1.Sampling
2. Interpolation and Curve Fitting
3. Correlation and Convolution
- 3.1.1. autocorrelation and interpretation

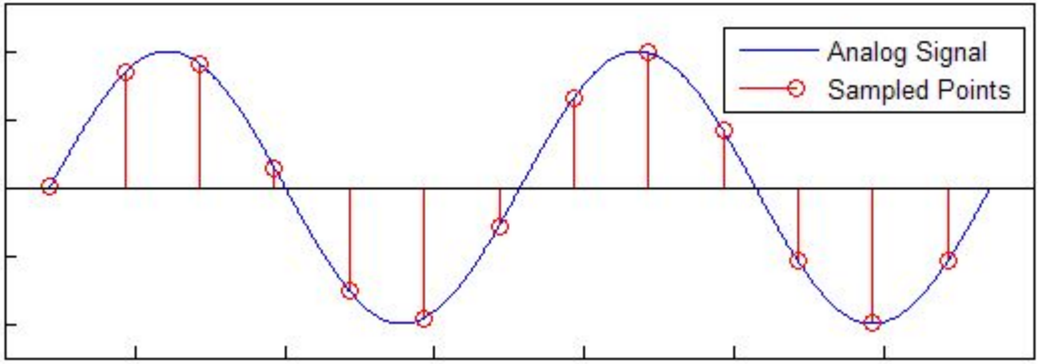
4. Time/Frequency domain and Fourier Series coefficients
- 4.1.1. Periodic signal : Time <-> Frequency
- 4.1.2. Fourier coefficient : Amplitude vs. Phase

5. DFT and Sampling rate
- 5.1.1. Nyquist Frequency (higher bound)
- 5.1.2. F(0) (lower bound)

6. Numerical FFT : scipy.fft
7. Time-Freq. analysis and filtering (exercice)

Sampling

In signal processing, <u>*sampling*</u> is the ***reduction of a continuous (analog) time signal to a discrete-time signal (digital, numerical)***



- For functions that vary with time, let $s(t)$ be a continuous function (or "analog signal") to be sampled

- sampling is performed by **measuring the value of the continuous function *every ω seconds***, which is called the sampling interval or the ***<u>sampling period.</u>***
- The sampled function is given by the sequence: $s(n\omega)$**,** for integer values of n.
- The ***<u>sampling frequency</u>*** or ***<u>sampling rate,</u>*** f_s , is the *average number of samples obtained in one second*, thus $f_s = 1/\omega$.
- Its units are *samples per second* or **hertz "*Hz*"** e.g. ***48 kHz*** is ***48,000*** samples per second.

If a function $x(\mathrm{t})$ contains no frequencies higher than B hertz, it is completely determined by giving its ordinates at a series of points spaced $1/(2B)$ seconds apart.

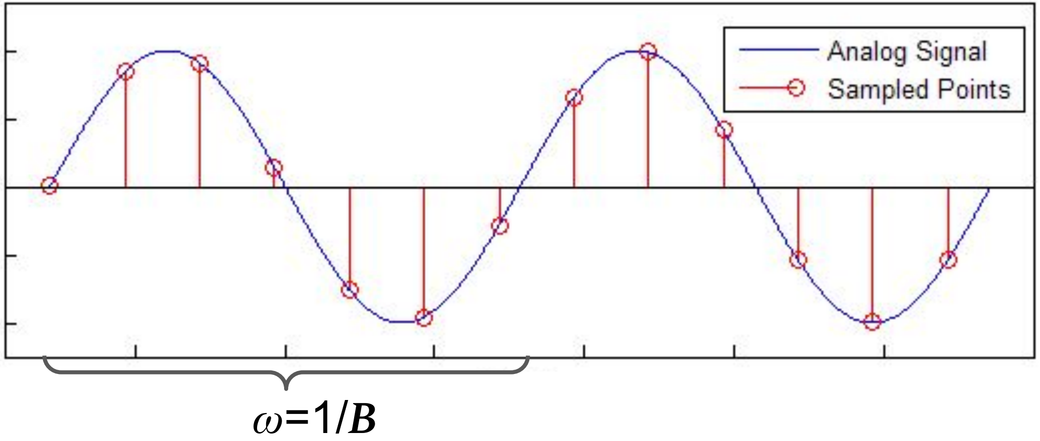
A sufficient sample-rate is therefore **<u>anything larger than 2B samples per second</u>**.

Equivalently, for a given sample rate f_s , perfect reconstruction is guaranteed possible for a *<u>band limit</u>*

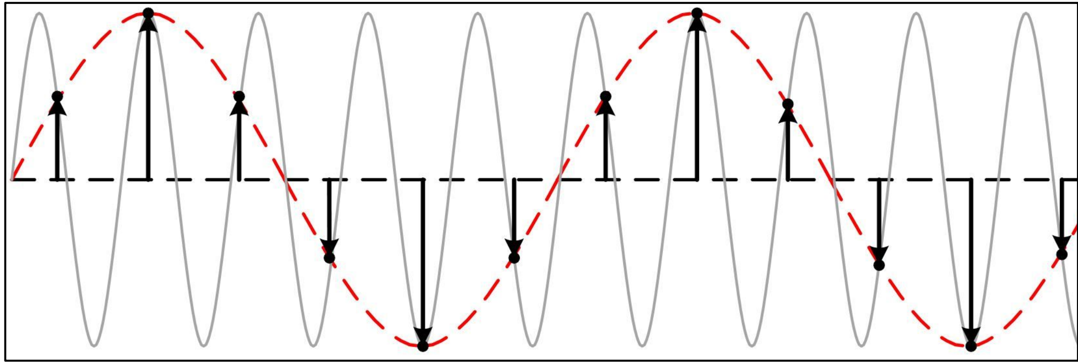
$B < f_s / 2$

B : Nyquist frequency = $f_s / 2$

good f_s



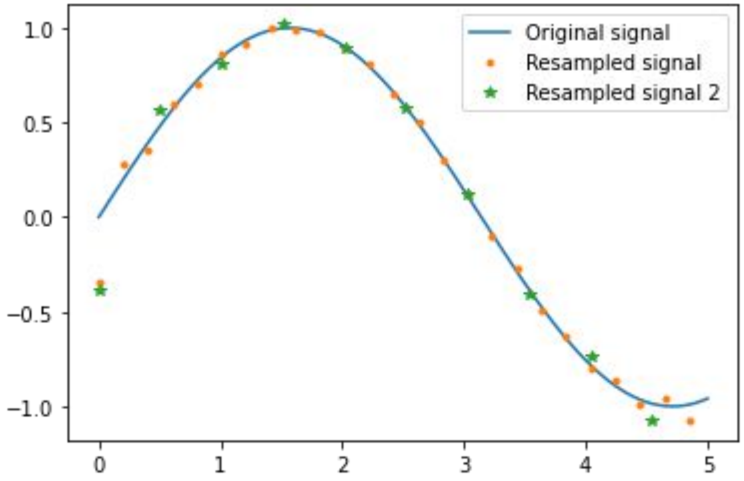
bad f_s



Signal sampling representation is done in the figure below.

The continuous signal S(t) is represented with a blue colored line while the discrete samples are indicated by the dots.

```
t = np.linspace(0, 5, 100) # large number of point (high freqeuncy)
x = np.sin(t)              # signal “looks” continuous
from scipy import signal
import matplotlib.pyplot as plt
x_resampled = signal.resample(x, 25)
x_resampled_2 = signal.resample(x, 10)
plt.plot(t, x, label = "Original signal")
plt.plot(t[::4], x_resampled, '.', label = 'Resampled signal')
plt.plot(t[::10], x_resampled_2, '*', label = 'Resampled signal 2')

plt.legend()
```



Reconstructing a continuous function from samples is done by interpolation algorithms.

1.Sampling
2. Interpolation and Curve Fitting
3. Correlation and Convolution
    3.1.1. autocorrelation and interpretation
4. Time/Frequency domain and Fourier Series coefficients
    4.1.1. Periodic signal : Time <-> Frequency
    4.1.2. Fourier coefficient : Amplitude vs. Phase
5. DFT and Sampling rate
    5.1.1. Nyquist Frequency (higher bound)
    5.1.2. F(0) (lower bound)
6. Numerical FFT : scipy.fft
7. Time-Freq. analysis and filtering (exercice)

Linear Interpolation

Linear interpolation is a method of curve fitting using linear polynomials to construct new data points within the range of a discrete set of known data

If the two known points are given by the coordinates x(0), y(0) and x(1), y(1) ; the linear interpolant is the straight line between these points.

For a value x in the interval x(0), x(1), the value y along the straight line is given from the equation of slopes.

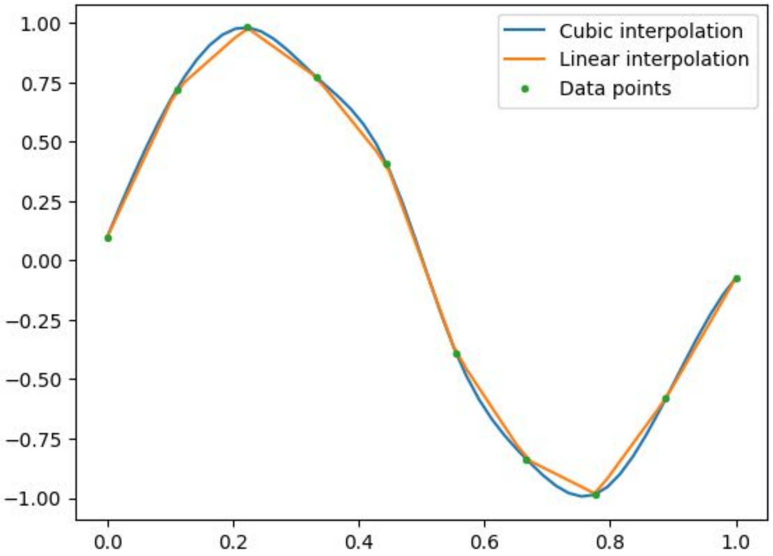
$$\frac{y - y_0}{x - x_0} = \frac{y_1 - y_0}{x_1 - x_0}$$

Solving the equation for y, gives the formula for linear interpolation in the interval [x(0), x(1)] :

$$y = y_0 \frac{x_1 - x}{x_1 - x_0} + y_1 \frac{x - x0}{x_1 - x_0}$$

Linear interpolation is often used to approximate a value of some function f using two known values of that function at other points.

```
import matplotlib.pyplot as plt
from scipy.interpolate import interp1d
sampling_points = np.linspace( 0, 1, 10)
noise = (np.random.random( 10)*2 - 1) * 1e-1
datapoints = np.sin( 2 * np.pi * sampling_points) + noise
plt.plot(sampling_points,datapoints, "*")
datapoints.shape
linear_interp = interp1d(sampling_points, datapoints)
interpolation_time = np.linspace( 0, 1, 50)
linear_results = linear_interp(interpolation_time)
cubic_interp = interp1d(sampling_points, datapoints, kind= 'cubic')
cubic_results = cubic_interp(interpolation_time)
plt.plot(interpolation_time, cubic_results,  label =  "Cubic interpolation" )
plt.plot(interpolation_time, linear_results, label =  "Linear interpolation" )
plt.plot(sampling_points, datapoints, ".", label = "Data points" )
plt.legend()
```



Curve fitting is a type of optimization that finds an optimal set of parameters for a defined function that best fits a given set of observations.

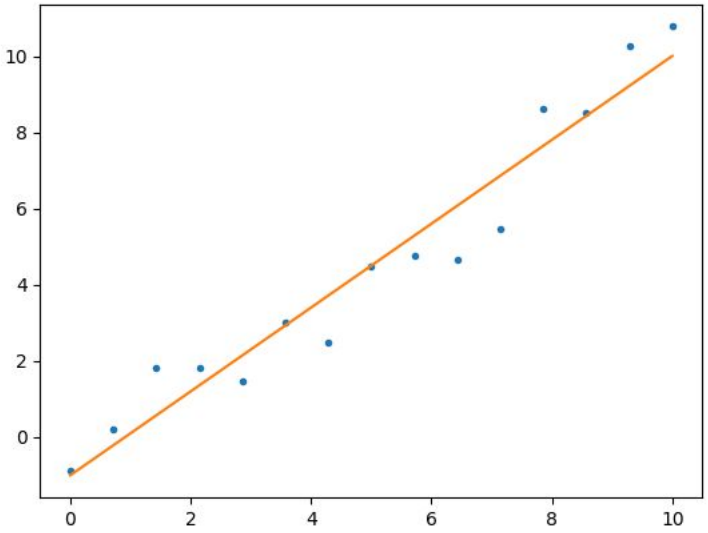
Unlike supervised learning, curve fitting requires that you define the function that maps examples of inputs to outputs.

The mapping function, also called the basis function can have any form you like, including a straight line (linear regression), a curved line (polynomial regression), and much more.

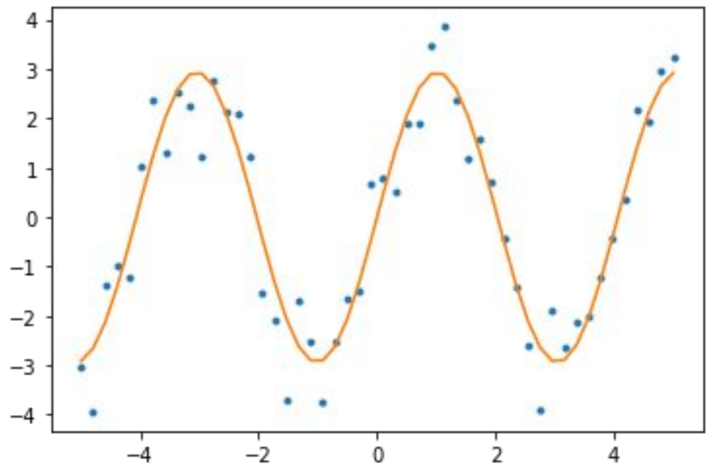
This provides the flexibility and control to define the form of the curve, where an optimization process is used to find the specific optimal parameters of the function.

The scipy.optimize module provides algorithms for function minimization (scalar or multi-dimensional), curve fitting and root finding.

```
from scipy import optimize
x = np.linspace(0,10,15)
y = np.linspace(0,10,15) + np.random.normal(size = 15)
def function(x,a,b):
   return a*x + b

params, params_covariance = optimize.curve_fit(function, x, y)
a,b = params
plt.plot(x, y, ".")
plt.plot(x, function(x, params[0], params[1]))
```

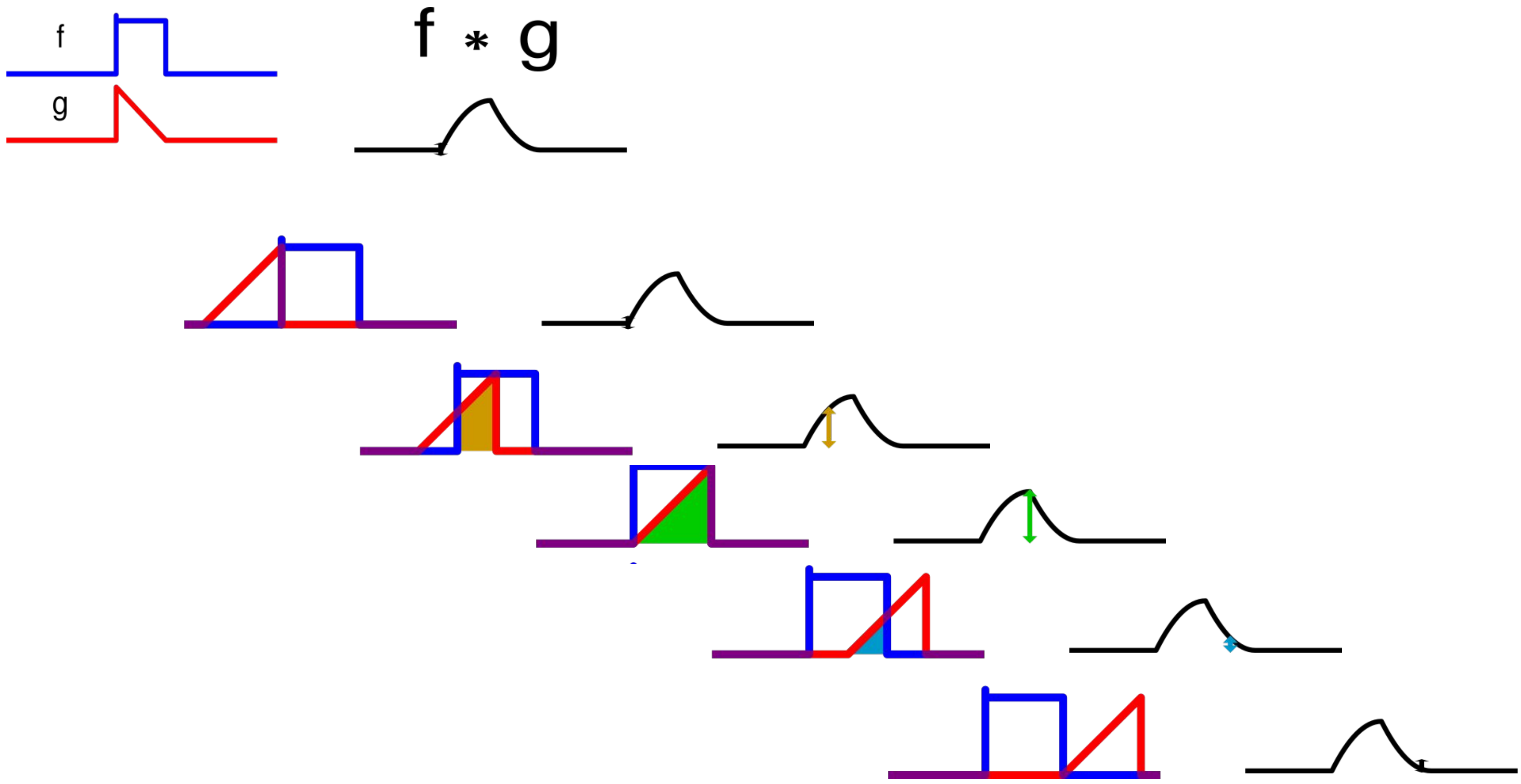


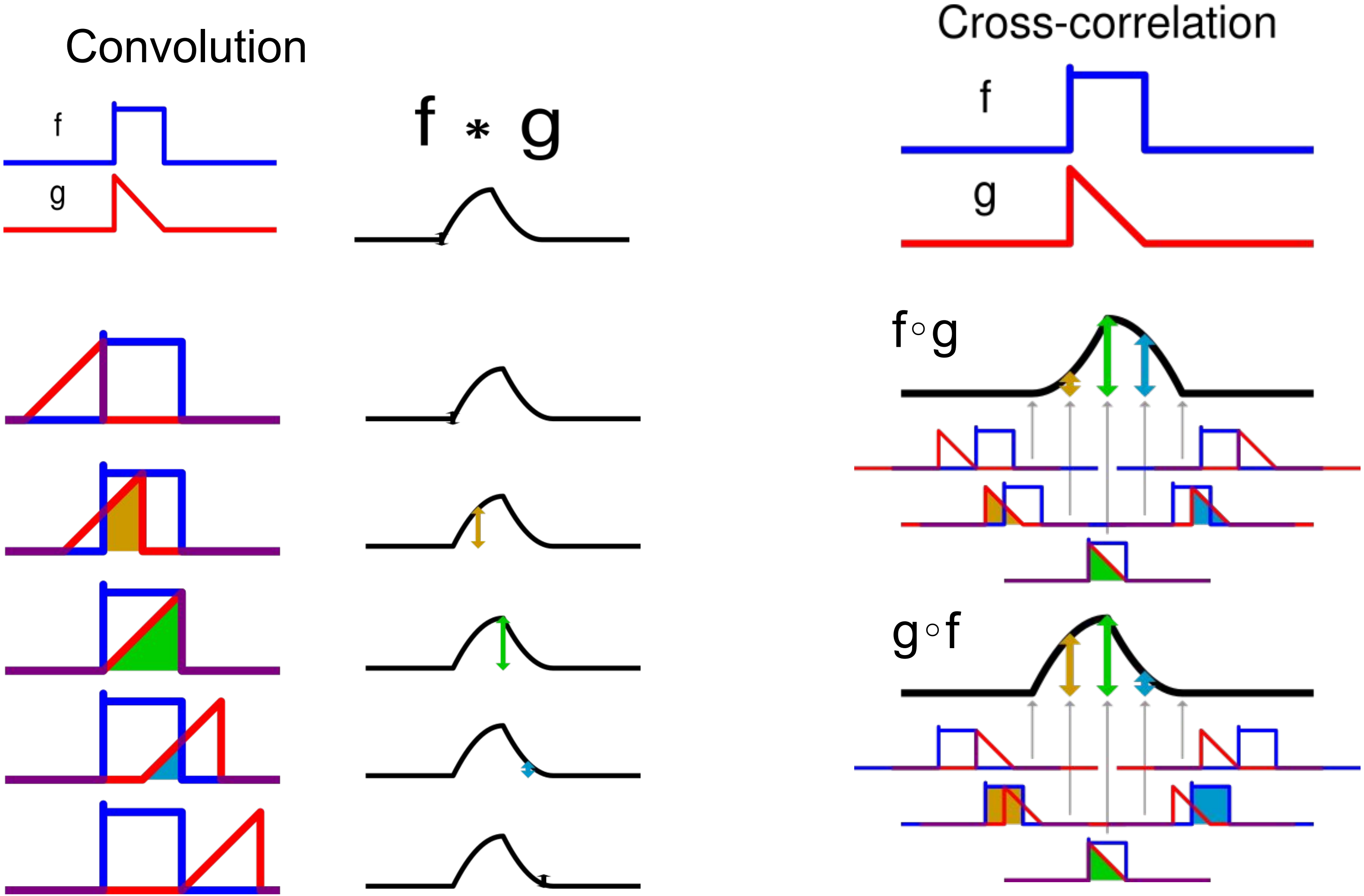
```
from scipy import optimize
x_data = np.linspace(-5,5,num = 50)
y_data = 2.9 * np.sin(1.5 * x_data) + np.random.normal(size = 50)
def test_func(x,a,b):
    return a*np.sin(b*x)

params, params_covariance = optimize.curve_fit(test_func, x_data, y_data)
print(params)
a,b = params
plt.plot(x_data, y_data, ".")
plt.plot(x_data, test_func(x_data, params[0], params[1]))
```



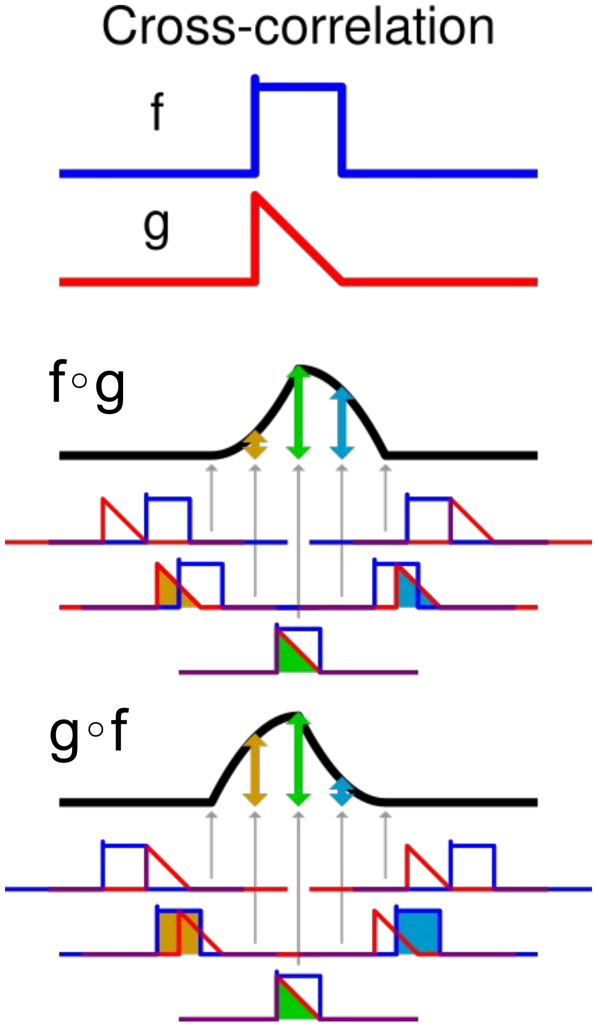
1. Sampling
2. Interpolation and Curve Fitting
3. Correlation and Convolution
    3.1.1. autocorrelation and interpretation
4. Time/Frequency domain and Fourier Series coefficients
    4.1.1. Periodic signal : Time <-> Frequency
    4.1.2. Fourier coefficient : Amplitude vs. Phase
5. DFT and Sampling rate
    5.1.1. Nyquist Frequency (higher bound)
    5.1.2. F(0) (lower bound)
6. Numerical FFT : scipy.fft
7. Time-Freq. analysis and filtering (exercice)

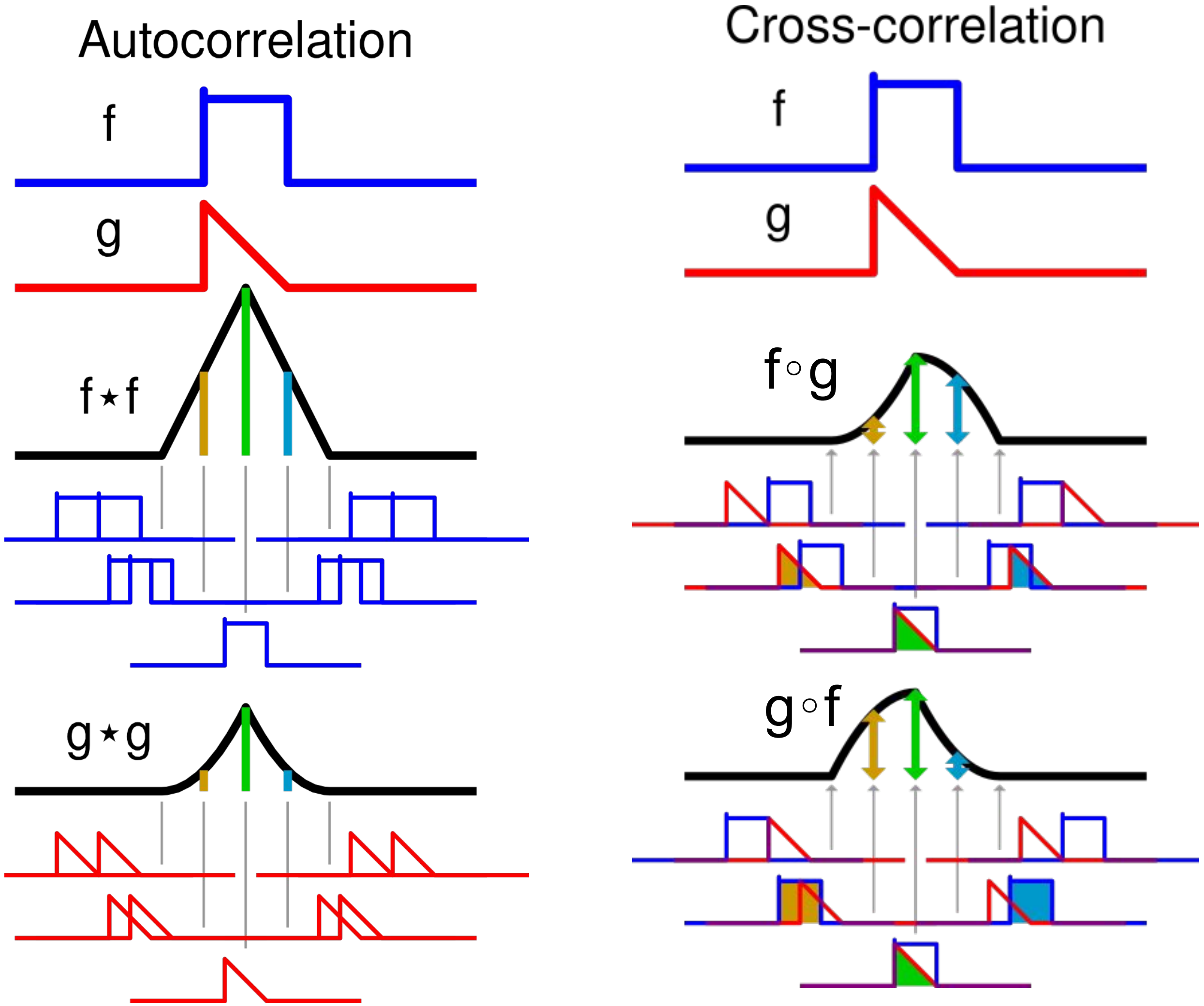
Convolution



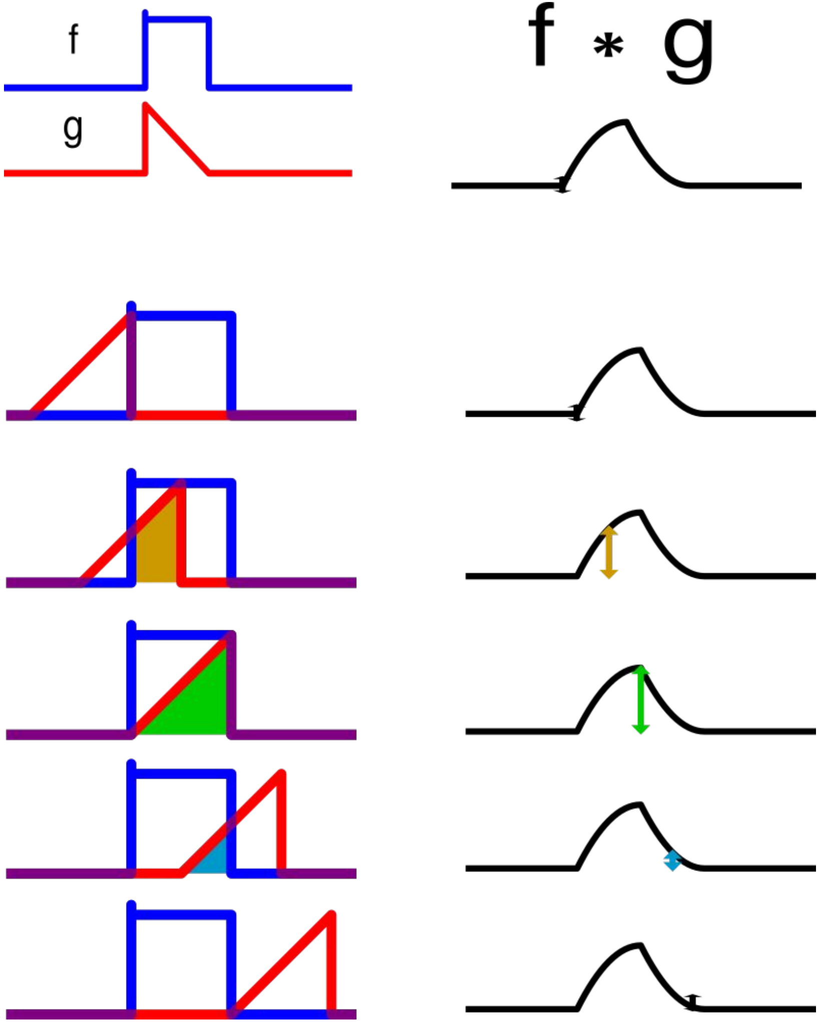

Convolution
f
g
f ∗ g
Cross-correlation
f
g
f∘g
g∘f

$$f \circ g(x) = \sum_{i=-N}^{N} f(i)g(x+i)$$





Autocorrelation
f
g
f⋆f
g⋆g
Cross-correlation
f
g
f∘g
g∘f



$$f * g(x) = \sum_{i=-N}^{N} f(i)g(x-i)$$

1.Sampling

2. Interpolation and Curve Fitting

3. Correlation and Convolution

    3.1.1. autocorrelation and interpretation

4. Time/Frequency domain and Fourier Series coefficients

    4.1.1. Periodic signal : Time <-> Frequency

    4.1.2. Fourier coefficient : Amplitude vs. Phase

5. DFT and Sampling rate

    5.1.1. Nyquist Frequency (higher bound)

    5.1.2. F(0) (lower bound)

6. Numerical FFT : scipy.fft

7. Time-Freq. analysis and filtering (exercice)

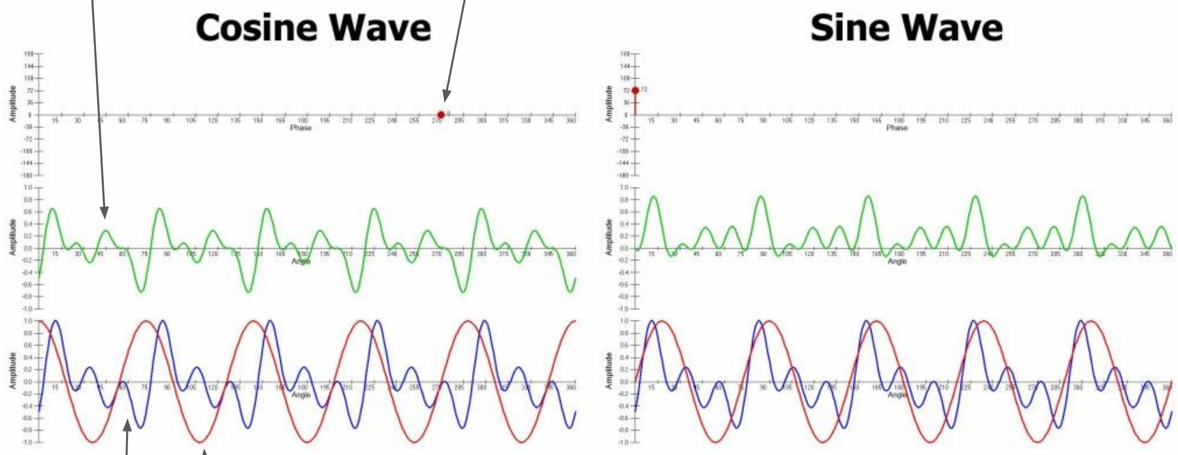
A Fourier transform takes functions back and forth between time and frequency domains.

*The coefficient at frequency **ω** are obtained using convolution with cos(ωt) and sin(ωt) signal*

$f(t) * \cos(\omega t) = \int f(t) \,.\, \cos(\omega)$
sum(area + area -)

f (t) . cos(ω)

$F(\omega) \sim f(t) * \cos(\omega t) - i\, f(t) * \sin(\omega t)$

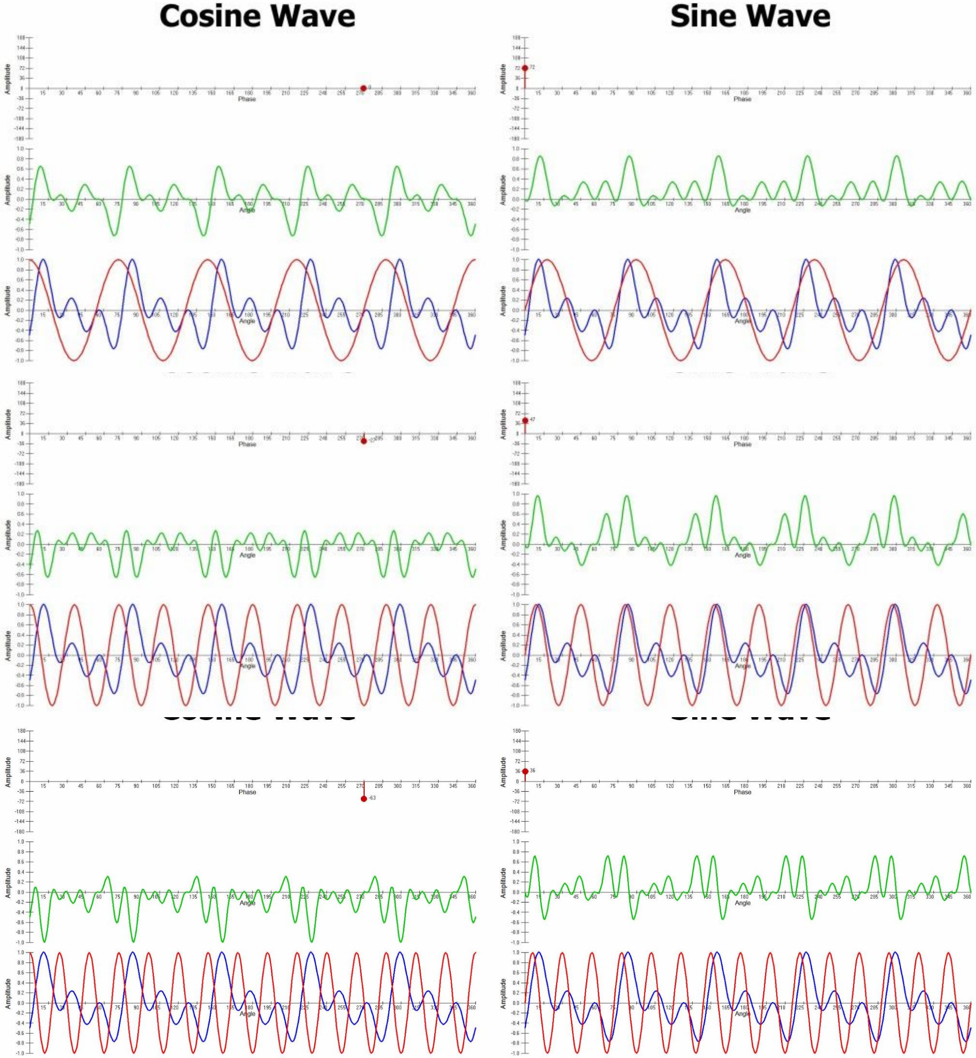


signal f (t)

cos(ωt)

There is an exact way to write the transformation with complex numbers :

$$F(\omega) = \int f(t)\, e^{-j\omega t}\, dt$$



ω=5Hz
no phase
pure sin wave

ω=10Hz

ω=15Hz

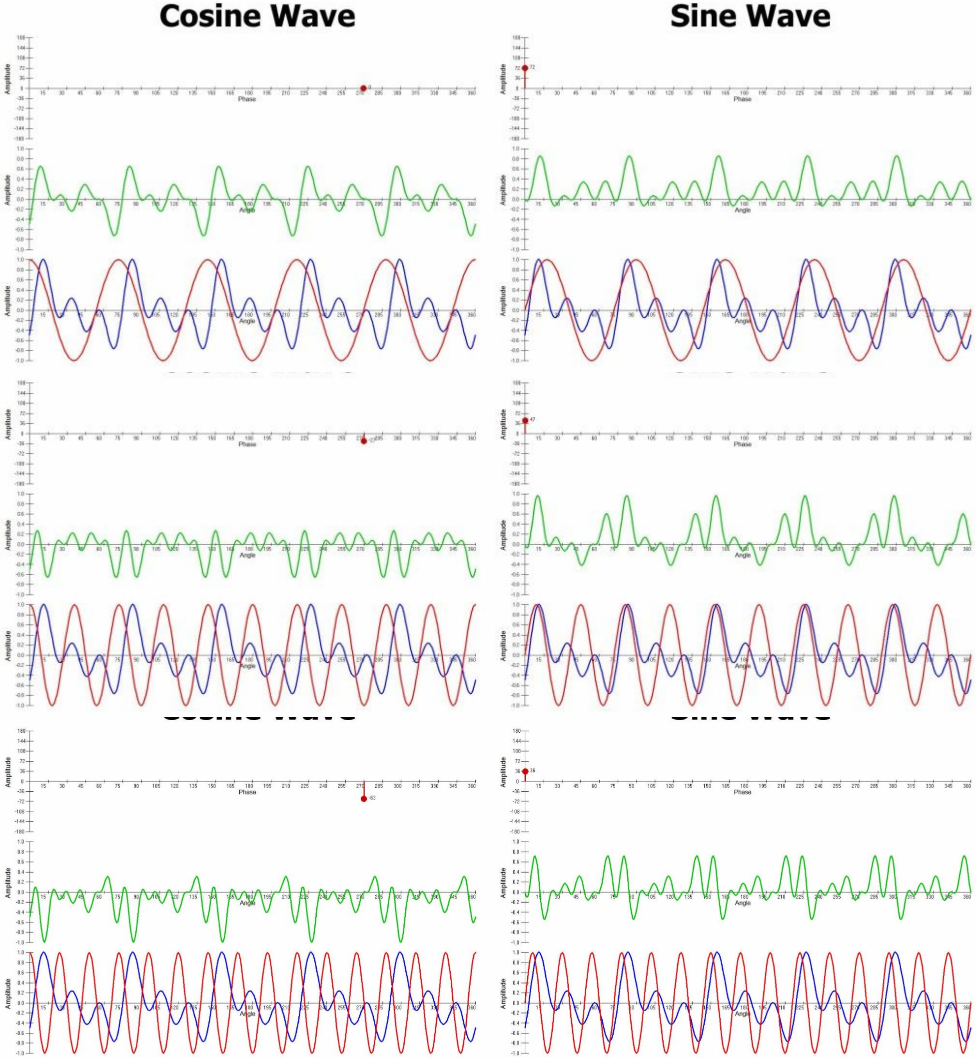
A Fourier transform takes functions back and forth between time and frequency domains.

The coefficient at frequency $\boldsymbol{\omega}$ *are obtained using convolution with cos(ωt) and sin(ωt) signal*

$$F(\omega) \sim f(t) * \cos(\omega t) - i\, f(t) * \sin(\omega t)$$

There is an exact way to write the transformation with complex numbers :

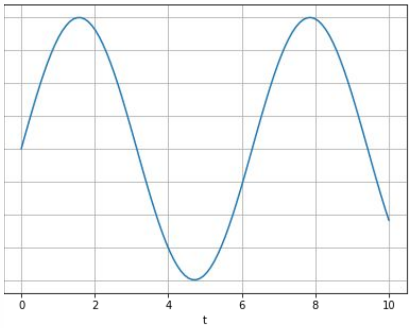
$$F(\omega) = \int f(t)\, e^{-j\omega t}\, dt$$

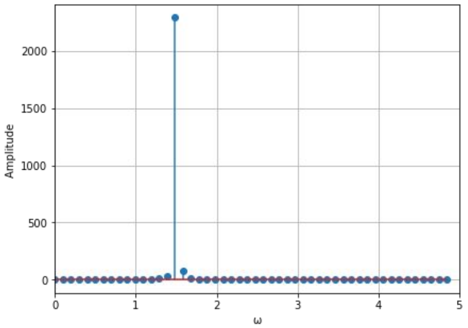


A discrete Fourier transform (DFT) operates on a signal represented as a finite dimensional vector.

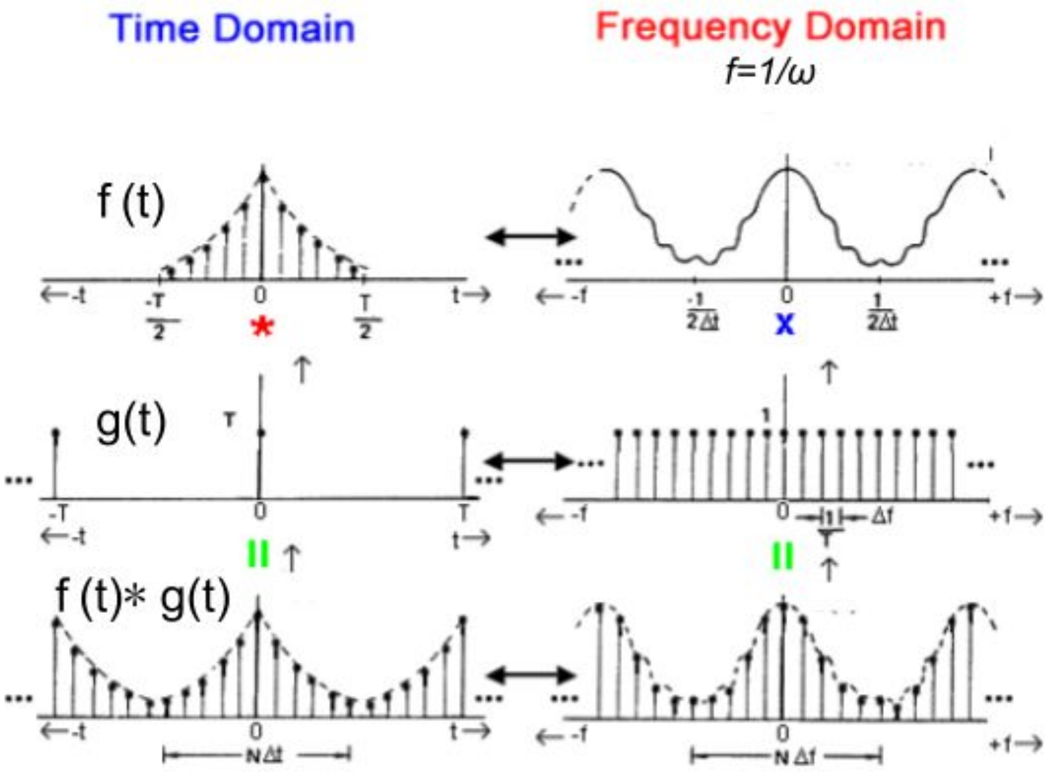
$$F_{\{k\}} = \frac{1}{N} \sum_{n=0}^{N-1} f_n e^{-2\pi i k n / N}$$

The Fast Fourier transform (FFT) is an algorithm that compute the discrete fourier coefficient on *N* frequencies



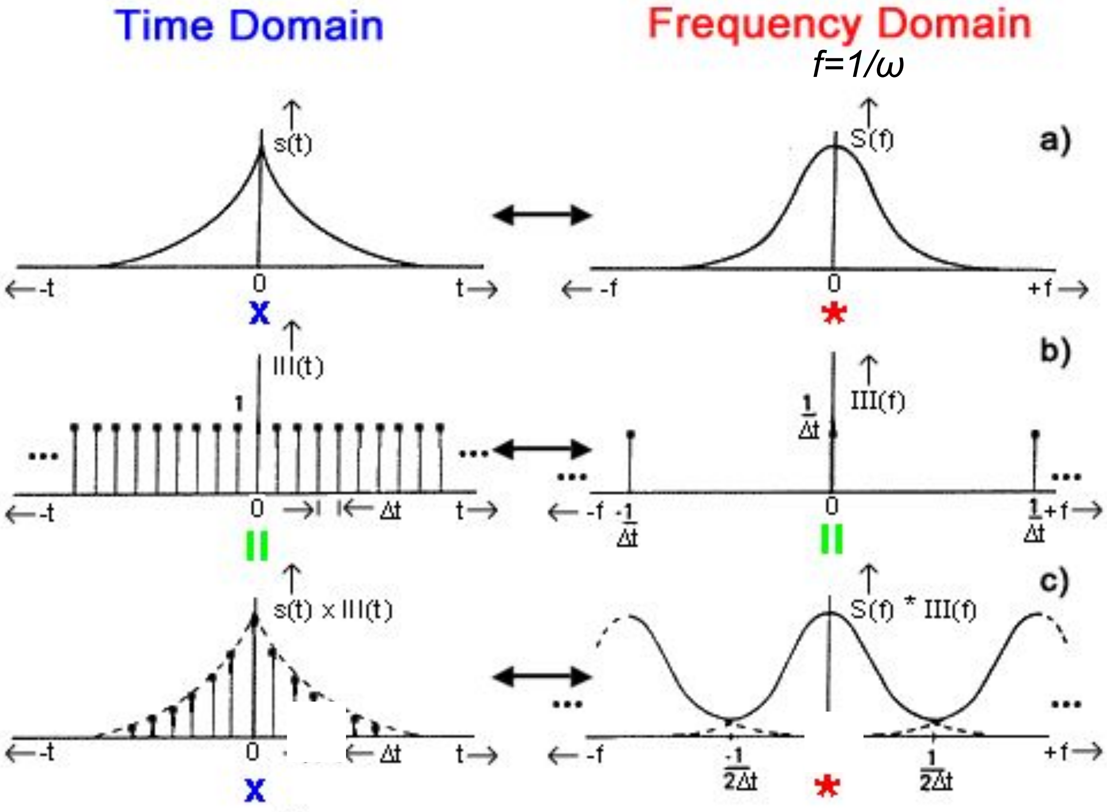


convolutions in the time domain become pointwise multiplication in the frequency domain.



$$f(t) * g(t) \longleftrightarrow \{F \cdot G\}(\omega)$$

pointwise multiplication in the time domain become convolution in the frequency domain.



$$f(t) \cdot g(t) \longleftrightarrow \{F * G\}(\omega)$$

1. Sampling
2. Interpolation and Curve Fitting
3. Correlation and Convolution
    3.1.1. autocorrelation and interpretation
4. Time/Frequency domain and Fourier Series coefficients
    4.1.1. Periodic signal : Time <-> Frequency
    4.1.2. Fourier coefficient : Amplitude vs. Phase
5. DFT and Sampling rate
    5.1.1. Nyquist Frequency (higher bound)
    5.1.2. F(0) (lower bound)
6. Numerical FFT : scipy.fft
7. Time-Freq. analysis and filtering (exercice)

scipy.fft returns the N coefficients $y[k]$ for k=0 to k=N-1

$$y[k] = \sum_{n=0}^{N-1} x[n] \times exp^{\left(-2j\pi\frac{k.n}{N}\right)}$$

The function **fftfreq** returns the N sample frequency points where the frequency f=k/n is found at $y[k]$.

Minimum frequency : y[0] ; f=0

$$y[0] = \sum_{n=0}^{N-1} x[n]$$

What is happening when the signal is centered on 0 ?

The frequency $f=k/n$ is found at $y[k]$

$$y[k] = \sum_{n=0}^{N-1} x[n] \times exp^{\left(-2j\pi\frac{k.n}{N}\right)}$$

The function **fftfreq** returns the N sample frequency points where the frequency $f=k/n$ is found at $y[k]$

Reminder from Sampling theory for a signal of period B :

A sufficient sample-rate is therefore **anything larger than 2B samples per second**.

Equivalently, for a given sample rate f_s , perfect reconstruction is guaranteed possible for a *band limit* $B < f_s / 2$

Maximum frequency : y[N] ; $B = f_s / 2$ (Nyquist frequency)