Multirate Signal Processing
Lecture 7,
Sampling

Gerald Schuller, TU Ilmenau

(Also see: Lecture ADSP, Slides 06)

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.

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 a **multiplication** with the sampling signal,

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

Observe that this is similar to **modulation**, but instead of a sinusoidal signal we use the sampling signal. For modulation we saw that it shifts the signal spectrum to the frequency of the sinusoidal signal.
Here the **signal is shifted** to the **frequency components of the sampling signal**. It can be shown that the frequency components of the sampling signal consist

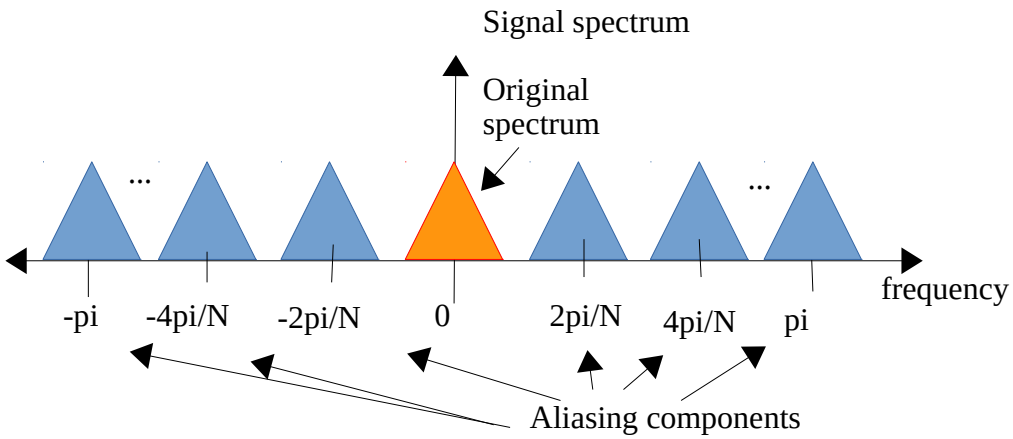
of a **DC** component, the fundamental frequency of **1/N**, and **harmonics** at integer multiples of its fundamental frequency, and all these components have the **same strength**.

This $x^d(n)$ signal is now an intermediate signal, which gets the zeros removed before transmission or storage in an **encoder**, 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?

The derivation can be found in lecture ADSP, slides 06, sampling.
It 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**, at the above mentioned frequency components of the sampling signal,

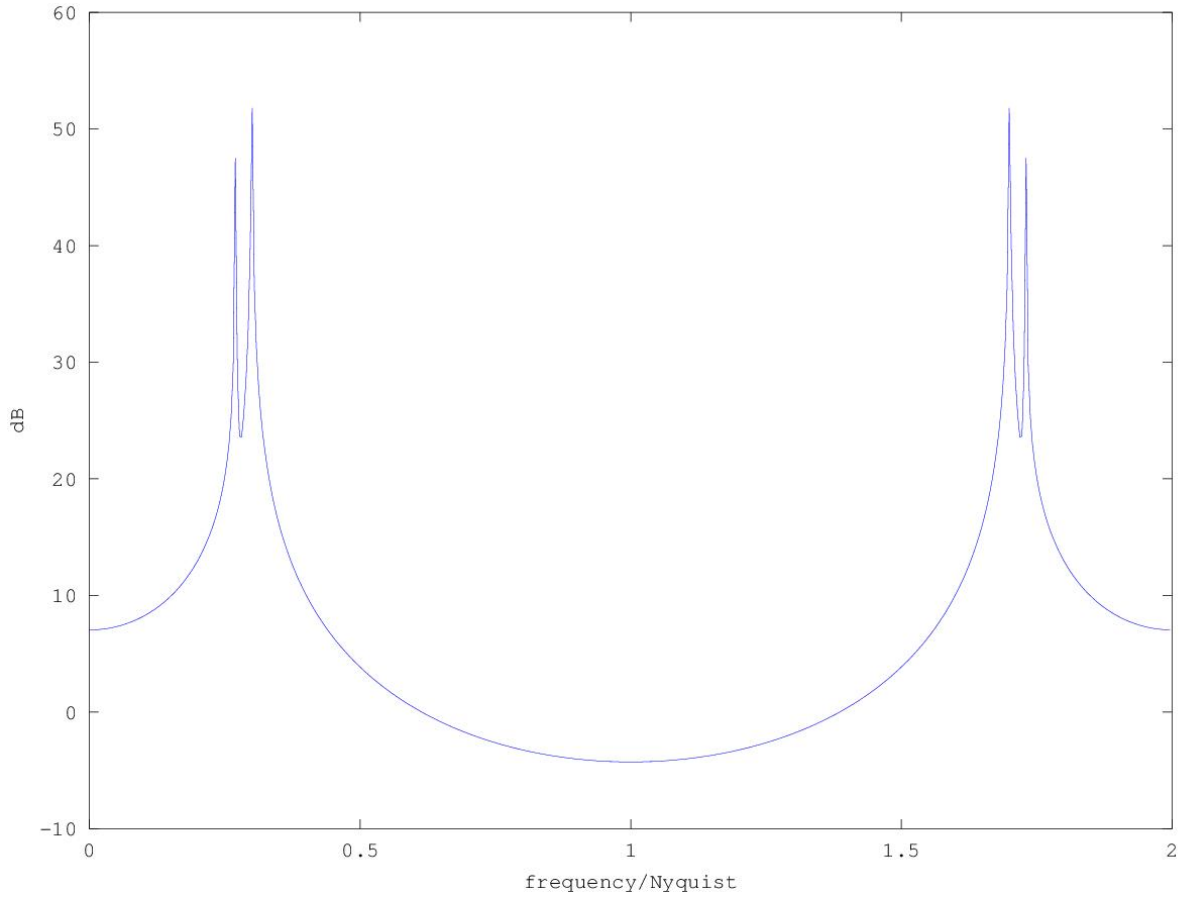
$$X^d(\Omega)=\frac{1}{N}\sum_{k=0}^{N-1} X\left(\frac{-2\pi}{N}\cdot k+\Omega\right)$$ (eq.1)



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.

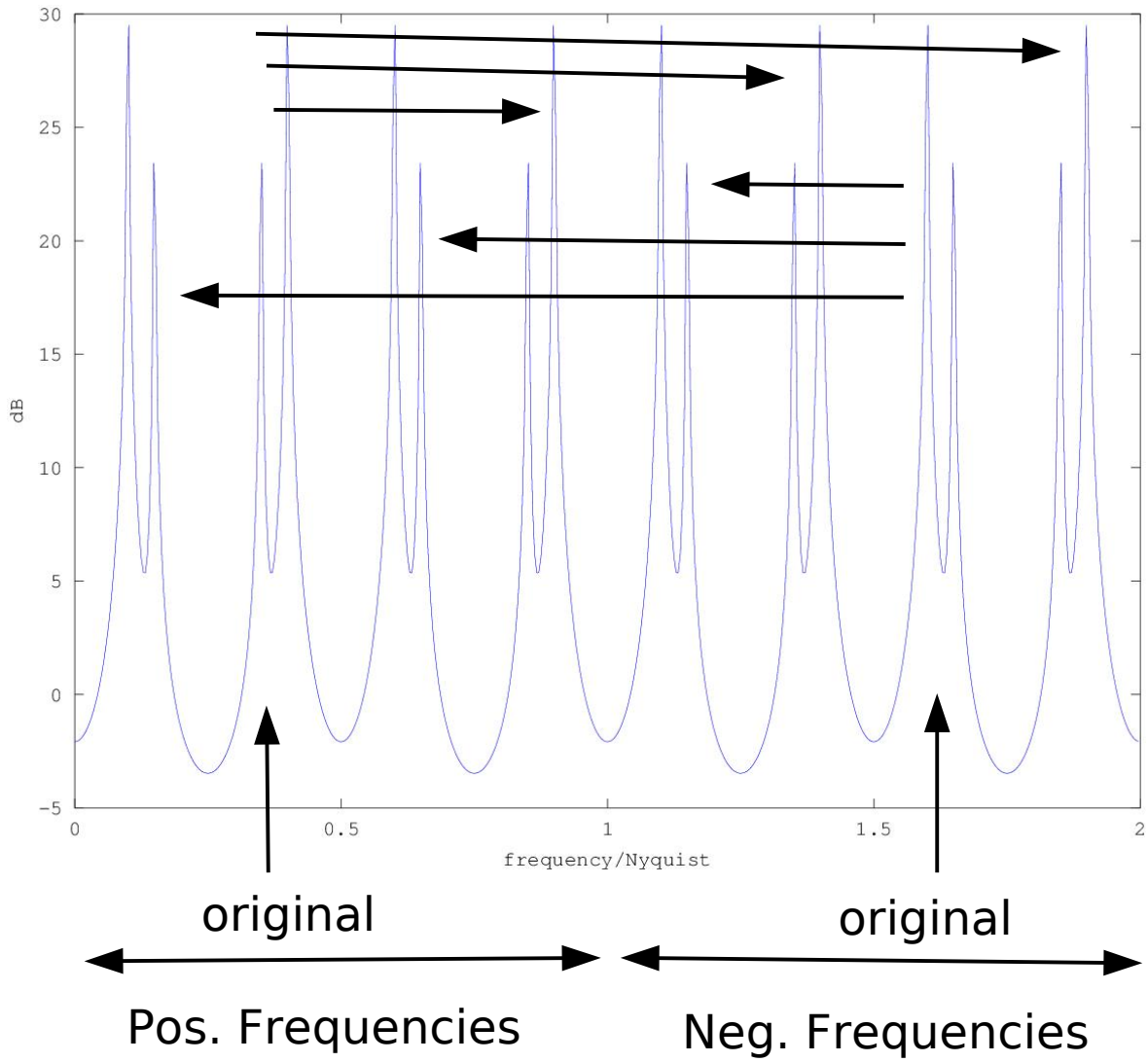
Next is another example, also including the negative frequencies that now show up above normalized frequency 1 (1 being the

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



original

original

Pos. Frequencies

Neg. Frequencies

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 Python as:

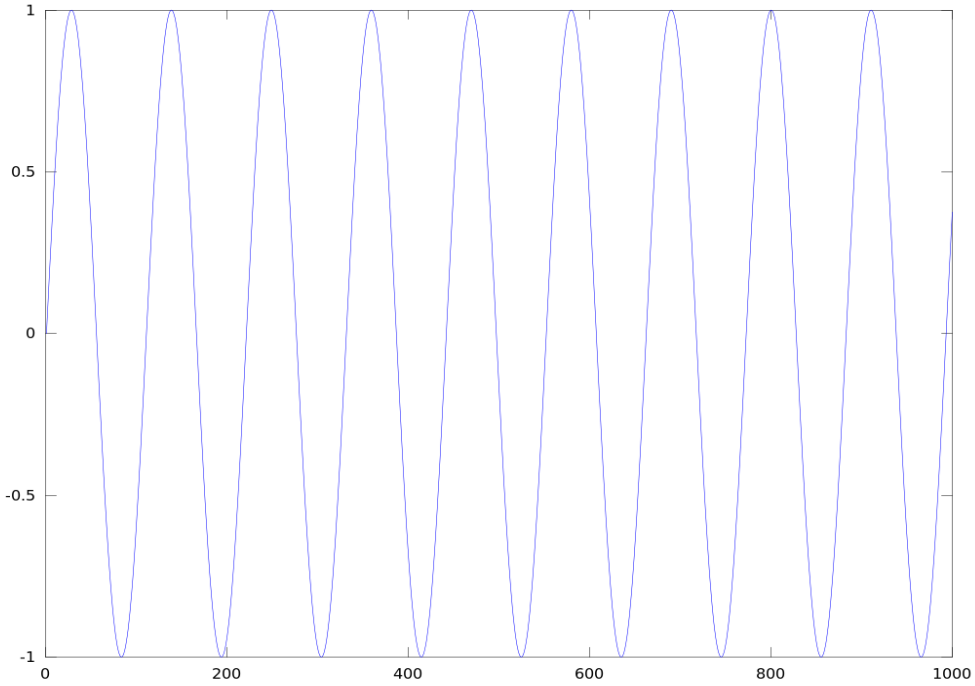
```
ipython --pylab
fs = 44100
f = 400.0
s=sin(2*pi*f*arange(0,1,1.0/fs))
```

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

```
from sound import sound
sound((2**15)*s,fs)
```

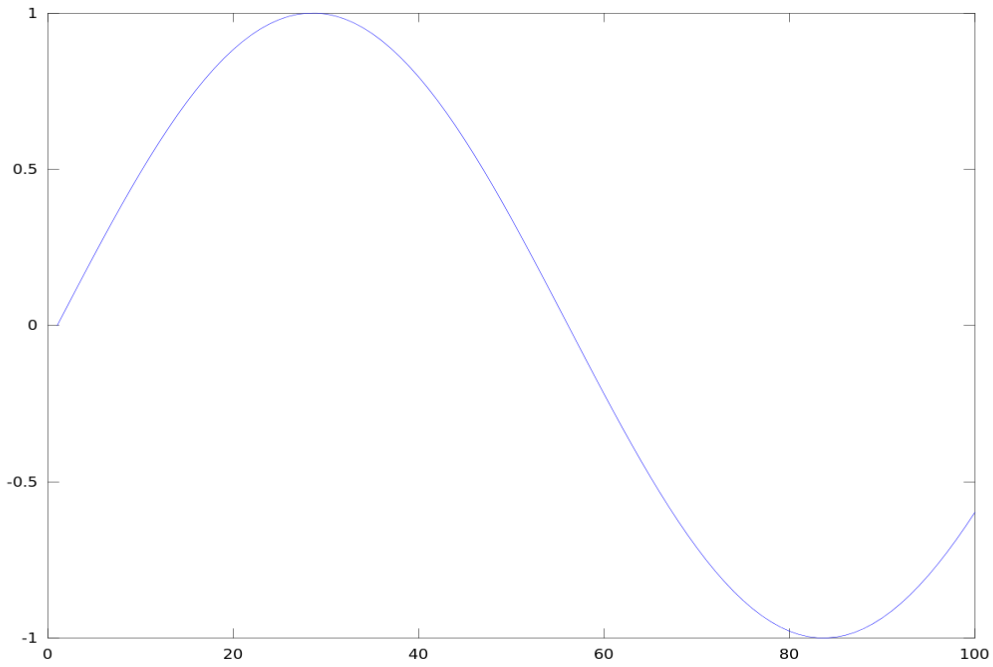
Now plot the first 1000 samples:

```
plot(s[0:1000])
```



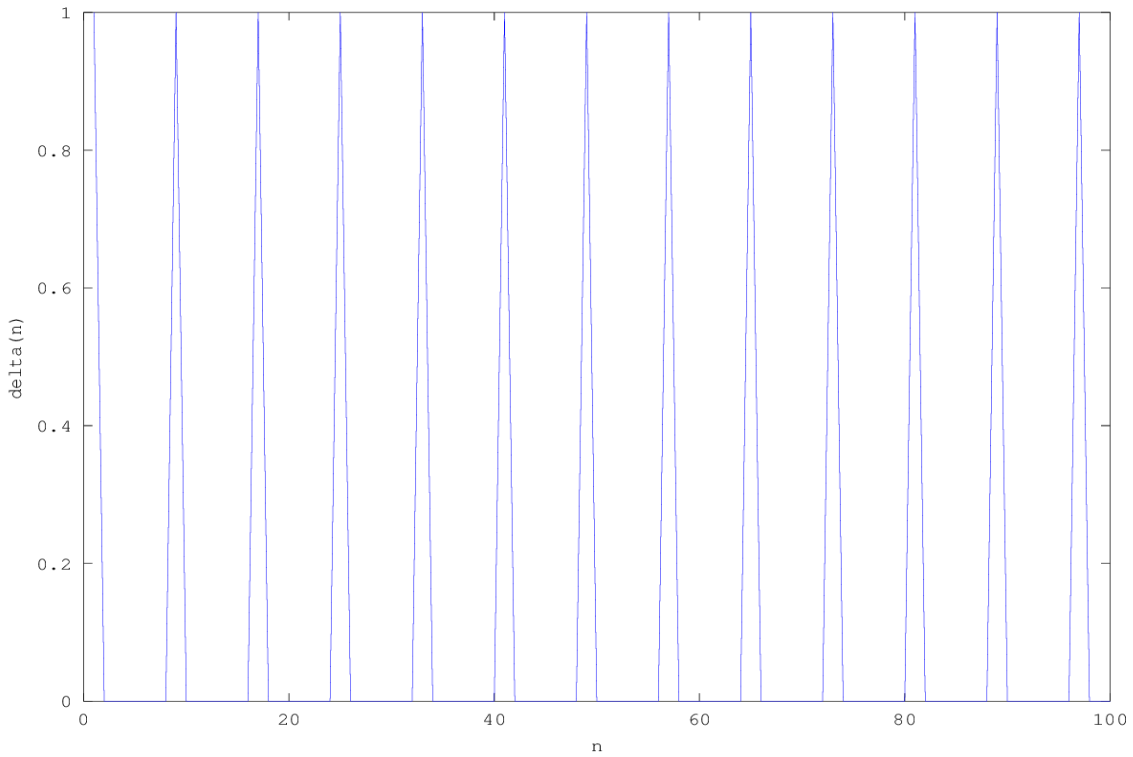
Next plot the first 100 samples:

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



Now we can multiply this sine tone signal with a unit pulse train, with N=8.
We generate the unit impulse train,

```
unit = zeros(44100)
unit[0::8] = 1
plt.plot(unit[0:100])
plt.xlabel('n')
plt.ylabel('unit(n)')
```



Listen to it, with scaling to the value range for 16 bit/sample:

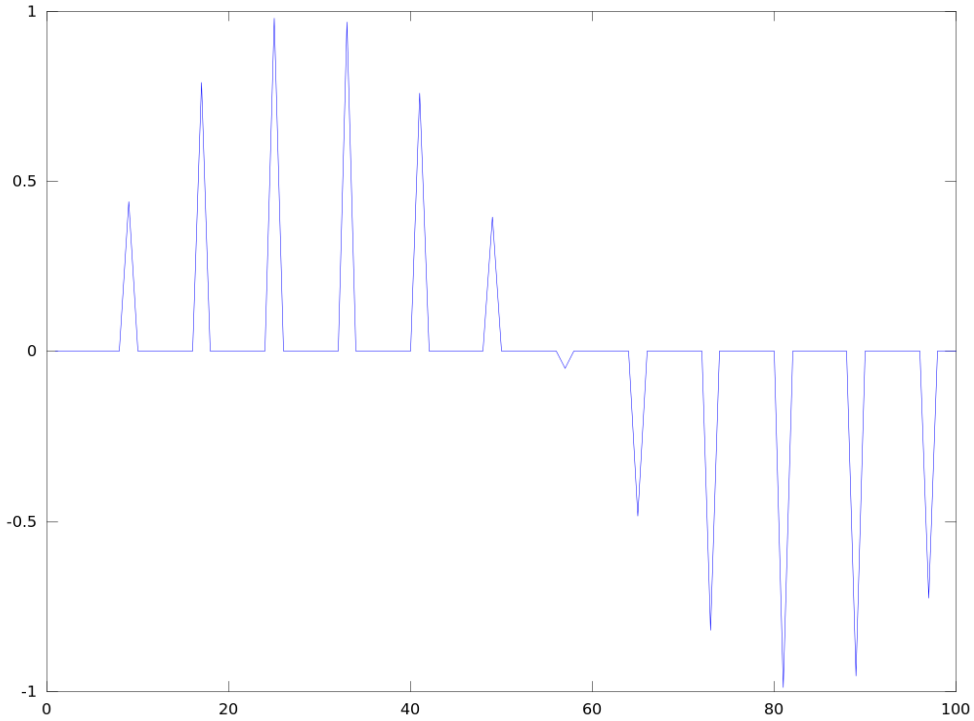
```
from sound import sound
sound(unit*2.0**15,44100)
```

The multiplication with the unit impulse train:

```
sdu=s*unit
```

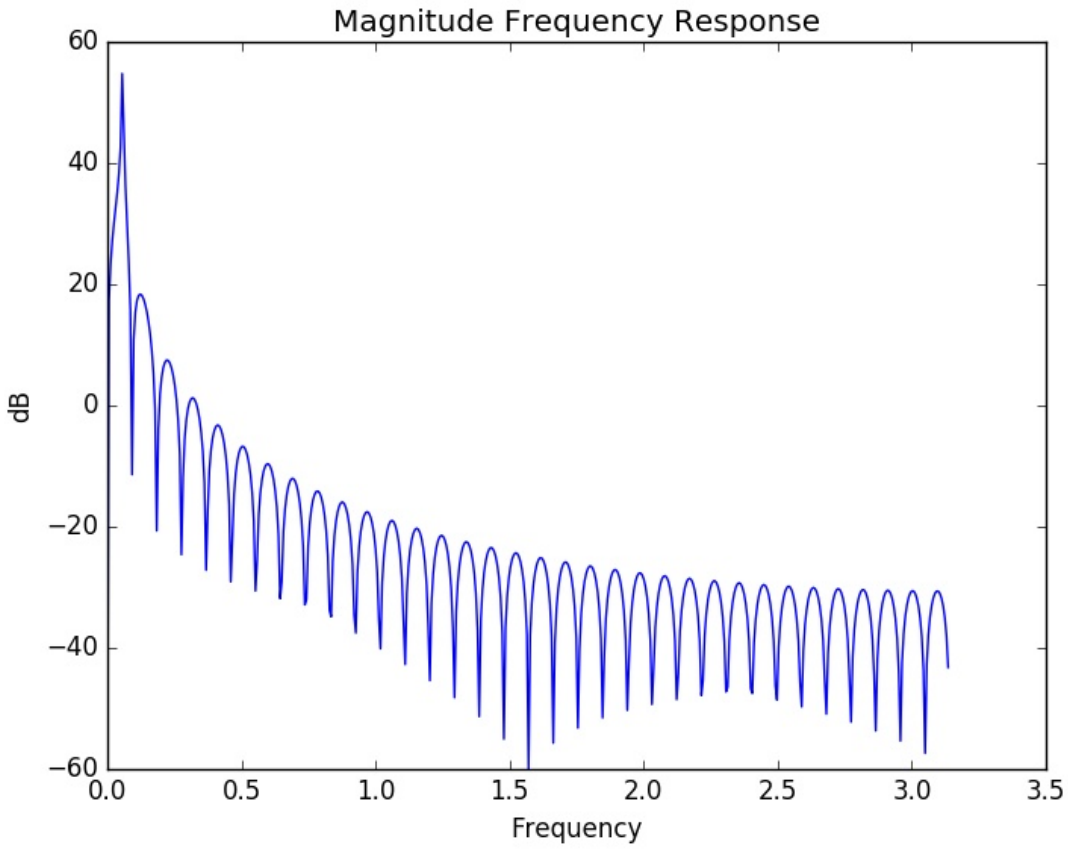
(This multiplication is also called „frequency mixing“).
Now plot the result, the first 100 samples:

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



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

```
from scipy.signal import freqz
w,H=freqz(s)
plot(w, 20*log10(abs(H)+1e-3))
xlabel('Frequency')
ylabel('dB')
title('Magnitude Frequency Response')
```

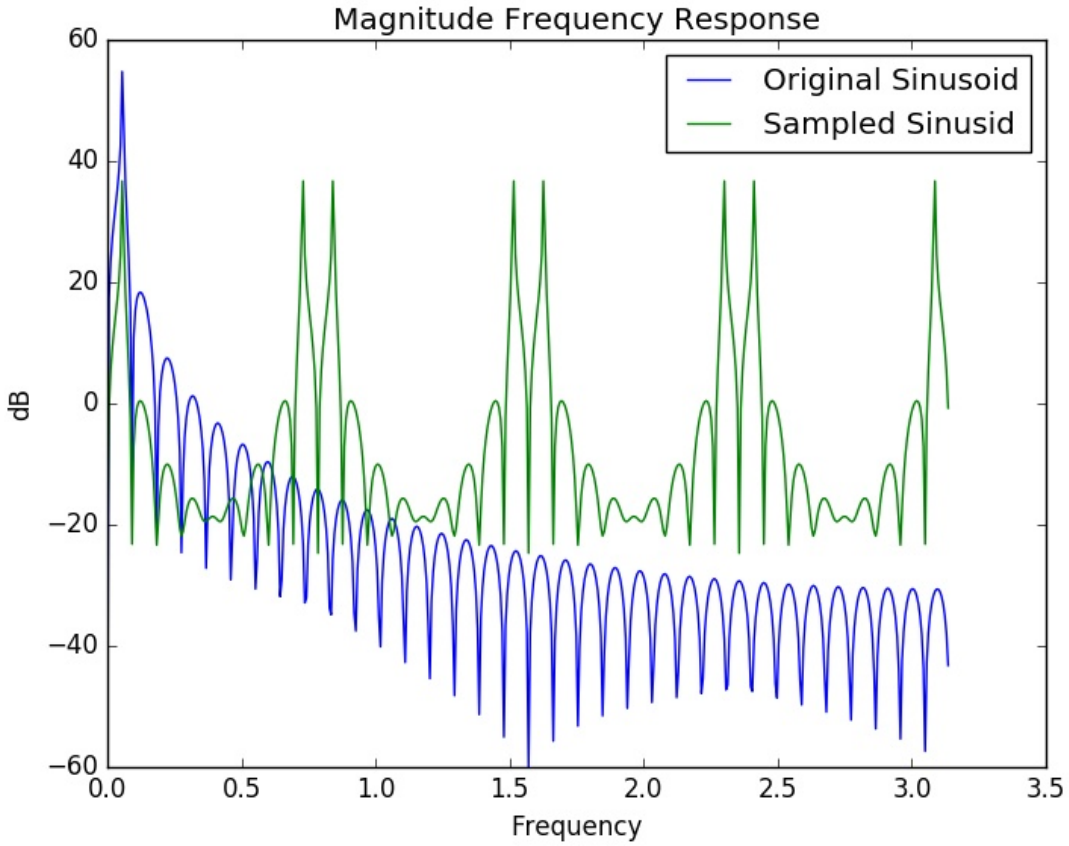


The plot shows the magnitude of the frequency spectrum of our signal. Observe that the frequency axis (horizontal) is a normalized frequency, normalized to the Nyquist frequency as π, in our case 22050 Hz. Hence our sinusoid should appear as a peak at normalized frequency 400.0/22050*pi=0.05699, which we indeed see.

Now we can compare this to our signal with the zeros, sdu:

```
w,H=freqz(sdu)
plot(w, 20*log10(abs(H)+1e-3))
```

```
legend(('Original Sinusoid','Sampled Sinusid'))
```



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 -0.05699.

Now also listen to the signal with the zeros:

```
sound(2**15*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

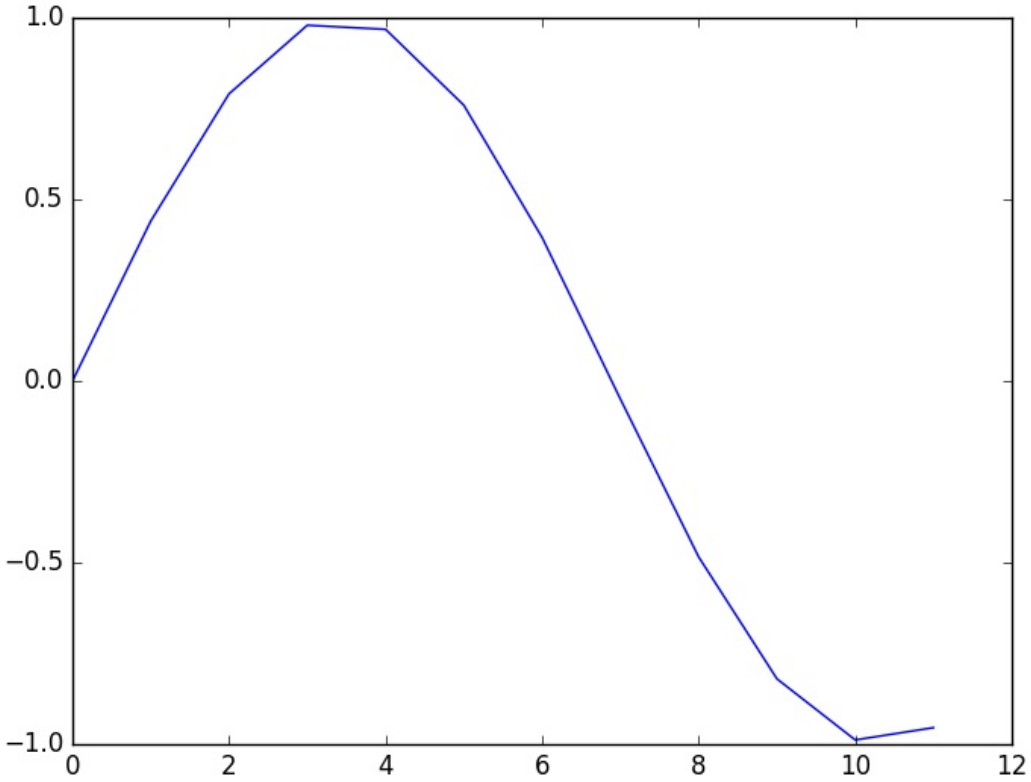
The final step of downsampling is now to omit the zeros between the samples, to obtain the lower sampling rate. Let's call the signal without the zeros

$$y(m),$$

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

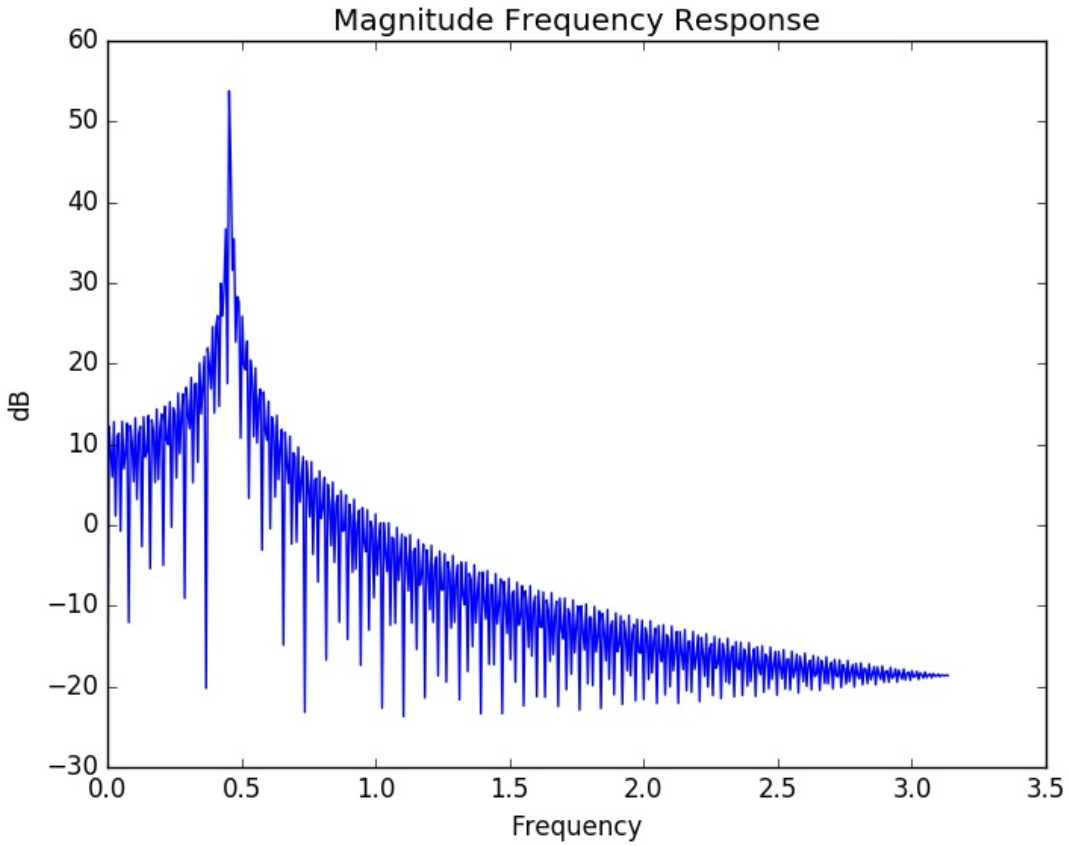
In our Python example this is:

```
sd = sdu[0:44100:8]
plot(sd[0:(100/8)])
```



We can now take a look at the spectrum with

```
w,H=freqz(sd)
plot(w, 20*log10(abs(H)+1e-3))
xlabel('Frequency')
ylabel('dB')
title('Magnitude Frequency Response')
```



Observe that the sine signal now appear at normalized frequency of 0.455, 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 $400*3.14/22050\cdot 8\approx 0.455$. 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 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 information 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 also be seen in following pictures,

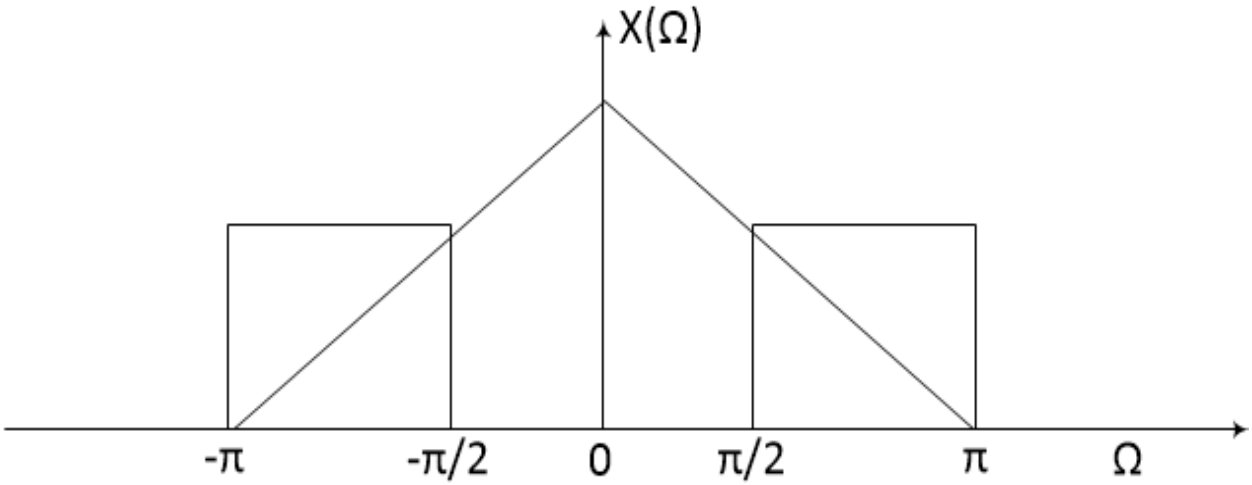


Figure 1: The magnitude spectrum of a signal. The 2 boxes symbolize the passband of an ideal bandpass, here a high pass.

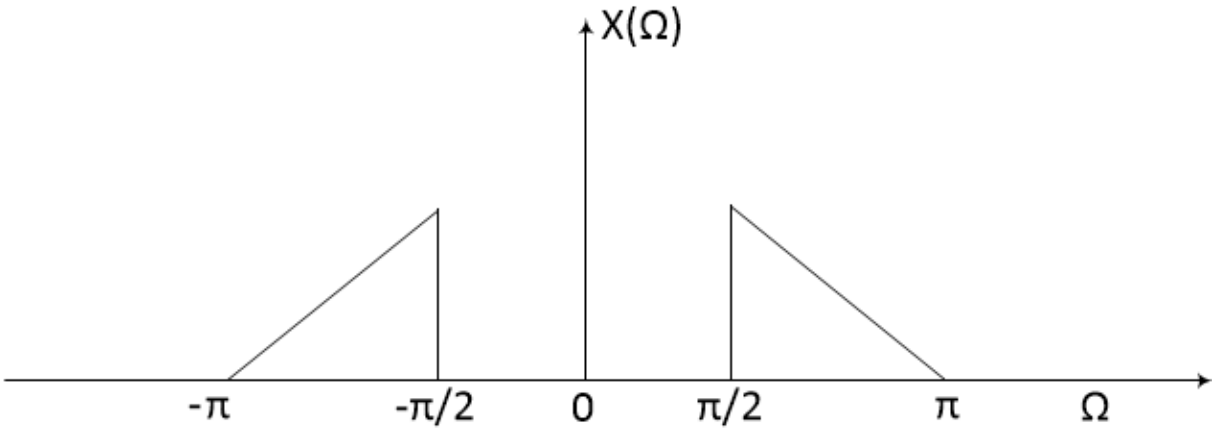


Figure: The signal spectrum after passing through the high pass.

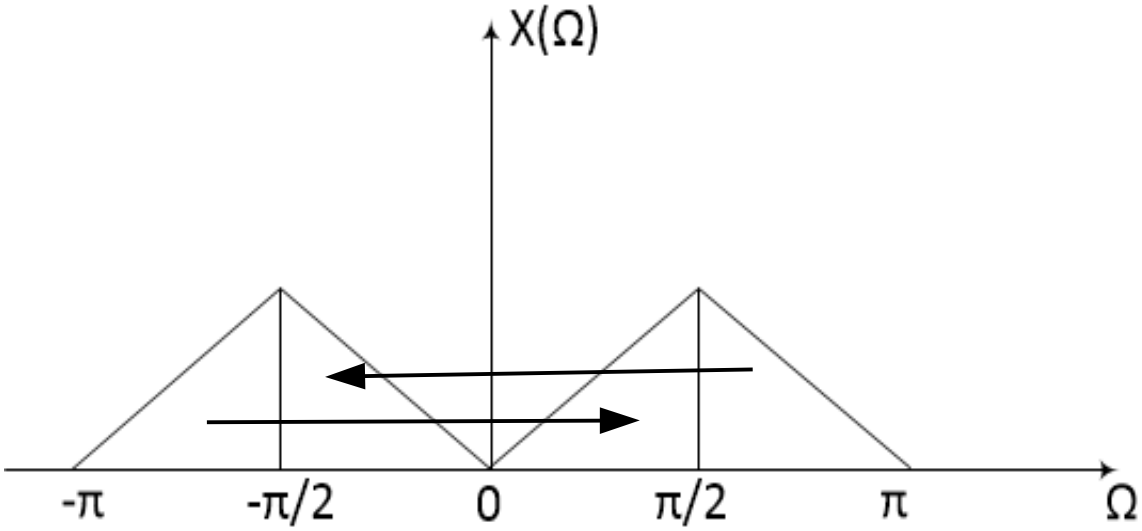


Figure: Signal spectrum after multiplication with the unit pulse train, for N=2, hence setting every second value to zero (the zeros still in the sequence). Observe the we shift and add the signal by multiples of $2\pi/2=\pi$, and in effect we obtain „mirrored“ images of the high frequencies to the low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

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

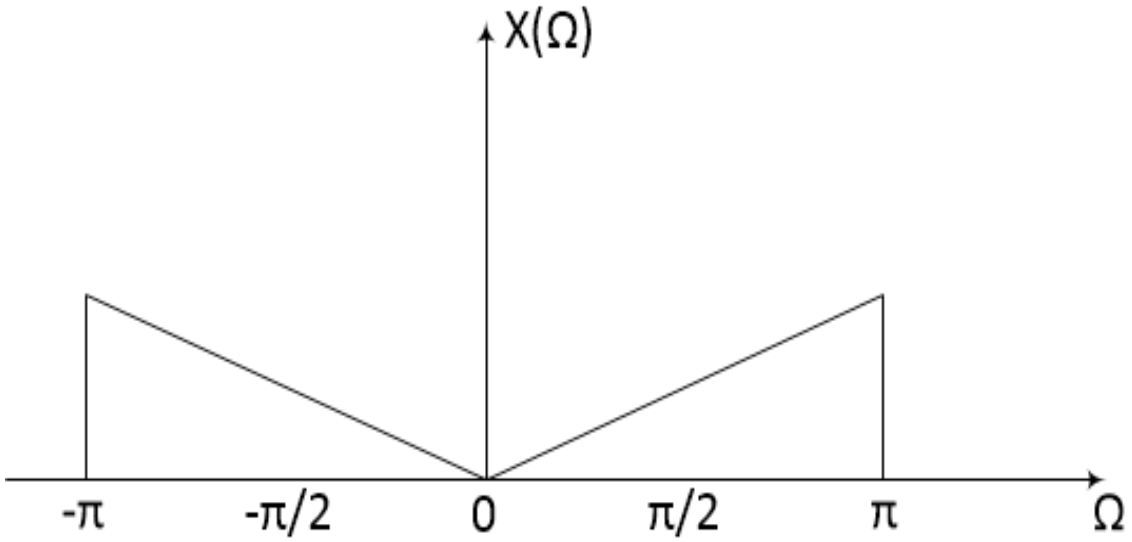


Figure: Signal spectrum after downsampling (removing the zeros) by N (2 in this example). Observe the stretching of the spectrum by a factor of 2.

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

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

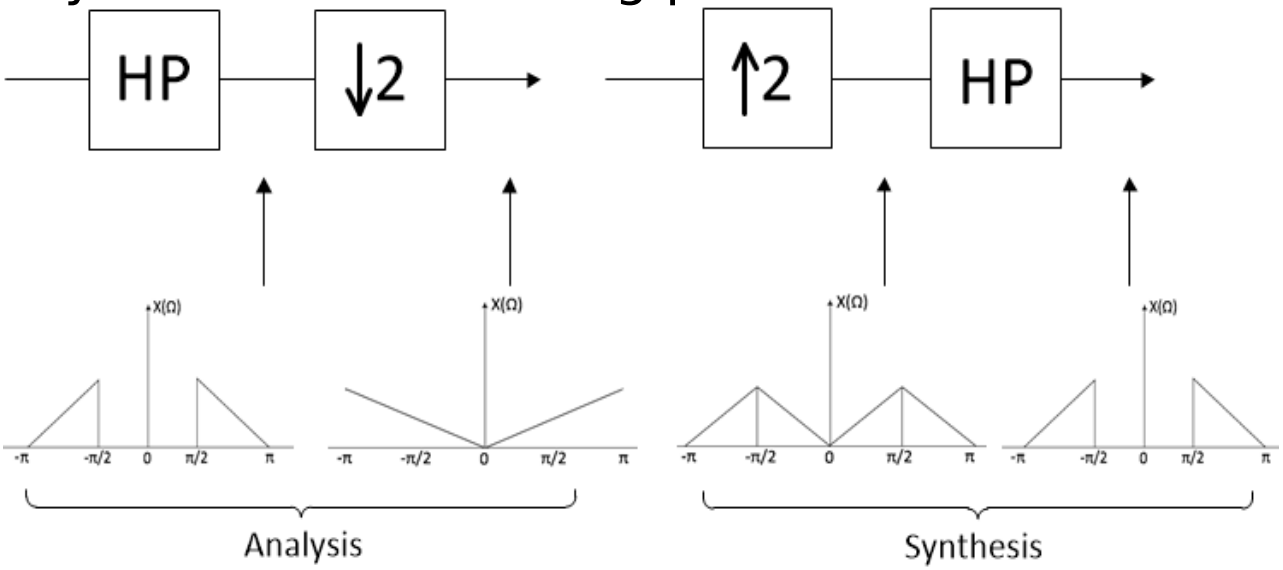
We are now just coming from y(m), going to the now upsampled signal $x^d(n)$.
For instance if we had the frequency π before upsampling, it becomes $\pi/2$ for the upsampled signal, if we have N=2. In this way we now get an „extended“ frequency range.

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

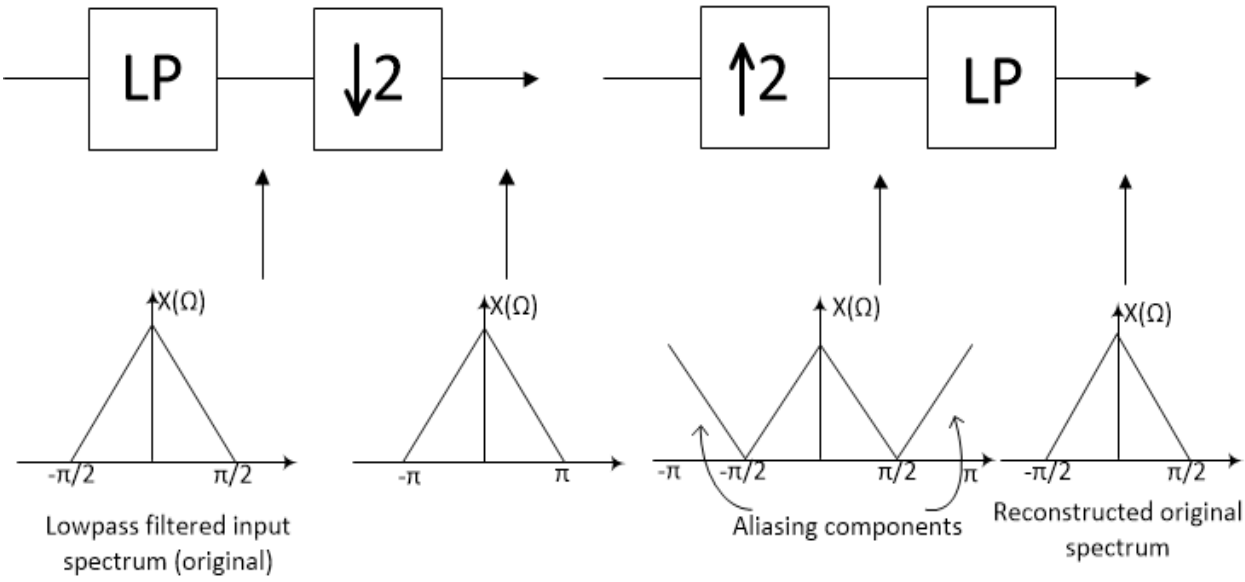


Observe that we violate the conventional Nyquist criterium, because our high pass passes the high frequencies. But then the sampling mirrors those frequencies to the lower range, such that we can apply the traditional Nyquist sampling theorem. This method is also known as bandpass Nyquist. This is an 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).

In genereal this is true for **complex** filters. For **rea**l valued filters observe that this simple assumption only works 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!
For proof and details see ADSP, lecture 06, sampling.

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:



Python Example

This program can be **controlled by keyboard**, and features **sampling** by a factor of N=8 (with keeping the zeros), which can be turned on and off, and a **low pass filter** before and after sampling to supress the alias components, which can be turned on and off.
It shows the magnitude of the resulting DFT spectrum after reconstruction (after the last low pass filter) in a so-called **waterfall spectogram:**
-The **magnitude** of the DFT is displayed as **color** (yellow is high, blue is low),
-horizontal axis is **frequency** (0 to Nyquist frequency)
-vertical axis is **time** (up is past).

Start it with

```
python pyrecspecwaterfallsampling.py
```

Observe: When we turn on **sampling**, we clearly **hear** the **aliasing artifacts**, and we also **see** them as spectral copies in the waterfall spectrogram.
When we turn on the **low pass filter**, most of the aliasing artifacts are **not audible** anymore, and we also **don't see** most of them anymore in the waterfall spectrogramm, but it also sounds **more muffled**.

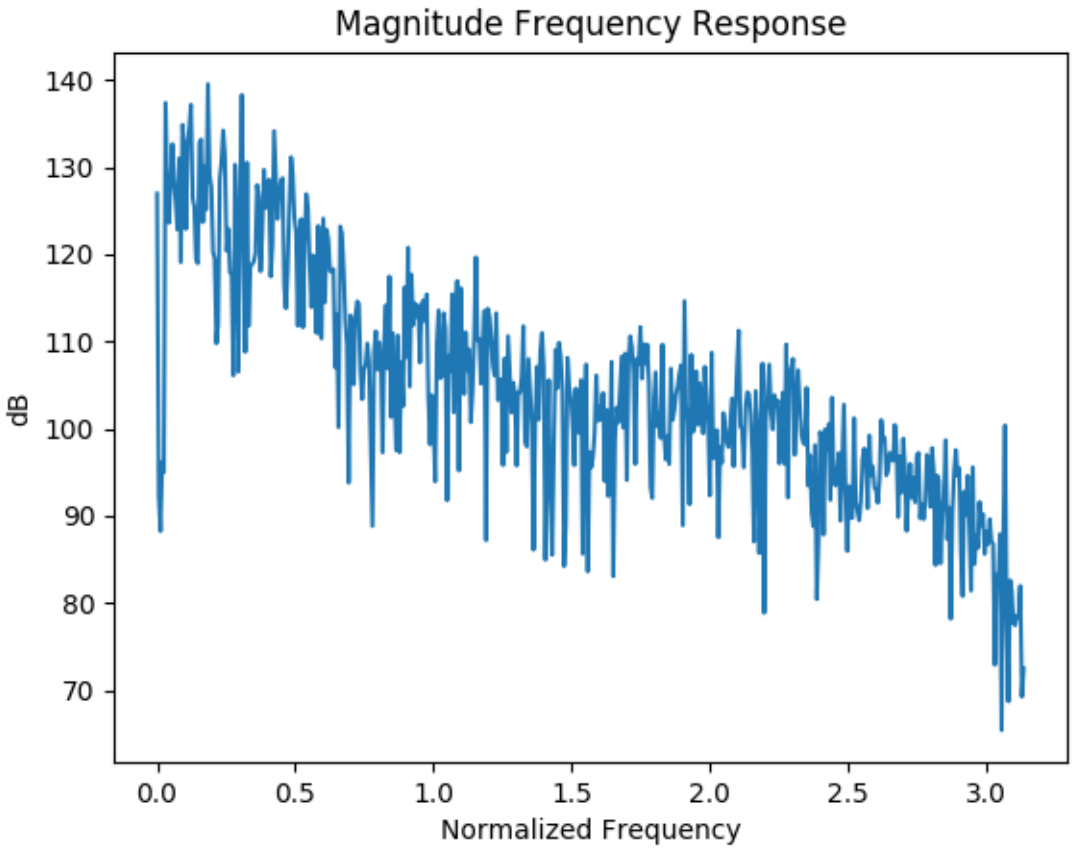
Python Example:

Just take an audio signal and read it into Python, and filter it with our previous low pass filter (e.g. Kaiser Window with Beta=8),

```
ipython –pylab
from scipy.io import wavfile
import scipy.signal as sig
from sound import sound

fs, x =
wavfile.read('04_topchart.wav');
%listen to it:
sound(x,fs)
%Look at its spectrum:
```

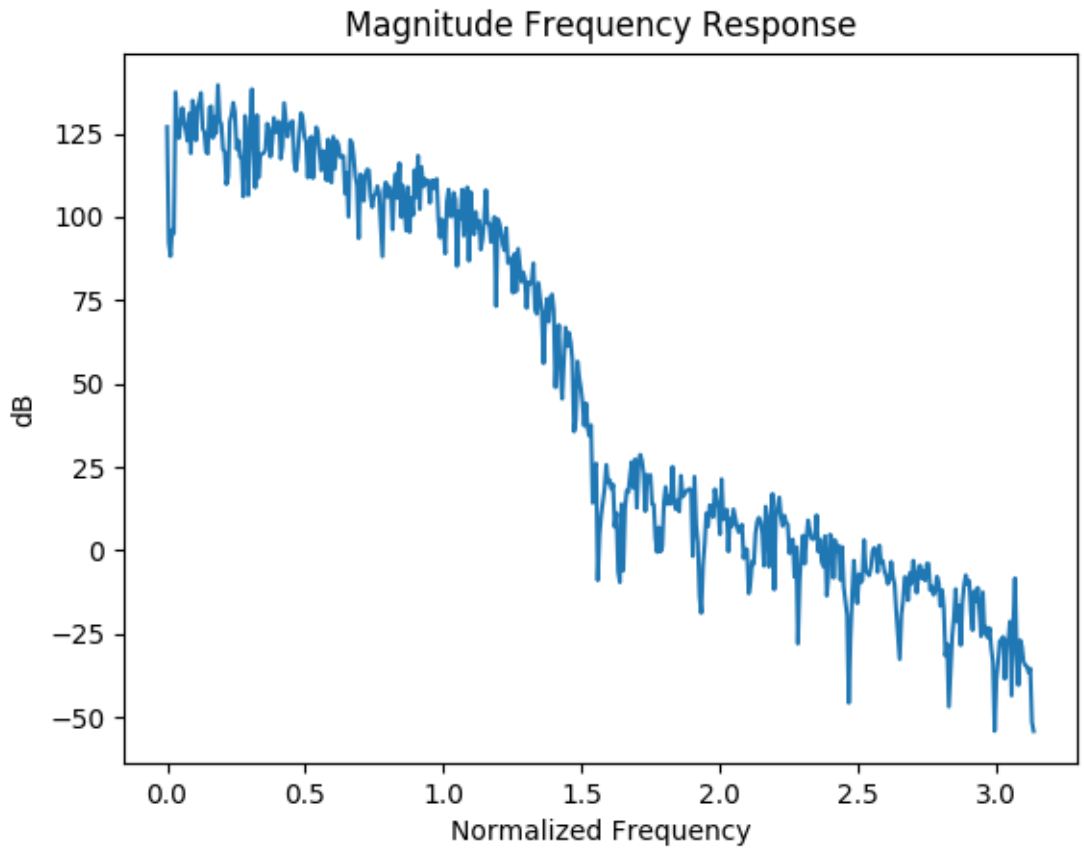
```
w,H=sig.freqz(x)
plot(w,20*log10(abs(H)))
xlabel('Normalized Frequency')
ylabel('dB')
title('Magnitude Frequency Response')
```



This is the spectrum of our original. Observe its already low pass characteristic.
Now we can low pass filter it. We take our Kaiser Window low pass filter design from slides Nr. 6,

```
n = arange(32)
w_c = 0.33*pi
n_d = (len(n)-1)/2.0
#ideal impulse response:
h = sin(w_c*(n-n_d))/(pi*(n-n_d))
#Kaiser window:
hk = np.kaiser(32,8)
#multiply ideal filter and Kaiser
window:
hfilt = hk*h
xlp=convolve(x,hfilt)

w,H=sig.freqz(xlp)
plot(w,20*log10(abs(H)))
xlabel('Normalized Frequency')
ylabel('dB')
title('Magnitude Frequency Response')
```



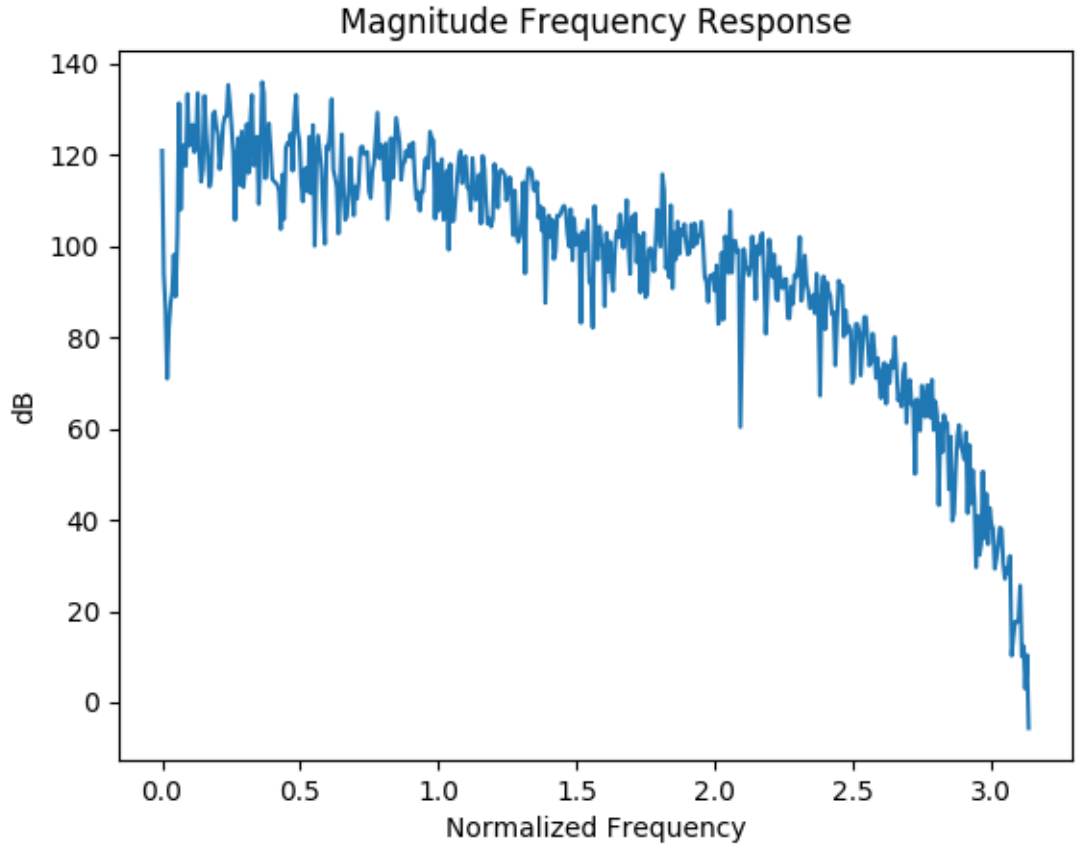
Observe that beginning at frequency 1.5 we have indeed much attenuation and hardly any signal left.
Listen to the low pass version:

```
sound(xlp,fs);
```

Now we can down-sample it by a factor of N=2, including the removal of the zeros,

```
xds = xlp[::2]
w,H=sig.freqz(xds)
plot(w,20*log10(abs(H)))
xlabel('Normalized Frequency')
ylabel('dB')
```

```
title('Magnitude Frequency Response')
```



Observe that we now obtain the “streched” spectrum.
Listen to it:

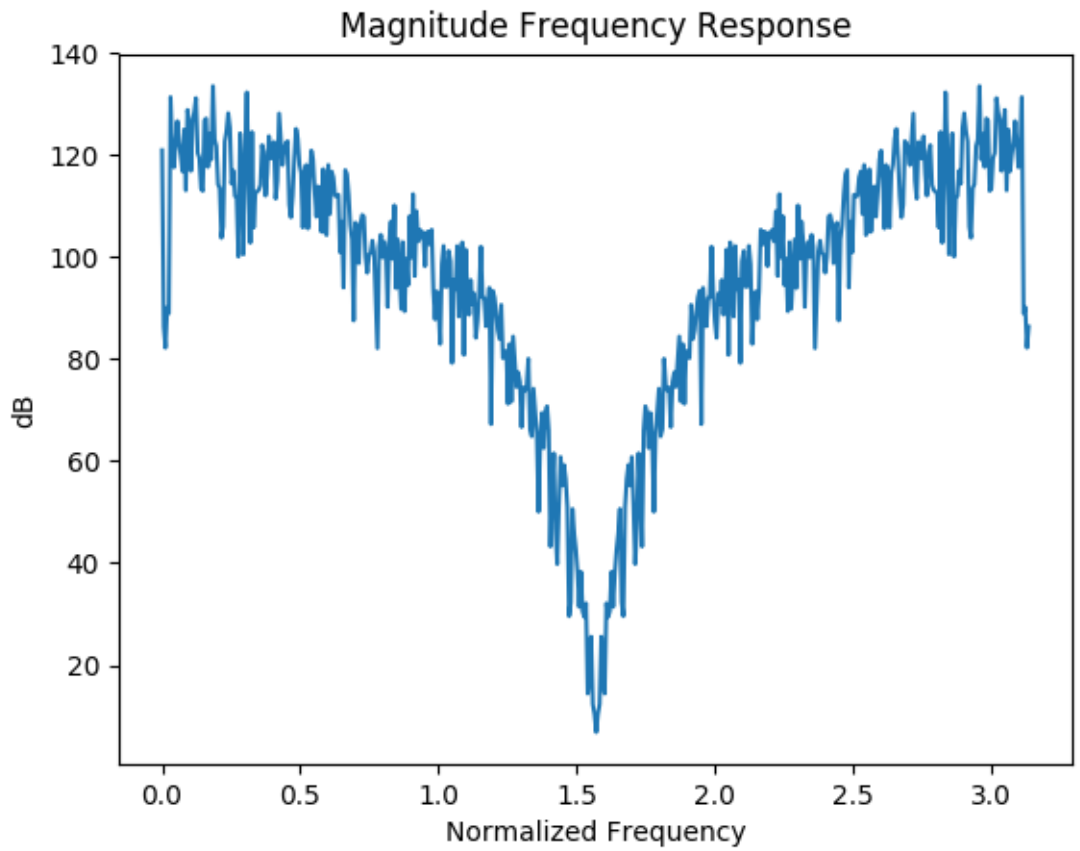
```
sound(xds,fs/2)
```

It should sound more “muffled”, but otherwise the same, but at now half the sampling rate!

Now we can upsample again, using the same indexing trick, now just on the receiving side, effectively inserting a zero after each sample. Observe that in this way

we can avoid the function “downsample” or “upsample”, which makes it clearer to see and check what happens,

```
xups = np.zeros(2*max(xds.shape))
xups[::2] = xds
w,H=sig.freqz(xups)
plot(w,20*log10(abs(H)))
xlabel('Normalized Frequency')
ylabel('dB')
title('Magnitude Frequency Response')
```

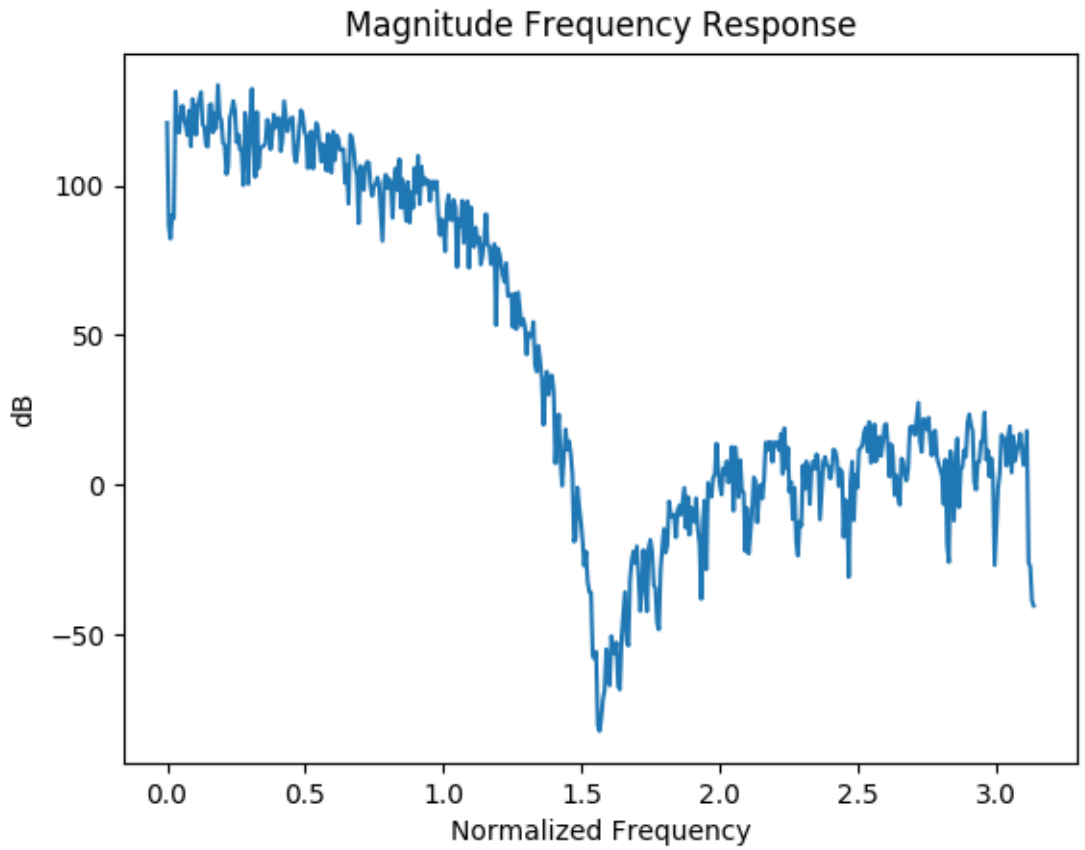


Observe the “shrinking” and periodic continuation of the spectrum, the aliasing component at the high frequencies.
Listen to the signal including aliasing,

```
sound(xups,fs);
```

Now low pass filter the result,

```
xupslp = np.convolve(xups,hfilt)
w,H=sig.freqz(xupslp)
plot(w,20*log10(abs(H)))
xlabel('Normalized Frequency')
ylabel('dB')
title('Magnitude Frequency Response')
```



Observe that now we removed the aliasing component at high frequencies.
Now listen to it,

```
sound(xupslp,32000);
```

Observe: it should now sound the same as at the lower sampling rate, but now at the higher sampling rate of 32 kHz! (Possible differences are due to not sufficiently attenuated aliasing).