a full range signal, its values are from $-A/2$ to $+A/2$, as in the previous cases.

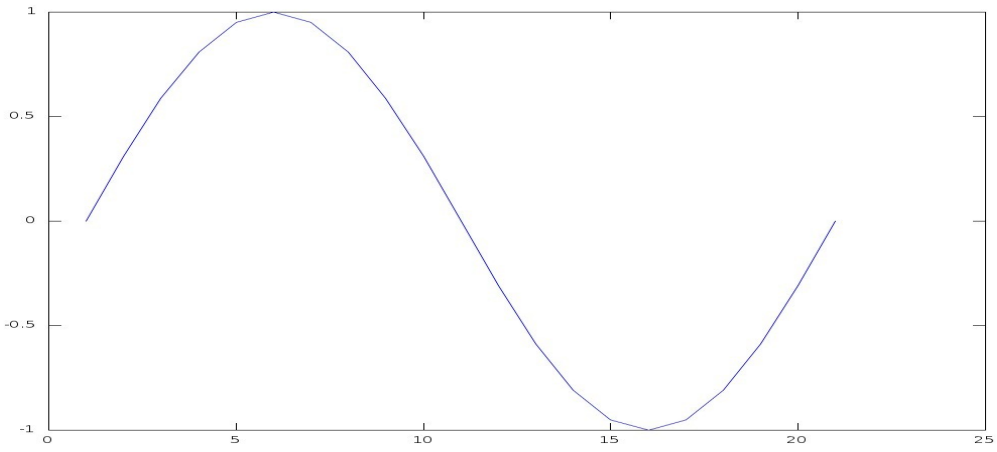
Remark: This plot was made in python with the following terminal commands:

```
ipython --pylab
t=linspace(0,6.28,100);
s=sin(t);
plot(t,s)
```

Digital Signal Processing 2/ Advanced Digital Signal Processing
Lecture 3,
SNR, non-linear Quantisation

Gerald Schuller, TU Ilmenau

What is our SNR if we have a sinusoidal signal? What is its pdf? Basically it is its normalized histogram, such that its integral becomes 1, to obtain a probability distribution.



If we look at the signal, and try to see how probable it is for the signal to be in a certain small interval on the y axis, we see that the signal stays longest around +1 and -1, because there the signal slowly turns around. Hence we would expect a pdf, which has peaks at +1 and -1.
If you calculate the pdf of a sine wave, x=sin(t), with t being continous and with a range larger than 2pi, then the result is

$$p(x)=\frac{1}{\pi\cdot\sqrt{1-x^2}}$$

This results from the derivative of the inverse sine function (arcsin). This derivation can be found for instance on Wikipedia. For our pdf we need to know how fast a signal x passes through a given bin in x. This is what we obtain if we compute the inverse function $x=f^{-1}(y)$, and then its derivative $df^{-1}(x)/dy$.

Here we can see that p(x) indeed becomes infinite at x=+/-1! We could now use the same approach as before to obtain the expectation of the power, multiplying it with x^2 and integrating it. But this seems to be somewhat tedious. But since we now have a deterministic signal, we can also try an **alternative** solution.

We can simply directly compute the power of our sine signal over t, and then take the average over at least one period of the sine function.

$$E(x^2)=\frac{1}{2\pi}\int_{t=0}^{2\pi}\sin^2(t)\,dt=\frac{1}{2\pi}\int_{t=0}^{2\pi}(1-\cos(2t))/2\,dt$$

the cosine integrated over complete periods becomes 0, hence we get

$$=\frac{1}{2\pi}\int_{t=0}^{2\pi}1/2\,dt=\frac{1}{2\pi}\cdot\pi=\frac{1}{2}$$

What do we get for a sinusoid with a different amplitude, say $A/2\cdot\sin(t)$?

The expected power is

$$E(x^2)=\frac{A^2}{8}$$

So this leads to an SNR of

$$SNR = \frac{A^2/8}{\Delta^2/12} = \frac{3 \cdot A^2}{2 \cdot \Delta^2}$$

Now assume again we have a A/D converter with N bits, and the sinusoid is at full range for this converter. Then

$$A = 2^N \cdot \Delta$$

We can plug in this result into the above equation, and get

$$SNR = \frac{3 \cdot 2^{2N} \cdot \Delta^2}{2 \cdot \Delta^2} = 1.5 \cdot 2^{2N}$$

In dB this will now be

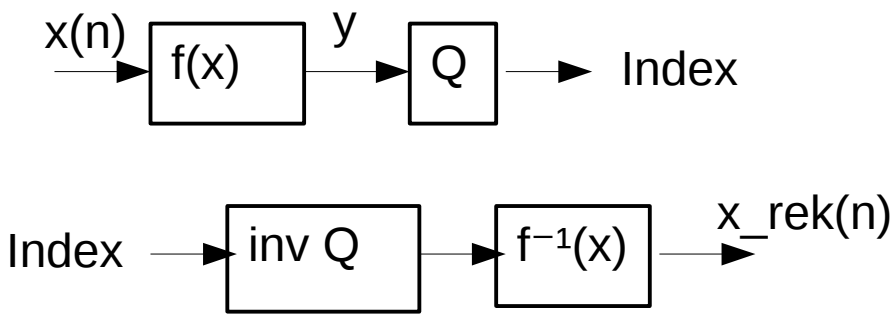
$$10 \cdot \log_{10}(SNR) = 10 \cdot \log_{10}(1.5) + N \cdot 20 \cdot \log_{10}(2) =$$
$$= 1.76\,dB + N \cdot 6.02\,dB$$

Here we can see now, that using a sinusoidal signal instead of a uniformly distributed signal gives us a **boost of 1.76 dB** in SNR. This is because it is more likely to have larger values!

Companding

This is a scheme to make the SNR less dependent on the signal size.
This is a synonym for compression and expanding. Uniform quantization can be seen as a

quantization value which is constant on the absolute scale. Non-uniform quantization, using companding, can be seen as having step sizes which stay constant relative to the amplitude, their step size grows with the amplitude.
We obtain this non-uniform quantization by first applying a non-linear function to the signal (to boost small values), and then apply a uniform quantizer. On the decoding side we first apply the inverse quantizer, and then the inverse non-linear function (we reduce small values again to restore their original size).



The range of (index) values is compressed, smaller values become larger, large values don't grow as fast. The following functions are standardized as “ μ -Law” and “A-Law”:

EQUATION 22-1
Mu law companding. This equation provides the nonlinearity for μ255 law companding. The constant, μ, has a value of 255, accounting for the name of this standard.

$$y = \frac{\ln(1+\mu x)}{\ln(1+\mu)} \quad \text{for } 0 \le x \le 1$$

EQUATION 22-2
"A" law companding. The constant, A, has a value of 87.6.

$$y = \frac{1+\ln(Ax)}{1+\ln(A)} \quad \text{for } 1/A \le x \le 1$$

$$y = \frac{Ax}{1+\ln(A)} \quad \text{for } 0 \le x \le 1/A$$

(From: http://www.dspguide.com/ch22/5.htm, also below)

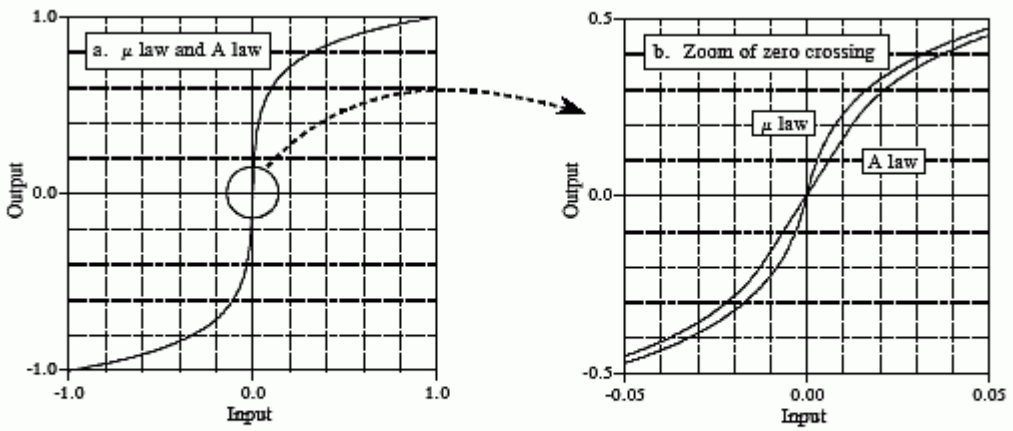


FIGURE 22-7
Companding curves. The μ255 law and "A" law companding curves are nearly identical, differing only near the origin. Companding increases the amplitude when the signal is small, and decreases it when it is large.

This "**compression**" function is applied **before a uniform quantizer** in the **encoder**. In the decoder, after uniform reverse quantisation, the inverse function is applied, turning y back into x. Observe that these equations assume that we first normalize our signal and keep its sign separate.

Example: For $\mu = 255$ (which is used in the standard) and an x with a maximum amplitude of A, hence $-A \le x \le A$ we obtain:

$$y = sign(x) \cdot \frac{\ln(1+255 \cdot |(x/A)|)}{\ln(1+255)}$$

In the example of 8-bit mu-law PCM the quantization **index** is then

$$index = round(2^7 \cdot y)$$

since we take 7 bits for the magnitude of the signal and 1 bit for its sign. This index is then encoded as a 7+1= 8 bit **codeword**.

In the **decoder** we compute the inverse quantized y including its sign from the index,

$$y = index/(2^7)$$

and we compute the inverse compression function, the "**expanding**" function (hence the name "**companding**")
We obtain the inverse through the following steps,

$$|(y)| = \frac{\ln(1+255 \cdot |(x/A)|)}{\ln(1+255)}$$

$$|y| \cdot \ln(256) = \ln(1+255 \cdot |x/A|)$$

$$\rightarrow e^{\ln(256) \cdot |y|} = 1+255 \cdot |x|/A$$

$$\rightarrow \frac{(256^{|y|}-1)}{255} \cdot A = |x|$$

$$\rightarrow x = sign(y) \frac{(256^{|y|}-1)}{255} \cdot A$$

This x is now our **de-quantized value** or signal. Observe that with this companding, the effective quantisation step size remains approximately constant **relative** to the **signal amplitude**. Large signal components have large step sizes and hence larger quantisation errors, small signals have smaller quantisation step sizes and hence smaller quantisation errors. In this way we get a more or less **constant SNR** over a wide **range of signal amplitudes**.

Important point to remember: this approach is **identical** to having **non-uniform quantization** step sizes, smaller step sizes at smaller signal values, and larger step sizes at larger signal values. The compression and expanding of the signal makes the uniform step sizes “look” relatively smaller to the signal, it has more quantization steps to cover. And this has the same effect as a smaller signal with smaller quantization steps.

Python example to mu-law quantizer:

First listen to the uniform mid-tread quantizer using our python script, and make sure the part for the mid-tread quantizer is un-commented:

```
python pyrecplay_quantizationblock.py
```

Observe that the voice dissapear, is rounded to zero, if the speaker is at some distance from the microphone.

Now use mu-law quantization with:

```
python pyrecplay_mulawquantizationblock.py
```

Observe: At the distance at which the speech disappeared in the uniform case, the speech is now there.

At closer distance there is not much difference.

Lloyd-Max Quantizer

This is a type of non-uniform quantizer, which is adapted to the signals pdf. It basically minimizes the expectation of the quanization power (its second moment), given the pdf of the signal to quantize.

Let's call our Quantisation function Q(x) (this is quantization followed by inverse quantization). You can also think of non-uniform quantization as first applying this non-linear function and then to use uniform quantization. Then the expectation of our quantization power is

$$D = E\left(\left(x - Q\left(x\right)\right)^2\right)$$

Observe that we use the square here, and not for instance the magnitude of the error, because the square leads to an easier solution for minimum, which we would like to find.

Digital Signal Processing 2/ Advanced Digital Signal Processing
Lecture 4,
Lloyd-Max Quantizer, LBG

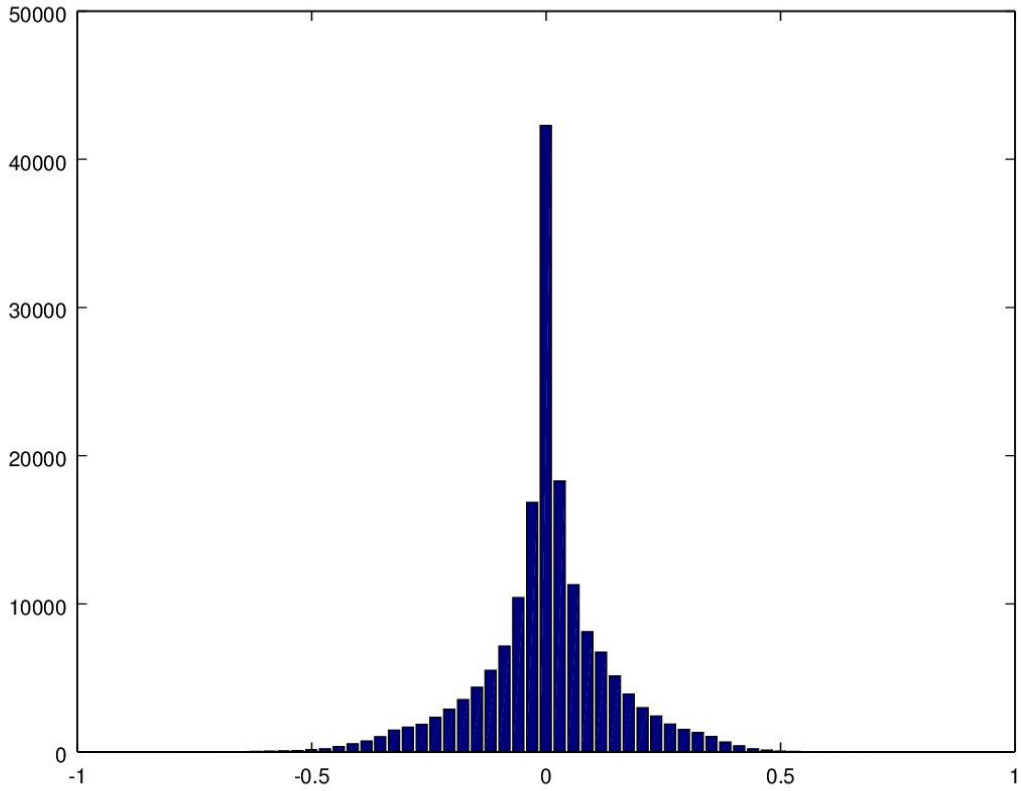
Gerald Schuller, TU Ilmenau

Lloyd-Max Quantizer

Look at the histogram of a typical mixed speech and music file in Octave:

```
snd=wavread('mixedshort.wav');
hist(snd,50)
```

This means we count how often the signal samples in our signal “snd” fall into one of 50 bins between -max and +max amplitude, in this case between -1 and 1, since Octave normalizes the sound with wavread to this range. This is the result:



Observe: This distribution is far from being uniform! On the contrary, it is very “peaky”, with most samples being in bin 0, which means most samples have a very small amplitude (magnitude value).
Idea: Wouldn't it be helpful if we choose our **quantzation steps smaller** where **signal samples appear most often**, to reduce the quantization error there, and make the quantization step size (and also the error) larger, where there are only a few samples?
This is the idea behind the Lloyd-Max quantizer.

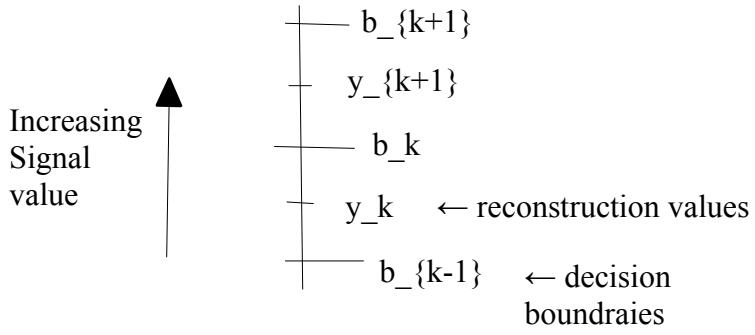
This is a type of non-uniform quantizer, which is adapted to the signals pdf. It basically minimizes the expectation of the quanization power (its second moment), given the pdf of the signal to quantize.
Let's call our Quantisation function Q(x) (this is quantization followed by inverse quantization).
You can also think of non-uniform quantization as first applying this non-linear function and then to use uniform quantization. Then the expectation of our quantization power is

$$D = E\left(\left(x - Q\left(x\right)\right)^2\right)$$

Observe that we use the square here, and not for instance the magnitude of the error, because the square leads to an easier solution for minimum, which we would like to find.

Our **goal** is to **minimize this expectation** of the quantisation error power D.

Starting with the pdf of our signal, and the result should be our quantisation intervals and reconstruction values. Since we now assume non-uniform intervals, we need to give those intervals and their reconstruction values names, which can be see in the following graphic



We call b_k the decision boundaries, and the y_k the reconstruction values (after quantisation and inverse quantisation). In the multidimensional case, they are also called a "codeword".

So using these definitions, and the pdf of our signal p(x), we can re-write our equation for the distortion:

$$D = E\left(\left(x - Q(x)\right)^2\right) = \int_{-\infty}^{\infty} \left(x - Q(x)\right)^2 p(x)\, dx =$$

we can now subdivide the integral over the quantisation intervals, assuming we have M quantization intervals, by just adding the quantization error power of all the quantisation intervals (see also: Wikipedia: quantization (signal processing)):

$$D = \sum_{k=1}^{M} \int_{b_{k-1}}^{b_k} (x - y_k)^2 \, p(x) \, dx$$

We would now like to have the minimum of this expression for the decision boundaries b_k and the reconstruction values y_k. Hence we need to take the first derivative of the distortion D with respect to those variables and obtain the zero point.

Lets start with the decision boundaries b_k:

$$\frac{\partial D}{\partial b_k} = 0$$

To obtain this derivative, we could first solve the integral, over 2 neighbouring quantisation intervals, because each decision interval b_k appears in two intervals (one where it is the upper boundary, and one where it it the lower boundary).

$$D_k = \int_{b_k}^{b_{k+1}} (x - y_{k+1})^2 \, p(x) \, dx + \int_{b_{k-1}}^{b_k} (x - y_k)^2 \, p(x) \, dx$$

Here we cannot really get a closed form solution for a general probability function p(x). Hence, to simplify matters, we make the **assumption** that p(x) is **approximately constant** over our 2 neighbouring quantisation intervals. This means we assume that our quantisation intervals are small in comparison with the changes of p(x)! We need to keep this assumption in mind, because the derived algorithm is based on this assumption!

Hence we can set:

$$p(x) = p$$

Using this simplification we can now solve this integral:

$$\frac{D_k}{p}=\frac{(b_k-y_k)^3}{3}-\frac{(b_{k-1}-y_k)^3}{3}+\frac{(b_{k+1}-y_{k+1})^3}{3}-\frac{(b_k-y_{k+1})^3}{3}$$

Since we now have a closed form solution, we can easily take the derivative with respect to b_k (which only influences D_k in D, hence we can drop the k in the derivative):

$$\frac{\partial D/p}{\partial b_k}=(b_k-y_k)^2-(b_k-y_{k+1})^2$$

We can set this then to zero, and observing that $y_{k+1} > b_k$ (see above image), we can take the positive square root of both sides:

$$(b_k-y_k)^2-(b_k-y_{k+1})^2=0$$

$$\rightarrow (b_k-y_k)=(y_{k+1}-b_k)$$

$$\rightarrow b_k=\frac{y_{k+1}+y_k}{2}$$

This means that we put our decision boundaries right in the middle of two reconstruction values. But remember, this is only optimal if we assume that the signals pdf is roughly constant over the 2 quantisation intervals! This approach is also called the "**nearest neighbour**", because any signal value or data point is always quantized to the **nearest reconstruction value**. This is one important result of this strategy.

Now we have the decision boundaries, but we still need the reconstruction values y_k. To obtain

them, we can again take the derivative of D, and set it to zero. Here we cannot start with an assumption of a uniform pdf, because we would like to have a dependency on a non-uniform pdf. We could make this assumption before, because we only assumed it for the (small) quantisation intervals. This can be true in practice also for non-uniform pdf's, if we have enough quantisation intervals.
But to still have the dependency on the pdf, for the reconstruction values y_k we have to start at the beginning, take the derivative of the original formulation of D.

$$D=\sum_{k=1}^{M}\int_{b_{k-1}}^{b_k}(x-y_k)^2\,p(x)\,dx$$

Here we have the pdf p(x) and the reconstruction values (codewords) y_k. Now we start with taking the derivative with respect to y_k and set it to 0:

$$0=\frac{\partial D}{\partial y_k}=-\sum_{k=1}^{M}\int_{b_{k-1}}^{b_k}2\cdot(x-y_k)\,p(x)\,dx$$

Since the y_k is only in 1 interval, the sum disappears:

$$0=\frac{\partial D}{\partial y_k}=-\int_{b_{k-1}}^{b_k}2\cdot(x-y_k)\,p(x)\,dx$$

Since we have a sum, we can split this integral in 2 parts (and remove the - sign):

$$0 = \int_{b_{k-1}}^{b_k} 2 \cdot x \, p(x) \, dx - \int_{b_{k-1}}^{b_k} 2 \cdot y_k \, p(x) \, dx =$$

$$= \int_{b_{k-1}}^{b_k} x \cdot p(x) \, dx - y_k \cdot \int_{b_{k-1}}^{b_k} p(x) \, dx$$

Hence we get the result

$$y_k = \frac{\int_{b_{k-1}}^{b_k} x \cdot p(x) \, dx}{\int_{b_{k-1}}^{b_k} p(x) \, dx}$$

Observe that we now got a result without making any assumptions on p(x).

This can be interpreted as a **conditional expectation** of our signal value over the quantization interval (given the signal is in this interval), or also its "**centroid**" as reconstruction value (codeword). The value in the numerator can be seen as the expectation value of our signal in the interval, and the denominator can be seen as the probability of that signal being in that interval!

Since the b_k depend on the y_k, and the y_k in turn depend on the b_k, we need to come up with a way to compute them. The approach for this is an **iterative algorithm**.

This could look like the following:

1) Start (initialize the iteration) with a **random** assignment of M **reconstruction values** (codewords) y_k

2) Using the reconstruction values y_k , **compute** the **boundary values** b_k as mid-points between 2 reconstruction values / codewords (**nearest neighbour rule**)

3) Using the pdf of our signal and the boundary values b_k , **compute new reconstruction values (codewords)** y_k as **centroids** or **conditional expectation** over the quantisation areas between b_k and b_k-1

4) Go to 2) until update is sufficiently small (< epsilon)

This algorithm usually converges (it finds an equilibrium and doesn't change anymore), and it results in the minimum distortion D.

Hint for numerical integration: An integral of the form

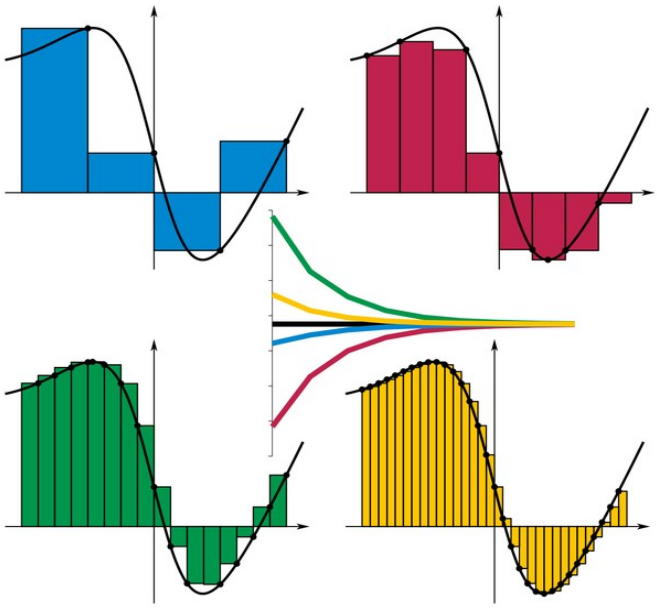
$$I(k) = \int_{b_{k-1}}^{b_k} f(x)\,dx$$

can be approximated in Matlab using a sum. For instance we could divide the integration interval into 100 steps in Matlab:

dx=(b(k)-b(k-1))/100;

An interval at some point x' can then be seen as a small rectangle with area $f(x')\cdot dx$. The

integral is then approximated as the sum of the areas of all these rectangles, as illustrated in following image:



(from: http://en.wikipedia.org/wiki/Integral)

The vector of all function values in the interval b(k-1) to b(k) with spacing dx in Matlab is:

f(b(k-1):dx:b(k))

Hence the integral becomes

I(k)=sum(f(b(k-1):dx:b(k))*dx);

Example 1 for Max-Lloyd Iteration

Assume we have a signal x between $0 \le x \le 1$, uniformly distributed (p(x)=1 on this interval) and we want to have 2 reconstruction values/ codewords y_k , and hence 3 boundaries b_k (where $b_0 = 0$ and $b_2 = 1$).

1) **Random initialization**: $y_1 = 0.3$, $y_2 = 0.8$

2) **Nearest neighbour**: $b_1=(0.3+0.8)/2=0.55$

3) **Conditional expectation**:

$$y_k=\frac{\int_{b_{k-1}}^{b_k} x\cdot p(x)\,dx}{\int_{b_{k-1}}^{b_k} p(x)\,dx}$$

now we use that $p(x)=1$:

$$y_1=\frac{\int_0^{0.55} x\,dx}{\int_0^{0.55} 1\,dx}=\frac{0.55^2/2}{0.55}=0.275$$

$$y_2=\frac{\int_{0.55}^{1} x\,dx}{\int_{0.55}^{1} 1\,dx}=\frac{1/2-0.55^2/2}{1-0.55}=0.775$$

4) Go to 2), **nearest neighbour**:

$b_1=(0.275+0.775)/2=0.525$

3) **Conditional expectation**:

$$y_1=\frac{0.525^2/2}{0.525}=0.26250$$

$$y_2=\frac{1/2-0.525^2/2}{1-0.525}=0.76250$$

and so on until it doesn't change much any more. This should converge to $y_1=0.25$, $y_2=0.75$, and $b_1=0.5$.

Example 2: Like above, but now with a **non-uniform**, Laplacian pdf:

$$p(x)=e^{-0.5\cdot|x|}$$

1) **Random initialization**: $y_1=0.3$, $y_2=0.8$

2) **Nearest neighbour**: $b_1=(0.3+0.8)/2=0.55$

3) **Conditional expectation**:

$$y_k=\frac{\int_{b_{k-1}}^{b_k} x\cdot p(x)\,dx}{\int_{b_{k-1}}^{b_k} p(x)\,dx}$$

Now we need Octave/Matlab to compute the numerator integral, for y_1 :

$$\int_0^{b_1} x\cdot p(x)\,dx=\int_0^{0.55} x\cdot e^{-0.5\cdot|(x)|}\,dx$$

In Octave we can use the function “quad” for integration (type “help quad” in Octave to get information about its use),

```
Num=quad(inline("x*exp(-0.5*abs(x))"),0,0.55)
Num =  0.12618
```

For the denominator integral we get,

$\int_0^{0.55} p(x)\,dx$, hence

```
Den=quad(inline("exp(-0.5*abs(x))"),0,0.55)
Den =  0.48086
```

and hence we obtain,

$$y_1 = \frac{Num}{Den} = \frac{0.12618}{0.48086} = 0.26240$$

For y_2 we get,

```
Num=quad(inline("x*exp(-0.5*abs(x))"),0.55,1)
Num =  0.23463
Den=quad(inline("exp(-0.5*abs(x))"),0.55,1)
Den =  0.30608
Num/Den
ans =  0.76657
```

Hence $y_2 = 0.76657$.

Go back from here to step 2 until convergence.

Here you can see that it can be worthwhile to encapsulate the iteration in a function. Then you can call the function again and again until it doesn't change anymore.

Digital Signal Processing 2/ Advanced Digital Signal Processing
Lecture 5,
Vector Quantizer, LBG

Gerald Schuller, TU Ilmenau

Remember, The Lloyd-Max iteration could look like the following:

1) Start (initialize the iteration) with a random assignment of M reconstruction values (codewords) y_k
2) Using the reconstruction values, compute the boundary values b_k as mid-points between 2 reconstruction values / codewords (**nearest neighbour rule**)
3) Using the pdf of our signal and the boundary values, compute new reconstruction values (codewords) y_k as centroids over the quantisation areas (**conditional expectation/centroid**)
4) Go to 2) until update is sufficiently small (< epsilon)

This algorithm usually converges (it finds an equilibrium and doesn't change anymore), and it results in the minimum distortion D.

It is interesting that this can be generalized to the multidimensional case, to the so-called Vector Quantisation.

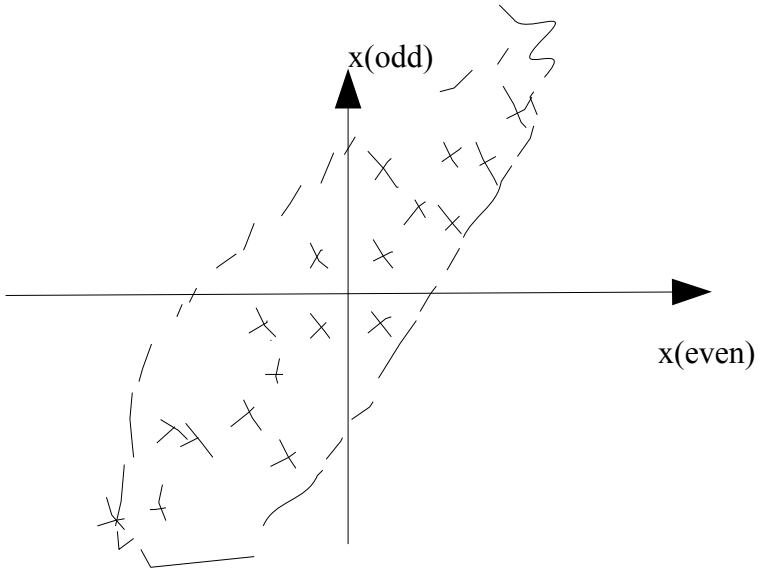
Vector Quantisation

Scalar quantisation usually makes the assumption that the signal to quantize, the source, is **memoryless**, which means each sample is statistically independent of any other sample in the sequence. This can be seen as having no “memory” between the samples. Examples of these sources might be: Thermal noise, white noise, or a sequence of dice tosses, lottery numbers.
But many signals do **have memory**, they have samples which are statistically dependent on other samples in the sequence. Example are: Speech signals, pink noise (noise which has a non-flat spectrum), temperature values over the year, image signals, audio signals.
Since many signals of interest indeed have memory, this suggests that we can do a better job. One possible approach to deal with memory (statistical dependencies) in our signal, is to use the so-called **Vector Quantisation (VQ)**.
A possible reference is: “Introduction to Data Compression”, Section about Vector Quantisation, by Khalid Sayood, Morgan Kaufmann publishers.
How does VQ work? Instead of quantizing each scalar value (each sample) individually, we first group the sequence of samples x(n) into groups of N samples:

[x(0),...,x(N-1)],[x(N),...,X(2N-1)],[x(2N),..,x(3N-1)],...
In this way we obtain a sequence of blocks (also called **vectors**) of size **N samples** each. In this way we obtain a sequence of samples in an **N-dimensional** space. In such a way we can capture or use the memory between samples within each block or vector. The resulting samples with memory in the N-dimensional space will then lie on or near a hyperplane or subspace within this N-dimensional space. Hence we don't need to sample the entire space, but we only need to sample the part of our space where our samples are actually located.
Example: Take correlated samples, such that one sample is always similar to the previous sample, just like in a sequence of speech or audio samples (usually we don't have very high frequencies there, and that means the curve through the samples is more or less smooth). Now take the dimension N=2. Then we obtain a 2-dimensional vector space of samples, which could look like in the following diagram. The resulting sequence of vectors is:
[x(0),x(1)],[x(2),x(3)],[x(4),x(5)],...

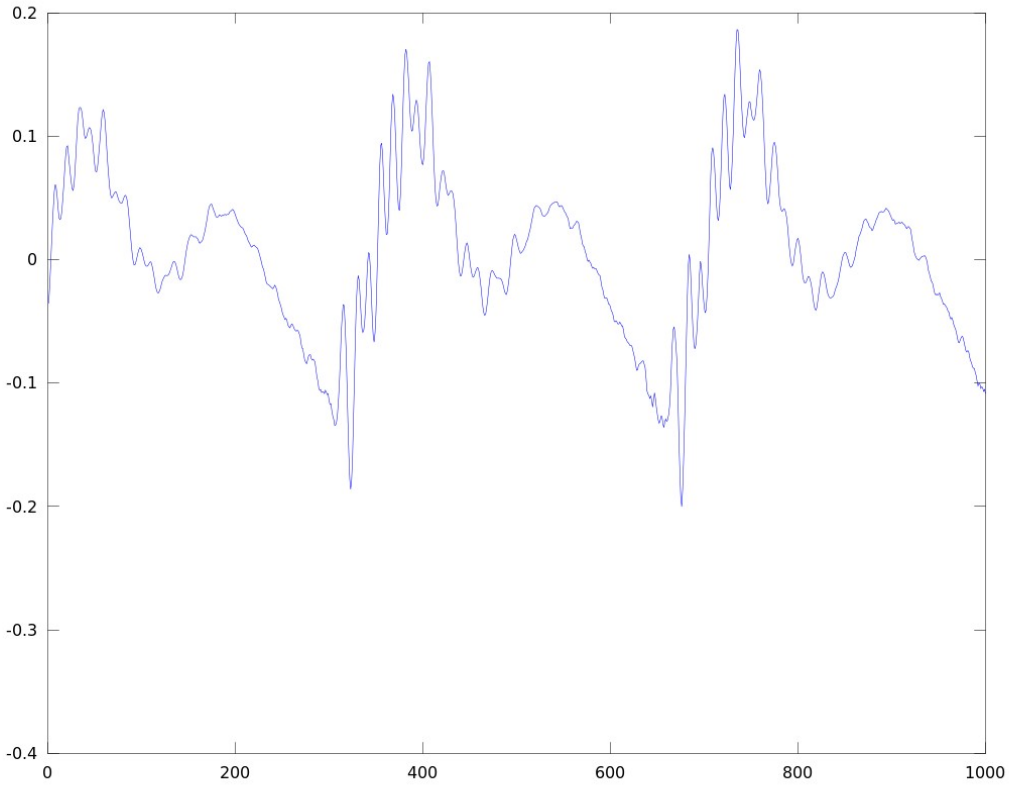
For instance, our signal could be:
x=[23,45,21,4,-23,-4], then we get the following **sequence of vectors:**
[23,45]; [21,4]; [-23,-4]

Each of those vectors can now be represented as a point in this 2-dimensional space:

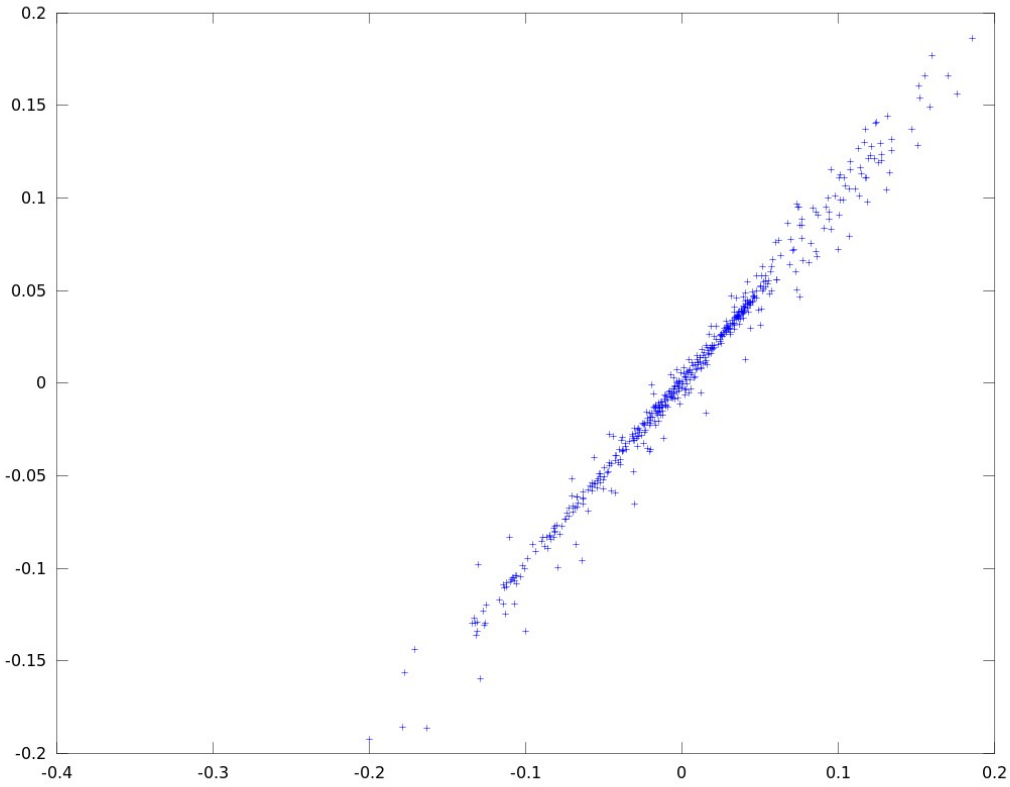


Matlab/Octave Example:

```
%Take a speech signal and read it into
%Matlab/Octave:
sp=wavread('mspeech.wav');
%Take an excerpt of 1000 samples, starting at
%sample 2001 and plot it:
spex=sp(2000+(1:1000));
plot(spex)
```



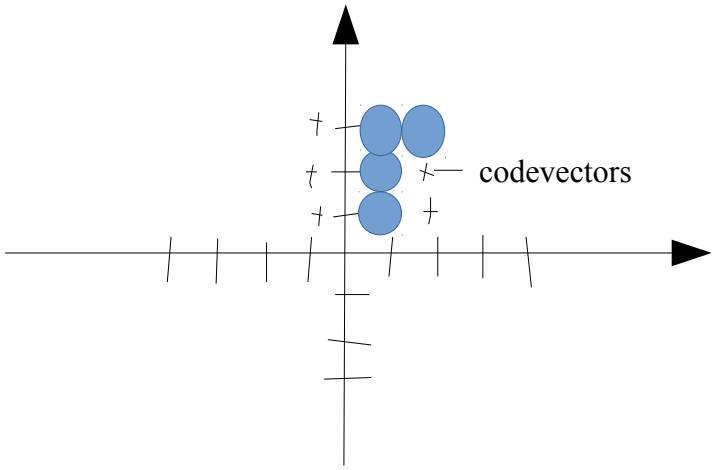
%Now plot the 2 dimensional vectors resulting
%from groups of even and odd samples:
plot(spex(2:2:end),spex(1:2:end),'+')



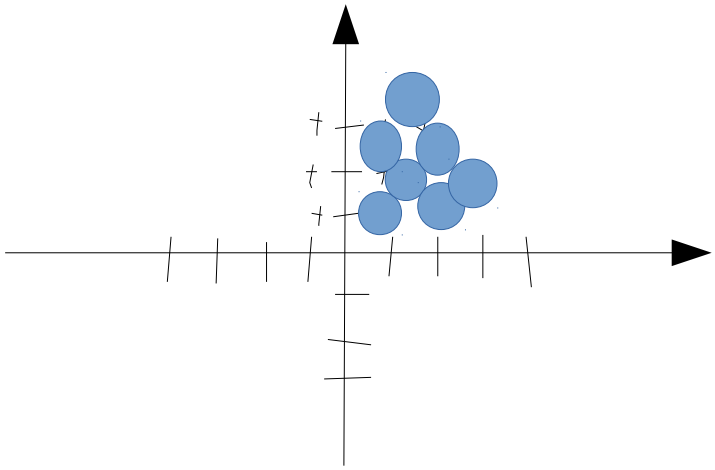
We can see: Since the odd and the even samples are similar to each other, we get a distribution of vector points near the diagonal of the space! Hence we also only need to sample this space only near the diagonal, or more generally speaking, we should sample more densely near this diagonal.

An interesting property is that vector quantizers not only give an advantage for signals with memory, but also for signals without memory. In the scalar case, we can only sample an N-dimensional space on a regular grid, which is given by the coordinate axis of this space, whereas with VQ we can use something like a

densest sphere packing in this N-dimensional space, such that we reduce the distance between reconstruction vectors (the so-called **codewords**), and hence reduce the Expectation of the quantisation error even in this case.
The following image illustrates the densest sphere packing for the case of memoryless signals:
The case of a scalar quantizer:



The case of a vector quantizer with N=2:



Observe that in this way we get a denser “packing”, shifting the spheres in the gaps of the neighbouring layers, which results in a reduced reconstruction error.

How do we do the **quantisation in the N-dimensional case** in general? We choose N-dimensional reconstruction values, which we now call codewords. We use the nearest neighbour rule to map each N-dimensional **signal vector to** the **nearest codevector**. You could think of the neighbourhood as n-dimensional spheres around each codevector. Each codevector has an index and this index is then transmitted to the receiver, which uses this codevector as a reconstruction value. The collection of all codewords is called a **codebook**. The size of the codebook also determines how many bits are needed for their index. Usually codebooks are fixed, pre-defined, but there are also adaptive codebooks, for instance in speech coding.

So how do we **obtain our codebook**, our codevectors? Basically like in the Lloyd-Max case, we are just extending it to the N-dimensional case. For N-dimensional case this is called the Linde-Buzo-Gray (LBG) Algorithm (see also the Book Introduction to Data Compression):

This could look like the following:

1) Start (**initialize** the iteration) with a **random** assignment of M N-dimensional **codewords**, $\boldsymbol{y}_k$ (bold-face to indicate a vector)

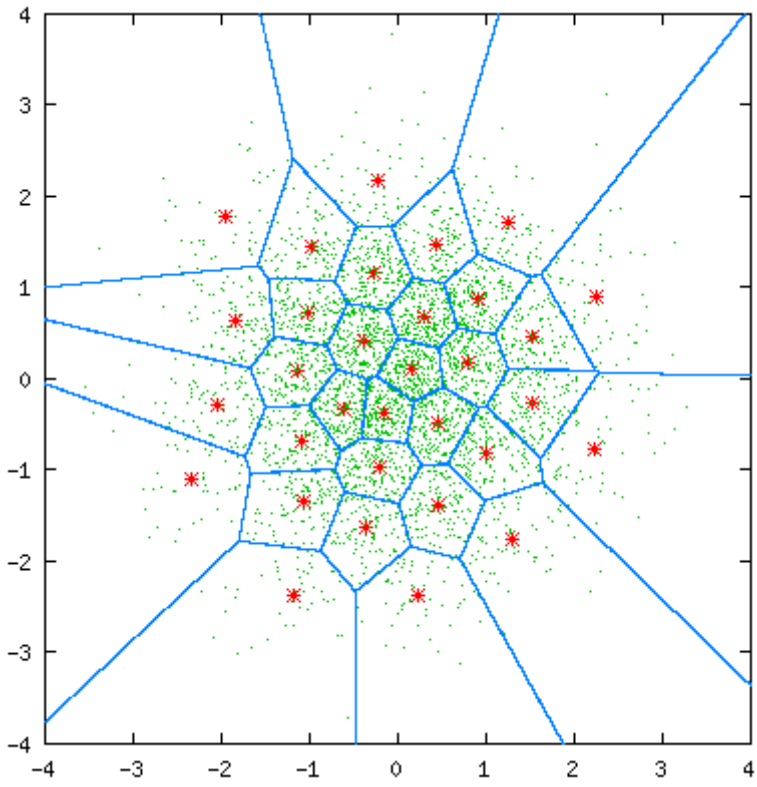
2)Using the codewords, compute the **decision boundary** $\boldsymbol{b}_k$ (bold face to indicate that a line or hyper-plane) as the set of **all** points with equal distance between 2 reconstruction values / codewords (the such constructed regions are also called **Voronoi-regions**), using the **nearest neighbour** rule. To assign a vector to a specific region, we use the nearest neighbour rule directly. We simply test which codeword is closest to the observed vector.
3) Using the pdf of our signal and the decision boundary (Voronoi region), compute new **codewords** y_k as **centroids (center of mass)** or **conditional expectation** over the quantisation areas (the Voronoi region). The same as in 1 dimension, just here the integral is going over N dimension
4) Go to 2) until update is sufficiently small (< epsilon)

Here we assume we have a pdf of the signal. Observe that here we would need a multi dimensional pdf, which is difficult to obtain. Hence, instead of a multi dimensional pdf, we often only have a so-called **training set** to obtain our codebook. The training set is a set of signals which have statistics like our targeted signals, but which are **only** used to “train” (using LBG) our codebook vectors. To **test** resulting vector quantizer, we should use signals which are **not** in the training set.

We can still use this same algorithm as with the pdf, we just have to compute the centroid or conditional expectation differently. Assume we have L samples in our Voronoi region, and want to compute the centroid of this Voronoi region. We could do that by assigning each signal vector **x**(k) of the training set to a probability of 1/L (we assume each vector is equally likely), and then just use the above formula for the centroid, replacing the integrals by sums. This then results in

$$\boldsymbol{y}_k = \frac{\sum_{i \in Voronoi\,region\,k} \boldsymbol{x}(i)}{Number\ of\ signal\ vectors \in Voronoi\ region}$$

This is simply the average of all observed training vectors in our Voronoi region. After training the codebook with the training set (only for training or learning to obtain the codebook), we have a (fixed) codebook which we can then use for encoding our data. Observe that the training set should be different from our data.
The resulting Vector Quantizer could look like in following image for dimension N=2, where the blue lines are the boundaries of the Voronoi regions (consisting of our $\boldsymbol{b}_k$), the red stars are the codevectors $\boldsymbol{y}_k$, and the green dots the signal vectors:



(From: http://www.data-compression.com/vq.html)

Example. Determine the codebook vectors of a LBG vector quantizer for dimension N=2, and number of codevectors M=2, after one iteration for two vectors with the given training set **x** = [3,2,4,5,7,8,8,9]. Initial codebook vectors are $\mathbf{y_1}$=[1,2], $\mathbf{y_2}$ =[5,6].
The training set vectors are hence: [3,2], [4,5], [7,8], [8,9].

The solution follows the algorithm which was explained in the lecture.
1) We start with the given **randomly assigned codebook vectors** $\mathbf{y_k}$.
2)The next step is to calculate **decision boundary** $\mathbf{b_k}$ using the **nearest neighbour** rule to obtain the Voronoi region.. This results in a set of midpoints. These mid-points then form the Voronoi boundary line, which is perpendicular to the connecting line of the 2 codewords. The direct mid-point is

$$b_k = \frac{y_1 + y_2}{2} \quad .$$

$\mathbf{b_k}$ in our case is a point of the line going through the point [3,4], the line consisting of **all points** which have **equal distance** to the neighboring codevectors. Now we can draw the given training set vectors, codebook vectors, and the Voronoi boundaries. Observe that the computation of the **boundary line** of the mid-points is useful **only for drawing** this picture, but in a computational

implementation we don't need to compute it, but use the nearest neighbour rule directly.

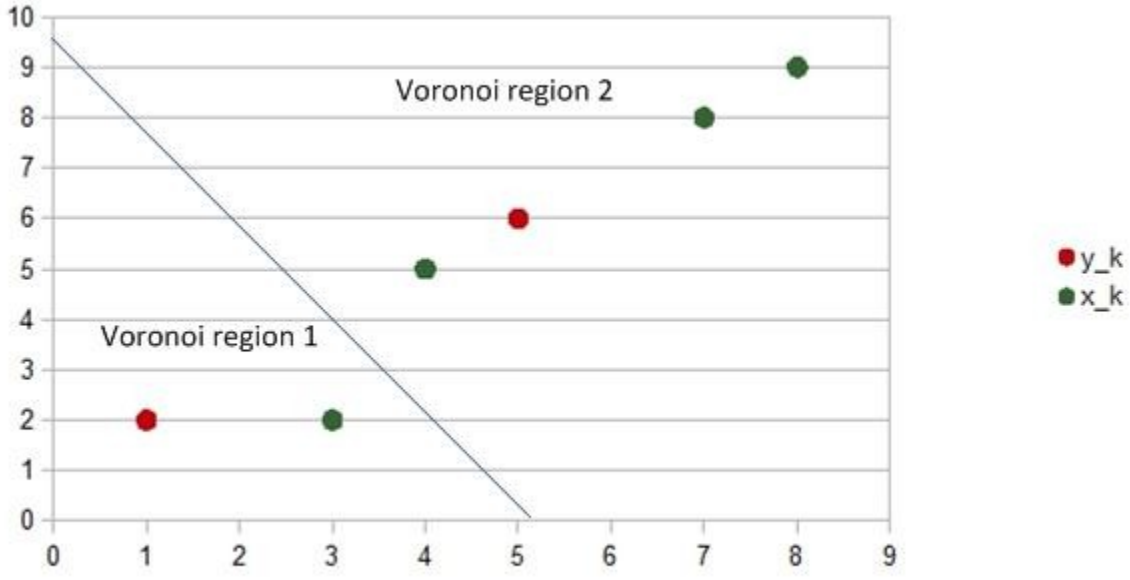


Fig. 1: Illustration of LBG.

3) The next step is to compute the new **codewords y_k** as a **centroid** or **conditional expectation** over a quantization area or Voronoi region. In order to do that, the formula for the trainings set from the lecture should be used:

$$y_k = \frac{\sum_{i \in Voronoi\ region\ k} x(i)}{Number\ of\ signal\ vectors \in Voronoi\ region}$$

In order to find out in **which Voronoi region** a vector is located, we use the nearest neighbour rule. For that we need to calculate the Euclidean **distances** between all the **training set vectors** and **codebook vectors** and then decide which

training vectors are closer to which codebook vector.
For each trainings set vector we compute to which codebook vector it has the closest distance, with the Euclidean distances calculated in the following way:

- For trainigs set vector $\mathbf{x_1}$=[3,2]:
  - distance to codebook vector $\mathbf{y_1}$=[1,2]:
  - $d_1=\sqrt{(3-1)^2+(2-2)^2}=\sqrt{4}$
  - distance to codebook vector $\mathbf{y_2}$ =[5,6]:
  - $d_2=\sqrt{(5-3)^2+(6-2)^2}=\sqrt{20}$
  - hence $\mathbf{y_1}$ is closer
- For trainigs set vector $\mathbf{x_2}$=[4,5]
  - $d_1=\sqrt{(4-1)^2+(5-2)^2}=\sqrt{18}$
  - $d_2=\sqrt{(5-4)^2+(6-5)^2}=\sqrt{2}$
  - Hence $\mathbf{y_2}$ is closer
- For trainigs set vector $\mathbf{x_3}$=[7,8]
  - $d_1=\sqrt{(7-1)^2+(8-2)^2}=\sqrt{72}$
  - $d_2=\sqrt{(7-5)^2+(8-6)^2}=\sqrt{8}$
  - Hence $\mathbf{y_2}$ is closer
- For trainigs set vector $\mathbf{x_4}$=[8,9]
  - $d_1=\sqrt{(8-1)^2+(9-2)^2}=\sqrt{98}$
  - $d_2=\sqrt{(8-5)^2+(9-6)^2}=\sqrt{18}$
  - Hence $\mathbf{y_2}$ is closer

Now we can compute the **centroid or conditional expectation** for each of the 2 Voronoi regions.

- **Voronoi region 1** only contains trainings set vector $\mathbf{x_1}$=[3,2], hence its centroid and new codebook vector is identical to $\mathbf{x_1}$ and we get the **new codebook vector 1** as

$$y_1 = [3,2]$$

- **Voronoi region 2** contains the remaining 3 vectors. We obtain its centroid by averaging over them, and obtain the **new codebook verctor 2** as

$$y_2 = [\frac{4+7+8}{3}, \frac{5+8+9}{3}] = [6+\frac{1}{3}, 7+\frac{1}{3}]$$

So the updated codebook vectors will look in the following way:

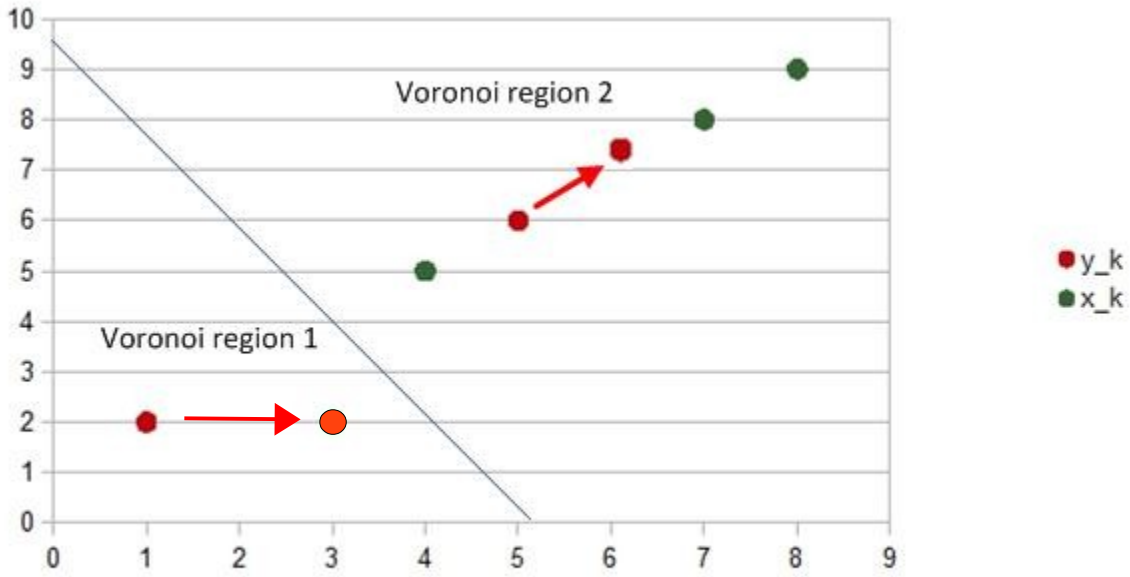


Fig. 2: Updated quantization areas.

4) Go back to step 2) and repeat the procedure until the result does not change much anymore.

Vector Quantization in an Encoder and Decoder

-In an Encoder we first convert our signal sample stream to our vectors.
-Then we map those vectors to the nearest code vectors.
-We transmit the indices of those codevectors to the decoder
-The decoder converts the indices back to the codevectors
-The codevectors are then concatenated and converted back into a stream of samples.

Example:

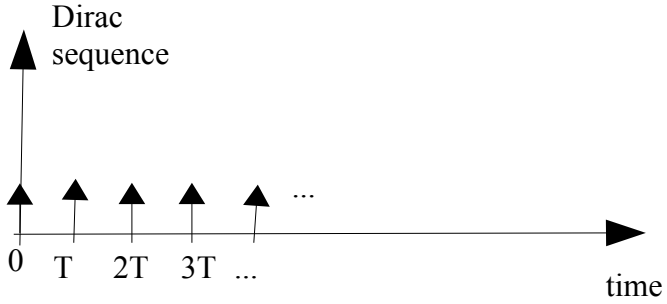
Encoder:
stream to vectors:
x: [3,4],[7,8],...
Vectors to codevectors:
y: [4,5],[6.7]
to indices:
k: 4, 5
Decoder:
Indices to vectors:
y: [4,5],[6,7]
to stream of samples:
xrek: 4,5,6,7

Digital Signal Processing 2/ Advanced Digital Signal Processing
Lecture 6,
Sampling, z-Transform

Gerald Schuller, TU Ilmenau

Sampling, Downsampling, Upsampling
Sampling the analog signal, normalized frequency

To see what happens when we sample a signal, lets start with the analog signal s(t). Sampling it means to sample the signal at sample intervals T (example: 1/8000 s in the ISDN example), or the sampling frequency f_s (8kHz in the ISDN example). Mathematically, sampling can be formulated as multiplying the signal with a Dirac impulse at the sampling time instances nT, where n is the number of our sample (n=0,... for causal systems).



The sequence or train of Dirac impulses can be written as:

$$\Delta_T(t) := \sum_{n=0}^{\infty} \delta(t - nT)$$

Remember that the integral over a Dirac impulse is 1:

$$\int_{t=-\infty}^{\infty} \delta(t)\,dt = 1$$

(see also: http://en.wikipedia.org/wiki/Dirac_delta_function)

The integral over the product of a function with a Dirac impulse is the value of the function at the position of the impulse:

$$\int_{t=-\infty}^{\infty} s(t)\,\delta(t - nT)\,dt = s(nT)$$

This is the mathematical formulation of sampling at time-point nT. We can now use this description to compute the resulting **spectrum** with the Fourier transform.
First, if the look at the Fourier transform of the analog system, we get

$$S^c(\omega) = \int_{t=-\infty}^{\infty} s(t) \cdot e^{-j\omega t}\,dt$$

where the superscript c denote the continuous version, with $\omega = 2\pi f$ the angular frequency. If we now compute the Fourier Transform for the sampled signal $s(t) \cdot \Delta_T(t)$, with the replacement

$$s(t) \leftarrow s(t) \cdot \Delta_T(t)$$

we obtain the following sum

$$S^d(\omega) = \sum_{n=-\infty}^{\infty} s(nT) \cdot e^{-j\omega nT}$$

with the superscript d now denoting the discrete time version. Now we can see that the frequency variable only appears as ωnT , and T is the inverse of the sampling frequency. Hence we get

$$\omega T = \omega / f_s =: \Omega$$

This is now our **normalized frequency**, it is without a physical unit, since the unit Hertz in ω and f_s cancel. In the normalized frequency 2π represents the sampling frequency and π is the so called **Nyquist frequency** (the upper limit of our usable frequency range, defined as **half** the sampling frequency). Observe that we use the capital Ω to signify that this is the normalized frequency. The capitalized version is commonly used to distinguish it from the continuous, non-normalized, version, if both are used. Otherwise, also the small ω is used in the literature, for the normalized frequency, if that is all we need, if there is no danger of confusion, as we did before. To indicate that we are now in the discrete domain, we rename our signal to x(n)=s(nT). Its spectrum or frequency response is then

$$X(\Omega) = \sum_{n=-\infty}^{\infty} x(n) e^{-j\Omega n}$$

Because n is integer here (no longer real valued like t), we get a 2π **periodicity** for $X(\Omega)$. This is

the first important property for discrete time signals. The above transform is called the Discrete Time Fourier transform or **DTFT** (not the Discrete Fourier Transform, which is for finite or periodic discrete time sequences. Here we still have an infinite “block length”).
Also observe that for **real valued** signals, the spectrum of the negative frequencies is the conjugate complex of the positive frequencies,

$$X(-\Omega)=X^{*}(\Omega)$$

where * denotes the conjugate complex operation, because $e^{-j(-\Omega)n}=(e^{-j\Omega n})^{*}$.
(we also have to use that the conjugate complex of a real valued signal does not change the signal).
This also means that for real valued signals we only need to look at the frequency range between 0 and π, since the negative frequencies are conjugate symmetric (and because of the 2pi periodicity the range between -pi and pi is sufficient to look at). This is also again what the Nyquist Theorem tells us, that the frequencies between 0 and π (the Nyquist frequency) are sufficient to completely describe a (real valued) signal.

Sampling a discrete time signal

So what happens if we further downsample an already discrete signal x(n), to reduce its sampling rate? **Downsampling by N** means we only keep every Nth sample and discard every

sample in between. Observe that this results in a **normalized frequency which is a factor of N higher**.
This downsampling process can also be seen as first multiplying the signal with a sequence of **unit pulses** (a 1 at each sample position), zeros in between, and later dropping the zeros. This multiplication with the unit pulse train can now be used to mathematically analyse this downsampling, looking at the resulting spectra, first still including the zeros. The frequency response now becomes

$$X^d(\Omega) = \sum_{n=mN} x(n) e^{-j\Omega n}$$

$$= \sum_{m=-\infty}^{\infty} x(mN) e^{-j\Omega mN}$$

for all integers m.

Again we can write down-sampling as a multiplication of the signal with a sampling function. In continuous time it was the sequence of Dirac impulses, here it is a **sequence of unit pulses** at positions of multiples of N,

$$\Delta_N(n) = \begin{pmatrix} 1, & if \quad n = mN \\ 0, & else \end{pmatrix}$$

Then the sampled signal, with the zeros still in it, becomes

$$x^d(n) = x(n)\Delta_N(n)$$

This signal is now an intermediate signal, which gets the zeros removed before transmission or storage, to reduce the needed data rate. The decoder upsamples it by re-inserting the zeros to obtain the original sampling rate.
Observe that this is then also the signal that we obtain after this upsampling in the decoder. Hence this signal looks interesting to us, because it appears in the encoder and also in the decoder.
What does its **spectrum** or **frequency response** look like?
We saw that the downsampled signal $x^d(n)$ can be written as the **multiplication** of the original signal with the unit pulse train $\Delta_N(n)$. In the spectrum or frequency domain this becomes a **convolution with their Fourier transforms**. The Fourier transform of the unit pulse train is a dirac impulse train. But that is not so easy to derive, so we just apply a simple trick, to make it mathematically more easy and get converging sums. We have a **guess** what the Fourier transform of the unit pulse train is: we get Dirac impulses at frequencies of $\frac{2\pi}{N}k$ (fundamental frequency 2pi/N and its harmonics), with a constant factor that becomes clear later,

$$\delta_{2\pi/N}(\Omega)=\sum_{k=0}^{N-1}\frac{2\pi}{N}\delta\left(\Omega-\frac{2\pi}{N}\cdot k\right)$$

Where $\delta_{2\pi/N}(\Omega)$ is the Fourier transform of the unit impulse train (the Dirac impulse train). Its inverse Fourier transform is

$$\Delta_N(n)=\frac{1}{2\pi}\int_0^{2\pi}\delta_{2\pi/N}(\Omega)e^{jn\Omega}\,d\Omega$$

$$=\frac{1}{2\pi}\int_0^{2\pi}\sum_{k=0}^{N-1}\frac{2\pi}{N}\delta\left(\Omega-\frac{2\pi}{N}\cdot k\right)e^{jn\Omega}\,d\Omega$$

$$=\frac{1}{N}\sum_{k=0}^{N-1}e^{j\frac{2\pi}{N}\cdot k\cdot n}$$

Now we have to prove that this is indeed our unit pulse train. To do that, we can look at the sum. It is a geometric sum (a sum over a constant with the summation index in the exponent),

$S=\sum_{k=0}^{N-1}c^k$, $S\cdot c=\sum_{k=1}^{N}c^k$, $Sc-S=c^N-1$, hence we get

$$S=\frac{c^N-1}{c-1}$$

Here we get $c = e^{j\frac{2\pi}{N}n}$, and the sum can hence be computed in a closed form,

$$\sum_{k=0}^{N-1} e^{j\frac{2\pi}{N}\cdot n\cdot k} = \frac{e^{j\frac{2\pi}{N}\cdot n\cdot N}-1}{e^{j\frac{2\pi}{N}\cdot n}-1}$$

In order to get our unit pulse train, this sum must become N for n=mN. We can check this by simply plugging this into the sum. The right part of the sum is undefined in this case (0/0), but we get an easy result by looking at the left hand side: $e^{j\frac{2\pi}{N}\cdot mN\cdot k} = 1$. Hence the sum becomes N, as desired!
For $n \neq mN$ we need the sum to be zero. Here we can now use the right hand side of our equation. The denominator is unequal zero, and the numerator becomes zero, and hence the sum is indeed zero, as needed! This proves that our **assumption was right**, and we have

$$\Delta_N(n) = \frac{1}{N}\sum_{k=0}^{N-1} e^{j\frac{2\pi}{N}\cdot k\cdot n}$$

Now we can use this expression for the unit pulse train in the time domain and compute the Fourier transform,

$$x^d(n) = x(n) \cdot \Delta_N(n) = x(n) \cdot \frac{1}{N} \sum_{k=0}^{N-1} e^{j\frac{2\pi}{N} \cdot k \cdot n}$$

Taking its Discrete Time Fourier transform now results in

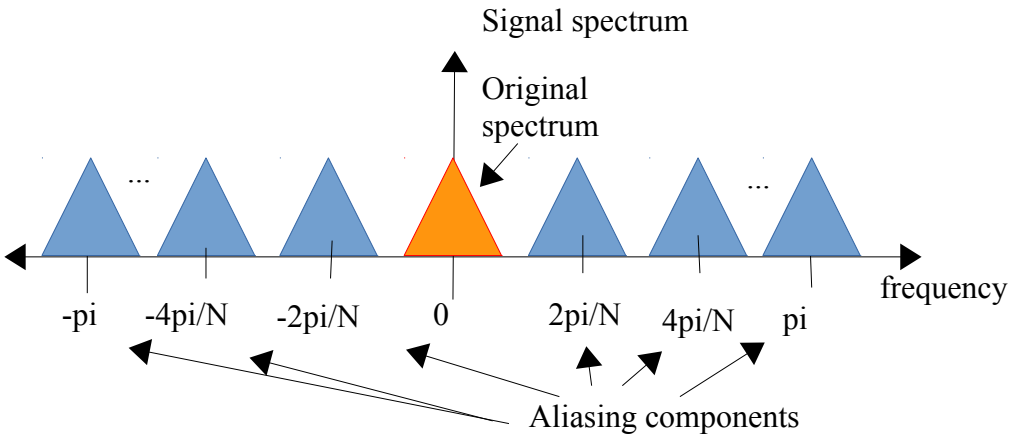
$$X^d(\Omega) = \sum_{n=-\infty}^{\infty} x(n) \cdot \frac{1}{N} \sum_{k=0}^{N-1} e^{j\frac{2\pi}{N} \cdot k \cdot n} \cdot e^{-j\Omega n} =$$

$$= \frac{1}{N} \sum_{k=0}^{N-1} \sum_{n=-\infty}^{\infty} x(n) \cdot e^{-j(-\frac{2\pi}{N} \cdot k + \Omega) \cdot n}$$

$$= \frac{1}{N} \sum_{k=0}^{N-1} X(-\frac{2\pi}{N} k + \Omega) \quad \blacktriangleleft$$

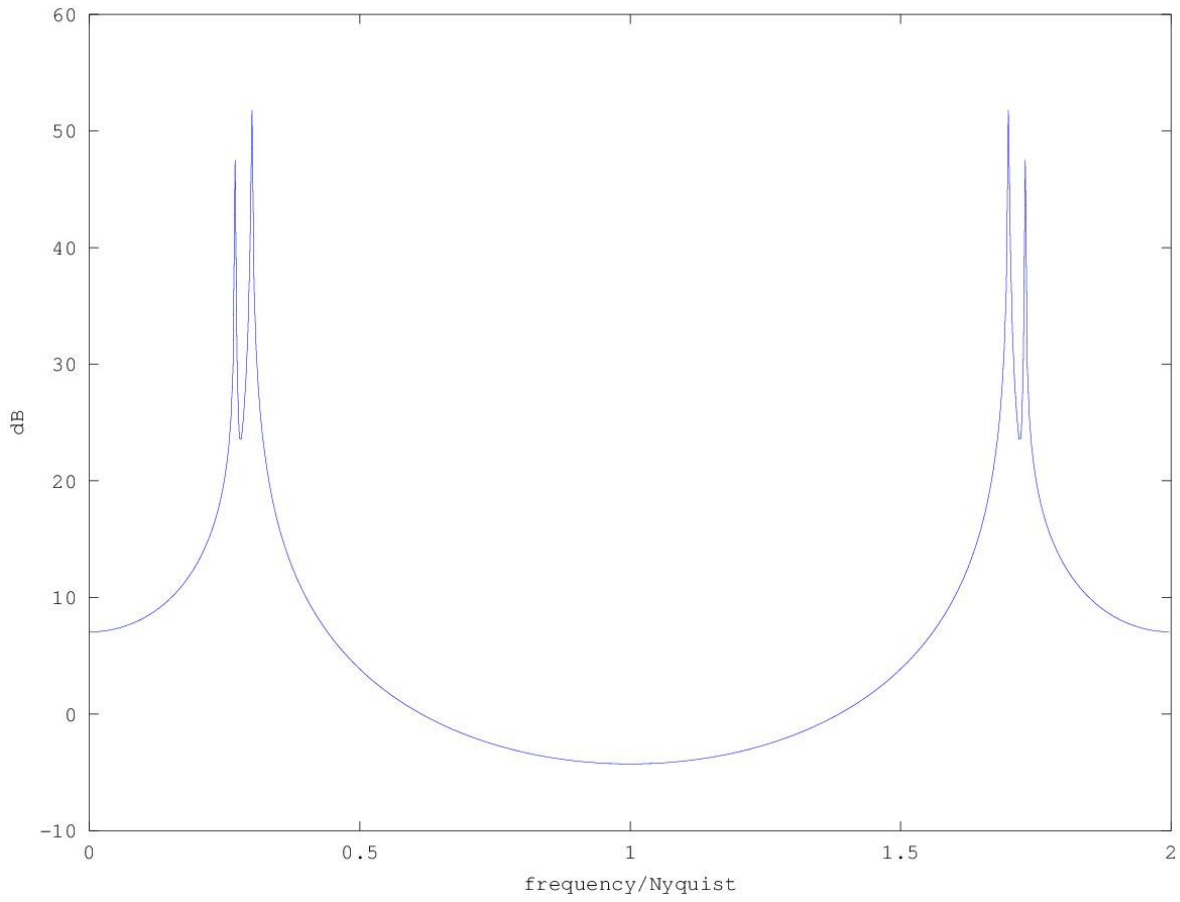
This shows, that sampling, still including the zeros, leads (in the frequency domain) to **multiple shifted versions** of the signal spectrum, the so-called **aliasing components**,

$$X^d(\Omega) = \frac{1}{N} \sum_{k=0}^{N-1} X(\frac{-2\pi}{N} \cdot k + \Omega) \quad \text{(eq.1)}$$



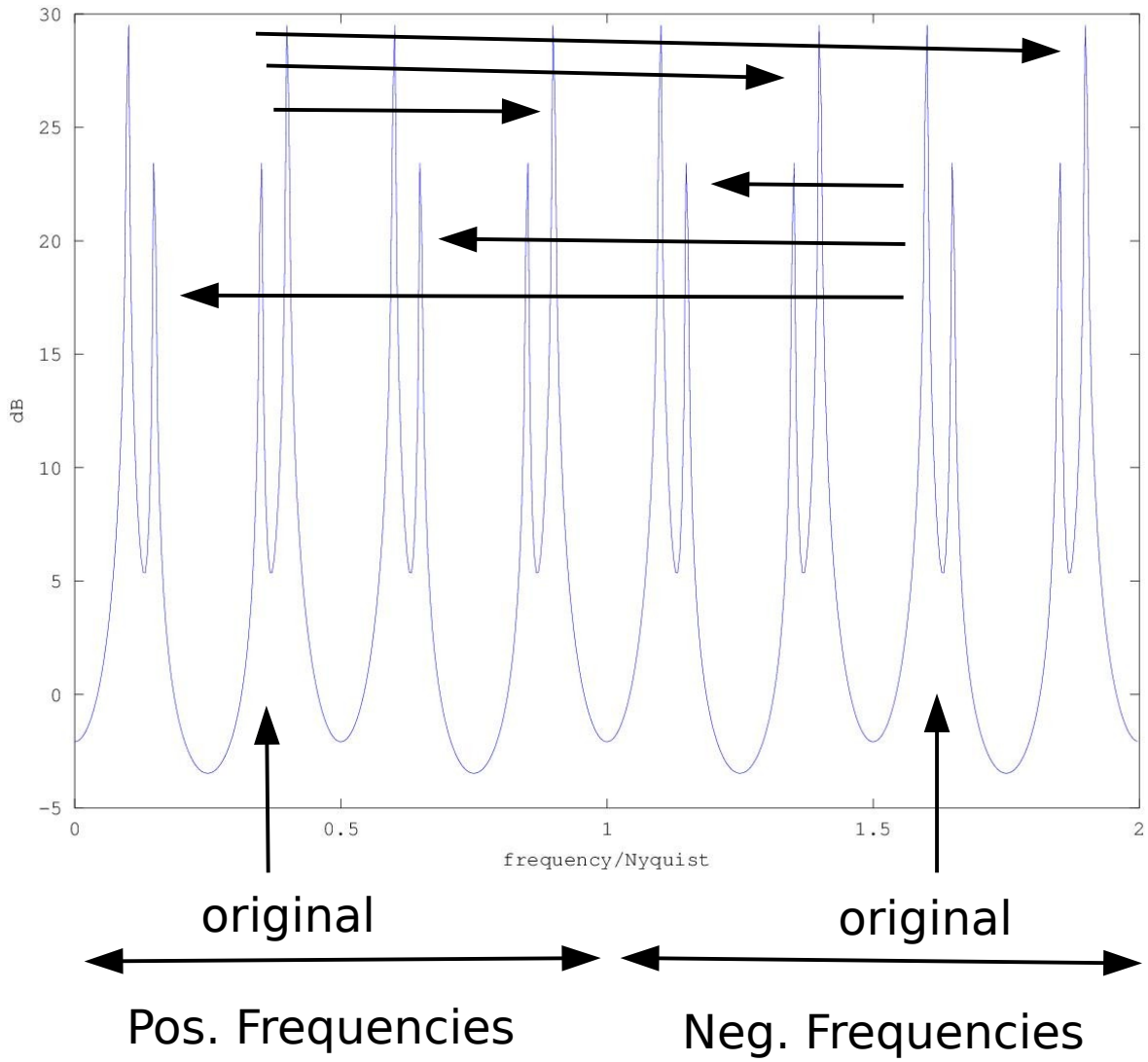
Observe: The spectral components don't overlap if their bandwidths is below $2\pi/N$ for complex signals, or for a real low pass signal, it should be below π/N ! If we want to reconstruct the original signal, hence we need to make sure the aliasing components don't overlap by suitable filtering at the high sampling rate, to prepare for the down-sampling. Observe that the term „Aliasing“ in the literature is sometime only used for overlapping alias components, and sometimes more broadly, like we do here, to mean any additional shifted frequency component.

Next is another example, also including the negative frequencies that now show up above normalized frequency 1 (1 being the Nyquist frequency here), and showing 2 sine signals at different strength at normalized frequencies 0.4 and 0.35. This can also be seen as a narrow band signal, resulting e.g. from a passband filter.



After sampling by a factor of N=4, still including the zeros, we get the following spectrum:

Spectral Copies



The picture shows that the spectrum still contains the **original spectrum**, **plus** the spectral **copies** at frequency shifts of $k \cdot 2 \cdot \pi / N$ from the originals.

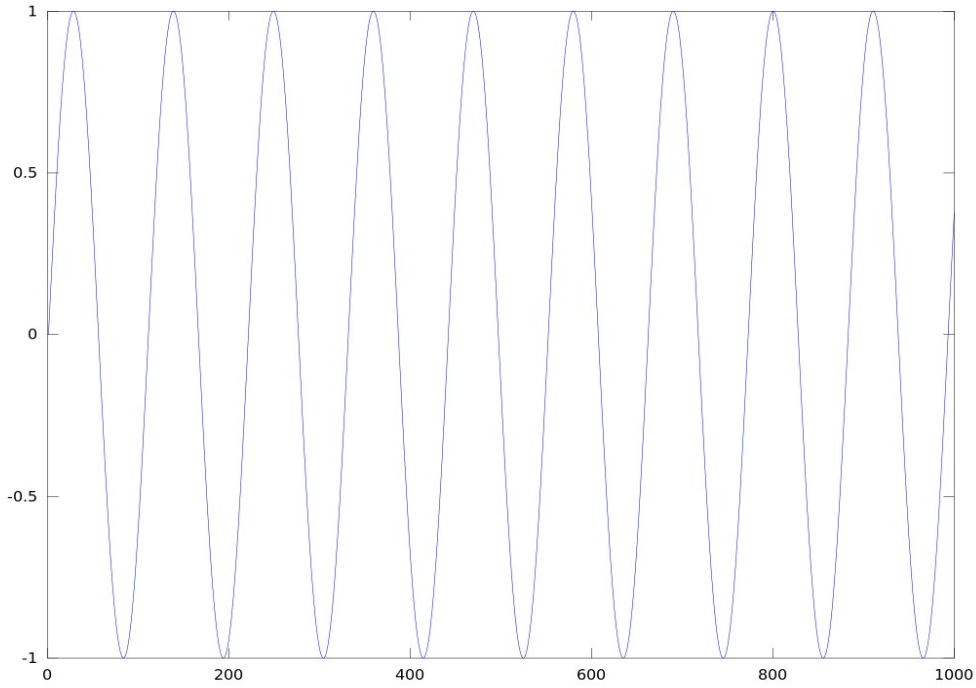
Observe: Since we have a real valued signal (the sinusoids), the spectrum of negative and positive frequencies are **symmetric** around frequency zero. This then leads to the **mirrored appearance** between the neighbouring spectral images or aliasing components.

BTW, Nyquist tells us to sample in such a way, that the shifted spectra of our signal do not overlap. Otherwise, if they overlap, we cannot separate those parts of the spectrum anymore, and we loose information, which we cannot reconstruct.

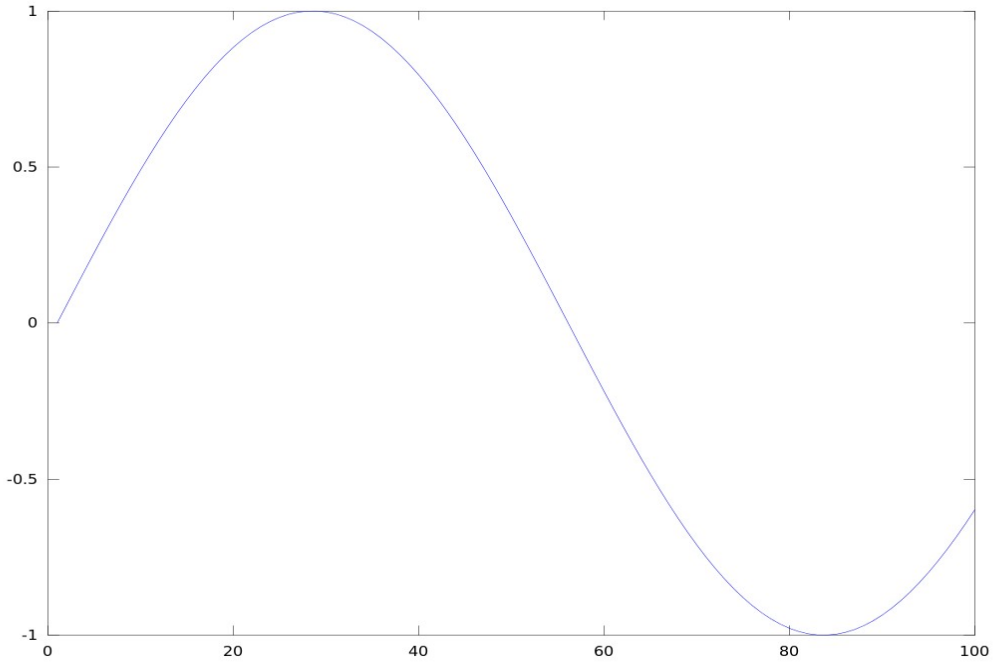
In conclusion: Sampling a signal by a factor of N, with keeping the zeros between the sample points, leads to N-1 aliasing components.

Example:
Make a sine wave which at 44100 Hz sampling rate has a frequency of 400 Hz at 1 second duration. Hence we need 44100 samples, and 400 periods of our sinusoid in this second. Hence we can write our signal in Octave/Matlab as:
*s=sin(2*pi*400*(0:1/44100:1));*

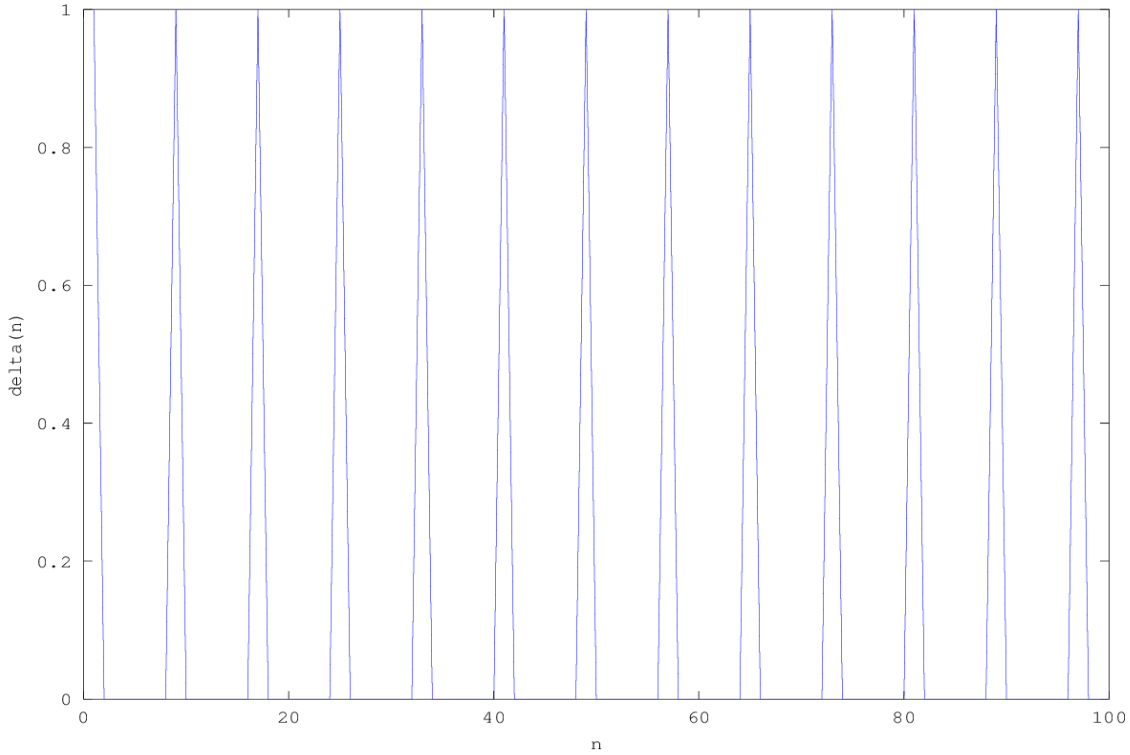
Listen to it:
sound(s,44100)
Now plot the first 1000 samples:
plot(s(1:1000))



Next plot the first 100 samples:
plot(s(1:100))

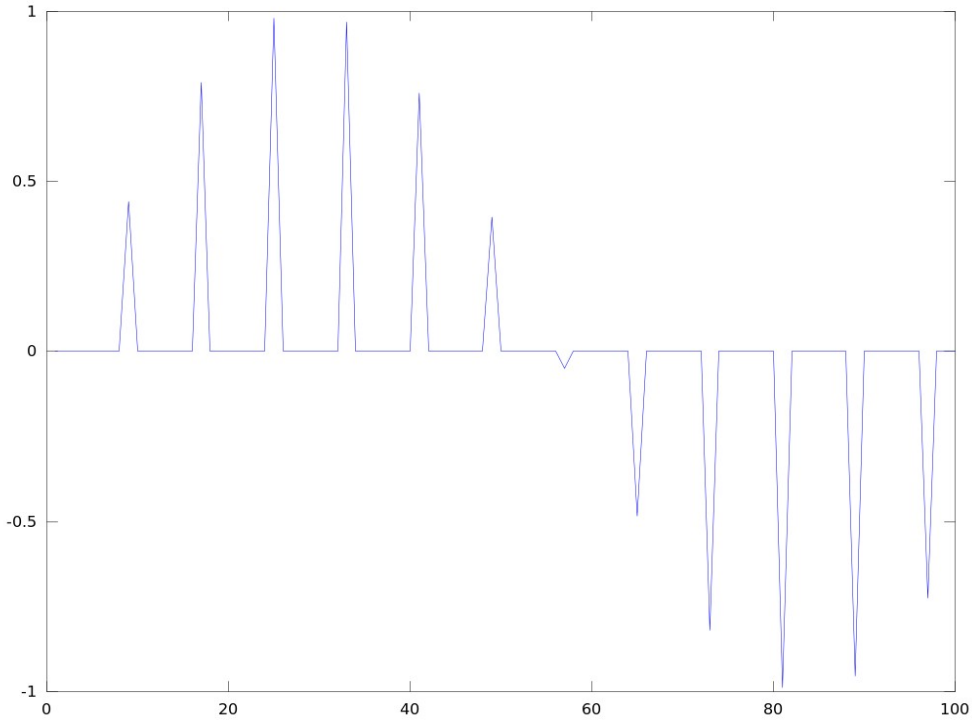


Now we can multiply this sine tone signal with a unit pulse train, with N=8.
We generate the unit impulse train,
unit(1:8:44101)=1;
plot(unit(1:100));
xlabel('n')
ylabel('unit(n)')

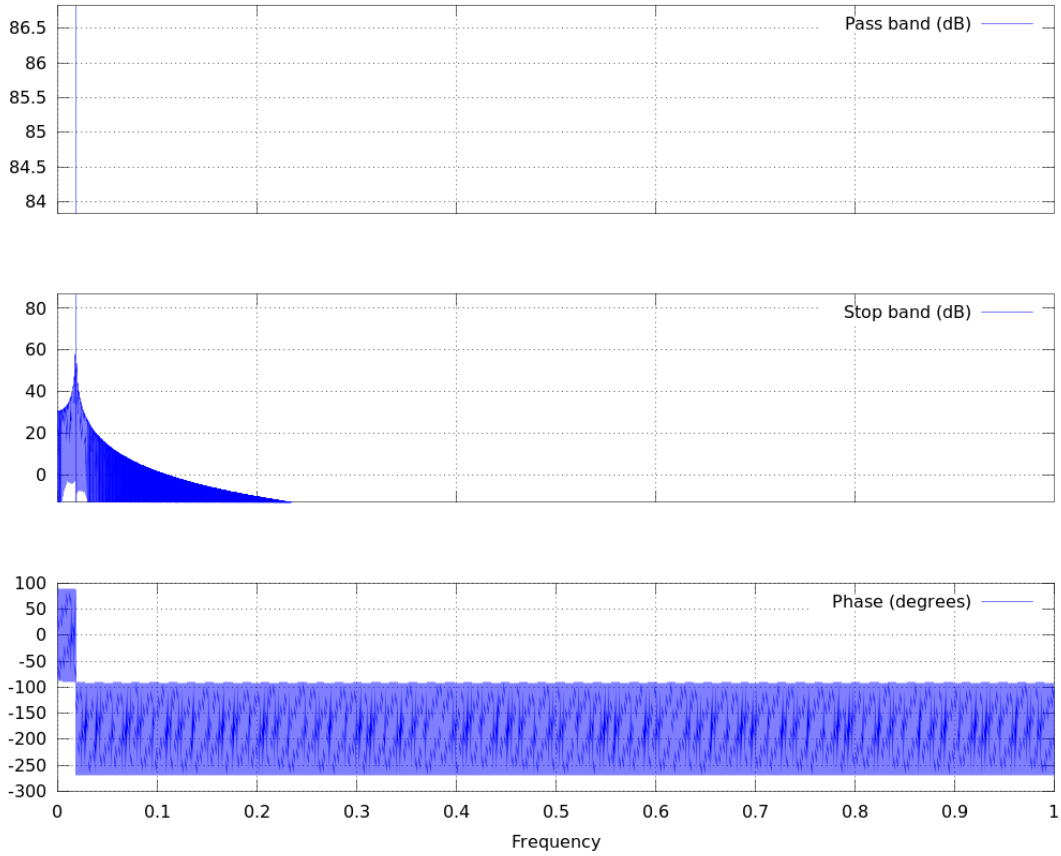


Listen to it:
sound(unit,44100);

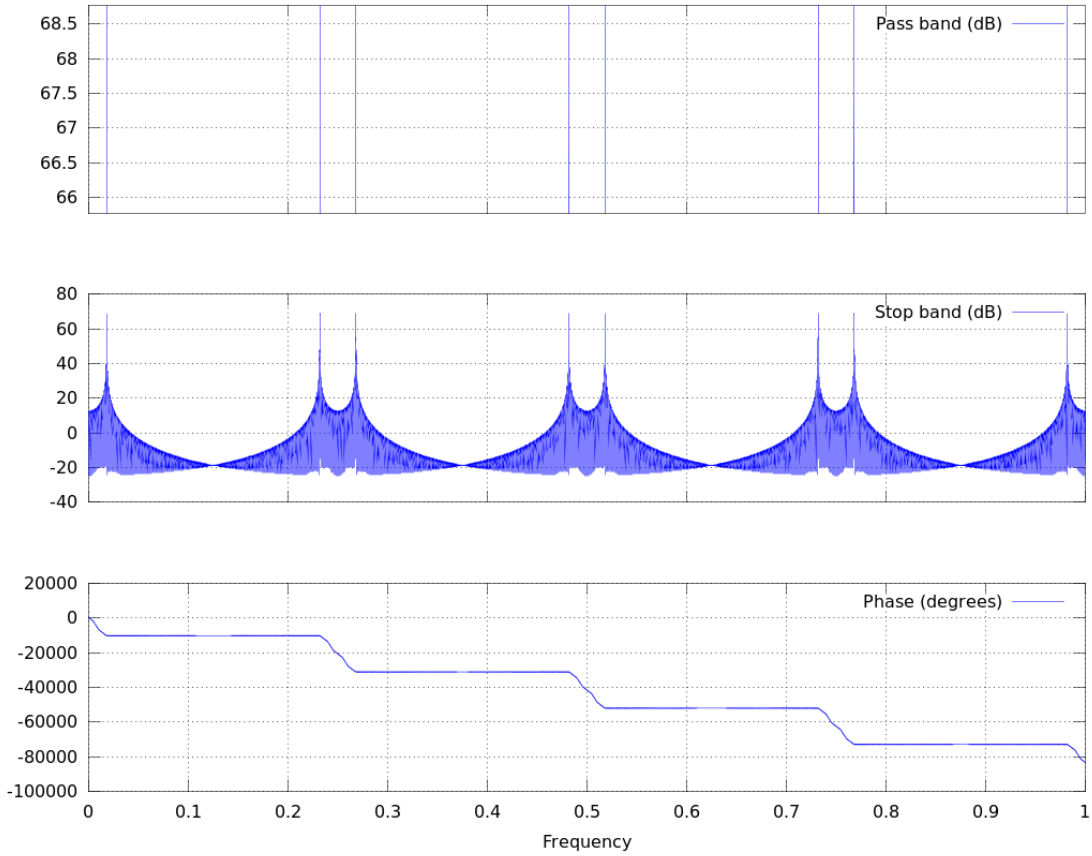
The multiplication with the unit impulse train:
*sdu=s.*unit;*
(This multiplication is also called „frequency mixing“).
Now plot the result, the first 100 samples:
plot(sdu(1:100))



This is our signal still with the zeros in it. Now take a look at the spectrum of the original signal s:
freqz(s)



The upper two plots show the magnitude of the frequency spectrum of our signal, the lowest plot is its phase. Observe that the frequency axis (horizontal) is a normalized frequency, normalized to the Nyquist frequency, in our case 22050 Hz. Hence our sinusoid should appear as a peak at normalized frequency 400/22050=0.018141, which we indeed see.
The uppermost plot shows a magnification of the top part of the middle plot, which is often used to assess a pass band for filters.
Now we can compare this to our signal with the zeros, sdu:
freqz(sdu);



Here we can see the original line of our 400 Hz tone, and now also the 7 new aliasing components. Observe that always 2 aliasing components are close together. This is because the original 400 Hz tone also has a spectral peak at the negative frequencies, at -400 Hz, or at normalized frequency -0.018 (which is 400/22050).

Now also listen to the signal with the zeros:
sound(sdu,44100);
Here you can hear that it sounds quite different from the original, because of the strong aliasing components!

Python real time audio **example**: This example takes the microphone input and samples it, without removing the zeros, and plays it back the the speaker in real time. It constructs a unit pulse train, with a 1 at every N'th sample, using the modulus function „%“,

```
s=(np.arange(0,CHUNK)%N)==0
```

Start it with:

```
python pyrecplay_samplingblock.py
```

Removing the zeros

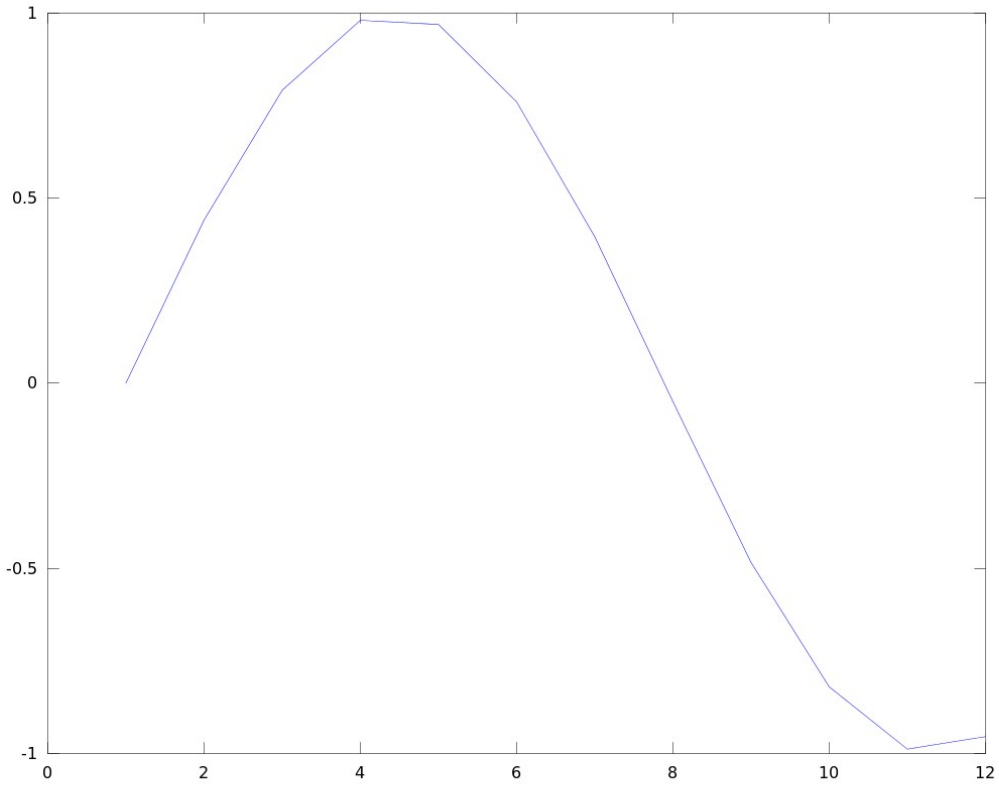
The final step of downsampling is now to omit the zeros between the samples, to obtain the lower sampling rate. Let's call the signal without the zeros

$$y(m),$$

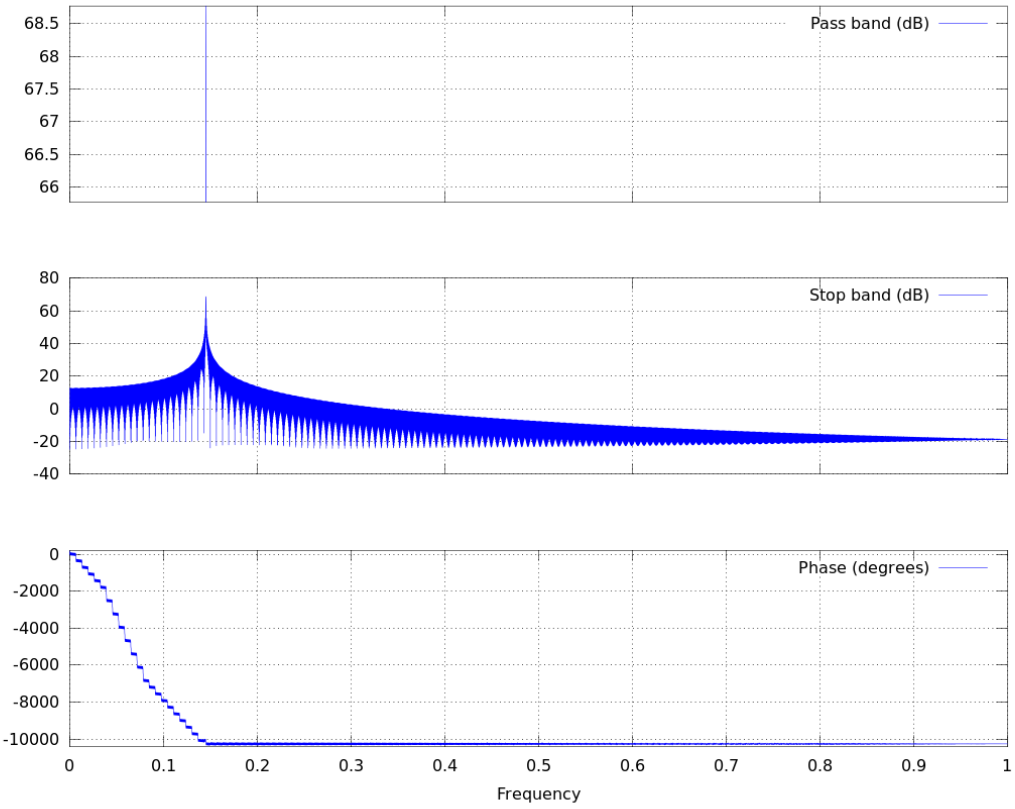
where the time index m denotes the **lower sampling rate** (as opposed to n, which denotes the higher sampling rate).

In our Matlab/Octave example this is:

```
sd=sdu(1:8:44100);
plot(sd(1:(100/8)));
```



We can now take a look at the spectrum with *freqz(sd);*



Observe that the sine signal now appear at normalized frequency of 0.145, a **factor of 8 higher** than before, with the zeros in it, because we **reduced the sampling rate by 8**. This is because we now have a new Nyquist frequency of 22050/8 now, hence our normalized frequency becomes $400/22050 \cdot 8 \approx 0.145$. This means removing the zeros scales or stretches our frequency axis.

Observe that here we only have $100/8 \approx 12$ samples left.

How are the frequency responses or spectra of $y(m)$ and $x^d(n)$ connected? We can simply take the Fourier transforms of them,

$$X^d(\Omega)=\sum_{n=-\infty}^{\infty} x^d(n)\cdot \mathrm{e}^{-j\Omega n}$$

still with the zeros in it. Hence most of the sum contains only zeros. Now we only need to let the sum run over the non-zeros entries (only every Nth entry), by replacing n by mN, and we get

$$X^d(\Omega)=\sum_{n=mN} x^d(n)\cdot \mathrm{e}^{-j\Omega n} \ ,$$

for all integer m, now without the zeros. Now we can make the connection to the Fourier transform of y(m), by making the index substitution m for n in the sum,

$$X^d(\Omega)=\sum_{m=-\infty}^{\infty} y(m)\cdot \mathrm{e}^{-j\Omega\cdot N m}=Y(\Omega\cdot N)$$

This is now our result. It shows that the downsampled version (with the removal of the zeros), has the same frequency response, but the frequency variable Ω is scaled by the factor N. For instance, the normalized frequency π/N before downsampling becomes π after removing the zeros! It shows that a small part of the spectrum before downsampling becomes the full usable spectrum after downsampling. Observe that we don't loose any frequencies this way, because by looking at eq. (1) we

see that we obtain multiple copies of the spectrum in steps of $2\pi/N$, and hence the spectrum already has a periodicity of $2\pi/N$. This means that the spectrum between $-\pi/N$ and π/N for instance (we could take any period of length $2\pi/N$) contains a unique and full part of the spectrum, because the rest is just a periodic continuation.
This can be seen in following pictures,

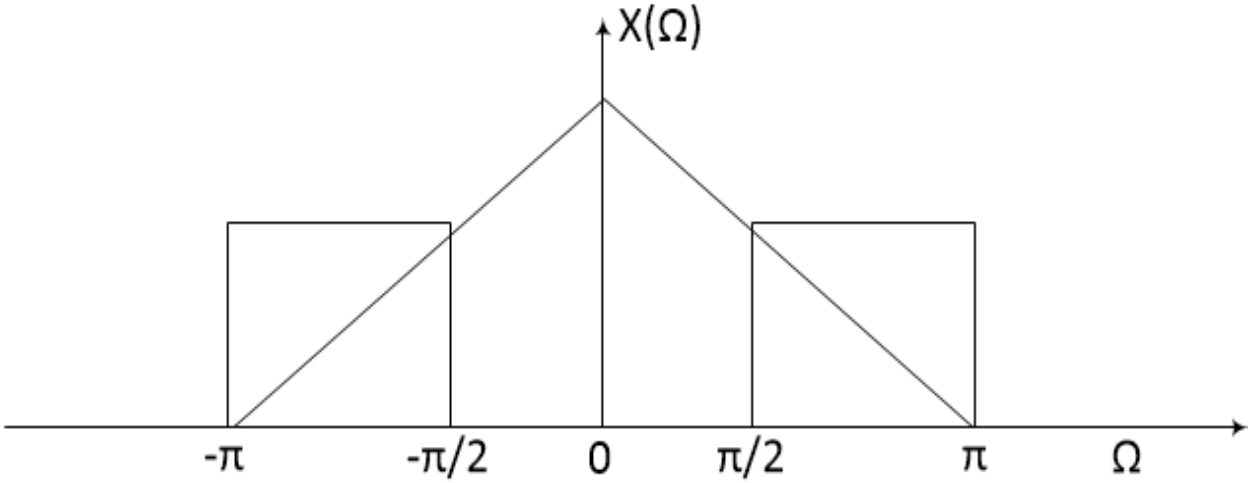


Figure 1: The magnitude spectrum of a signal. The 2 boxes symbolize the passband of an ideal bandpass, here a high pass.

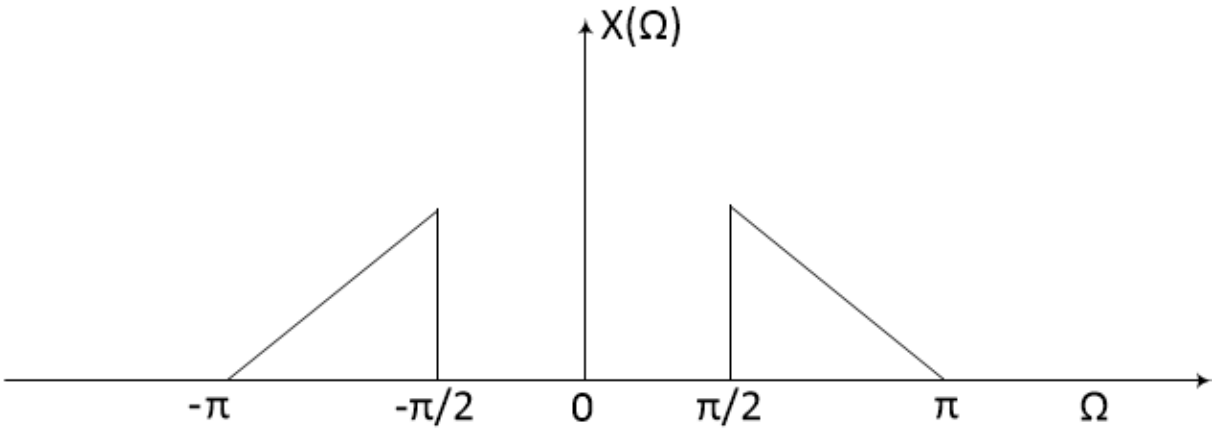


Figure: The signal spectrum after passing through the high pass.

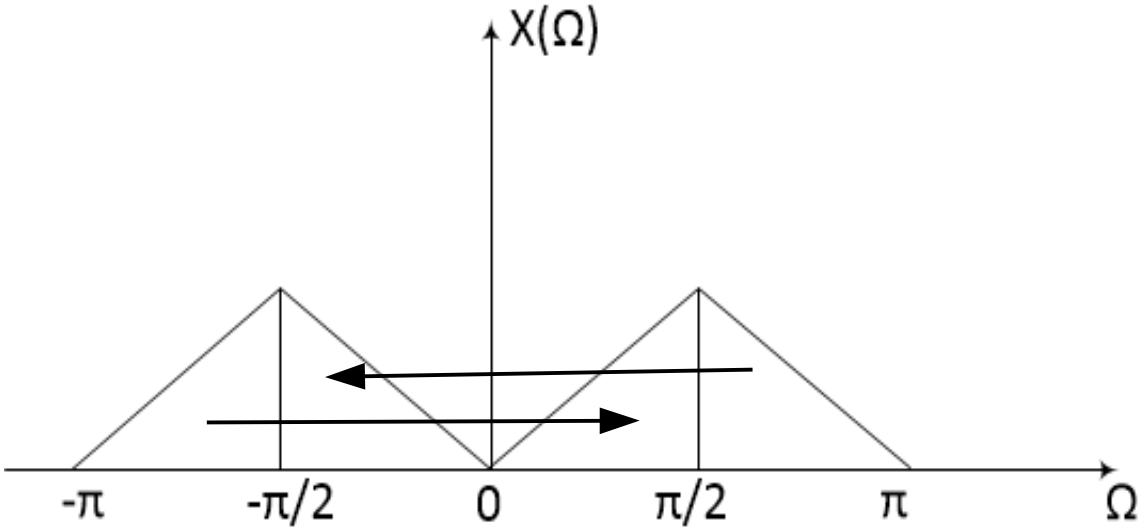


Figure: Signal spectrum after multiplication with the unit pulse train, for N=2, hence setting every second value to zero (the zeros still in the sequence). Observe the we shift and add the signal by multiples of $2\pi/2=\pi$, and in effect we obtain „mirrored“ images of the high frequencies to the low frequencies (since we assume a real valued signal). Observe that the mirrored spectra and the

original spectrum don't overlap, which makes reconstruction easy.

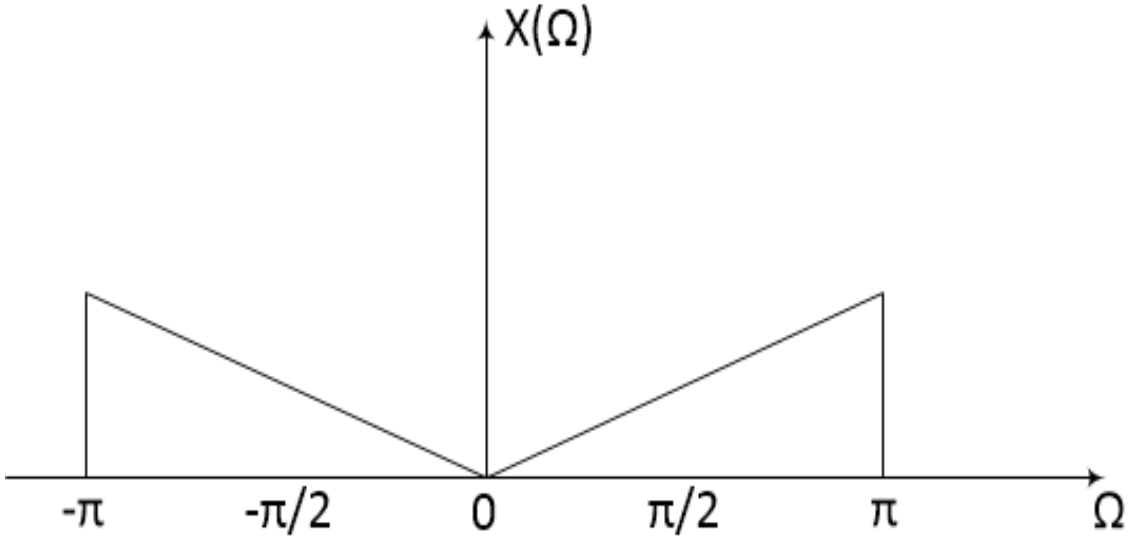


Figure: Signal spectrum after downsampling (removing the zeros) by N (2 in this example). Observe the stretching of the spectrum by a factor of 2.

Upsampling

What is still missing in our system is the upsampling, as the opposite operation of downsampling, for the case where we would like to increase our sampling rate. One of the first (**wrong!**) approaches to upsampling that often comes to mind if we want to do upsampling by a factor of N, is to simply repeat every sample N-1 times. **But** this is equivalent to first inserting N-1 zeros after each sample, and then filter the resulting

sequence by a low pass filter with an impulse response of N ones. This is a very special case, and we would like to have a more general case. Hence we assume that we upsample by always first inserting N-1 **zeros** after each sample, and then have some interpolation filter for it. Again we take the signal at the lower sampling rate as

$$y(m)$$

with index m for the lower sampling rate, and the signal at the higher sampling rate, with the zeros in it, as

$$x^d(n)$$

with index n for the higher sampling rate. Here we can see that this is simply the reverse operation of the final step of removing the zeros for the downsampling. Hence we can take our result from downsampling and apply it here:

$$X^d(\Omega) = Y(\Omega \cdot N)$$

or

$$X^d(\Omega / N) = Y(\Omega)$$

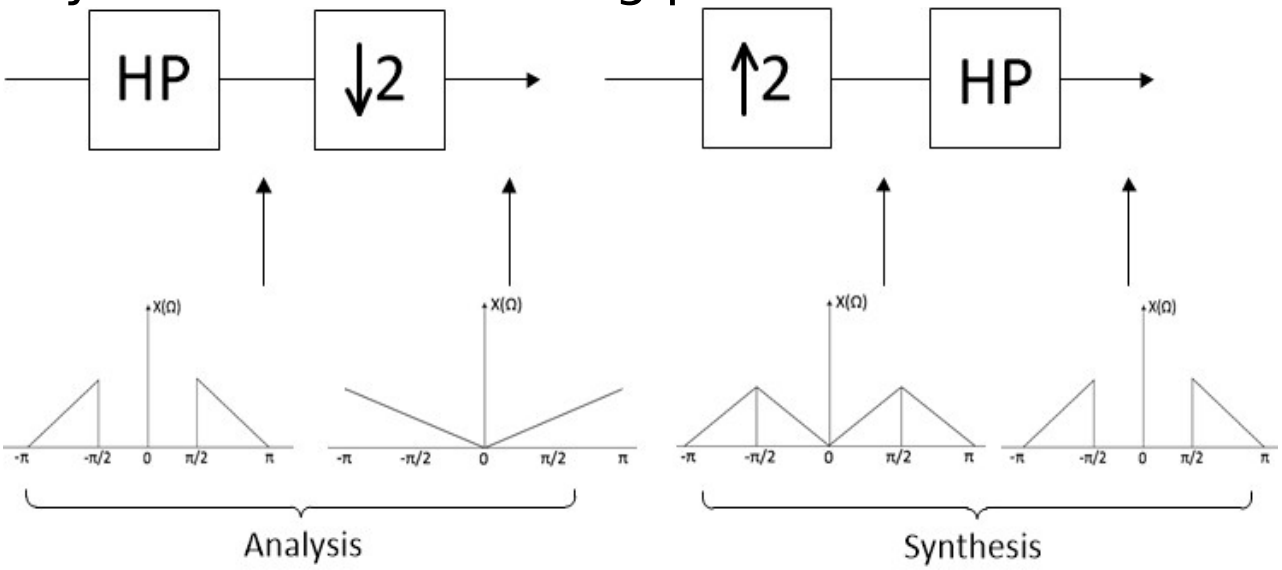
We are now just coming from y(m), going to the now upsampled signal $x^d(n)$.
For instance if we had the frequency π before upsampling, it becomes $\pi/2$ for the upsampled signal, if we have N=2. In this way we now get an „extended“ frequency range.

Since we now have again the signal including the zeros, $x^d(n)$, we again have the periodic spectrum, as before, as we progress through the same steps backwards now. We can also see that the result of upsampling is periodic in frequency, because the signal was 2π periodic before upsampling anyway, and after upsampling the frequency scale replaces $2\pi \cdot N$ by 2π .

Reconstruction

Observe that if the **aliasing components don't overlap**, we can **perfectly reconstruct** the signal by using a suitable filter. We can make sure that they don't overlap by **filtering** the signal at the **higher sampling rate**, before they can overlap. If they already overlap at the lower sampling rate, it would be too late to separate the different components, the signal would already be „destroyed“.

We can perfectly reconstruct the high pass signal in our example if we use ideal filters, using upsampling and ideal high pass filtering.
In this way we have for the analysis and synthesis the following picture



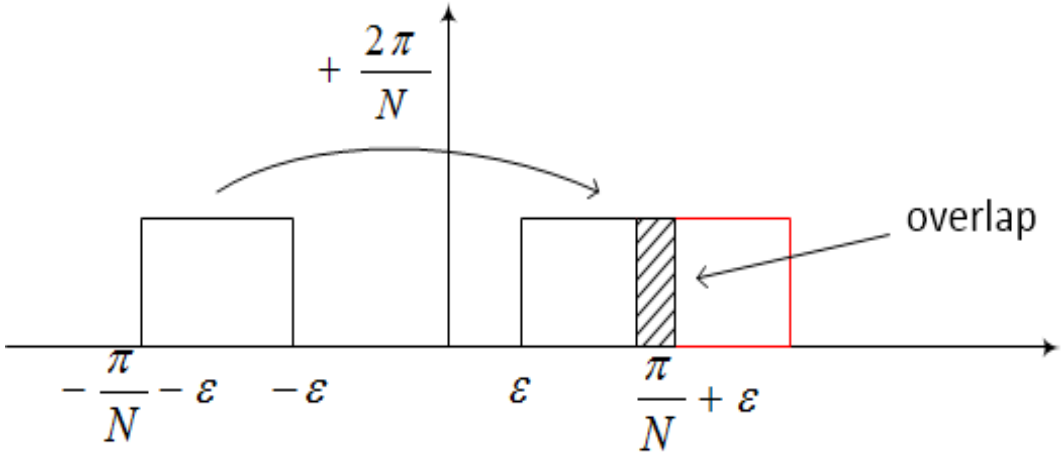
Observe that we violate the conventional Nyquist criterium, because our high pass passes the high frequencies. But then the sampling mirrors those frequencies to the lower range, such that we can apply the traditional Nyquist sampling theorem. This method is also known as bandpass Nyquist. This is an important principle for filter banks and wavelets. It says that we can perfectly reconstruct a bandpass signal in a filter bank, if we sample with twice the rate as the **bandwidth** of our bandpass signal (assuming ideal filters).

Observe that this simple assumption only works for real valued filters if we have bandpass filters which start at frequencies $\pi/N \cdot k$ (integer multiples of π/N) . Otherwise we could have overlap to the aliased components!

Example: We have a real valued bandpass filter which starts its passband at ϵ and ends at $\epsilon+\pi/N$ (since it is real values the the passband also appears at the same negative frequencies, $-\epsilon$ to $-\epsilon-\pi/N$). After multiplication with the unit pulse train, we get one (out of N) aliasing component by shifting the negative passband by $2\pi/N$ to the positive frequencies, which results in the range of $\pi/N-\epsilon$ to $2\pi/N-\epsilon$. Hence we have an overlap from $\pi/N-\epsilon$ to $\pi/N+\epsilon$ and we could not perfectly reconstruct our signal!

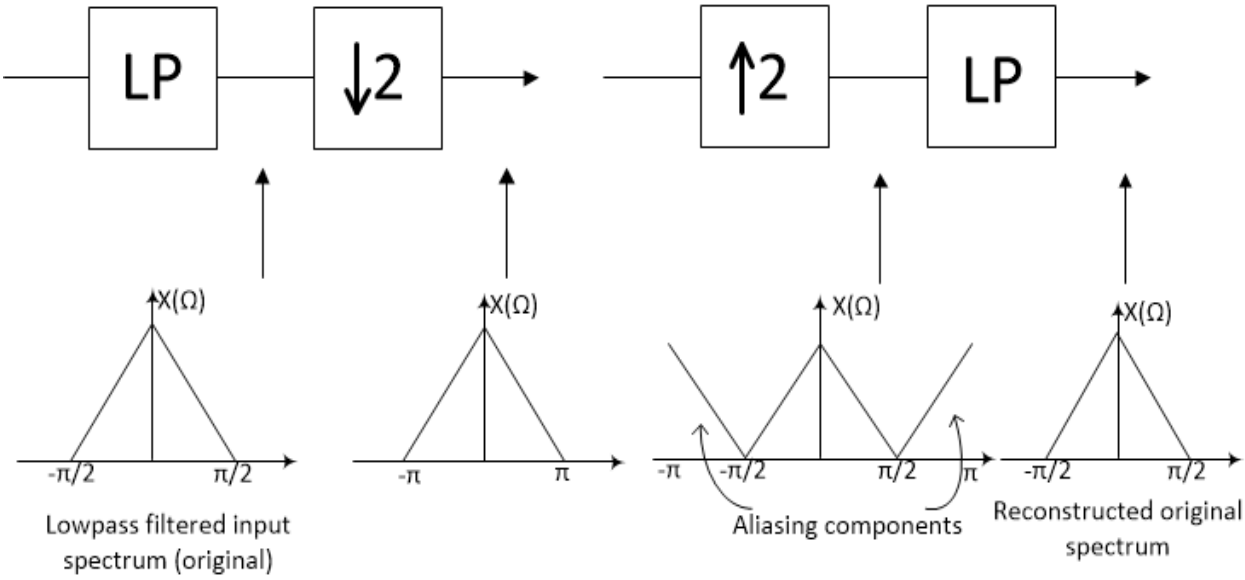
Hence, sampling of twice the bandwidth for real valued signals only works if the bandwidth are aligned with $\pi/N \cdot k$. What could we do otherwise to avoid overlapping aliasing components? We could simply increase the sampling rate, for instance to twice the usual sampling rate (4 times the bandwidth), to have a "safety margin". Another possibility would be to shift the bandpass signals in frequency, such that

they are now aligned with the above mentioned grid.
Observe that this restriction is not needed for complex signals or filter banks. That is also why complex filter banks are used for instance in acoustic echo cancellation, because there the sampling rate can be increased by a certain fraction (less than a factor of 2) to reduce aliasing artifacts, and still have a lower complexity.



Summary: if our band boarders are alligned with multiples of π/N then we can donwsample by N, otherwise we are on the save side by using N/2 as downsampling rate for real valued signals. For complex signals we can always downsample by N, regardles of the exact placement of the bandpass filter.

Compare with the standard Nyquist case: here we have a lowpass signal which we downsample and reconstruct:



The z-Transform

The z-Transform is a more general transform than the Fourier transform, and we will use it to obtain perfect reconstruction in filter banks and wavelets. Hence we will now look at the effects of sampling and some more tools in the z-domain.

Since we usually deal with causal systems in practice, we use the 1-sided z-Transform, defined as

$$X(z)=\sum_{n=0}^{\infty} x(n)z^{-n}$$

First observe that we get our usual frequency response if we evaluate the z-tranform along the unit circle in the z-domain,

$$z=\mathrm{e}^{j\Omega}$$

This connects the z-Transform with the DTFT, except for the sample index n, which for the so-called one-side z-Tranform starts at n=0, and for the DTFT starts at $n=-\infty$

Properties of the z-Transfrom:

Shift property:

Take two causal sequences (causal means sample value 0 for negative indices): Sequence x(n), and x(n-1), which is the

same sequence but delayed by one sample. Then their z-transforms are:

$$x(n) \rightarrow \sum_{n=0}^{\infty} x(n) \cdot z^{-n} =: X(z)$$

$$x(n-1) \rightarrow \sum_{n=0}^{\infty} x(n-1) \cdot z^{-n} = \sum_{n=1}^{\infty} x(n-1) \cdot z^{-n} =$$

Use the index substitution, $n' \leftarrow n-1$ or $n'+1 \leftarrow n$ to get rid of the "n-1" in the transform:

$$= \sum_{n'=0}^{\infty} x(n') \cdot z^{-(n'+1)} = z^{-1} \cdot \sum_{n'=0}^{\infty} x(n') \cdot z^{-n'} = X(z) \cdot z^{-1}$$

This shows that a **delay by 1 sample** in the signal sequence (time domain) corresponds to the **multiplication with** z^{-1} in the z-domain:

$$x(n) \rightarrow X(z)$$

$$x(n-1) \rightarrow X(z) \cdot z^{-1}$$

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:

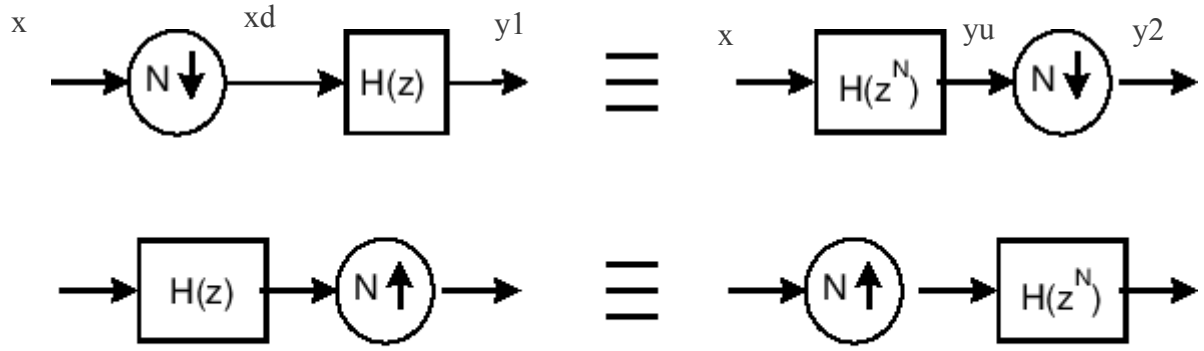
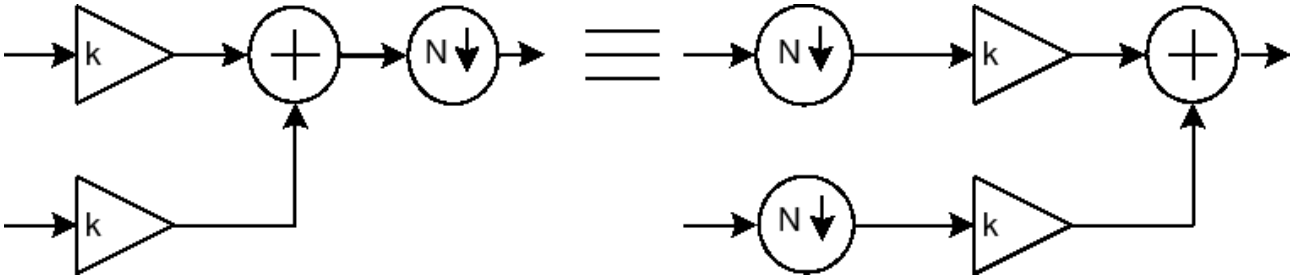


Figure 24.3: Multirate noble identities



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 $H_T(z)$:

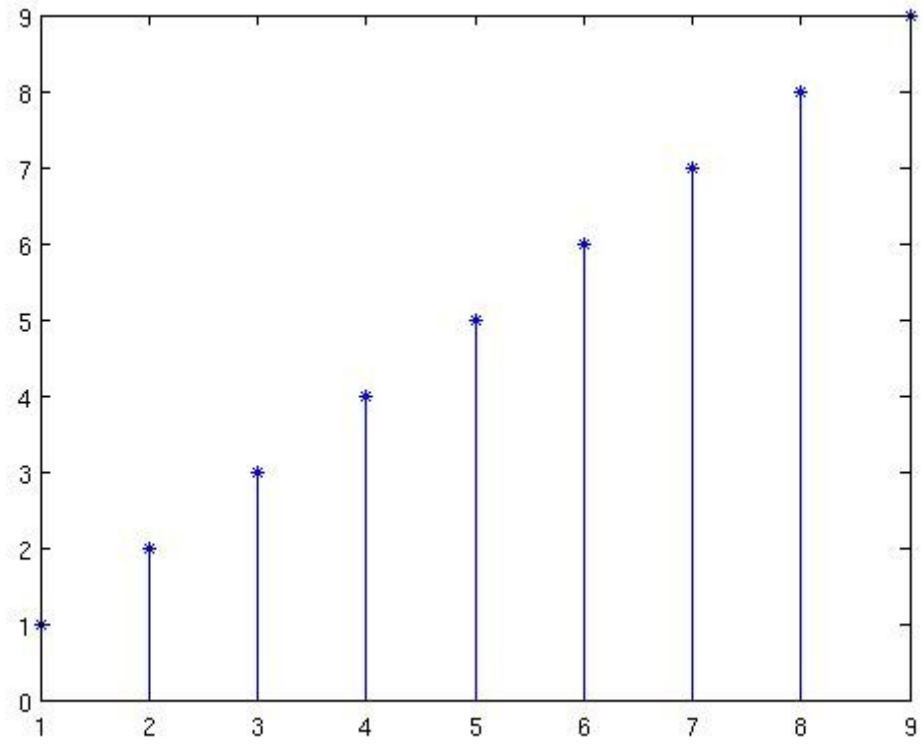
$$H_T(z)=H_0(z^N)+H_1(z^N)\cdot z^{-1}+...+H_{N-1}(z^N)\cdot z^{N-1}$$

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

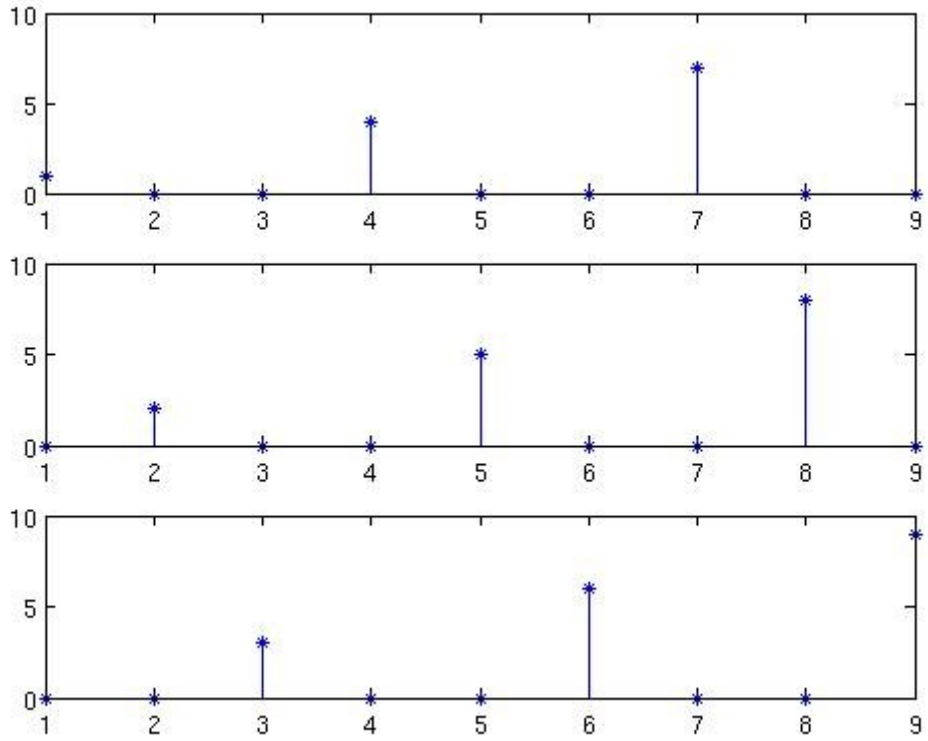
$H_1(z^N)$ contains the coefficients at positions mN+1,

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

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

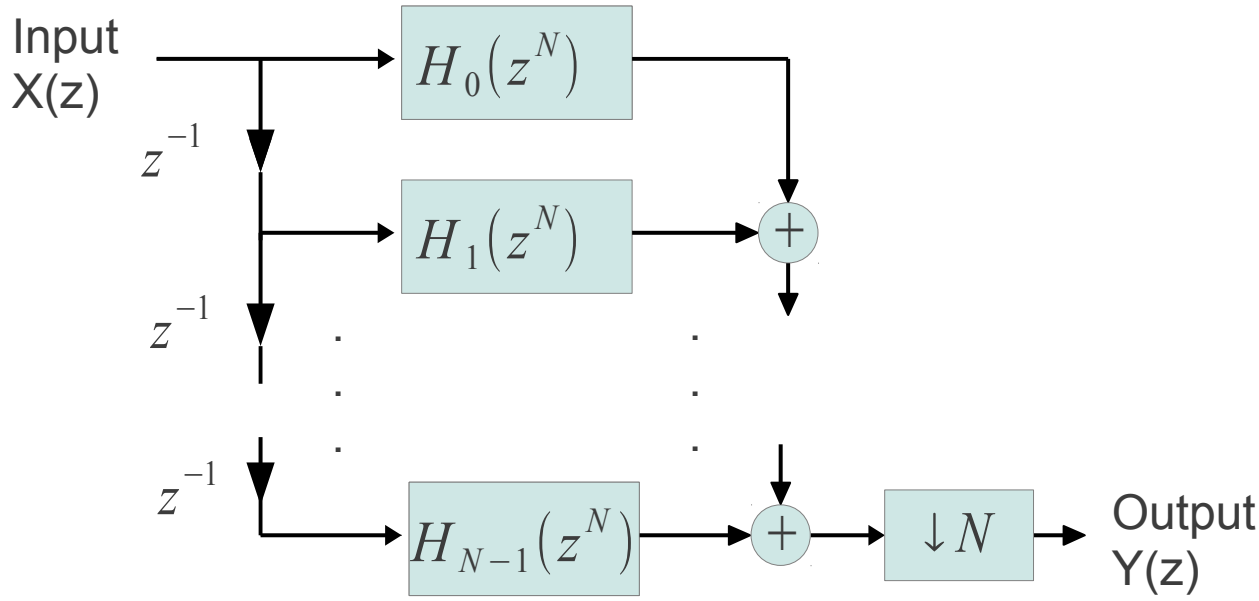
$$H_1(z^2) = 2 + 4z^{-2} + 6z^{-4} + ...$$

Hence we can combine the total filter from its

polyphase components,

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

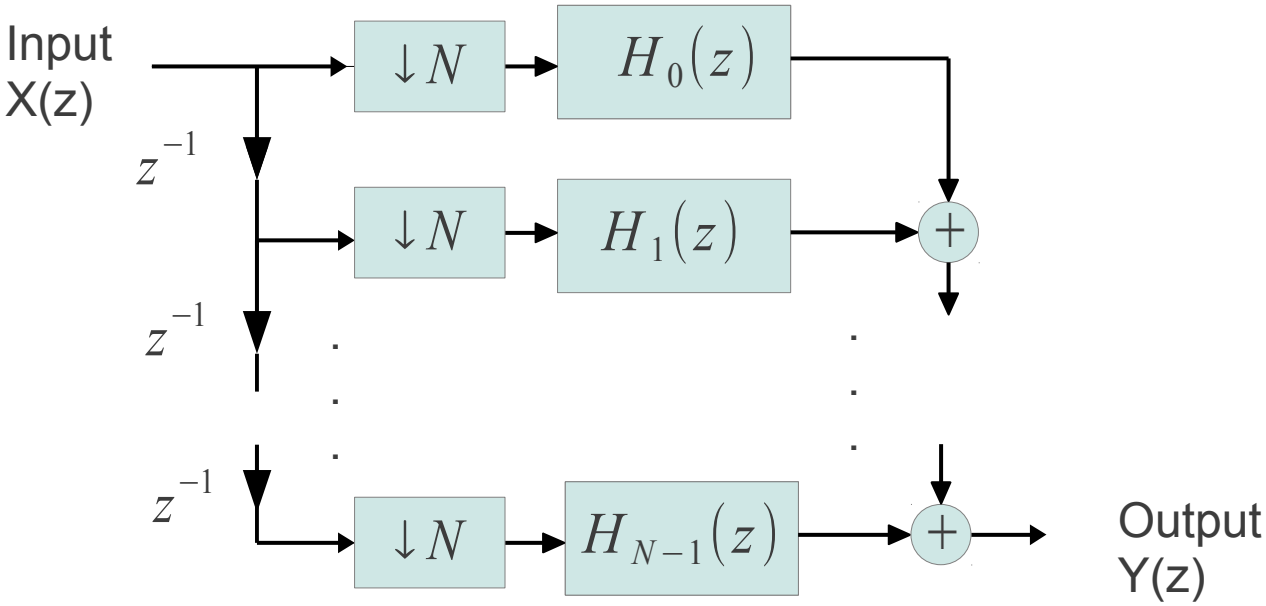
The general case is illustrated in the following block diagram, which consists of a delay chain on the left to implement the different delays z^{-i} , and the polyphase components $H_i(z^N)$ of the filter:



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);

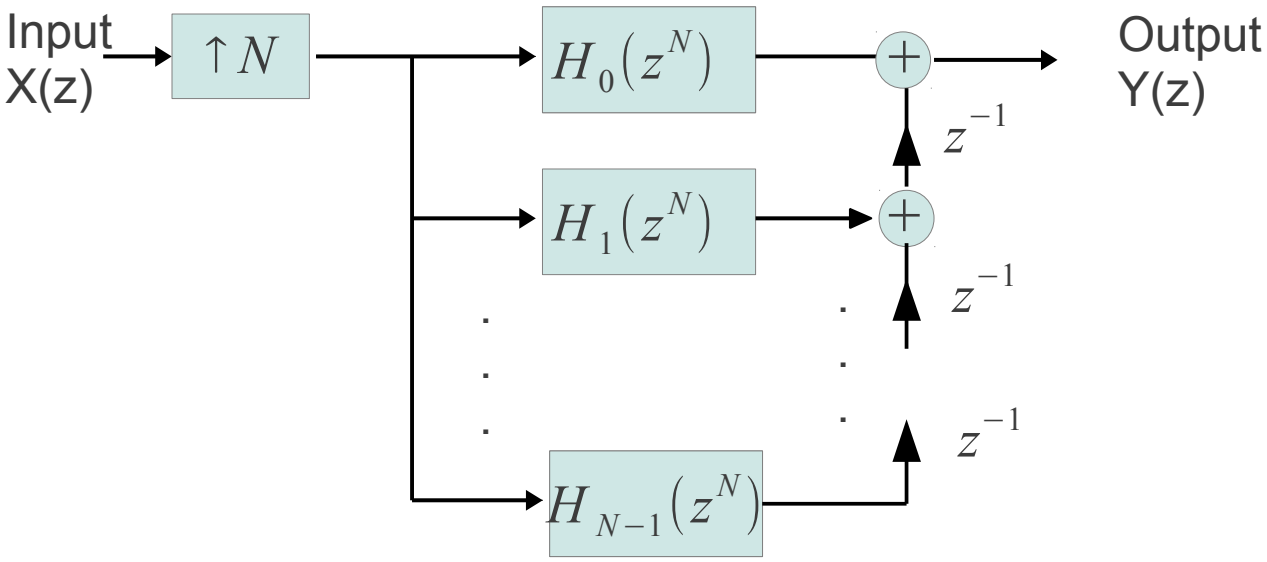
Correspondingly, **up-samplers** can be obtained with filters operating on the lower

sampling rate.

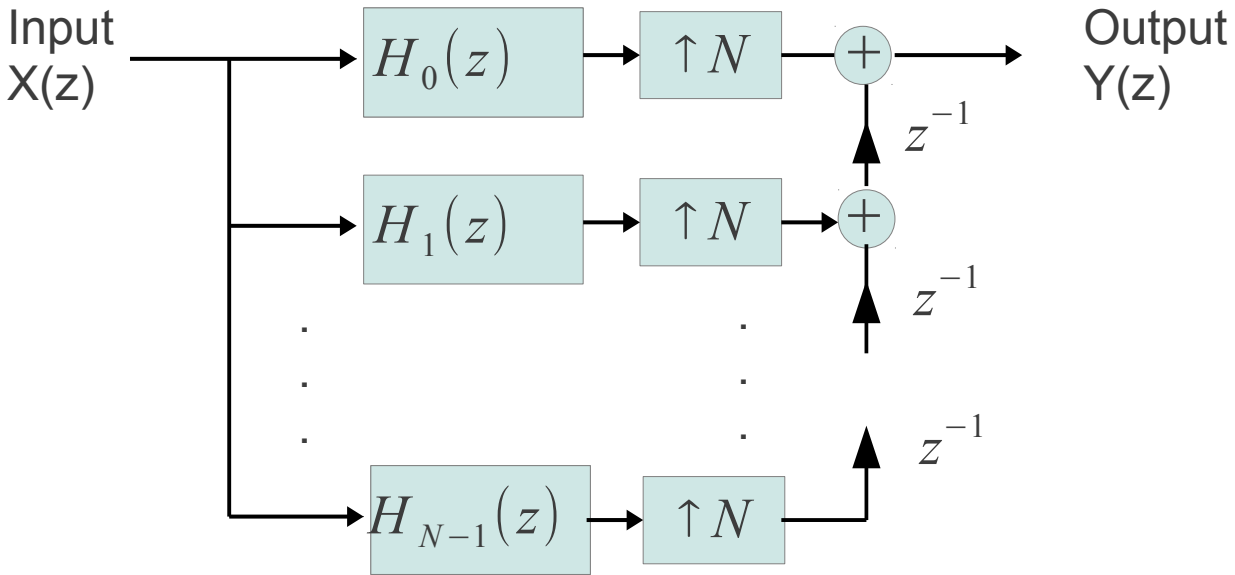
Since $H_i(z^N)$ and z^{-i} are linear time-invariant systems, we can exchange their ordering,

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

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');

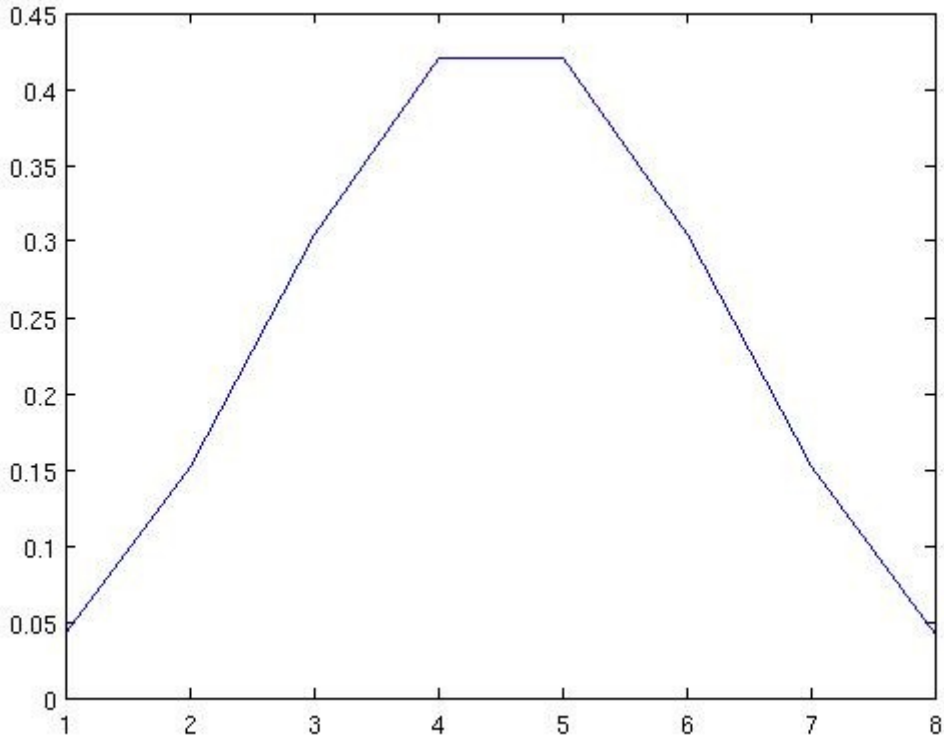
This approach has the advantage, that all kind of filters can be optimized, FIR and also IIR, with any kind of magnitude and phase behaviour, so that it is most universal.

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

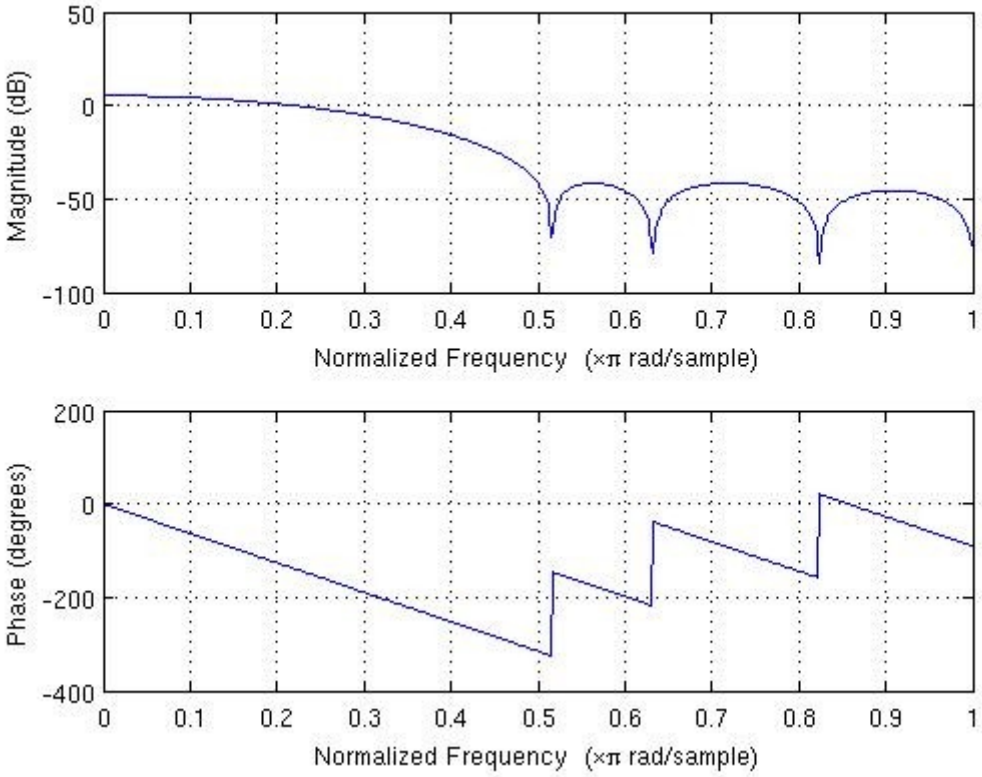
behaviour (all ripples have the same hight) of the filter in the frequency domain.

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.

Digital Signal Processing 2/ Advanced Digital Signal Processing
Lecture 7,
z-Transform, Filters

Gerald Schuller
TU-Ilmenau

The z-Transform

The z-Transform is a more general transform than the Fourier transform, and we will use it to obtain perfect reconstruction in filter banks and wavelets. Hence we will now look at the effects of sampling and some more tools in the z-domain.
Since we usually deal with causal systems in practice, we use the 1-sided z-Transform, defined as

$$X(z)=\sum_{n=0}^{\infty} x(n)z^{-n}$$

First observe that we get our usual frequency response if we evaluate the z-tranform along the unit circle in the z-domain,

$$z=\mathrm{e}^{j\Omega}$$

This connects the z-Transform with the DTFT, except for the sample index n, which for the so-called one-side z-Tranform starts at n=0, and for the DTFT starts at $n=-\infty$.

In general, we can write z with an angle and a magnitude,

$$z = r \cdot e^{j\Omega}$$

where we can interpret the Ω as the normalized angular frequency, and the r a damping factor for an exponentially decaying oscillation (or exponentially growing if r >1). Observe: This damping factor is not in the DTFT. This means in the z-Transform we can have a converging sum of the transform even for unstable signals or system, by just choosing r large enough! This means the **Region of Convergence** (**ROC**) just becomes smaller. Remember, in the z-transform sum we have $z^{-1} = \frac{1}{r} \cdot e^{-j\Omega}$.

Properties of the z-Transfrom:

Shift property:

Take two causal sequences (causal means sample value 0 for negative indices): Sequence x(n), and x(n-1), which is the same sequence but delayed by one sample. Then their z-transforms are:

$$x(n) \rightarrow \sum_{n=0}^{\infty} x(n) \cdot z^{-n} =: X(z)$$

$$x(n-1) \rightarrow \sum_{n=0}^{\infty} x(n-1) \cdot z^{-n} = \sum_{n=1}^{\infty} x(n-1) \cdot z^{-n} =$$

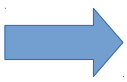
Use the index substitution, $n' \leftarrow n-1$ or $n'+1 \leftarrow n$ to get rid of the "n-1" in the

transform:

$$= \sum_{n'=0}^{\infty} x(n') \cdot z^{-(n'+1)} = z^{-1} \cdot \sum_{n'=0}^{\infty} x(n') \cdot z^{-n'} = X(z) \cdot z^{-1}$$

This shows that a **delay by 1 sample** in the signal sequence (time domain) corresponds to the **multiplication with** z^{-1} in the z-domain:

$$x(n) \rightarrow X(z)$$



$$x(n-1) \rightarrow X(z) \cdot z^{-1}$$

Z-Transform Properties

z-Transform definition:

$$x(n) \rightarrow \sum_{n=0}^{\infty} x(n) \cdot z^{-n} =: X(z)$$

The z-transform turns a sequence into a polynomial in z.

Example: $x(n) = [2,4,3,1]$

$$X(z) = 2 + 4 \cdot z^{-1} + 3 \cdot z^{-2} + z^{-3}$$

Shift property:

$$x(n) \rightarrow X(z)$$

$$x(n-1) \rightarrow X(z) \cdot z^{-1}$$

Recommended reading:

Alan V. Oppenheim, Ronald W. Schafer: "Discrete Time Signal Processing", Prentice Hall.

Related to the shift property is the z-transform

of the shifted unit pulse. The unit pulse is defined as

$$\Delta(n)=\begin{cases}1, & if \quad n=0\\ 0, & else\end{cases}$$

so it is just a zero sequence with a 1 at time 0. Its z-Transform is then:

$$\Delta(n)\rightarrow 1$$

The z-transform of the shifted unit pulse is

$$\Delta(n-d)\rightarrow z^{-d}$$

Shifted by d samples.

The “unit step” function is defined as:

$$u(n)=\begin{cases}1, & if \quad n\geq 0\\ 0, & else\end{cases}$$

Linearity: $a\cdot x(n)\rightarrow a\cdot X(z)$

$$x(n)+y(n)\rightarrow X(z)+Y(z)$$

Convolution:

$$x(n)*y(n)\rightarrow X(z)\cdot Y(z)$$

The z-transform turns a convolution into a multiplication.

Remember: the convolution is defined as:

$$x(n)*y(n)=\sum_{m=-\infty}^{\infty}x(m)\cdot y(n-m)$$

This is because the convolution of 2 sequences behave in the same way as the multiplication of 2 polynomials (the z-transform) of these sequences. This is one of the main advantages of the z-Transform, since it turns convolution

into a simpler multiplication (which is in principle invertible).

Example z-transform: Exponential decaying sequence:

$x(n)=p^n,$ for n=0,1,..., meaning the sequence

$$1, p, p^2, p^3, \ldots$$

$$\rightarrow X(z)=\sum_{n=0}^{\infty} p^n \cdot z^{-n}$$

Remember from last time: we had a closed form solution for this type of geometric sums:

$$S=\sum_{k=0}^{N-1} c^k$$

its solution was:

$$S=\frac{c^N-1}{c-1}$$

But now we have an infinite sum, which means N goes towards infinity. But we have the expression c^N in the solution. If $|c|<1$, then this goes to zero $c^N \rightarrow 0$. Now we have $c=p\cdot z^{-1}$. Hence, if $|p\cdot z^{-1}|<1$ we get

$$\rightarrow X(z)=\frac{1}{1-p\cdot z^{-1}}=\frac{z}{z-p}$$

Observe that this fraction has a **pole** at position z=p, and a **zero** at position z=0.

Keep in mind that this solution is only valid for all p which fullfill $|p\cdot z^{-1}|<1$. We see that this is

true for $|z|>|p|$. This is also called the “**Region of Convergence” (ROC)**. The ROC is connected to the resulting stability of the system or signal. The region of convergence is outside the pole locations. If the region of convergence include the unit circle, we have a stable system. This means: if the **poles are inside the unit circle**, we have a s**table system.**
The sum of x(n) **converges** (we get the sum if we set $z=1$) if **abs(p)<1**. In this case we also say that the signal or system is **stable** (meaning we obtain a bounded output for a bounded input, so-called “BIBO stability”. In this case we see that the resulting pole of our z-transform is **inside the unit circle**. If abs(p)>1, we have an exponential growth, which is basically an “exploding” signal or system (meaning the output grows towards infinity), hence **unstable**.
In general we say that a system or a signal is **stable**, if the **poles** of its z-transform are **inside the unit circle** in the z-domain, or **unstable** if **at least one pole is outside the unit circle** (it will exponentially grow).

These are basic properties, which can be used to derive z-transforms of more complicated expressions, and they can also be used to obtain an inverse z-transform, by inspection. For instance if we see a fraction with a **pole** in

the z-Transform, we know that the underlying time sequence has an **exponential decay** in it.

One of the main differences compared to the DTFT: With the z-transform we can see if a signal or system is stable by looking at the position of the poles in the z-domain. This is not possible for the DTFT, since there we don't the positions of the poles.

Now take a look at our down sampled signal from last time:

$$x^d(n) = x(n) \cdot \Delta_N(n) = x(n) \cdot \frac{1}{N} \sum_{k=0}^{N-1} e^{j\frac{2\pi}{N} \cdot k \cdot n}$$

Now we can z-transform it

$$\sum_{n=0}^{\infty} x^d(n) \cdot z^{-n} = \sum_{n=0}^{\infty} x(n) \cdot \frac{1}{N} \sum_{k=0}^{N-1} e^{j\frac{2\pi}{N} \cdot k \cdot n} \cdot z^{-n}$$

Hence the effect of **multiplying our signal with the delta impulse train** in the z-domain is

$$X^d(z) = \frac{1}{N} \sum_{k=0}^{N-1} X\left(\mathrm{e}^{-j\frac{2\pi}{N} \cdot k} \cdot z\right)$$

Observe that here the aliasing components appear by multiplying z with $e^{-j\frac{2\pi}{N} \cdot k}$, which in effect is a shift of the frequency.
The effect is the **removal or re-insertion of the zeros** from or into the signal $x^d(n)$ at the

higher sampling rate and $y(m)$ at the lower sampling rate in the z-domain is

$$Y(z)=X^d(z^{1/N})$$

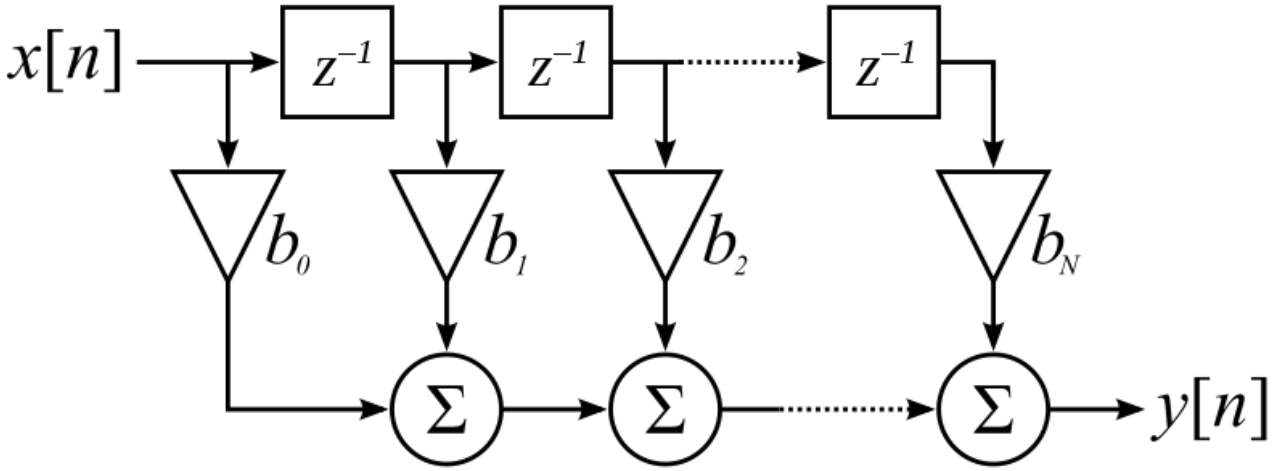
Filters

A simple Finite Impulse Response (**FIR**) filter has a difference equation like the following, with x(n) the input of our filter, and y(n) its output:

$$y(n)=\sum_{m=0}^{L} b(m)x(n-m)$$

Observe that this is the **convolution** of the signal *x(n)* with *b(n)*. Here, the *b(m)* are the coefficients of the filter, or its impulse response. These are commonly also referred to as “taps”, because this system can be viewed as “tapping” a delay line, as seen in the picture below.

This difference equation is also how usually filters are implemented in Matlab or other programming languages. A typical block diagram of an FIR filter is as follows



(From: http://en.wikipedia.org/wiki/Finite_impulse_response)

The z-transform of this difference equation is:

$$Y(z) = \sum_{m=0}^{L} b(m) \cdot z^{-m} \cdot X(z)$$

Now we can compute the transfer function, defined as the output divided by the input,

$$H(z) := \frac{Y(z)}{X(z)} = \sum_{m=0}^{L} b(m) \cdot z^{-m}$$

Observe that this is the z-transform of the coefficients b(m)! This is the z-transform of the impulse response of the FIR filter!
Now we can obtain the **frequency response** (so that we can see which frequencies are attenuated and which are not) from our transfer function of the filter by just replacing z by $e^{j\Omega}$:

$$H\left(e^{j\Omega}\right)=\sum_{m=0}^{L} b(m)\cdot e^{-j\,\Omega\cdot m}$$

Since $e^{j\Omega}$ is a complex number, our frequency response is also complex. Hence H is a complex number for each frequency ω . Usually it is plotted as a **magnitude** plot and a **phase** plot over frequency. Its magnitude tells us the attenuation at each frequency, and the phase its phase shift for each frequency. Using those 2 plots or properties, we can also design our filters with desired properties (for instance a stop-band at given frequencies). The Matlab function to generate a magnitude and phase plot of a transfer function or signal is “freqz”, which we already saw.

IIR Filters

(Infinite Impulse Response Filters).
Their difference equation is:

$$y(n)=\sum_{m=0}^{L} b(m)\cdot x(n-m)+\sum_{r=1}^{R} a(r)\cdot y(n-r)$$

(See also: Oppenheim, Schafer: “Discrete Time Signal Processing”, Chapter 6 in Ed. 1989)

Here we have **2 convolutions**. Observe the feedback from the output y back to the input in this sum. Also observe that the feedback part

starts with a delay of r=1. This is because we want to avoid so-called delayless loops. We cannot use the value y(n) before we computed it! Again, this difference equation is the usual implementation using Matlab or Octave.
The z-transform of this difference equation is

$$Y(z)=\sum_{m=0}^{L} b(m)\cdot X(z)\cdot z^{-m}+\sum_{r=1} a(r)\cdot Y(z)\cdot z^{-r}$$

Observe: **Matlab and Octave** is defining the coefficients *a* with **opposite signs** as we and Oppenheim/Schafer are defining. See for instance “help filter”.
To obtain the transfer function, we first move the Y(z) to one side:

$$Y(z)\left(1-\sum_{r=1}^{R} a(r)\cdot z^{-r}\right)=\sum_{m=0}^{L} b(m)\cdot X(z)\cdot z^{-m}$$

Hence the resulting transfer function is

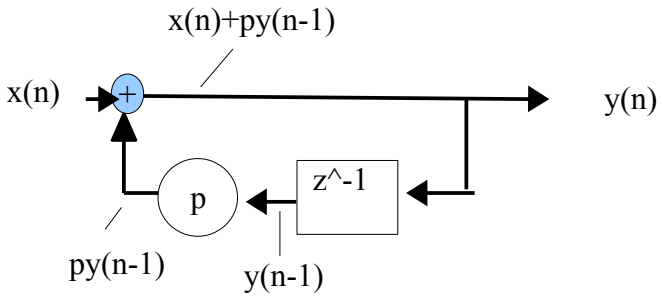
$$H(z)=\frac{Y(z)}{X(z)}=\frac{\sum_{m=0}^{L} b(m)\cdot z^{-m}}{1-\sum_{r=1}^{R} a(r)\cdot z^{-r}}$$

Here we can see that we obtain a polynomial in the denominator of our transfer function. The zeros of this denominator polynomial become the poles of the transfer function. If these poles are **inside the unit circle**, we have a **stable filter**!
Going back to our simple example of an

exponential decaying signal, this shows how to implement it. We just need a system with a pole at position *p*. In the above equation we obtain it by setting *b(0)=1* and *a(1)=p*. Hence we obtain a simple difference equation

$$y(n)=1\cdot x(n)+ p\cdot y(n-1)$$

This can also be written in the form of a block diagram:



In the z-domain this is

$$Y(z)=X(z)+ p\cdot z^{-1}\cdot Y(z)$$

$$\rightarrow H(z)=\frac{Y(z)}{X(z)}=\frac{1}{1-p\cdot z^{-1}}$$

In this structure we can now see the feedback loop.

Example: The Matlab or Octave function "*freqz*" can be used to plot the magnitude and phase plot of the transfer function of this filter. Its input are directly the coefficients *a* and *b* of

the transfer function H(z), in the form:

freqz(B,A),

where B and A are vectors containing the coefficients.

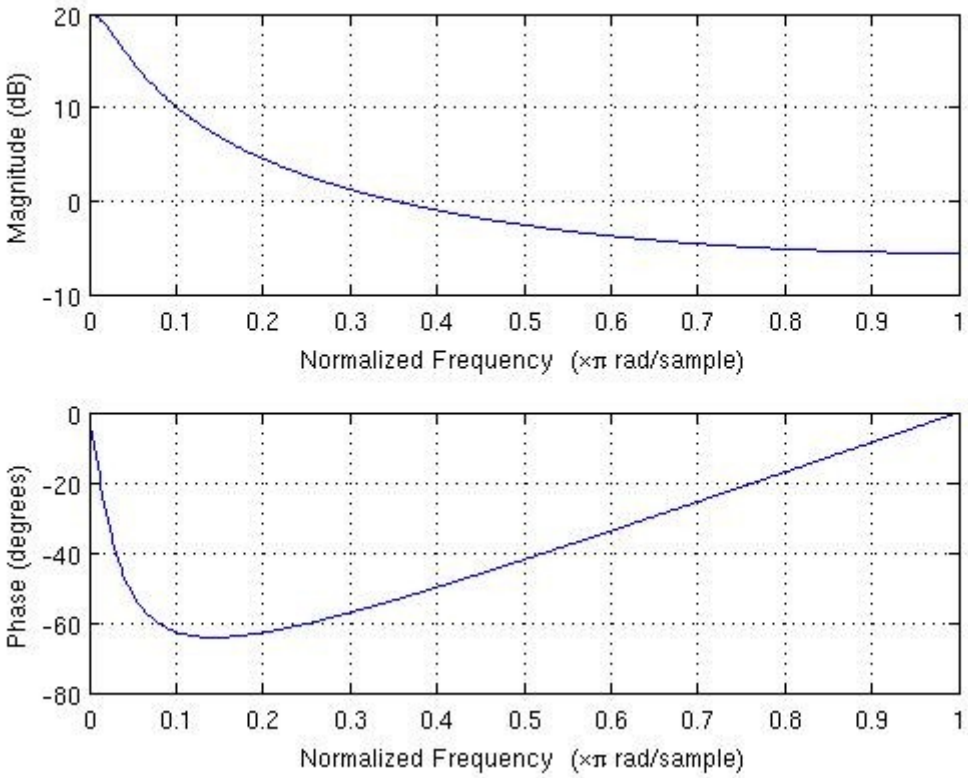
If we choose a(1)=*p=0.9* in our example, we obtain

A=[1,-0.9] and *B=1*.

Hence we type:

freqz(1,[1,-0.9])

and obtain the following plot:

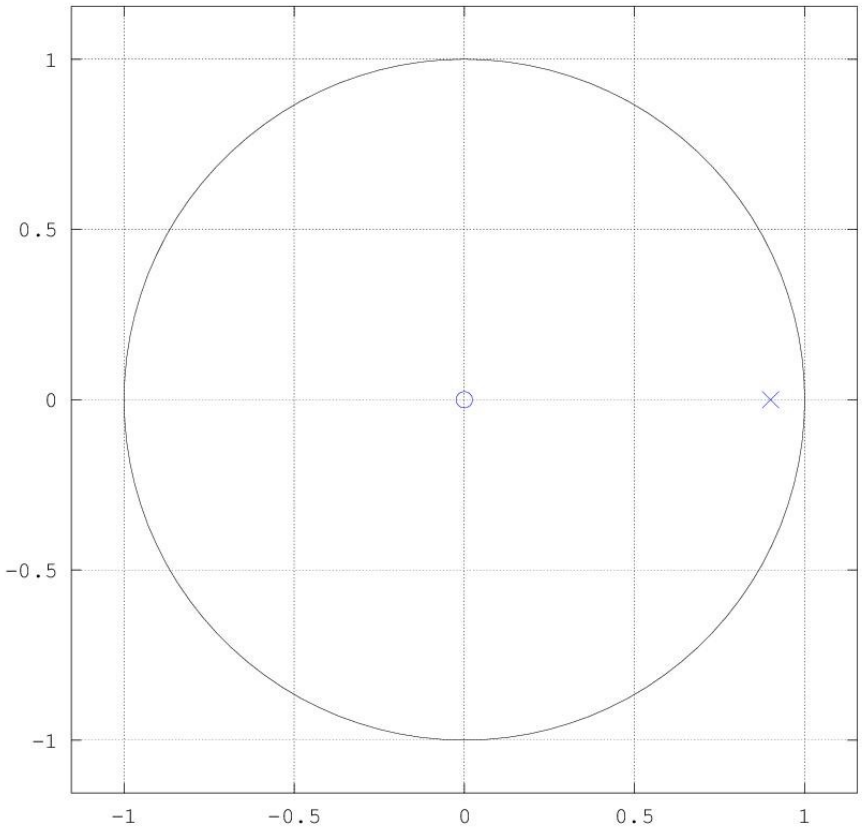


Observe that the horizontal axis is the normalized frequency (see last lecture), its right hand side is pi, which is the Nyquist frequency or half the sampling frequency. The frequency response we see here has a low pass characteristic.

We can use the command “zplane” to plot the location of the zeros and poles in the complex z-plane, with

zplane(B,A);

The resulting plot is:



Zeros are marked with an “**o**”, and **poles** are marked with an “**x**”. Here we see the pole at location z=0.9.

In general, the **closer a pole** is to the **unit circle**, the larger is the corresponding **peak** in the **magnitude** of the frequency response at a normalized **frequency** identical to the **angle of the pole** to the origin. Here we have an angle of 0 degrees. In the example we can indeed observe a peak in the magnitude

response at normalized frequency $\Omega=0$ above.

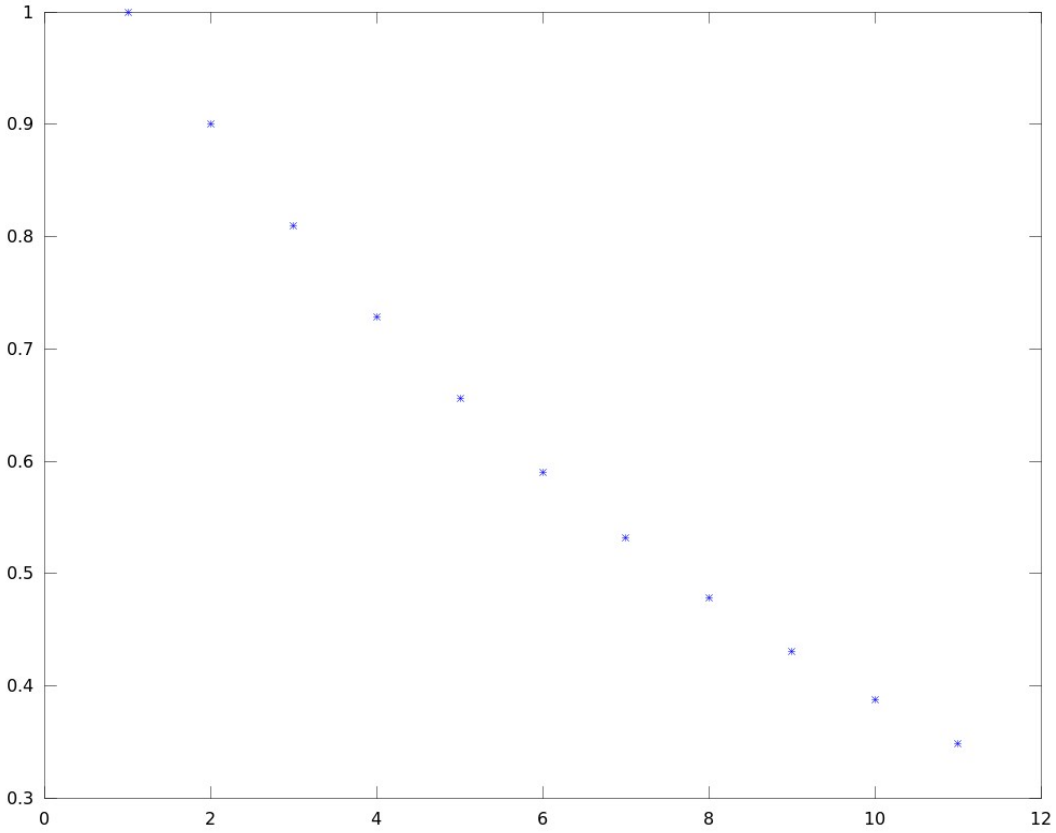
The **filtering** operation itself works similarly in Matlab or Octave, in the time domain. The function is "*filter*". Given an input signal in the vector x, and filter coefficients in vectors A and B, the filtered output y of our filter is simply:

y=filter(B,A,x);

Often used orders for the zeros and poles L and R are a few to a few dozen coefficients.

Octave/Matlab example:
Use the function *filter* to obtain the impulse response of this IIR filter.
Start with a unit impulse as input x:

```
x=[1,zeros(1,10)];
%B and A are given as before:
A=[1,-0.9]; B=1;
%Now calculate the impulse response:
y=filter(B,A,x);
plot(y,'*')
```



Here we can see the indeed exponetial decaying function (the sequence $ir(n) = p^n$ for p=0.9). In this way we can also test more complicated IIR filters. This exponential decaying impulse response again shows the stability of the filter, which was to be expected because the pole of its transfer function in the z-domain is placed at z=0.9, and hence inside the unit-circle!

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 "disolve" 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:
errorstopband*1000=errorpassband*1, and hence errorstopband=errorpassband/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, 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 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];
```

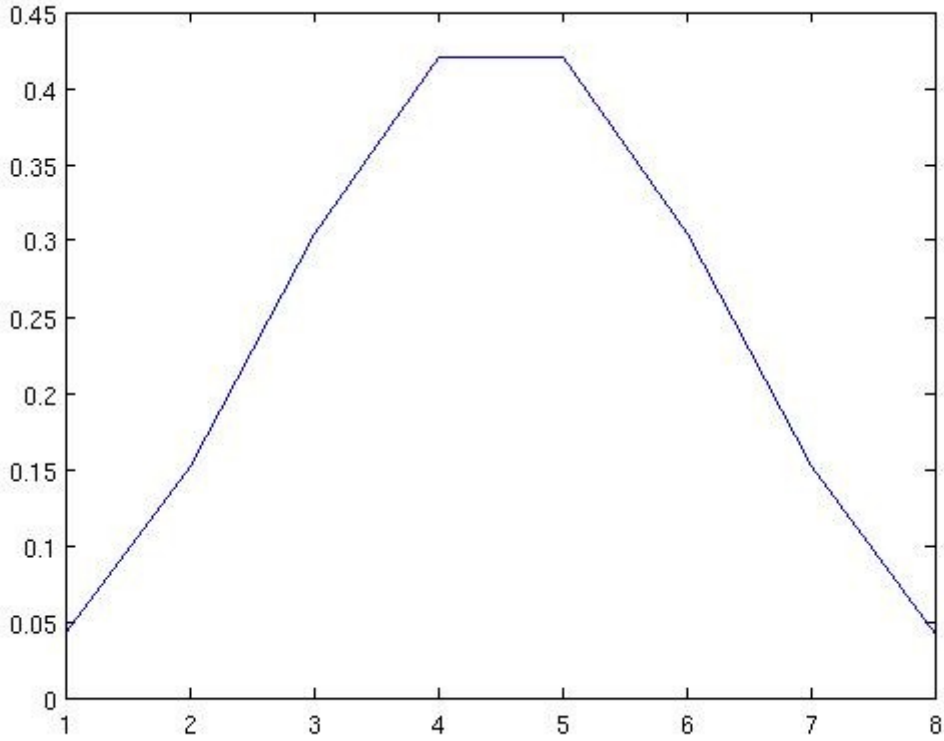
%and we call:

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

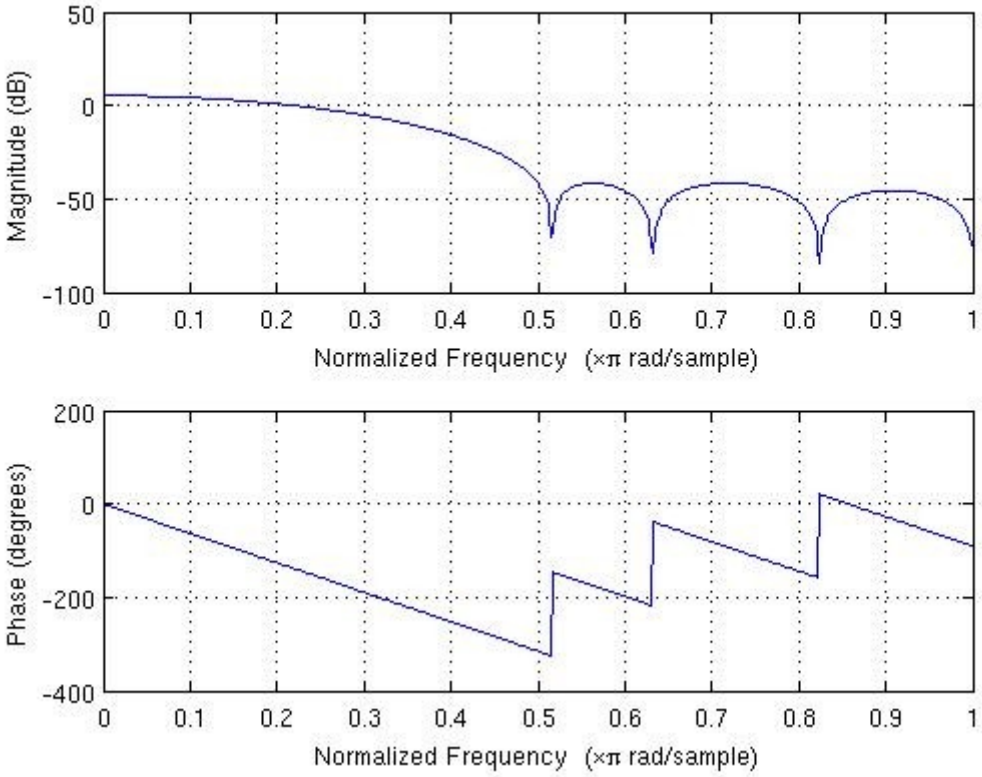
or

```
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).*

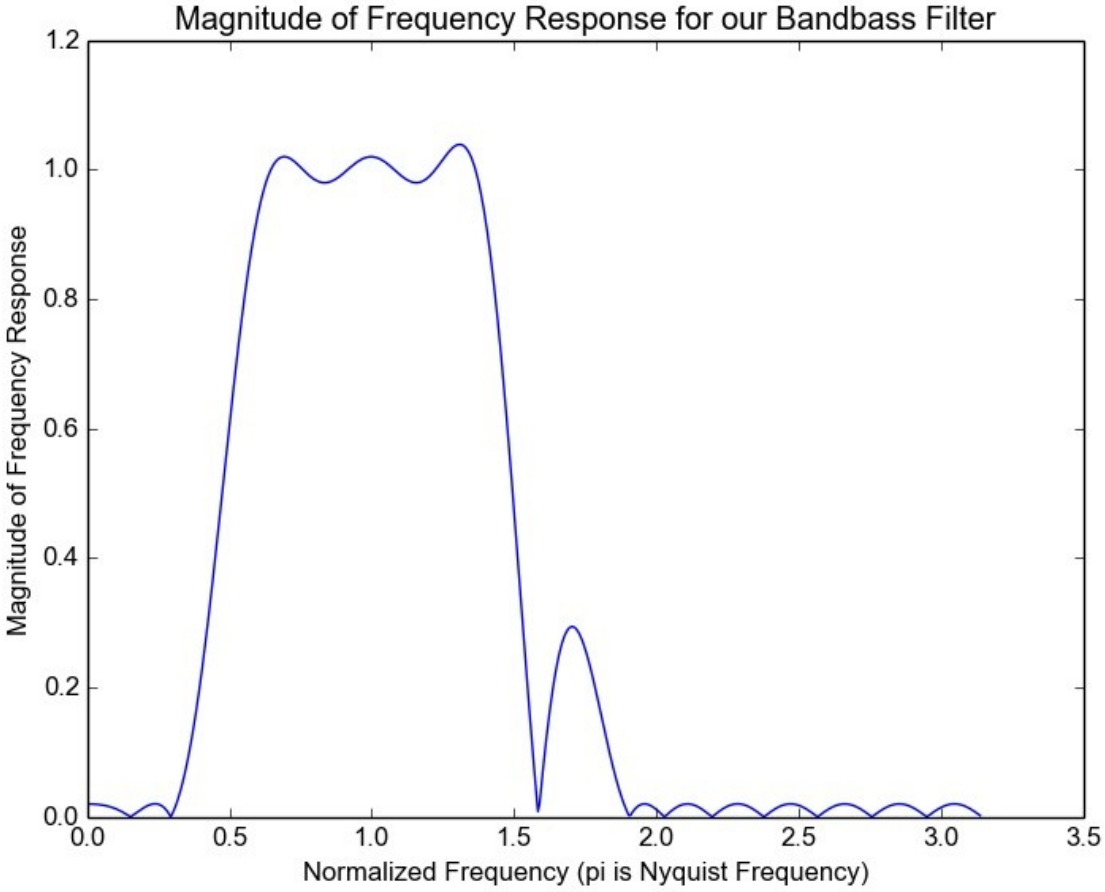
Observe: in scipy.signal.remez, by default the Nyquist frequency is 0.5 and the sampling frequency is 1!

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*8000=**800Hz** and 0.2*8000=**1600 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 Bandbass 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:

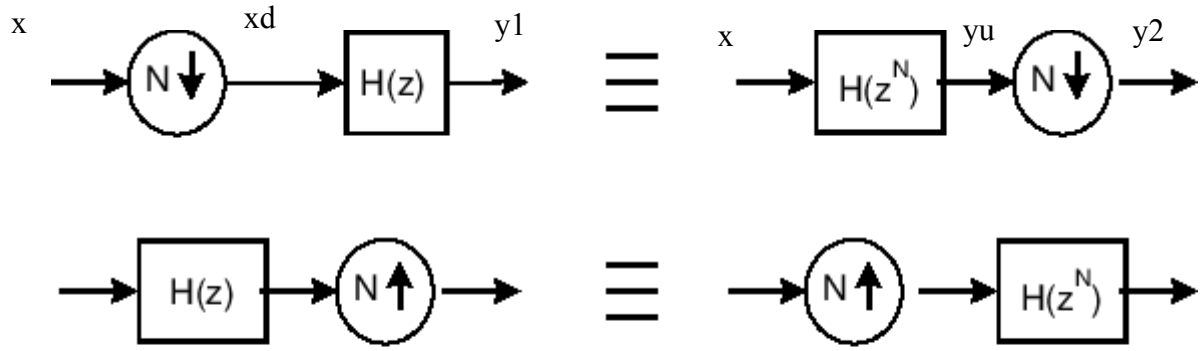
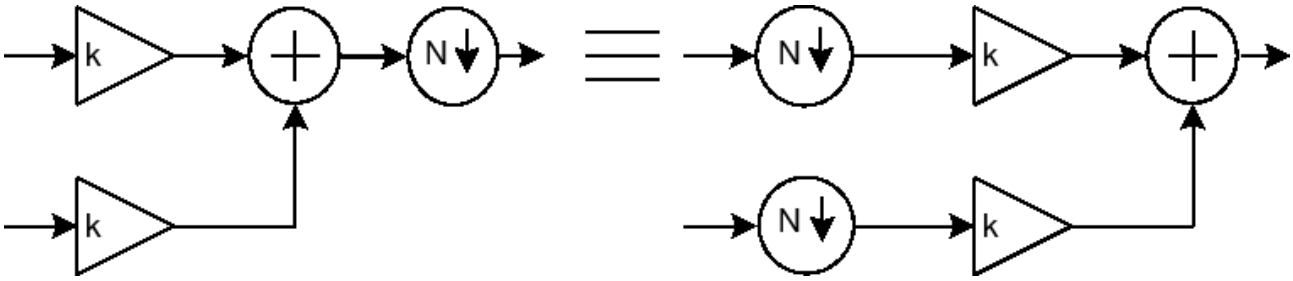


Figure 24.3: Multirate noble identities



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: upsamling 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 $hu=[1,0,2,0,3]$.

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,...] 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 *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 (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 $H_T(z)$:

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

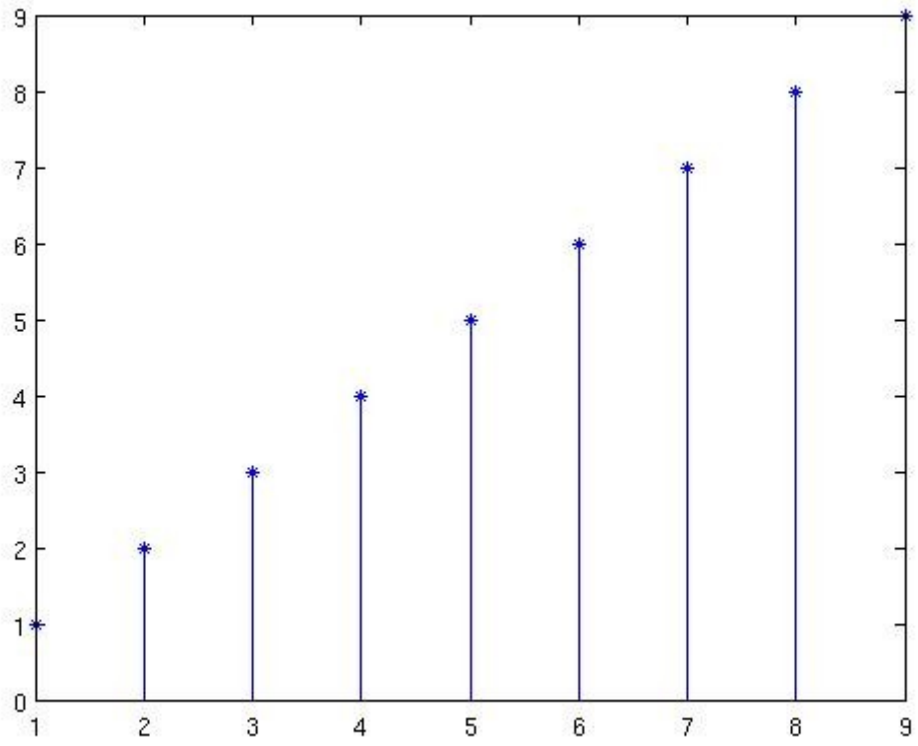
Here, $H_0(z^N)$ has all coefficients of positions at integer multiples of N, mN, at phase 0, $H_1(z^N)$ contains the coefficients at positions mN+1, at phase 1, and in general $H_i(z^N)$ contains the coefficients at positions mN+i, at phase i. This means :

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

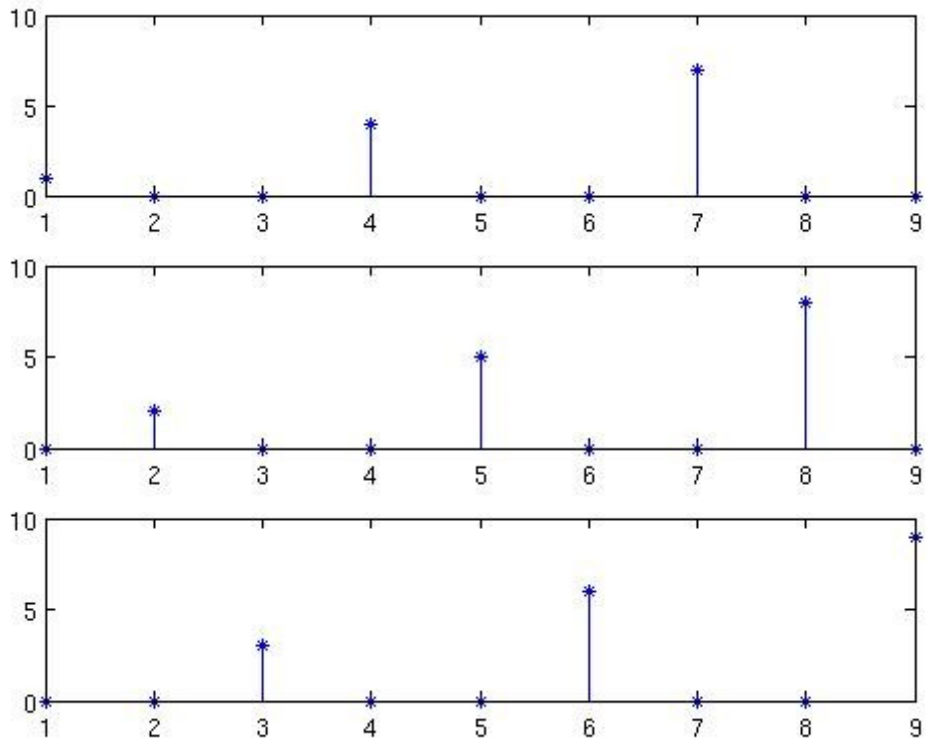
Since i can be seen as many different “phases” of our impulse response, the components

$H_i(z)$ 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 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

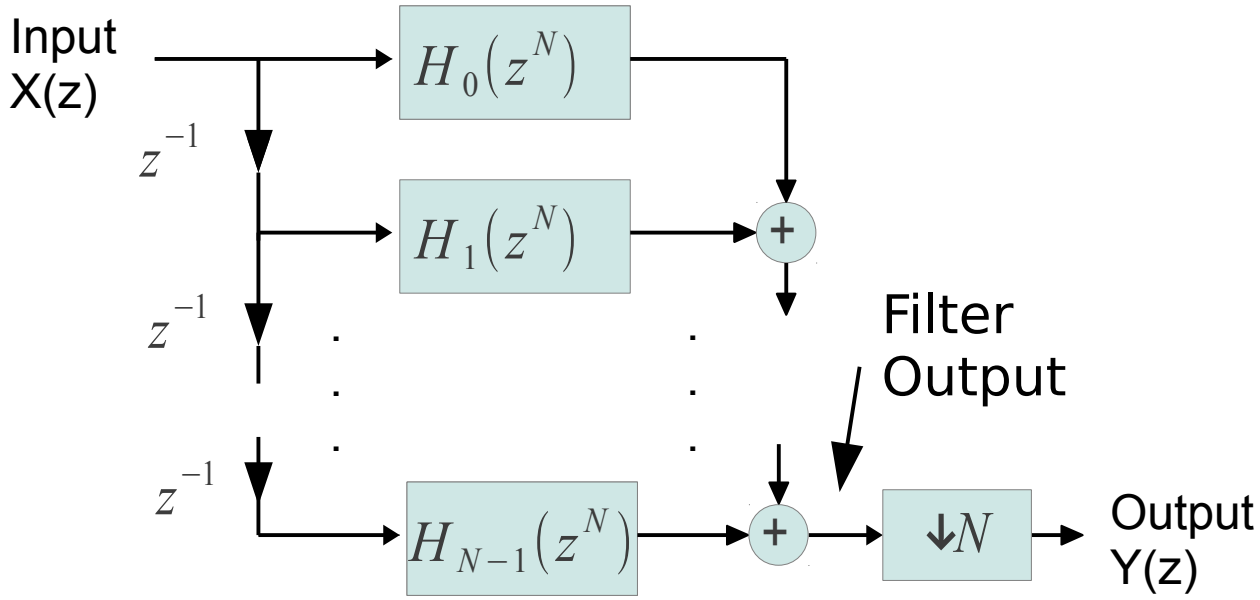
$$H_0(z^2) = 1 + 3z^{-2} + 5z^{-4} + \ldots$$

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

Hence we can combine the total filter from its polyphase components,

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

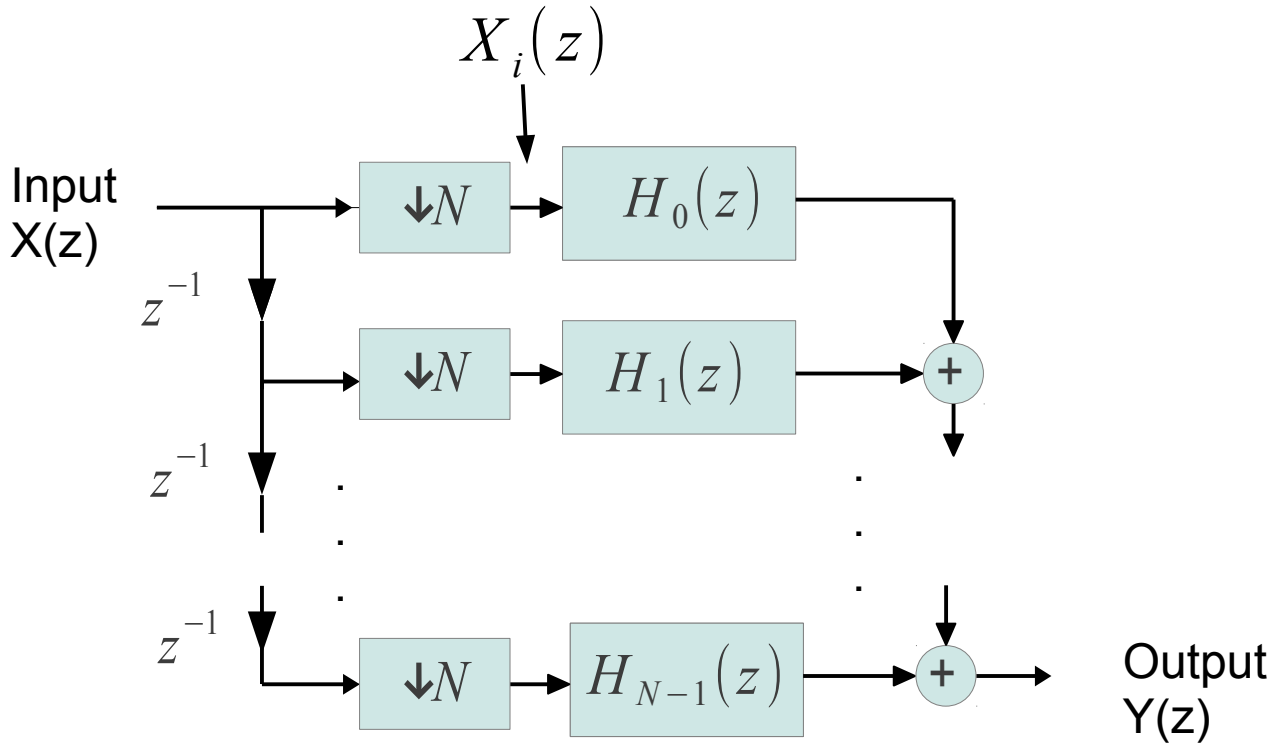
The general case is illustrated in the following block diagram, which consists of a delay chain on the left to implement the different delays z^{-i} , and the polyphase components $H_i(z^N)$ of the filter:



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 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,

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

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

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

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

in the time domain (in Matlab or Octave)

```
h0=[0.5 1.1] ; h1=[1 0.6];
```

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:

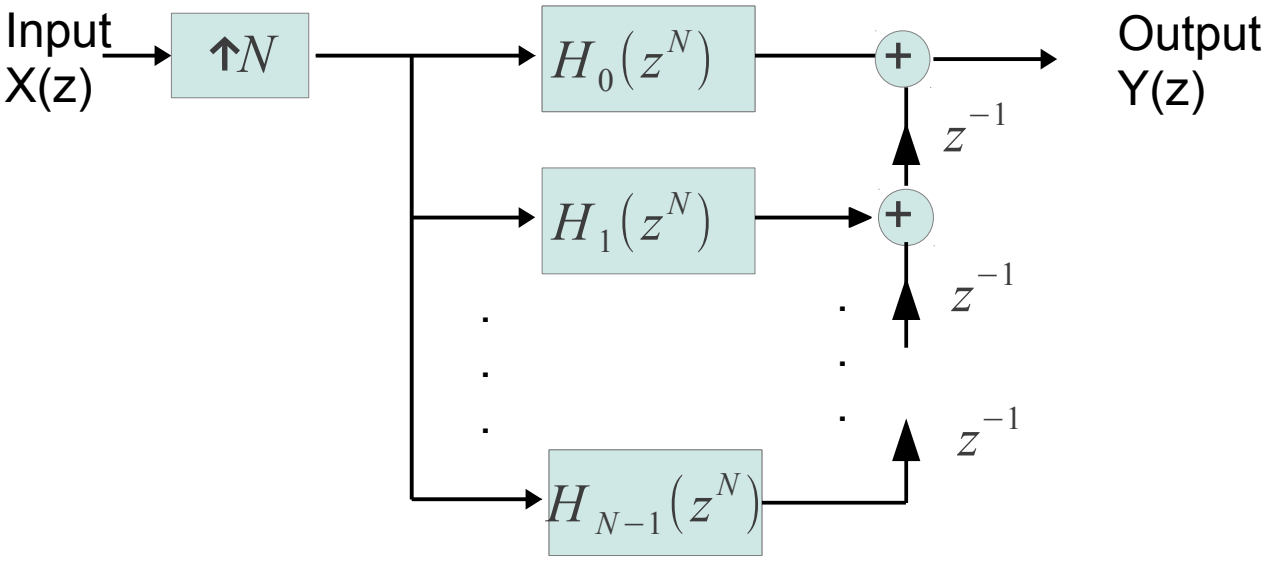
```
sound(y,4000);
```

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,

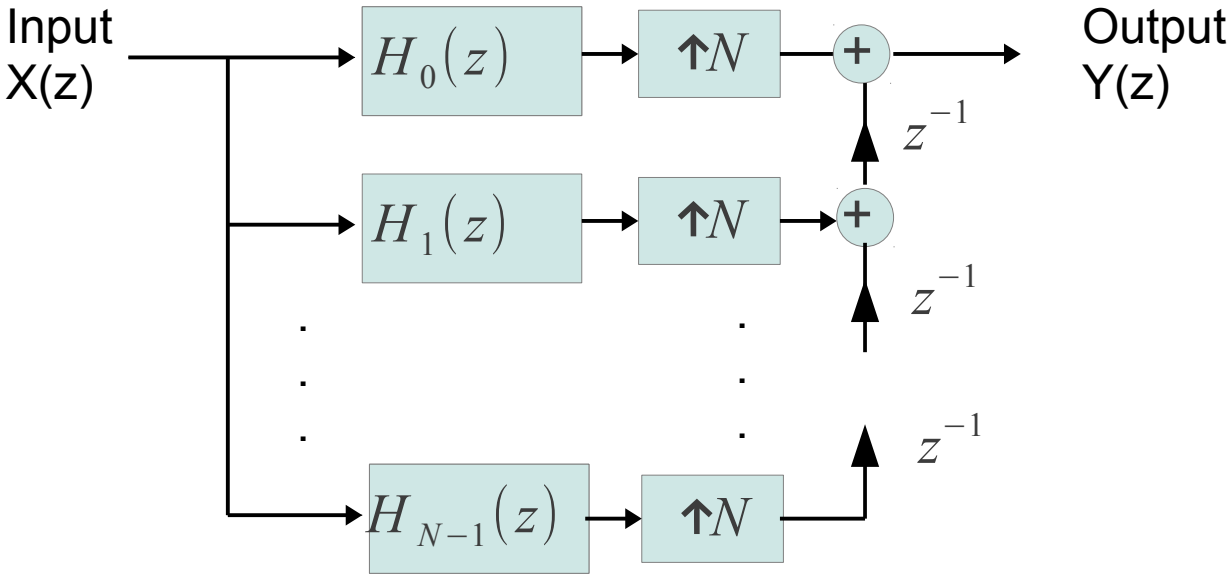
$$H_i(z^N)\cdot z^{-i}=z^{-i}\cdot H_i(z^N)$$

Hence we can redraw the polyphase decomposition for an up-sampler followed by a (e.g. low pass) filter (at the high samnpling 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**

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

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 `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,y); y1=filter(h1,1,y);
```

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

```
L=max(size(y));
yu(1:2:(2*L))=y0; yu(2:2:(2*L))=y1;
```

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

Now listen to the up-sampled signal:

```
sound(yu,16000);
```

Digital Signal Processing 2/ Advanced Digital Signal Processing, Audio/Video Signal Processing
Lecture 9,

Gerald Schuller, TU Ilmenau

Allpass Filters

So far we specified the magnitude of our frequency response and didn't care much about the phase. For allpass filters, it is basically the other way around.
In the beginning of filter design, we saw that we can write a transfer function as

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

Here we specify, or rather, alter the phase, and keep the the magnitude of our frequency response at constant 1, meaning

$$A\left(e^{j\Omega}\right)=1$$

Hence we would like to have a filter with transfer function H of magnitude constant 1,

$$\left|H\left(e^{j\Omega}\right)\right|=1$$

The simplest allpass filter has one pole and one zero in the z-domain for the transfer function,

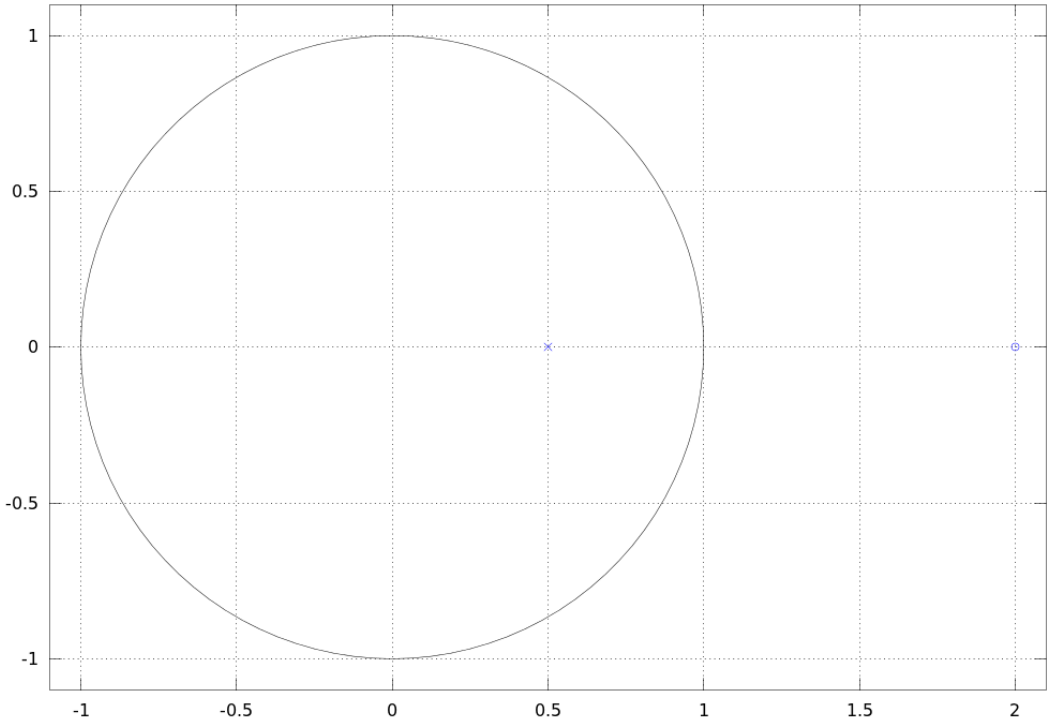
$$H_{ap}(z)=\frac{z^{-1}-\bar{a}}{1-a\,z^{-1}}$$

where a is a complex number, and $\bar{a}$ specifies the conjugate complex number.

Observe that here we have a zero at $z=\frac{1}{\bar{a}}$ and a pole at $z=a$! The **pole and the zero are at conjugate reverse locations!**

Example: If a=0.5, we obtain the pole/zero plot with Matlab/Octave,

```
a=0.5;
B=[-a' 1]; %the numerator polynomial of H_AP
A=[1 -a]; %the denominator polynomial
zplane(B,A); %plot the pole/zero diagram
axis([-1.1 2.1 -1.1 1.1],'equal') %have equal aspect ratio
```



In this plot, the cross at 0.5 is the pole, and the circle at 2 is the zero.

How can we see that the magnitude of the frequency response $H\left(e^{j\cdot\Omega}\right)$ is 1? We can re-write it as

$$H_{ap}\left(e^{j\Omega}\right)=\frac{e^{-j\Omega}-\bar{a}}{1-a\,e^{-j\Omega}}=e^{-j\Omega}\,\frac{1-\bar{a}\,e^{j\Omega}}{1-a\,e^{-j\Omega}}$$

Here you can see that the expression in the numerator is the conjugate complex of the denominator, hence their magnitude cancels to one. The exponential before the fraction also has magnitude 1, hence the entire expression has magnitude 1,

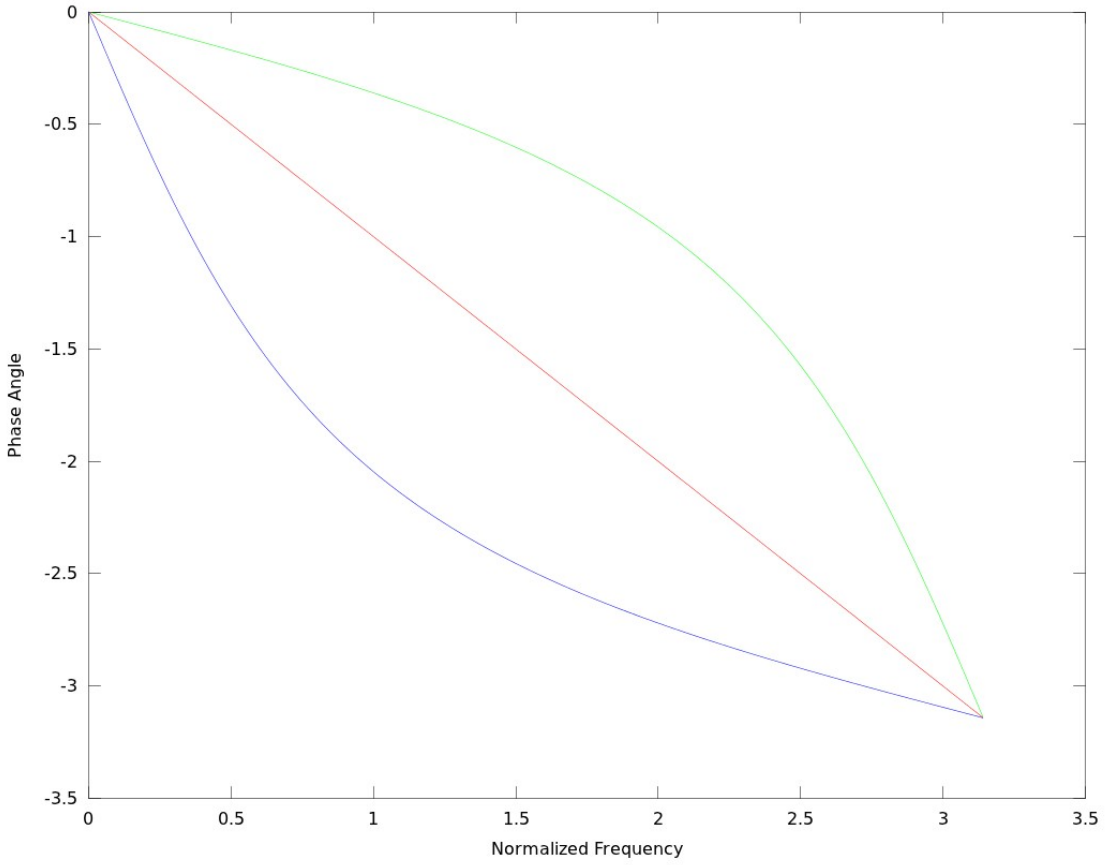
$$\left|H_{ap}\left(e^{j}\Omega\right)\right|=1$$

Here we can see, using just 1 pole and one zero, we can obtain a magnitude of constant 1. More interesting now is the resulting phase. The phase function can be found in the book Oppenheim/Schafer, “Discrete Time Signal Processing”:

$$\varphi(\Omega)=-\Omega-2\arctan\left(\frac{r\sin(\Omega-\theta)}{1-r\cos(\Omega-\theta)}\right)$$

where r is the magnitude of a and θ is the phase angle of a (hence $a=r\cdot e^{-j\theta}$). Observe that so far we assumed the phase to be linearly dependent on the frequency ($\varphi(\Omega)=-\Omega\cdot d$), and here we see it to be quite non-linear, with the trigonometric functions!

We can now plot the resulting phase over the normalized frequency, and compare it with the phase of a delay of 1 sample (of z^{-1}), where we get $\varphi(\Omega)=-\Omega$. This can be seen in the following plot, for $r=0.5$ and $r=-0.5$:



Here, the blue line is the allpass phase for r=0.5, the green line for r=-0.5, and the red line is for r=0, the phase of a pure 1 sample delay z^{-1} . Here it can be seen that the beginning and end of the curves are identical (at frequencies 0 and pi), and only in between the allpass phase deviates from the 1 sample delay! For a=0 the allpass indeed becomes identical to z^{-1} , a delay of 1 sample. So we can see that it behaves very **similar to a delay**.

The plot was produced with a simple Matlab/Octave function for the phase function,

```
function wy=warpingphase(w,a);
%produces phase wy for an allpass filter
%w: input vector of normlized frequencies (0..pi)
%a: allpass coefficient

%phase of allpass zero/pole :
theta=angle(a);
%magnitude of allpass zero/pole :
r=abs(a);
wy=-w-2*atan((r*sin(w-theta))./(1-r*cos(w-theta)));
```

The **phase** at the output of our phase function can also be **interpreted as a normalized frequency**.
An interesting observation is, that an allpass with coefficient $-\bar{a}$ is the inverse function of the allpass with coefficient a !
We can try this in Matlab/Octave.

```
%frequency range:
w=(0:0.01:pi);
a=0.5*(1+i)
wyy=(warpingphase(warpingphase(w,a),-a'));
plot(w,wyy)
xlabel('Normalized Frequency')
ylabel('Phase Angle')
```

If we define

$$H_{ap}(a, z)$$

as the z-transform of our allpass with coefficient a. If we use $z = e^{j\Omega}$ as its input (like for obtaining the frequency response). We know:

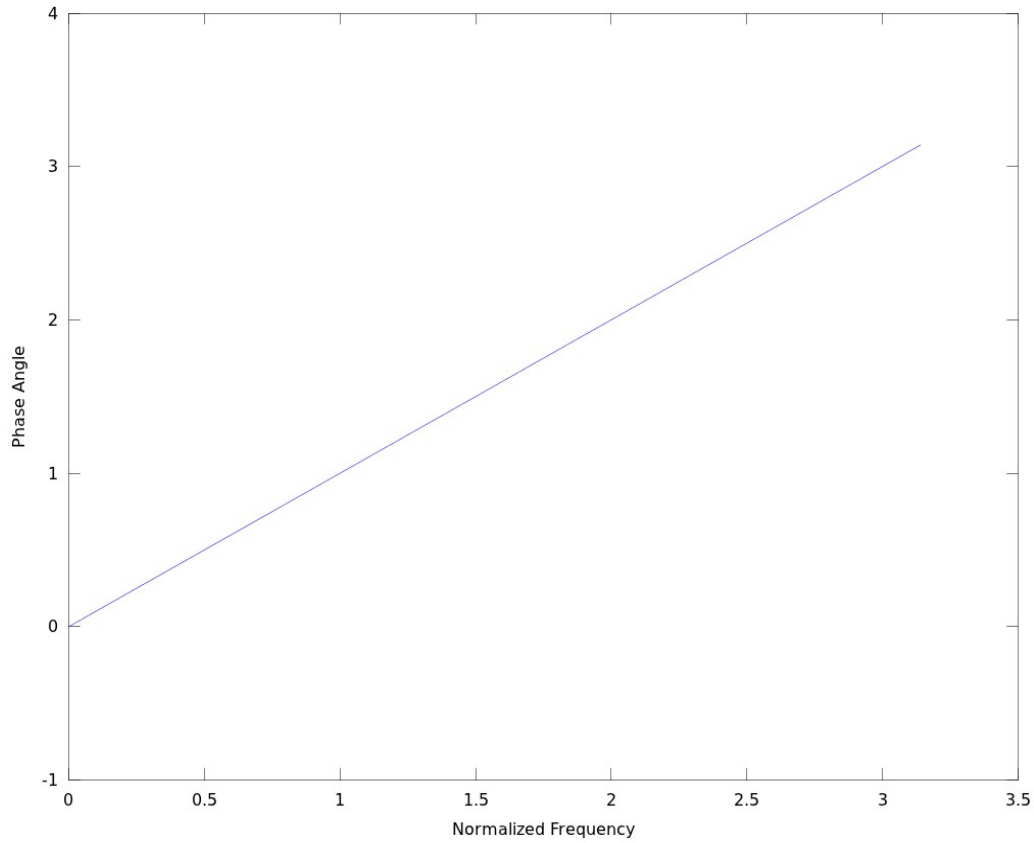
$$H_{ap}(e^{j\Omega}) = e^{j\varphi(\Omega)} \cdot A(e^{j\Omega}) = e^{j\varphi(\Omega)}$$

so replacing $z \leftarrow H_{ap}(a, z)$ and then setting $z = e^{j\Omega}$ for the frequencency response is like the total replacement $z = e^{j\varphi(\Omega)}$ where we now have this non-linear function $\varphi(\Omega)$ for the phase.

This function plots the **phase angle** of the frequency response of the concatenated system, where we replaced the z by our allpass, and in this way obtain our non-linear phase input,

$$H_{ap}(a, H_{ap}(-\bar{a}, e^{j\Omega}))$$

Observe that the **output** of the first allpass is i**nterpreted as frequency** here. The resulting plot is



Here we see that it is the **identity** function. This shows that interpreting the allpass as a normalized frequency “warper”, the allpass with a is inverse to the allpass with -a'.

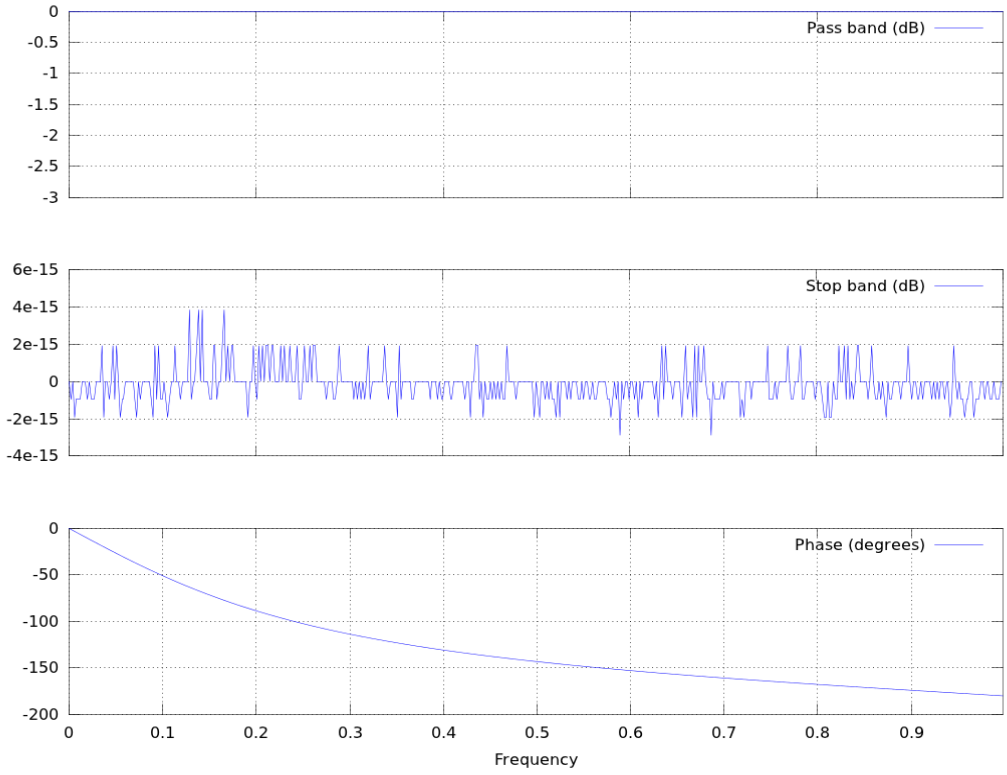
What is the frequency response of an example allpass filter? For a=0.5, we can use freqz. Looking at the z-transform

$$H_{ap}(z)=\frac{z^{-1}-\bar{a}}{1-a\,z^{-1}}$$

we get our coefficient vectors to

```
a=0.5;
B=[-a' 1];
A=[1 -a];
```

(observe that for freqz the higher exponents of z^-1 appear to the right)
Now plot the frequency response and impulse response:
freqz(B,A);
And we get

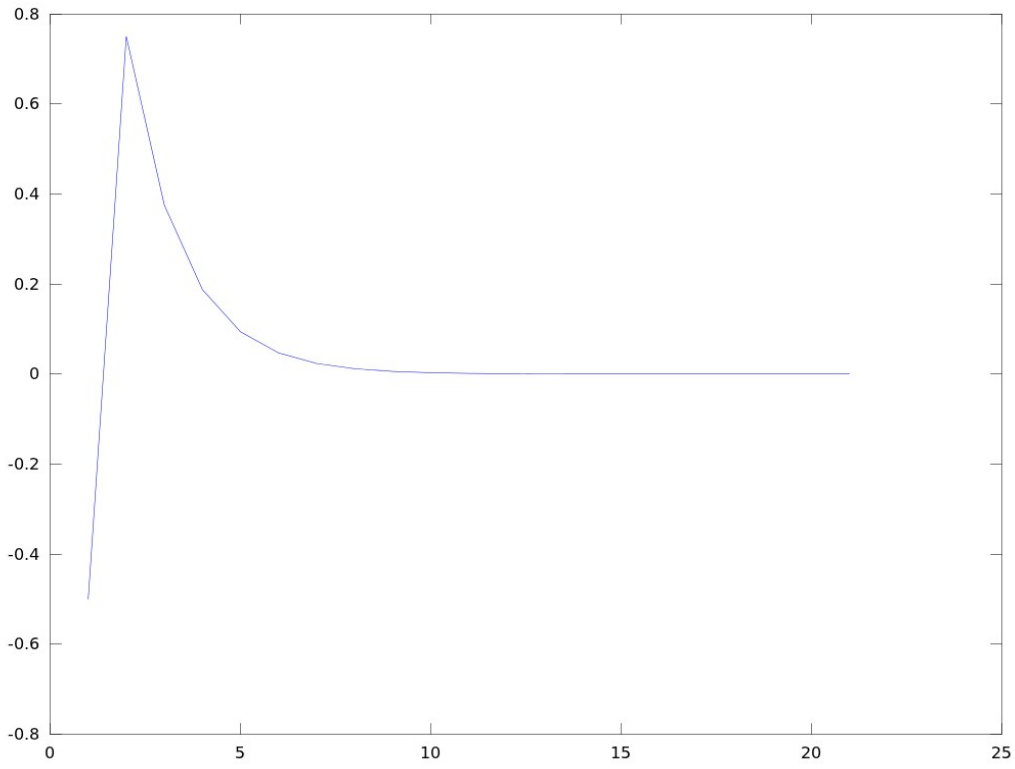


Here we can see in the above plot of the magnitude, that we indeed obtain a constant 1 (which is 0 dB), and that we have the **non-linear** phase in the lower plot, as in the phase plots before.

To obtain the impulse response, we can use the function “filter”, and input a unit impulse into it.

```
Imp=[1,zeros(1,20)];
h=filter(B,A,Imp);
plot(h);
```

we obtain the following impulse response plot



Here we can see that we have the first, non-delayed, sample not at zero, but at -0.5. This

can also be seen by plotting the first 4 elements of our impulse response:

```
h(1:4)
ans =
-0.50000 0.75000 0.37500 0.18750
```

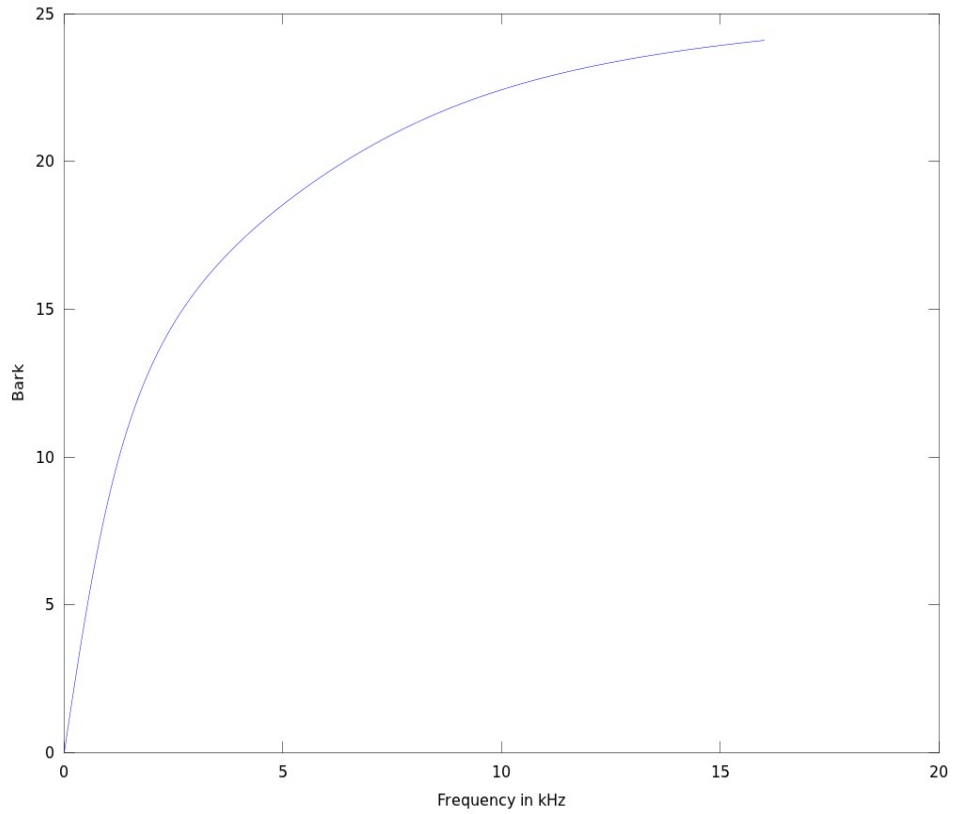
The second element corresponds to the delay of 1 sample, our z^{-1} , with a factor of 0.75. But then there are more samples, going back into the past, exponentially decaying. This means, not only the past samples goes into our filtering calculation, but also more past samples, and even the **non-delayed** sample, with a factor of -0.5. This is actually a problem for the so-called frequency warping (next section), if we want to use frequency warping in IIR filters, because here we would get delay-less loops, which are difficult to implement! (With **FIR filters** this is **no problem** though)

Frequency Warping

These properties of the allpass can now be used to “warp” the frequency scale of a filter (by effectively replacing $e^{j\Omega} \leftarrow e^{j\varphi(\Omega)}$ in our frequency response), for instance to map it according to the so-called **Bark scale**, used in psycho-acoustics.
A common **approximation** of the Bark scale is

$$Bark = 13 \cdot \arctan\left(0.0076 \cdot f\right) + 3.5 \cdot \arctan\left(\left(f/7500\right)^2\right)$$

(From Wikipedia, Bark scale), where f is the frequency in Hz. The Bark scale can be seen as an approximation of the changing frequency resolution over frequency of the inner ear filters of the human cochlea.
Because of the structure of our chochlea, the ear has different sensitivites for different frequencies and different signals. The signal dependent threshold of audibility of the ear is called the **Masking Threshold.** It has more spectral detail at lower than at higher frequencies, according to the Bark scale.



Here we can see, that 1 bark at lower frequency has a much lower bandwidth than at

higher frequencies. This means the ear can be seen as having a higher frequency resolution at lower frequencies than at higher frequencies. Imagine, we want to **design a filter** or system for **hearing purposes**, for instance, we would like to model the masking threshold of the ear for any given signal by some linear filter (FIR or IIR). Then it would be useful, to give this filter a **higher frequency resolution at lower frequencies**, such that it matches the smaller details of the **masking threshold** at lower frequencies. Then it would be useful, to give this filter a **higher frequency resolution at lower frequencies**, such that it matches the smaller details of the masking threshold at lower frequencies. But if we look at **the usual design methods**, they distribute the filter de**tails independent of the frequency range** (for instance what we saw with the remez method, where we have equally distributed ripples). Here we can now use frequency warping, such that we **enlarge the low frequency range** and shrink the high frequency range accordingly, such that our filter now works on the **warped frequency**, and **“sees” the lower frequencies in more detail**, the lower frequencies are more spread out in comparison to the higher frequencies.

How do we do this? For some frequency response $H\left(e^{j\Omega}\right)$ we would like to warp the

frequency Ω with some function $\varphi(\Omega)$ according to our desired frequency scale, such that we get

$$H\left(e^{j\cdot\varphi(\Omega)}\right)$$

But this is exactly the principle of an **allpass filter**, which has the frequency response

$$H_{ap}\left(e^{j\Omega}\right)=e^{j\cdot\varphi_{ap}(\Omega)}$$

Usually we would like to map positive frequencies to again positive frequencies, and we saw that $\varphi_{ap}(\Omega)$ becomes negative, hence we take the approach to **replace** z in the argument of our transfer function with the reverse of our **allpass** transfer function:

$$z^{-1}\leftarrow H_{ap}(a,z)$$

This is replacing all delays by our allpass filter. In this way we replace our linear function on the unit circle in z with the non-linear, warped function on the unit circle H_{ap} .

Hence we get the warped transfer function as

$$H_{warped}(z)=H\left(H_{ap}(a,z)^{-1}\right)$$

and the resulting frequency response becomes

$$H_{warped}\left(e^{j\Omega}\right)=H\left(e^{-j\cdot\varphi_{ap}(\Omega)}\right)$$

Here we can now see that we obtained the **desired frequency warping**.

What does this mean for the filter implementation? We know that our FIR filters always consist of many delay elements z^{-1} . Our replacement for z in the argument of our

transfer function means that we need to replace each delay element in our filter with

$$z^{-1} \leftarrow H_{ap}(a, z)$$

Example: Take an FIR filter,

$$H(z) = \sum_{m=0}^{L} b(m) \cdot z^{-m}$$

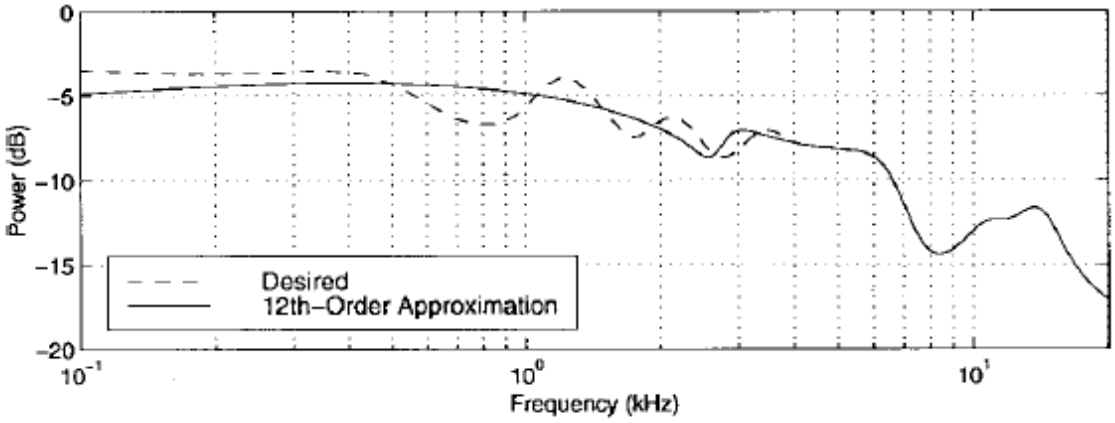
its warped version is:

$$H\left(H_{ap}(a, z)^{-1}\right) = \sum_{m=0}^{L} b(m) \cdot H_{ap}^{m}(a, z)$$

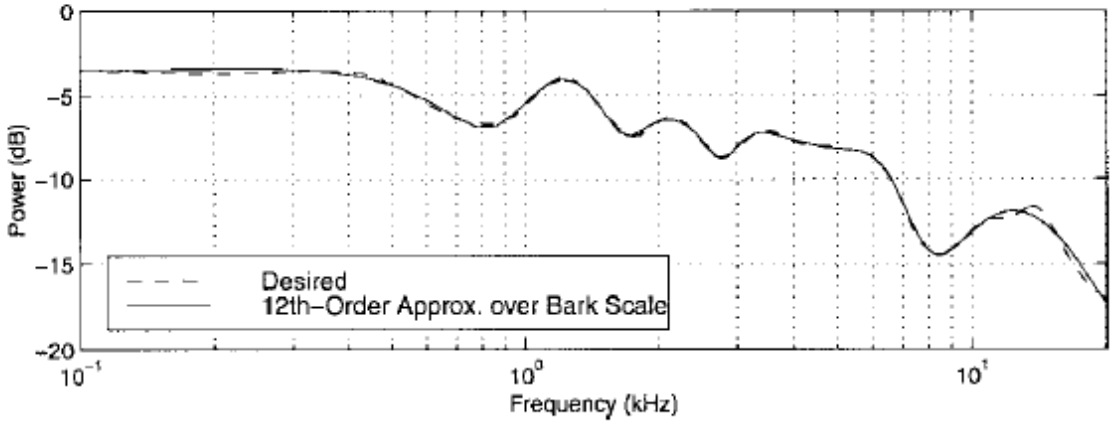
To obtain a desired filter, we now first have to **warp our desired filter,** and then **design** our filter in the **warped domain**.

Observe that the warping turns an **FIR filter into an IIR filter**, because the allpass has poles outside of zero.

An example of this kind of design can be seen in the following picture.



(a)



(b)

Fig. 10. Filter design example: Overlay of measured and modeled magnitude transfer functions, where the model is a twelfth-order filter designed by Prony's method. (a) Results without prewarping of the frequency axis. (b) Results using the Bark bilinear transform prewarping.

(From [1])
Here we can see that the 12th order filter successfully approximated the more detailed curve at low frequencies, using the warping approach.

[1] Julius O. Smith and Jonathan S. Abel, “Bark and ERB Bilinear Transforms,” IEEE Transactions on Speech and Audio Processing, vol. 7, no. 6, pp. 697 – 708, November 1999.

[2] S. Wabnik, G. Schuller, U. Kraemer, J. Hirschfeld: "Frequency Warping in Low Delay Audio Coding", IEEE International Conference on Acoustics, Speech, and Signal Processing, Philadelphia, PA, March 18–23, 2005

Digital Signal Processing 2/ Advanced Digital Signal Processing, Audio/Video Signal Processing
Lecture 10,

Gerald Schuller, TU Ilmenau

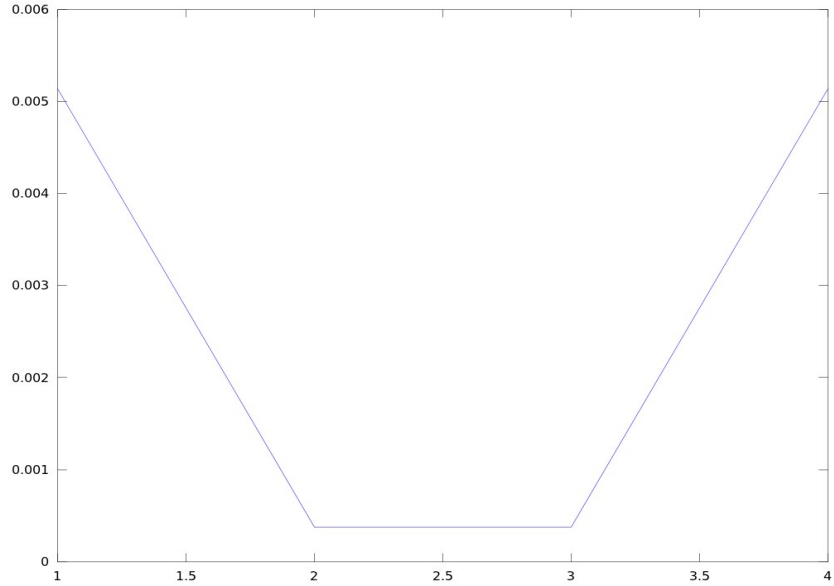
Frequency Warping, Example

Example: Design a **warped low pass filter** with cutoff frequency of 0.05*pi (pi is the Nyquist frequency). Observe: here this frequency is the end of passband, with frequency warping close to the Bark scale of human hearing.

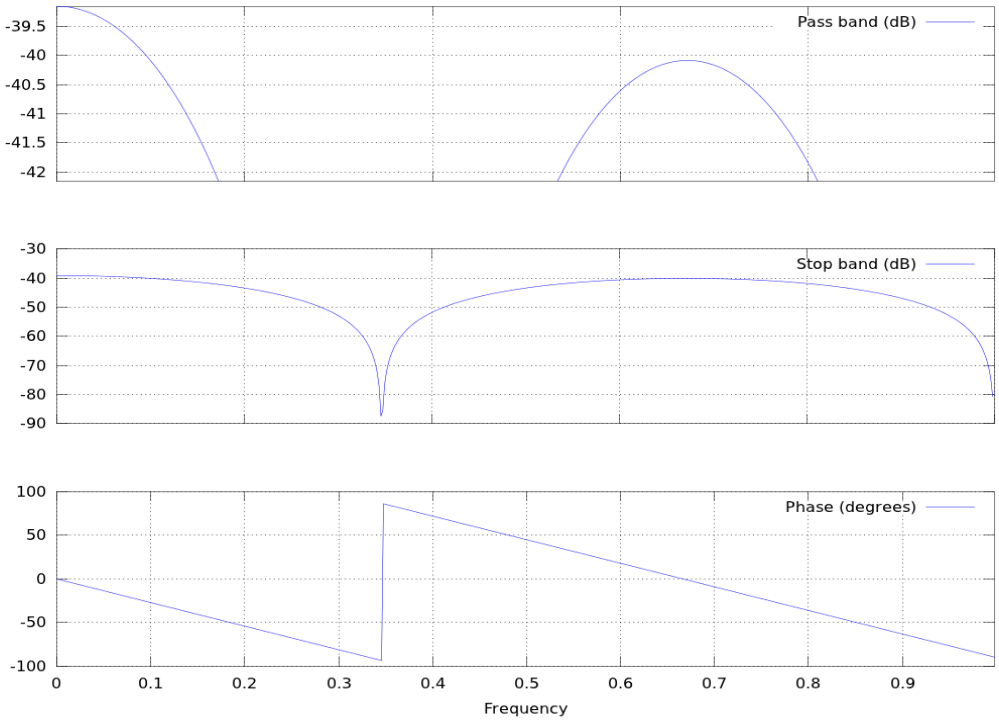
First as a comparison: design an **unwarped filter** with 4 coefficients/taps with these specifications:

In Matlab/Octave:

```
cunw=remez(3,[0 0.05, 0.05+0.05 1],[1 1 0 0],[1 100])
%cunw =
%   5.1365e-03
%   3.7423e-04
%   3.7423e-04
%   5.1365e-03

%impulse response:
plot(cunw)
```



%frequency response:
freqz(cunw,1);



Here we can see that this is not a good filter. The passband is too wide (up to about 0.15), and there is almost no stopband attenuation (in

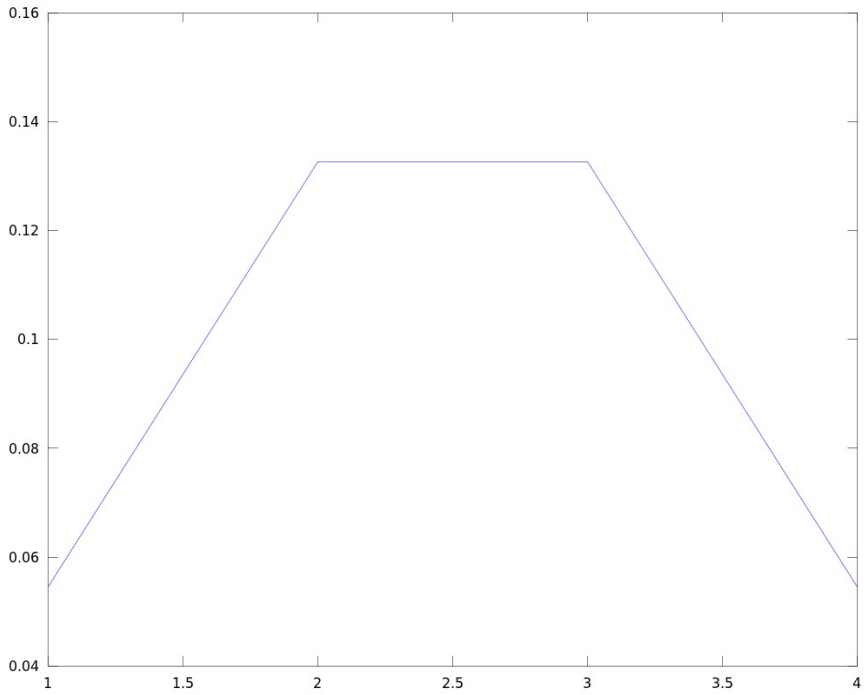
the range of 0.5 to 0.9). So this filter is probably **useless** for our application.

Now design the FIR low pass filter (4th order), which we then want to **frequency warp** in the next step, with a warped cutoff frequency. First we have to compute the allpass coefficient „a“ for our allpass filter which results in an approximate Bark warping, according to [1], eq. (26):

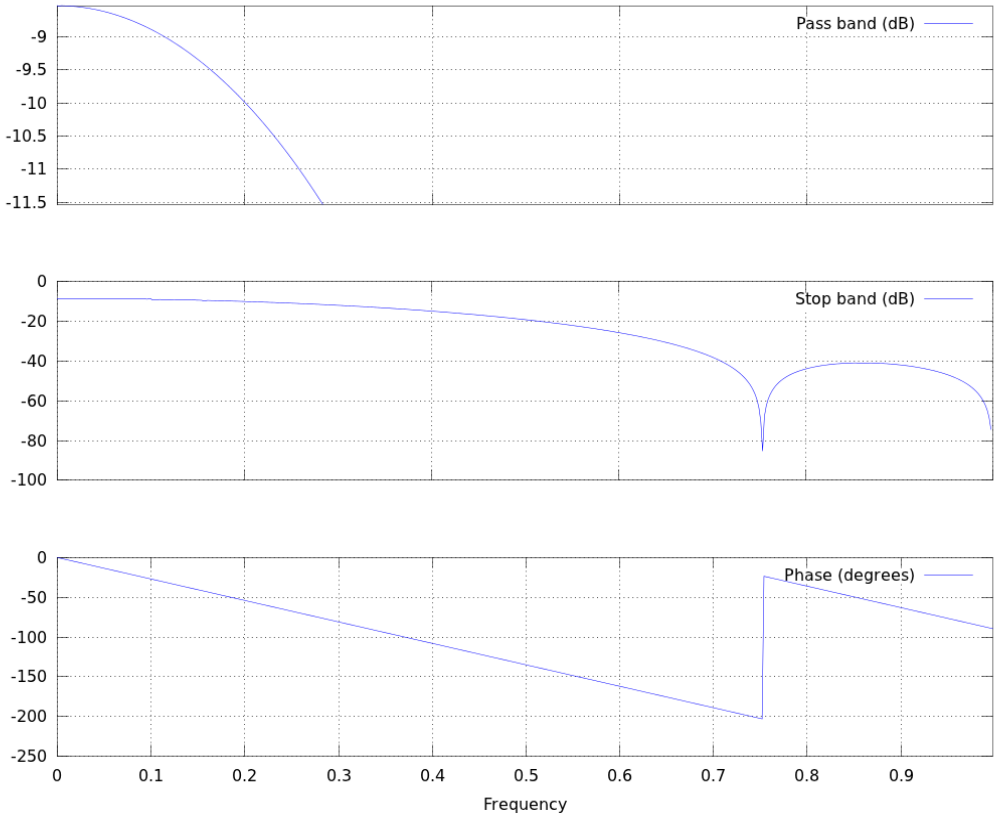
$$a = 1.0674 \cdot \left(\frac{2}{\pi} \cdot \arctan\left(0.6583 * f_s\right) \right)^{0.5} - 0.1916$$

with f_s the sampling frequency in kHz. Our warped design is then

```
%warping allpass coefficient:
a=1.0674*(2/pi*atan(0.6583*32))^0.5 -0.1916
%ans =  0.85956
%with f_s=32 in kHz. from [1]
%The warped cutoff frequency then is:
fcw=-warpingphase(0.05*pi,0.85956)
%fcw = 1.6120; %in radiants
%filter design:
%cutoff frequency normalized to nyquist:
fcny=fcw/pi
%fcny =  0.51312
c=remez(3,[0 fcny, fcny+0.2 1],[1 1 0 0],[1 100]);
%The resulting Impulse Response:
plot(c);
```

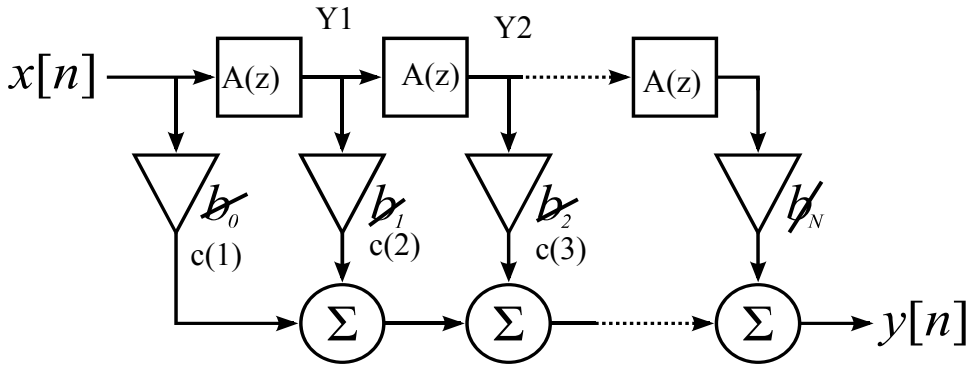


%The resulting Frequency response:
freqz(c,1);

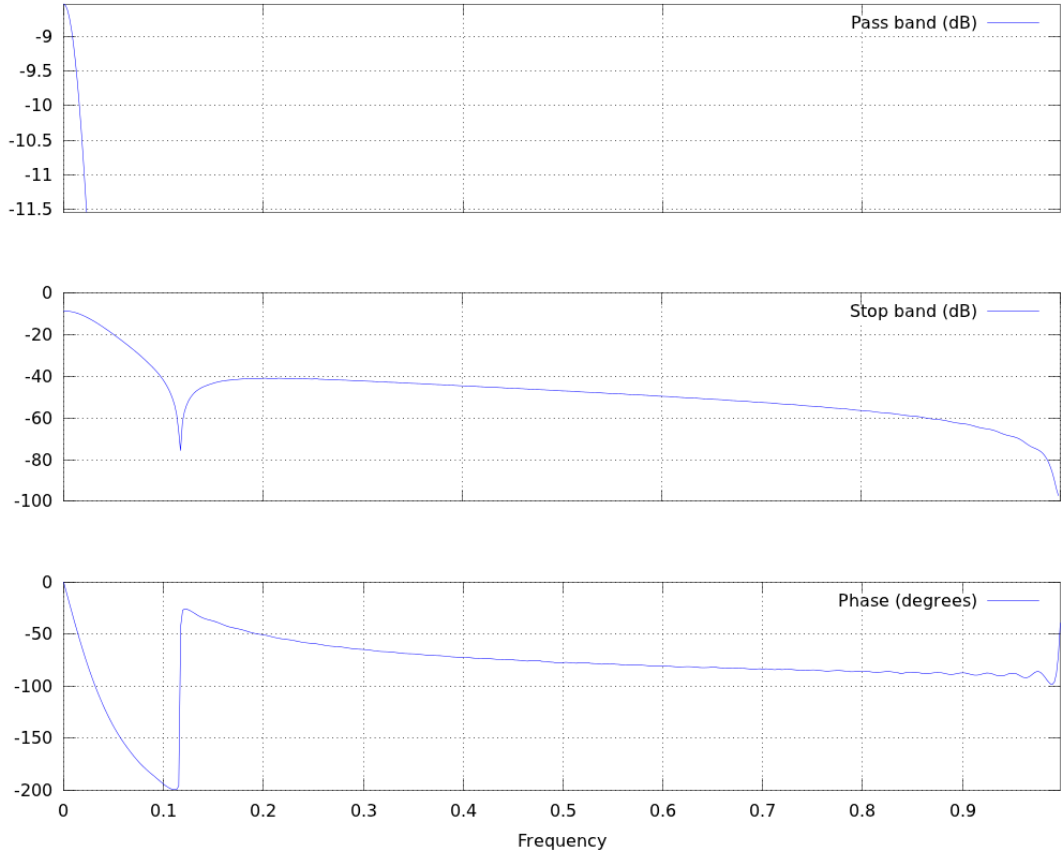


Here we can see that in the warped domain, we obtain a reasonable low pass filter. In the passband from 0 to somewhat above 0.5 it has a drop of about 10 dB, and in the stopband we obtain about -30 dB attenuation, which is much more than before (it might still not be enough for practical purposes though)

%Replace Delays in FIR filter with Allpass filter (in this way we go from frequency response H(z) to H(A(z)):

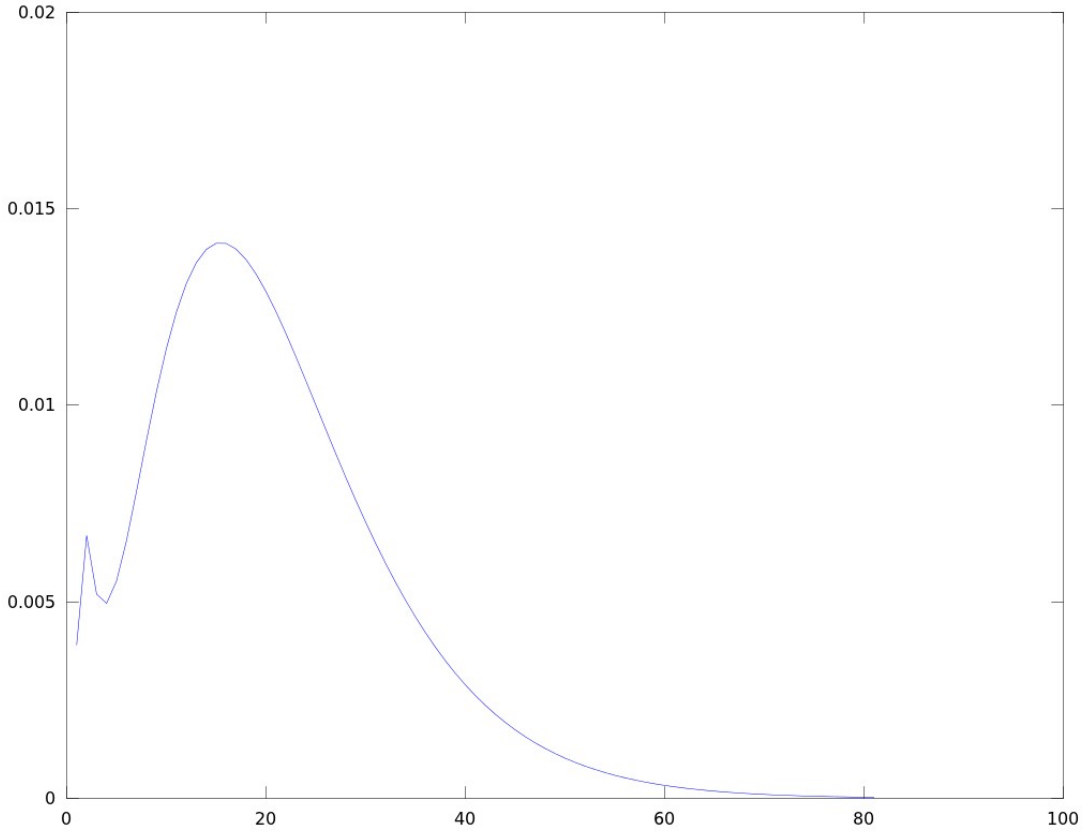



```
%Warping Allpass filters:
B=[-a' 1];
A=[1 -a];
%Impulse with 80 zeros:
Imp=[1,zeros(1,80)];
x=Imp;
%Y1(z)=A(z), Y2(z)=A^2(z),...
%Warped delays:
y1=filter(B,A,x);
y2=filter(B,A,y1);
y3=filter(B,A,y2);
%Output of warped filter with impulse as input:
yout=c(1)*x+c(2)*y1+c(3)*y2+c(4)*y3;
%frequency response:
freqz(yout,1);
```



Here we can now see the frequency response of our final warped low pass filter. We can see that again we have a drop of about 10 dB in the passband, now from 0 to 0.05pi, and a stopband attenuation of about 30dB, which is somewhat reasonable.

```
%Impulse response:
figure;
plot(yout);
```



This is the resulting impulse response of our warped filter. What is most obvious is its length. Instead of just 4 samples, as our original unwarped design, it easily reaches 80 significant samples, and in principle is infinite in extend. This is also what makes it a much better filter than the unwarped original design!

References:

[1] Julius O. Smith and Jonathan S. Abel, “Bark and ERB Bilinear Transforms,” IEEE Transactions on Speech and Audio Processing, vol. 7, no. 6, pp. 697 – 708, November 1999.

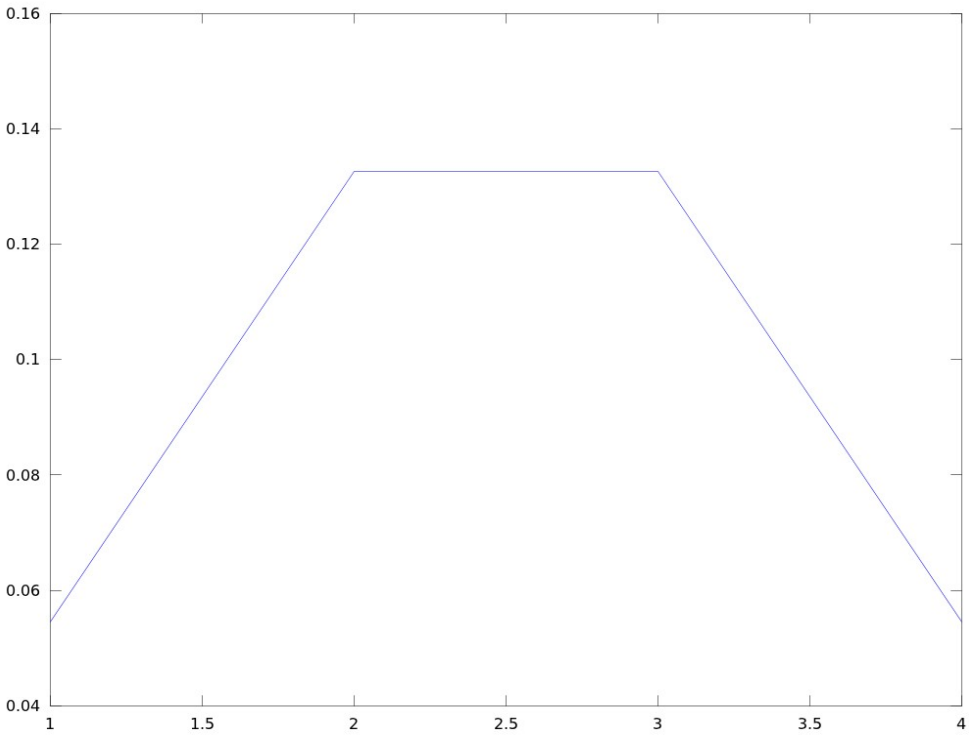
[2] S. Wabnik, G. Schuller, U. Kraemer, J. Hirschfeld: "Frequency Warping in Low Delay Audio Coding", IEEE International Conference on Acoustics, Speech, and Signal Processing, Philadelphia, PA, March 18–23, 2005

Minimum Phase Filters

Remember linear phase filters. Its phase function is linear:

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

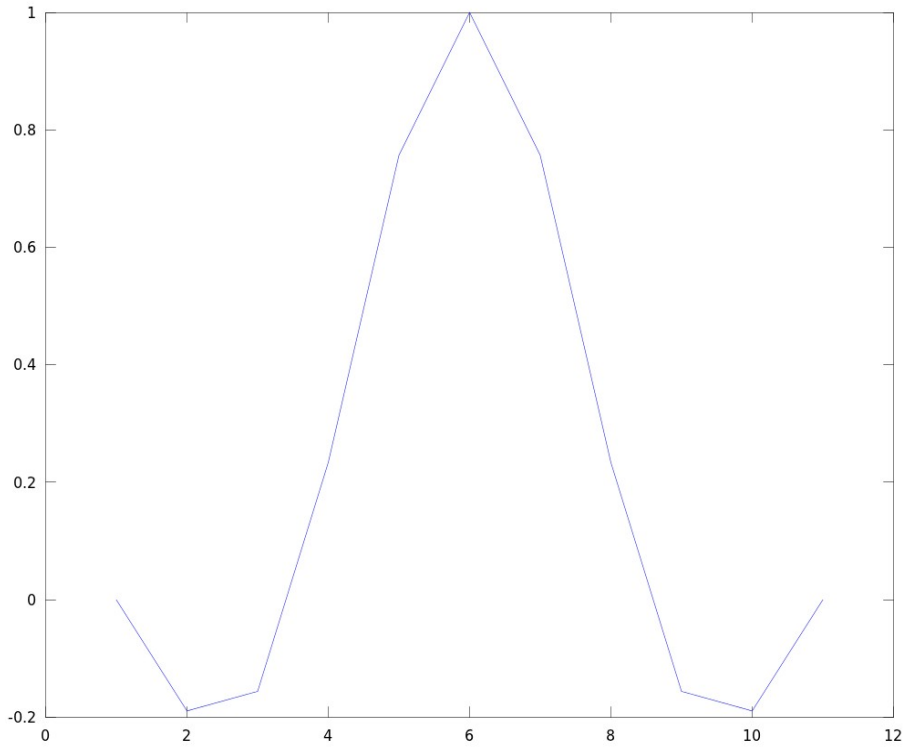
with a group delay of constant d. The impulse responses of linear phase filters have the property of being (even) symmetric around some center. Example:



Here we have a 4 sample impulse response, and starting at 0, we have a symmetry around d=1.5, hence we have a constant delay of this system of d=1.5 samples.

Another example for a linear phase filter is a piece of a sinc function. In Matlab/Octave:

```
hsinc=sinc(-2:0.4:2)
%hsinc =
%Columns 1 through 6:
%-3.8980e-17 -1.8921e-01 -1.5591e-01 2.3387e-01
%7.5683e-01 1.0000e+00
%Columns 7 through 11:
%7.5683e-01 2.3387e-01 -1.5591e-01 -1.8921e-01
%-3.8980e-17
plot(hsinc)
```



This FIR filter has a constant delay factor of d=5 (starting to count the samples at 0 instead of 1 in the plot).

The delay factor d is the center of the impulse response, because we can factor it out from the DTFT of the symmetric impulse response:

$$H\left(e^{j\Omega}\right)=\sum_{n=0}^{2\mathrm{d}} h\left(n\right)\cdot e^{-j\Omega n}$$

We factor out the center exponential,

$$H\left(e^{j\Omega}\right)=e^{-j\Omega d}\cdot\sum_{n=0}^{2\mathrm{d}} h\left(n\right)\cdot e^{-j\Omega\left(n-d\right)}$$

since h(d-n)=h(d+n) we get:

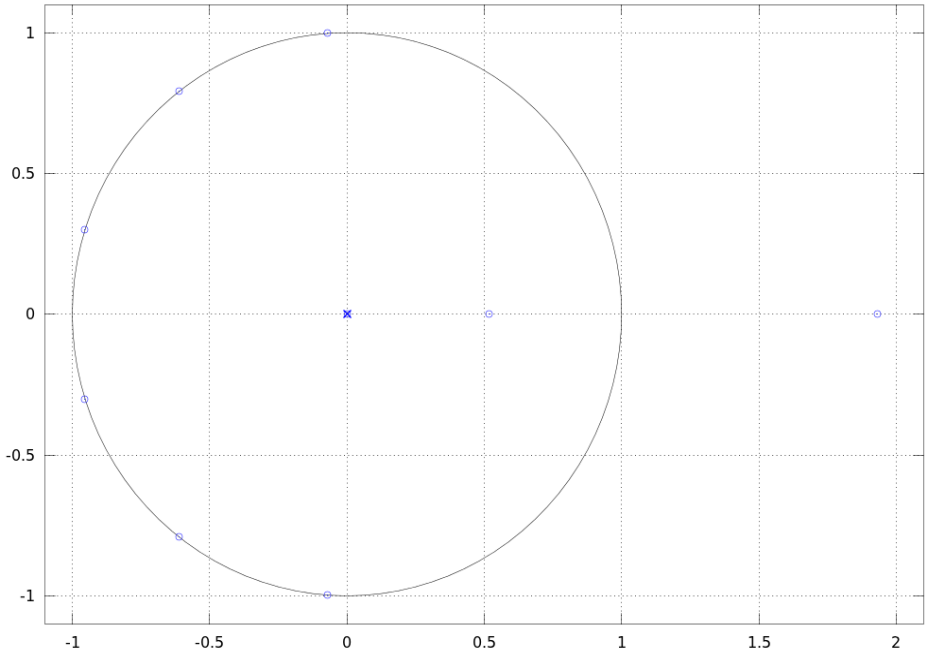
$$H\left(e^{j\Omega}\right)=e^{-j\Omega d}\cdot\sum_{n=0}^{d}h(n)\cdot\left(e^{-j\Omega(d-n)}+e^{-j\Omega(d+n)}\right)$$

$$H\left(e^{j\Omega}\right)=e^{-j\Omega d}\cdot\sum_{n=0}^{d}h(n)\cdot 2\cdot\cos\left(\Omega(d-n)\right)$$

Hence the phase is: $\varphi(\Omega)=-d\,\Omega$

Now we can plot its zeros in the zplane, using the command “zplane”:

```
zplane(hsinc,1)
axis([-1.1 2.1 -1.1 1.1],'equal')
```



Observe the zeros near 1.9 and near 0.5, and on the unit circle.
Its zeros are computed with the command “roots”, and their magnitude with “abs”:

```
abs(roots(hsinc))
ans =
4.8539e+15
1.9309e+00
1.0000e+00
1.0000e+00
1.0000e+00
1.0000e+00
1.0000e+00
1.0000e+00
5.1789e-01
2.0601e-16
```

Here we can see that we have one zero at location 0, and one at infinity, 6 zeros are on the unit circle, one 1 at distance 1.9309 from the origin, and one is at distance `5.1789e-01=1/1.9309.`
Hence for those 2 zeros we have one zero inside the unit circle at distance r, and one outside the unit circle at distance 1/r.

Linear phase systems and filters have the property, that their **zeros are inside and outside the unit circle** in the z-domain. For stabilty, only poles need to be inside the unit circle, not the zeros. But if we want to invert such a filter (for instance for equalization purposes), the zeros turn into poles, and the **zeros outside the unit** circle turn into **poles outside the unit circle**, making the **inverse filter unstable**!

To avoid the instability of the inverse filter, we define **minimum phase filters** such that their **inverse is also stable**!
This means, all their **zeros need to be inside the unit circle** in the z-domain.

We can write all linear filters as a concatenation of a minimum phase filter with an allpass filter,

$$H(z) = H_{min}(z) \cdot H_{ap}(z)$$

This can be seen from a (hypothetical) minimum phase system $H_{min}(z)$, which has all its zeros inside the unit circle. Now we concatenate/multiply it with an allpass filter, such that its poles coincide with some of the zeros inside the unit circle. These poles and zeros then cancel, and what is left is the zeros outside the unit circle at a reverse conjugate position $1/a'$, if “a” was the position of the original zero. In this way, we can **„mirror out“ zeros from inside the unit circle to the outside.** The **magnitude response does not change**, because we used an allpass for mirroring out the zeros. As a result we have a system with the same magnitude response, but now with zeros outside the unit circle.

Assume we would like to equalize or compensate a given transfer function, for

instance from a recording. As we saw above, this transfer function can be written as the product

$$H(z)=H_{min}(z)\cdot H_{ap}(z)$$

Only $H_{min}(z)$ has a stable inverse. Hence we design our compensation filter as

$$H_c(z)=\frac{1}{H_{min}(z)}$$

If we apply this compensation filter after our given transfer function, for instance from a recording, we obtain the overall system function as

$$G(z)=H(z)\cdot H_c(z)=H_{ap}(z)$$

This means the overall transfer function now is an allpass, with a constant magnitude response and only phase changes.
(see also A. Oppenheim, R. Schafer: “Discrete Time Signal Processing”, Prentice Hall)

How can we **obtain a minimum phase version** from a given filter? We basically “mirror in” the zeros outside the unit circle.
Take our above example of the piece of the sinc function filter.
In Matlab/Octave we compute the zeros with

```
rt=roots(hsinc)
```

```
rt =
-4.8539e+15 + 0.0000e+00i
1.9309e+00 + 0.0000e+00i
-9.5370e-01 + 3.0077e-01i
```

```
-9.5370e-01 - 3.0077e-01i
-6.1157e-01 + 7.9119e-01i
-6.1157e-01 - 7.9119e-01i
-7.1160e-02 + 9.9746e-01i
-7.1160e-02 - 9.9746e-01i
5.1789e-01 + 0.0000e+00i
-2.0601e-16 + 0.0000e+00i
```

We see the zero at 1.93 which we need to mirror in (we neglect the zero at infinity, which comes from starting with a zero sample). To achieve this, we first take the z-domain polynomial of the impulse response, and cancel that zero by dividing through the polynomial with only that zero, $1-1.93\cdot z^{-1}$. Fortunately we have the function “deconv”, which is identical to polynomial division, to do this:

```
[b, r] = deconv (hsinc, [1,-rt(2)])
b =
Columns 1 through 6:
-3.8980e-17 -1.8921e-01 -5.2126e-01 -7.7264e-01
-7.3509e-01 -4.1941e-01
Columns 7 through 10:
-5.3021e-02 1.3149e-01 9.7987e-02 -8.9291e-09
r =
Columns 1 through 6:
0.0000e+00 0.0000e+00 0.0000e+00 0.0000e+00
1.1102e-16 0.0000e+00
Columns 7 through 11:
0.0000e+00 0.0000e+00 0.0000e+00 0.0000e+00
-1.7241e-08
```

Here, r is the remainder. In our case it is practically zero, which means we can indeed divide our polynomial without any remainder.

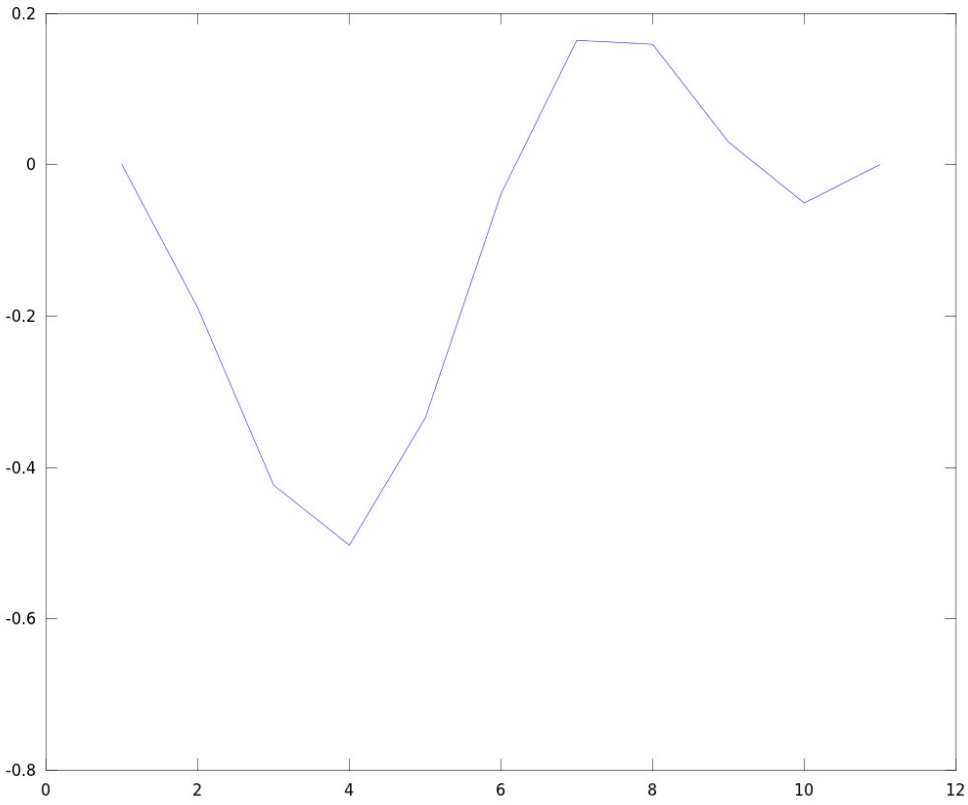
After that we can multiply the obtained polynomial b with the zero inside the unit circle, at position 1/1.93, by multiplying it with the polynomial with only that zero: $1-1/1.93\cdot z^{-1}$:

```
hsincmp=conv(b,[1,-1/rt(2)'])
```

```
hsincmp =
Columns 1 through 6:
-3.8980e-17 -1.8921e-01 -4.2327e-01 -5.0269e-01
-3.3495e-01 -3.8715e-02
Columns 7 through 11:
1.6418e-01 1.5895e-01 2.9889e-02 -5.0746e-02
4.6242e-09
```

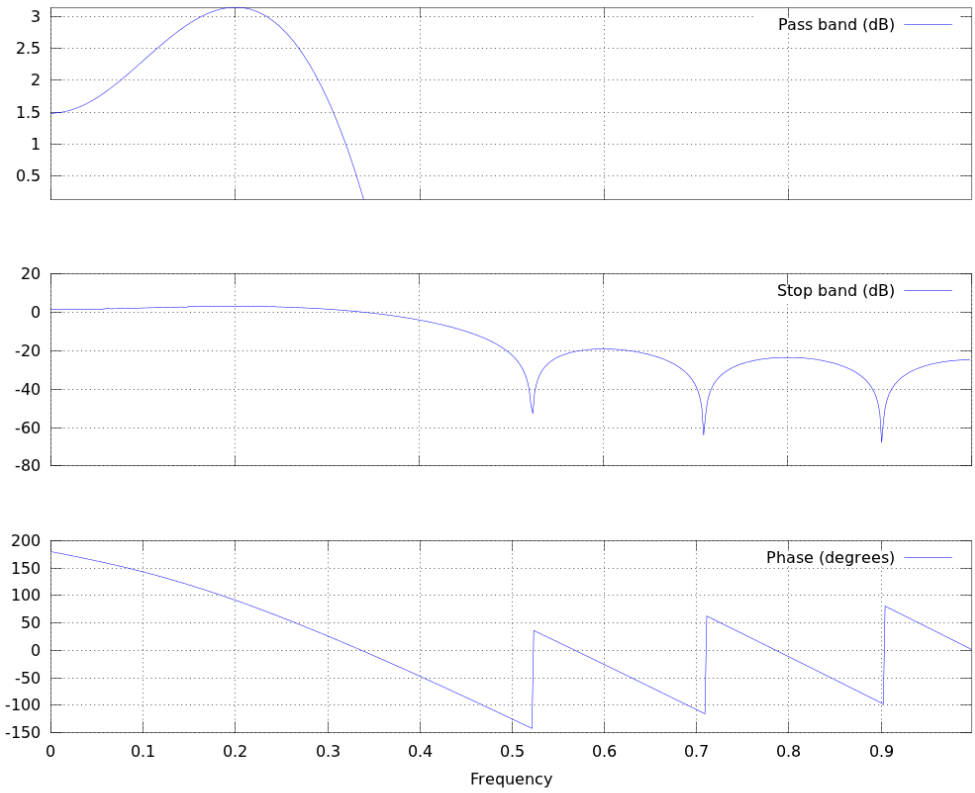
This hsincmp is now our **minimum phase version** of our filter!
Now we can take a look at the impulse response:

```
plot(hsincmp)
```



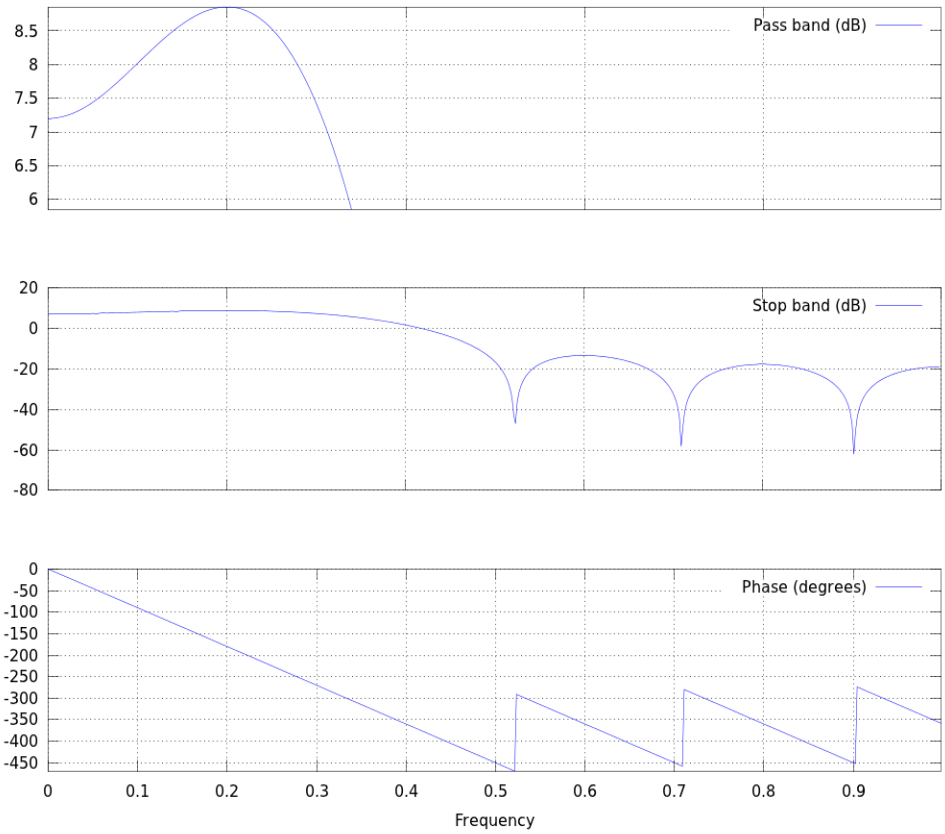
Observe that our filter now became **non-symmetric**, with the main peak at the beginning of the impulse response!
The resulting frequency response is obtained with

`freqz(hsincmp)`



Now compare the above frequency response of our minimum phase filter with the linear phase version, with

```
figure
freqz(hsinc)
```



Here we can see that the magnitude of the frequency plot is indeed identical between the linear phase and the minimum phase version (except for an offset of about 5 dB, which is not important because it is a constant gain factor). But looking at the phase, we see that the minimum phase version has less phase lag. Looking at frequency 0.5, we see that the linear phase filter has a phase lag of about 450 degrees, whereas, the minimum phase filter has a **reduced phase lag** of about 300 degrees (from frequency zero to 0.5)!
If we take the derivative of the phase function to obtain the group delay, we will get correspondingly lower values, which means the minimum phase filter will have **less group**

delay than the linear phase filter. In fact, it has the **lowest possible delay for the given magnitude response** of the filter. So if you have a given magnitude filter design, and want to obtain the **lowest possible delay**, you have to take **minimum phase filters**.

Digital Signal Processing 2/ Advanced Digital Signal Processing, Audio/Video Signal Processing
Lecture 10,

Gerald Schuller, TU Ilmenau

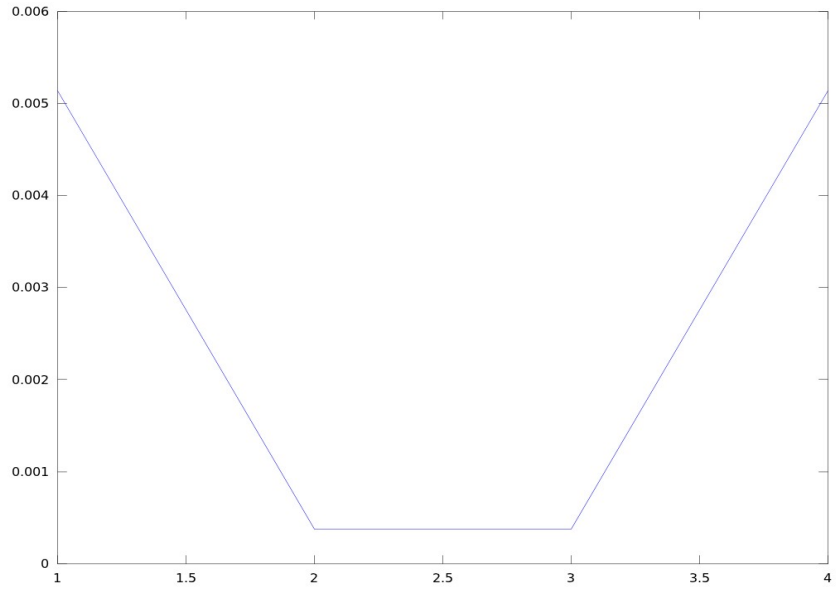
Frequency Warping, Example

Example: Design a **warped low pass filter** with cutoff frequency of 0.05*pi (pi is the Nyquist frequency). Observe: here this frequency is the end of passband, with frequency warping close to the Bark scale of human hearing.

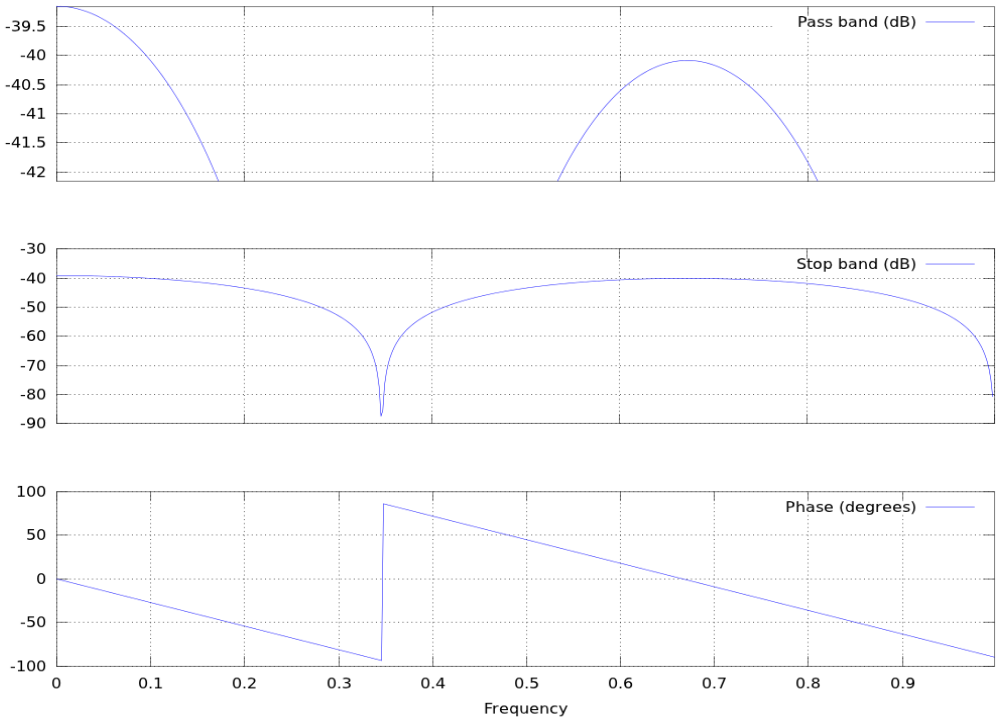
First as a comparison: design an **unwarped filter** with 4 coefficients/taps with these specifications:

In Matlab/Octave:

```
cunw=remez(3,[0 0.05, 0.05+0.05 1],[1 1 0 0],[1 100])
%cunw =
%   5.1365e-03
%   3.7423e-04
%   3.7423e-04
%   5.1365e-03

%impulse response:
plot(cunw)
```



%frequency response:
freqz(cunw,1);



Here we can see that this is not a good filter. The passband is too wide (up to about 0.15), and there is almost no stopband attenuation (in

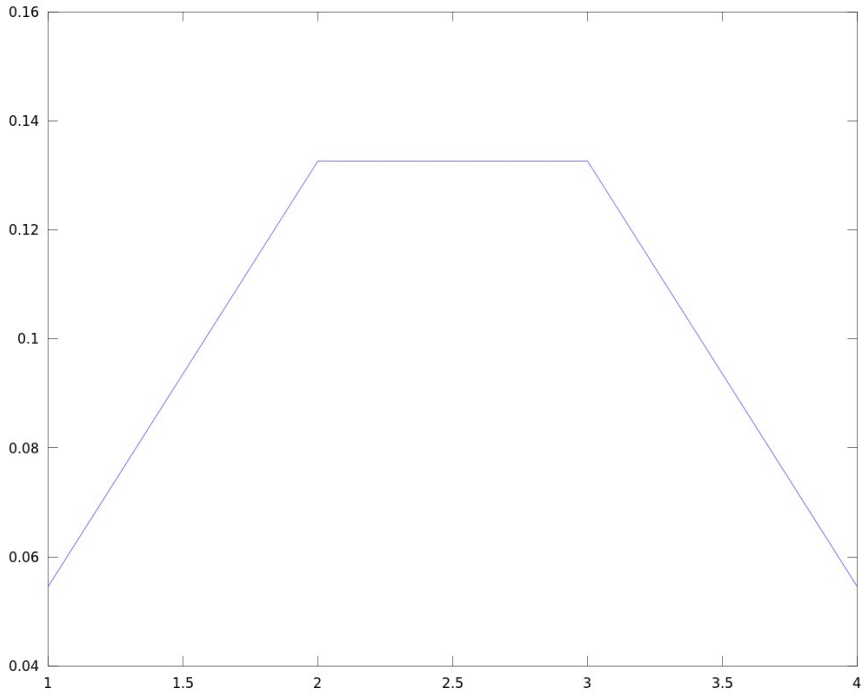
the range of 0.5 to 0.9). So this filter is probably **useless** for our application.

Now design the FIR low pass filter (4th order), which we then want to **frequency warp** in the next step, with a warped cutoff frequency. First we have to compute the allpass coefficient „a“ for our allpass filter which results in an approximate Bark warping, according to [1], eq. (26):

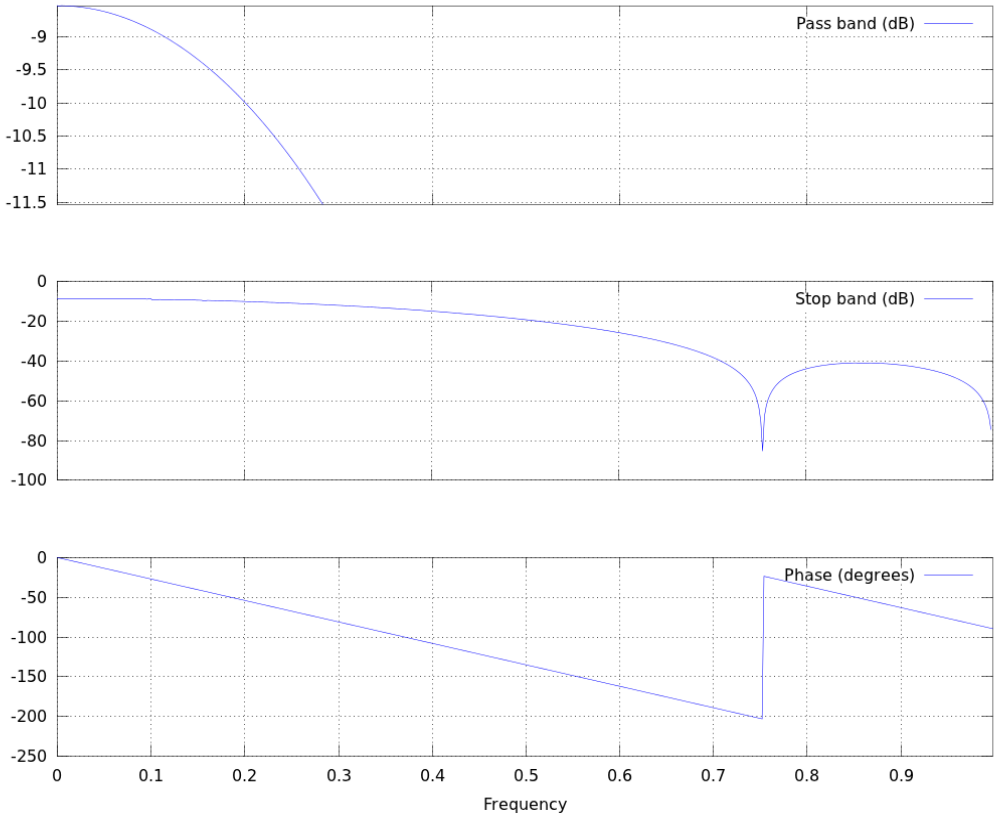
$$a = 1.0674 \cdot \left(\frac{2}{\pi} \cdot \arctan\left(0.6583 * f_s\right) \right)^{0.5} - 0.1916$$

with f_s the sampling frequency in kHz. Our warped design is then

```
%warping allpass coefficient:
a=1.0674*(2/pi*atan(0.6583*32))^0.5 -0.1916
%ans =  0.85956
%with f_s=32 in kHz. from [1]
%The warped cutoff frequency then is:
fcw=-warpingphase(0.05*pi,0.85956)
%fcw = 1.6120; %in radiants
%filter design:
%cutoff frequency normalized to nyquist:
fcny=fcw/pi
%fcny =  0.51312
c=remez(3,[0 fcny, fcny+0.2 1],[1 1 0 0],[1 100]);
%The resulting Impulse Response:
plot(c);
```

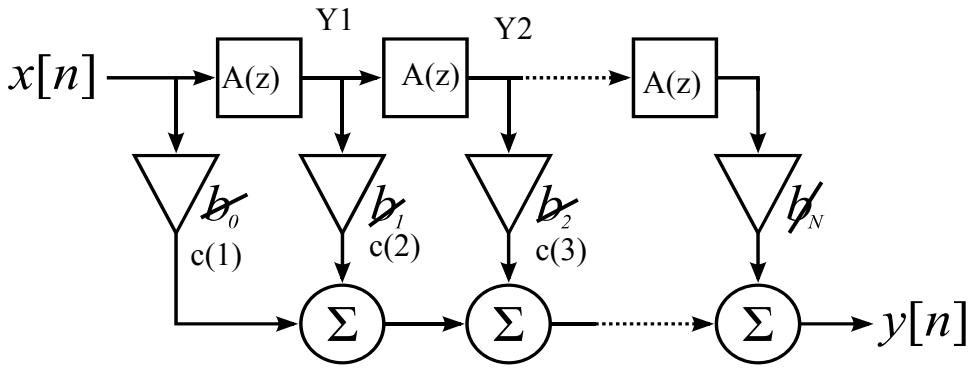


%The resulting Frequency response:
freqz(c,1);

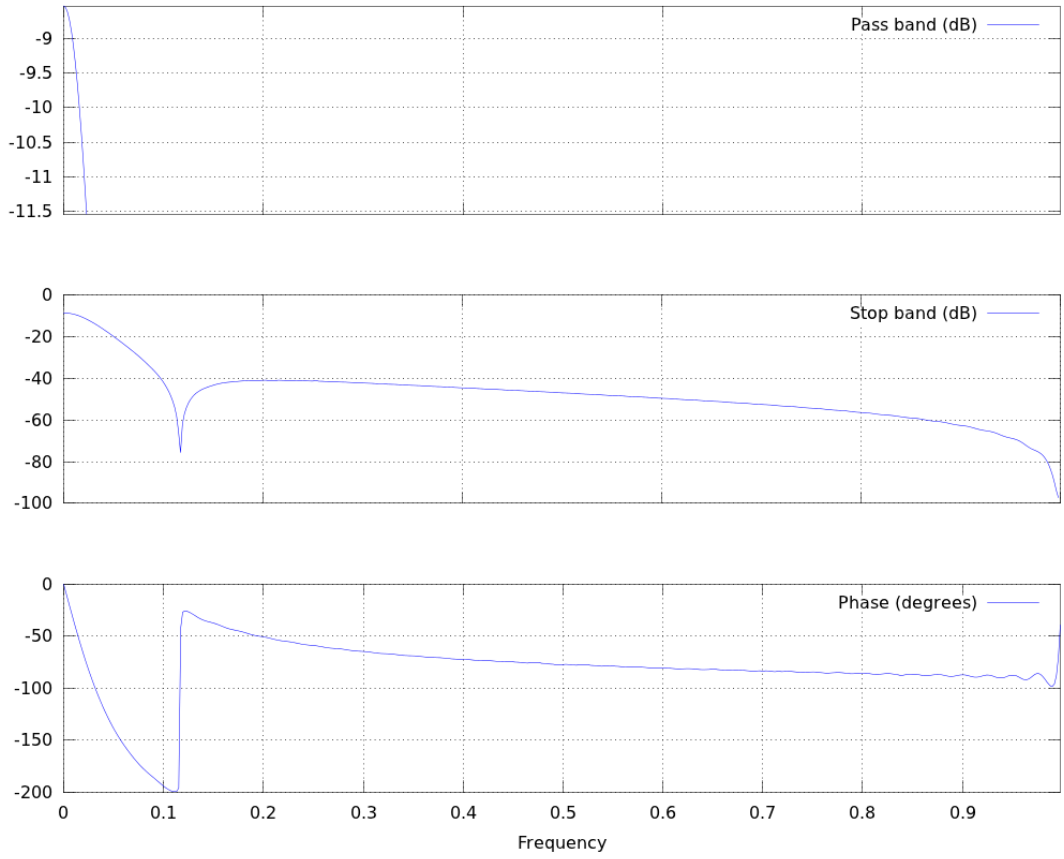


This is the filter we obtain from the c coefficients if we don't replace the delays by allpasses. Here we can see that in the warped domain, we obtain a reasonable low pass filter. In the passband from 0 to somewhat above 0.5 it has a drop of about 10 dB, and in the stopband we obtain about -30 dB attenuation, which is much more than before (it might still not be enough for practical purposes though)

Now we use the same c coefficients, but replace the Delays in the FIR filter with Allpass filter (in this way we go from frequency response H(z) to H(A(z)):

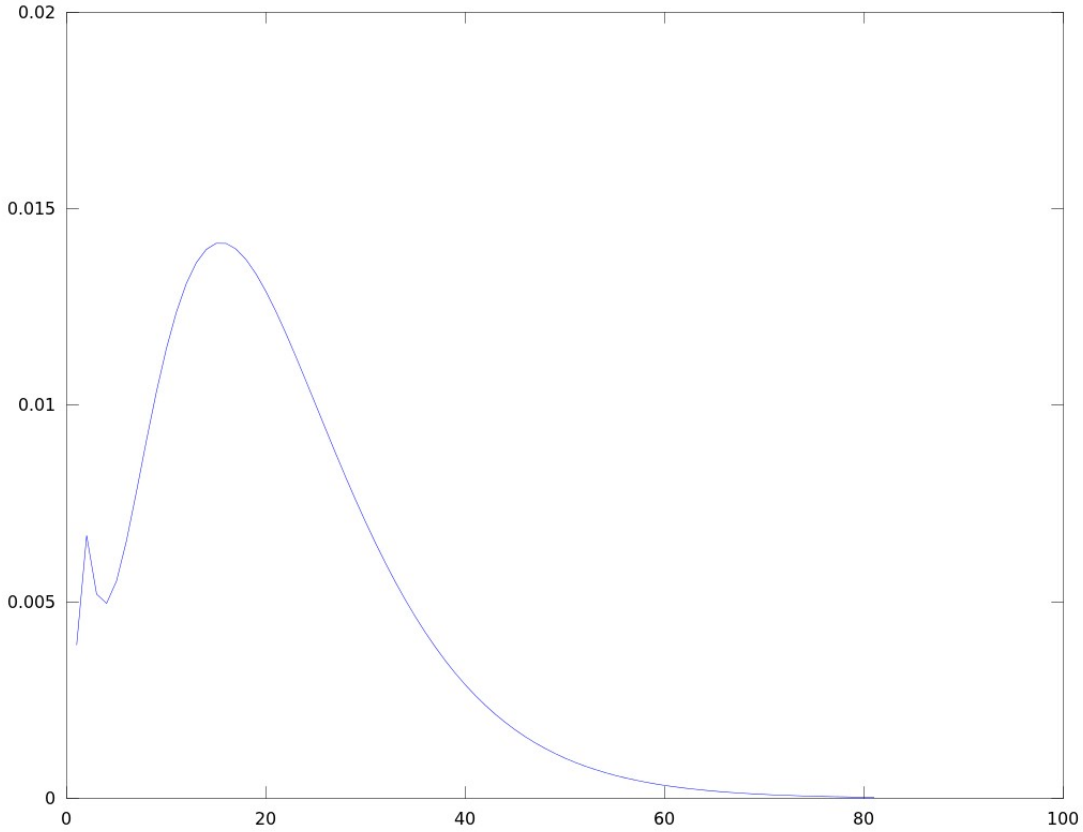



```
%Warping Allpass filters:
B=[-a' 1];
A=[1 -a];
%Impulse with 80 zeros:
Imp=[1,zeros(1,80)];
x=Imp;
%Y1(z)=A(z), Y2(z)=A^2(z),...
%Warped delays:
y1=filter(B,A,x);
y2=filter(B,A,y1);
y3=filter(B,A,y2);
%Output of warped filter with impulse as input:
yout=c(1)*x+c(2)*y1+c(3)*y2+c(4)*y3;
%frequency response:
freqz(yout,1);
```



Here we can now see the frequency response of our final warped low pass filter. We can see that again we have a drop of about 10 dB in the passband, now from 0 to 0.05pi, and a stopband attenuation of about 30dB, which is somewhat reasonable.

```
%Impulse response:
figure;
plot(yout);
```



This is the resulting impulse response of our warped filter. What is most obvious is its length. Instead of just 4 samples, as our original unwarped design, it easily reaches 80 significant samples, and in principle is infinite in extend. This is also what makes it a much better filter than the unwarped original design!

References:

[1] Julius O. Smith and Jonathan S. Abel, “Bark and ERB Bilinear Transforms,” IEEE Transactions on Speech and Audio Processing, vol. 7, no. 6, pp. 697 – 708, November 1999.

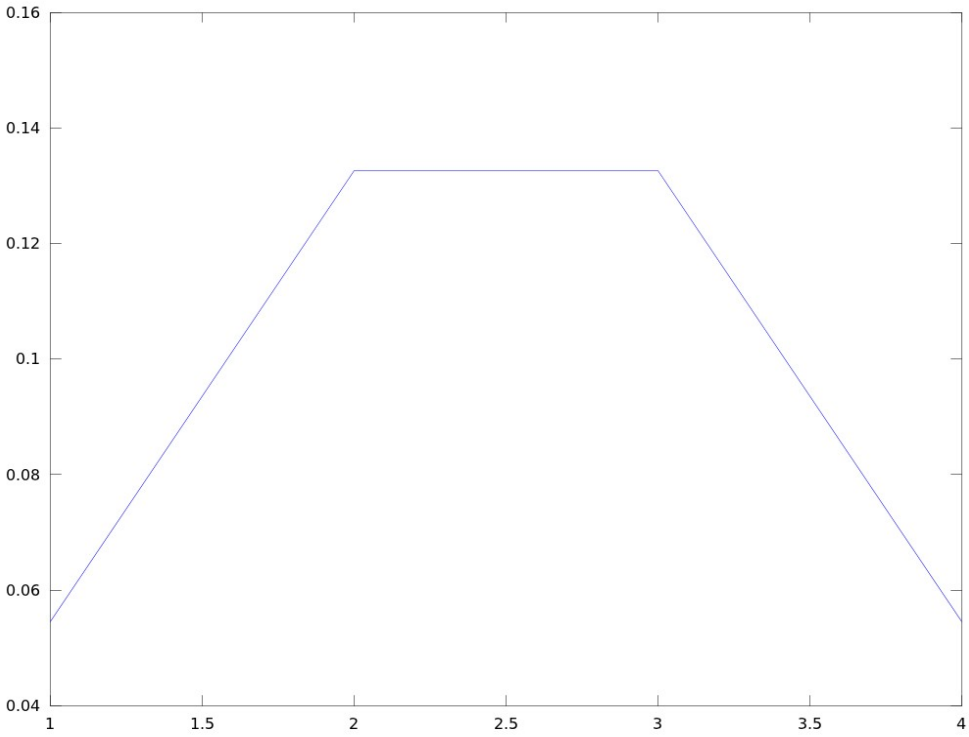
[2] S. Wabnik, G. Schuller, U. Kraemer, J. Hirschfeld: "Frequency Warping in Low Delay Audio Coding", IEEE International Conference on Acoustics, Speech, and Signal Processing, Philadelphia, PA, March 18–23, 2005

Minimum Phase Filters

Remember linear phase filters. Its phase function is linear:

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

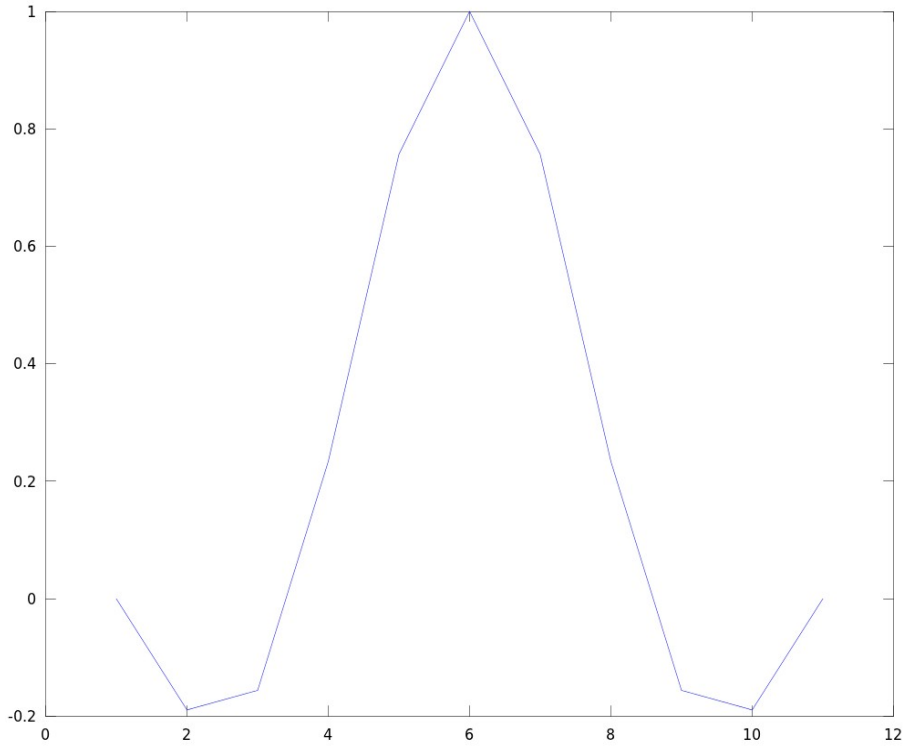
with a group delay of constant d. The impulse responses of linear phase filters have the property of being (even) symmetric around some center. Example:



Here we have a 4 sample impulse response, and starting at 0, we have a symmetry around d=1.5, hence we have a constant delay of this system of d=1.5 samples.

Another example for a linear phase filter is a piece of a sinc function. In Matlab/Octave:

```
hsinc=sinc(-2:0.4:2)
%hsinc =
%Columns 1 through 6:
%-3.8980e-17 -1.8921e-01 -1.5591e-01 2.3387e-01
%7.5683e-01 1.0000e+00
%Columns 7 through 11:
%7.5683e-01 2.3387e-01 -1.5591e-01 -1.8921e-01
%-3.8980e-17
plot(hsinc)
```



This FIR filter has a constant delay factor of d=5 (starting to count the samples at 0 instead of 1 in the plot).

The delay factor d is the center of the impulse response, because we can factor it out from the DTFT of the symmetric impulse response:

$$H\left(e^{j\Omega}\right)=\sum_{n=0}^{2\mathrm{d}} h(n)\cdot e^{-j\Omega n}$$

We factor out the center exponential,

$$H\left(e^{j\Omega}\right)=e^{-j\Omega d}\cdot\sum_{n=0}^{2\mathrm{d}} h(n)\cdot e^{-j\Omega(n-d)}$$

since h(d-n)=h(d+n) we get:

$$H\left(e^{j\Omega}\right)=e^{-j\Omega d}\cdot\sum_{n=0}^{d}h(n)\cdot\left(e^{-j\Omega(d-n)}+e^{j\Omega(d-n)}\right)$$

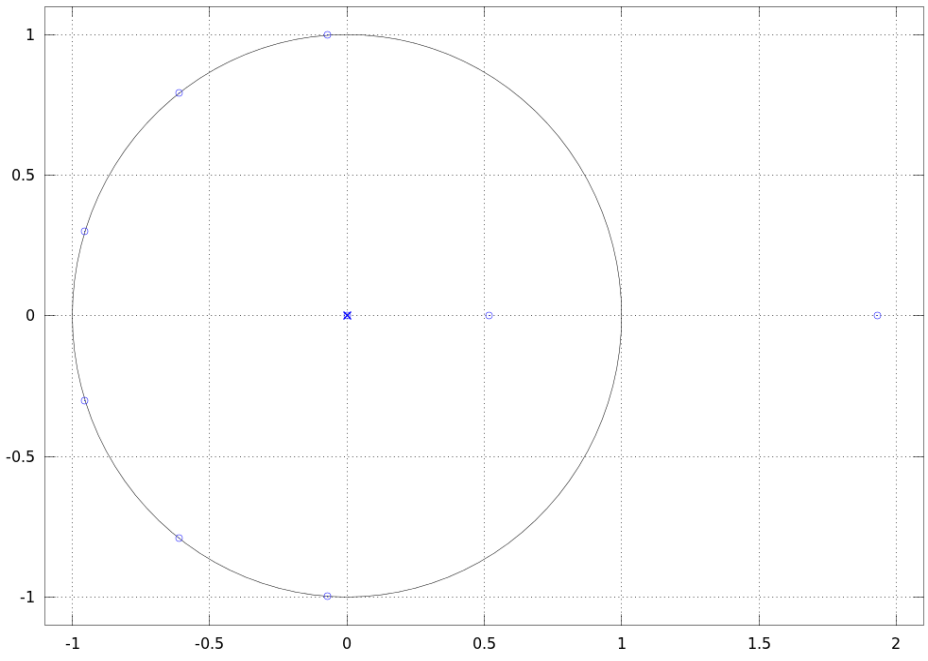
$$H\left(e^{j\Omega}\right)=e^{-j\Omega d}\cdot\sum_{n=0}^{d}h(n)\cdot 2\cdot\cos\left(\Omega(d-n)\right)$$

Hence the phase is:

$$angle\left(H\left(e^{j\Omega}\right)\right)=\varphi(\Omega)=-d\,\Omega$$

Now we can plot its zeros in the zplane, using the command “zplane”:

```
zplane(hsinc,1)
axis([-1.1 2.1 -1.1 1.1],'equal')
```



Observe the zeros near 1.9 and near 0.5, and on the unit circle.
Its zeros are computed with the command

“roots”, and their magnitude with “abs”:

```
abs(roots(hsinc))
```

```
ans =
4.8539e+15
1.9309e+00
1.0000e+00
1.0000e+00
1.0000e+00
1.0000e+00
1.0000e+00
1.0000e+00
5.1789e-01
2.0601e-16
```

Here we can see that we have one zero at location 0, and one at infinity, 6 zeros are on the unit circle, one 1 at distance 1.9309 from the origin, and one is at distance `5.1789e-01=1/1.9309.`
Hence for those 2 zeros we have one zero inside the unit circle at distance r, and one outside the unit circle at distance 1/r.

Linear phase systems and filters have the property, that their **zeros are inside and outside the unit circle** in the z-domain. For stabilty, only poles need to be inside the unit circle, not the zeros. But if we want to invert such a filter (for instance for equalization purposes), the zeros turn into poles, and the **zeros outside the unit** circle turn into **poles outside the unit circle**, making the **inverse**

filter unstable!

To avoid the instability of the inverse filter, we define **minimum phase filters** such that their **inverse is also stable**!
This means, all their **zeros need to be inside the unit circle** in the z-domain.

We can write all linear filters as a concatenation of a minimum phase filter with an allpass filter,

$$H(z) = H_{min}(z) \cdot H_{ap}(z)$$

This can be seen from a (hypothetical) minimum phase system $H_{min}(z)$, which has all its zeros inside the unit circle. Now we concatenate/multiply it with an allpass filter, such that its poles coincide with some of the zeros inside the unit circle. These poles and zeros then cancel, and what is left is the zeros outside the unit circle at a reverse conjugate position $1/a'$, if “a” was the position of the original zero. In this way, we can **„mirror out“ zeros from inside the unit circle to the outside.** The **magnitude response does not change**, because we used an allpass for mirroring out the zeros. As a result we have a system with the same magnitude response, but now with zeros outside the unit circle.

Assume we would like to equalize or

compensate a given transfer function, for instance from a recording. As we saw above, this transfer function can be written as the product

$$H(z)=H_{min}(z)\cdot H_{ap}(z)$$

Only $H_{min}(z)$ has a stable inverse. Hence we design our compensation filter as

$$H_c(z)=\frac{1}{H_{min}(z)}$$

If we apply this compensation filter after our given transfer function, for instance from a recording, we obtain the overall system function as

$$G(z)=H(z)\cdot H_c(z)=H_{ap}(z)$$

This means the overall transfer function now is an allpass, with a constant magnitude response and only phase changes.
(see also A. Oppenheim, R. Schafer: “Discrete Time Signal Processing”, Prentice Hall)

How can we **obtain a minimum phase version** from a given filter? We basically “mirror in” the zeros outside the unit circle.
Take our above example of the piece of the sinc function filter.
In Matlab/Octave we compute the zeros with

```
rt=roots(hsinc)
rt =
-4.8539e+15 + 0.0000e+00i
```

```
1.9309e+00 + 0.0000e+00i
-9.5370e-01 + 3.0077e-01i
-9.5370e-01 - 3.0077e-01i
-6.1157e-01 + 7.9119e-01i
-6.1157e-01 - 7.9119e-01i
-7.1160e-02 + 9.9746e-01i
-7.1160e-02 - 9.9746e-01i
5.1789e-01 + 0.0000e+00i
-2.0601e-16 + 0.0000e+00i
```

We see the zero at 1.93 which we need to mirror in (we neglect the zero at infinity, which comes from starting with a zero sample). To achieve this, we first take the z-domain polynomial of the impulse response, and cancel that zero by dividing through the polynomial with only that zero, $1-1.93\cdot z^{-1}$. Fortunately we have the function “deconv”, which is identical to polynomial division, to do this:

```
[b, r] = deconv (hsinc, [1,-rt(2)])
b =
Columns 1 through 6:
-3.8980e-17 -1.8921e-01 -5.2126e-01 -7.7264e-01
-7.3509e-01 -4.1941e-01
Columns 7 through 10:
-5.3021e-02 1.3149e-01 9.7987e-02 -8.9291e-09
r =
Columns 1 through 6:
0.0000e+00 0.0000e+00 0.0000e+00 0.0000e+00
1.1102e-16 0.0000e+00
Columns 7 through 11:
0.0000e+00 0.0000e+00 0.0000e+00 0.0000e+00
-1.7241e-08
```

Here, r is the remainder. In our case it is

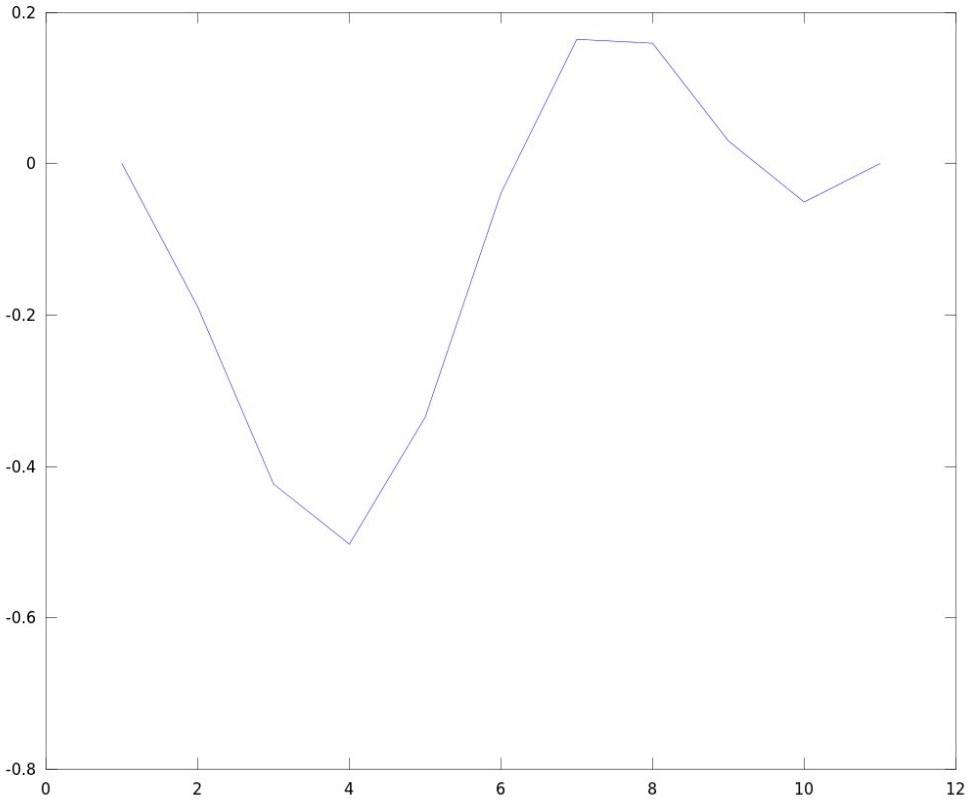
practically zero, which means we can indeed divide our polynomial without any remainder. After that we can multiply the obtained polynomial b with the zero inside the unit circle, at position 1/1.93, by multiplying it with the polynomial with only that zero: $1-1/1.93\cdot z^{-1}$:

```
hsincmp=conv(b,[1,-1/rt(2)'])
```

```
hsincmp =
Columns 1 through 6:
-3.8980e-17 -1.8921e-01 -4.2327e-01 -5.0269e-01
-3.3495e-01 -3.8715e-02
Columns 7 through 11:
1.6418e-01 1.5895e-01 2.9889e-02 -5.0746e-02
4.6242e-09
```

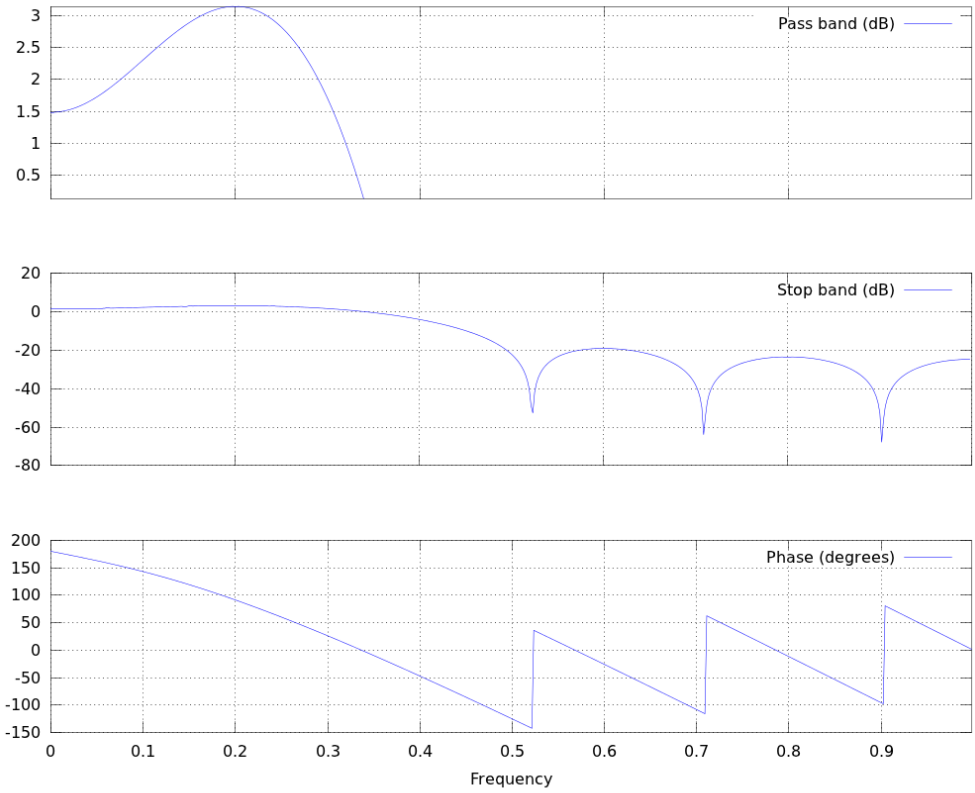
This hsincmp is now our **minimum phase version** of our filter!
Now we can take a look at the impulse response:

```
plot(hsincmp)
```



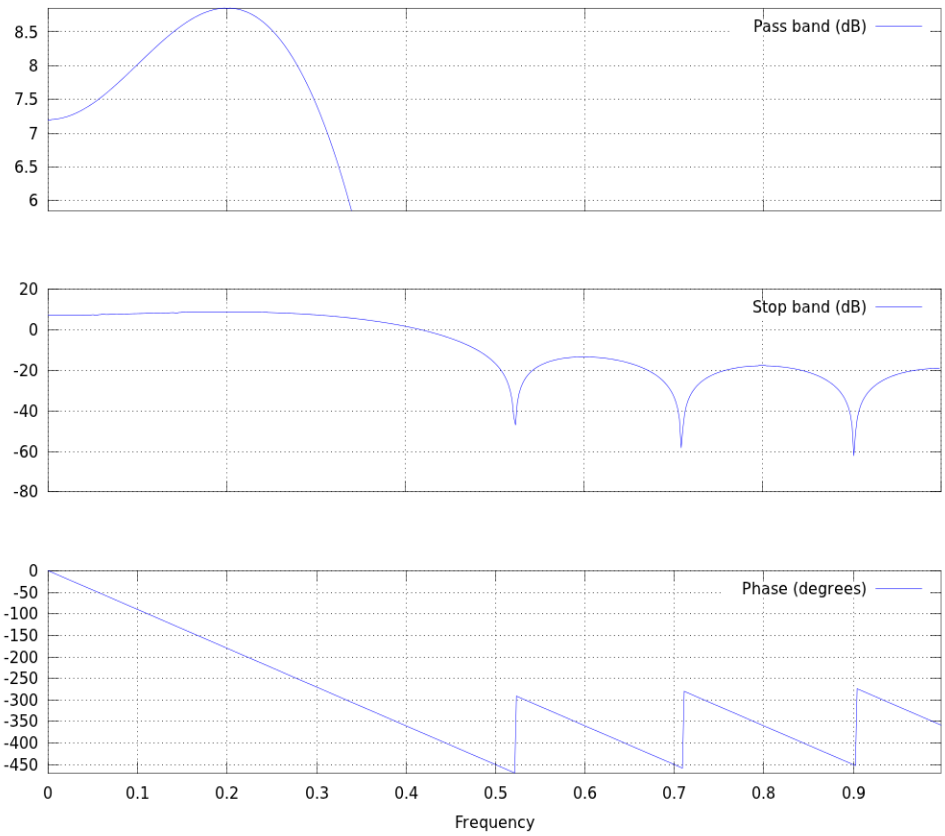
Observe that our filter now became **non-symmetric**, with the main peak at the beginning of the impulse response!
The resulting frequency response is obtained with

`freqz(hsincmp)`



Now compare the above frequency response of our minimum phase filter with the linear phase version, with

```
figure
freqz(hsinc)
```



Here we can see that the magnitude of the frequency plot is indeed identical between the linear phase and the minimum phase version (except for an offset of about 5 dB, which is not important because it is a constant gain factor). But looking at the phase, we see that the minimum phase version has less phase lag. Looking at frequency 0.5, we see that the linear phase filter has a phase lag of about 450 degrees, whereas, the minimum phase filter has a **reduced phase lag** of about 300 degrees (from frequency zero to 0.5)!
If we take the derivative of the phase function to obtain the group delay, we will get correspondingly lower values, which means the minimum phase filter will have **less group**

delay than the linear phase filter. In fact, it has the **lowest possible delay for the given magnitude response** of the filter. So if you have a given magnitude filter design, and want to obtain the **lowest possible delay**, you have to take **minimum phase filters**.

Digital Signal Processing 2/ Advanced Digital Signal Processing
Lecture 11, Complex Signals and Filters, Hilbert Transform

Gerald Schuller, TU Ilmenau

Imagine we would like to know the precise, **instantaneous, amplitude of a sinusoid**. Just looking at the function this might not be so easy, we would have to determine the maximum or minimum, and depending on the sinusoidal frequency this might take some time, during which the amplitude also might have changed. But if we have a **complex exponential** with samples

$$x(n) = A \cdot e^{j\Omega n} = A \cdot \cos(\Omega n) + j\, A \cdot \sin(\Omega n)$$

(we know x(n) but not A)
we can easily determine the amplitude A only knowing one sample x(n) by taking the magnitude of this complex exponential,

$$A = \sqrt{\Re(x(n))^2 + \Im(x(n))^2} =$$
$$= \sqrt{((A \cdot \cos(\Omega n))^2 + (A \cdot \sin(\Omega n))^2)}$$

Observe that this computation of A is independent of the time index n, so it can be done **at every time instance**.
So if we not only have the sine or cosine function in itself, but both, we can easily compute the instantaneous amplitude in this

way. This means, instead of just having the sinusoidal function, we also have the **90 degrees phase shifted** version of it, to compute the amplitude. The problem is, if we only have one part (e.g. the real part) how do we get this 90 degrees phase shifted version? For example, look at the sine function. It already consists of 2 complex exponentials,

$$\sin(\Omega n)=\frac{1}{2\mathrm{j}}\left(e^{j\Omega n}-e^{-j\Omega n}\right)$$

we just have one exponential too many. If we look at it in the Fourier Domain, we see that one exponential is at positive frequencies, and the other is at negative frequencies. If we could remove one of the 2 exponentials, for instance at the negative frequencies, we would have reached our goal of obtaining a complex exponential for amplitude computations. So what we need is **a filter**, which **attenuates the negative frequencies**, and leaves the **positive frequencies unchanged**!
So how do we obtain such a filter? First we formulate our requirement in the **frequency domain** (the DTFT domain):

$$H(\Omega)=\begin{Bmatrix}1\ for\ \Omega>0\\0\ for\ \Omega<0\end{Bmatrix}$$

We could multiply our signal spectrum with this frequency domain formulation (which is often

not practical), or in the time domain convolve with the **impulse response** of the resulting filter obtained with **inverse DTFT**,

$$h(n)=\frac{1}{2\pi}\int_0^{\pi} 1\cdot e^{j\Omega n}\,d\Omega=$$

for $n\neq 0$ this becomes

$$=\frac{1}{2\pi}\left(\frac{1}{jn}\cdot e^{j\pi n}-\frac{1}{jn}\right)=$$

$$=\frac{1}{2\pi}\left(\frac{-j}{n}\cdot(-1)^n+\frac{j}{n}\right)=$$

$$=\begin{Bmatrix}\frac{j}{\pi n}\ for\ n\ odd\\ 0\ for\ n\ even\end{Bmatrix}$$

For n=0 this inverse DTFT integral becomes

$$h(0)=\frac{1}{2\pi}\cdot\pi=\frac{1}{2}$$

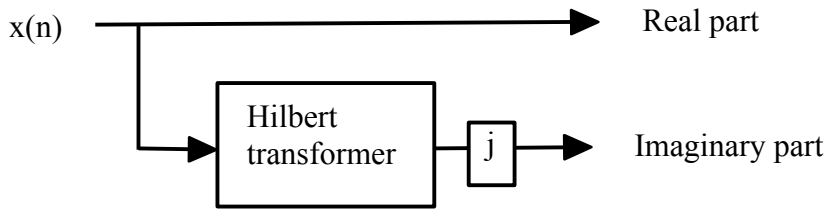
Hence the resulting impulse response of this one-sided filter becomes

$$h(n)=\frac{1}{2}\cdot\delta(n)+\begin{Bmatrix}\frac{j}{\pi n}\ for\ n\ odd\\ 0\ for\ n\ even\end{Bmatrix}$$

This is now the resulting **impulse response** (time domain) of our **filter which passes all the positive frequencies and attenuates the negative frequencies.**

Here we can see that the first part with the **delta function** represents the **real part**, which is the signal itself (signal convolved with

the delta impulse), except for a factor of 2. The second part represents the imaginary part of our one-sided signal (pos. frequencies). Multiplying both parts with this factor of 2 for simplicity, we obtain the following structure,



where the **Hilbert transformer** $h_H(n)$ is

$$h_H(n)=\begin{cases}\frac{2}{\pi n} \; for\, n\, odd \\ 0 \; for\, n\, even\end{cases}$$

(See also: Oppenheim, Schafer, “Discrete-Time Signal Processing”). The x(n) is our real valued signal. Observe that only this part, which creates the imaginary signal part, is the Hilbert Transformer. We can use it to construct a filter that suppresses the negative frequencies.

This means, we take our **original signal** (the sinusoid) and **define it as our real part**. Then we take the **Hilbert filtered signal**, filtered with the above $h_H(n)$, and define it as our **imaginary part**. Then both together have a **one-sided, positive only spectrum**! That also means, our Hilbert transform filter is our

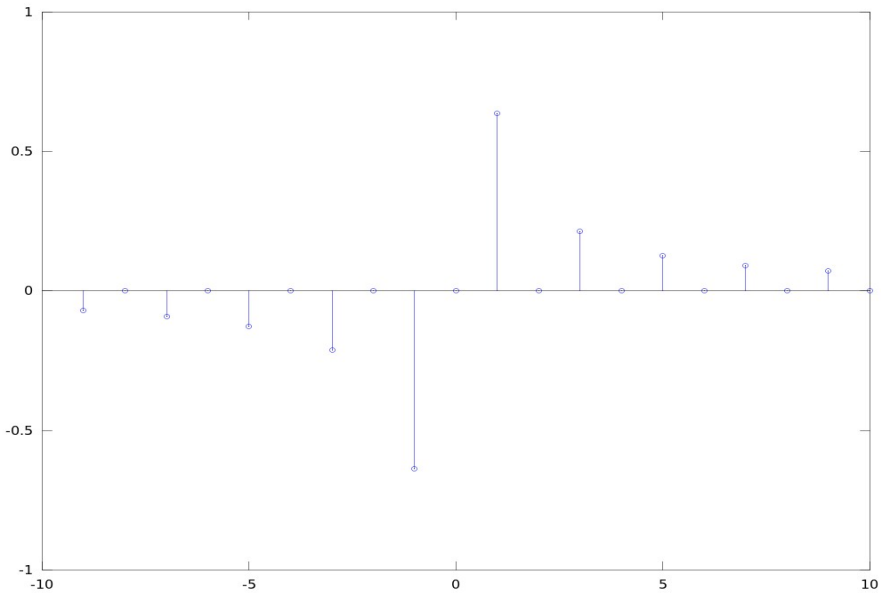
90 degrees phase shifter that we where looking for!

You can also imagine, if the real part is a $\cos$ signal, then the Hilbert Transformer generates the $j \cdot \sin$ part.

Matlab/Octave example: Plot the Hilbert transformer for n=-10 ..10:

```
h=zeros(1,20);
n=-9:2:10;
h(n+10)=2./(pi*n);
stem(-9:10,h)
```

Here we see that we have negative indices. If



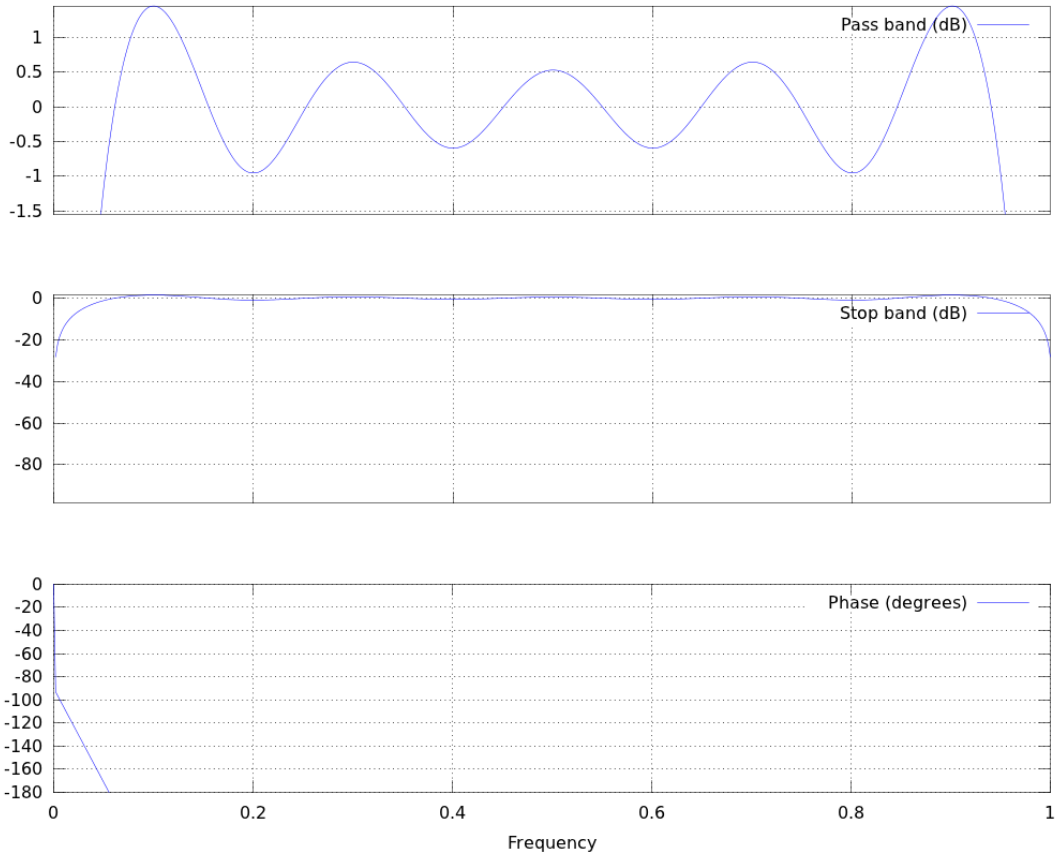
we want to obtain a causal system, we need to shift them to at or above zero using a suitable delay, hence the Hilbert transform involves some delay. Hence to make the imaginary and

real part to fit to each other, we have to **delay the real part accordingly**.

This type of complex signal, with a 1-sided spectrum, is also called an “**analytic signal**”.

Let's take a look at the frequency response of our filter. Since we have included a delay, our phase should be our 90 degrees phase shift plus the linear phase from the delay. Hence our phase curve should hit the phase axis at frequency 0 at 90 degrees, for which we would like to zoom in to this part.

```
freqz(h);
axis([0 1 -180 0])
```



Here we can see that the phase curve indeed would hit the 90 degree mark at the phase axis, except that it goes to 0 before it. This is because a finite length Hilbert transformer does not work at frequency zero. This is also what we see at the magnitude plot. It only reaches about 0 dB attenuation at about frequency 0.08 and reaches higher attenuations at frequencies below about 0.05 and above about 0.95. Hence it is only a working Hilbert transformer within this

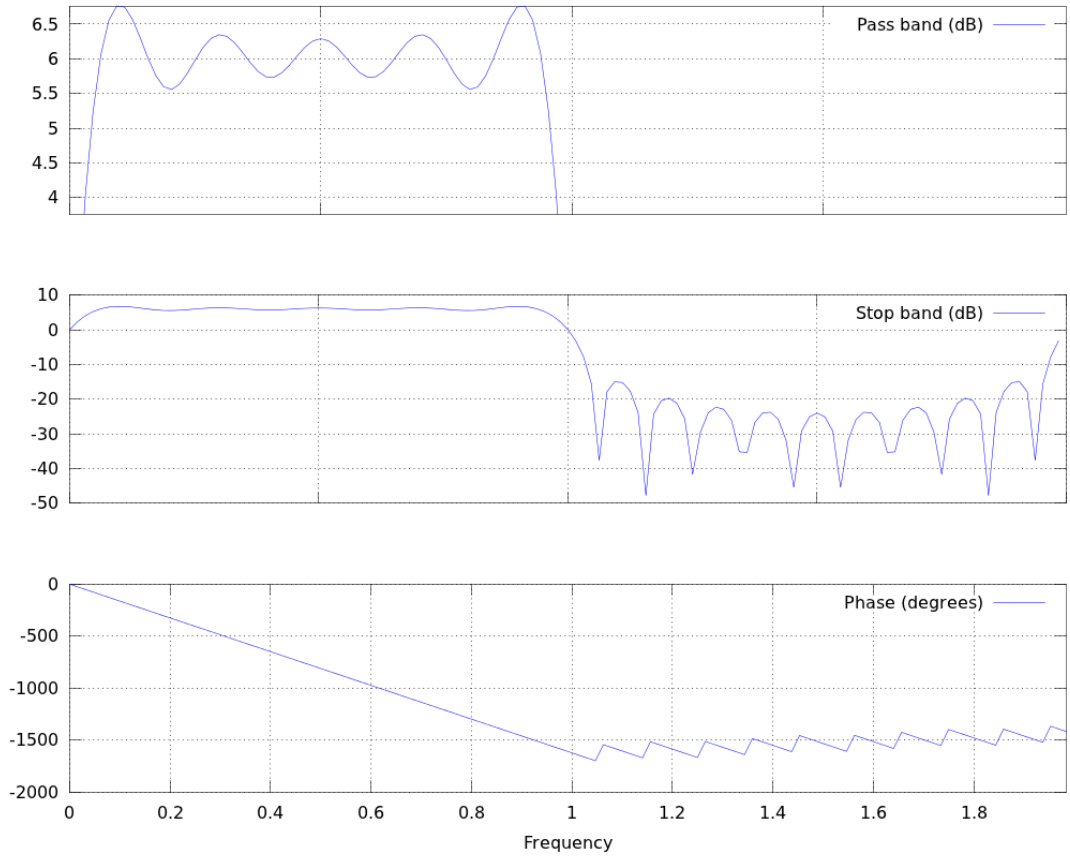
frequency range.

If we want to plot the frequency response of our **entire filter** (not just the Hilbert transformer part), which passes only the positive frequencies, we first need to construct our resulting complex filter, and then plot the frequency response on the **whole** frequency circle. First we need to create the correspondingly delayed unit impulse as the real part:

```
%construct a delayed impulse to implement the delay for the real part:
delt=[zeros(1,9),1,zeros(1,10)]
%delt =
%0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0
%0
%Then we need to add our imaginary part as our Hilbert transform to obtain our complex filter:
h=zeros(1,20);
n=-9:2:10;
h(n+10)=2./(pi*n);

hone=delt+j*h
hone =
Columns 1 through 3:
0.00000 - 0.07074i 0.00000 + 0.00000i 0.00000 - 0.09095i
Columns 4 through 6:
0.00000 + 0.00000i 0.00000 - 0.12732i 0.00000 + 0.00000i
Columns 7 through 9:
0.00000 - 0.21221i 0.00000 + 0.00000i 0.00000 - 0.63662i
```

```
Columns 10 through 12:
1.00000 + 0.00000i 0.00000 + 0.63662i 0.00000 +
0.00000i
Columns 13 through 15:
0.00000 + 0.21221i 0.00000 + 0.00000i 0.00000 +
0.12732i
Columns 16 through 18:
0.00000 + 0.00000i 0.00000 + 0.09095i 0.00000 +
0.00000i
Columns 19 and 20:
0.00000 + 0.07074i 0.00000 + 0.00000i
>>> freqz(hone,1,128,'whole');
```

The resulting plot is:

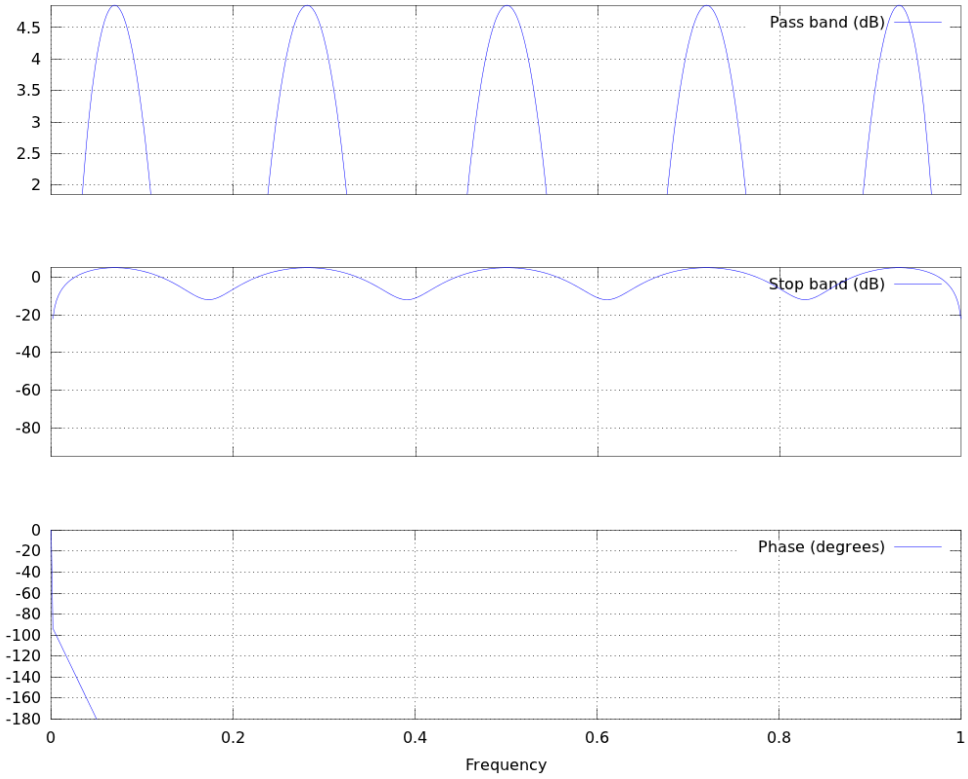


Here we can see that we have indeed a passband at the positive frequencies between 0 and 1. Observe that the passband is at about

6dB above 0dB, because we multiplied our filter by 2 to make it simpler.
The negative frequencies appear between 1 and 2 (or pi and 2pi) on the frequency axis, and we can see that we get about -30 dB attenuation there, which is not too much, but which we could increase it by making the filter longer. This also gives us a good indication of how well our filter is working!

The Octave/Matlab function “remez” also has an option for a Hilbert transform filter.

```
b=remez(20,[0  1],[1 1 ],'hilbert');
freqz(b);
axis([0 1 -180 0]);
```

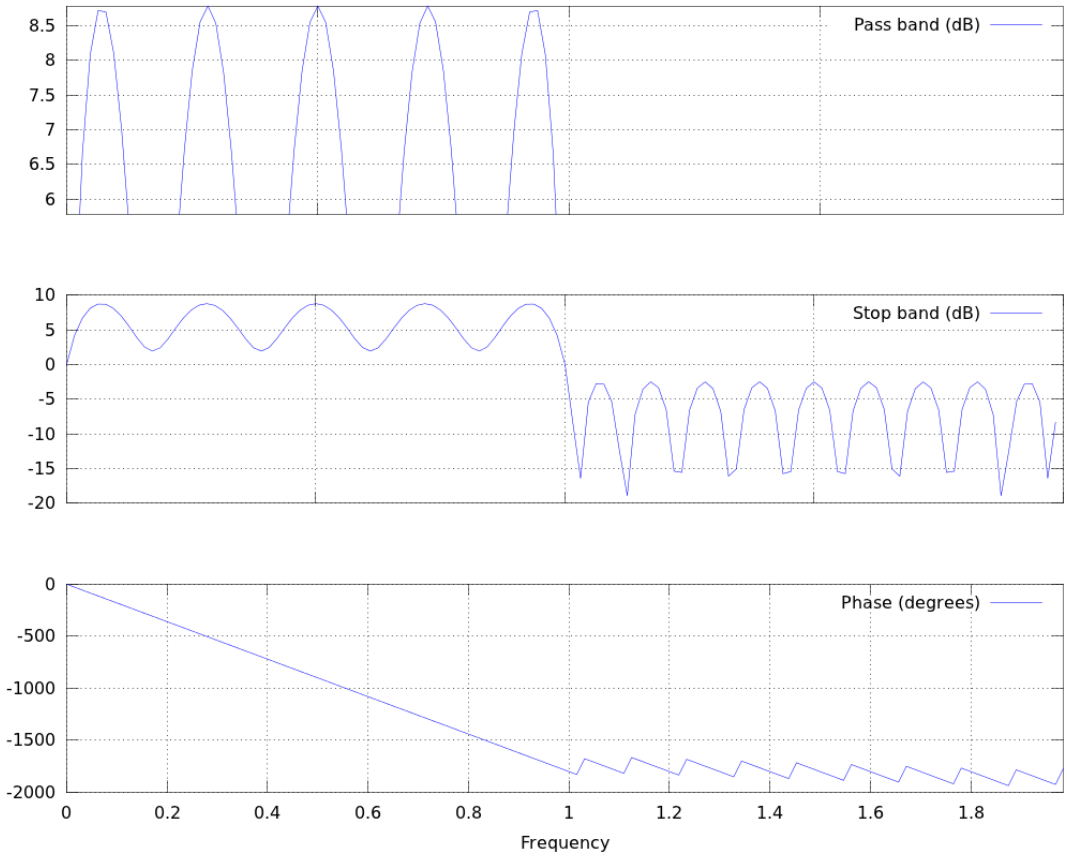


This filter also fullfils the 90 degree phase shift requirement, but seems to have much stronger ripples in the magnitude of the frequency response!
But observe the equi-ripple behaviour in the passband, which is what we expect from remez.

Let's look at the whole frequency circle again, in Octave:

```
%Delay for the real part:
delt=[zeros(1,10),1,zeros(1,10)]
delt =
```

```
Columns 1 through 20:
0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0
0
Column 21:
0
honeremez=delt+j*b'
honeremez =
Columns 1 through 3:
0.00000 - 0.00000i 0.00000 - 0.41954i 0.00000 +
0.00000i
Columns 4 through 6:
0.00000 - 0.09991i 0.00000 - 0.00000i 0.00000 -
0.13309i
Columns 7 through 9:
0.00000 - 0.00000i 0.00000 - 0.21540i 0.00000 -
0.00000i
Columns 10 through 12:
0.00000 - 0.63701i 1.00000 + 0.00000i 0.00000 +
0.63701i
Columns 13 through 15:
0.00000 + 0.00000i 0.00000 + 0.21540i 0.00000 +
0.00000i
Columns 16 through 18:
0.00000 + 0.13309i 0.00000 + 0.00000i 0.00000 +
0.09991i
Columns 19 through 21:
0.00000 - 0.00000i 0.00000 + 0.41954i 0.00000 +
0.00000i
>>> freqz(honeremez,1,128,'whole');
```

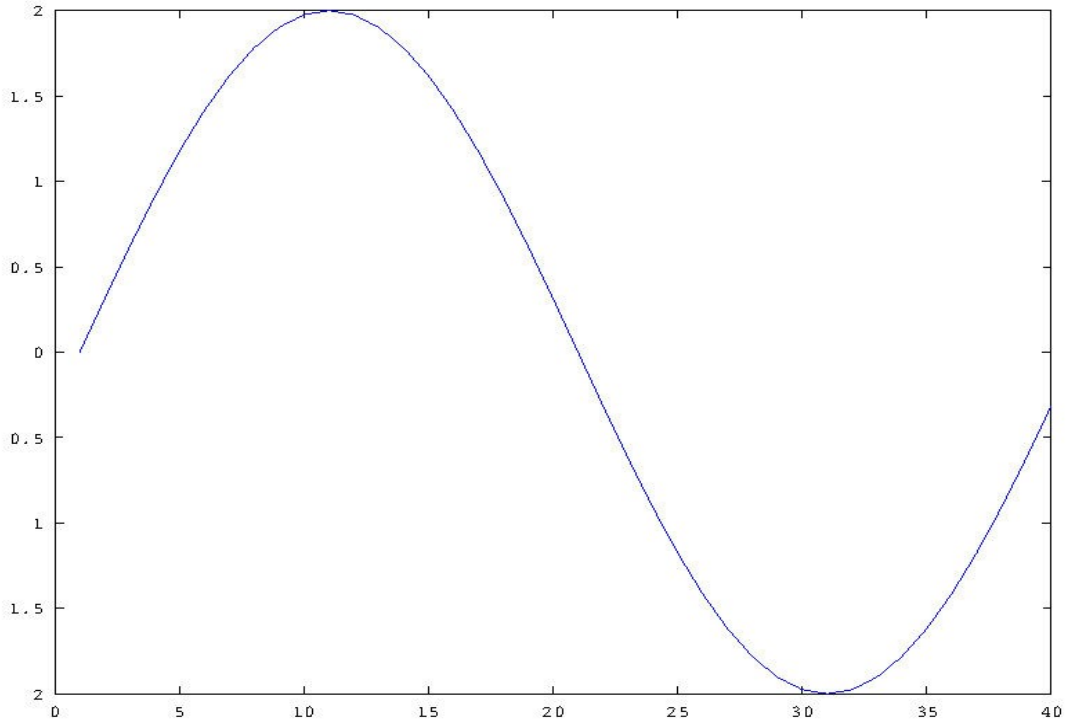


Here we can see that we have **less attenuation** as before (only about -10 dB in the stopband compared to the pass band!), more ripples in the passband, but we also have a **narrower transition band** at the high and low frequency end of the spectrum, compared to the previous filter! Also again observe the equi-ripple behaviour in the pass band and also the stop band. Probably we obtain more practical filters, if we change the corner frequecies to slightly above 0 and slightly below 1.

Example for the Measurement of the Amplitude:

We can now test our application example of measuring the Amplitude of a sinusoid with our Hilbert transform design. We saw that the lower end for the passband of our design is at normalized frequency of about 0.05. Hence we test a sinusoid of that frequency,

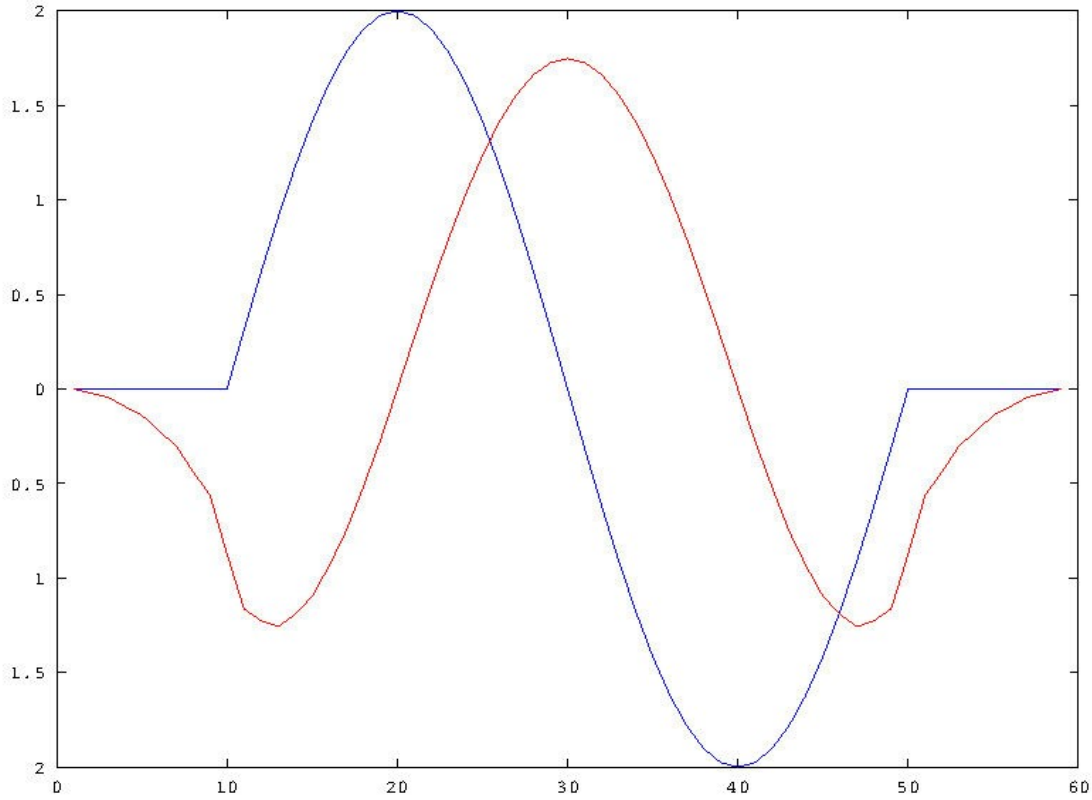
```
x=2*sin(pi*0.05*(0:39));
plot(x)
```



Now we can filter it with our filter which passes only positive frequencies “hone”,

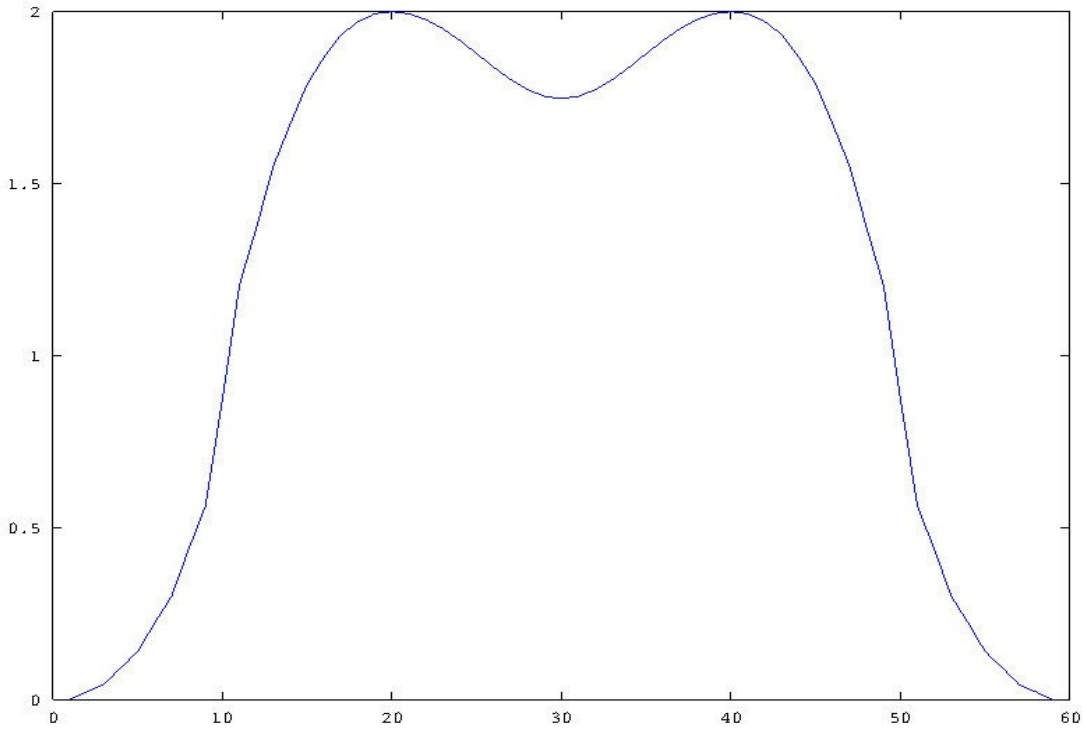
```
xhone=conv(x,hone);
plot(real(xhone))
hold
plot(imag(xhone),'r')
```

Here we can see that we get indeed a 90 degree phase shifted version, the red curve, about between sample 15 and 45.



Now we can compute the magnitude of this complex signal “xhone” to obtain the amplitude of our sinusoidal signal,

```
hold off
plot(abs(xhone))
```



We see that between about sample 15 and 45 we obtain the **amplitude** of our sinusoidal signal with about **10% accuracy**, which roughly corresponds to the -20dB attenuation (corresponding to an attenuation factor of 0.1) that our filter “hone” provides. This also hints at the fact that we can improve the magnitude estimation by having a filter with a **higher attenuation** at negative frequencies.

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

Gerald Schuller, TU Ilmenau

Filters 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.

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

$$x(n) = y(n) * h_w(n)$$

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) can be written as a matrix multiplication:

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

Let's define 2 vectors.

The first is a vector of the the **past L samples of our noisy signal** y, up to the present sample at time n, (bold face font to indicate that it it a vector)

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

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

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

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 column vector,

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

To obtain this column 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

$$\boldsymbol{A} \cdot \boldsymbol{h}_W = \boldsymbol{x}$$

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

$$\begin{bmatrix} y(1) & y(2) \\ y(2) & y(3) \\ y(3) & y(4) \\ \vdots & \vdots \end{bmatrix} \cdot \begin{bmatrix} h_W(2) \\ h_W(1) \end{bmatrix} = \begin{bmatrix} x(1) \\ x(2) \\ x(3) \\ \vdots \end{bmatrix}$$

Observe again that the vector $\boldsymbol{h}_W$ in this equation is the **time reversed** impulse response of our filter. 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 so-called Moore-Penrose **Pseudo Inverse** (http://en.wikipedia.org/wiki/Moore%E2%80%93Penrose_pseudoinverse)
This pseudo-inverse finds the column vector h which minimizes the distance to a given x with the matrix A (which contains our signal y to be filtered):

$$A \cdot h_W \rightarrow x$$

Matrix A and vector x are known, and vector h_W is unknown.
This problem can be solved exactly if the matrix A is **square and invertible**. Just multiplying the equation with A^{-1} from the left would give us the solution

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

This cannot be done, if 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 x . We would like to obtain the solution which comes **closest** to x in a mean squared error distance sense (also called **Euclidean Distance**).

This solution is derived using the Pseudo-Inverse:

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

Here, $A^T \cdot A$ is now a square matrix, and usually invertible, such that we obtain our solution

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

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.

Octave/Matlab Example for denoising speech:

```
x=wavread('fspeech.wav');
sound(x,32000);
%additive zero mean white noise (for
-1<x<1):
y=x+0.1*(rand(size(x))-0.5); %column vector
sound(y,32000);
```

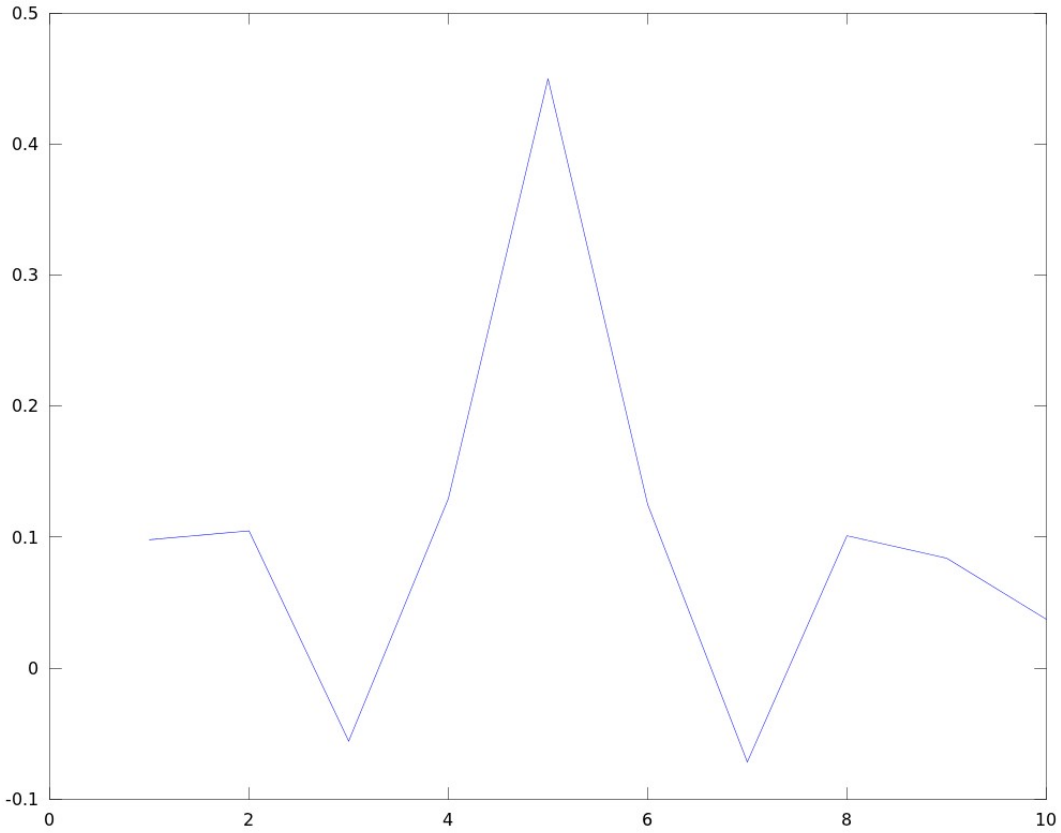
%we assume 10 coefficients for our Wiener filter.
%10 to 12 is a good number for speech signals.

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

%Our matrix has 100000 rows and 10 colums:
size(A)
```

```
%ans =
%100000 10
```

%Compute Wiener Filter:

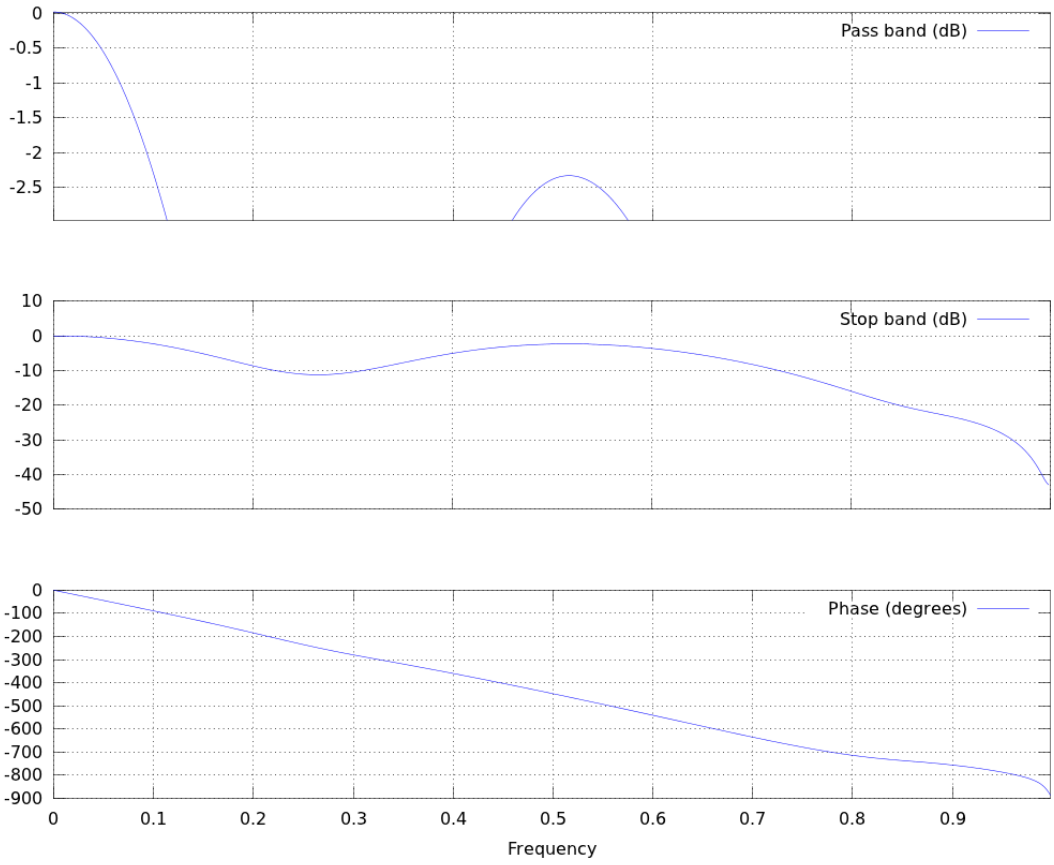
```
%Trick: allow (flipped) filter delay of 5
%samples to get better working denoising.
%The desired signal hence is x(6:100005):
h=inv(A'*A) * A' * x(6:100005)
plot(flipud(h))
```



Observe that for this non-flipped impulse response we see a delay of 4 samples, since the first sample corresponds to a delay of zero, and the location of biggest sample, which corresponds to the signal delay, is at delay 5-

1=4.

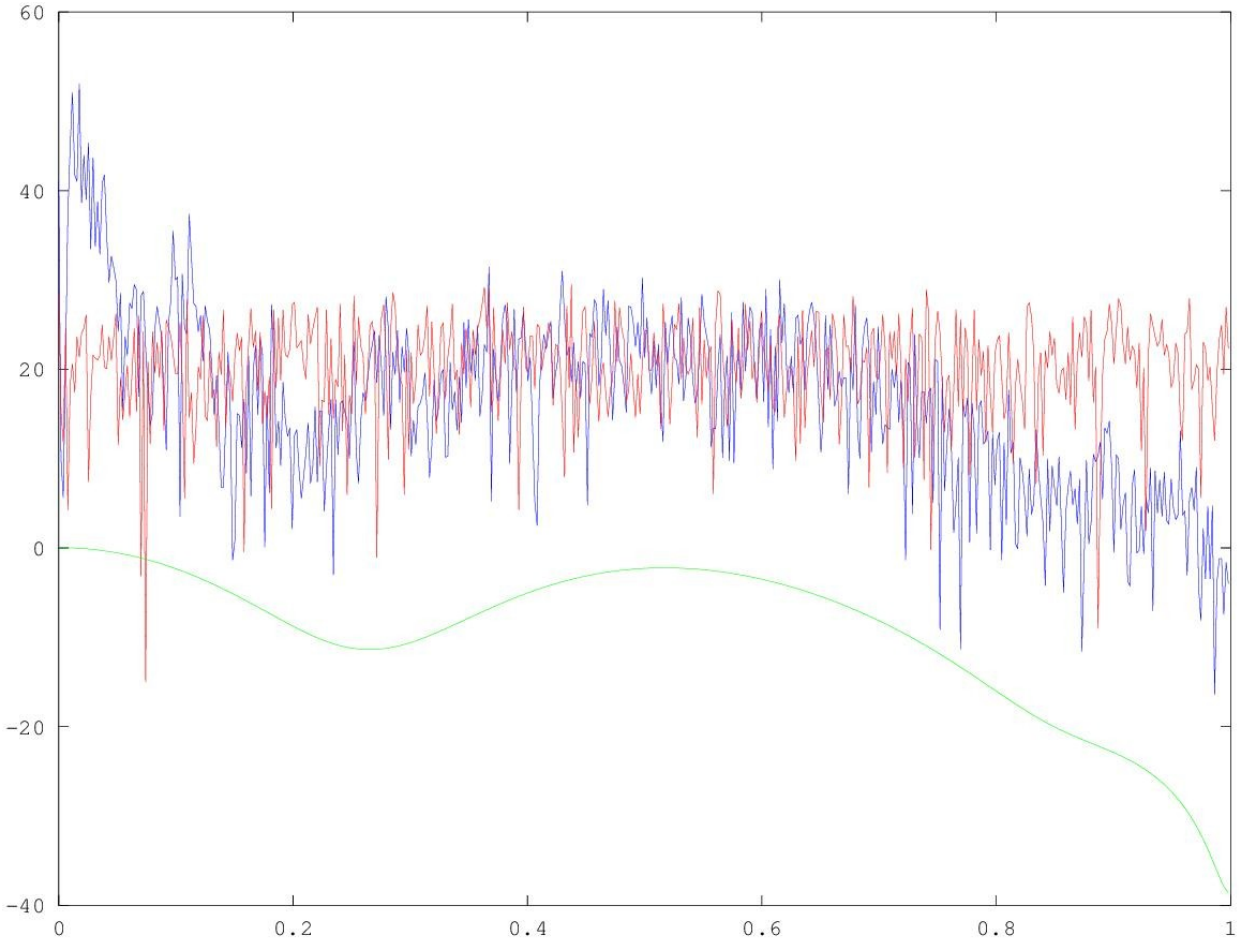
Its frequency response is *freqz(flipud(h))*



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 on can observe with *freqz(x)*. If we look at the spectrum of the white noise with *freqz(0.1*(rand(size(x))-0.5))*, then

we see that at low frequencies speech is domianting, and at high frequencies, noise is dominating, and we need to remove or attenuate that latter part of the spectrum. We can plot the spectra of the speech and the noise together with:

```
Hspeech=freqz(x);
Hnoise=freqz(0.1*(rand(size(x))-0.5));
Hw=freqz(flipud(h));
plot((0:511)/512,20*log10(abs(Hspeech)));
hold
plot((0:511)/512,20*log10(abs(Hnoise)),'r');
plot((0:511)/512,20*log10(abs(Hw)),'g');
```



The speech spectrum is blue, the noise

spectrum is red, and as a comparison the Wiener filter transfer function is green. 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 and listen to it:

```
xw=filter(flipud(h),1,y);
sound(xw,32000)
```

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.

Let's compare the mean quadratic error. For the noisy signal it is

```
size(x)
%ans =
%207612 1
sum((y(1:200000)-x(1:200000)).^ 2)/200000
%ans = 8.3499e-04
```

For the Wiener filtered signal it is (taking into account 4 samples delay from our filter (5 from the end from flipping), beginning to peak).

```
sum((xw(4+(1:200000))-x(1:200000)).^
2)/200000
```

```
ans = 3.4732e-04
```

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'*A
ans =
Columns 1 through 8:
1011.69 859.44 806.14 842.55 837.94 797.05 774.33
745.8
1
859.44 1011.69 859.44 806.14 842.55 837.94 797.05
774.3
3
806.14 859.44 1011.69 859.44 806.14 842.55 837.94
797.0
5
842.55 806.14 859.44 1011.69 859.44 806.14 842.55
837.9
4
837.94 842.55 806.14 859.44 1011.69 859.44 806.14
842.5
5
797.05 837.94 842.55 806.14 859.44 1011.69 859.44
806.1
4
774.33 797.05 837.94 842.55 806.14 859.44 1011.69
859.4
4
745.81 774.33 797.05 837.94 842.55 806.14 859.44
1011.6
9
713.58 745.81 774.33 797.05 837.94 842.55 806.14
859.4
4
693.25 713.58 745.81 774.33 797.05 837.94 842.55
```

```
806.1
4
Columns 9 and 10:
713.58 693.25
745.81 713.58
774.33 745.81
797.05 774.33
837.94 797.05
842.55 837.94
806.14 842.55
859.44 806.14
1011.69 859.44
859.44 1011.69
```

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 (circularly) shifted by 1 sample to the right.

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

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

(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**".

The expression $A^T \cdot x$ in our formulation of the

Wiener filter becomes the cross correlation vector

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

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

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

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. That is sufficient to compute the Wiener filter coefficients, using above formulations.

Digital Signal Processing 2/ Advanced Digital Signal Processing
Lecture 13, Matched Filter, Prediction

Gerald Schuller, TU Ilmenau

Matched Filters

Remember the goal of a matched filter $h_M(n)$:

$$y(n) * h_M(n) \rightarrow x(n+n_d) * h_M(n)$$

with some signal delay n_d (no signal fidelity, just high SNR for detection), where $y(n) = x(n+n_d) + v(n)$ is our (delayed) signal with additive noise $v(n)$.

Application examples are in communications, where you would like to detect a 0 or 1,for instance in CDMA, where each user gets a unique pseudo-random 1/0 sequence (so-called chip-sequences) to represent 0 or 1, in which different users and signals are separated using matched filters. Another example is for detecting **known** signals or patterns, like object or face recognition in images. In general we would like to detect **deterministic signals** $x(n)$.

This means our goal is to **maximize the SNR** at the moment of detection, with our original signal $x(n)$ and noise $v(n)$,

$$SNR = \frac{|x(n) * h_M(n)|^2}{E(|v(n) * h_M(n)|^2)}$$

We would like to maximize this SNR at the time of detection, using $h_M(n)$. To do that, first we assume $v(n)$ to be independent white noise. Then the denominator of the SNR expression is just a scaled fixed power expression. Using our formulation of a matrix V for the noise signal and the vector $\boldsymbol{h_M}$ for our filter (again, it contains the time reversed impulse response), we obtain

$$E(|v(n) * h_M(n)|^2) = E(|\boldsymbol{V} \cdot \boldsymbol{h_M}|^2) = E(\boldsymbol{h_M^T} \cdot \boldsymbol{V^T} \cdot \boldsymbol{V} \cdot \boldsymbol{h_M}) =$$
$$= \boldsymbol{h_M^T} \cdot E(\boldsymbol{V^T} \cdot \boldsymbol{V}) \cdot \boldsymbol{h_M} = \boldsymbol{h_M^T} \cdot \sigma_v^2 \cdot \boldsymbol{I} \cdot \boldsymbol{h_M} =$$
$$= \sigma_v^2 \cdot \boldsymbol{h_M^T} \cdot \boldsymbol{h_M}$$

(Remember: the autocorrelation function of noise is a delta function, since noise samples are uncorrelated to all their neighbour samples, there are only correlated with themselves, and the correlation with itself is simply the noise power σ^2 . The autocorrelation matrix $\boldsymbol{V^T} \cdot \boldsymbol{V}$ hence has all zero entries, except on the diagonal, where it is the noise power, hence noise power time the identity matrix $\sigma^2 \cdot \boldsymbol{I}$)

The last expression is simply the squared norm (the sum of the squares of its coefficients) of our vector of our filter coefficients $\boldsymbol{h_M}$

multiplied with the noise power σ_v^2 .

Keeping the above **norm** of our filter vector **constant**, we only need to **maximize the numerator** of our SNR fraction to maximize the SNR. We apply the Cauchy-Schwartz inequality (see e.g. http://en.wikipedia.org/wiki/Cauchy%E2%80%93Schwarz_inequality), which says, for 2 (column) vectors $\boldsymbol{a}$ and $\boldsymbol{b}$ and their scalar product we get

$$\boldsymbol{a}^T \cdot \boldsymbol{b} \leq \sqrt{\boldsymbol{a}^t \cdot \boldsymbol{a}} \cdot \sqrt{\boldsymbol{b}^t \cdot \boldsymbol{b}}$$

This is also written with the norm $\|\boldsymbol{a}\|$ and $\langle \boldsymbol{a}, \boldsymbol{b} \rangle$ scalar product as

$$|\langle \boldsymbol{a}, \boldsymbol{b} \rangle| \leq \|\boldsymbol{a}\| \cdot \|\boldsymbol{b}\|$$

We obtain equality if both vectors are co-linear, $\boldsymbol{b} = k \cdot \boldsymbol{a}$, with some scalar k . This tells us how to solve the maximization task.

We get for our **numerator,**

$$|x(n) * h_M(n)|^2 = |\boldsymbol{x}(n) \cdot \boldsymbol{h_M}|^2$$

where we rewrote our convolution, analog to our matrix formulation, as a scalar vector multiplication, with $\boldsymbol{h_M}$ as our **vector** of the **time-reversed** matched filter impulse response, and now with only one row of the signal matrix at a time, for only one convolution sample at a time,

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

where L is the size of our filter vector.
Here we can now apply Cauchy-Schwartz,

$$|\boldsymbol{x}(n)\cdot\boldsymbol{h}_{\boldsymbol{M}}|^2\leq\left\|\boldsymbol{x}(n)\right\|^2\cdot\left\|\boldsymbol{h}_{\boldsymbol{M}}\right\|^2$$

and we get the equality (the maximum) if we set

$$\boldsymbol{h}_{\boldsymbol{M}}=k\cdot\boldsymbol{x}(n)$$

where we can choose the factor as $k=1$.
Since we have this inequality for all time steps n of our convolution, we choose the one where the vector $\boldsymbol{x}(n)$ has the maximum energy. This is where we **capture the entire, non-zero, wave form** of our signal $\boldsymbol{x}(n)$ with our filter.
Since our filter vector $\boldsymbol{h}_{\boldsymbol{M}}$ contains the time-reversed impulse response, we obtain the entire **time reversed signal** as our **matched filter**:

$$h_M(n)=x(L-1-n)$$

assuming our signal to detect is located between $0\leq n< L$.
Since we have a convolution of the signal with its time reverse version, we get a convolution length of $2\mathrm{L}-1$ samples, with its maximum at the center, when both waveforms completely overlap, after L samples, which is exactly the signal length. Hence we get the **detection** of our signal after we completely received it, at the **end of our signal**.
Observe: Since we convolve the signal with the time-reversed version of our pattern to be

detected, this becomes identical to computing the correlation of the signal with the pattern to be detected.
For the continuous-time version see, for instance:
http://www.ece.gatech.edu/research/labs/sarl/tutorials/ECE4606/14-MatchedFilter.pdf

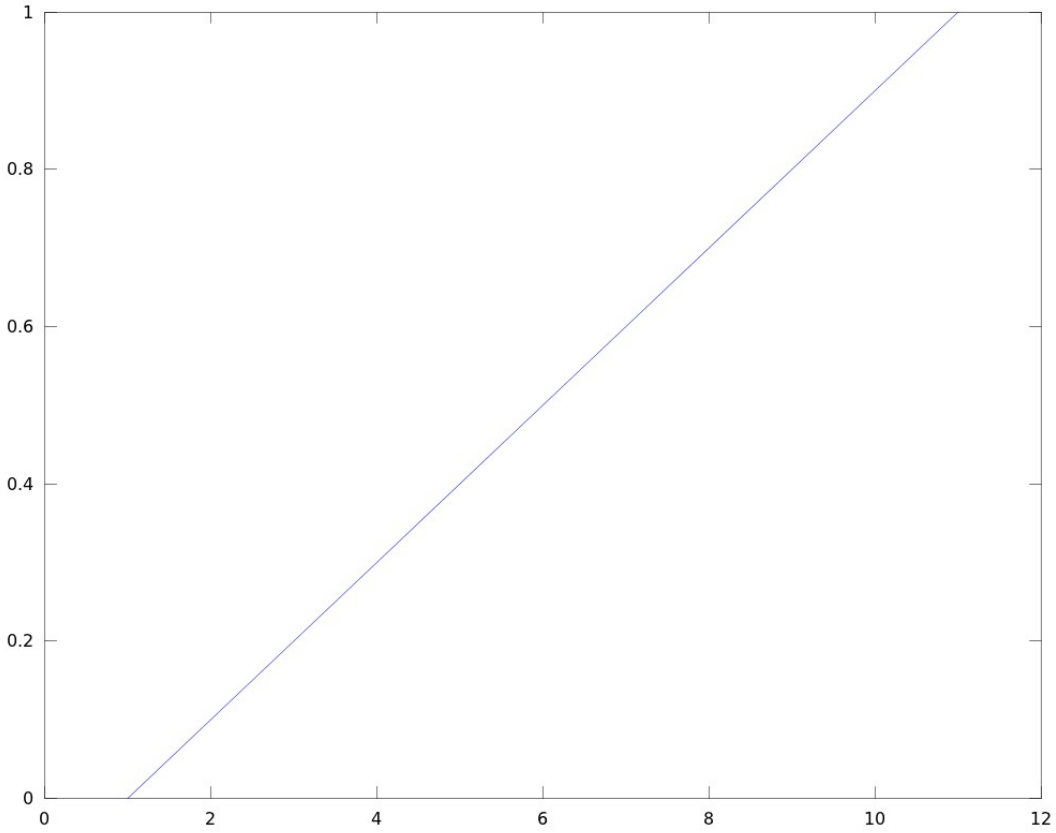
In Conclusion: The matched filter has the shape of the time reversed signal to be searched for.

Matlab/Octave Example

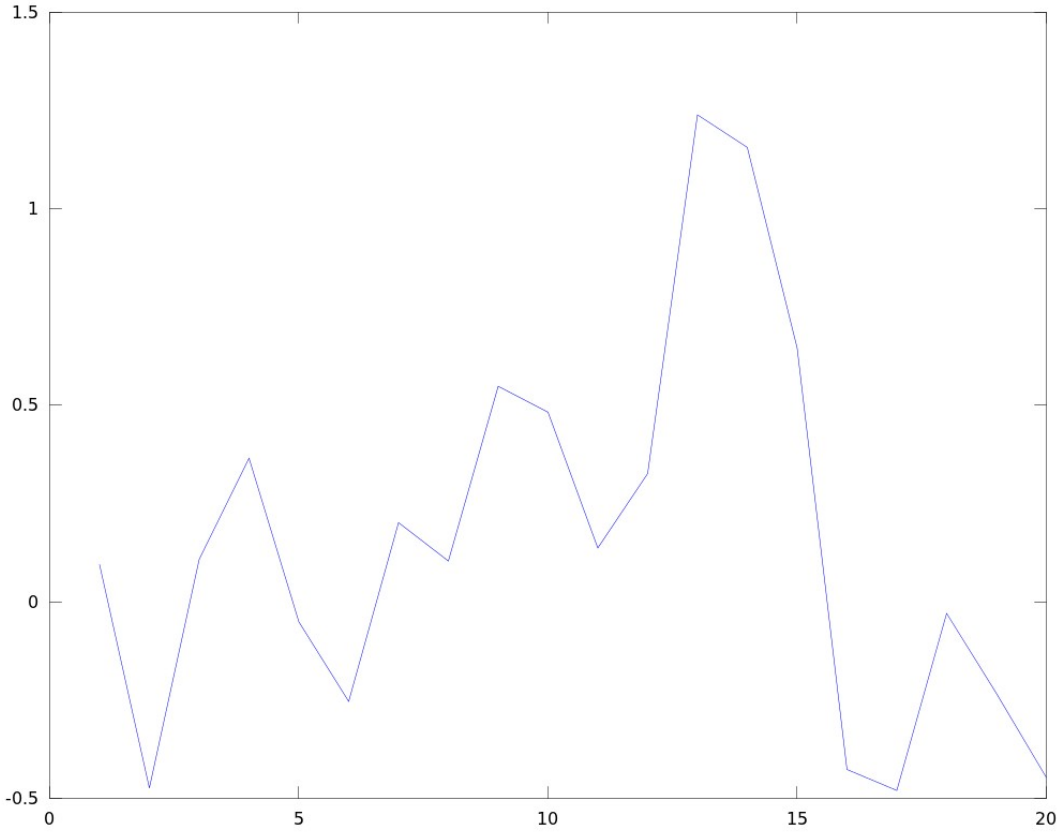
Construct a signal sig (length 11):

```
sig=0:0.1:1
```

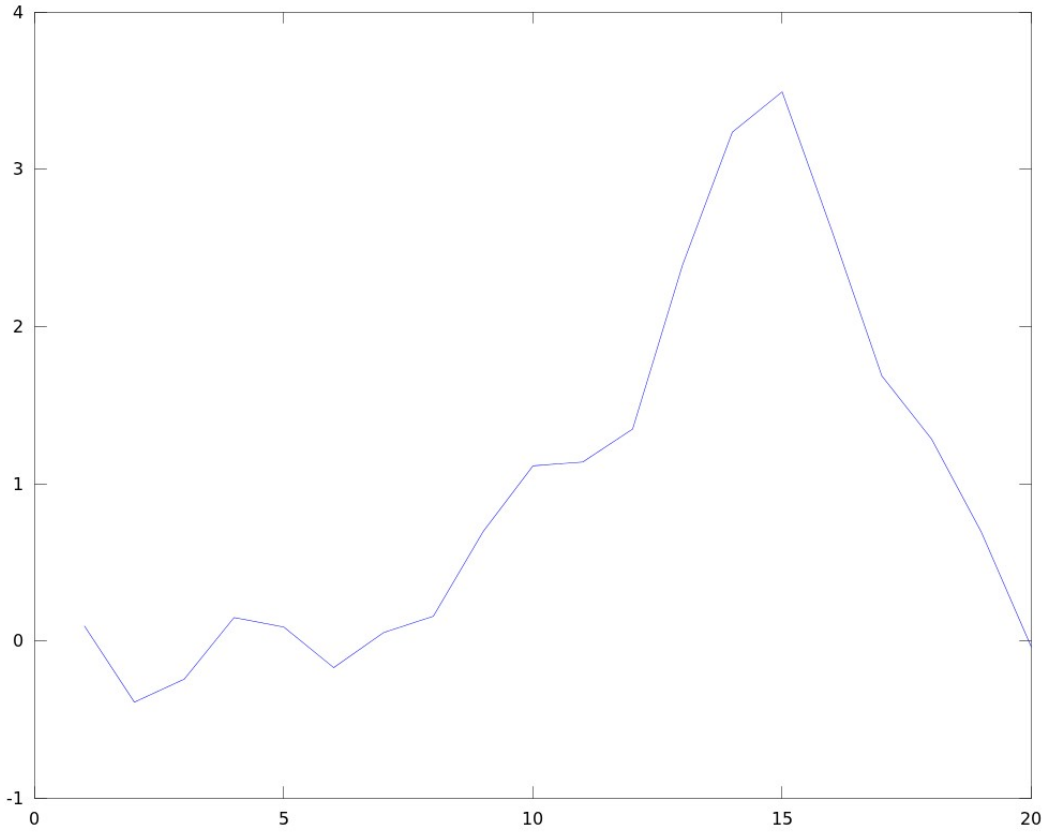
```
%sig =
%Columns 1 through 7:
%0.00000 0.10000 0.20000 0.30000 0.40000 0.50000 0.60000
%Columns 8 through 11:
%0.70000 0.80000 0.90000 1.00000
plot(sig)
```




```
signoise=rand(1,20)-0.5+ [zeros(1,4),sig,zeros(1,5)];
plot(signoise)
```




```
h=fliplr(sig);
signoisemf=filter(h,1,signoise);
plot(signoisemf)
```



This is now the output of our matched filter. We can see that we have a maximum at time 15, which signals the end of our detected signal. Hence we know that the signal started at 15-L(length of the filter)=15-11=4, which was indeed the case since we added 4 zeros in the beginning.
The matched filtering process can also be viewed as computing the correlation function of the noisy signal with the original signal.

Prediction

Prediction can be seen as a special case of a Wiener filter, where the noise of our signal corresponds to a shift of our signal into the past. Our goal is to make a “good” estimation of the present sample of our signal, based on **past signal samples.** “Good” here means, again in a mean squared error sense.
Basically we can now take our Wiener Filter formulation, and specialize it to this case.
Looking at our matrix formulation, we get

$$\begin{bmatrix} 0 & x(0) \\ x(0) & x(1) \\ x(1) & x(2) \\ \vdots & \vdots \end{bmatrix} \cdot \begin{bmatrix} h(2) \\ h(1) \end{bmatrix} = \begin{bmatrix} x(1) \\ x(2) \\ x(3) \\ \vdots \end{bmatrix}$$

This means, the input to our filter is always starting at 1 sample in the past, going down further into the past, and its goal is to estimate or “predict” the next coming sample. Basically this means that instead of additive white noise, our **distortion** is now a **delay** operator (fortunately, this is a linear operator).
Now we can again use our approach with pseudo-inverses to obtain our mean-squared error solution,

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

with the matrix $\boldsymbol{A}$ now defined as our above matrix. This now also leads to a statistical

description, with $A^T \cdot A$ converging to

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

This is plausible, because now y(n) is just the delayed signal, and the auto-correlation function of the delayed signal is identical to the auto-correlation function of the original function.

Next we need the cross-correlation $A^T \cdot x$. Since we now just have this 1 sample delay as our target vector, this converges to the auto-correlation vector, starting at **lag 1**,

$$A^T \cdot x \rightarrow r_{xx} = \begin{bmatrix} r_{xx}(1) \\ r_{xx}(2) \\ \vdots \end{bmatrix}$$

So together we get the solution for our prediction filter as

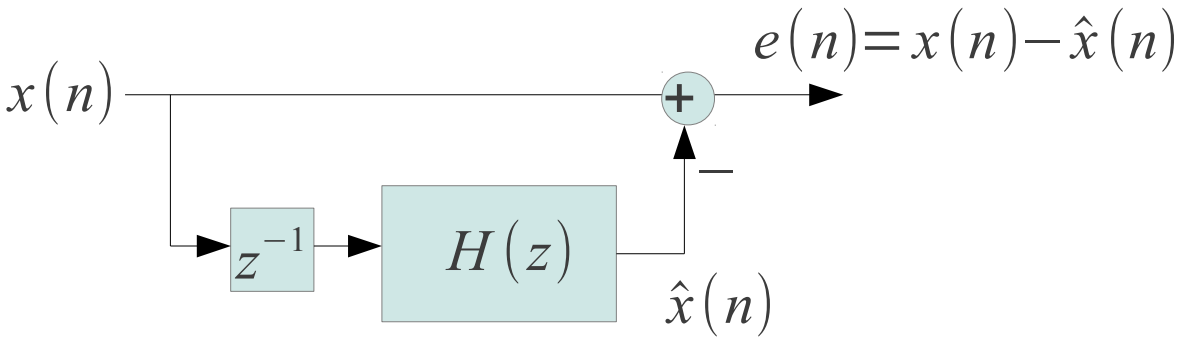
$$h = (R_{xx})^{-1} r_{xx}$$

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

Gerald Schuller, TU Ilmenau

Prediction

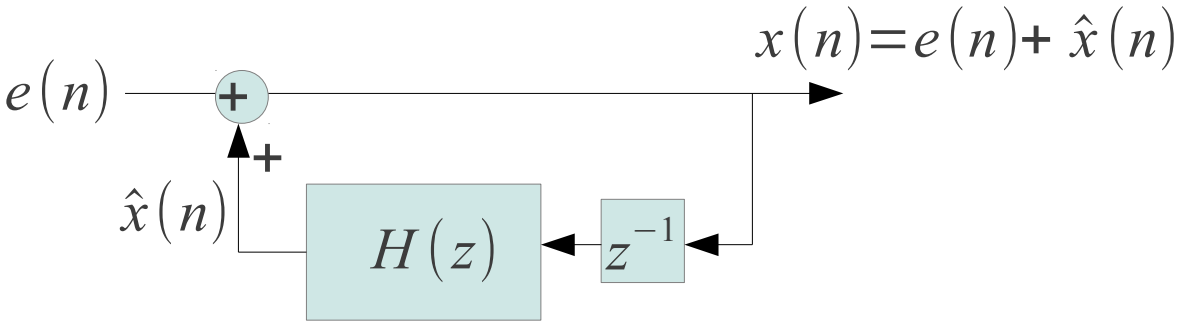
A system which produces the **prediction error** (for instance of part of an encoder) as an output is the following,



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

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

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.

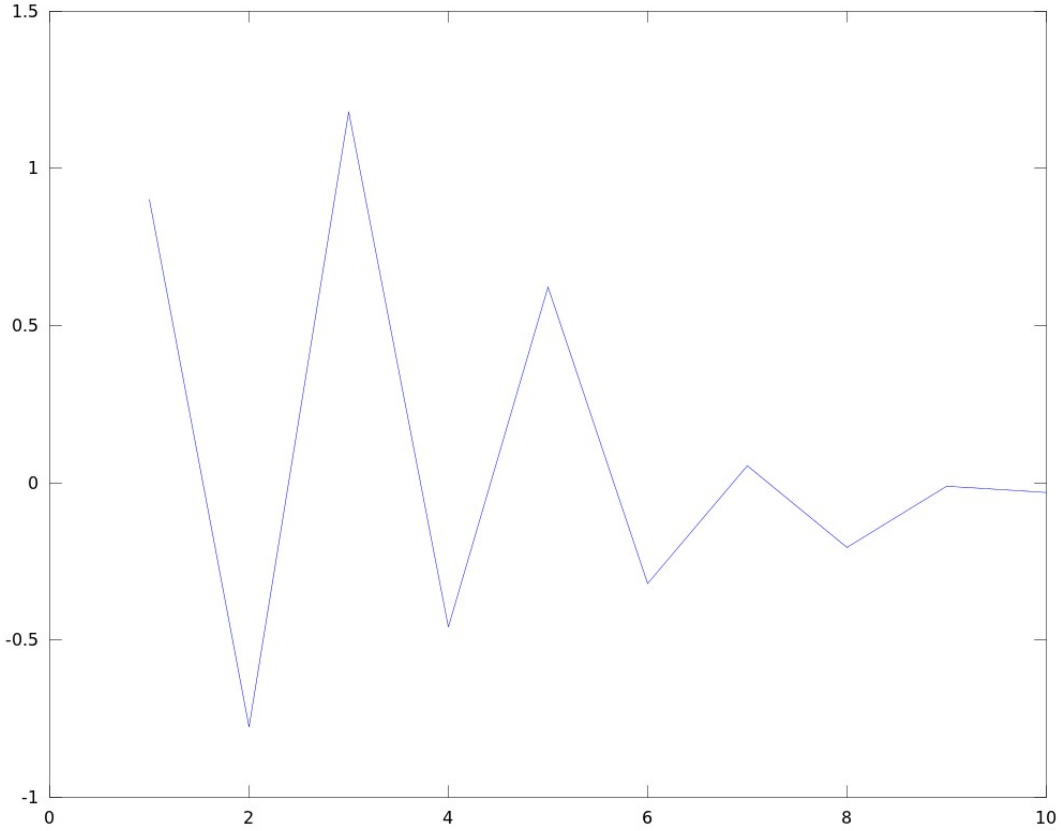
Read in the female speech sound:

```
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))
```

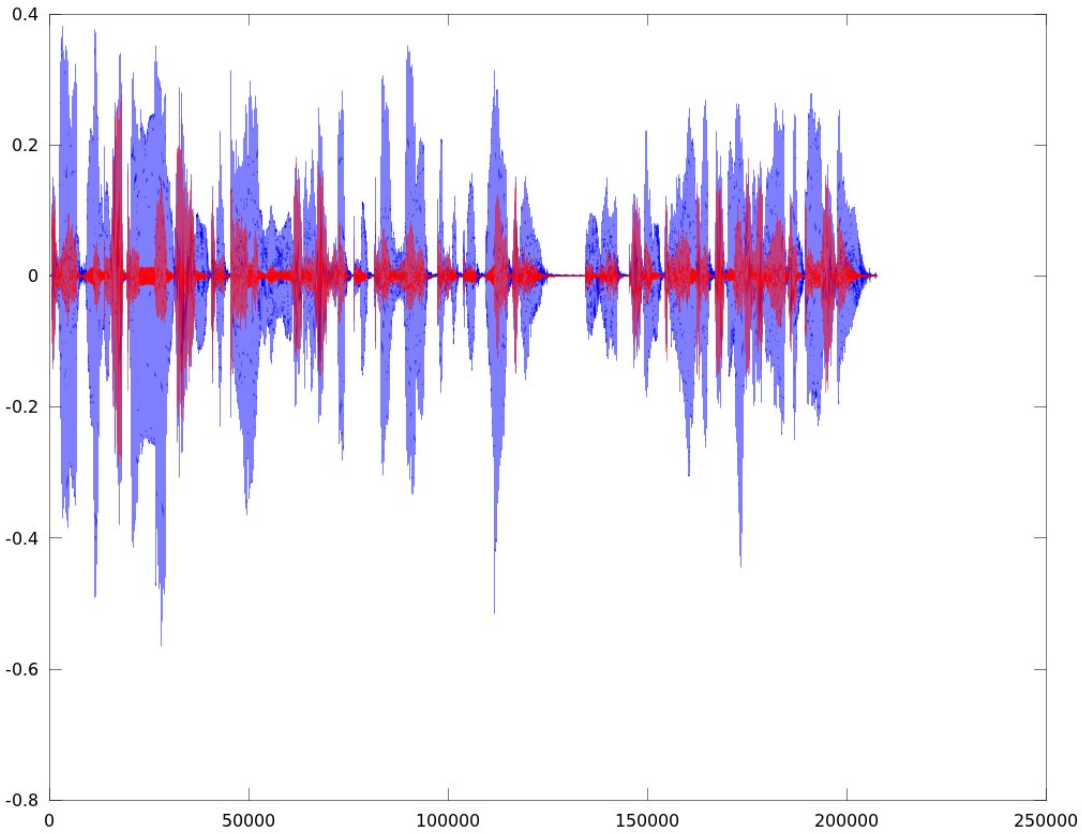


The impulse response of the resulting prediction filter.
Our corresponding prediction error filter is $H_{perr}(z)=1-z^{-1}\cdot H(z)$, in Octave/Matlab:

```
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

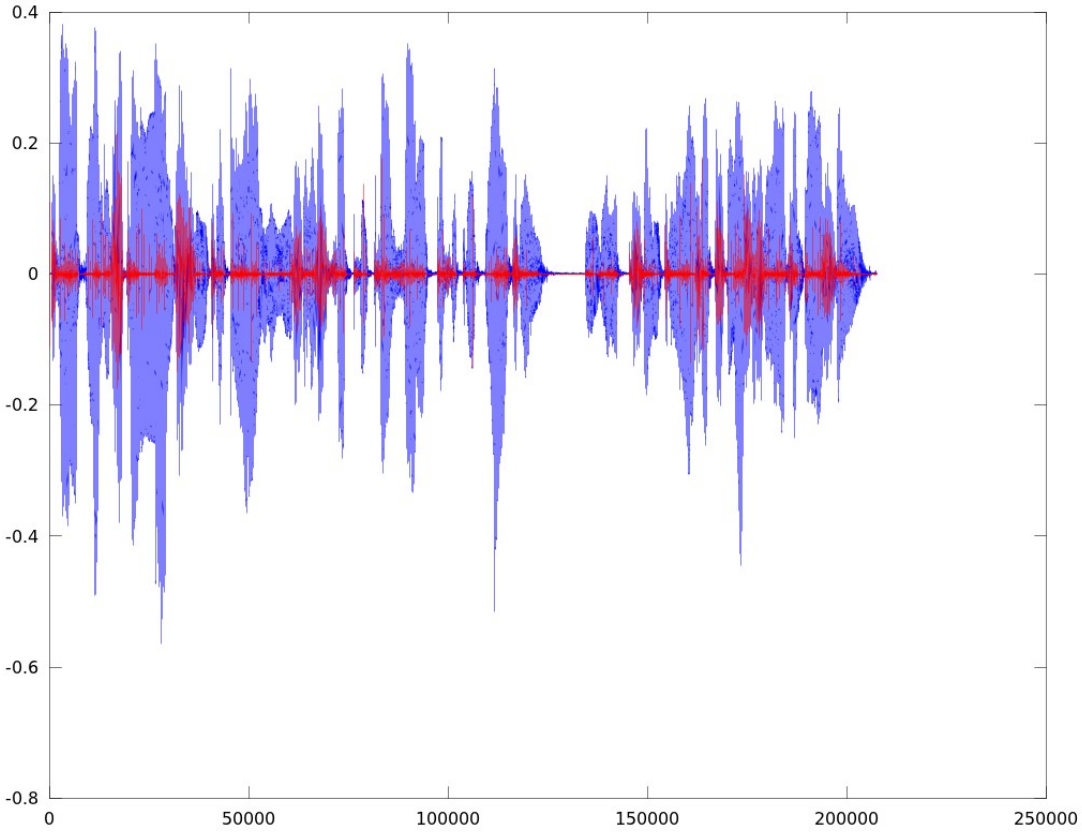
The previous example calculated the prediction coefficients for the entire speech file. But when we look at the signal waveform, we see that its characteristics and hence its statistics is changing, it is **not stationary.** Hence we can expect a prediction improvement if we divide the speech signal 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 $e(n)=x(n)-\hat{x}(n)$),

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

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,

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

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 $k = 1, \ldots L$, 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 $\boldsymbol{x}(n) = \left[x(n-L), x(n-L+1), \ldots, x(n-1)\right]$.

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.

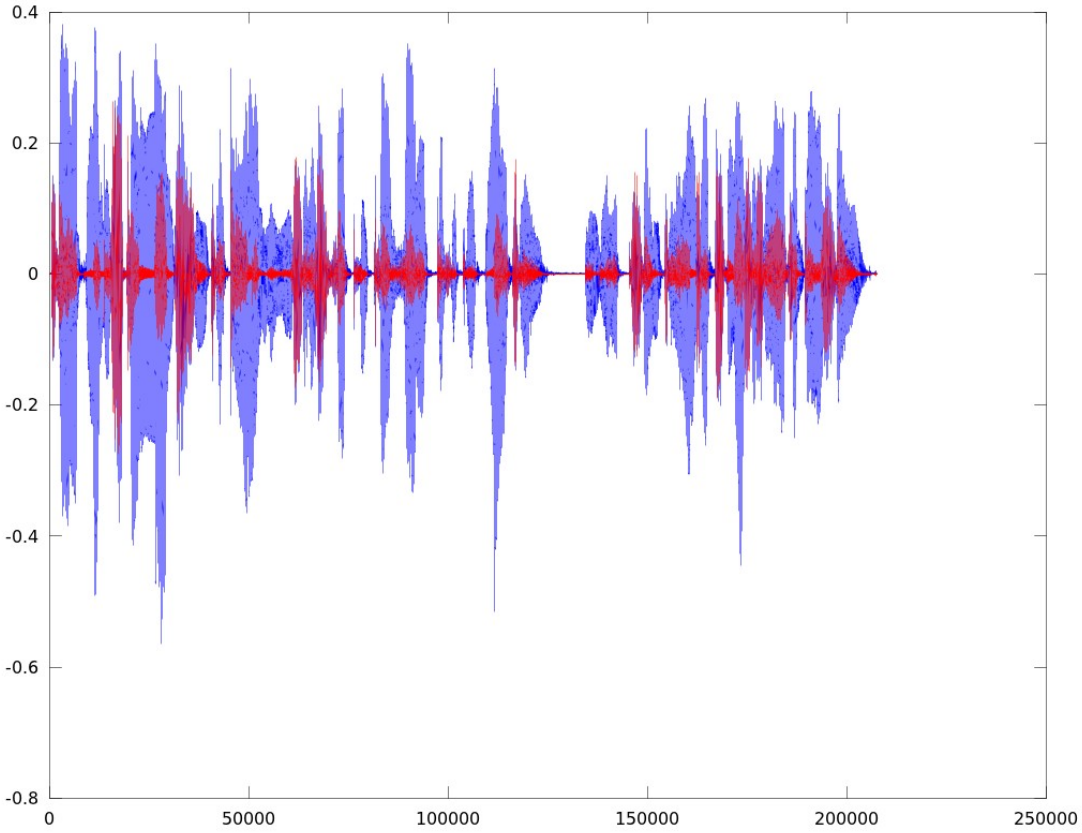
LMS Octave/Matlab Example

```
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

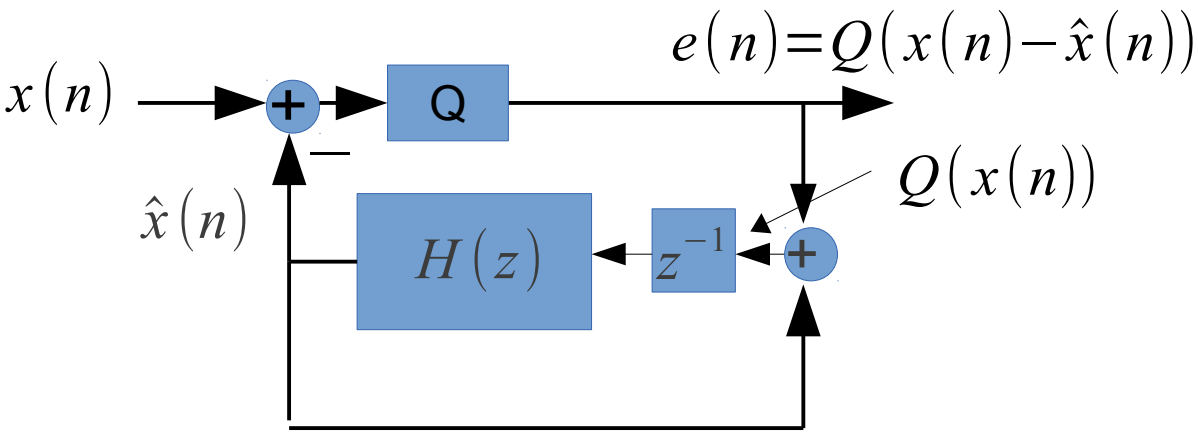
```
h=zeros(10,1);
xrek=zeros(size(x));

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**.