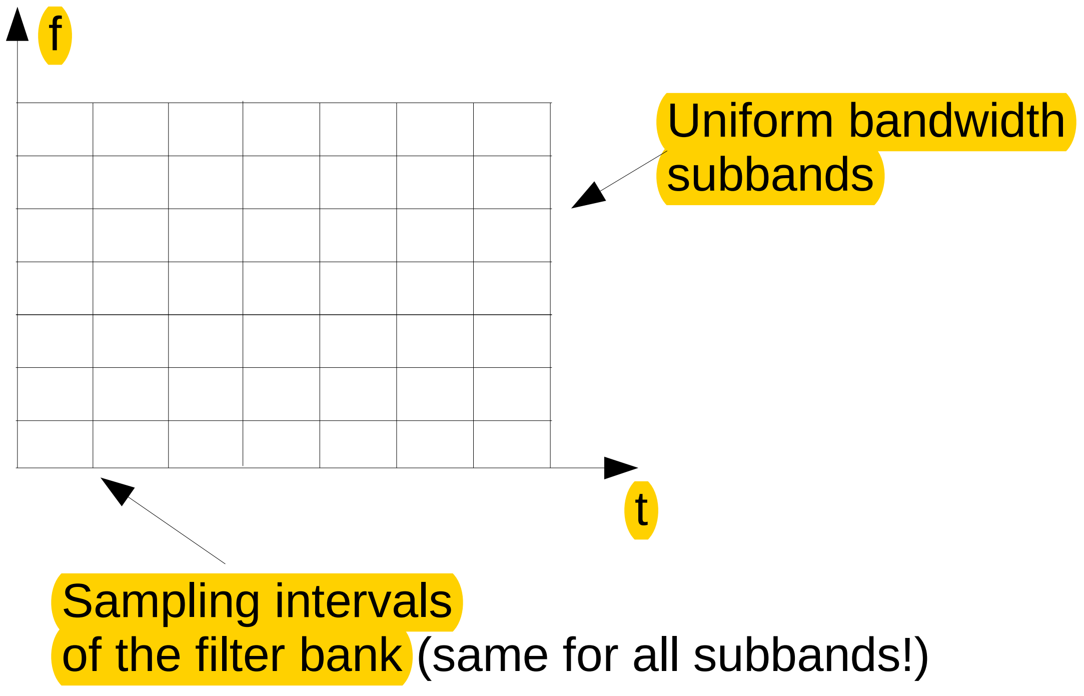
Multirate Signal Processing, Multiresolution

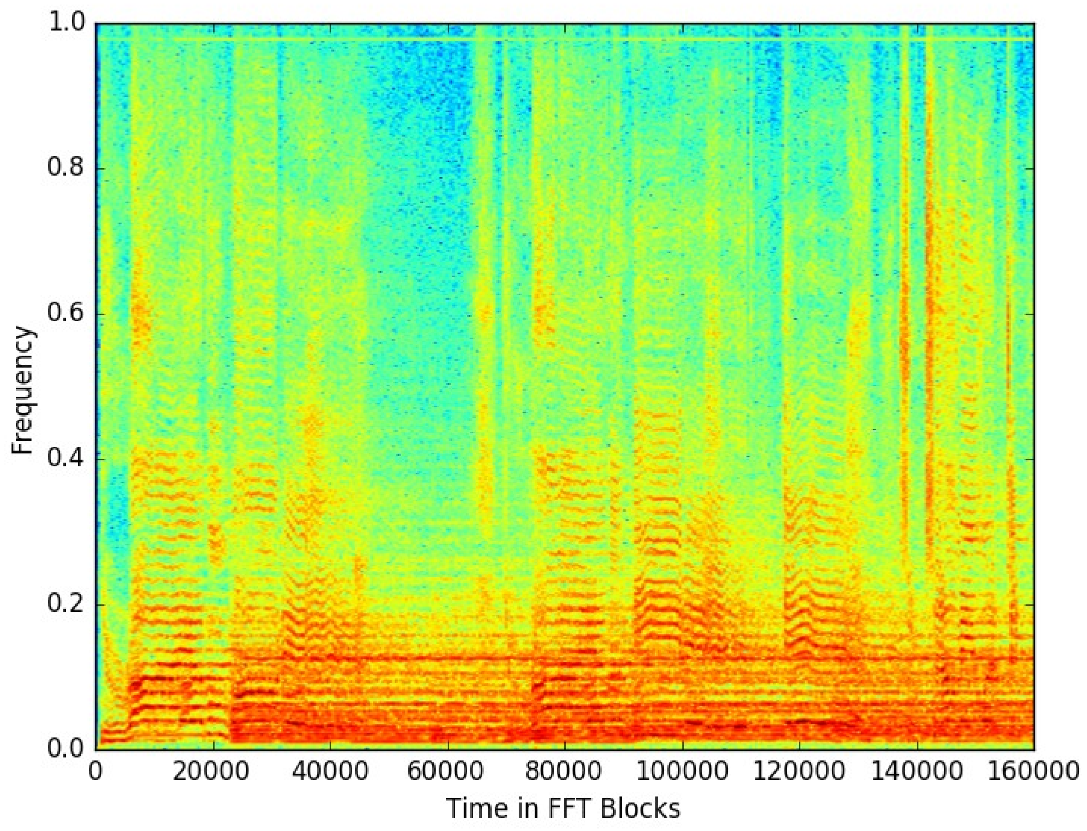
Frequency Decomposition of a **uniform filter bank:**


f
Uniform bandwidth subbands
t
Sampling intervals of the filter bank (same for all subbands!)

Example: a spectrogram shows the amount of signal energy in each of those time/frequency rectangles (also called “tile” or “bin”). In Python a spectrogram is produced with the command “specgram” (in the pyplot library). An example for an audio signal “topchart” is:

```
ipython -pylab

import scipy.io.wavfile as wav

rate, snd = wav.read('04_topchart.wav')

size(snd)

#320000

#A spectrogram with 1024 frequency bins:

specgram(snd,NFFT=1024)
```

Normalized
frequency



Here the **color** indicates the signal energy or magnitude of the coefficient of the underlying FFT bin, at a given time and frequency,
-Blue: low magnitude of the FFT coefficientss or samples
-Red: high magnitudes of the FFT coefficient or samples

To listen to the signal, we use our sound library „sound.py“, which you can find on our Moodle Webpage:

```
from sound import sound
sound(snd,32000)
```

Observe that the size of the time/frequency bins of the FFT is independent of time or frequency.

Real time **Python example**:

```
python pyrecspecwaterfall.py
```

Observe: In the live spectrogram the time is on the vertical axis, and the frequency is on the horizontal axis.

Python Example for Aliasing:

To hear how aliasing sounds if we don't take care of the Shannon-Nyquist theorem, we start the python script with

```
python pyrecspecwaterfallsampling.py
```

Observe: You can hear high pitched artifacts in the voice. This is the aliasing and spectral copies.

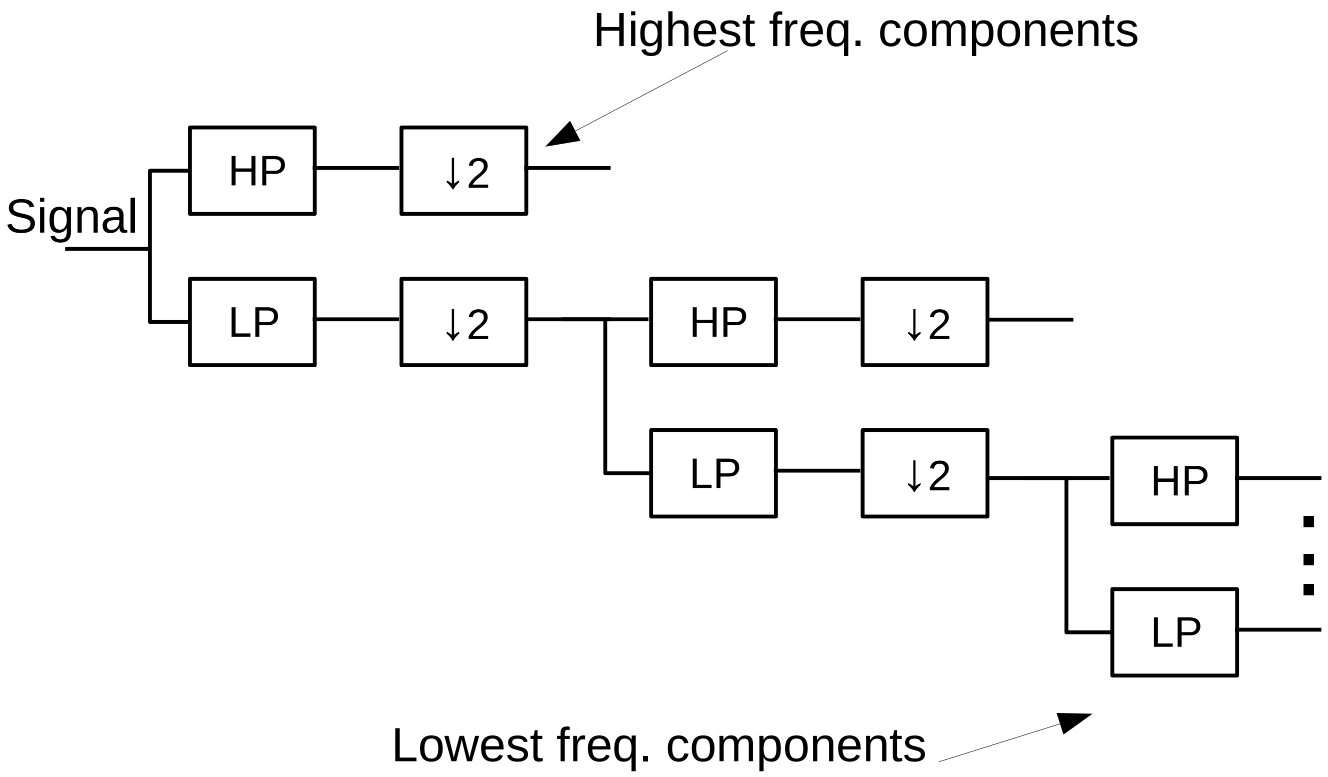
In the waterfall spectrogram you can see the aliasing as spectral copies of the original into higher frequencies.

Also observe how we can reduce the aliasing using the low pass filter. Then the higher spectral copies are attenuated, and we hear fewer artifacts.

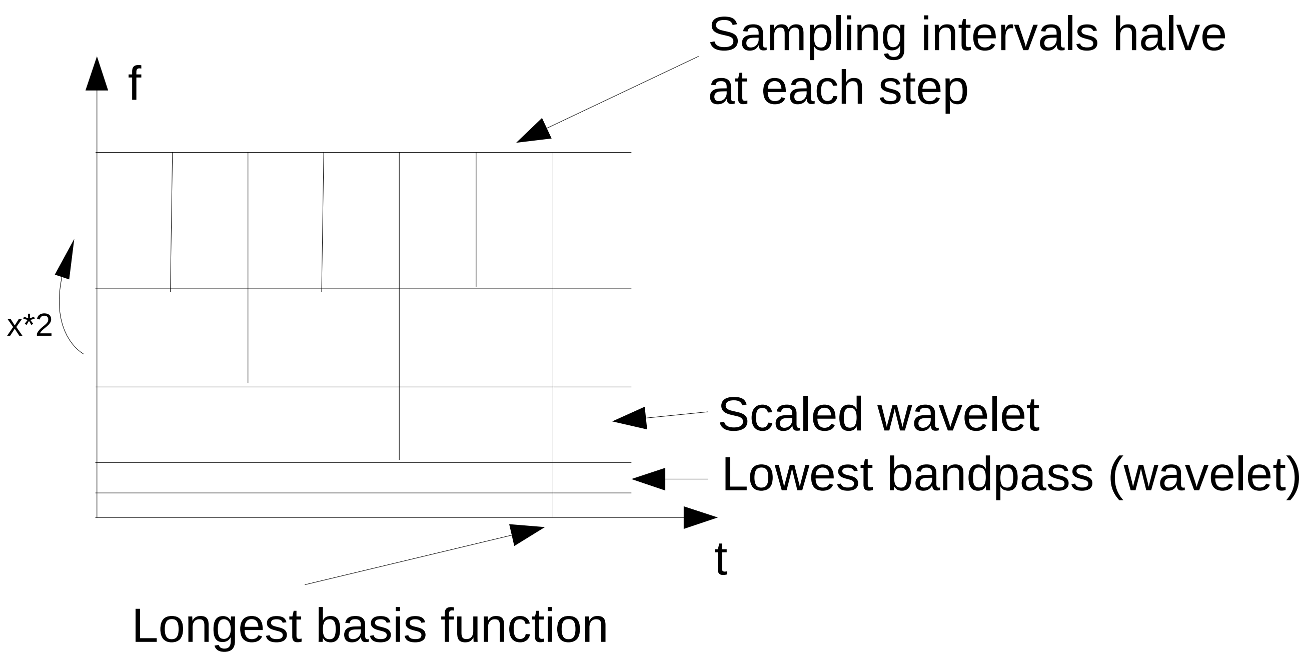
In this example we lost the upper 14 khz of our signal. To avoid this, we use so-called filter banks, which not only use a low pass, but also band passes and a high pass.

Non-Uniform Frequency Decomposition:

Example: The so-called **Discrete Wavelet Transform (DWT)**, the discrete time version of Wavelets, is basically a 2-band decomposition after each low pass filter,



This results in the following time-frequency decomposition,


Sampling intervals halve at each step
f
x*2
Scaled wavelet
Lowest bandpass (wavelet)
t
Longest basis function

Frequency Domain and Notation

-The sign ":=" means "defined as", for instance $a := b$.

-We will use letters like t,T, f for continuous values, and n,m,k,l for discrete values.

-We will use lower case letters for time domain signals, like $x(n)$, and upper case letters for transform domain signals, like $X(\omega)$ or $X(z)$.

-We will use bold face letters to denote vectors or matrices in both time domain and transform domain, like $\boldsymbol{x}(m)$ or $\boldsymbol{X}(z)$.

-The conjugate complex operation is symbolized with a superscript asterisk, like $x^*(n)$.

-For transform domain signals, like $X(z)$, the asterisk ($X^*(z)$) denotes the conjugate complex operation on their coefficients only, not on their argument z .

-The overline $\overline{X(z)}$ denotes the conjugate complex operation on the final result, meaning including the argument z .

- $E(x)$ is the "expectation" of $x(n)$, for our purposes the average of $x(n)$.

$\downarrow N$ Symbolizes downsampling by a factor of N. If $x(n)$ is the signal we downsample, then we also write the downsampled signal as

$$x_{n_0}^{\downarrow N}(m) := x(mN + n_0)$$

where n_0 is the index of the first sample we keep in the downsampling, or the “**phase**”, with $0 \leqslant n_0 \leqslant N-1$.

$\uparrow N$ symbolizes upsampling by a factor of N, including insertion of the zeros. If $y(m)$ is the signal we upsample, then we also write the upsampled signal as

$$y_{n_0}^{\uparrow N}(n) = \begin{cases} y(m), n = mN + n_0 \\ 0, else \end{cases}$$

where n_0 is the index of the first non-zero sample in the upsampled signal, or the phase, with $0 \leqslant n_0 \leqslant N-1$.

Used Types of Frequency Transforms:

Discrete Time Fourier Transform (DTFT):

It is **time-discrete**: Our signal is sampled with sampling interval T, hence our samples only exist at the sampling time-points $t=nT$ (with n integer) or $t=n/f_s$, with the sampling frequency $f_s=1/T$. With an infinite signal length in time we get the forward DTFT:

$$X(e^{j\omega})=\sum_{n=-\infty}^{\infty} x(n)e^{-j(2\pi f/f_s)\cdot n}=\sum_{n=-\infty}^{\infty} x(n)e^{-j\omega\cdot n}$$

where $\omega=2\pi f/f_s$ is the **normalized (angular) frequency**, normalized to the sampling frequency.

Properties: **Continuous in frequency** (because it is not periodic in time in general) and 2π periodic in frequency.

(remember: $e^{j\omega}=\cos(\omega)+j\sin(\omega)$, hence $e^{j(\omega+2\pi)}=e^{j\omega}$). Also for n>1 we have a 2π periodicity, even a $2\pi/n$ periodicity, which in the sum adds up to a 2π periodicity. If $\omega=2\pi$ then $f=f_s$ is the samling frequency, and for $\omega=\pi$ then $f=\frac{1}{2}f_s$ is half the sampling frequency, which we also call the **Nyquist Frequency**.

Observe: If one domain is **discrete**, the other domain is **periodic**!

The **Inverse** Discrete Time Fourier Transform is

$$x(n)=\frac{1}{2\pi}\cdot\int_{\omega=-\pi}^{\pi} X(e^{j\omega})e^{j\omega n}d\omega$$

A **convolution** in time becomes a multiplication in the DTFT domain:

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

(remember: a convolution is defined as $y(n)=x(n)*h(n)=\sum_{m=-\infty}^{\infty} x(m)\cdot h(n-m)$, which is mathematically more complicated than a simple multiplication in the transform domain!)

Discrete Fourier Transform (DFT):

It is **time-discrete** and **finite in time** with signal length of N samples, where it is assumed that beyond this signal block the signal is periodic with period N. The forward transform is defined as,

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

with the discrete frequency index $k=0,\ldots,N-1$.

We transform N samples in the time domain into N samples in the frequency domain.

Observe: we obtain the **DFT from the DTFT** as a special case, if we take the finite length signal $x(n)$ and make it **periodic** by repeating it infinitely many times into the future and the past. Then we apply the DTFT to it, and we obtain a **discrete spectrum**, since the signal is periodic. The spectrum consists of the **fundamental frequency**, which is the inverse of the length of $x(n)$, and of its **harmonics at multiple frequencies of the fundamental frequency**. The coefficients of these frequencies are output of the DFT.

In **Python**:

```
scipy.fftpack.fft(..)
```

Properties: **Discrete in frequency**, because of finite extend in time, **periodic in time**. Hence the frequency index is now k and an integer number between 0 and N-1. It is periodic in frequency, with period N, because it is discrete in time (k=N corresponds to the normalized frequency 2π , our sampling frequency).

Its **inverse transform** is:

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

In **Python**:

```
scipy.fftpack.ifft(..)
```

Since the DFT assumes the signal to be **periodic** with period N, the convolution of this periodic signal, called **circular convolution**, becomes a multiplication in the DFT domain.

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

where the circular convolution is

$$x(n) *_c y(n) := \sum_{n'=0}^{N-1} x(n') \cdot y((n-n') \, mod \, N)$$

where mod is the Modulus function, which is the remainder after division by N, and is between 0 and N-1. Corresponds to the Python numpy function “remainder”.

Observe: A finite/periodic signal in time leads to a discrete spectrum, and a sampled, time-discrete signal leads to a periodic spectrum. Because of the symmetry between time and frequency the same holds also the other way around (discrete frequency → periodic time, periodic frequency → discrete time).

DCT:

Another widely used block transform is the Discrete Cosine Transform. The so-called **DCT Type 4** is defined as

$$y_k(m) = \sqrt{\frac{2}{N}} \sum_{n=0}^{N-1} x(mN+n) \cdot \cos\left(\frac{\pi}{N}(n+0.5)(k+0.5)\right)$$

for the analysis (k=0,...,N-1). Characteristic here is the shift of 0.5 in the subband index k and the time index n. Observe the index m, it is the block index, used for processing streams.

In **Python**:

`scipy.fftpack.dct(..,norm='ortho')`
Unfortunately, the fftpack only has the dct types 1 to 3, and not 4. The type 4 can be obtained using the type 3.

For instance:

```
def DCT4(samples):
    """samples : (1D Array) Input samples
     to be transformed
     Returns:y (1D Array) Transformed
    output samples
    """
    import numpy as np
    N=len(samples)
    # Initialize
    samplesup=np.zeros(2*N)
    # Upsample signal:
    samplesup[1::2]=samples
    y = spfft.dct(samplesup,type=3,
    norm='ortho')*np.sqrt(2)
    return y[0:N]
```

The **inverse DCT 4** transform is

$$x(mN+n)=\sqrt{\frac{2}{N}}\sum_{k=0}^{N-1} y_k(m)\cdot\cos\left(\frac{\pi}{N}(n+0.5)(k+0.5)\right)$$

Observe that it is identical to the forward transform. In general, the DCT-4 inverse is identical to its forward transform, but up to a factor. If this factor is 1, then it is an “orthonormal” transform. The python function needs an argument norm=’ortho’ for this property.

In Python:

`scipy.fftpack.idct(..,norm='ortho')`
again, type 4 can be obtained using type 3, and is the **same function** as the forward transform.

Also observe: The DCT provides twice the subband “density” compared to the DFT, the frequency distance between 2 neighbouring subbands is only half as much as for the DFT, which can be seen at the factor of π/N instead of $2\pi/N$ at the DFT.

The z-Transform

Recall that the Discrete Time Fourier Transform is:

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

In reduced notation:

$$X(\omega) = \sum_{n=-\infty}^{\infty} x(n) e^{-jn\omega}$$

Observe that the expression $e^{j\omega}$ is a complex number on the unit circle on the complex plane. We can replace it by a more general complex number z, which does not need to be on the unit circle, and obtain the **z-transform**:

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

Observe: The z-transform simply turns a **sequence** into a **polynomial**!

In **Python** for the z-transform of a finite length sequence $x(n)$ is:

```
z=sympy.symbols('z')
```

```
xz=sympy.Poly(np.flipud(x), z**(-1))
```

The z-transform is more powerful because it is more general. Using it we can also determine, for instance, if a system or a signal has damping on it (in that case the signal would be inside the unit circle). Also, we can compute the z-transform even for unstable signals or systems (a pole outside the unit circle, exponentially growing signals), which would not be possible for the DTFT, because its sum would not converge.

Observe: This is the so-called two-sided z-transform, because the time-index n is running from minus infinity to infinity, hence it already existed for an infinite time. Usually we only deal with causal signals, which start at a certain point in time, and this yields the one-sided z-transform, which starts at n=0:

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

Convolution in time becomes a **multiplication** in the z-domain.

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

Example: Linear Audio amplifier and an audio input signal. The frequency response of the amplifier can be H(z) or $H(\omega)$ (depending on the transform), and the frequency response or spectrum of the signal would be X(z) or $X(\omega)$. The output of the amplifier would be Y(z)=X(z)H(z) or $Y(\omega) = X(\omega) \cdot H(\omega)$. We basically multiply the frequency components of our signal with the attenuation or gain at that frequency of the amplifier. Basically H is the frequency response of our amplifier.

Observe that **unlike a convolution** in the time domain, we can **invert the multiplication** in the frequency domain. This is a principle that is used for instance for equalizers, who try to flatten the amplifier frequency response by having a frequency response which is inverse to the frequency response of the amplifier (at each frequency bin).

The **inverse** z -transform is

$$x(n)=\frac{1}{2\pi j}\oint_{c} X(z) z^{n-1} dz$$

where C is a closed contour in the complex z-plane which contains all poles of $X(z)$ in its inside, and its path is followed counter-clockwise in the mathematically positive sense.

If we have a causal, stable z-transform $X(z)$ then all poles are inside the unit circle, hence our contour C can be the unit circle around the origin, and we can set $z=e^{j\omega}$ for $0\leq\omega<2\pi$. We can now apply an integration variable substitution

$$\frac{dz}{d\omega}=j e^{j\omega}$$

$$\rightarrow dz=d\omega\cdot j e^{j\omega}$$

(using the so-called Leibnitz notation).

Now the contour integral becomes

$$x(n)=\frac{1}{2\pi j}\oint_{c} X(z) z^{n-1} dz$$

$$x(n) = \frac{1}{2\pi j}\int_{\omega=0}^{2\pi} X(e^{j\omega}) e^{j\omega(n-1)} d\omega \cdot j e^{j\omega}$$
$$= \frac{1}{2\pi}\int_{\omega=0}^{2\pi} X(e^{j\omega}) e^{j\omega n} d\omega$$

Now we can see that the last line is **identical** to the **inverse DTFT**, which means that for **causal stable** signals or systems the inverse z-transform becomes identical to the inverse DTFT.

Observe: Finte length sequences are always causal and stable, hence their inverse z-Transform becomes the inverse DTFT, which is the **coefficients** of the z-Transform **polynomial**.

Inverse z-tramsform for finite length sequences in **Python**:

```
y=np.flipud(yz.coeffs())
```

Observe that we have transform pairings of:

*periodic time - discrete frequency, and
*discrete time - periodic frequency.

Observe: In practice we always have **discrete finite signals**, which are assumed to be periodically continued into infinity. Hence our **frequency domain** will be **discrete and periodic**.

Example: x(n) is an exponentially decaying function, starting at n=0 with x(0)=1,

$$x(n) := \begin{Bmatrix} 0.5^n \; for \; n \geq 0 \\ 0 \; else \end{Bmatrix}$$

$$\rightarrow X(z) = \sum_{n=0}^{\infty} (0.5 \cdot z^{-1})^n = \frac{1}{1 - 0.5 \cdot z^{-1}}$$

Observe that the sum of the z-transform converges for $|z| > 0.5$. This is also called the „Region of Convergence“ (ROC).

Here we can also see that we obtain a pole of the expression for z=0.5.

In this way we can also see what time signal corresponds to a pole (a exponential time function). We have a **correspondence**:

exponential time function- **pole** in z-domain

Explanation for the geometric sum:

A geometric sum has the following form:

$$S=\sum_{n=0}^{N} a^n$$

with some constant a. What is S?

We apply a trick: compute aS:

$$a\cdot S=\sum_{n=0}^{N} a^{(n+1)}=\sum_{n=1}^{N+1} a^n$$

Now take aS-S:

$$S\cdot(a-1)=aS-S=a^{N+1}-1$$

$$S=\frac{a^{N+1}-1}{a-1}=\frac{1-a^{N+1}}{1-a}$$

Hence we get:

$$\sum_{n=0}^{N}(0.5\cdot z^{-1})^n=\frac{1-(0.5\cdot z^{-1})^{N+1}}{1-(0.5\cdot z^{-1})}$$

We have $N\rightarrow\infty$ and a stable system where the magnitude or our factor is smaller than 1 (something we need for convergence, see above), and we get:

$$\sum_{n=0}^{\infty}(0.5\cdot z^{-1})^n=\frac{1}{1-(0.5\cdot z^{-1})}$$

The Short-Time Fourier Transform (STFT)

The STFT is basically a DFT applied to short blocks of the signal. For that, the signal is first divided into overlapping blocks of length N and with “hop-size” M.

The **analysis equation** is

$$Y_k(m)=\sum_{n=0}^{N-1} h(n)\cdot x(m\cdot M+n)e^{-j\frac{2\pi}{N}\cdot k\cdot n}$$

with m=0 until the end of x again the block index, and $h(n)$ is a **window function** of length N for improved filtering properties. Assume N=LM.

For the synthesis the overlapped blocks are added up (overlap-add). The **synthesis equation** is

$$\hat{x}_m(n)=\frac{h(n)}{N}\cdot\sum_{k=0}^{N-1} Y_k(m)\,e^{j\frac{2\pi}{N}\cdot k\cdot n},\; n=0,\ldots,N-1$$

overlap-add:

$$x(m_0\cdot M+n)=\sum_{m=0}^{L-1} \hat{x}_{m_0-m}(mM+n)$$

n=0,...,M-1.

The window has the **overlap-add property**

$$\sum_{m=0}^{L-1} h^2(n+mM)=1,\; n=0,\ldots,M-1$$

The STFT is also a **filter bank**, but with **non-critical sampling**, since usually M<N.

In the literature the **window** function h(n) is usually only applied to the analysis part. Here we also applied it to the **synthesis** part, because this improves the resulting synthesis filters, and is more similar to usual filter banks.

Translations to Python Code

We can translate these formulas into Python code:
In a shell we start Python by typing "python", to get into interactive mode.
Then:

```
#simple example for signal sequence x and
filter impulse response h:
x=[1,2,3,4]
h=[1,2]


#z-Transform, turn sequence into
# polynomial, in z-transf. Order:
```
$X(z)=\sum_{n=-\infty}^{\infty} x(n) z^{-n}$

```
import sympy
import numpy as np
z=sympy.symbols('z')
xz=sympy.Poly(np.flipud(x), z**(-1))
print(xz)
```

```
>>>Poly(4*1/z**3 + 3*1/z**2 + 2*1/z + 1,
1/z, domain='ZZ')
```

$H(z)=\sum_{n=-\infty}^{\infty} h(n)z^{-n}$

```
hz=sympy.Poly(np.flipud(h), z**(-1))
print(hz)
>>>Poly(2*1/z + 1, 1/z, domain='ZZ')
```

#**Multiplication in the z-Domain:**

$Y(z)=X(z)\cdot H(z)$

```
yz=xz*hz
print(yz)
>>>Poly(8*1/z**4 + 10*1/z**3 + 7*1/z**2 +
4*1/z + 1, 1/z, domain='ZZ')
```

#**Inverse z-transform**, turn polynomial into coefficients, in z-transf. Order (works for Finite Impulse Response systems):

$y(n)=\frac{1}{2\pi j}\oint_c Y(z)z^{n-1}dz$

```
y=np.flipud(yz.coeffs())
print(y)
>>>array([1, 4, 7, 10, 8], dtype=object)
```

#**Convolution in time domain:**

$x(n)*h(n)$

```
import numpy as np
np.convolve(x,h)
array([ 1,  4,  7, 10,  8])
```

#**Observe: samples after convolution are identical to polynomial coefficients after z-Transform above.**

```
#But convolve is usually faster than symbolic calculation.
```

#**Convolution as sum:**

$$y(n)=x(n)*h(n)=\sum_{l=0}^{L-1} x(n-l)\cdot h(l)$$

```
Nx=len(x)
Nh=len(h)
#convolution length:
Deg=Nx+Nh-1
y=np.zeros(Deg)
#loop over convolution index:
for n in range(0,(Deg)):
      #loop over sum index:
      for l in range(0,n+1):
          #only over parts where x and y
```

```
                  #are defined:
             if ((n-l)<Nx and l<Nh):
                 y[n] = y[n]+ x[(n-l)]*h[l];
print(y)
>>>array([  1.,   4.,   7.,  10.,   8.])
```

#Observe: We get again the same result as above, as expected.