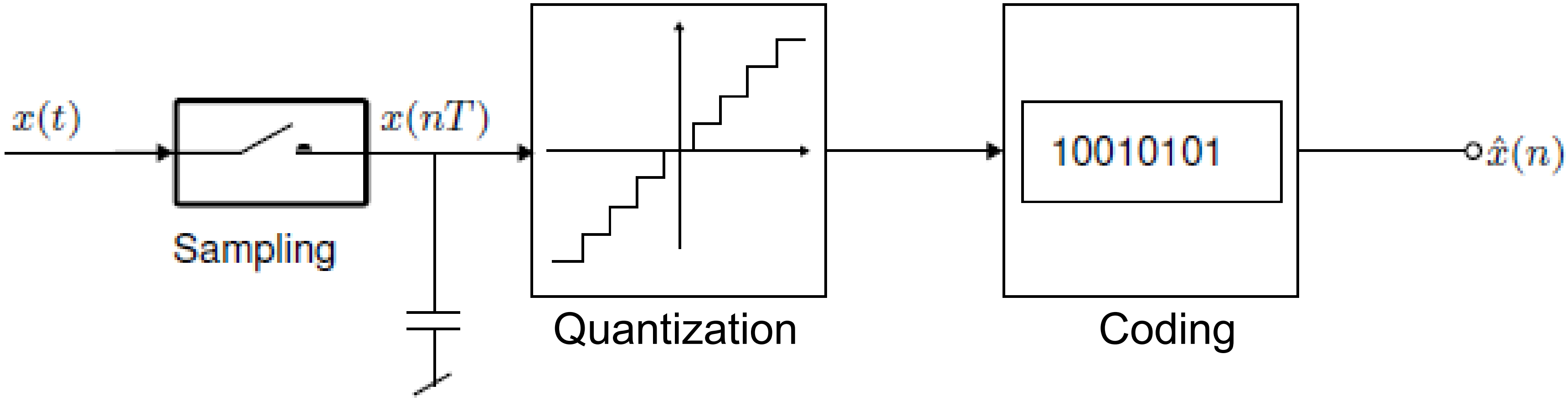
Implementation

Zhuoqi Cheng
zch@mmmi.sdu.dk
SDU Robotics

SDU

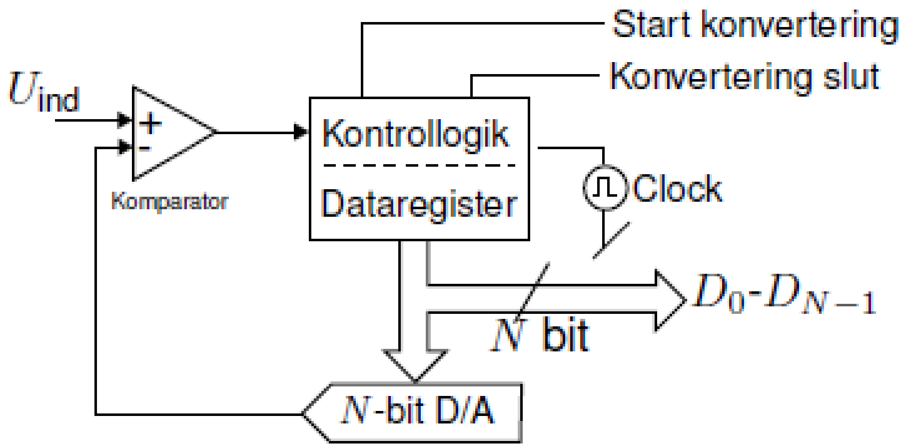
Quantization

In analog-digital conversion, the input signal $x(t)$ is transformed into a sequence with finite resolution as shown below.



Analog-Digital converter principle

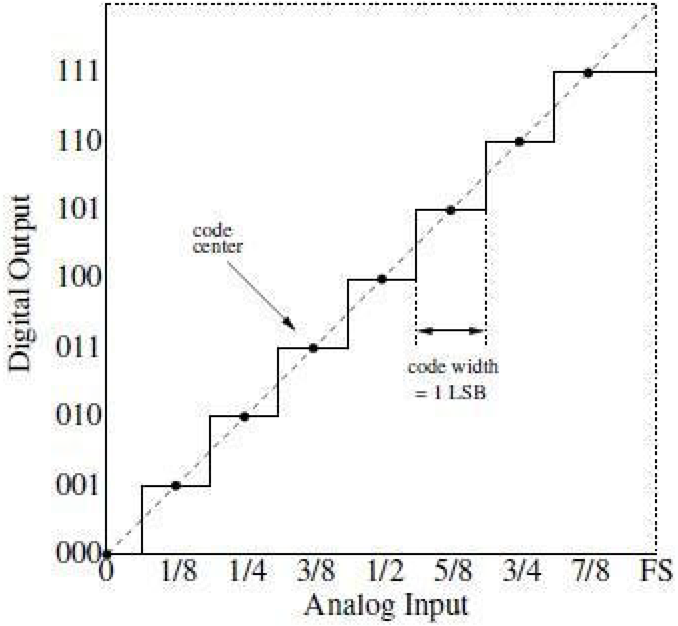
A Successive Approximation A/D converter is built according to the principle shown below, where the output of the A/D converter is compared with the input sequence through a comparator.

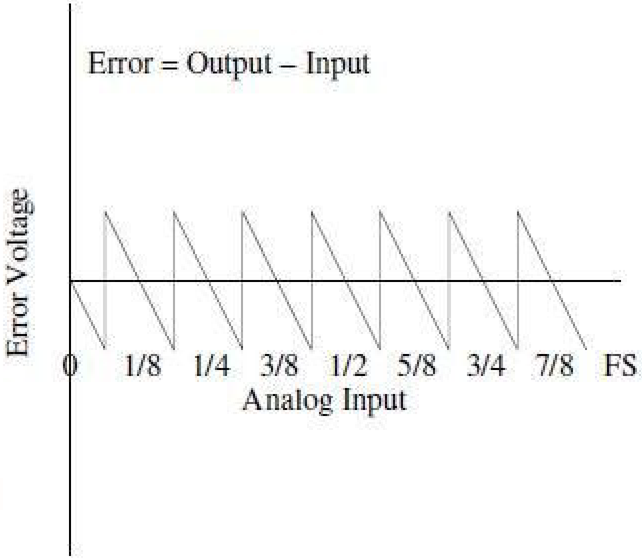


If the conversion is to take place faster (in the MHz range), then a Flash A/D converter is used.

Quantization

During A/D conversion, the input signal is quantized so that it has a word length of N bits, i.e. the representation has 2^N different quantization levels.





SDU

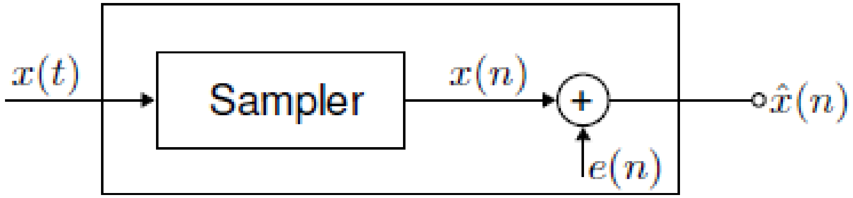
Quantization error

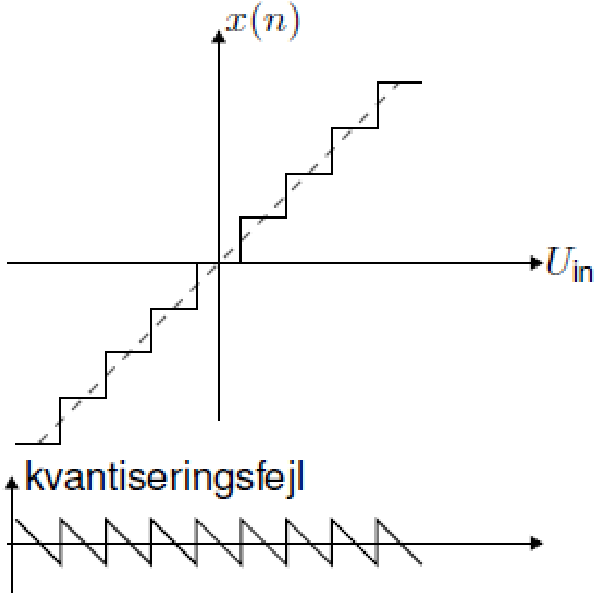
The quantization error means that the value of the input sequence $x(n)$ and the quantized input sequence $\hat{x}(n)$ are different

$$\hat{x}(n) = x(n) + e(n)$$

where $e(n)$ is the quantization error.

The quantization error can be thought of as a noise sequence as shown below.

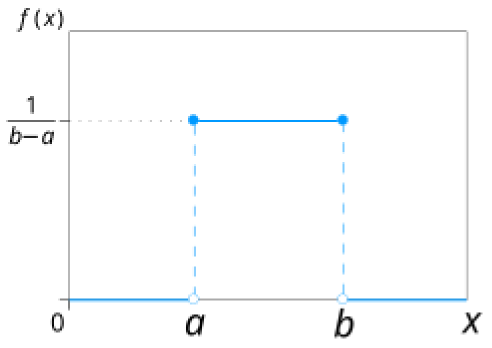




The magnitude of the quantization error is bounded as

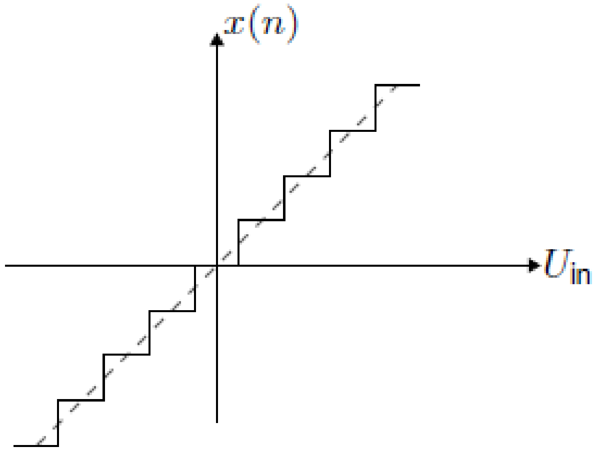
$$-\frac{\Delta V}{2} \leq e_q \leq \frac{\Delta V}{2}$$

Quantization error



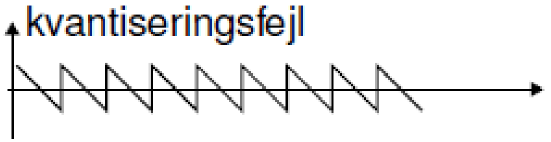
The signal varies from –V to +V, and has 2^N quantization levels.

$$2V = 2^N \Delta V$$



The quantization error is modeled as a uniform distribution in $-\frac{\Delta V}{2} \leq e_q \leq \frac{\Delta V}{2}$. According to the theory of probability and random variables, the quantization noise is calculated as

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



Signal-to-Quantization Noise Ratio (SQNR) can be calculated as

$$SQNR = \frac{E((V/\sqrt{2})^2)}{E(e_q^2)} = \frac{\left(\frac{2^N \Delta V}{2\sqrt{2}}\right)^2}{\frac{\Delta V^2}{12}} = (2^N \cdot \frac{3}{2})^2$$

$$SQNR_{dB} = 20\log_{10}(2^N \cdot \frac{3}{2}) \approx 6N \ \ [\mathrm{dB}]$$

Multirate sampling

Multirate sampling is used, for example, if data is to be processed at a different sample frequency than it was recorded or if several signals sampled with different frequencies are to be put together.

This procedure is also known as **resampling**. It can be
→ **Upsampling**
→ **Downsampling**

Matlab function:
```
y = downsample(x,n)
&
y = upsample(x,n)
```

Please check them by your own.

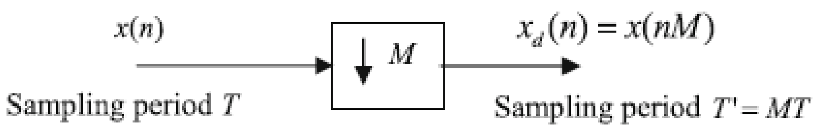
Downsampling

Let $x(n)$ be a sequence obtained by sampling with the sample time T. When downsampling, a sequence with sample interval $T' > T$ is desired. Specifically desired $T' = MT$ and thus have

$$x_d(n) = x(nM)$$

where **M is an integer**.

A block diagram for a downsampling is shown here.




```
% Matlab
fs_original = 16000;       % Original sampling rate: 16 kHz
t = 0:1/fs_original:0.05;

f1 = 500;   % 0.5 kHz
f2 = 1000;  % 1 kHz
f3 = 2000;  % 2 kHz

% original signal
x_mixed = sin(2*pi*f1*t) + sin(2*pi*f2*t) + sin(2*pi*f3*t);
figure;
plot(t, x_mixed);

% Frequency spectrum
N = length(x_mixed);
f = (-N/2:N/2-1)*(fs_original/N);
X_mixed = 1/N*fftshift(abs(fft(x_mixed)));
figure;
plot(f, X_mixed);
```

```
% downsampling
downsample_factor = 2;
fs_down = fs_original / downsample_factor;

x_down = x_mixed(1:downsample_factor:end);
t_down = t(1:downsample_factor:end);
figure;
plot(t_down, x_down);

% Frequency spectrum of downsampled
N_down = length(x_down);
f_down = (-N_down/2:N_down/2-1)*(fs_down/N_down);
X_down = 1/N_down*fftshift(abs(fft(x_down)));
figure;
plot(f_down, X_down);
```

SDU

Why downsampling?

→ Signal length reduces (less memory)
→ Faster processing time
→ If you are using FIR filter, its order (or length) can be reduced.

$$\boxed{\text{FIR half length } M = \frac{B_n f_s}{2\Delta f}}$$

Downsampling spectrum

By Fourier transform of x_d, the following spectrum function is obtained

$$X_d\left(e^{j\omega}\right) = \frac{1}{M}\sum_{k=0}^{M-1} X(e^{\frac{j(\omega-2\pi k)}{M}})$$

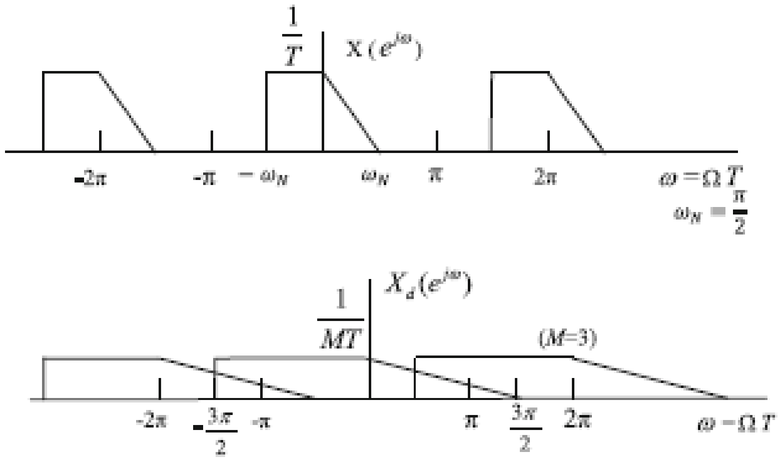
Recap: DFT

$$X(m) = \frac{1}{N}\sum_{n=0}^{N-1} x(n)e^{-j2\pi mn/N}$$

Example here showing:

$$\omega_N = \frac{\pi}{2}$$

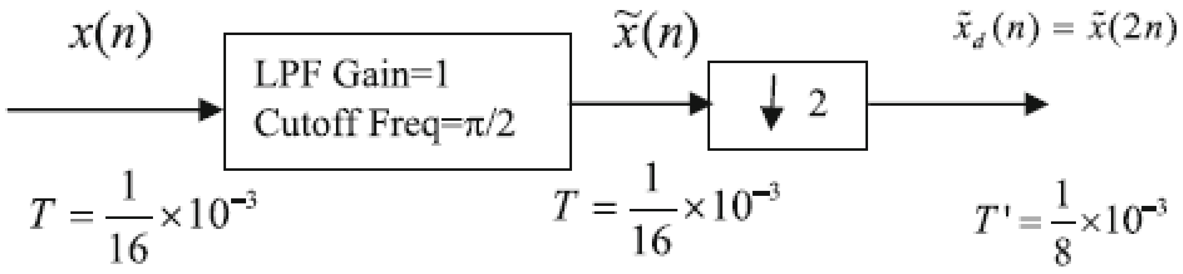
and $M = 3$



SDU

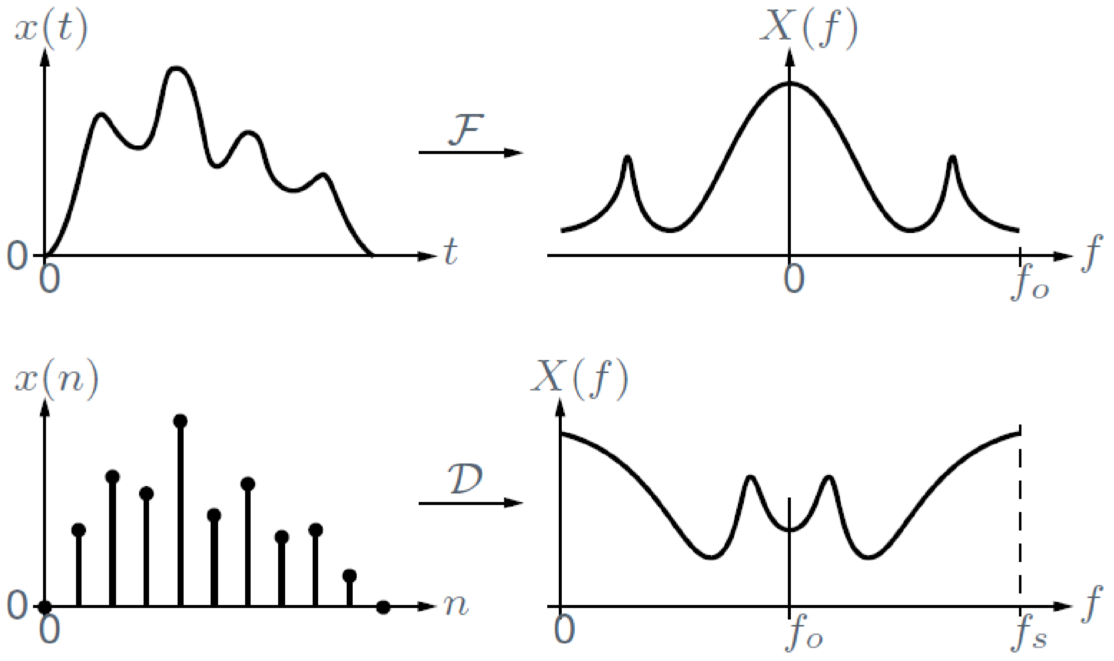
Downsampling anti-aliasing

To avoid aliasing, an anti-aliasing filter $H(z)$ (low-pass filter) must be added before downsampling.



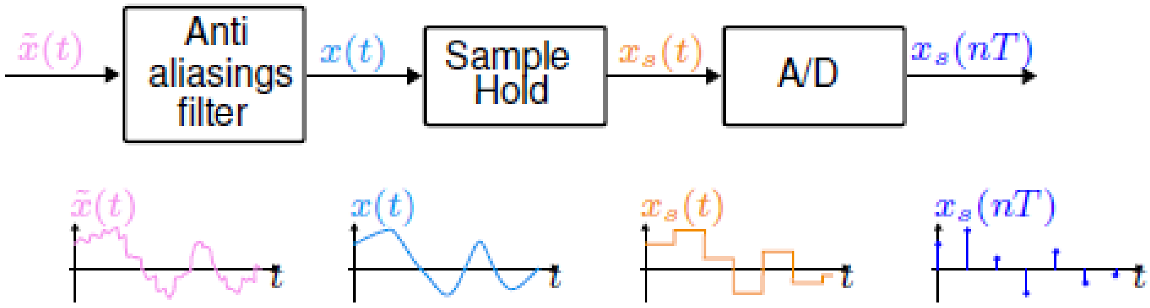
Recap: Aliasing

When performing an FFT, it is important to limit the frequency content of the input signal $x(t)$ is analyzed. If $x(t)$ has frequency components above the convolution frequency $f_o = f_s/2$, then aliasing occurs.



Anti-aliasing filter

To avoid aliasing, a low-pass filter must be inserted before sampling the signal.
This filter must remove frequencies above f_o.



The cutoff frequency of the anti-aliasing filter must be smaller than f_o.
In practice, the filter is better to limit the analysis area to approximately $f_s/3$.

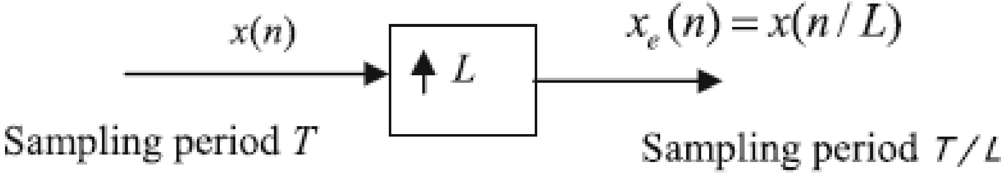
Upsampling

Let $x(n)$ be a sequence obtained by sampling with the sample time T. When upsampling, a sequence with sample interval $T' < T$ is desired. Specifically desired $T' = T/L$ and thus have

$$x_u(n) = \begin{cases} x(n/L) & \text{hvis } n/L \in \mathbb{Z} \\ 0 & \text{ellers} \end{cases}$$

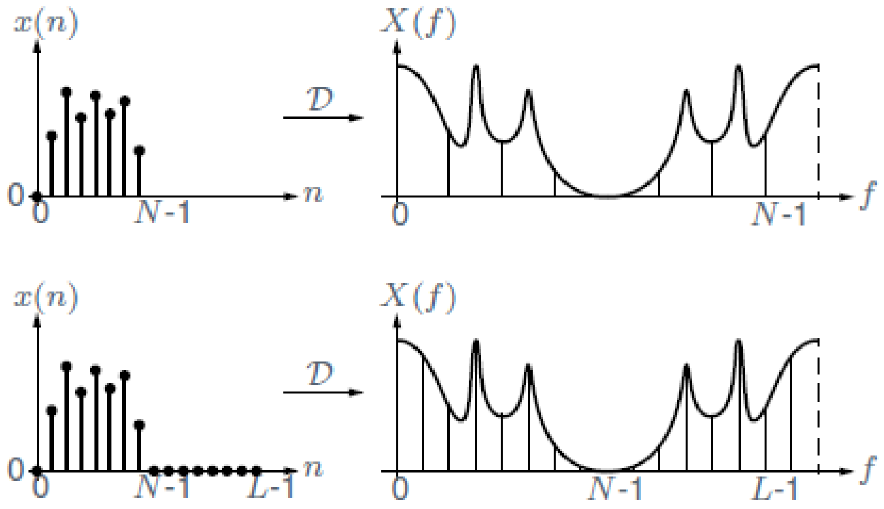
where L **is an integer**

A block diagram for an upsampling is shown here.

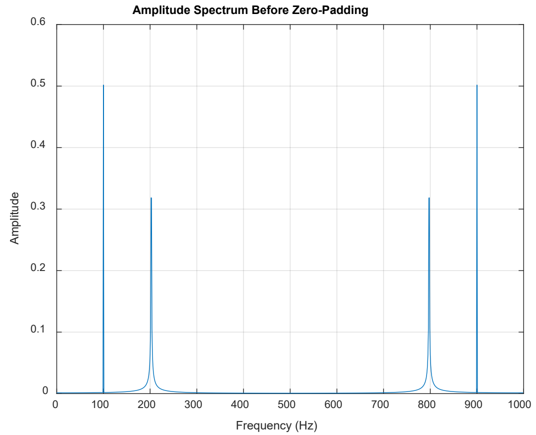


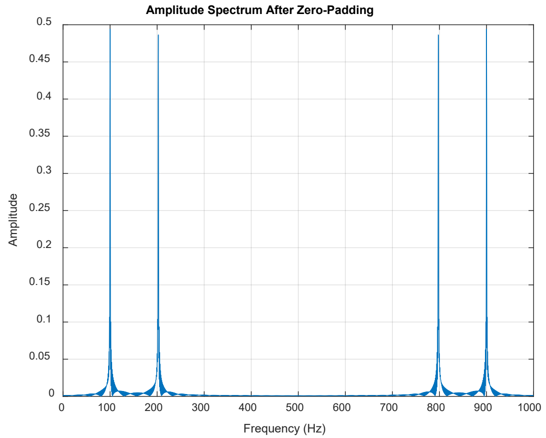
Zero-padding

When an N-point Fourier Transform is performed, the frequency step is $F = f_s/N$. If a higher resolution of the frequency spectrum is desired, then zero filling can be used so that the signal is increased to L samples.




```
fs = 1e3; % [Hz] sampling frequency
t = 0:0.001:1-0.001;
x = cos(200*pi*t)+sin(405*pi*t); % Signal

% Plot the time domain signal
figure;
plot(t, x);
title('Original Signal in Time Domain');

% FFT Analyse
N = length(x); % Original signal length
X = fft(x);
f = fs*(0:N-1)/N;

figure;
plot(f, abs(X)/N);
title('Amplitude Spectrum Before Zero-Padding');

% Zero-padding
N_padded = 2048;
X_padded = fft(x, N_padded); % FFT with zero-padding

% Frequency axis for padded signal
f_padded = fs*(0:N_padded-1)/N_padded;

figure;
plot(f_padded, abs(X_padded)/N);
title('Amplitude Spectrum After Zero-Padding');
```



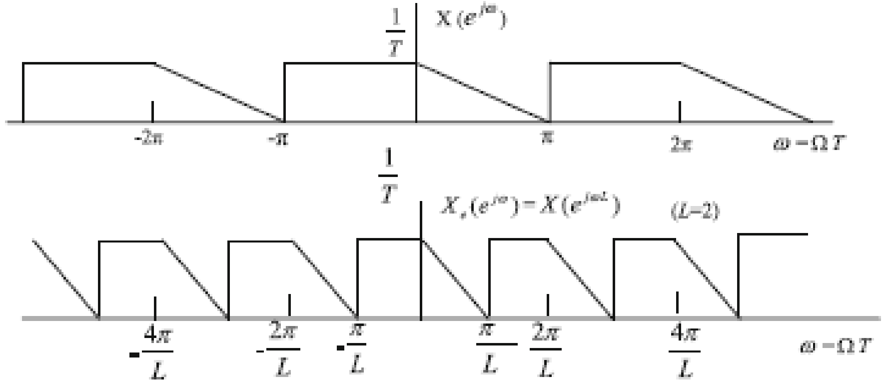


SDU

Upsampling spectrum

By Fourier transform of x_u, the following spectrum function is obtained

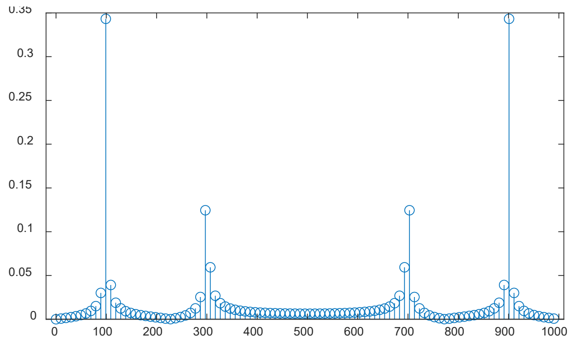
$$X_u(e^{j\omega}) = \sum_{n=Lk} x_u(n)e^{-j\omega n} = \sum_k x_u(Lk)e^{-jL\omega k} = \sum_k x(k)\, e^{-jL\omega k} = X(e^{jL\omega})$$

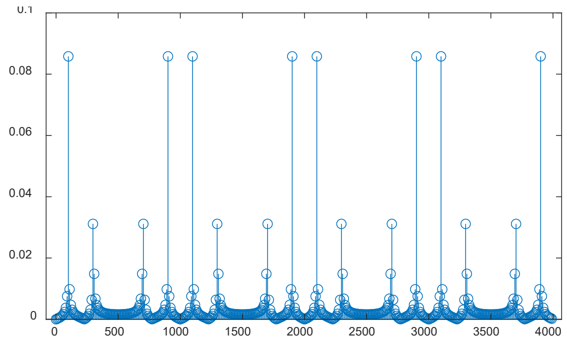


Matlab – upsampling example

```
fs_original = 1000;        % Original sampling rate: 1 kHz
t_original = 0:1/fs_original:0.1;

f1 = 100;   % 100 Hz
f2 = 300;   % 300 Hz

x_original = 0.7 * sin(2*pi*f1*t_original) + 0.3 * sin(2*pi*f2*t_original);
figure;
plot(t_original, x_original);

% Compute FFT
N_original = length(x_original);
f_original = (0:N_original-1) * (fs_original/N_original);
X_original = fft(x_original);
figure;
stem(f_original, abs(X_original)/N_original);
```

```
upsample_factor = 4;
fs_upsampled = fs_original * upsample_factor; % New sampling rate: 4 kHz

% Upsampling: Insert zeros between samples
x_upsampled_zeros = zeros(1, length(x_original) * upsample_factor);
x_upsampled_zeros(1:upsample_factor:end) = x_original;

% Time vector for upsampled signal
t_upsampled = (0:length(x_upsampled_zeros)-1) / fs_upsampled;
figure;
plot(t_upsampled, x_upsampled_zeros);

% Compute FFT
N_upsampled = length(x_upsampled_zeros);
f_upsampled = (0:N_upsampled-1) * (fs_upsampled/N_upsampled);

X_upsampled_zeros = fft(x_upsampled_zeros);
figure;
stem(f_upsampled, abs(X_upsampled_zeros)/N_upsampled);
```

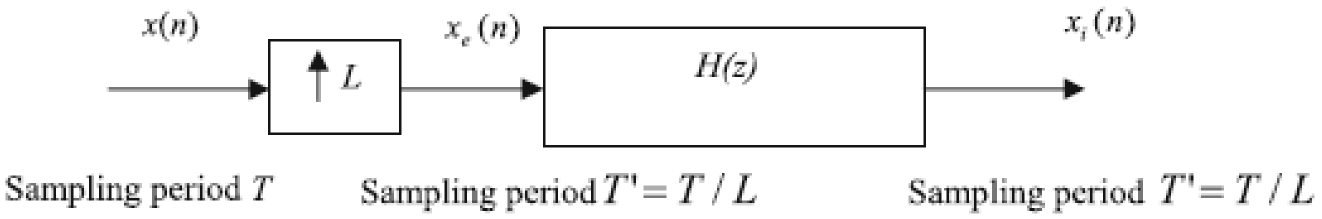


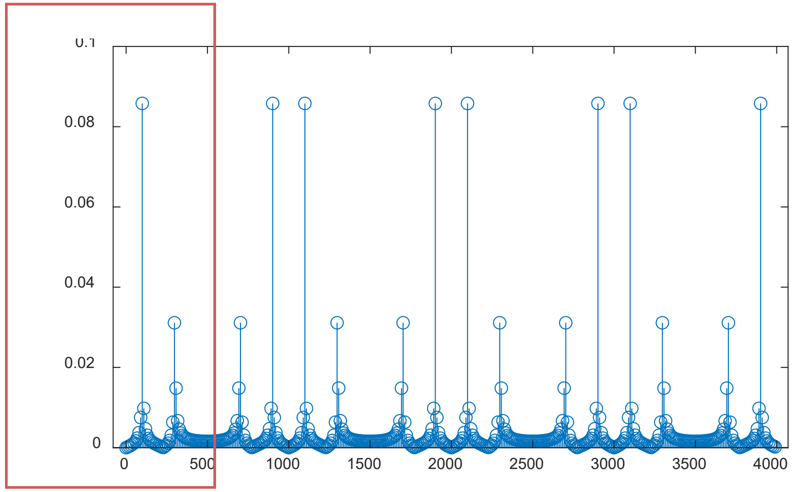


SDU

Correction of spectrum

To avoid the repeated spectrum, a low-pass filter $H(z)$ must be added after the upsampling.

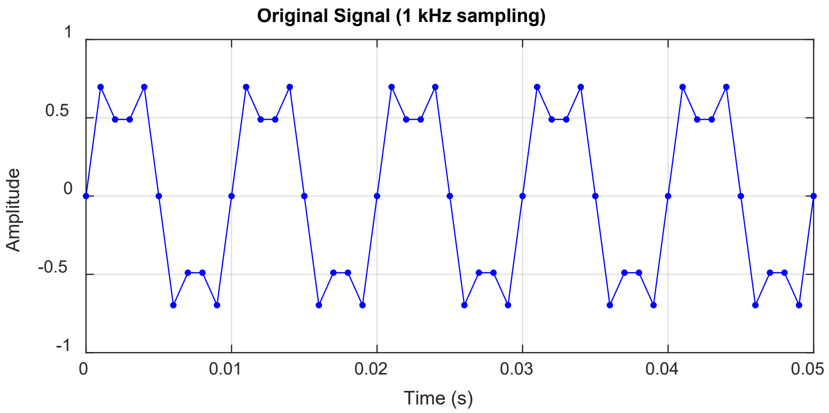


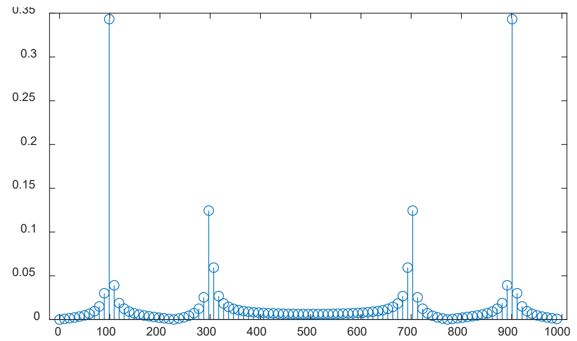


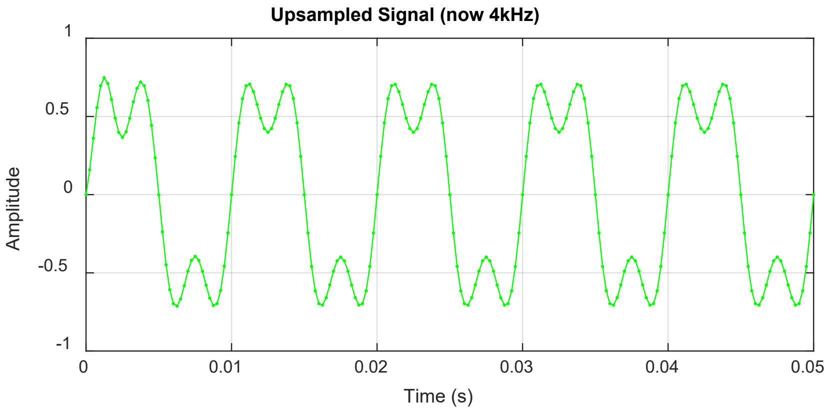
SDU

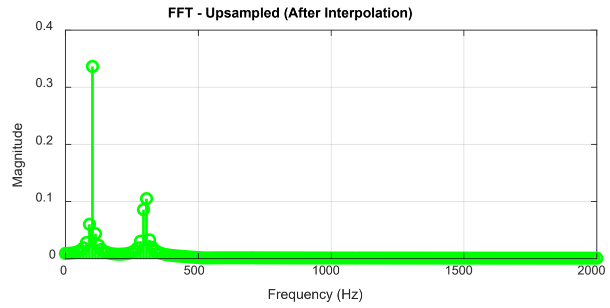
FIR lowpass filter is used

```
cutoff = fs_original/2;     % Cutoff at original Nyquist frequency
filter_order = 100;
b = fir1(filter_order, cutoff/(fs_upsampled/2));

% Apply filter to remove spectral images
x_upsampled_filtered = conv(x_upsampled_zeros, b * upsample_factor, ‘same’);

t_upsampled_filtered = (0:length(x_upsampled_filtered)-1) / fs_upsampled;

% Plot interpolated signal
plot(t_upsampled_filtered, x_upsampled_filtered);

% Compute FFT
N_filtered = length(x_upsampled_filtered);
f_filtered = (0:N_filtered-1) * (fs_upsampled/N_filtered);

X_upsampled_filtered = fft(x_upsampled_filtered);

stem(f_filtered, abs(X_upsampled_filtered)/N_filtered);
```









SDU