FONDAZIONE
BRUNO KESSLER

Digital Signal Processing

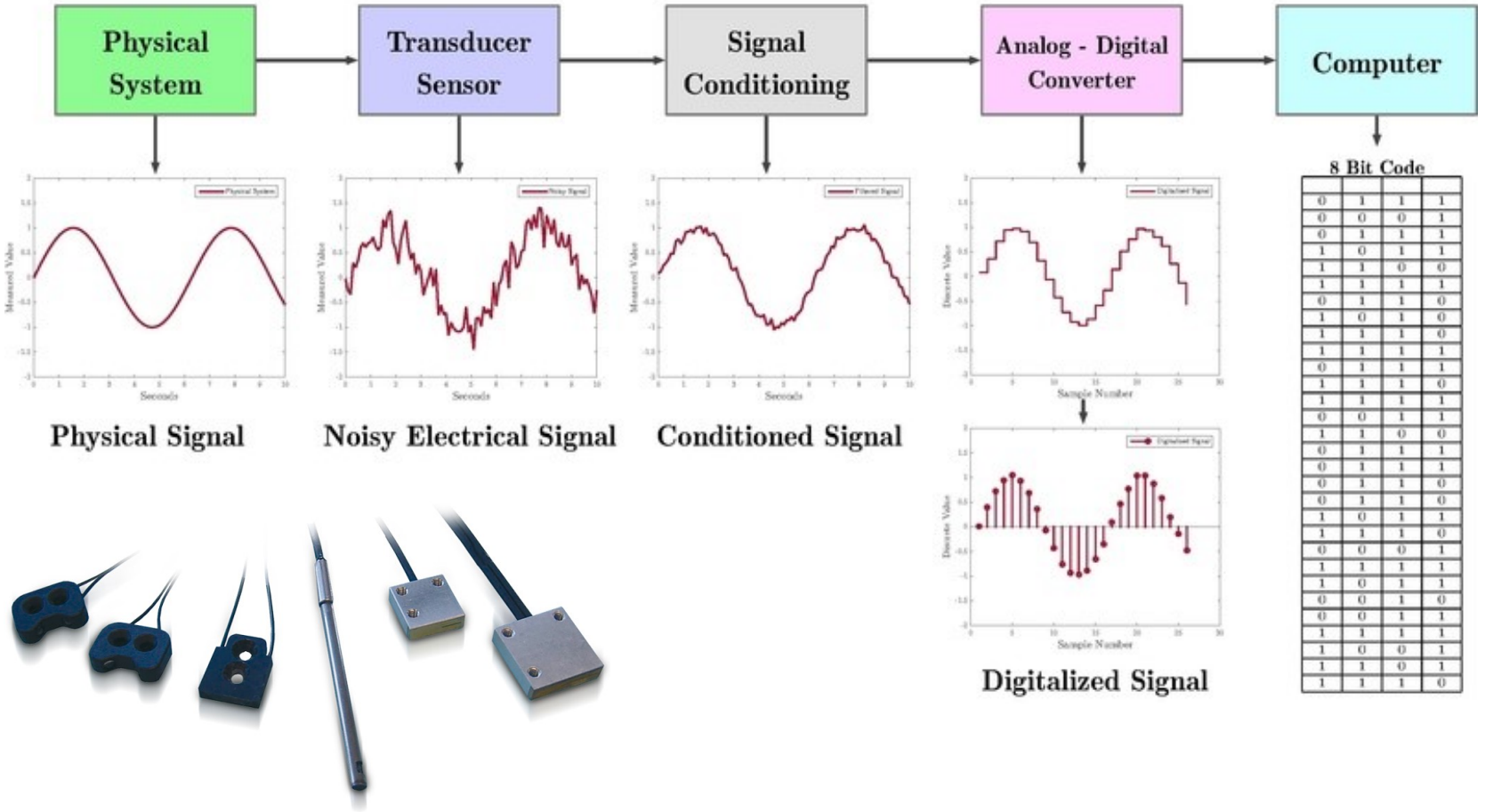
Fondazione Bruno Kessler

Piergiorgio Svaizer

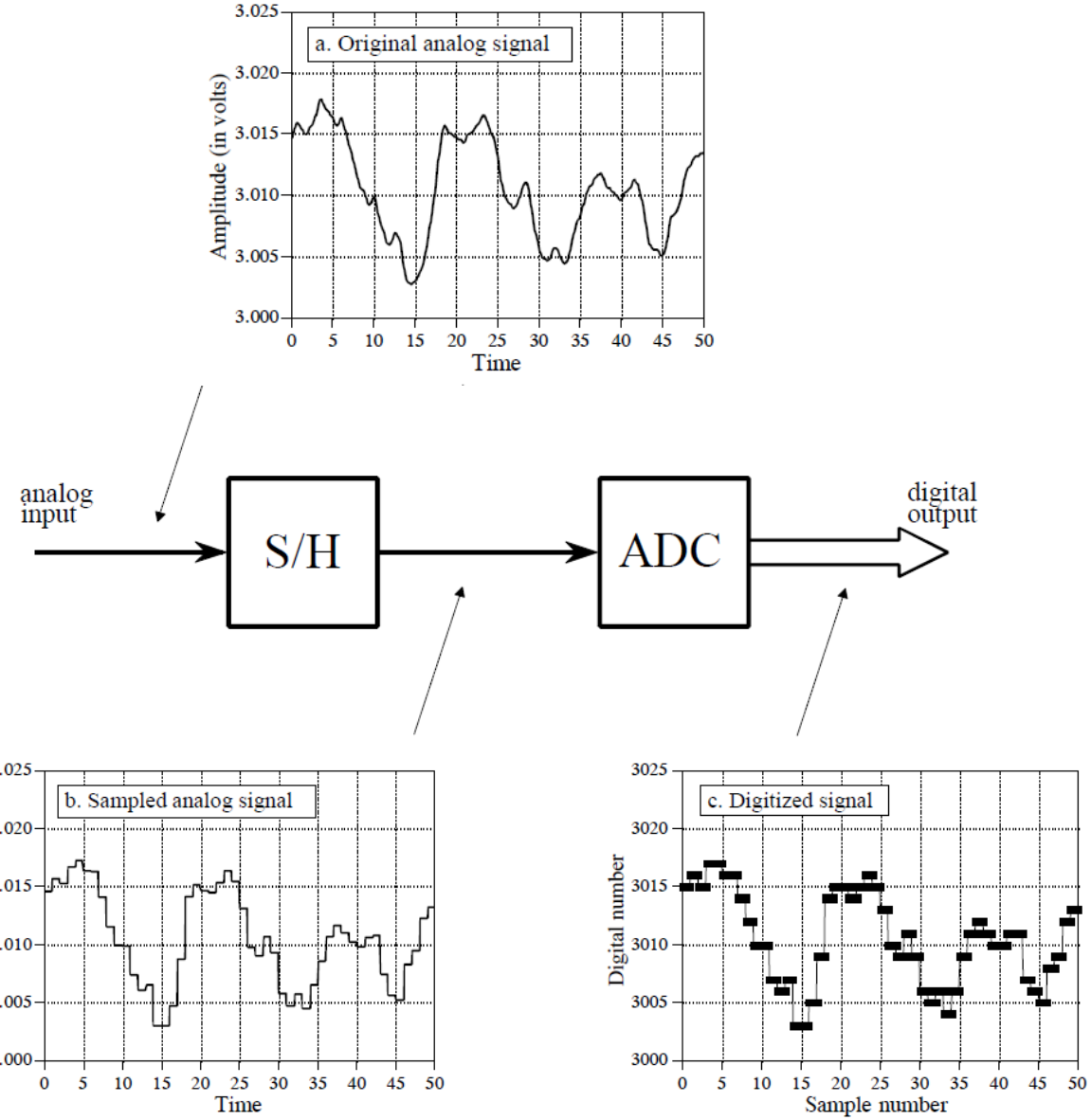
svaizer@fbk.eu

Digital Data Acquisition

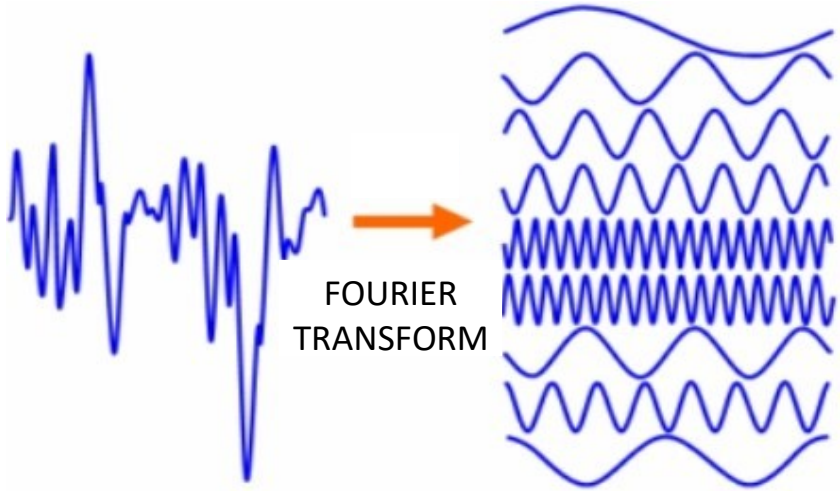
The first step in making any data analyzable (sound, images, physical signals,…) is to transform them into arrays of numbers



From analog signal to digital signal



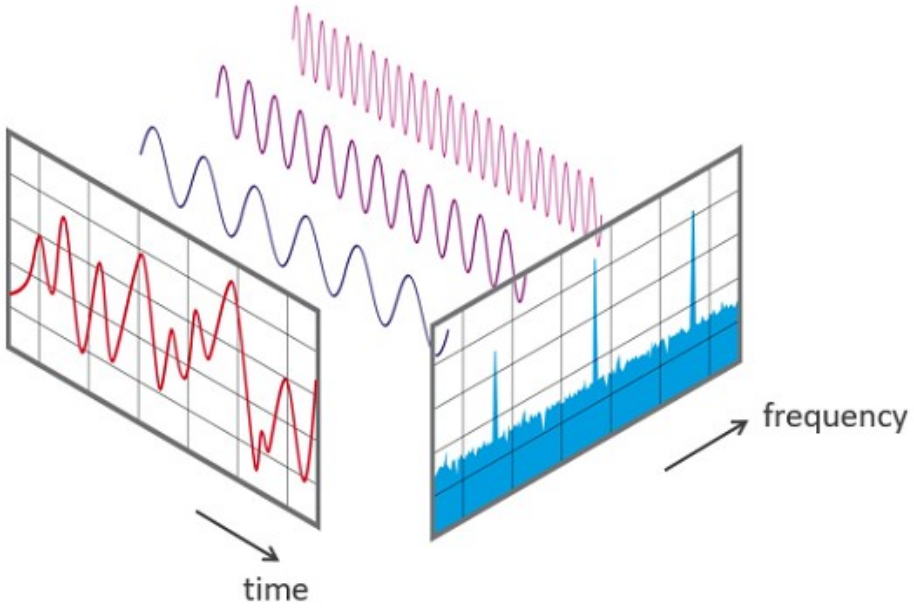
Time and frequency domains: The Fourier transform



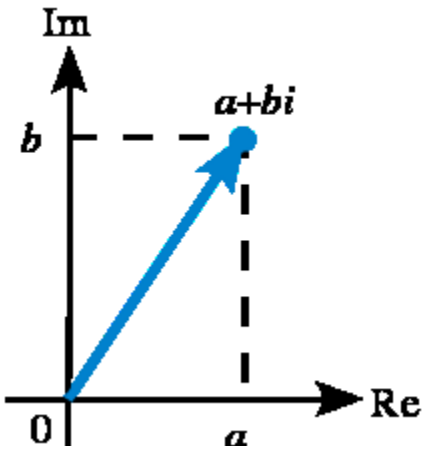
$$X(\omega)=\int_{-\infty}^{+\infty} x(t)e^{-j\omega t}dt$$

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

$$\omega=2\pi f$$

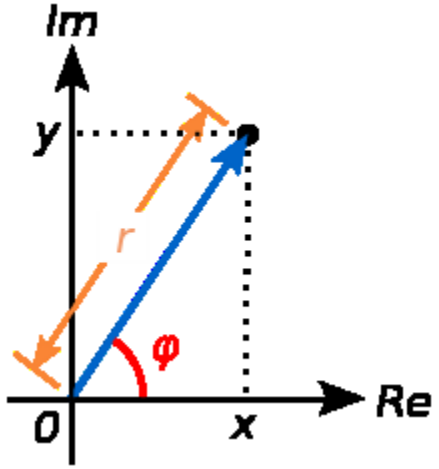


Complex numbers

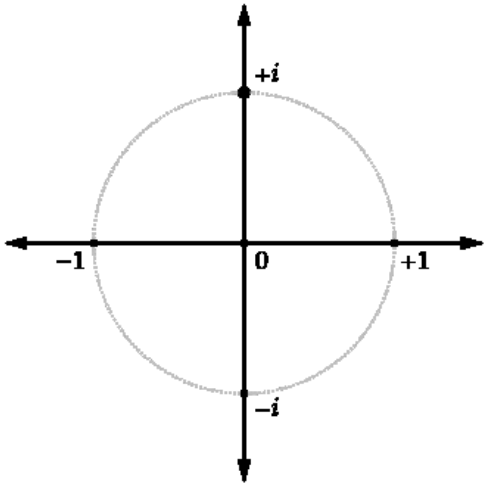


A **complex number** is a number that can be expressed in the form $a + bi$, where a and b are real numbers, and i is a solution of the equation $x^2 = -1$

They are used to express both amplitude and phase of a sinusoidal component in the spectrum of a signal



Multiplying a quantity by i corresponds to change its phase of 90°. Multiplying again by i produces an overall phase shift of 180°, i.e. a phase inversion. This explains the meaning of the equation $i^2 = -1$



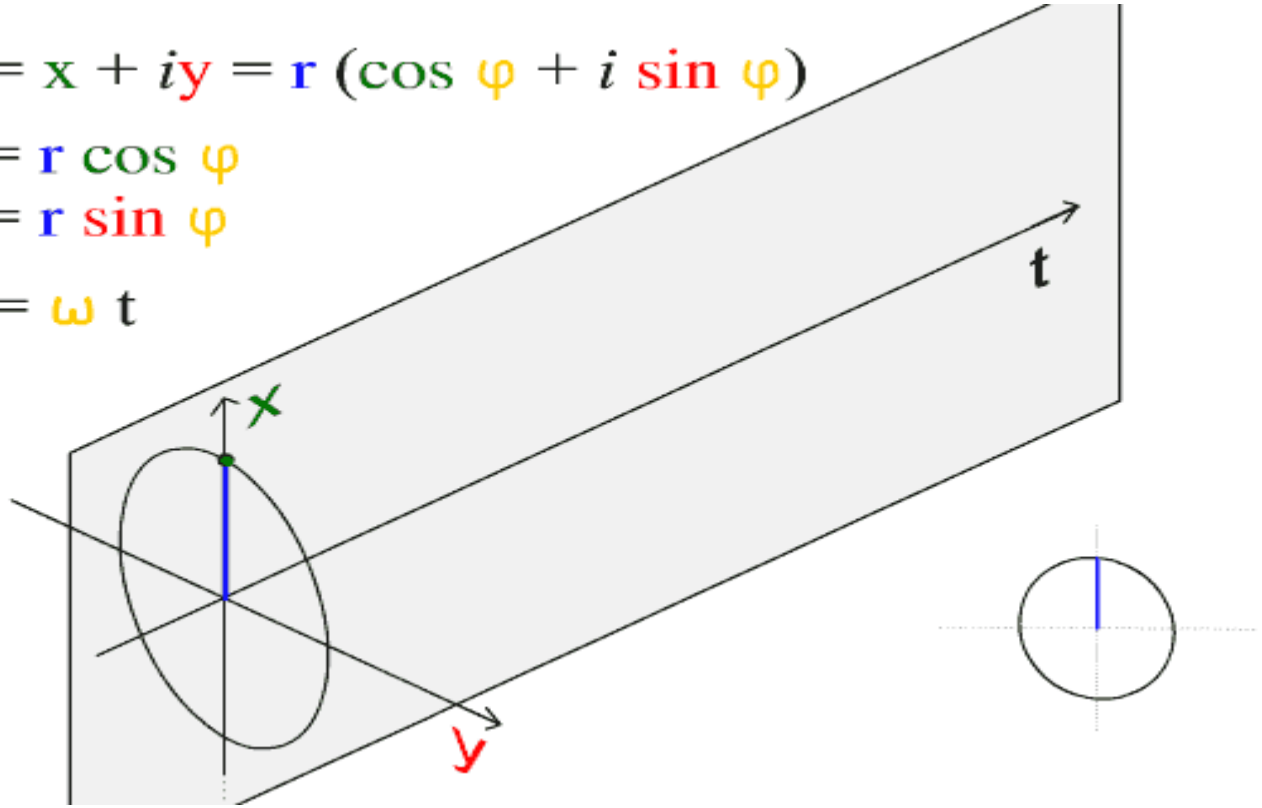
Sinusoidal components generated by a rotating vector

$$z = x + iy = r\,(\cos\varphi + i \sin\varphi)$$

$$x = r \cos\varphi$$

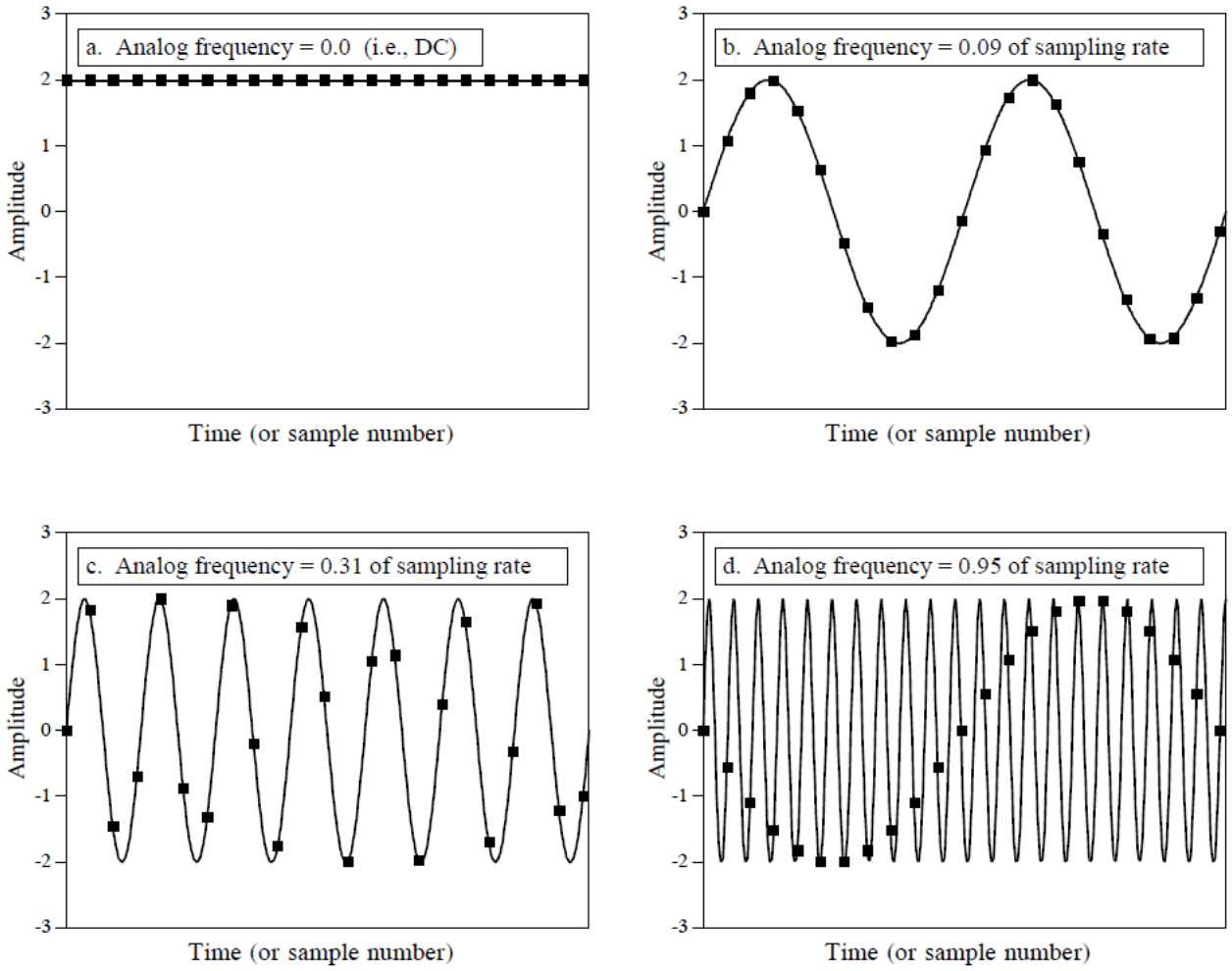
$$y = r \sin\varphi$$

$$\varphi = \omega t$$



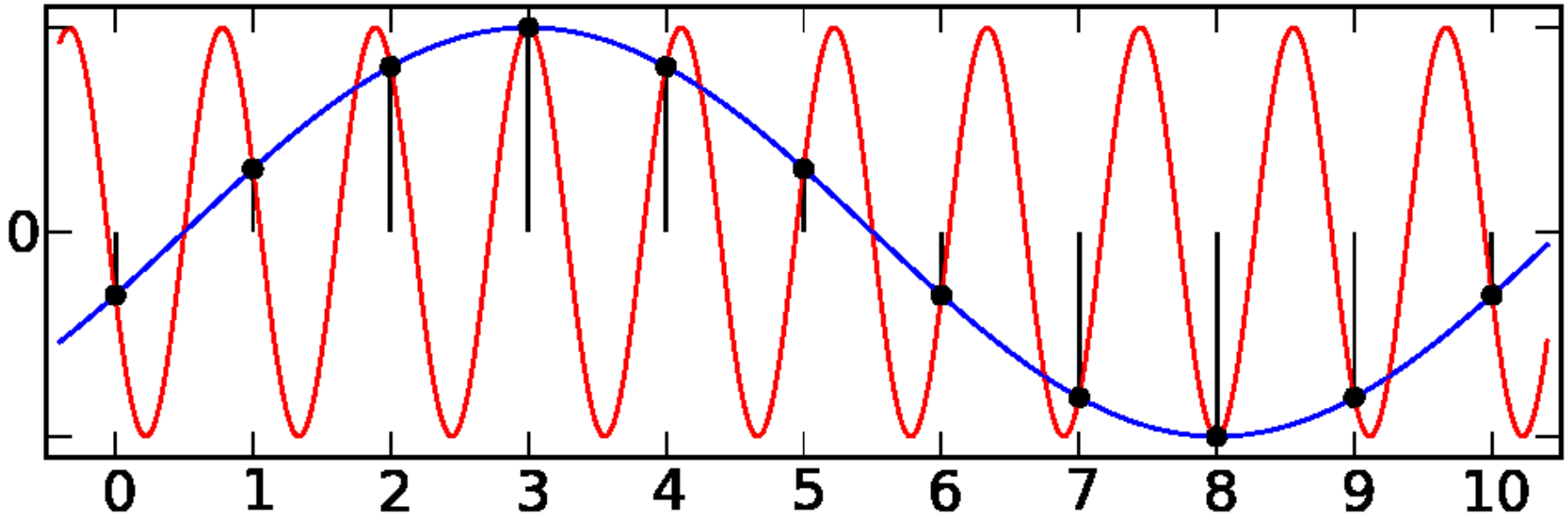
$\omega = 2\pi f$ f is the frequency in Hertz [Hz]

Sampling: from analog signal to discrete-time signal

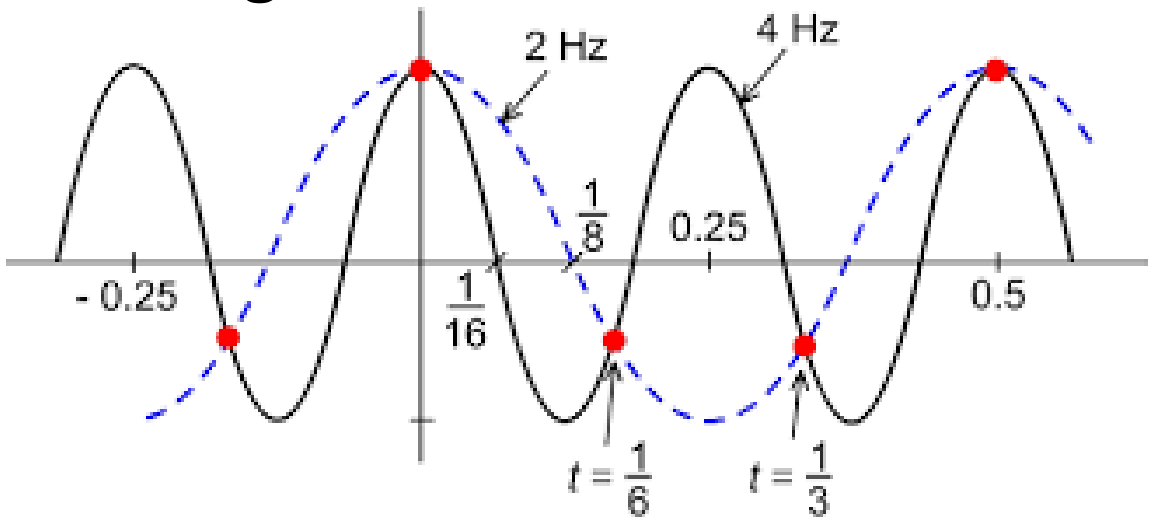


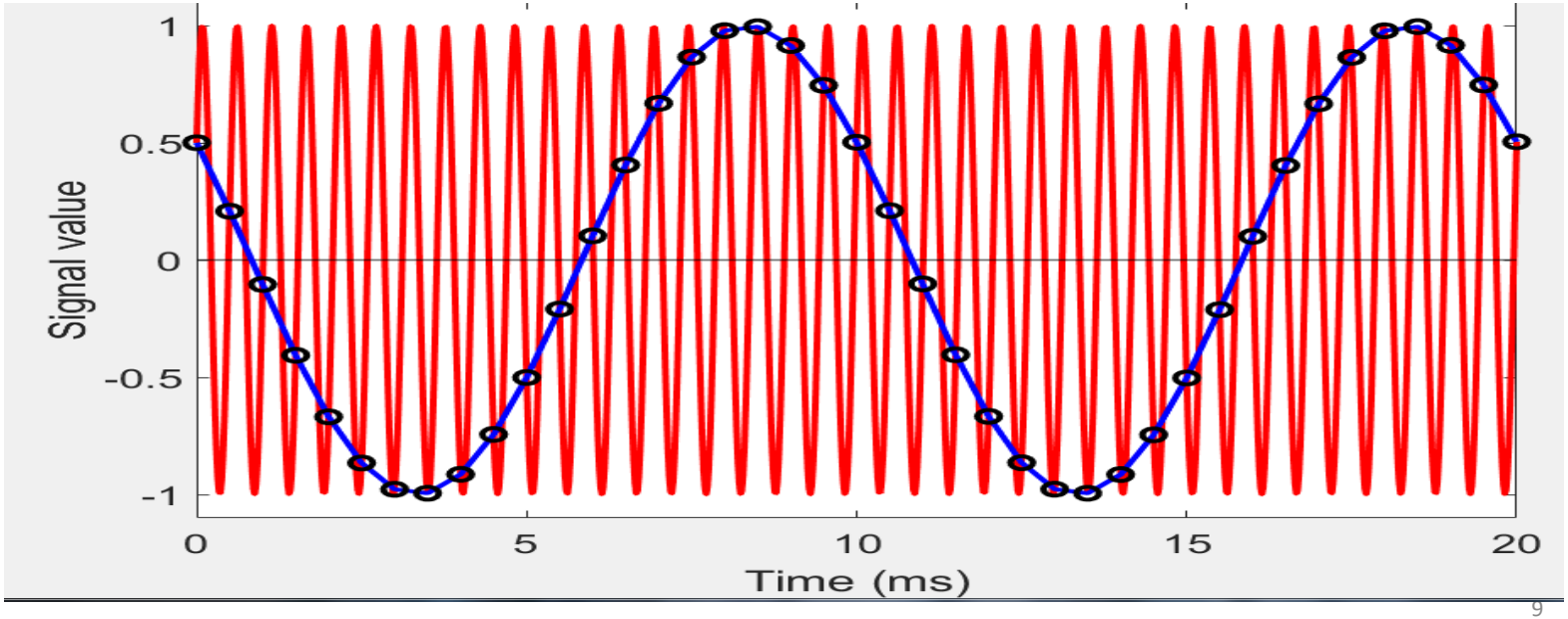
The Nyquist sampling theorem

A bandlimited continuous-time signal can be sampled and perfectly reconstructed from its samples if the waveform is sampled at least twice as fast as its highest frequency component

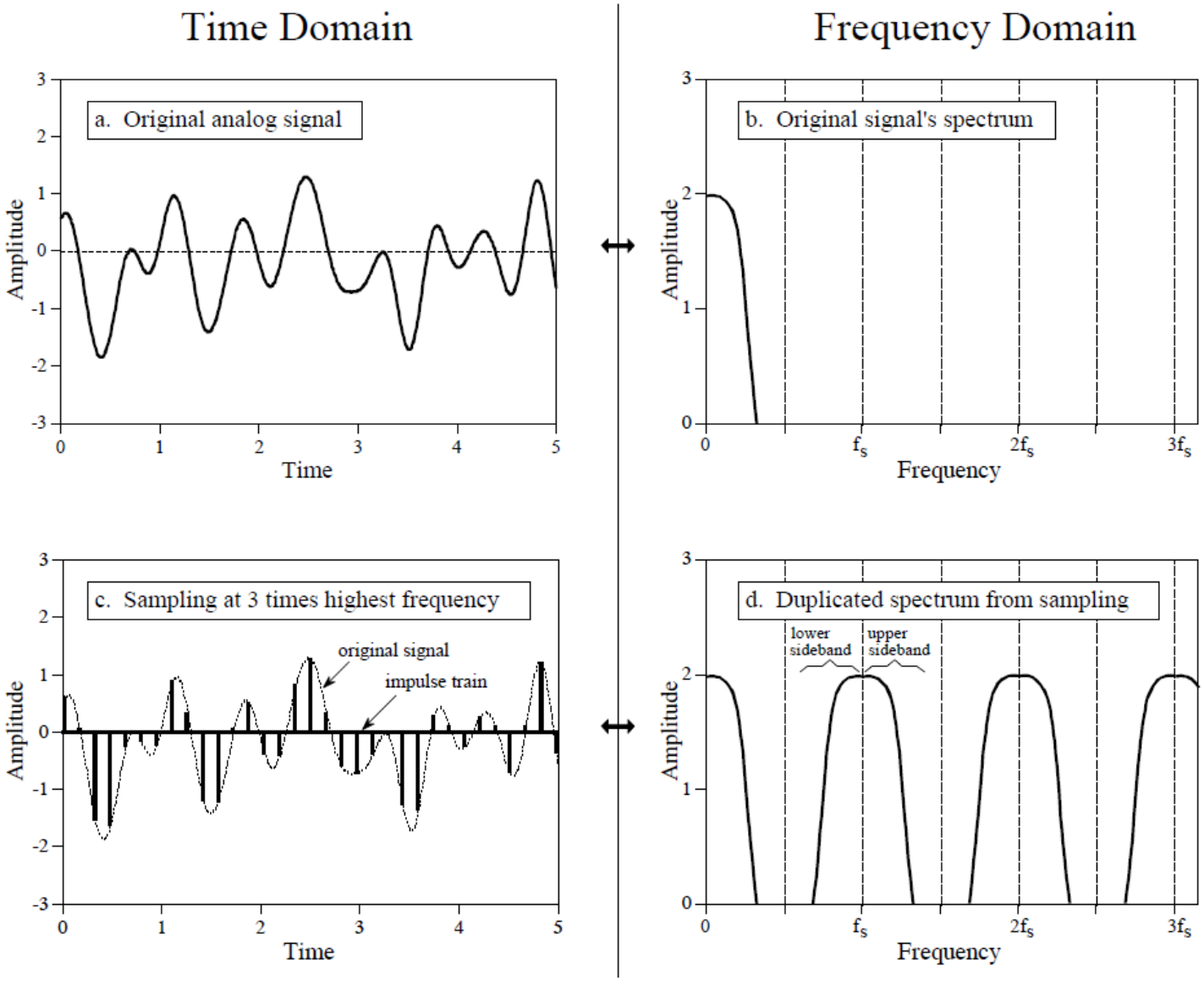


Examples of insufficient sampling rate: time aliasing

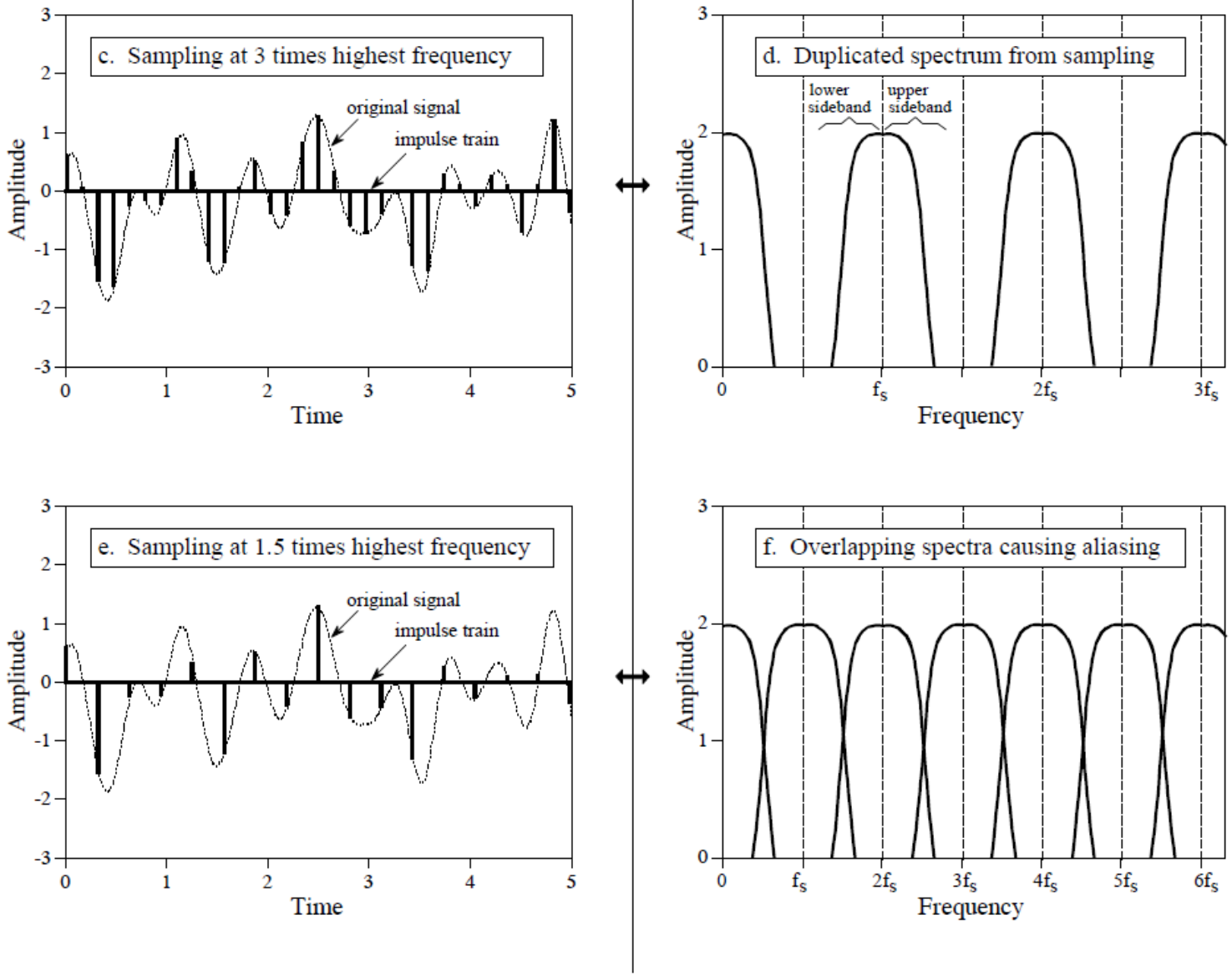




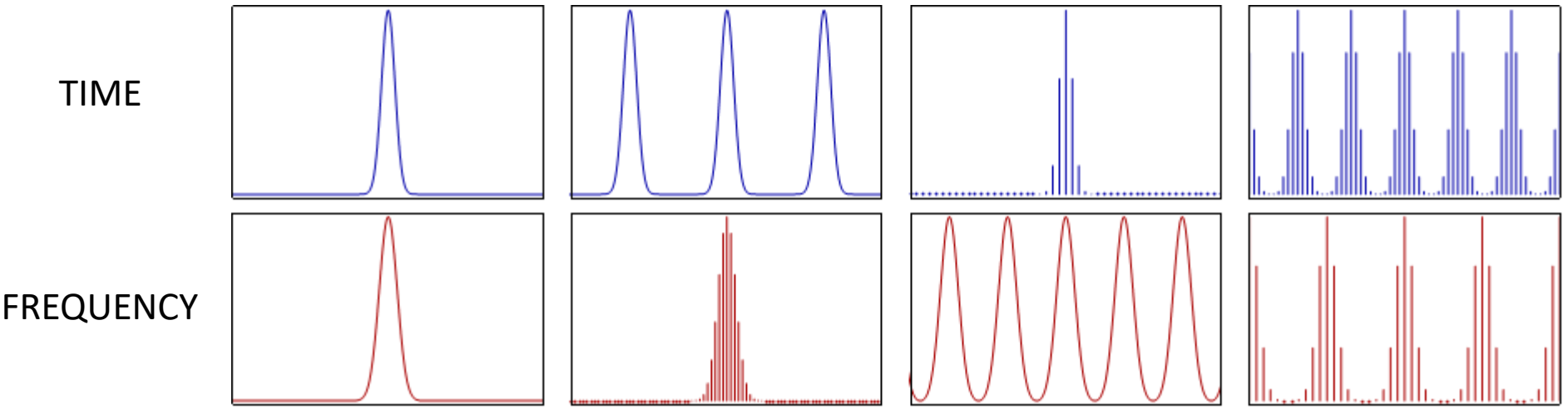
Effect of sampling: periodic repetition of the spectrum



Effect of sampling: periodic repetition of the spectrum



From continuous to discrete-time Fourier Transform



Sampling in one domain corresponds to **periodical replication** in the other domain.

The Discrete Fourier Transform (DFT) transforms a periodical discrete-time sequence into a periodical discrete-frequency sequence of Fourier coefficients

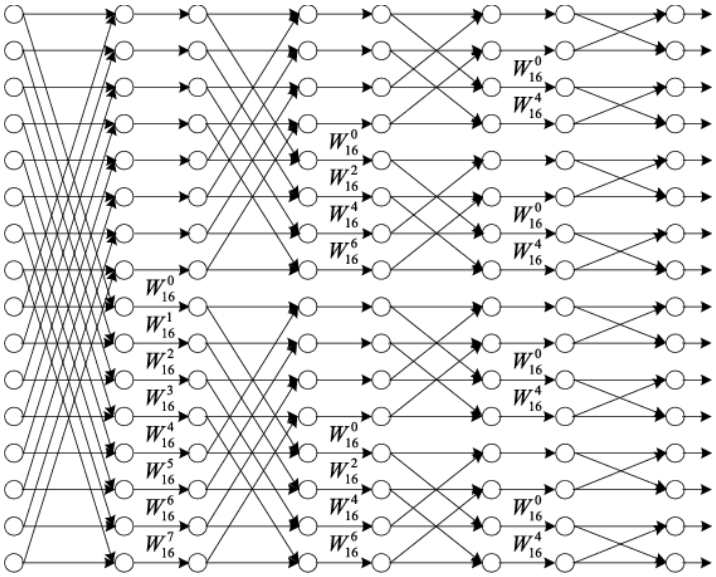
DFT: $$X_k = \sum_{n=0}^{N-1} x_n \cdot e^{-i2\pi kn/N}$$

INVERSE DFT: $$x_n = \frac{1}{N} \sum_{k=0}^{N-1} X_k \cdot e^{i2\pi kn/N}$$

The Fast Fourier Transform (FFT)

A **Fast Fourier Transform** (**FFT**) is an algorithm that computes the Discrete Fourier Transform (DFT) of a sequence, or its inverse (IDFT), in a very efficient way.

The FFT algorithms manage to reduce the complexity of computing the DFT from $O(N^2)$, which arises if one simply applies the definition of DFT, to $O(N \log N)$, where N is the data size.
The difference in speed can be enormous, especially for long data sets where *N* may be in the thousands or millions.



Using Python for data analysis and signal processing

NumPy: Numerical Python

NumPy provides an efficient way to store and manipulate multidimensional arrays in Python.

It provides a readable and efficient syntax for operating on array data, from simple element-wise arithmetic to more complicated linear algebraic operations.

```
In [1]:  import numpy as np
         x = np.arange(1, 10)
         x

Out [1]: array([1, 2, 3, 4, 5, 6, 7, 8, 9])

In [2]:  x ** 2

Out [2]: array([ 1,  4,  9, 16, 25, 36, 49, 64, 81])

In [4]:  M = x.reshape((3, 3))
         M

Out [4]: array([[1, 2, 3],
                [4, 5, 6],
                [7, 8, 9]])
```

```
In[17]: x[:5]  # first five elements
Out[17]: array([0, 1, 2, 3, 4])
In[18]: x[5:]  # elements after index 5
Out[18]: array([5, 6, 7, 8, 9])
In[19]: x[4:7]  # middle subarray
Out[19]: array([4, 5, 6])
```

Using Python for data analysis and signal processing

NumPy includes all common mathematical functions that can be applied to arrays: trigonometric, exponent and logarithmic, absolute value, scalar and matrix products, comparisons and boolean operators …

+	np.add	Addition (e.g., 1 + 1 = 2)
-	np.subtract	Subtraction (e.g., 3 - 2 = 1)
-	np.negative	Unary negation (e.g., -2)
*	np.multiply	Multiplication (e.g., 2 * 3 = 6)
/	np.divide	Division (e.g., 3 / 2 = 1.5)
//	np.floor_divide	Floor division (e.g., 3 // 2 = 1)
**	np.power	Exponentiation (e.g., 2 ** 3 = 8)
%	np.mod	Modulus/remainder (e.g., 9 % 4 = 1)

==	np.equal
!=	np.not_equal
<	np.less
<=	np.less_equal
>	np.greater
>=	np.greater_equal

np.sum	Compute sum of elements
np.prod	Compute product of elements
np.mean	Compute median of elements
np.std	Compute standard deviation
np.var	Compute variance
np.min	Find minimum value
np.max	Find maximum value
np.argmin	Find index of minimum value
np.argmax	Find index of maximum value
np.median	Compute median of elements
np.percentile	Compute rank-based statistics of elements
np.any	Evaluate whether any elements are true
np.all	Evaluate whether all elements are true

Using Python for data analysis and signal processing

Pandas: Labeled Column-Oriented Data

Pandas provides a labeled interface to multidimensional data, in the form of a DataFrame object, a tabular structure with functionalities for data indexing and manipulation

```
In [8]:
import pandas as pd
df = pd.DataFrame({'label': ['A', 'B', 'C', 'A', 'B', 'C'],
                   'value': [1, 2, 3, 4, 5, 6]})
df
```

```
Out [8]:    label value
         0  A     1
         1  B     2
         2  C     3
         3  A     4
         4  B     5
         5  C     6
```

```
In [9]:  df['label']
```

```
Out [9]: 0    A
         1    B
         2    C
         3    A
         4    B
         5    C
         Name: label, dtype: object
```

```
In [11]:  df['value'].sum()
```

```
Out [11]: 21
```

```
In [12]:  df.groupby('label').sum()
```

```
Out [12]:       value
         label
         A      5
         B      7
         C      9
```

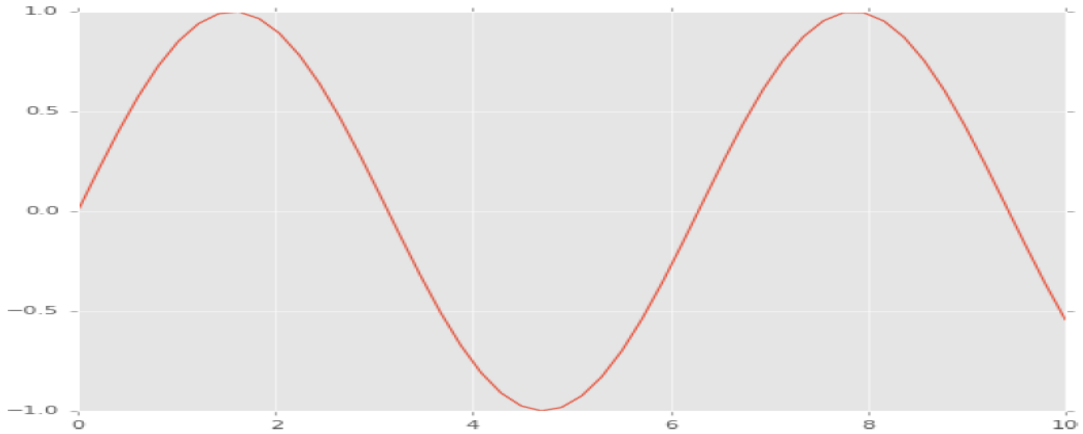
Using Python for data analysis and signal processing

Matplotlib: scientific data visualization

Matplotlib is a powerful library for creating a large range of plots.

```
In [14]:
import matplotlib.pyplot as plt
```

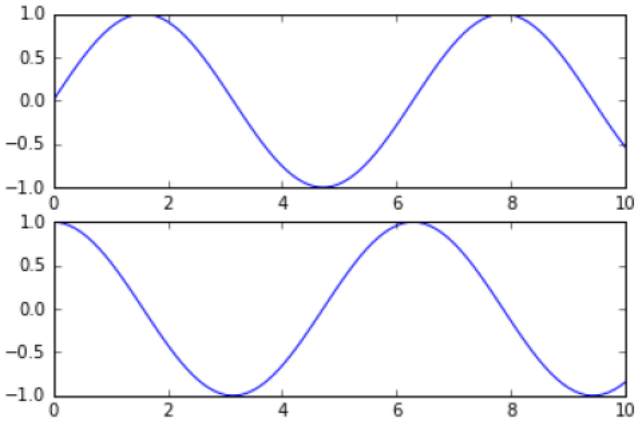
```
In [15]: x = np.linspace(0, 10)  # range of values from 0 to 10
         y = np.sin(x)           # sine of these values
         plt.plot(x, y);         # plot as a line
```



Using Python for data analysis and signal processing

Matplotlib: Multiple plots in one figure:

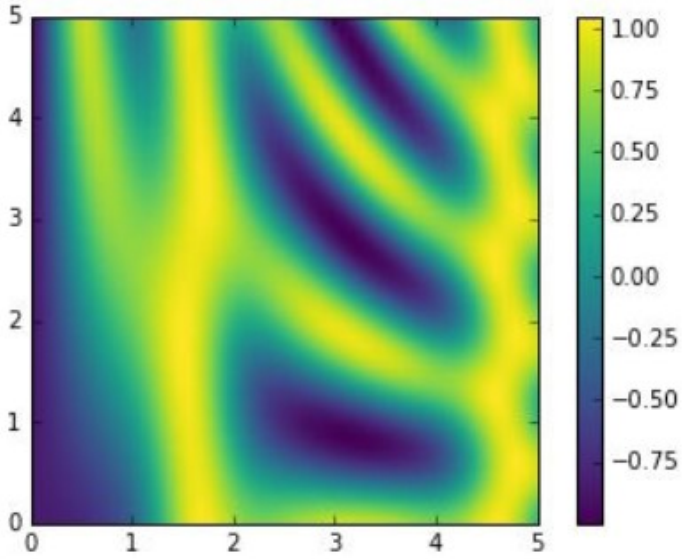
```
In[10]: # First create a grid of plots
        # ax will be an array of two Axes objects
        fig, ax = plt.subplots(2)

        # Call plot() method on the appropriate object
        ax[0].plot(x, np.sin(x))
        ax[1].plot(x, np.cos(x));
```



Matplotlib: Plotting a two-dimensional function:

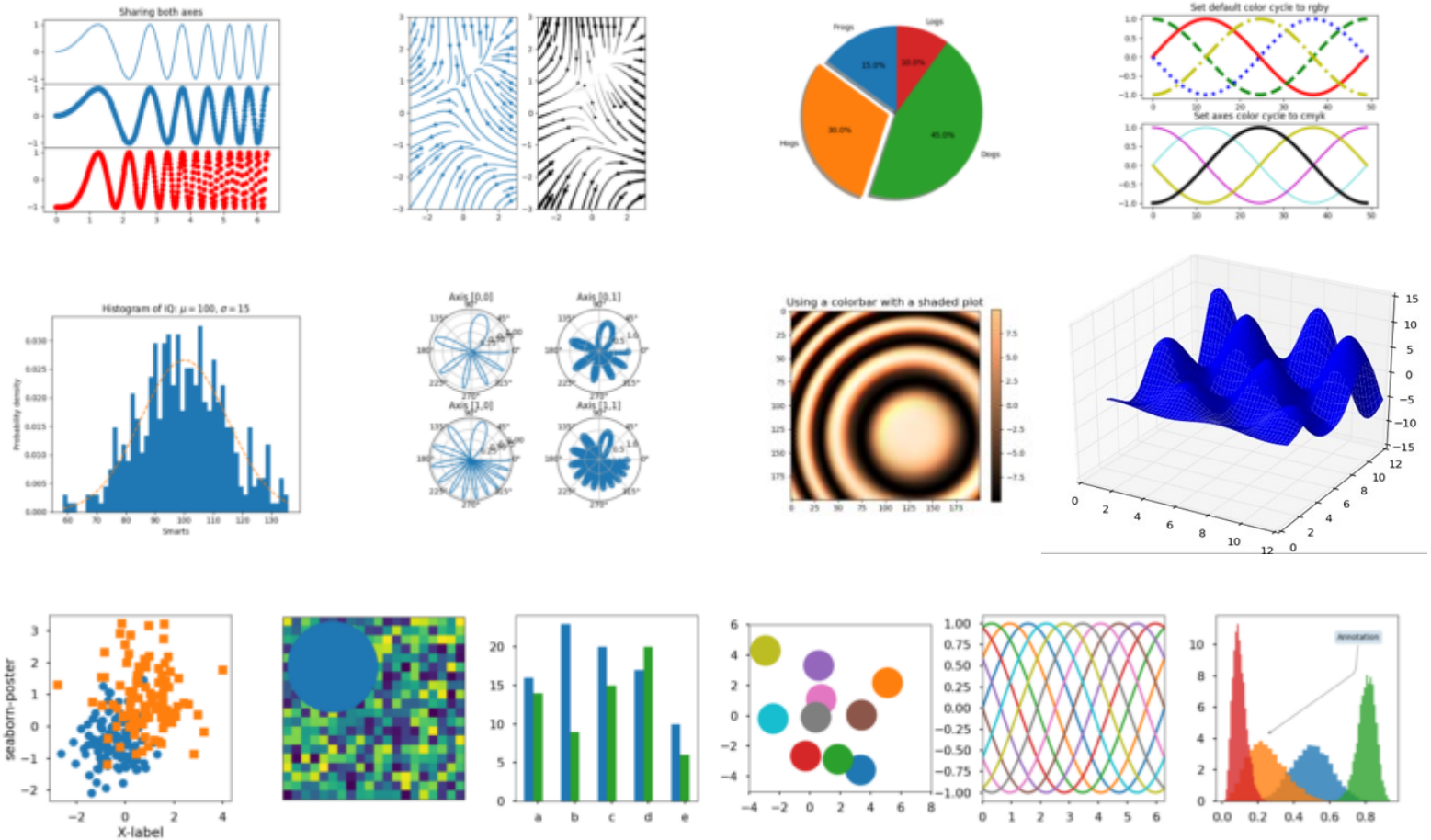
```
In[21]: # x and y have 50 steps from 0 to 5
        x = np.linspace(0, 5, 50)
        y = np.linspace(0, 5, 50)[:, np.newaxis]

        z = np.sin(x) ** 10 + np.cos(10 + y * x) * np.cos(x)
```

```
In[22]: %matplotlib inline
        import matplotlib.pyplot as plt

In[23]: plt.imshow(z, origin='lower', extent=[0, 5, 0, 5],
                   cmap='viridis')
        plt.colorbar();
```



Using Python for data analysis and signal processing

Matplotlib enables to produce a large variety of plots
(see a gallery of examples at https://matplotlib.org/gallery.html)



Using Python for data analysis and signal processing

SciPy: Scientific Python

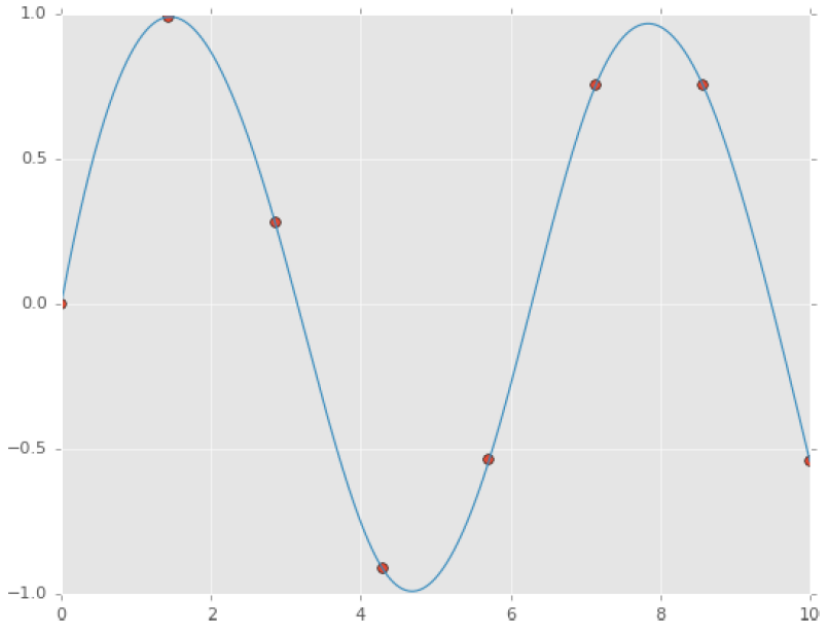
SciPy is a collection of scientific functionality that is built on NumPy.

The package is arranged as a set of submodules, each implementing some class of numerical algorithms.

`scipy.fftpack`	Fast Fourier transforms
`scipy.integrate`	Numerical integration
`scipy.interpolate`	Numerical interpolation
`scipy.linalg`	Linear algebra routines
`scipy.optimize`	Numerical optimization of functions
`scipy.sparse`	Sparse matrix storage and linear algebra
`scipy.stats`	Statistical analysis routines

For example:

```
In [16]: from scipy import interpolate

         # choose eight points between 0 and 10
         x = np.linspace(0, 10, 8)
         y = np.sin(x)

         # create a cubic interpolation function
         func = interpolate.interp1d(x, y, kind='cubic')

         # interpolate on a grid of 1,000 points
         x_interp = np.linspace(0, 10, 1000)
         y_interp = func(x_interp)

         # plot the results
         plt.figure()  # new figure
         plt.plot(x, y, 'o')
         plt.plot(x_interp, y_interp);
```



scipy.signal: Signal Processing in Python

The signal processing toolbox contains functions for spectral analysis, filtering and filter design.

Filtering

Filtering is a generic name for any system that modifies an input signal in some way. In SciPy a signal can be thought of as a NumPy array. There are different kinds of filters for different kinds of operations.

Convolution / correlation

Convolution and correlation are operations that involve one or more input signals and correspond to the effects produced by linear systems on their input signals

Spectral analysis

Spectral analysis includes techniques to obtain representations of signals in the frequency domain, in order to study the contributions of the single periodic components

Filter design

It includes the methods to design suitable filters in order to produce the desired modifications on the input signal

Save and load data in Python

```
a = np.array([1, 2, 3, 4])
np.save("array_file.npy", a)
b = np.load("array_file.npy")
print(b)
```

```
[1 2 3 4]
```

```
b *= 6
np.savez("dict_array.npz", ar1=a, ar2=b, ar3=a+b)
data = np.load("dict_array.npz")
print(data['ar1'])
print(data['ar2'])
print(data['ar3'])
```

```
[1 2 3 4]
[ 6 12 18 24]
[ 7 14 21 28]
```

```
np.savetxt("text_array.txt", data['ar3'])
c = np.loadtxt("text_array.txt")
print(c)
```

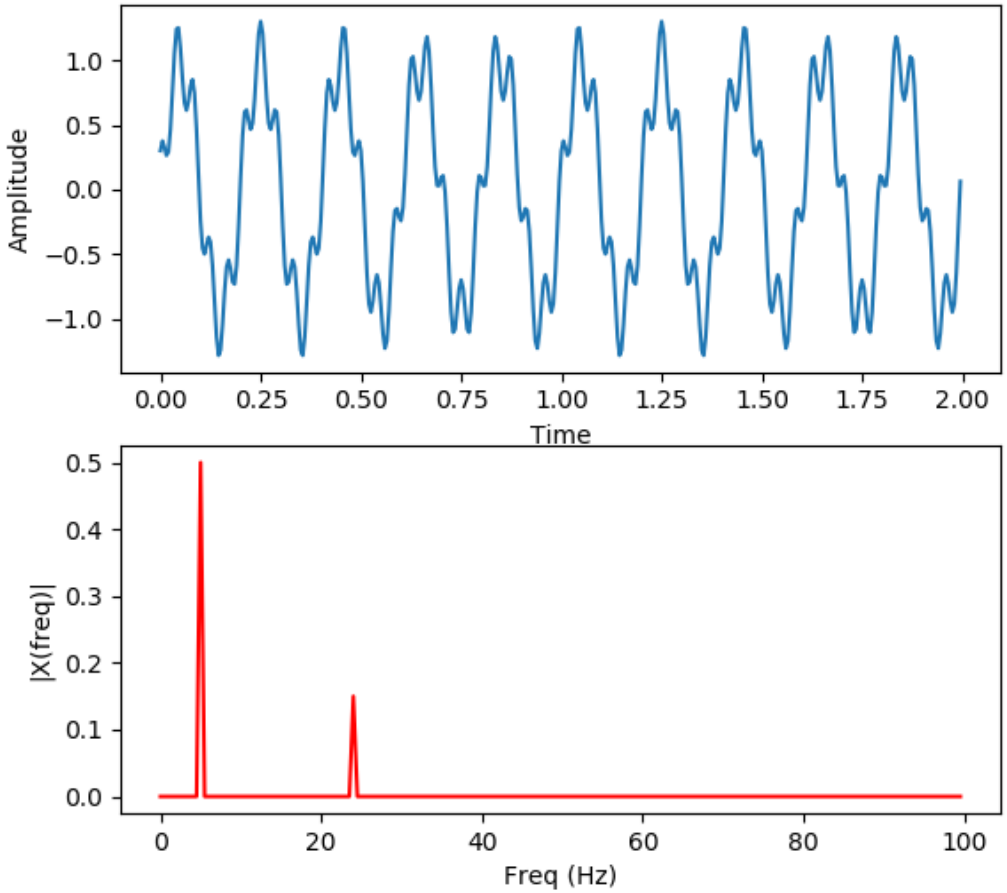
```
[  7.  14.  21.  28.]
```

To save any Python object structure to disk: use the Python module **pickle**
To load data of various format, many specific Python module exist.

FONDAZIONE
BRUNO KESSLER

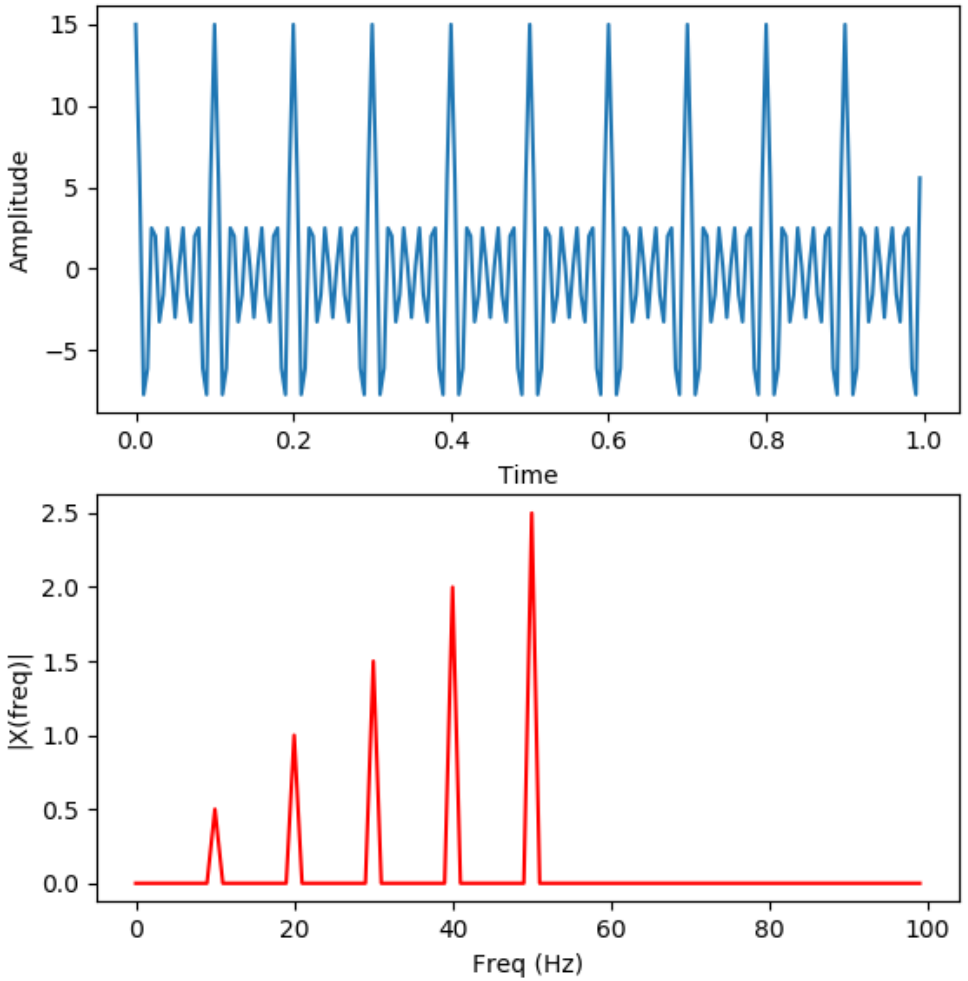
Example of spectrum computation with FFT in Python

```
import matplotlib.pyplot as plt
import numpy as np

Fs = 200.0;  # sampling rate
Ts = 1.0/Fs; # sampling interval
t = np.arange(0,2,Ts) # time vector

f1 = 5;   # frequency of the first component
f2 = 24;
x = np.sin(2*np.pi*f1*t)+0.3*np.cos(2*np.pi*f2*t)

N = len(x) # length of the signal
k = np.arange(N)

frq = Fs*k/N # two sides frequency range

X = np.fft.fft(x)/N # fft computing and normalization

fig, ax = plt.subplots(2, 1)
ax[0].plot(t,x)
ax[0].set_xlabel('Time')
ax[0].set_ylabel('Amplitude')

# plotting the spectrum:
ax[1].plot(frq[range(N/2)],abs(X[range(N/2)]),'r')
ax[1].set_xlabel('Freq (Hz)')
ax[1].set_ylabel('|X(freq)|')
```

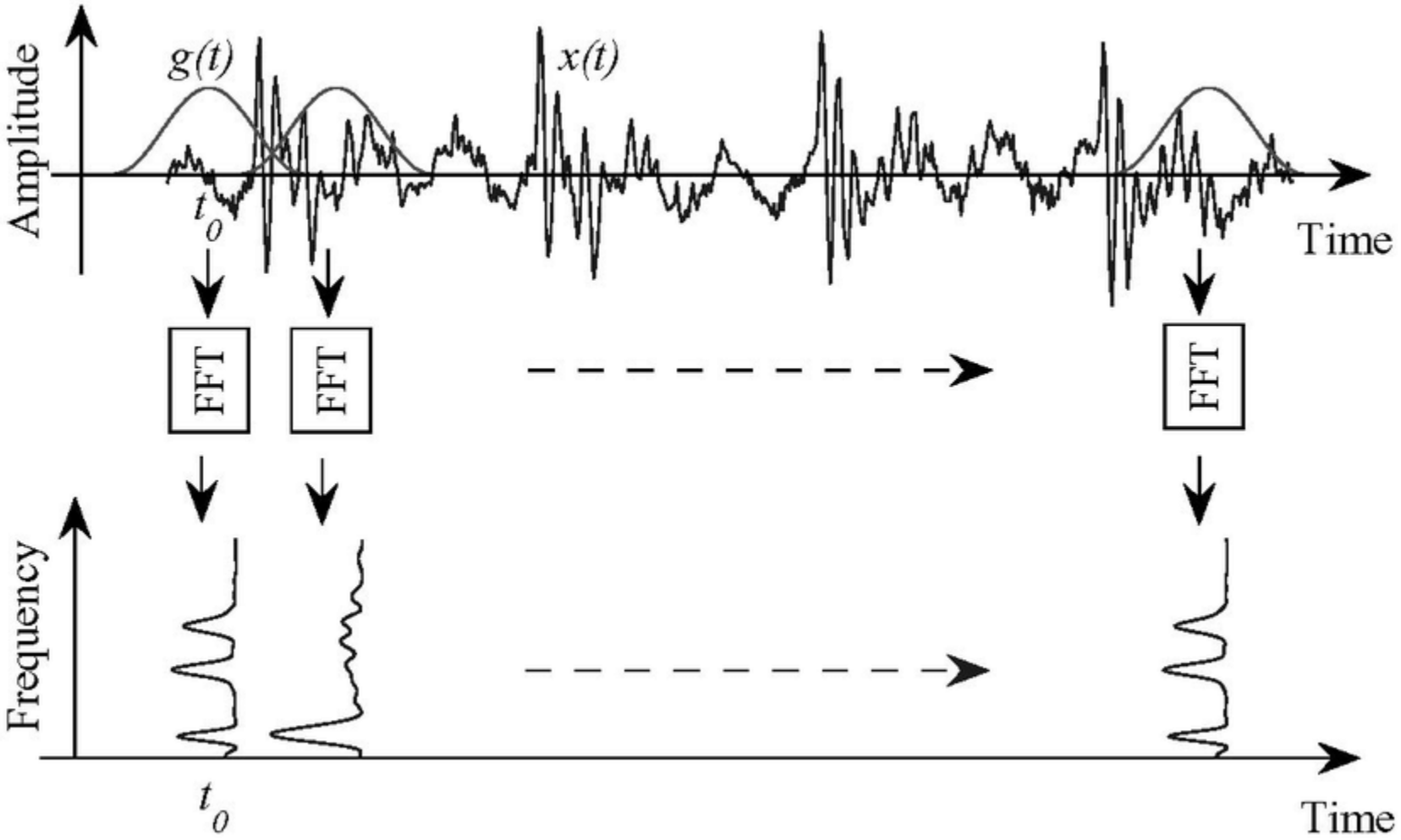


Example of spectrum computation with FFT in Python

```
import matplotlib.pyplot as plt
import numpy as np

Fs = 200.0;  # sampling rate
Ts = 1.0/Fs; # sampling interval
t = np.arange(0,1,Ts) # time vector

f1 = 10;   # frequency of the first component
x = np.cos(2*np.pi*f1*t) \
    +2*np.cos(2*np.pi*2*f1*t) \
    +3*np.cos(2*np.pi*3*f1*t) \
    +4*np.cos(2*np.pi*4*f1*t) \
    +5*np.cos(2*np.pi*5*f1*t)

N = len(x) # length of the signal
k = np.arange(N)

frq = Fs*k/N # two sides frequency range

X = np.fft.fft(x)/N # fft computing and normalization

fig, ax = plt.subplots(2, 1)
ax[0].plot(t,x)
ax[0].set_xlabel('Time')
ax[0].set_ylabel('Amplitude')

# plotting the spectrum:
ax[1].plot(frq[range(N/2)],abs(X[range(N/2)]),'r')
ax[1].set_xlabel('Freq (Hz)')
ax[1].set_ylabel('|X(freq)|')
```

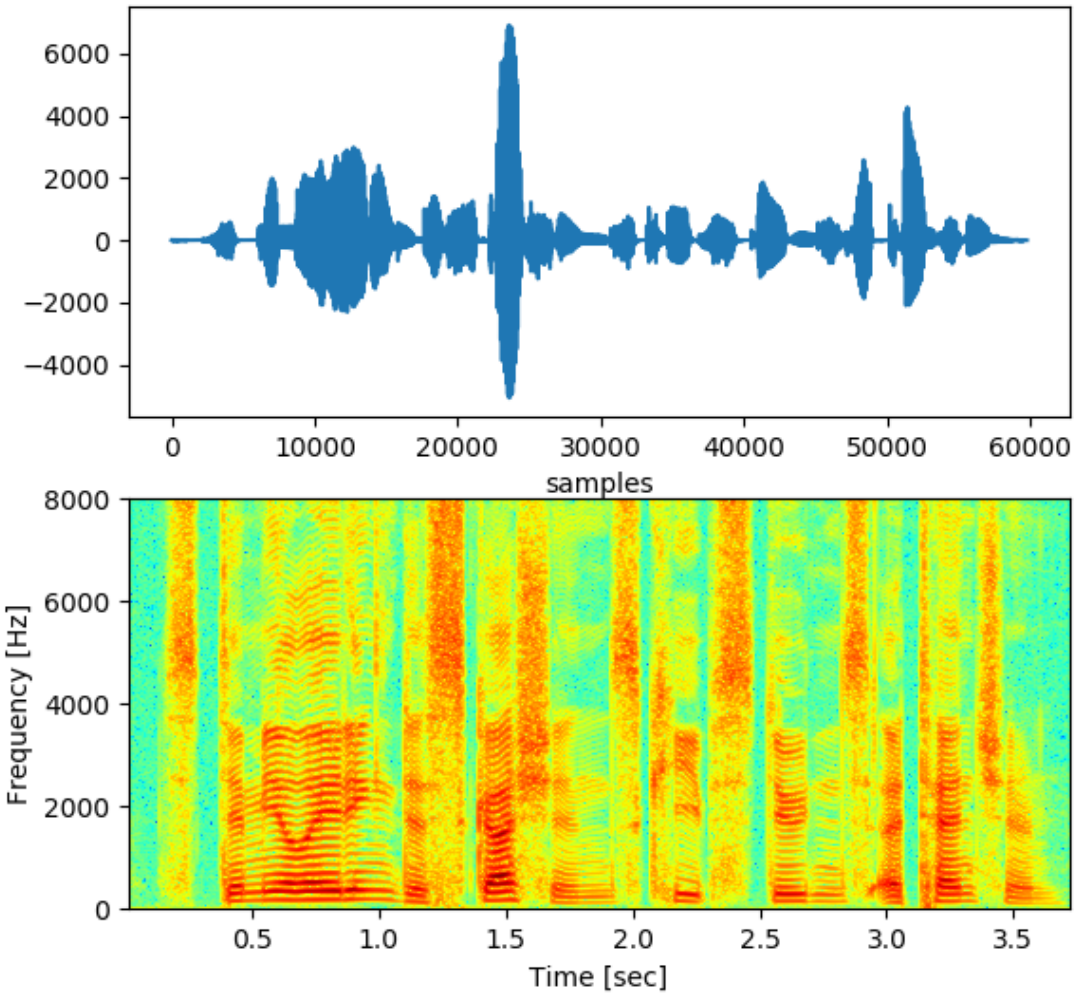


Time/frequency analysis: the spectrogram



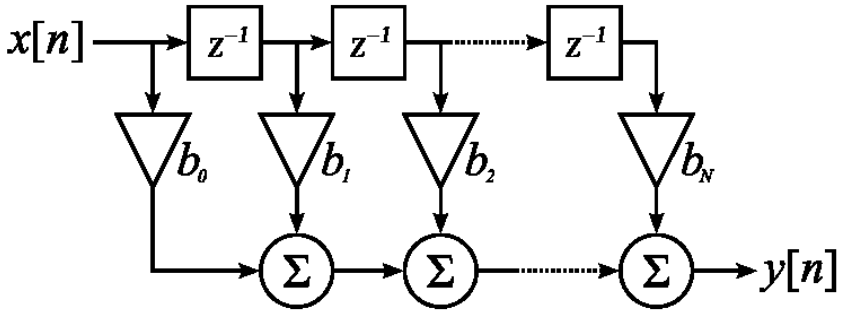
Example of spectrogram computation in Python for a speech signal

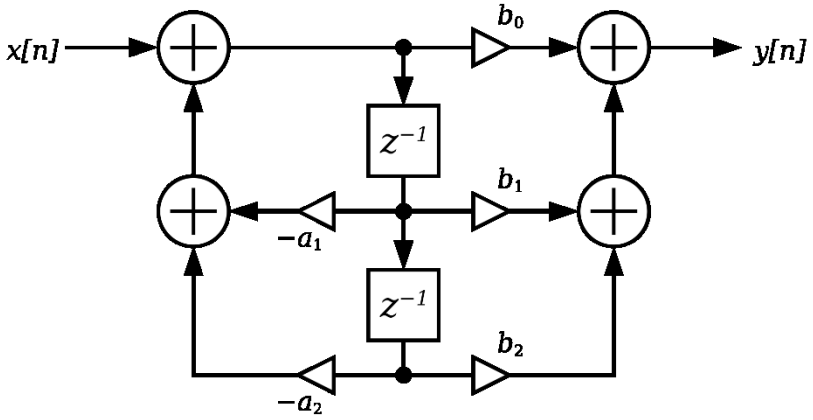
```
import matplotlib.pyplot as plt
from scipy.io import wavfile as wav

fs, x = wav.read('speech.wav') #read speech signal

fig, ax = plt.subplots(2, 1)

plt.axes(ax[0])
plt.plot(x)
plt.xlabel('samples')

plt.axes(ax[1])
plt.specgram(x, NFFT=512, pad_to=2048, \
                Fs=fs, noverlap=500)
plt.ylabel('Frequency [Hz]')
plt.xlabel('Time [sec]')
plt.jet()
```

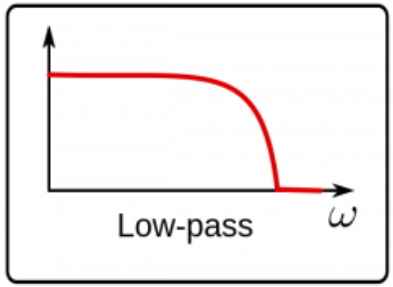


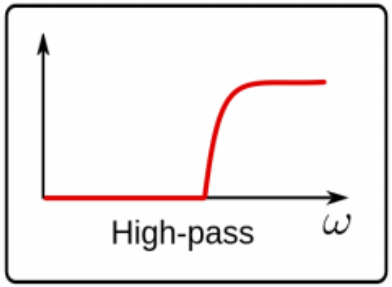
Frequency response of digital filters

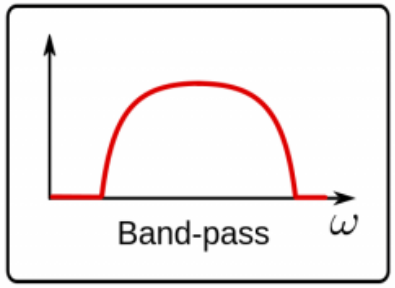
A **digital filter** is a system that performs mathematical operations on a sampled, discrete-time signal to modify certain aspects of that signal.
Often its characteristics are defined in the frequency domain, by specifying its **frequency response**.

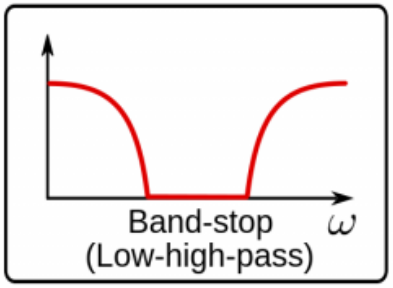






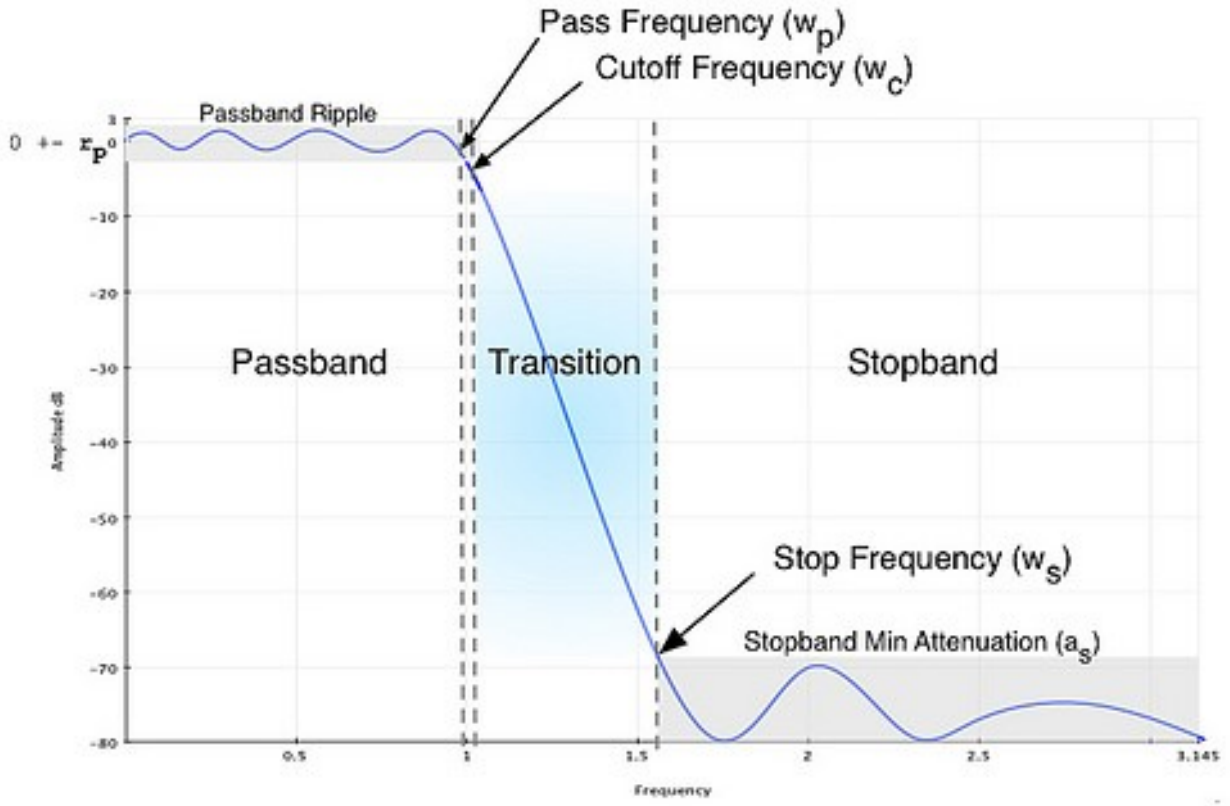




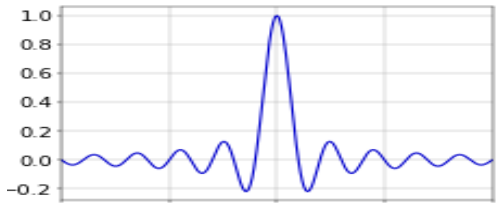


Frequency response of digital filters

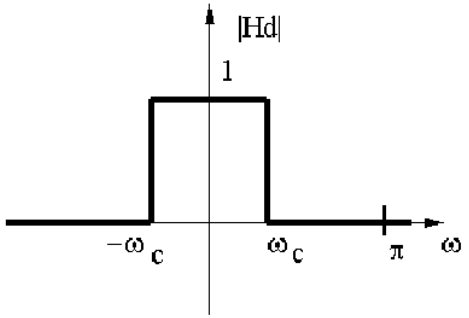
Real filters approximate the frequency response of ideal filters. The tolerance of the approximation is specified by means of requirements: *cutoff* frequency, maximum *oscillation in the passband*, minimum *attenuation in the stopband*, maximum *complexity* of the filter…



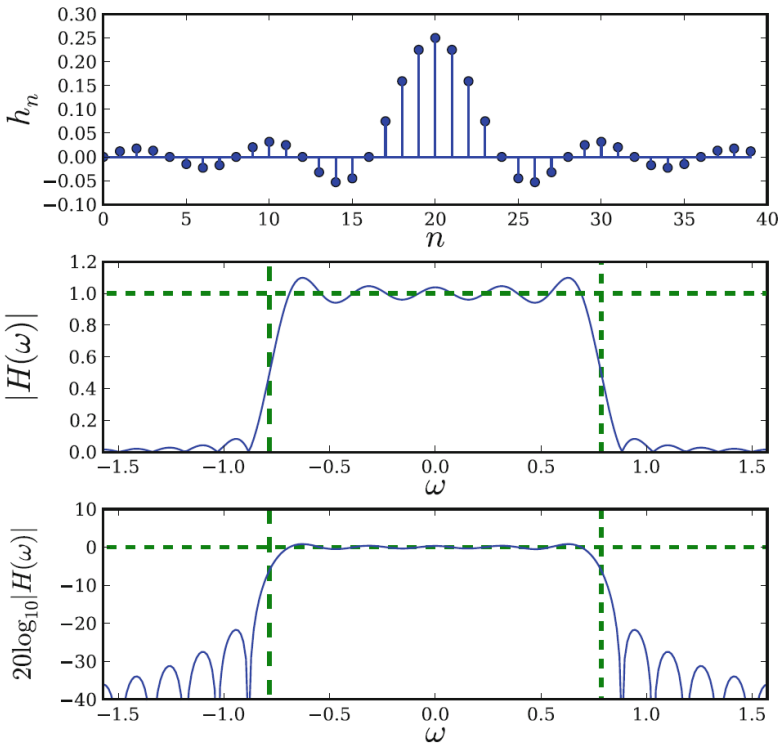
Effects of truncation and windowing on filters



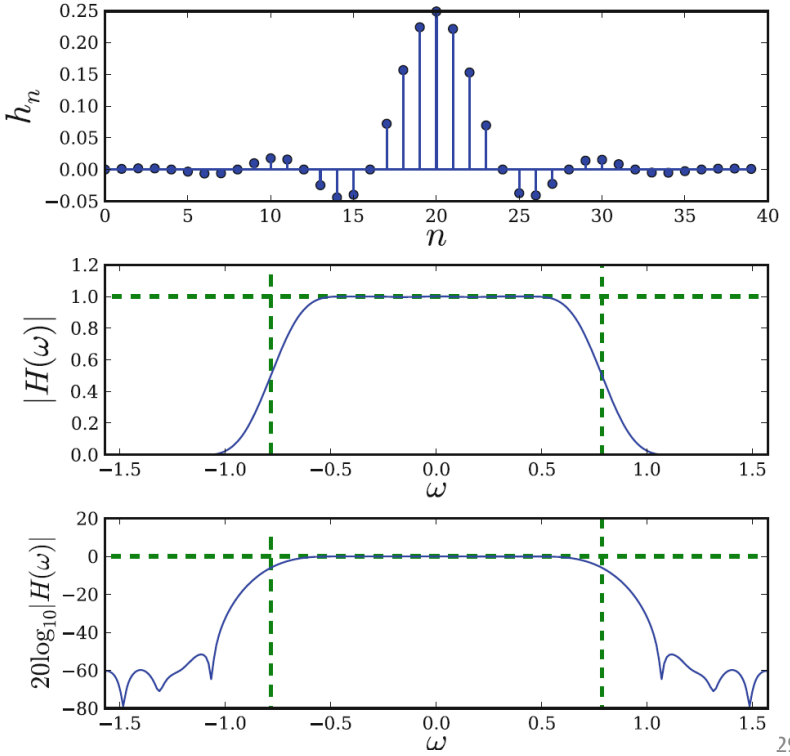
Ideal filter (infinite length)



TRUNCATION



TRUNCATION + WINDOWING



Examples of filter design in Python

Filter Design

Time-discrete filters can be classified into finite response (FIR) filters and infinite response (IIR) filters. FIR filters can provide a linear phase response, whereas IIR filters cannot. SciPy provides functions for designing both types of filters.

FIR Filter

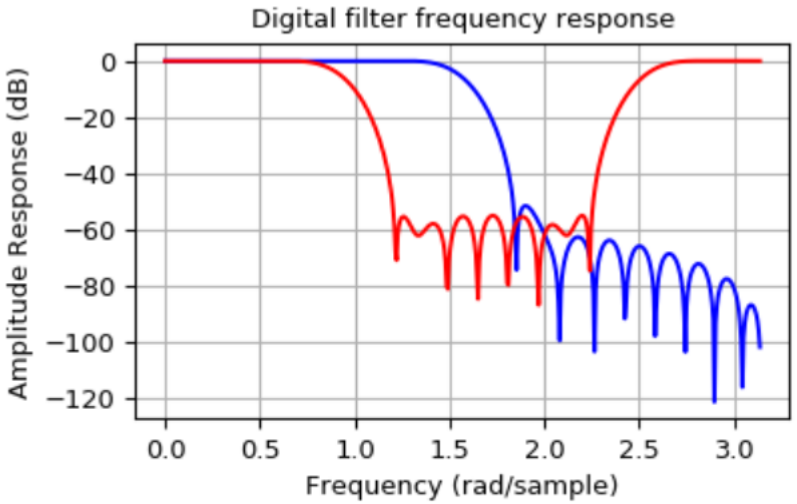
The function **firwin** designs filters according to the window method. Depending on the provided arguments, the function returns different filter types (e.g. low-pass, band-pass...).

The example below designs a low-pass and a band-stop filter, respectively.

```
>>> import numpy as np
>>> import scipy.signal as signal
>>> import matplotlib.pyplot as plt
```

```
>>> b1 = signal.firwin(40, 0.5)
>>> b2 = signal.firwin(41, [0.3, 0.8])
>>> w1, h1 = signal.freqz(b1)
>>> w2, h2 = signal.freqz(b2)
```

```
>>> plt.title('Digital filter frequency response')
>>> plt.plot(w1, 20*np.log10(np.abs(h1)), 'b')
>>> plt.plot(w2, 20*np.log10(np.abs(h2)), 'r')
>>> plt.ylabel('Amplitude Response (dB)')
>>> plt.xlabel('Frequency (rad/sample)')
>>> plt.grid()
>>> plt.show()
```



Examples of filter design in Python

IIR Filter

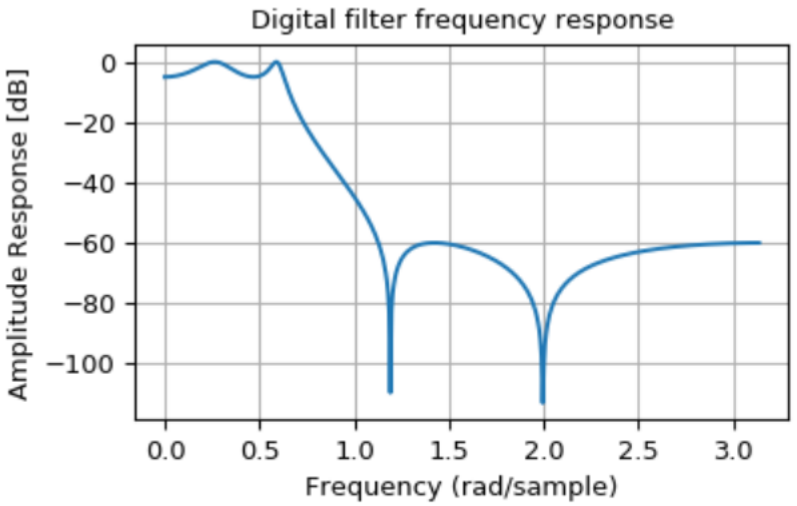
SciPy provides two functions to directly design IIR **iirdesign** and **iirfilter** where the filter type (e.g. elliptic) is passed as an argument and several more filter design functions for specific filter types; e.g. **ellip**.

The example below designs an elliptic low-pass filter with defined passband and stopband ripple, respectively. Note the much lower filter order (order 4) compared with the FIR filters from the examples above in order to reach the same stop-band attenuation of ≈ 60 dB.

```
>>> import numpy as np
>>> import scipy.signal as signal
>>> import matplotlib.pyplot as plt
```

```
>>> b, a = signal.iirfilter(4, Wn=0.2, rp=5, rs=60, btype='lowpass', ftype='ellip')
>>> w, h = signal.freqz(b, a)
```

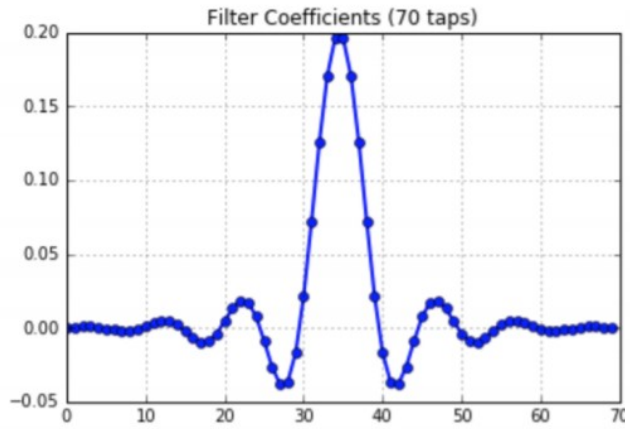
```
>>> plt.plot(w, 20*np.log10(np.abs(h)))
>>> plt.title('Digital filter frequency response')
>>> plt.ylabel('Amplitude Response [dB]')
>>> plt.xlabel('Frequency (rad/sample)')
>>> plt.grid()
>>> plt.show()
```

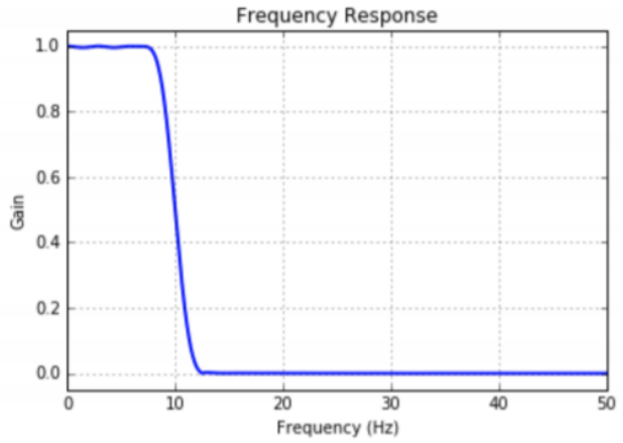


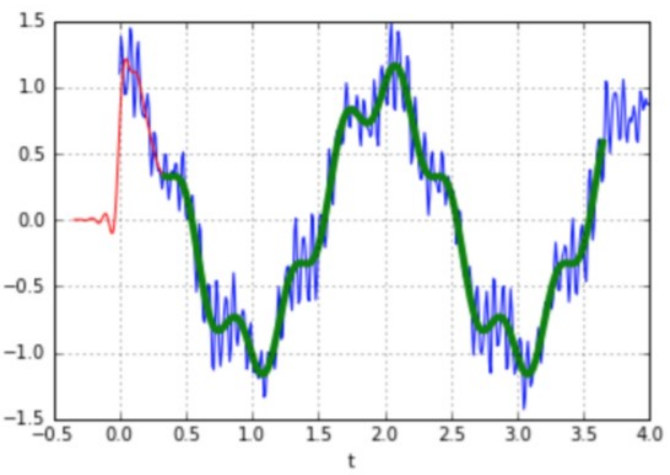
Example of filter application

```
sample_rate = 100
nsamples = 400

# signal generation
t = np.arange(nsamples) / sample_rate
x = np.cos(2*np.pi*0.5*t) + 0.2*np.sin(2*np.pi*2.5*t+0.1) + \
    0.2*np.sin(2*np.pi*15.3*t) + 0.1*np.sin(2*np.pi*16.7*t + 0.1)
    0.1*np.sin(2*np.pi*23.45*t+.8)

# FIR filter creation
nyq_rate = sample_rate / 2.0
cutoff_hz = 10.0
N = 70
taps = signal.firwin(N, cutoff_hz/nyq_rate)

# filter x with FIR filter
filtered_x = signal.lfilter(taps, 1.0, x)

# FIR filter coefficients plot
plt.figure(1)
plt.plot(taps, 'bo-', linewidth=2)
plt.title('Filter Coefficients (%d taps)' % N)
plt.grid(True)
plt.show()
```

```
# plot of frequency response of FIR filter
plt.figure(2)
w, h = signal.freqz(taps, worN=8000)
plt.plot((w/np.pi)*nyq_rate, np.absolute(h), linewidth=2)
plt.xlabel('Frequency (Hz)')
plt.ylabel('Gain')
plt.title('Frequency Response')
plt.ylim(-0.05, 1.05)
plt.grid(True)

# The phase delay of the filtered signal
delay = 0.5 * (N-1) / sample_rate
plt.figure(3)
# Plot the original signal
plt.plot(t, x)
# Plot the filtered signal, shifted to compensate for the phase delay.
plt.plot(t-delay, filtered_x, 'r-')
# Plot just the "good" part of the filtered signal  The first N-1
# samples are "corrupted" by the initial conditions
plt.plot(t[N-1:]-delay, filtered_x[N-1:], 'g', linewidth=4)
plt.xlabel('t')
plt.grid(True)
plt.show()
```







Example of spectral analysis in Python

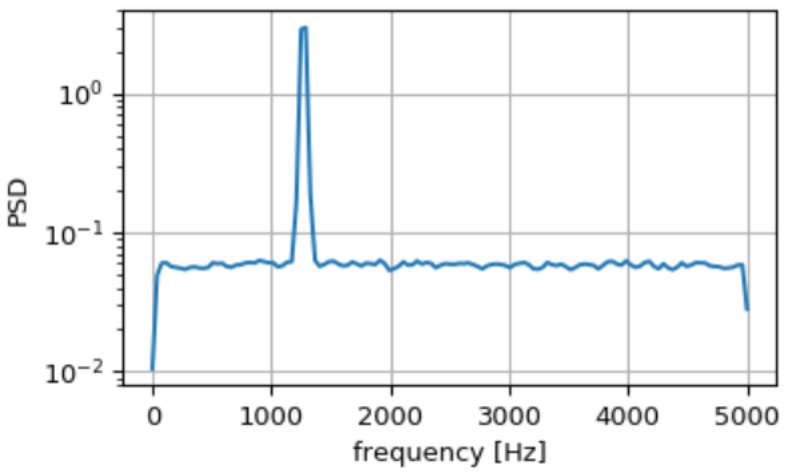
The scipy function welch provides a method to estimate the spectral density using the Welch's method.
The example below estimates the spectrum of a sine signal in white Gaussian noise

```
>>> import numpy as np
>>> import scipy.signal as signal
>>> import matplotlib.pyplot as plt
```

```
>>> fs = 10e3
>>> N = 1e5
>>> amp = 2*np.sqrt(2)
>>> freq = 1270.0
>>> noise_power = 0.001 * fs / 2
>>> time = np.arange(N) / fs
>>> x = amp*np.sin(2*np.pi*freq*time)
>>> x += np.random.normal(scale=np.sqrt(noise_power), size=time.shape)
```

```
>>> f, Pwelch_spec = signal.welch(x, fs, scaling='spectrum')
```

```
>>> plt.semilogy(f, Pwelch_spec)
>>> plt.xlabel('frequency [Hz]')
>>> plt.ylabel('PSD')
>>> plt.grid()
>>> plt.show()
```

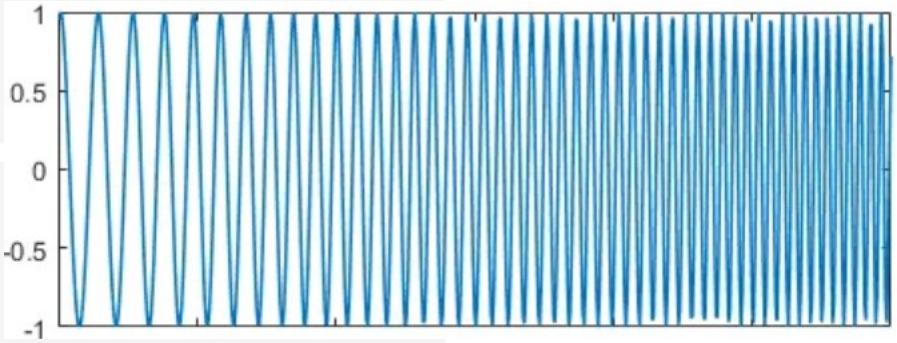


Example of Hilbert Transform in Python

In this example we use the Hilbert transform to determine the amplitude envelope and instantaneous frequency of an amplitude-modulated signal.

```
>>> import numpy as np
>>> import matplotlib.pyplot as plt
>>> from scipy.signal import hilbert, chirp
```

```
>>> duration = 1.0
>>> fs = 400.0
>>> samples = int(fs*duration)
>>> t = np.arange(samples) / fs
```

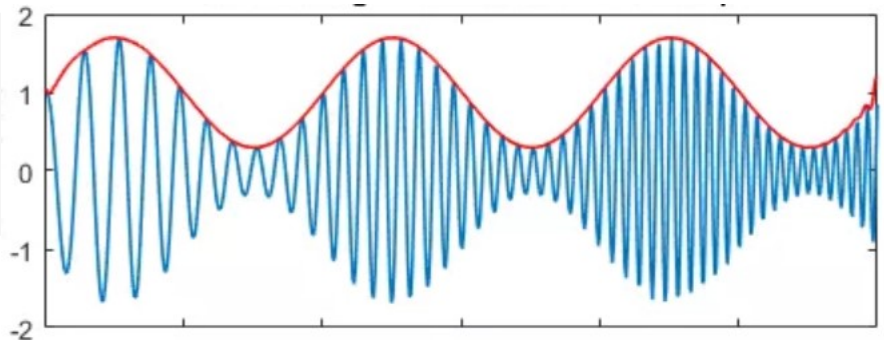


We create a chirp of which the frequency increases from 20 Hz to 100 Hz and apply an amplitude modulation.

```
>>> signal = chirp(t, 20.0, t[-1], 100.0)
>>> signal *= (1.0 + 0.5 * np.sin(2.0*np.pi*3.0*t) )
```

The amplitude envelope is given by magnitude of the analytic signal. The instantaneous frequency can be obtained by differentiating the instantaneous phase in respect to time. The instantaneous phase corresponds to the phase angle of the analytic signal.

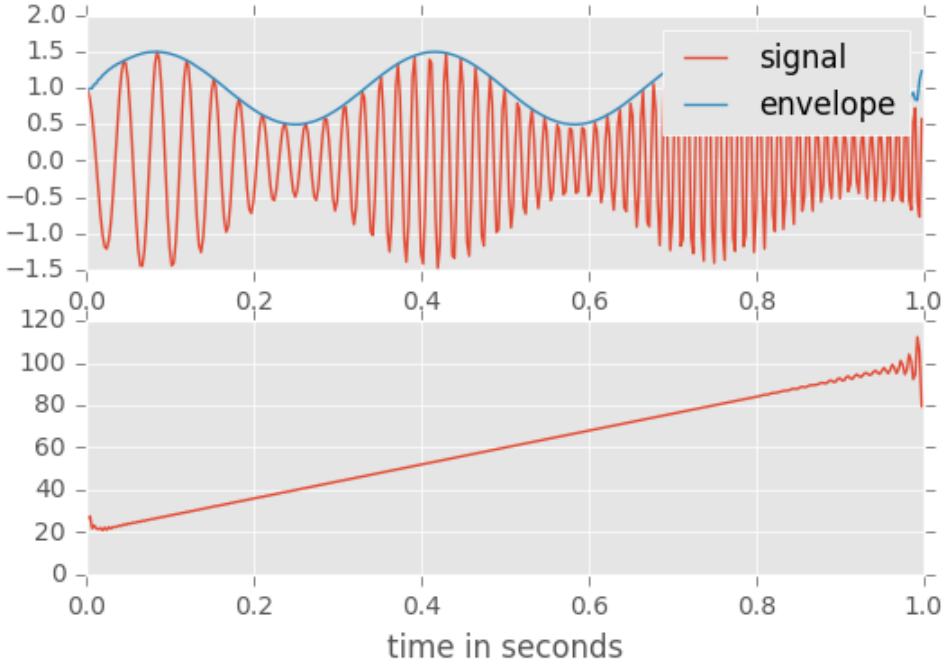
```
>>> analytic_signal = hilbert(signal)
>>> amplitude_envelope = np.abs(analytic_signal)
>>> instantaneous_phase = np.unwrap(np.angle(analytic_signal))
>>> instantaneous_frequency = (np.diff(instantaneous_phase) /
...                            (2.0*np.pi) * fs)
```



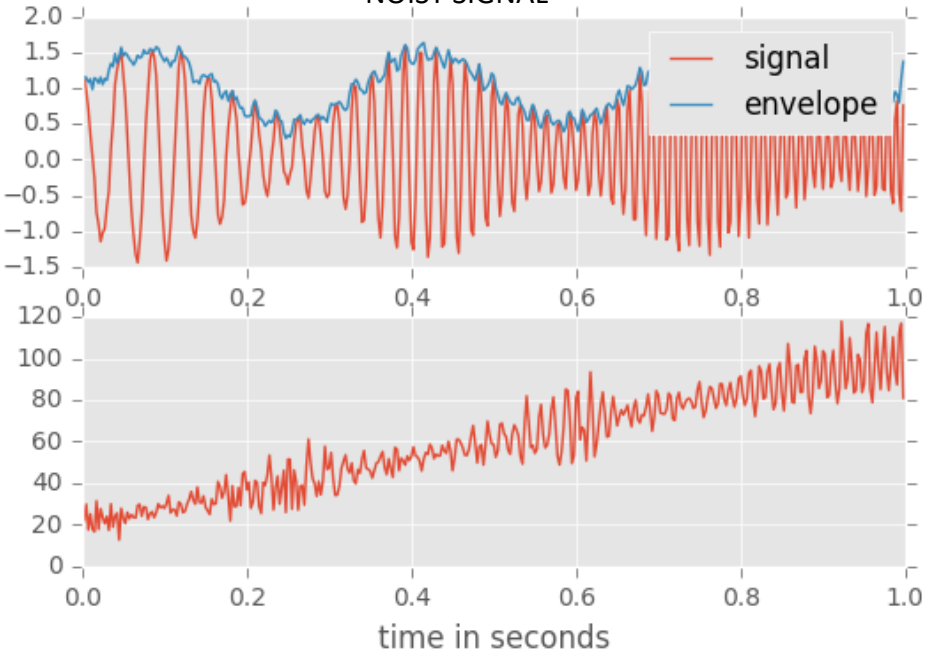
Example of Hilbert Transform in Python

```
>>> fig = plt.figure()
>>> ax0 = fig.add_subplot(211)
>>> ax0.plot(t, signal, label='signal')
>>> ax0.plot(t, amplitude_envelope, label='envelope')
>>> ax0.set_xlabel("time in seconds")
>>> ax0.legend()
>>> ax1 = fig.add_subplot(212)
>>> ax1.plot(t[1:], instantaneous_frequency)
>>> ax1.set_xlabel("time in seconds")
>>> ax1.set_ylim(0.0, 120.0)
```

CLEAN SIGNAL



NOISY SIGNAL

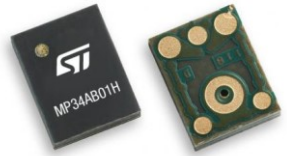


Sensors for signal acquisition

Microphones

A microphone is a transducer that converts **sound** into an electrical signal. Different methods can be used to convert the **air pressure variations** of a sound wave to an electrical signal (dynamic, condenser, piezoelectric microphones).

MEMS (MicroElectrical-Mechanical System) microphones are sensors directly integrated into a silicon chip with built-in preamplifier and analog-to-digital converter (ADC)

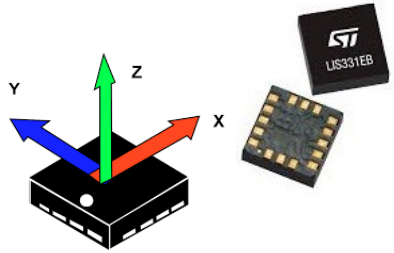


Accelerometers

An accelerometer is a device that measures **acceleration** (rate of change of velocity). Piezoelectric, piezoresistive and capacitive components are commonly used to convert the mechanical motion into an electrical signal.

Single- and multi-axis models of accelerometer are available to detect **magnitude** and **direction** of the acceleration, as a vector quantity, and can be used to sense orientation, coordinate acceleration, vibration, shock and falling.

(MEMS) accelerometers are widely used in portable electronic devices



FONDAZIONE BRUNO KESSLER

Signal analysis and feature extraction for condition monitoring

Time-domain features:

- signal energy (root-mean-square (RMS), variance, min/max, envelope)
- zero-crossing rate
- signal-to-noise ratio (SNR)
- autocorrelation (periodicity)

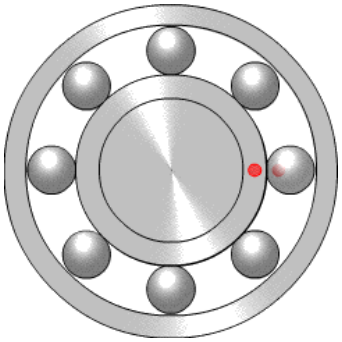
Frequency-domain features:

- Fourier coefficients (DFT, FFT, DCT)
- filter banks (uniform, log-scale, critical frequencies)
- instantaneous frequency (Hilbert Transform)

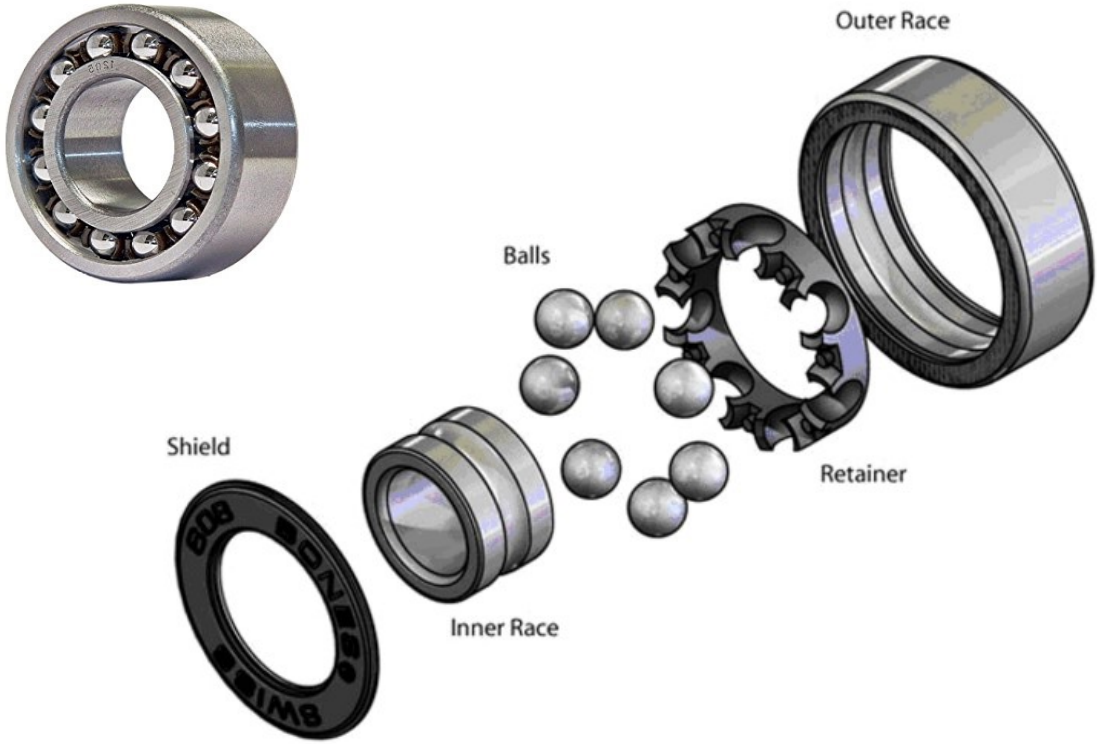
Time-frequency features:

- Spectrogram, STFT, Wigner-Ville distribution
- Empirical Mode Decomposition (EMD), Hilbert-Huang Transform

Rolling bearings

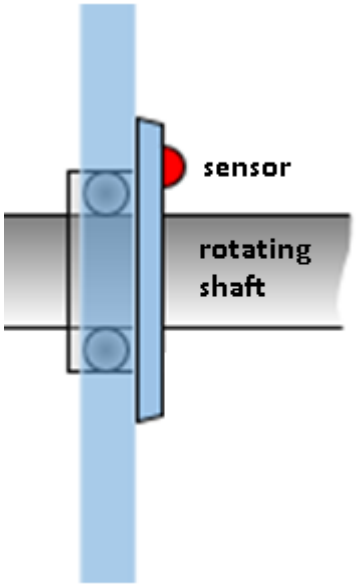


- Critical component of any rotating machinery
- Wearing and faults can be easily studied on test-bench platforms
- Fault diagnosis is achieved by vibration analysis



Rolling bearings

Advanced **signal processing** and **statistical methods** are needed to detect **non-linear** and **non-stationary** events related with fault evolution

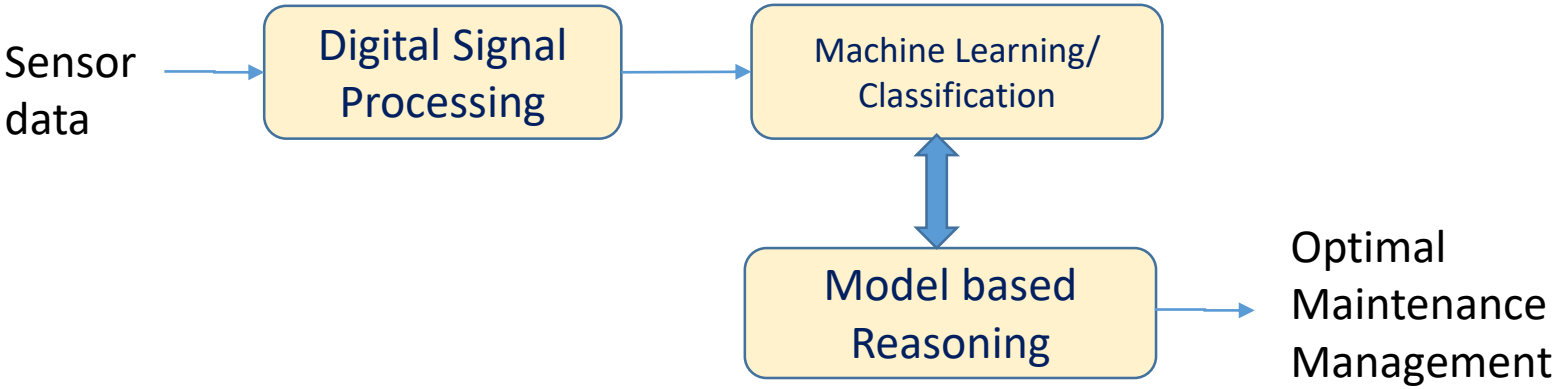


Meaningful **features** are extracted from data using

- Short Time Fourier Transform
- Empirical mode decomposition (EMD)
- Hilbert-Huang Transform (HHT)
- Constant-percentage-bandwidth (CPB) spectral analysis

- Spectral kurtosis
- Real cepstrum

Integrated predictive maintenance system

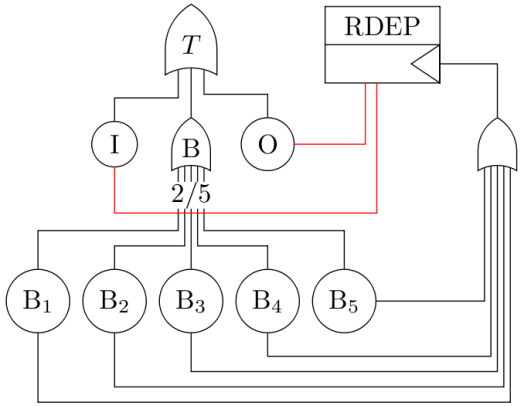


- Feature selection, extraction, and transformation, to determine which data are the most significant for predicting degradation and failures
- Development of predictive models using machine learning techniques
- System modeling (architecture, dependencies, fault trees)
- Warning about faults as they develop (localization, type, consequences) and optimization of maintenance management

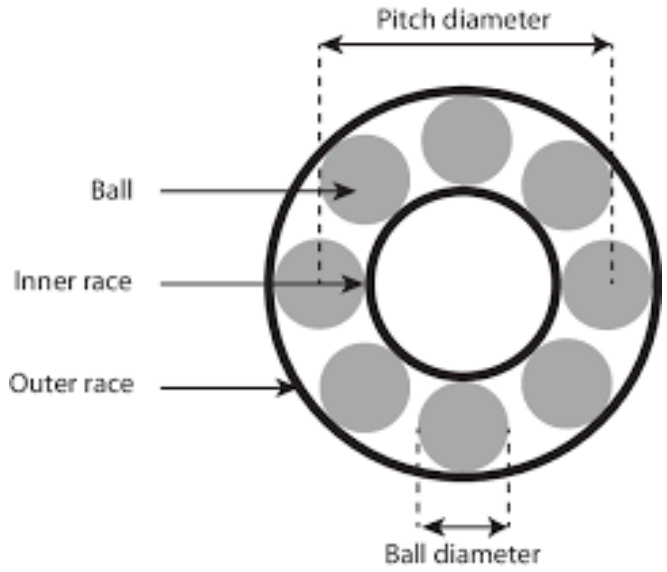
Integrated predictive maintenance system

Model-based Condition monitoring

- Design, train and tune suitable models for “normal” operating conditions and for known failure typologies (from explicit knowledge, annotated data, adaptation, machine learning, big data analysis)
- Differentiate normal wear from abnormal wear.
- Detect anomalous parameters and diagnose the problem (e.g. physical damage, imbalance conditions, lack of lubrication…)
- Forecast the evolution of equipment’s conditions



Fundamental fault frequencies for ball bearings



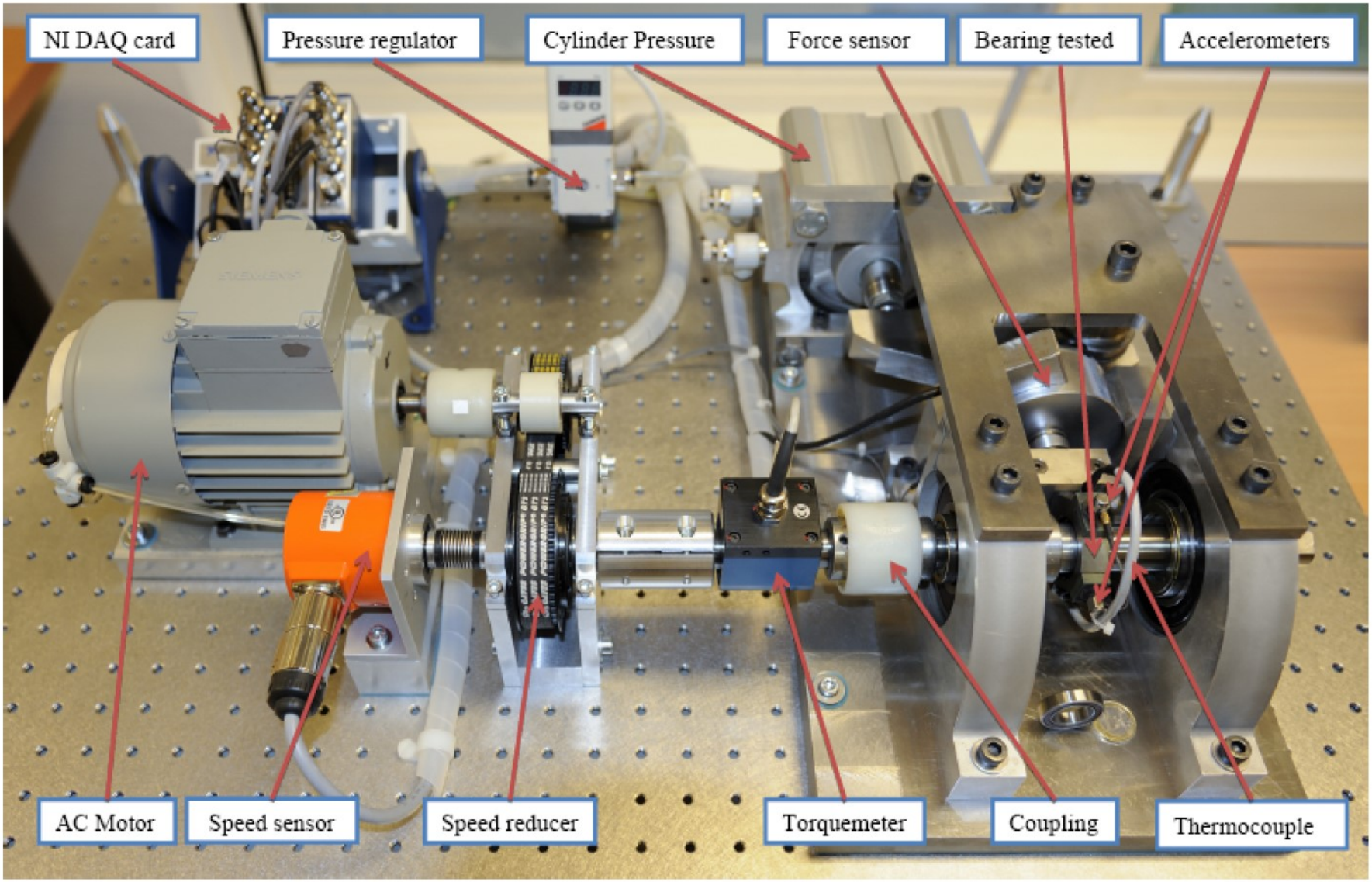
$$\mathrm{F}OR = \frac{NB}{2} * FR \ * (1 - \frac{DB}{DP} * \cos(\psi))$$

$$\mathrm{F}IR = \frac{NB}{2} * FR \ * (1 + \frac{DB}{DP} * \cos(\psi))$$

$$\mathrm{F}B = \frac{DP}{DB} * FR \ * \left(1 - \left(\frac{DB}{DP} * \cos(\psi)\right)^2\right)$$

NB -- ball number
FR -- rotation frequency in Hertz
DB -- ball diameter
DP -- pitch diameter
ψ -- contact angle in radiants

Rolling bearing test platform



FONDAZIONE BRUNO KESSLER

The Empirical Mode Decomposition (EMD) and the Hilbert-Huang Transform (HHT)

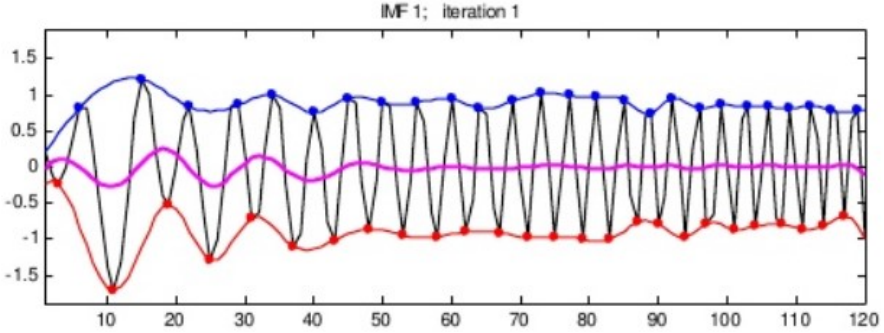
Empirical Mode Decomposition (EMD) is a procedure that breaks down signals into various components. Any complicated data set can be decomposed into a finite and often small number of components. These components form a complete and nearly orthogonal basis for the original signal.

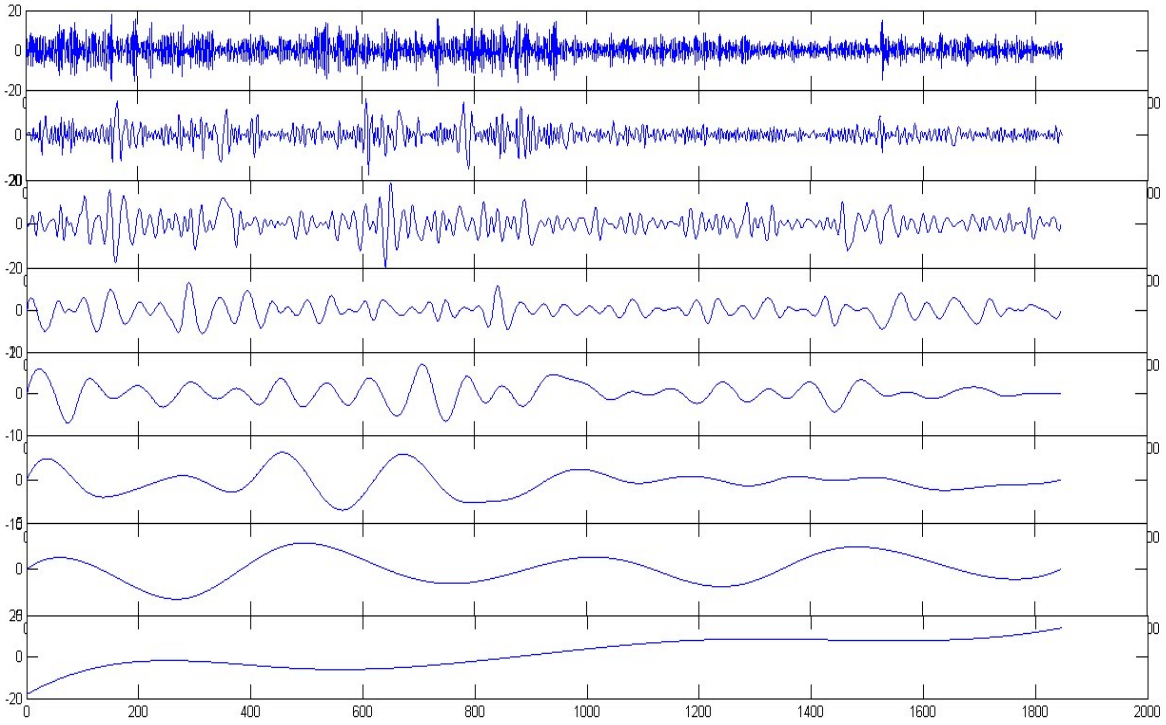
Each component represents a simple oscillatory mode which can have can have variable amplitude and frequency along the time axis.

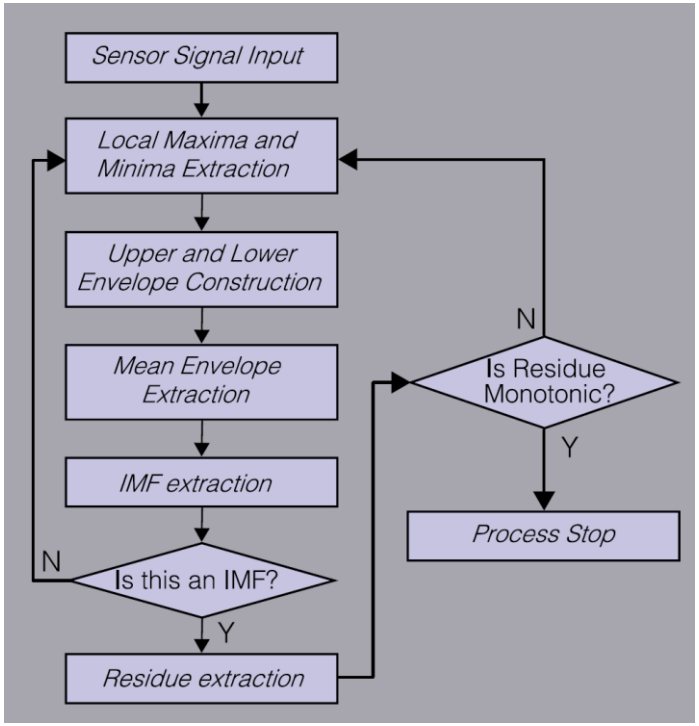
The **Hilbert–Huang transform** (**HHT**) is a way to decompose a signal into so-called intrinsic mode functions (IMF) along with a trend, and obtain instantaneous frequency data. It is designed to work well for data that is nonstationary and nonlinear.

HHT is the result of combining the empirical mode decomposition (EMD) with the Hilbert Spectral Analysis (HSA) which estimates instantaneous frequency and phase for a given signal component.

Empirical Mode Decomposition (EMD)







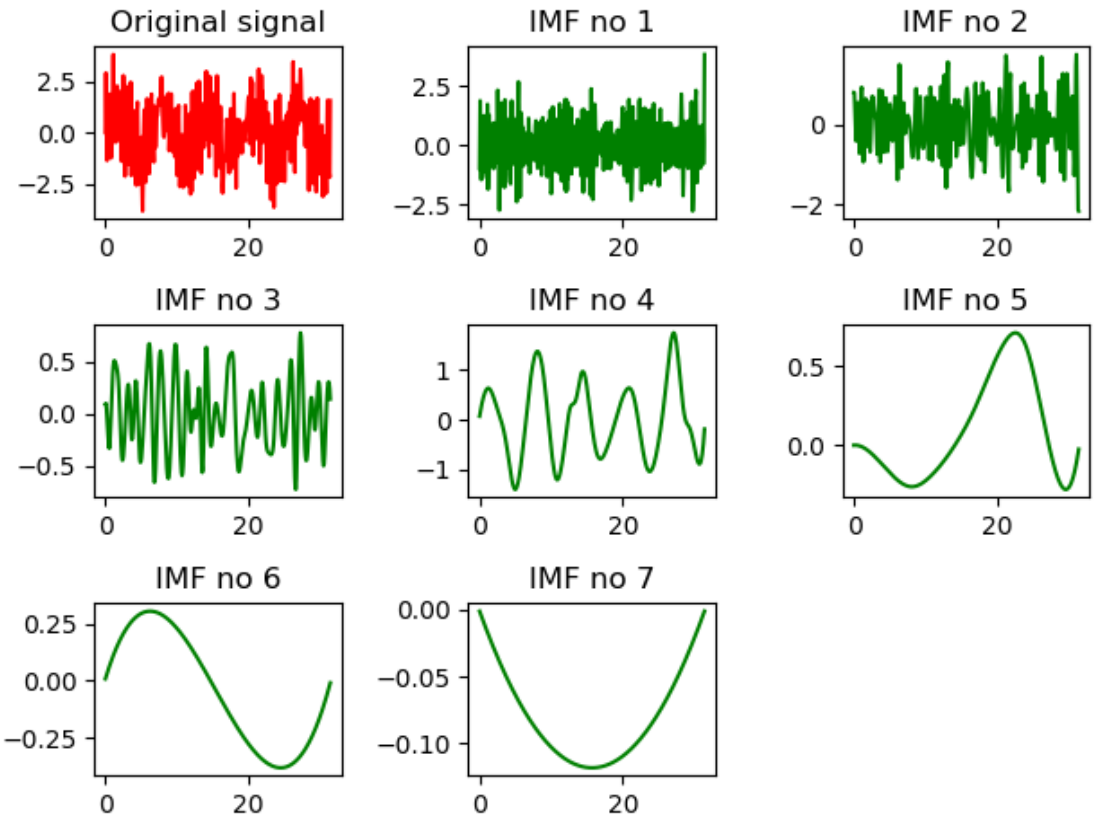
Example of Python implementation of EMD

```
import numpy as np
import pylab as py
from scipy import interpolate


def emd(x):
    imf=np.array([x])
    while (not ismonotonic(x)):
        x1=x
        sd=float('inf')

        c=1
        while((c<100) and ((sd > 0.1) or (not isimf(x1)))):
            s1 = getspline(x1)
            s2 = -getspline(-x1)
            x2 = x1-(s1+s2)/2.0;
            sd = np.sum((x1-x2)**2)/np.sum(x1**2)
            x1 = x2
            c=c+1
        imf=np.append(imf,[x1],axis=0)
        x=x-x1
    imf=np.append(imf,[x],axis=0)
    return imf

def findpeaks(x):
    n1=(np.diff(x)>0)*1
    n=np.array(np.where(np.diff(n1)<0))
    n=np.array(np.squeeze(n+1))
    return n

def ismonotonic(x):
    u1 = findpeaks(x).size*findpeaks(-x).size
    if u1>0:
        return 0
    else:
        return 1
```

```
def isimf(x):
    u1=np.sum((x[1:]*x[:-1]<0)*1)
    u2=findpeaks(x).size+findpeaks(-x).size
    if np.abs(u1-u2)>1:
        return 0
    else:
        return 1

def getspline(x):
    N=len(x)
    p=findpeaks(x)
    if p.size>1:
        t, c, k = interpolate.splrep(np.concatenate(([0],p+1,[N+1])), \
                  np.concatenate(([0],x[p],[0])), s=0, k=3)
        spline = interpolate.BSpline(t, c, k, extrapolate=False)
        s=spline(np.arange(1,N+1))
    elif p.size==1:
        t, c, k = interpolate.splrep(np.concatenate(([0],[p+1],[N+1])), \
                  np.concatenate(([0],[x[p]],[0])), s=0, k=2)
        spline = interpolate.BSpline(t, c, k, extrapolate=False)
        s=spline(np.arange(1,N+1))
    else:
        s=np.zeros(N,float)
    return s
```

Example of Python implementation of EMD

```
N = 400
timeLine = t = np.linspace(0, 10 * np.pi, N)
tS = 'np.sin(20*t*(1+0.2*t)) + np.sin(20*t*(1+0.7*t))'
tS += '+ np.sin(13*t) + np.sin(1*t)'
S = eval(tS)



IMF=emd(S)   # Empirical Mode Decomposition
imfNo=IMF.shape[0]

c = np.floor(np.sqrt(imfNo + 3))
r = np.ceil((imfNo + 1) / c)

py.ioff()
py.subplot(r, c, 1)
py.plot(timeLine, S, 'r')
py.title("Original signal")
for num in range(1,imfNo):
    py.subplot(r, c, num + 1)
    py.plot(timeLine, IMF[num], 'g')
    py.title("IMF no " + str(num))

py.tight_layout()
py.show()
```



Example of Python implementation of Hilbert-Huang Transform and Marginal Hilbert Spectrum

```
Fs=25600
NF=256
FBIN=NF*2+1


imf=emd(x)    # Empirical Mode Decomposition

NT=imf.shape[1]
HHT=np.zeros((FBIN,NT),float)
for k in range(1,imf.shape[0]):
    h=hilbert(imf[k,:])    # Hilbert transform
    A=np.abs(h)              # Amplitude
    phase=np.unwrap(np.angle(h))    # Phase
    frq=Fs*np.diff(phase)/(2*np.pi) # Instantaneous frequency
    fbins=np.round(NF*frq/(Fs*0.5)+NF).astype(int)
    for m in range(0,NT-1):
        HHT[fbins[m],m]=HHT[fbins[m],m]+A[m]**2    # Hilbert-Huang Transform

MHS=np.sum(HHT,axis=1) # Marginal Hilbert Spectrum
```

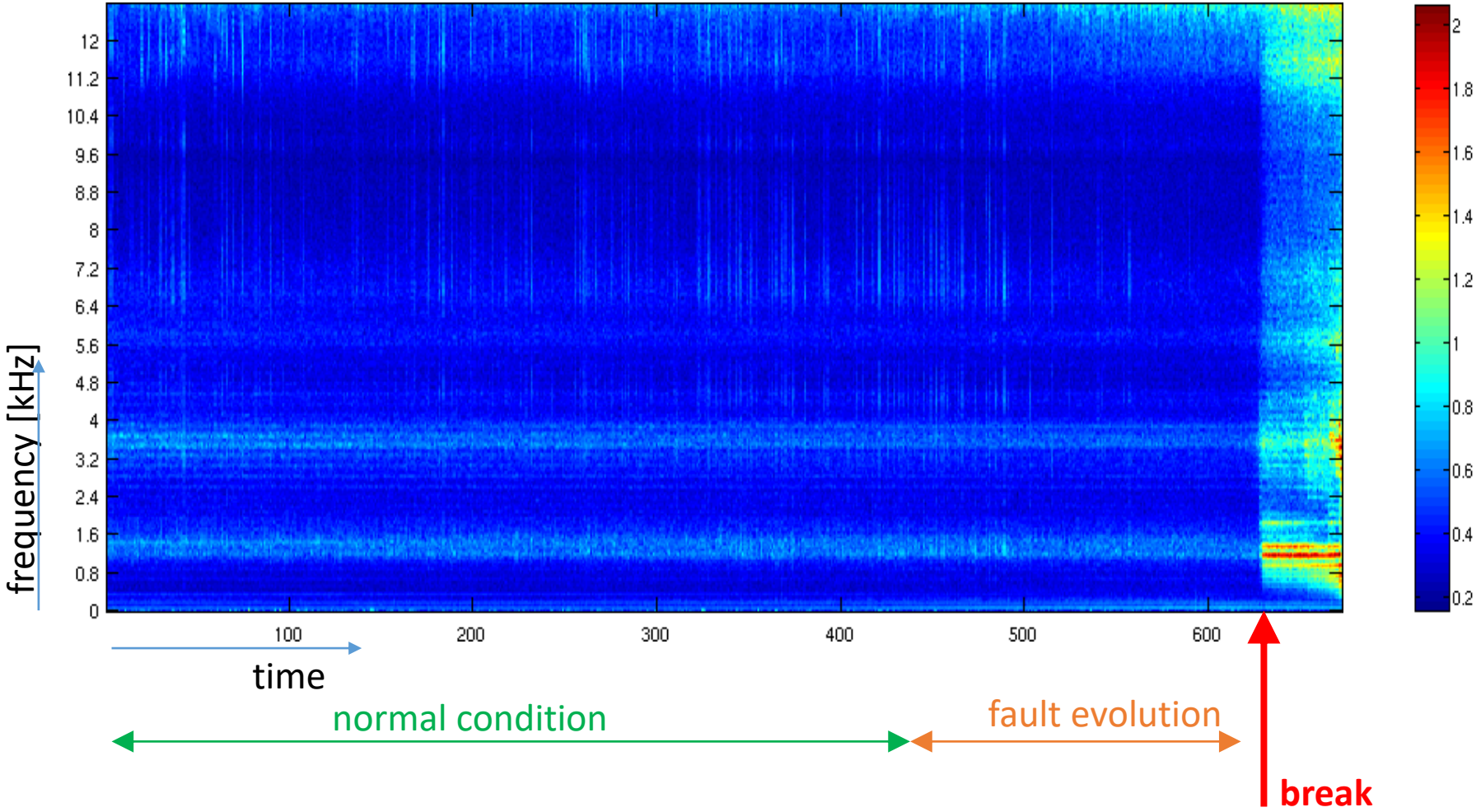
- Given the Hilbert Spectrum as $H(\omega, t)$, the **Marginal Hilbert Spectrum** is defined as

$$h(\omega) = \int_0^T H(\omega, t)dt$$

- The $h(\omega)$ spectrum offers a measure of the total amplitude contribution from the frequency ω of interest

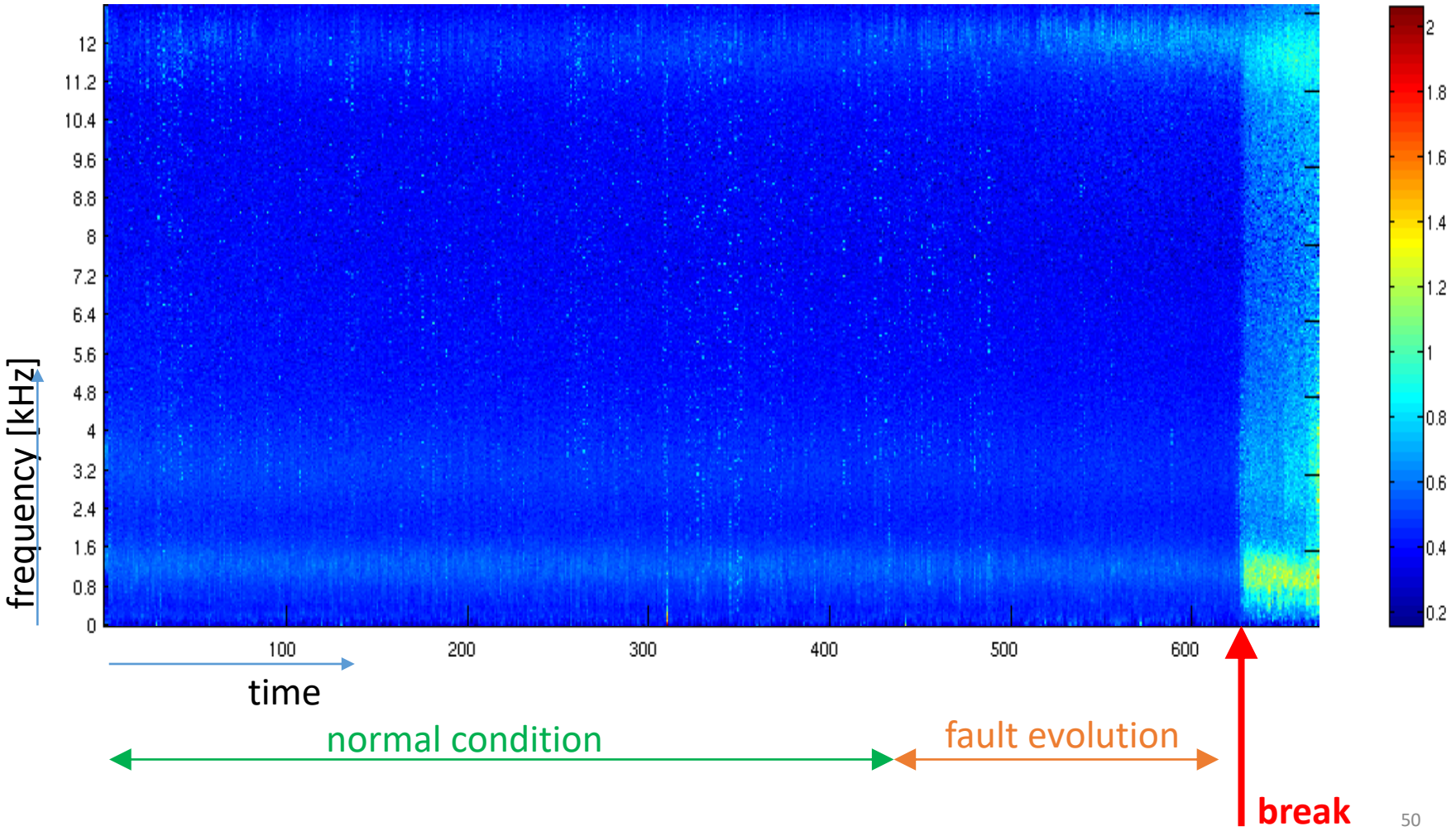
FONDAZIONE
BRUNO KESSLER

Conventional spectrogram (linear frequency) - Bearing test evolving to break



FONDAZIONE
BRUNO KESSLER

Hilbert-Huang marginal spectrum - Bearing test evolving to break



Power spectrum over a logarithmically-spaced frequency axis

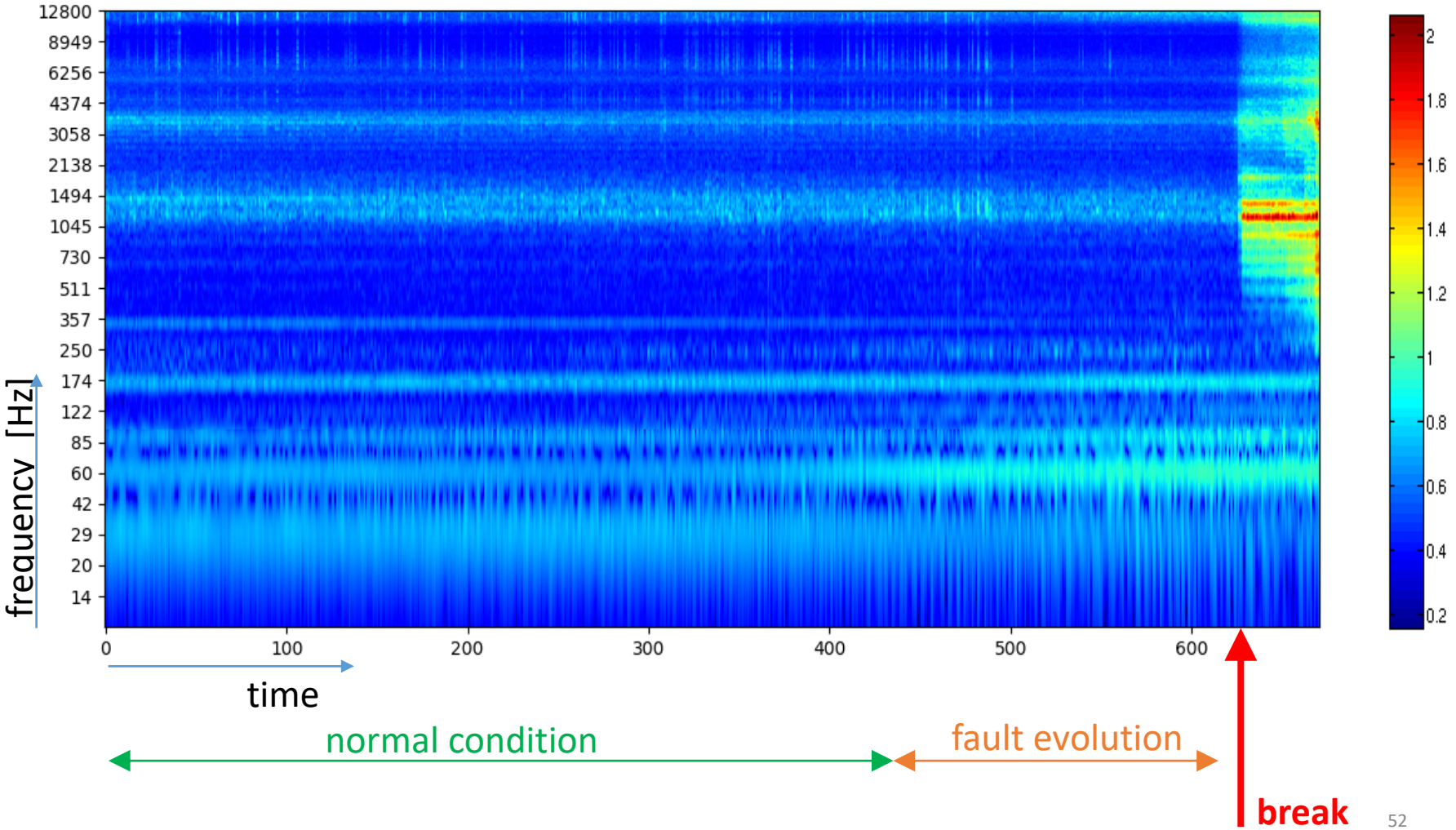
Estimates the power spectrum of the sequence x at Jdes frequencies equally spaced on a logarithmic scale from fmin to fmax

```
import numpy as np
import matplotlib.pyplot as plt


def lpsd1(x,fmin,fmax,Jdes,Kdes,Kmin,fs,xi):
    N=len(x)

    jj=np.array([i for i in xrange(Jdes)])
    g=np.log(fmax)-np.log(fmin)
    f=fmin*np.exp(jj*g/(Jdes-1))
    rp=fmin*np.exp(jj*g/(Jdes-1))*(np.exp(g/(Jdes-1))-1)
    ravg=(fs/N)*(1+(1-xi)*(Kdes-1))
    rmin=(fs/N)*(1+(1-xi)*(Kmin-1))

    case1=rp>=ravg
    case2=(rp<ravg)&(np.sqrt(ravg*rp)>rmin)
    case3=~(case1|case2)

    rpp=np.zeros(Jdes)

    rpp[case1]=rp[case1]
    rpp[case2]=np.sqrt(ravg*rp[case2])
    rpp[case3]=rmin

    L=np.round(fs/rpp)
    r=fs/L
    m=f/r

    Pxx=np.zeros(Jdes)
    S1=np.zeros(Jdes)
    S2=np.zeros(Jdes)
```

```
for jj in range(len(f)):
    D=np.int(np.round((1-xi)*L[jj]))
    K=np.int(np.floor((N-L[jj])/D+1))

    ii=np.arange(L[jj]).reshape(np.int(L[jj]),1)+D*np.arange(K)
    ii=ii.astype(int)
    data=x[ii]
    data=data-np.mean(data,axis=0)

    window=np.hamming(L[jj])
     window=np.hanning(L[jj])
    window=window.reshape(len(window),1)
    sinusoid=np.exp(2j*np.pi*np.arange(L[jj]).reshape(np.int(L[jj]),1)\
                    *m[jj]/L[jj])
    data=data*(sinusoid*window)

    Pxx[jj]=np.mean(abs(data.sum(axis=0))**2)

    S1[jj]=np.sum(window)
    S2[jj]=np.sum(window**2)

C_PS=2*np.sqrt(S1)
C_PSD=2/(fs*S2)

X=Pxx*C_PSD


return X,f
```

FONDAZIONE
BRUNO KESSLER

Example of CPB spectral analysis of vibration - Bearing test evolving to break



FONDAZIONE
BRUNO KESSLER

Some useful links for DSP and Python

Introduction to Digital Signal Processing:
https://en.wikipedia.org/wiki/Digital_signal_processing
(Nyquist theorem, Fourier Transform, FFT, Hilbert Transform, Digital filters,…)

"**A Whirlwind Tour of Python**" by Jake VanderPlas:
https://github.com/jakevdp/WhirlwindTourOfPython

Signal processing with scipy.signal:
https://docs.scipy.org/doc/scipy/reference/signal.html

"**The Scientist and Engineer's Guide to Digital Signal Processing**"
by Steven W. Smith
http://www.dspguide.com/