Lecture Audio Coding

Organisatorial Details - Overview

Lectures:
-14 lectures read by Prof. Brandenburg and Prof. Schuller

-**Practice lessons**:

Instructors: Oleg Golokolenko

Periodic homework assignments and quizzes, which will count 30% towards the final grade.

Small groups (2-3 people) to solve the homework and deliver a single solution for the whole group.

-**Tutorials**:

Instructor: Oleg Golokolenko

Topics: Psychoacoustics and Quantization

-**Exam**:

Midterm written exam in the middle of the semester

Final exam at the end of the semester, 90 minutes

Prerequisite:

- Basics of digital signal processing
- Check Moodle 2 website for literature
- Free access to the IEEE database via the university library (http:ieeexplore.ieee.org)

Basics of Multirate Signal Processing

Audio coders process sampled audio signals. As an input signal, they often have audio signals sampled at e.g. 48kHz or 44.1 kHz, and with 16 bits/sample. Internally they filter the signal and **down**- resp. **up-sample** it with **filter banks**, in the encoder and decoder. Hence before we take a closer look at the filter banks, we first take a short look at the basics of Multirate Signal Processing.

Sampling, Downsampling, Upsampling

Sampling the analog signal, normalized frequency

To see what happens when we sample a signal, lets start with the analog audio signal s(t). Sampling it means to sample the signal at sample intervals of T, or the sampling frequency f_s . 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). If we look at the Fourier transform of an analog time continuous system, we get

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

where the superscript c denotes the continuous version, with $\omega=2\pi f$. If we now compute the Fourier transform for the sampled signal, we get a sum, because the Dirac impulse is not zero only at the sample time instances, and the integral over a Dirac impulse is one,

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

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

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

This is now our normalized frequency, where 2π represents the sampling frequency and π is the so-called Nyquist frequency (the upper limit of our usable frequency range).

Important Point: We have a **normalized frequency**, normalized to the sampling frequency, where 2π denotes the sampling frequency, and π is the Nyquist frequency! (since we have no time units in a sampled sequence).

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 calculated with the **Discrete Time Fourier Transform**:

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

Because n in the exponent 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 (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,

$$X(-\Omega)=\overline{X(\Omega)}$$

where $\overline{X}$ denotes the conjugate complex operation, because in the sum of our DTFT we have $e^{-j(-\Omega)n}=e^{j\Omega n}=\overline{e^{-j\Omega n}}$.

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 N'th sample and discard every sample in between. This 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 (just like with the Dirac impulses, but now with 1's instead of the Dirac impulses). This multiplication with this “unit pulse train” can now be used to mathematically analyse this downsampling, first still including the zeros. The frequency response now becomes

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

for all integers m.
Again we can write downsampling 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,

$$\delta_N(n) = \begin{cases} 1, if\ n = mN \\ 0, else \end{cases}$$

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 $\frac{2\pi}{N}k$.
Remember that the inverse Discrete Time Fourier transform is

$$\delta_N(n) = \frac{1}{2\pi} \int_0^{2\pi} \Delta(\Omega) e^{jn\Omega} \, d\Omega$$

Where $\Delta(\Omega) = \frac{2\pi}{N} \sum_{k=0}^{N-1} \delta\left(\frac{2\pi}{N} k\right)$ is the Fourier transform of the unit pulse train.
We can now apply the inverse Fourier transform to our guess, and use it in the time domain,

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

Now we have to prove that this is indeed true (since we only applied a guess). 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),

$$S = \sum_{k=0}^{N-1} c^k \;, \quad S \cdot c = \sum_{k=1}^{N} c^k \;, \quad Sc - S = c^N - 1 \;,$$

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**. 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

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

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

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

We see, that sampling, still including the zeros, leads (in the frequency domain) to **multiple shifted versions of the signal spectrum**, the so-called **aliasing** components (N-1 aliasing components),

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

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

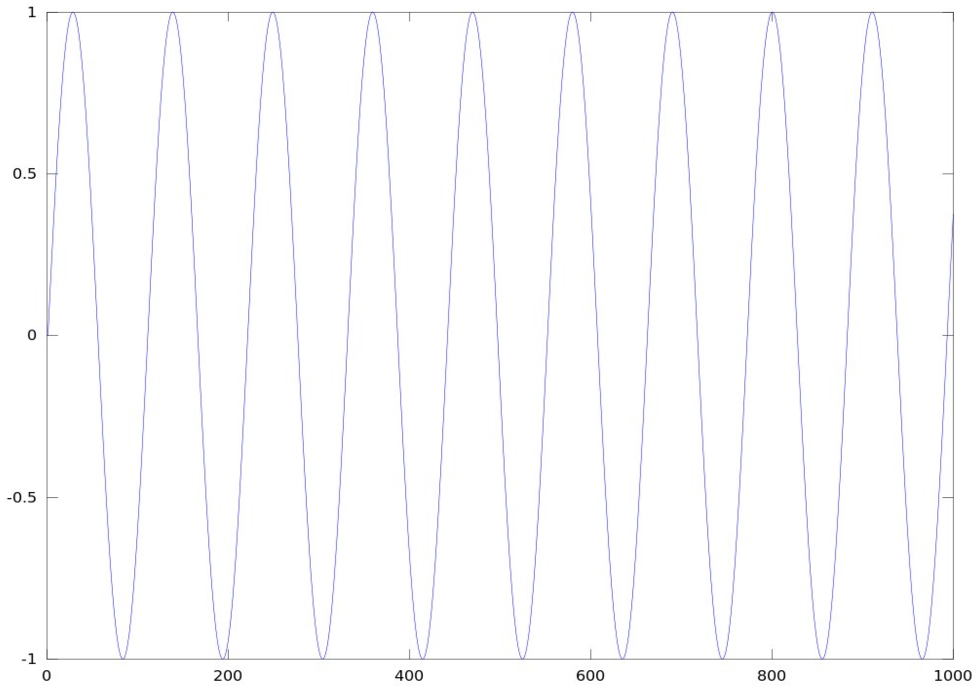
Example:
Make a sine wave which at 44100 Hz sampling rate has a frequency of 400 Hz at 1 second duration. Hence we need 44100 samples, and 400 periods of our sinusoid in this second. Hence we can write our signal Python. Start ipython with pylab:

```
ipython3 -pylab
x=arange(0,1,1/44100.0)
s=sin(2*pi*400*x);
```

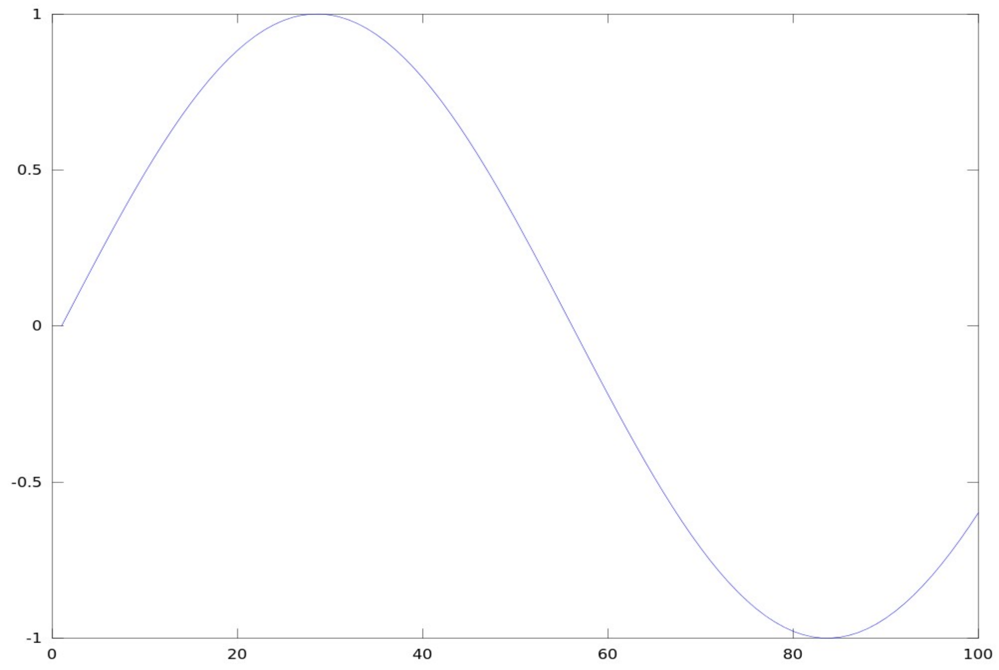
Listen to it (it uses our library sound.py which needs to be loaded from Moodle to the current directory):

```
from sound import *
sound(s*2**15,44100) #scaling by 2**15 for 16 bit range
```

Now plot the first 100 samples:



Next plot the first 100 samples:

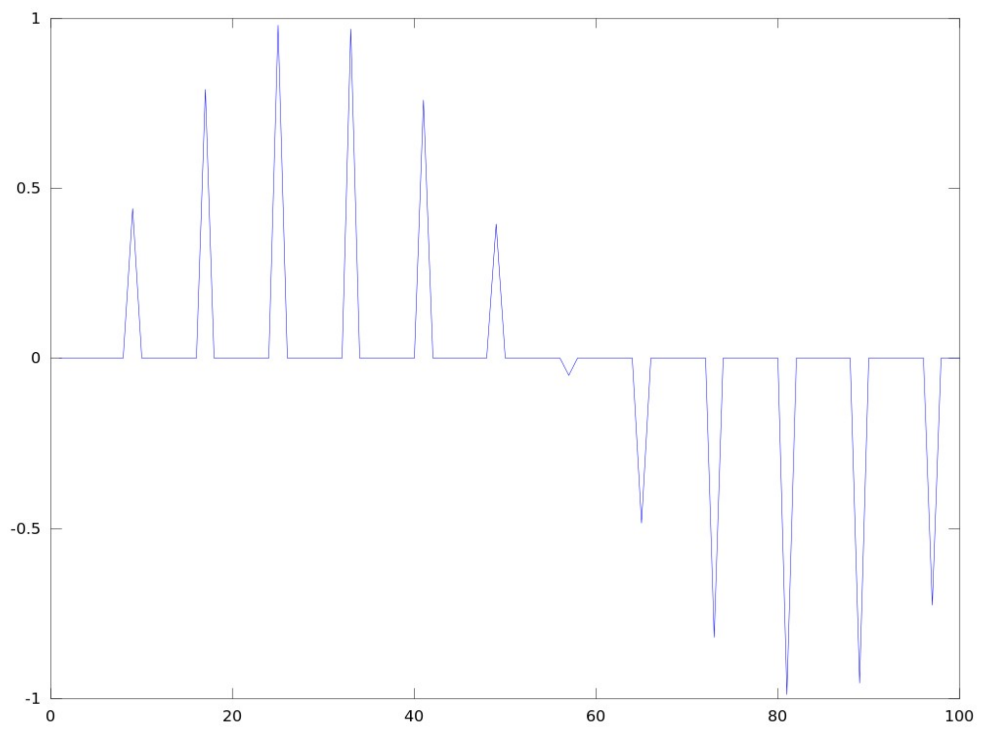


Now we can multiply this sine tone signal with a unit pulse train, with N=8.
We use an **indexing trick** to get the desired result of only keeping every 8th sample and having zeros in between:

```
sdu=np.zeros(44100)
sdu[0:44100:8]=s[0:44100:8];
```

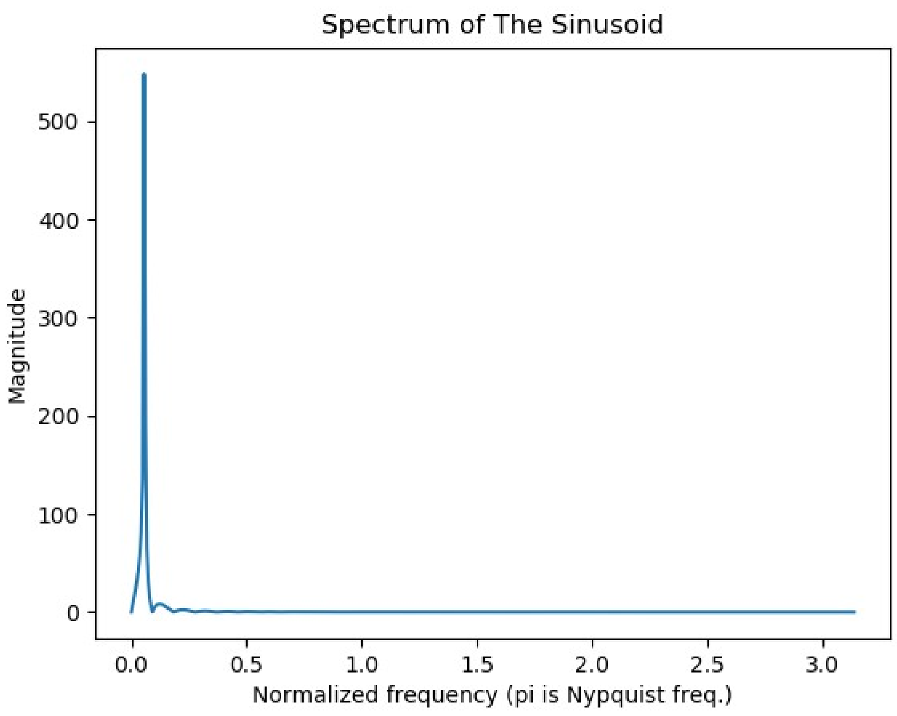
Now plot the result, the first 100 samples:

```
plot(sdu[0:100])
```



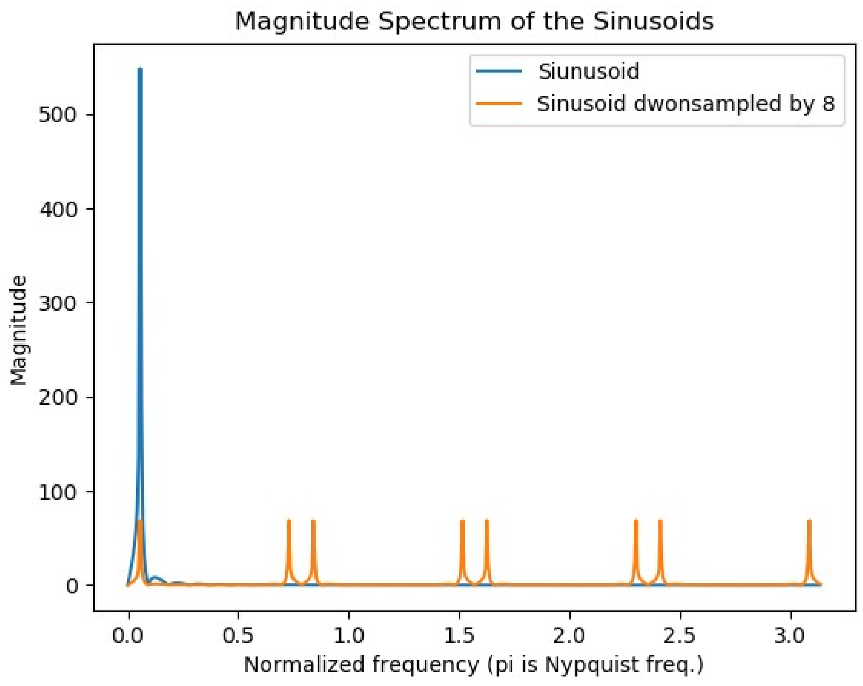
Now take a look at the spectrum of the original signal s:

```
import scipy.signal as sps
#first compute and plot magnitude spectrum of the
sinusoid:
w,h=sps.freqz(s)
plot(w,abs(h))
xlabel('Normalized frequency (pi is Nypquist freq.)')
ylabel('Magnitude')
title('Spectrum of The Sinusoid')
```



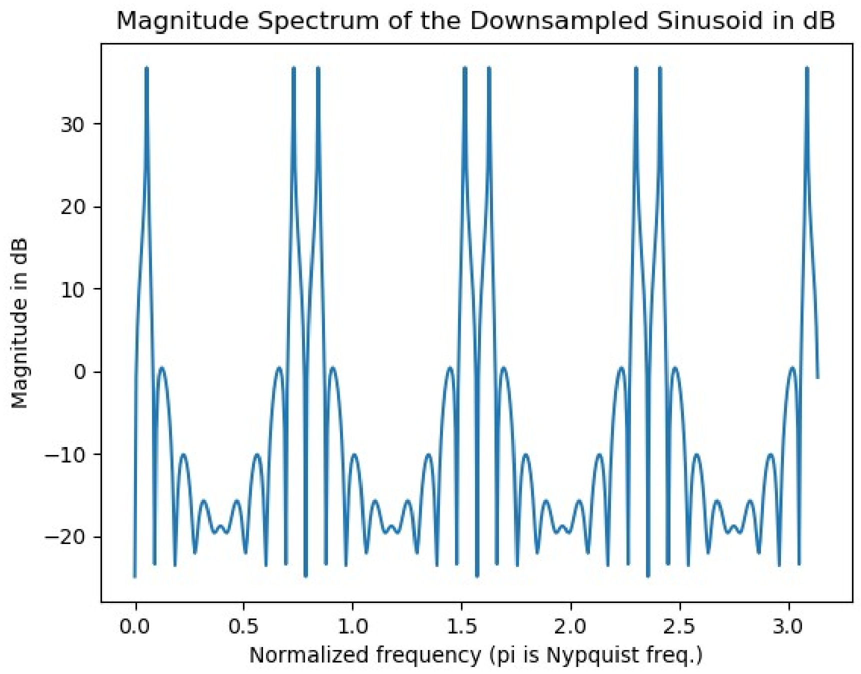
This plot shows the magnitude of the frequency spectrum of our sinus signal. Observe that the frequency axis (horizontal) is a normalized frequency, normalized to the Nyquist frequency (as pi), in our case 22050 Hz. Hence our sinusoid should appear as a peak at normalized frequency 400/22050*3.14=0.057, which we indeed see.
Now we can compare this to our downsampled signal with the zeros in it, sdu:

```
w,hsdu=sps.freqz(sdu)
plot(w,abs(hsdu))
xlabel('Normalized frequency (pi is Nypquist freq.)')
ylabel('Magnitude')
legend(('Siunusoid','Sinusoid dwonsampled by 8'))
title('Magnitude Spectrum of the Sinusoids')
```



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 rather -0.057 ...
Often, the small magnitude components on this plot are also interesting, for that reason magnitude frequency responses are usually plotted on a logarithmic Decibel (dB) scale, applying the function 20*log10(.) to it,

```
figure()
plot(w,20*log10(abs(hsdu)))
xlabel('Normalized frequency (pi is Nypquist freq.)')
ylabel('Magnitude in dB')
title('Magnitude Spectrum of the Downsampled Sinusoid
in dB')
```



Observe that the small magnitudes become much more visible.

Now also listen to the signal with the zeros:

```
sound(sdu*2**15,44100);
```

Here you can hear that it sounds quite different from the original, because of the string of aliasing components!

Real time Python example/demo:
First take a live look at the signal from the microphone, to see its amplitude fluctuations. To run this script you need Python installed, and additionally its library "python-pyaudio", "python-matplotlib" and "python.matplotlib.animation".
Start the script from a Linux shell with

```
python pyrecplotanimation.py
```

Then do the sampling. Start it from a linux shell with:

```
python pyrecplay_samplingblock.py
```

here you can hear the microphone signal as it is sampled at 32 kHz sampling rate, and then multiplied with a unit pulse train, and then played out, again at 32 kHz sampling rate. Observe: you can hear the high frequency aliasing artifacts, which sound a little like ringing. You can play with the script and see how the sound changes when you change the downsampling rate "N" in the script.

Removing the zeros

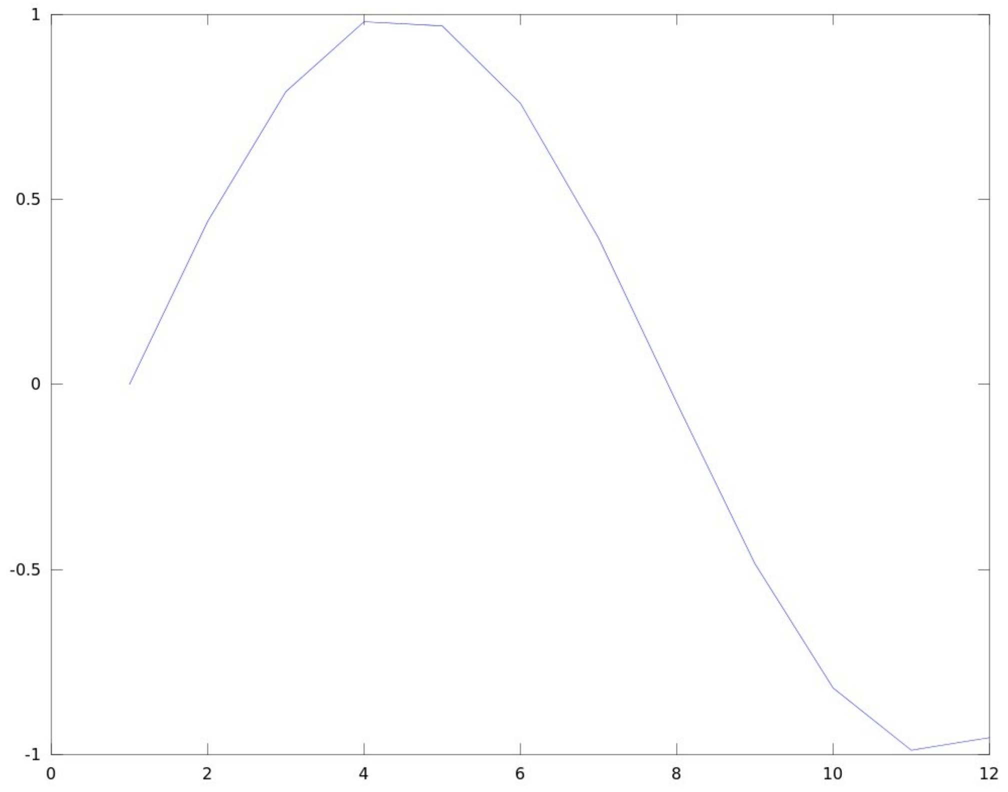
The final step of downsampling is now to omit the zeros between the samples, to obtain the lower sampling rate. Let's call the signal without the zeros

$$y(m),$$

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

In our Python example this is:

```
sd=sdu(0:44101:8);
plot(sd(0:(100/8)));
```



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

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

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

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

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

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 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,

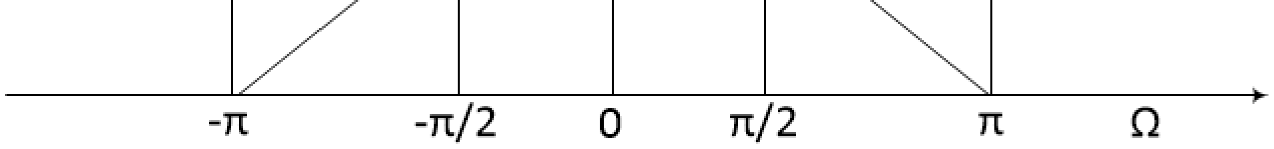


Figure 1: The magnitude spectrum of a signal. The 2 boxes symbolize the passband of an ideal bandpass, here a high pass.

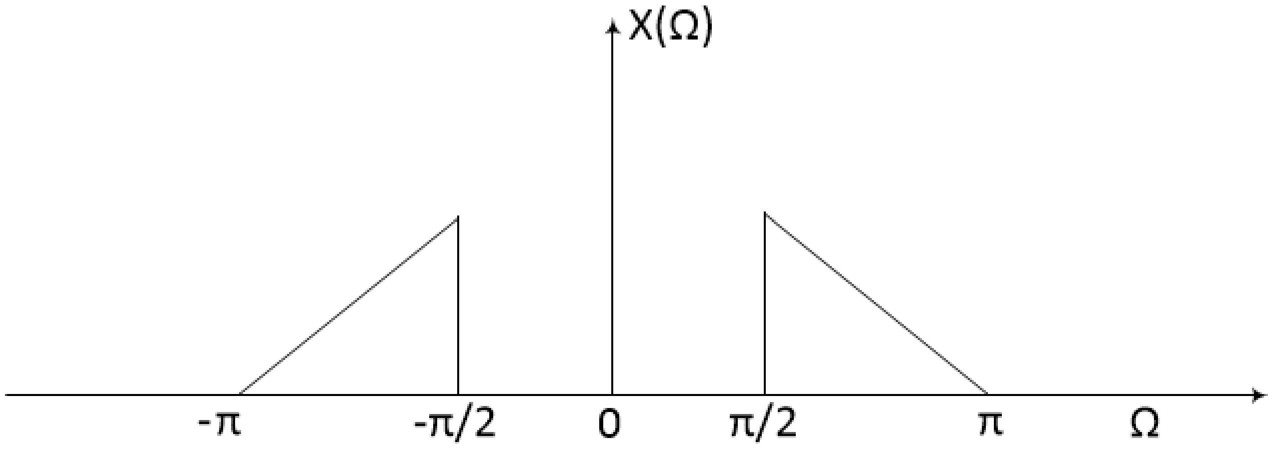


Figure: The signal spectrum after passing through the high pass.

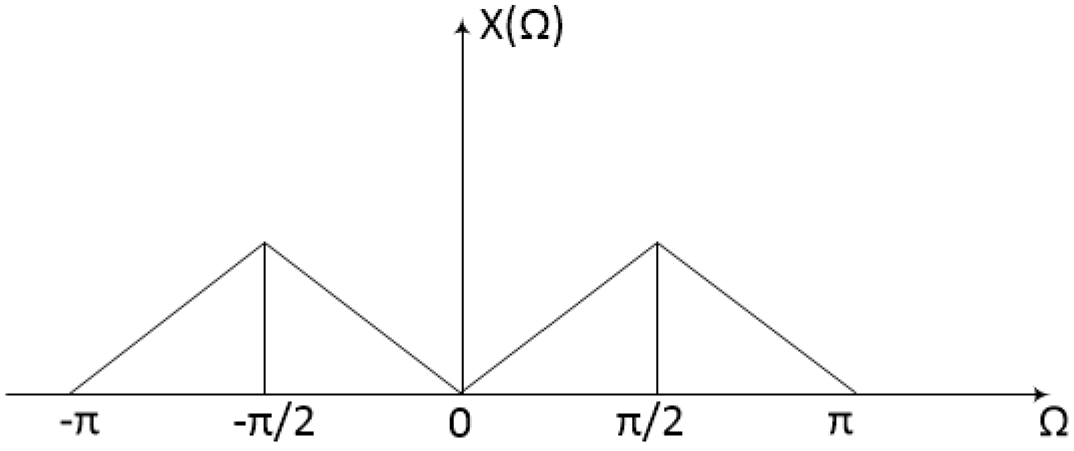


Figure: Signal spectrum after multiplication with the unit pulse train, for N=2, hence setting every second value to zero (the zeros still in the sequence). Observe the we shift and add the signal by multiples of $2\pi/2=\pi$, and we obtain „mirrored“ images of the high frequencies to the low frequencies (since we assume a real valued signal). Observe that the mirrored spectra and the original spectrum don't overlap, which makes reconstruction easy.

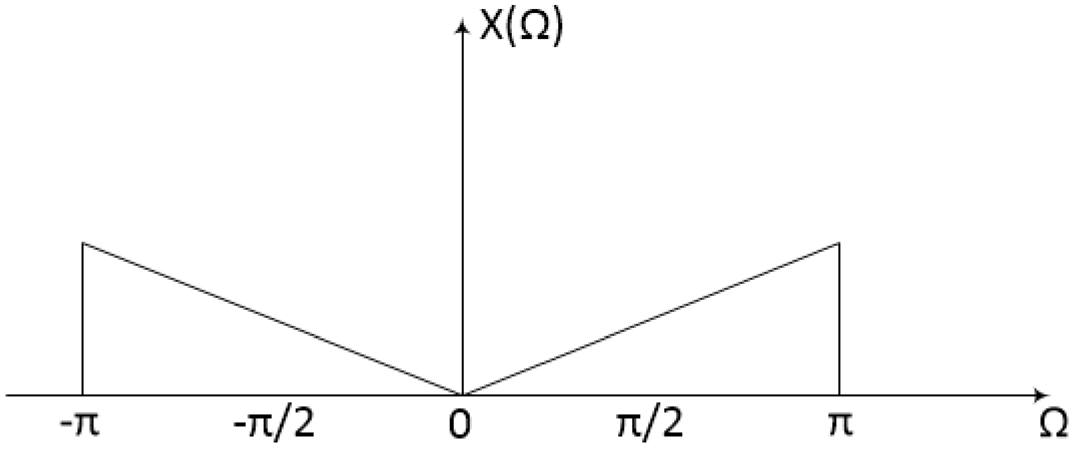


Figure: Signal spectrum after downsampling (removing the zeros) by N (2 in this example). Observe the stretching of the spectrum by a factor of 2.
In conclusion: Removing the N-1 zeros between samples stretches the spectrum of the signal by a factor of N.

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 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:

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

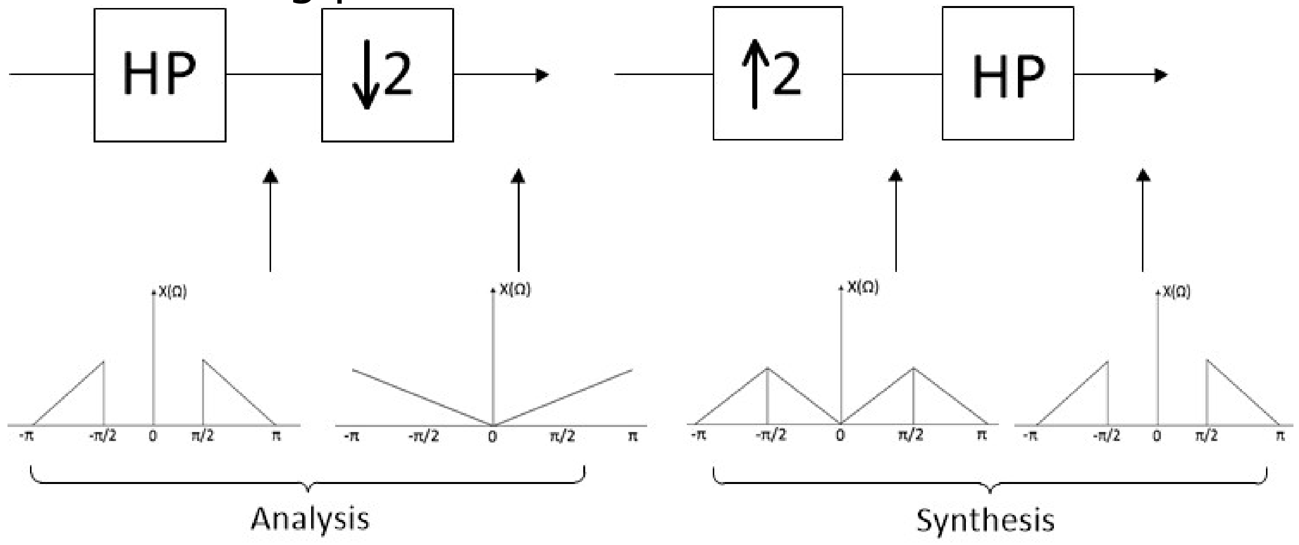
We are now just coming from y(m), going to the now upsampled signal $x^d(n)$.

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 the spectrum before upsampling was also 2π periodic, the extra spectrum we obtain contains a periodic continuation of our spectrum, which again is aliasing, which we would like to supress using filters (lowpass or bandpass) after upsampling.

We saw the effects of downsampling and upsampling. Observe that 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).

Effects in the z-Domain

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

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

Observe that a multiplication of the z-transform with z^{-1} can be interpreted as a **delay** of the signal by **1 sample**. It can be implemented as a **delay** or **memory** element.
First observe that we get our usual frequency response if we evaluate the z-transform along the unit circle in the z-domain,

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

What is now the effect of **multiplying our signal with the unit pulse** train in the z-domain? To see this we simply apply the z-transform,

$$X^d(z)=\sum_{n=0}^{\infty} x(n)\,\delta_N(n)\,z^{-n}$$
$$=\frac{1}{N}\sum_{k=0}^{N-1}\sum_{n=0}^{\infty} x(n)\left(e^{-j\frac{2\pi}{N}\cdot k}\cdot z\right)^{-n}$$
$$=\frac{1}{N}\sum_{k=0}^{N-1} X\left(e^{-j\frac{2\pi}{N}\cdot k}\cdot z\right)$$
or, in short,
$$X^d(z)=\frac{1}{N}\sum_{k=0}^{N-1} X\left(e^{-j\frac{2\pi}{N}\cdot k}\cdot z\right)$$
This is very similar to the Fourier transform formulation, just that we now don't have frequency shifts for the aliasing terms, but a multiplication of their exponential functions to z.

The next effect is the **removal or re-insertion of the zeros** from or into the signal. Let's again use our definition $y(m)=x^d(mN)$, meaning the y(m) is the signal without the zeros. Then the z-transform becomes,
$$Y(z)=\sum_{m=0}^{\infty} y(m)\,z^{-m}=$$
$$=\sum_{m=0}^{\infty} x^d(mN)\,z^{-m}=$$
Replacing the sum index m by n results in
$$=\sum_{n=0}^{\infty} x^d(n)\,z^{-n/N}=X^d(z^{1/N})$$
or, in short,
$$Y(z)=X^d(z^{1/N})$$

Another very useful tool is the so-called **modulation**. This is the multiplication of our signal with a periodic function, for instance an exponential function. It can be written as

$$x_M(n) := x(n) \cdot \mathrm{e}^{-j\Omega_M \cdot n}.$$

Observe that the modulation function here has a periodicity of $2\pi/\Omega_M$. Its z-transform hence becomes

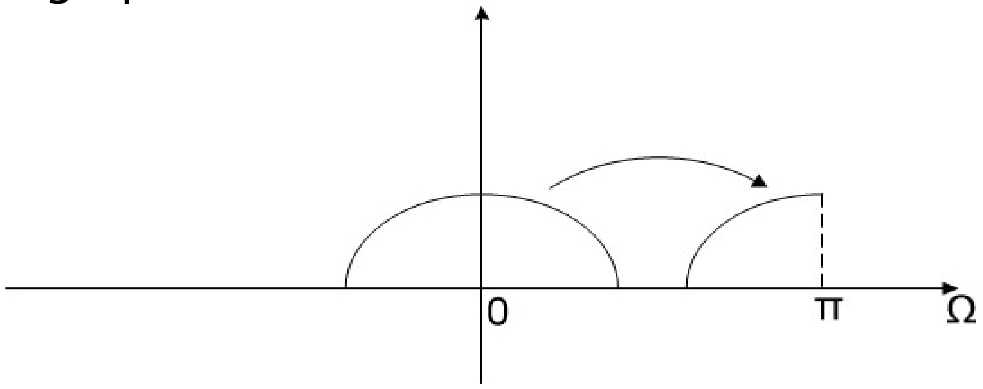
$$X_M(z) = X(\mathrm{e}^{j\Omega_M} \cdot z)$$

Observe that this has the effect of simply **shifting the frequency response** by Ω_M , which can be seen by simply replacing z by $\mathrm{e}^{j\Omega}$ to obtain the frequency response.

Python example to modulating a speech signal:

```
python pyrecplay_modulationblock.py
```

An intersting case is obtained if we set $\Omega_M = \pi$. Then the modulating function is just a sequence of +1,-1,+1,..., and multiplying this with our signal results in a frequency shift of π . If we have a real valued low pass signal, for instance, this will then be shifted by π , and we will obtain a high pass. In this way we can turn the impulse response of a low pass into a high pass, vice versa.



Another important tool is the **reversal of the ordering** of a finite length signal sequence, with length L (meaning x(n) is non-zero only for $n = 0, \ldots, L-1$),

$$x_r(n) := x(L-1-n).$$

Its z-transform is

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

we can now reverse the order of the summation (of course without affecting the result) by starting at the highest index, going to the lowest, replacing the index n by the expression $L-1-n'$ (index substitution),

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

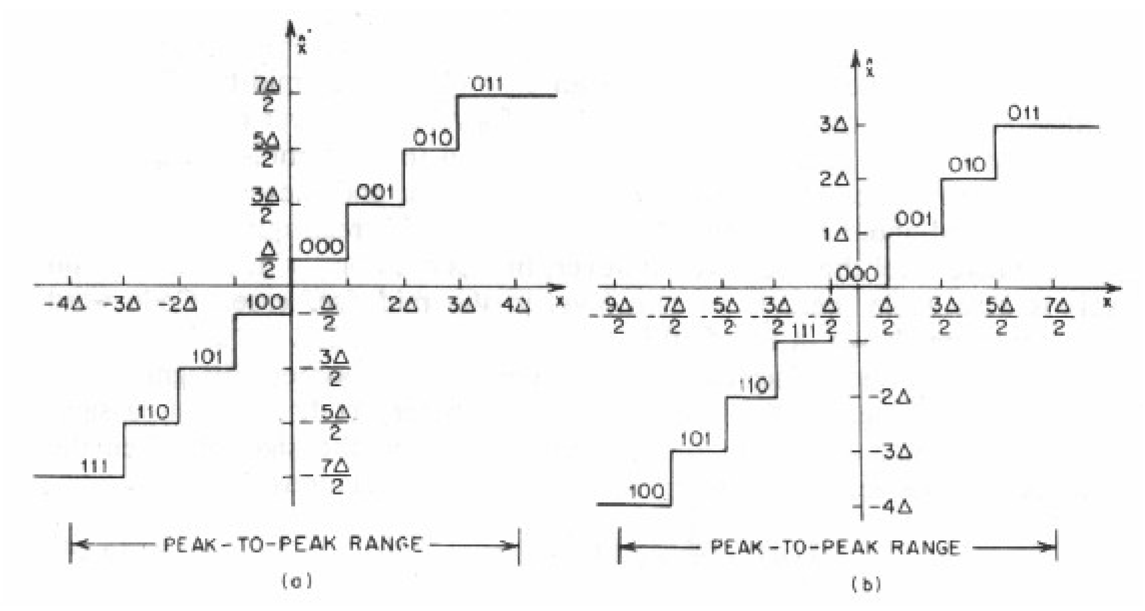
or, in short,

$$X_r(z)=z^{-(L-1)}\cdot X(z^{-1})$$

So what we obtain is the inverse of z in the z-transform (which signifies the time reversal), and a factor of $z^{-(L-1)}$, which is simply a delay of L-1 samples!

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 reverse quantization, which produces the quantized reconstructed value at the receiver. Q is the quantization step size, which is identical to Δ in the plots.
We define the difference between the reverse quantized (through encoder and decoder) and the original signal ($e = Q(f) - f$) as “**quantization error**”.
The resulting quantisation error power in both cases is:

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

Python real time example of the two quantization types: start in a Linux shell with

```
python pyrecplay_quantizationblock.py
```

The signal from the microphone is captured, which is in the range between -32000 to +32000, and then quantized with a

quantization step size of Q=4096.
Observe: Both quantizers distort the signal audibly.
The mid-tread quantizer does not let small signals through, it rounds them to zero, which also saves bits. The mid-rise quantizer does let small signals through, since it quantizes small values either to Q/2 or -Q/2, but distorted, and it also cost some bit rate, since these values need to be encoded.

In General you want to make the quantization step size bigger to save bit rate, but only big enough to avoid audible distortions. You can play with the quantization step size “q” in the script to find the quantization step size such that the distortions are just not audible.