Life Sciences and
Facility Management

Institute of
Computational Life Sciences

Filtering

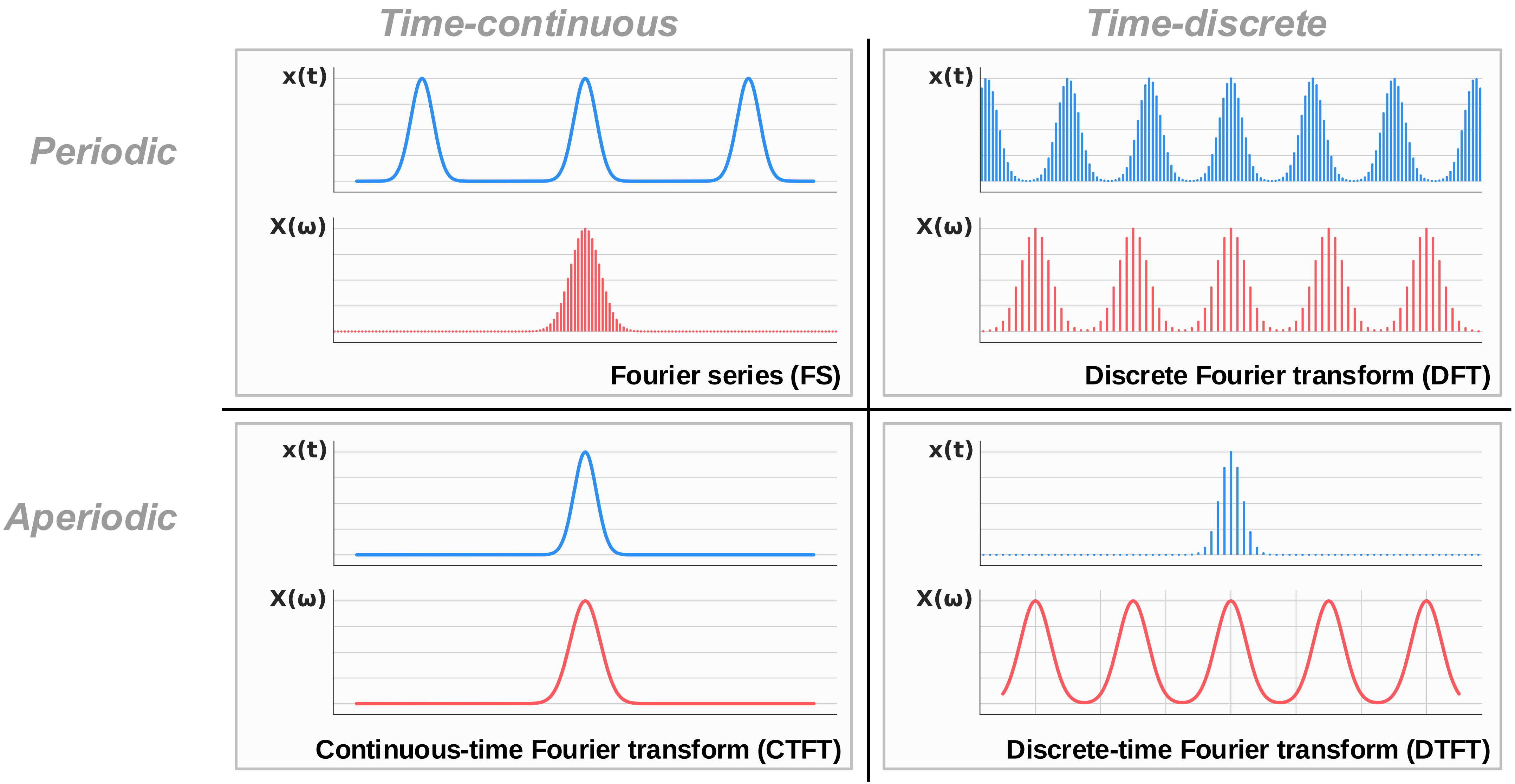
Image and Signal Processing

Norman Juchler

Fourier’s landscape

We are really wrapping this up now!

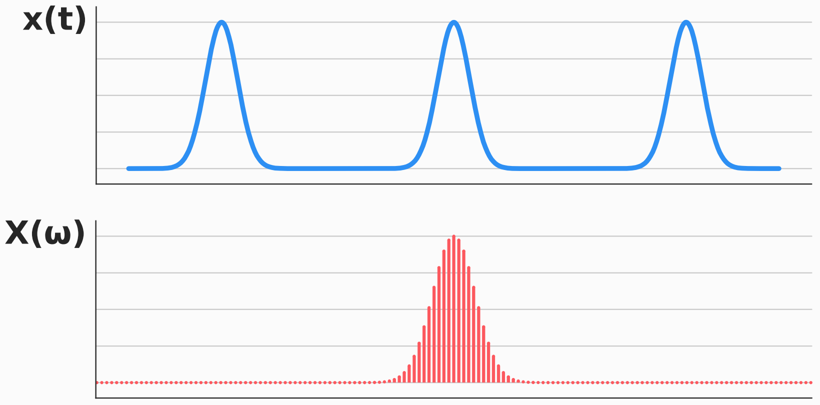
Fourier's landscape



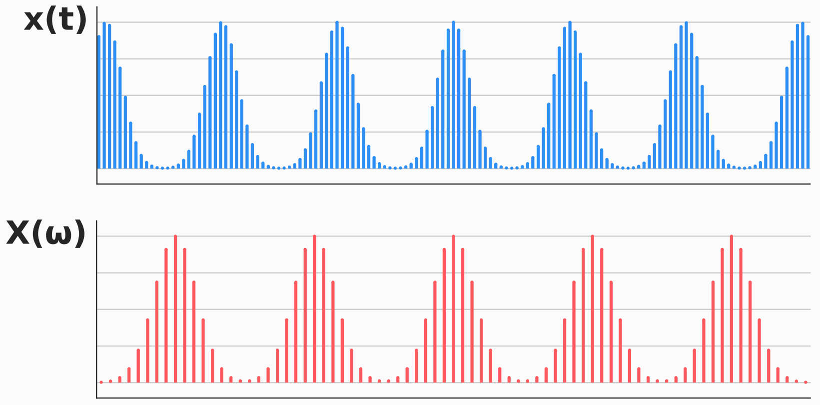
Fourier's landscape

Time-continuous | *Time-discrete*

Periodic



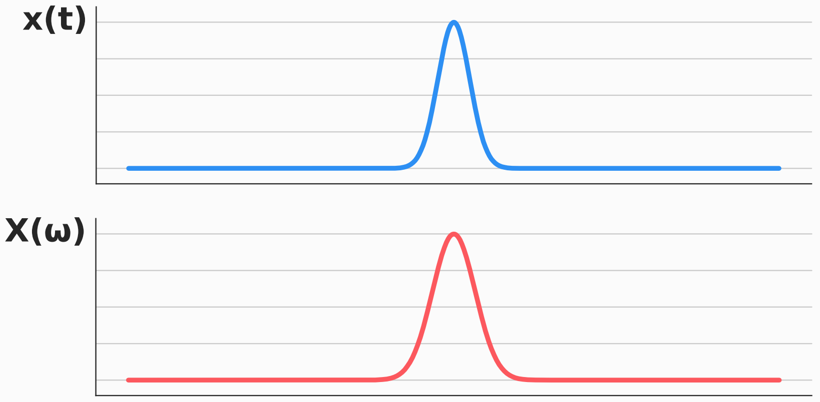
Fourier series (FS)



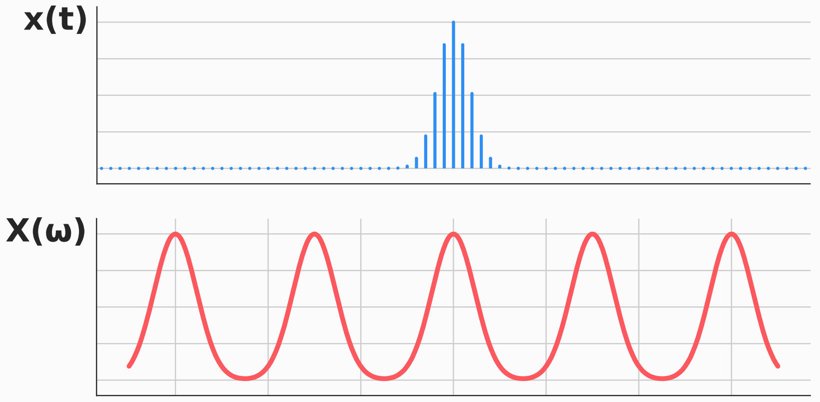
Discrete Fourier transform (DFT)

Related:
- **Fast Fourier transform**
- Discrete cosine transform
- Discrete sine transform

Aperiodic



Continuous-time Fourier transform (CTFT)



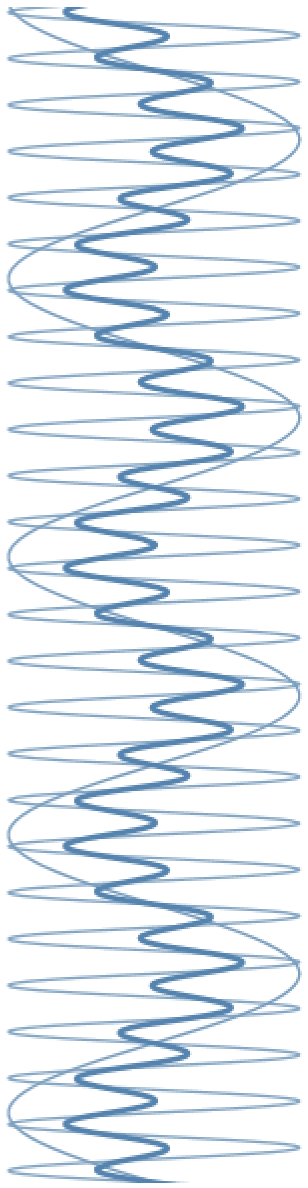
Discrete-time Fourier transform (DTFT)

Generalization:
Laplace Transform

Generalization:
z - Transform

A zoo of different transformations...

- ...but they share many common characteristics:
  - Each has a forward and inverse transformation.
  - They all exhibit duality between time-domain and frequency-domain representations.
  - They share similar mathematical properties, including:
    - Linearity
    - Time/ frequency shifting
    - Convolution theorem
    - ...
- Take-home messages:
  - Fourier theory allows us to decompose signals into their fundamental frequency components.
  - We can seamlessly switch between the time domain and the frequency domain, and vice versa.
  - Different types of Fourier transforms are suited for different types of signals, ensuring the most effective representation.



The discrete Fourier transform

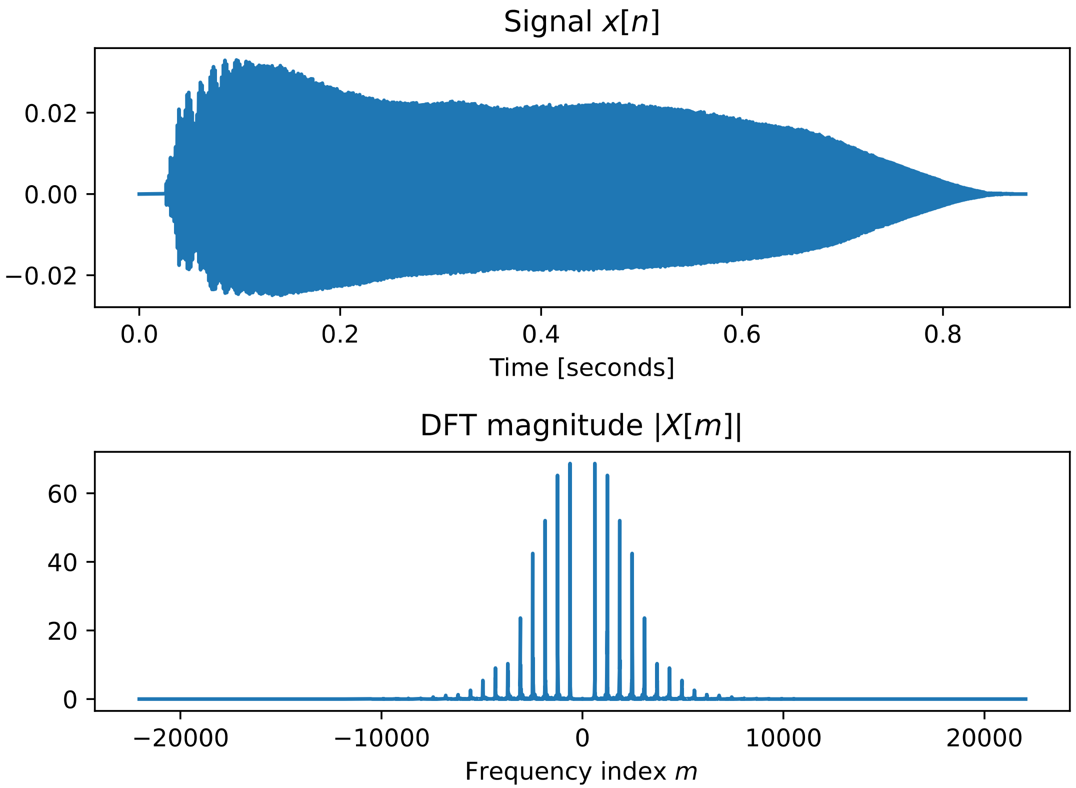
- **Input**: Finite sequence of N equally-spaced samples of a signal
- **Output**: Finite sequence of N equally-spaced samples of the DTFT

$$\boxed{X[m] = \sum_{n=0}^{N-1} x[n] \cdot e^{-i2\pi \frac{m}{N} n}}$$

$$f_m = \begin{cases} \frac{m}{N} \cdot f_s & \text{if } 0 \leq m < N/2 \\ \frac{m-N}{N} \cdot f_s & \text{if } N/2 \leq m < N \end{cases}$$

- **Observations**:
  - The $X[m]$ represents discrete samples of the (continuous) discrete-time Fourier transform
  - The $X[m]$ in general are complex numbers (with magnitude and phase)
  - The DFT $X[m]$ is periodic with period N (a fact leveraged by fftshift()...)
  - Every $X[m]$ is associated with the frequency f_m
    - Note that we usually consider frequencies f_m that are negative for $m \geq N/2$ (periodic extension)
    - The first value $X[0]$ is known as the **direct current** (DC) or static component, with $f_0 = 0$
    - $X[0]$ always equals the sum of the sample values

Example: A real sound



$$X[m] = \sum_{n=0}^{N-1} x[n] \cdot e^{-i2\pi \frac{m}{N} n}$$

$$f_m = \begin{cases} \frac{m}{N} \cdot f_s & \text{if } 0 \leq m < N/2 \\ \frac{m-N}{N} \cdot f_s & \text{if } N/2 \leq m < N \end{cases}$$

Look at the illustrations and try to answer the following **questions**:

- What is the sampling frequency?
- How many samples?
- What is the static component about? How is it related to the mean value?
- At which frequency is the peak?

Answers:

- Sampling at 44’100Hz
- Samples 0.88·44’100 ≈ 38’000
- Static component: first value X[0] = sum of x[n] ≈ 0
- Corresponds to the pitch of the note played (here: D_5#=618.8Hz)

The discrete Fourier transform – a naive implementation:

```
 1 def dft(x):
 2     '''Compute the Discrete Fourier Transform of an input signal x of N samples.'''
 3
 4     N = len(x)
 5     X = np.zeros(N, dtype=np.complex)
 6
 7     # Compute each X[m]
 8     for m in range(N):
 9
10         # Compute similarity between x and the m'th basis
11         for n in range(N):
12             X[m] = X[m] + x[n] * np.exp(-2j * np.pi * m * n / N)
13
14     return X
```

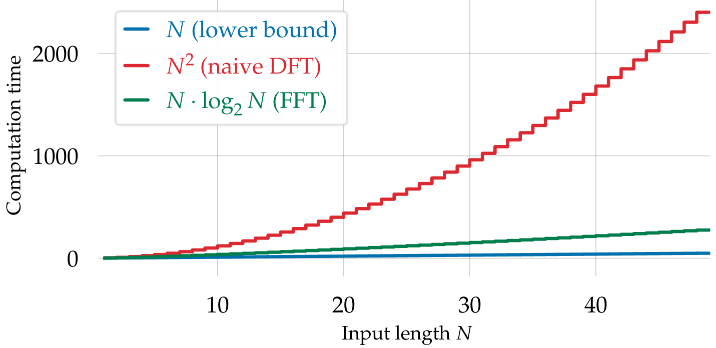
Question: How long does it take to compute the DFT for a signal x?

Answer: Line 12 needs to be calculated N^2 times, or using the big-O notation: The complexity of this DFT is $O(N^2)$

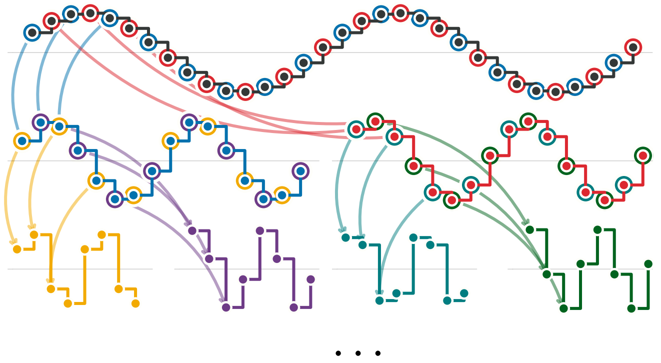
Code sample taken from Brian McFee / Digital Signals Theory

The Fast Fourier Transform (FFT)

- The FFT is an efficient algorithm to compute the DFT of a discrete signal
- History:
  - Some ideas of FFT can be traced back to C.F. Gauss (1805).
  - The algorithm was first published in 1965 by J. Cooley and J. Tukey.
  - The most popular version is called radix-2 Cooley-Tukey algorithm.
- Key idea: **Divide and conquer**
  - Exploit DFT symmetry to recursively break the computation into smaller sub-problems, eliminating redundant calculations.
  - The roots of unity play a central role here...
  - Instead of directly computing the DFT for $N = N_1 N_2$, it reformulates the problem in terms of for N_1 smaller problems of size N_2.
  - This approach reduces computational complexity from $O(N^2)$ for a naive implementation to $O(N \log N)$!



The increase of the amount of work (that is, the number of computations) increases with the problem size

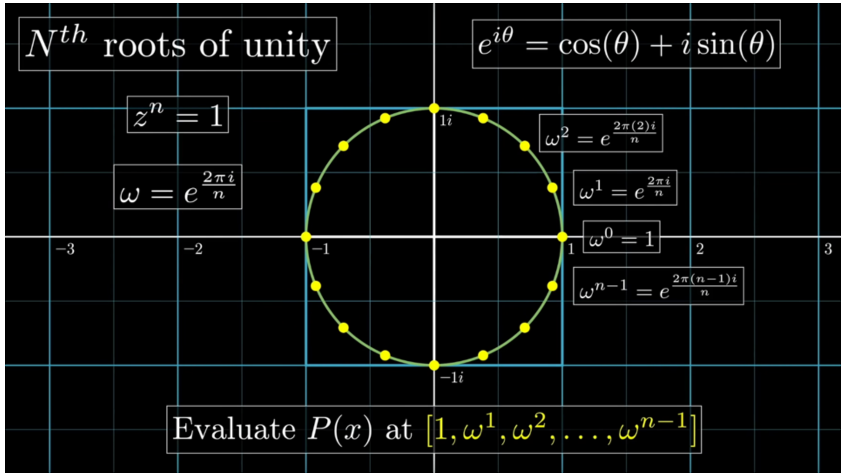


A discrete signal of N=32 samples (top row) is divided into its even and odd samples (middle-left and middle-right).

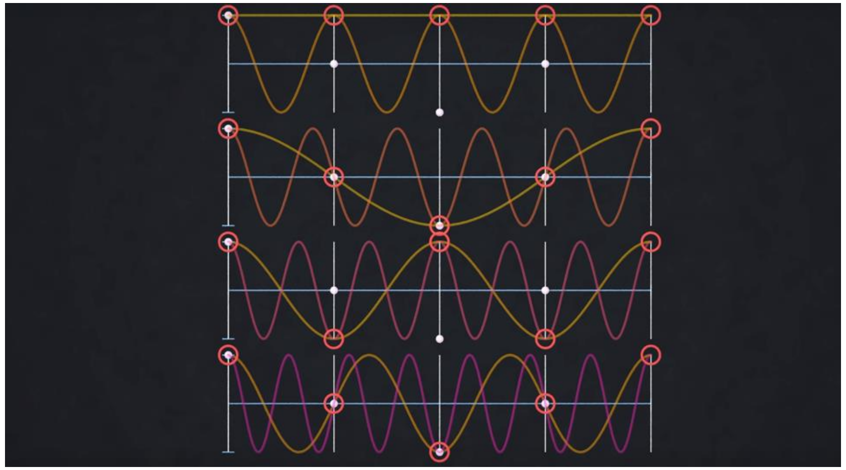
Illustrations: Brian McFee / Digital Signals Theory

Recommended videos

- Reducible / 2023: The Fast Fourier Transform (FFT): Most Ingenious Algorithm Ever? Link

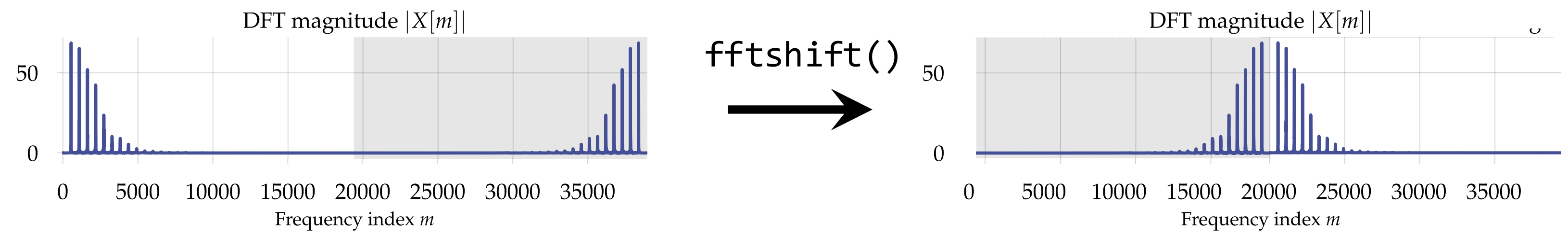


- Veritasium / 2023: The Most Important Algorithm Of All Time. Link



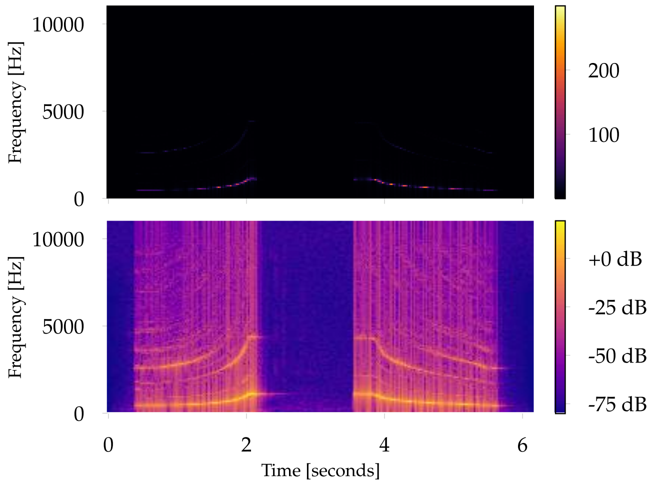
The Fast Fourier Transform (FFT)

- The package scipy.fft offers fast implementations of various FFT algorithms
- See this tutorial for an overview how to use FFT.
- The main functions:
  - `fft(), fft2(), fftn():` Compute the FFT on 1D, 2D, and nD input.
  - `ifft(), ifft2(), ifftn():` Compute the inverse DFT in 1D, 2D or nD
  - `rfft(), rfft2(), rfftn():` Compute the FFT on real input, faster than the fft*() functions.
  - `irfft(), irfft2(), irfftn():` Compute the inverse real DFT in 1D, 2D or nD
  - `fftshift():` fft*() and fftfreq() return all components at positive frequencies, followed by components at negative frequencies. This function swaps the two halves so that the zero-frequency component is in the center: [0, 1, 2, -3, -2, -1] → [-3, -2, -1, 0, 1, 2, 3].
  - `fftfreq(), rfftfreq():` Get the frequencies of each bin returned by the FFT functions, such as the X-axis.

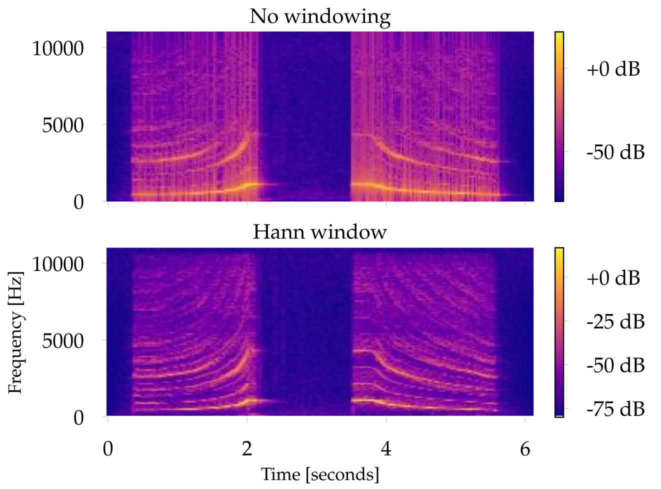


The short-time Fourier Transform (STFT)

- The DFT makes use of all samples in the signal
- Implicitly, this is assuming that frequency content is “stationary” over the duration of the signal.
- For time-varying signals, this is often not the case
- **Idea**: Divide a long signal up into short pieces (so-called **frames**), analyze each piece separately.
- Parameters:
  - Frame length (number of samples per frame)
  - Hop length (number of samples between two frames)
- STFTs can be visualized with **spectrograms**
  - Constructed by stacking the frames horizontally
  - Hint 1: Use the decibel scale for better visualization
  - Hint 2: Use windowing / apodization to avoid banding artifacts



Spectrogram with linear magnitude scaling (top) and with decibel scaling: $A_{\mathrm{dB}} = 20 \cdot \log_{10} A$



Spectrogram without (top) and with windowing (bottom). Use a window (here: Hann) to attenuate discontinuities at the ends of the frames.

Convolution

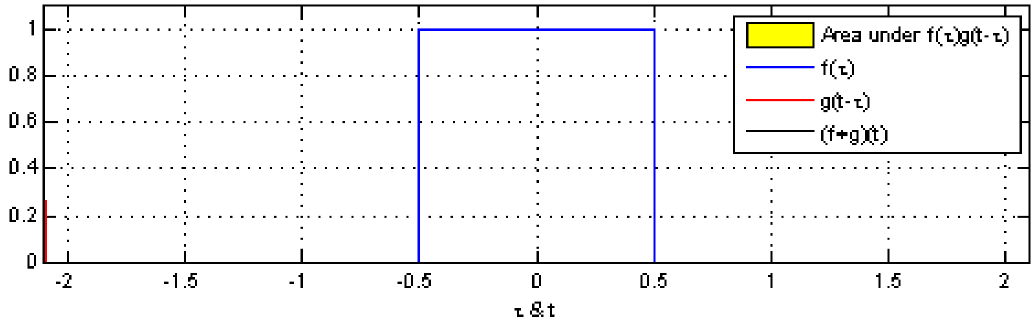
Revisited

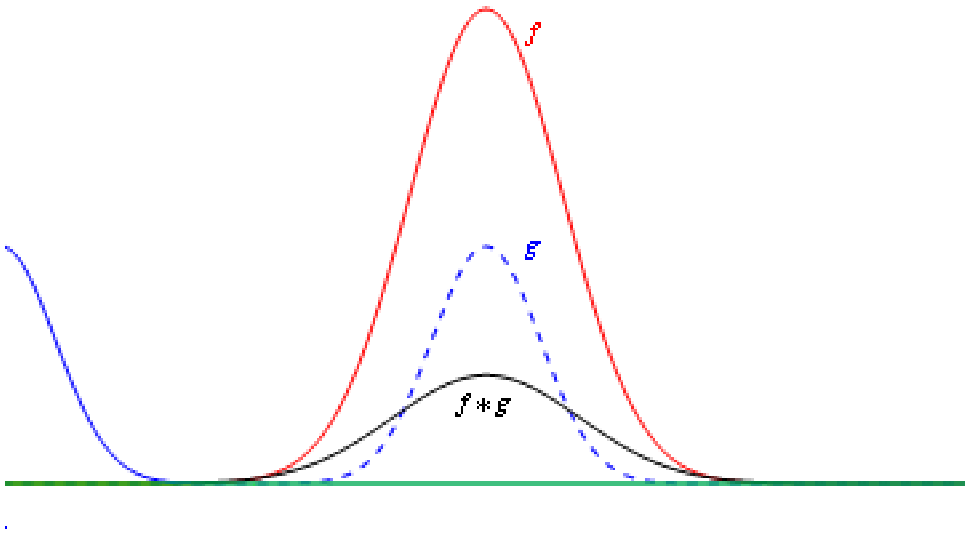
Convolution of continuous-time signals

- The convolution is an operation that combines two functions to produce a third function.

$$y(t) = (h * x)(t) := \int_{-\infty}^{\infty} h(\tau)x(t-\tau)\, d\tau$$

- In words, integral computes the area under the product of $h(\tau)$ and $x(t-\tau)$ as t varies.
- Demo applet
- **Video**: 3blue1brown / 2023. But what is a convolution? Link



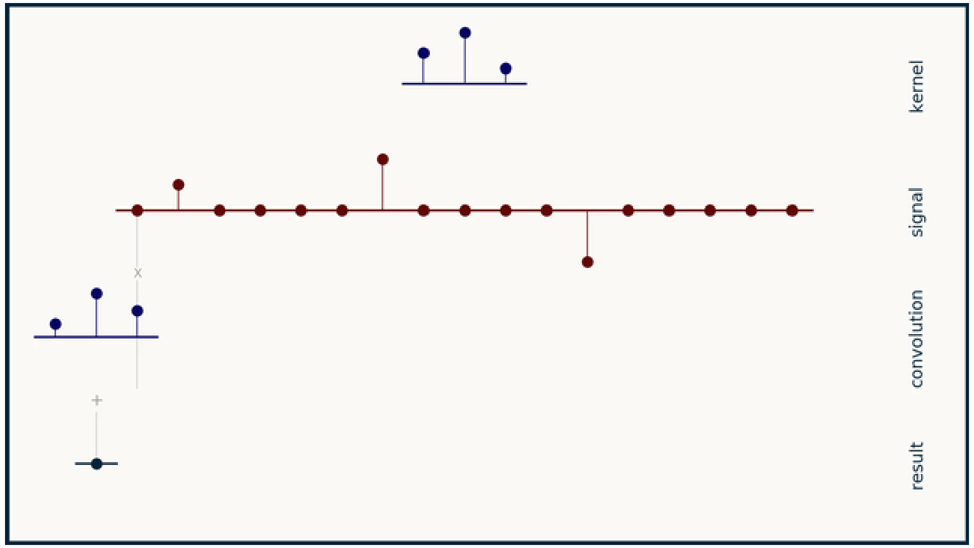


Convolution of discrete-time signals

- There exists also a discrete-time definition for convolution:

$$y[n] = (h * g)[n] = \sum_{m=-\infty}^{\infty} h[m]x[n-m]$$

- In the animation right is $x[n]$ the input signal, $h[n]$ the kernel, and $y[n]$ the output signal.



Animation: Link

Naive implementation of a discrete-time convolution

```python
# Compute the lengths of our input and filter
N = len(x)
K = len(h)

# Allocate an output buffer
y = np.zeros(N)

# Iterate over delay values
for k in range(K):

    # For each delay value, iterate over all sample indices
    for n in range(N):

        if n >= k:
            # No contribution from samples n < k
            y[n] += h[k] * x[n-k]
```

- scipy.signal.convolve() is a fast alternative
  - Makes use of FFT!
  - Question: Why again?

Convolution and Fourier transform

- **Result**: The Fourier transform simplifies the convolution operation to a simple multiplication in the frequency domain!

$$\boxed{\begin{aligned} y(t) &= h(t) * x(t) \quad \Leftrightarrow \quad Y(\omega) = H(\omega) \cdot X(\omega) \\ y[n] &= h[h] * x[n] \quad \Leftrightarrow \quad Y[m] = H[m] \cdot X[m] \end{aligned}}$$

- Computing the convolution in the frequency domain is very easy!
- We can efficiently recover the time-domain signal via inverse FFT
- Recommended procedure (for discrete signals):
  - Pad $x[n]$ and $h[n]$ to the same length (if necessary)
  - Use the FFT to compute DTFs $X[m]$ and $H[m]$
  - Multiply $X[m]$ and $H[m]$ elementwise
  - Compute the inverse DFT to recover $y[n]$

Signal filtering

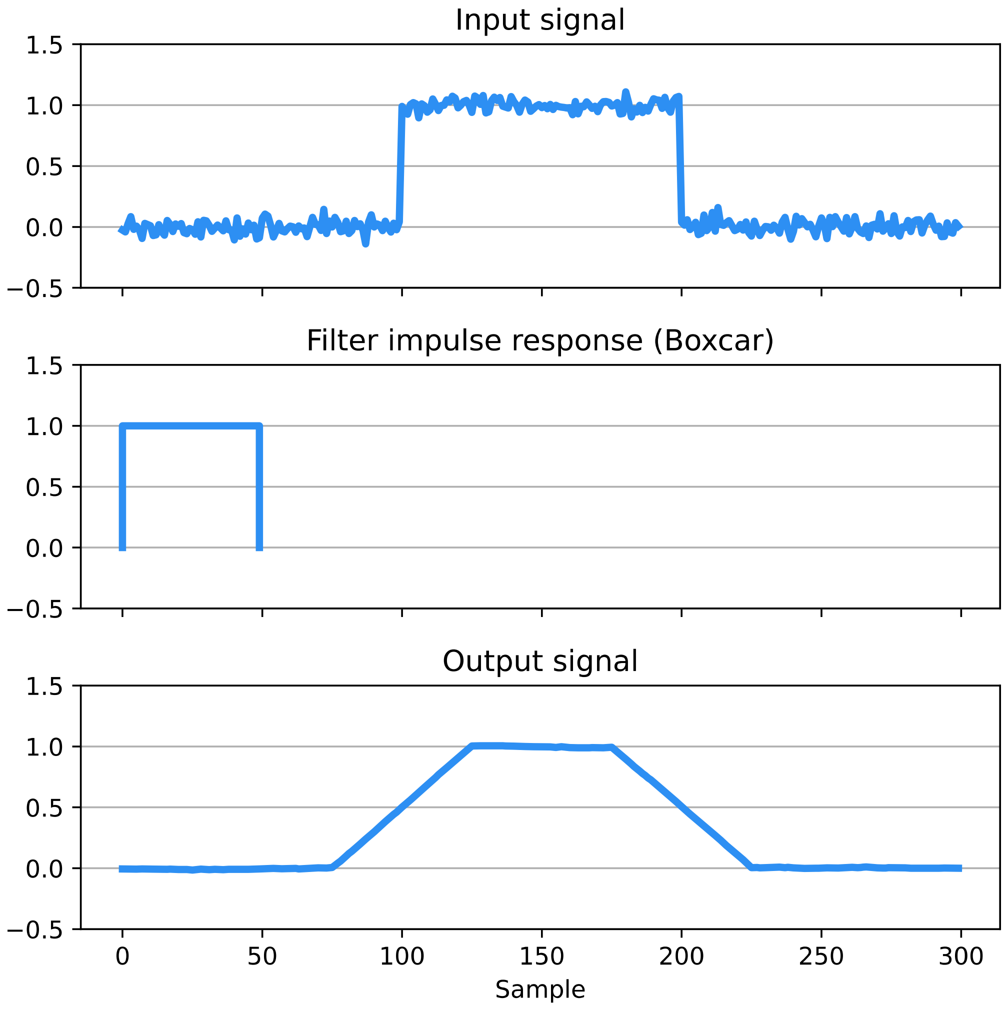
First concepts

Example: Signal smoothing using convolution

- Input:
  - Noisy signal $x(t)$
  - A filter window $h(t)$
- Output
  - Filtered signal $y(t)$
- Solution:

```
import numpy as np
from scipy import signal

noise = 0.05
width = 50
win = signal.windows.boxcar(width)
x = np.repeat([0., 1., 0.], 100)
x += np.random.normal(0, noise, x.shape)
y = signal.convolve(x, win, mode='same') / sum(win)
```

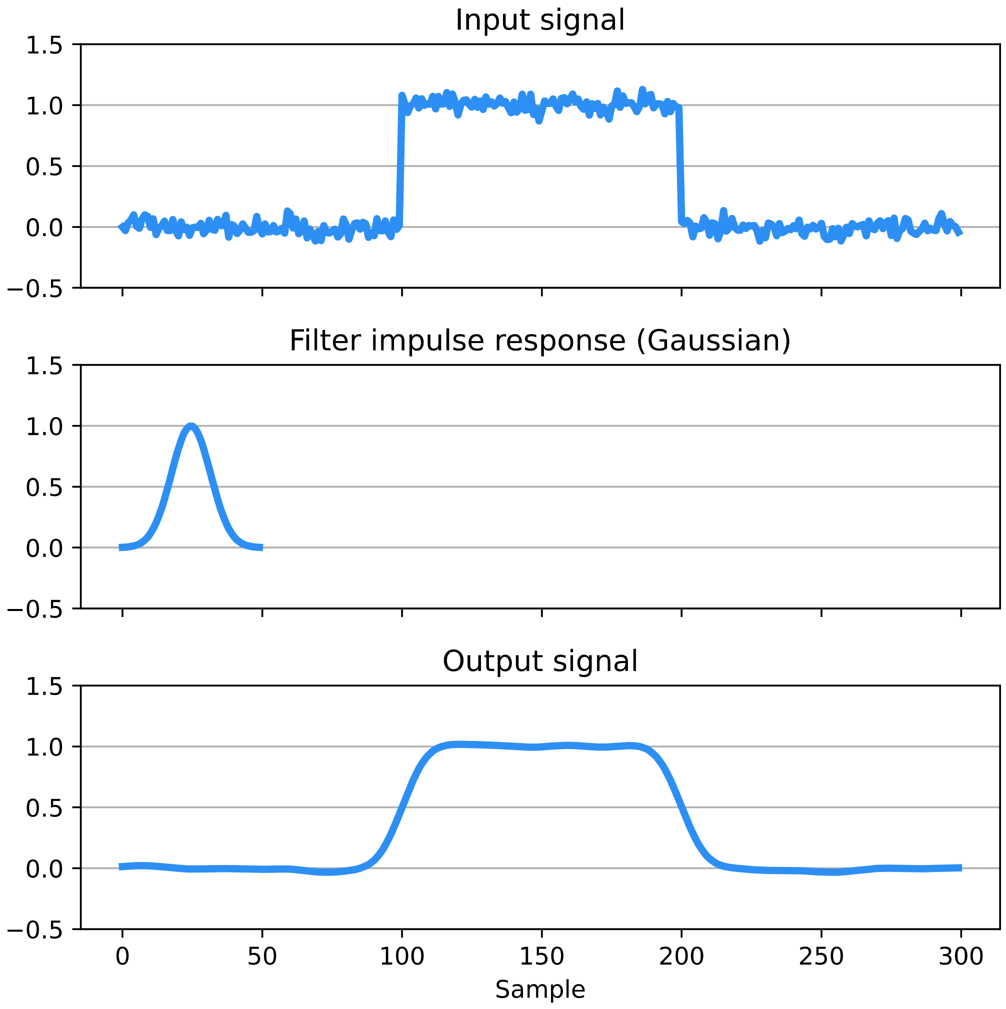


Example: Signal smoothing using convolution

- Observations:
  - We sacrifice some of the signal, but
  - We can remove the noise.

```
import numpy as np
from scipy import signal

noise = 0.05
width = 50
win = signal.windows.gaussian(width, std=7)
x = np.repeat([0., 1., 0.], 100)
x += np.random.normal(0, noise, x.shape)
y = signal.convolve(x, win, mode='same') / sum(win)
```

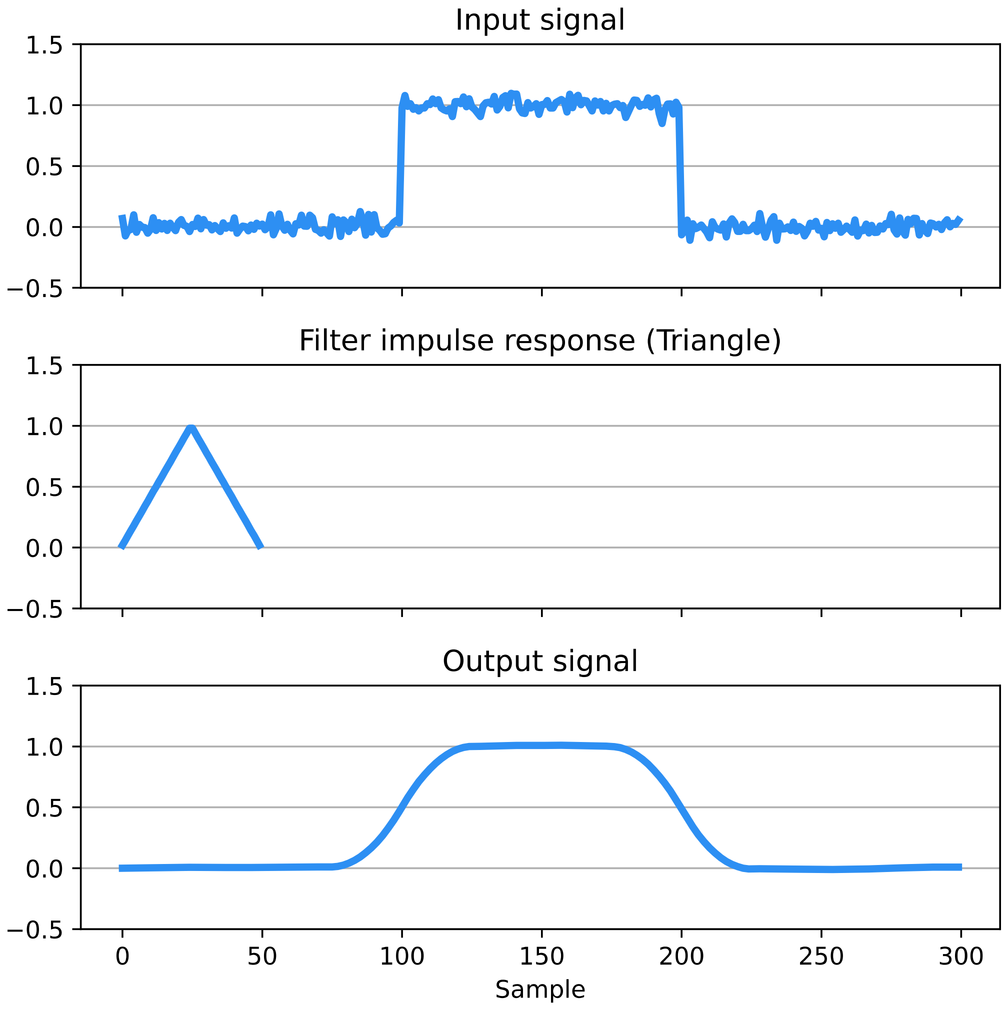


Example: Signal smoothing using convolution

- Many different window types are possible:
  - Box
  - Triangle
  - Blackman
  - Hamming
  - Hann
  - Parzen
  - Exponential
  - Tukey
  - Lanczos
  - ...

```
import numpy as np
from scipy import signal

noise = 0.05
width = 50
win = signal.windows.triang(width)
x = np.repeat([0., 1., 0.], 100)
x += np.random.normal(0, noise, x.shape)
y = signal.convolve(x, win, mode='same') / sum(win)
```

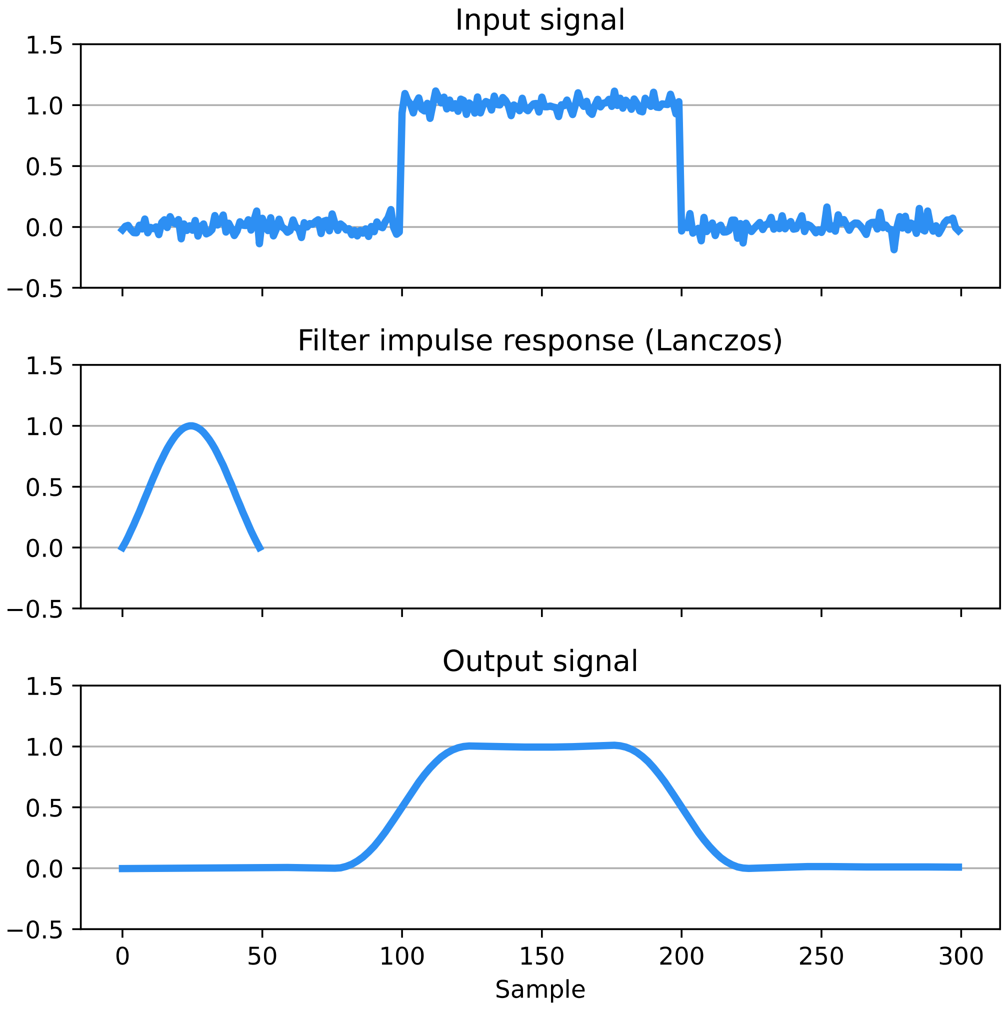


Example: Signal smoothing using convolution

- What type of smoothing window should be used? Which parameters to select?
  - A typical filter design question!
  - Requires an understanding of signal characteristics
  - Spectral analysis of signal and filter!

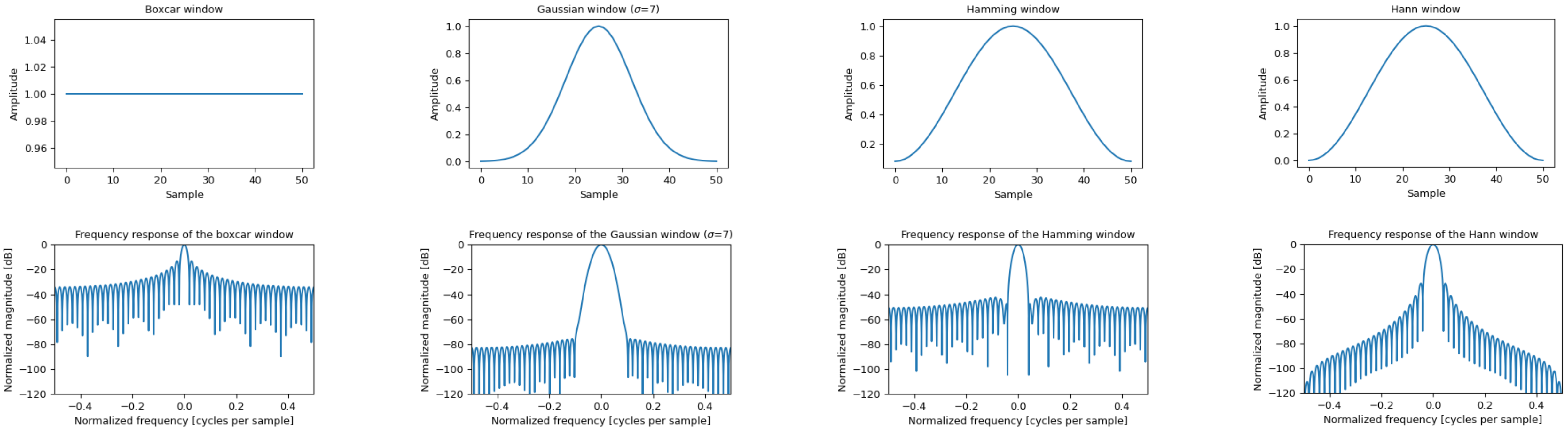
```python
import numpy as np
from scipy import signal

noise = 0.05
width = 50
win = signal.windows.lanczos(width)
x = np.repeat([0., 1., 0.], 100)
x += np.random.normal(0, noise, x.shape)
y = signal.convolve(x, win, mode='same') / sum(win)
```



Example: Signal smoothing using convolution

- Comparison of different windows



$h_{box}[n] = 1$

$h_{gauss}[n] = e^{\frac{1}{2}\left(\frac{n}{\sigma}\right)^2}$

$h_{lanc}[n] = sinc_{\pi}\left(\frac{2n}{M-1} - 1\right)$

$h_{hann}[n] = 0.5 - 0.5\cos\left(\frac{2\pi n}{M-1}\right)$

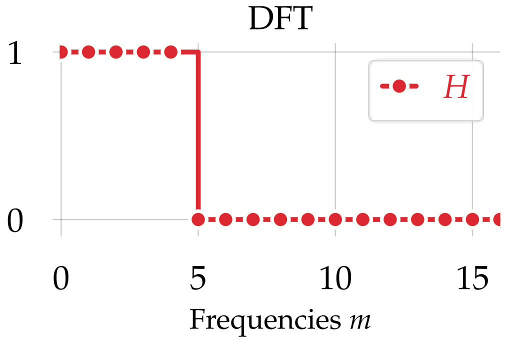
What is the theoretically best low-pass filter?

- Alternative approach: start in the frequency domain
- **Goal**: Get rid of all frequencies above some cut-off f_c
- **Idea**: Let's define the ideal low-pass filter:

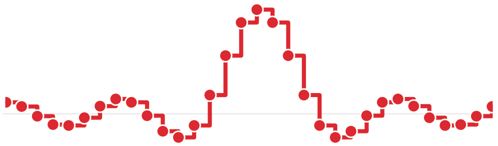
$$H[m] = \begin{cases} 0, & \text{if } f_m > f_c \\ 1, & \text{if } f_m \leq f_c \end{cases}$$

  We then can apply the filter in the frequency domain by multiplying it with the signal $X[m]$ and recover the resulting signal y[n] in the time-domain.

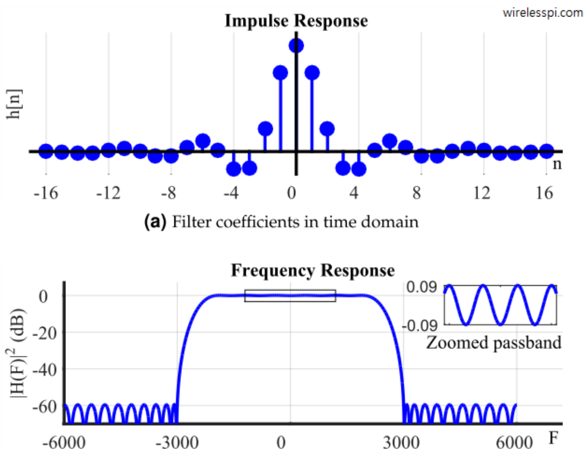
- Such a filter is called **ideal low-pass filter**, or **brick-wall filter**.



$H[m]$: The ideal low-pass filter in the frequency-domain



$h[n]$: Filter representation in the time-domain (so-called impulse response)

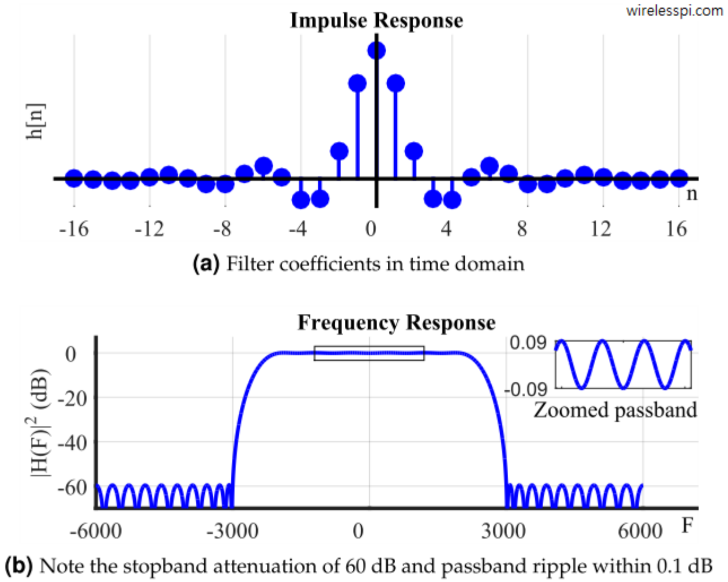


(a) Filter coefficients in time domain

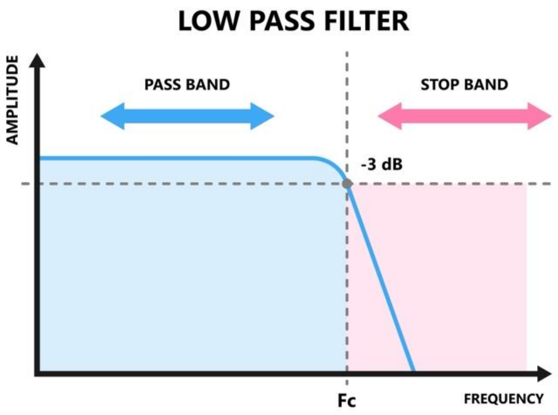
(b) Note the stopband attenuation of 60 dB and passband ripple within 0.1 dB

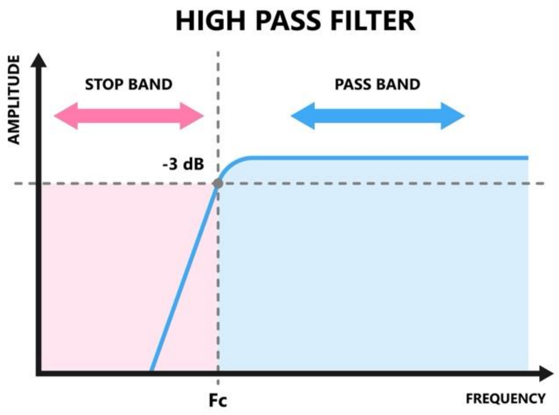
Problems with the ideal low-pass filter

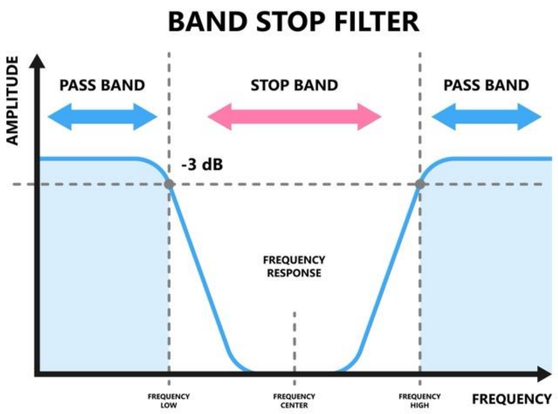
- Has an infinite representation in the time domain
- Cannot be handled properly with DTF, as the filter lengths are limited
- Observations:
  - We already saw that it is difficult to represent sharp edges in the time-domain using frequency components (we require infinitely high frequencies)
  - The same holds true in the frequency domain: The sharper the filter is in the frequency domain, the more time-domain samples are necessary.
  - Note that the filter requires infinitely many samples at t<0! The ideal filter therefore is a non-causal filter.
- See here for an illustrative demonstration.

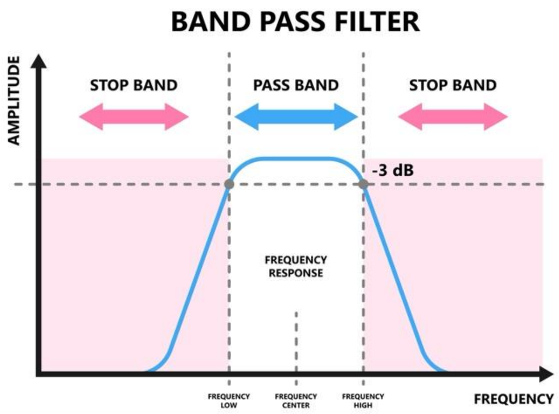


Overview: Types of filters









Frequency bands

- A signal is called **band-limited** if its frequency components are limited within frequencies:

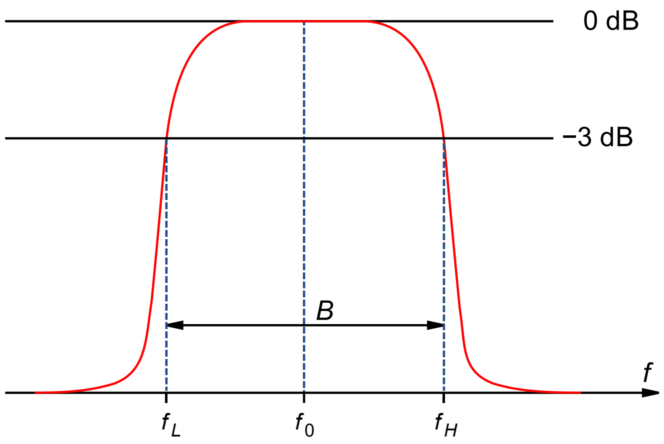
$$f_L \leq f \leq f_H$$

- The **bandwidth** of a signal is defined as the difference between the extreme frequencies:

$$B = \Delta f = f_H - f_L$$

- A strictly band-limited signal does not carry energy at frequencies outside the band limits

- In practice, a signal is considered band-limited if its energy outside of a frequency range is low enough to be considered negligible



Amplitude spectrum of a bandlimited signal. Source of illustration: Wikimedia