Contents

1 Covariance

In probability theory and statistics, covariance is a measure of how much two random variables change togheter. If the greater values of one variable mainly correspond with the greater values of the other variable, and the same holds for the smaller values, i.e the variables tend to show similar behaviour, the covariance is positive. In the opposite case, when the greater values of one variable mainly correspond to the smaller values of the other, i.e the variables tend to show opposite behaviour,

the covariance is negative. The sign of the covariance therefore shows the tendency in the linear relationship between the variables. The magnitude of the covariance is not easy to interpret. The normalized version of the covariance, the correlation coefficient, however, shows by its magnitude the strength of the linear relation.

A distinction must be made between 1) the covariance of two random variables, which is a population parameter that can be seen as a property of the joint probability distribution, and 2) the sample covariance, which serves as an estimated value of the parameter.

The equation for the covariance is:$\sigma(s_1(t), s_2(t)) =$, where$\sigma(X, Y)$is covariance between two variables$X$and$Y$,E– expected value,X– set of results for one variable,Y– set of results for the second variable.

1.1 Expected value

One should also know what exactly$E$is, before calculating the covariance coefficient. Suppose random variable$X$can take value$x_1$with probabilityp_1, value$x_2$with probabilityp_2, and so on, up to value$x_k$with probabilityp_k. Then the expectation of this random variable $X$is defined as:$E[X] = x_1p_1 + x_2p_2 + \ldots + x_kp_k$. Since all probabilities p_i add up to one, i.e $p_1 + p_2 + \ldots + p_k = 1$, the expected value can be viewed as the weighted average, with p_i's being the weights: $E[X] = \frac{x_1p_1 + x_2p_2 + \ldots + x_kp_k}{1} = \frac{x_1p_1 + x_2p_2 + \ldots + x_kp_k}{p_1 + p_2 + \ldots + p_k}$. If all outcomes$x_i$are equally likely (that is$p_1 = p_2 = \ldots = p_k$), then the weighted average turns into the simple average. This is intuitive: the expected value of a random variable is the average of all values it can take. Thus, the expected value is what one expects to happen on average. If the outcomes$x_i$are not equally probable, then the simple average must be replaced with the weighted average, which takes into account the fact that some outcomes are more likely than the others. The intuition however remains the same: the expected value of$X$is what one expects to happen on average.

Example: Let$X$represent the outcome of a roll of a fair six-sided die. More specifically,Xwill be the number of pips showing on the top face of the die after the toss. The possible values for$X$are 1, 2, 3, 4, 5, and 6, all equally likely (each having the probability of 1/6). The expectation of$X$is:$E[X] = 1 * (\frac{1}{6}) + 2 * (\frac{1}{6}) + 3 * (\frac{1}{6}) + 4 * (\frac{1}{6}) + 5 * (\frac{1}{6}) + 6 * (\frac{1}{6}) = 3.5$. If one rolls the die n times and computes the average (arithmetic mean) of the results, then as$n$grows, the average will almost surely converge to the expected value, a fact known as the strong law of large numbers. One example sequence of ten rolls of the die is 2, 3, 1, 2, 5, 6, 2, 2, 2, 6, which has the average of 3.1, with the distance of 0.4 from the expected value of 3.5. The convergence is relatively slow: the probability that the average falls withing the range$3.5 \mp 0.1$is 21.6% for ten rolls, 46.1% for a hundred rolls and

93.7% for a thousand rolls. See the Fig. 1 for an illustration of the averages of longer sequencies of rolls of the die and how they converge to the expected value of 3.5. More Generally, the rate of convergence can be rougly quantified by e.g. Chebyshev's inequality and the Berry-Esseen theorem.

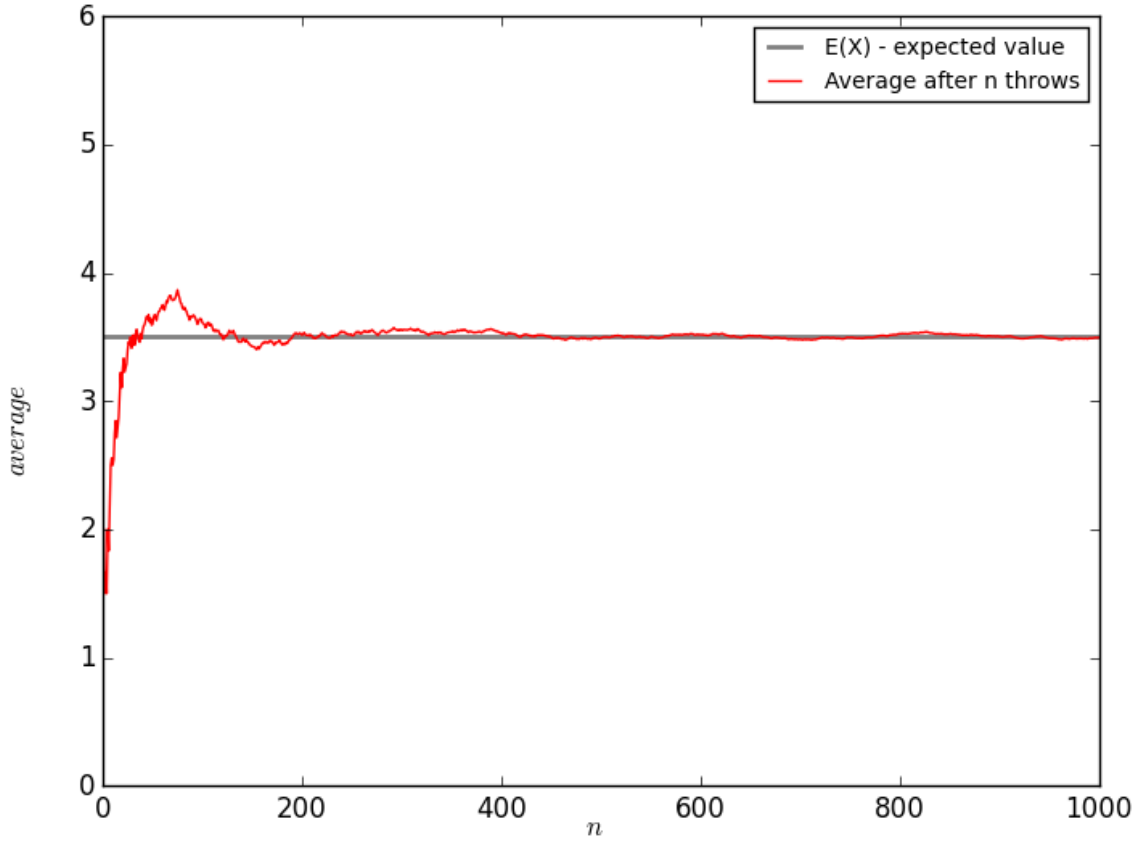

Figure 1: An illustration of the convergence of sequence averages of rolls of a die to the expected value of 3.5 as the number of rolls (trials) grows.

```
'''
Script for plotting the convergence of sequence averages of rolls of a die to
the expected value.
'''

import numpy as np
import matplotlib.pylab as pl

samples = 10000

X = np.array([1, 2, 3, 4, 5, 6]) # set of values
P = np.array([1/6, 1/6, 1/6, 1/6, 1/6, 1/6]) # set of probabilities
rolls = np.arange(1, samples + 1) # number of rolls

# Expected value function.
def E(X, P):
    expectedValue = 0
```

```
    for i in X-1:
        expectedValue += X[i] * P[i]
    return expectedValue

# Add x and y axes labels.
pl.figtext(0.5, 0.05, '$n$', rotation='horizontal', size='12')
pl.figtext(0.05, 0.5, '$average$', rotation='vertical', size='12')

# Plot expected value line.
pl.plot(rolls, np.full(samples, E(X, P)), label='E(X) - expected value',
linewidth=2.0, color='gray')

# Convergence of averages function.
def convergence(X, rolls):
    averages = np.array([])
    Xi = np.array([])
    for i in rolls - 1:
        Xi = np.append(Xi, np.random.choice(X))
        averages = np.append(averages, np.average(Xi))
    return averages

# Plot averages.
pl.plot(rolls, convergence(X, rolls), label='Average after n throws',
linewidth=1.0, color='red')

# Set y axis range.
pl.ylim([np.min(X), np.max(X)])

# Show legend and plot.
pl.legend(loc='upper right', fontsize='10')
pl.show()
```

1.2 Calculation of the covariance coefficient

In order to calculate the covariance coefficient between two variables X and Y, one must evaluate the product of the multiplication for each pair of variables values:

ID	X	Y	X*Y
1	156	54	8424
2	173	45	7785
3	183	89	16287
4	187	87	16269
5	176	76	13376
6	168	56	9408

Next, calculate the expected values for each variable and for the calculated multiplication products, i.e. values from the$X * Y$column.

ID	X	Y	X*Y
1	156	54	8424
2	173	45	7785
3	183	89	16287
4	187	87	16269
5	176	76	13376
6	168	56	9408
Expected values	E[X]=173,83	E[Y]=67,83	E[X*Y]=11924,83

Now multiplicate variables expected values $E[X] * E[Y] = 173{,}83 * 67{,}83 = 11790{,}89$. Thus, according to the covariance equation$\sigma(s_1(t), s_2(t)) =: \sigma(X, Y) = 11924{,}83 - 11790{,}89 = 133{,}94$. The covariance coefficient is equal to 133,94.

```
'''
Script for calculating the covariance coefficient between two variables.
'''

import numpy as np

X = np.array([156, 173, 183, 187, 176, 168])
Y = np.array([54, 45, 89, 87, 76, 56])

# Expected value function.
def E(X, P):
    expectedValue = 0
    for i in np.arange(0, np.size(X)):
        expectedValue += X[i] * (P[i] / np.size(X))
    return expectedValue

# Covariance coefficient function.
def covariance(X, Y):
    '''
    Calculate the product of the multiplication for each pair of variables
    values.
    '''
    XY = X * Y

    # Calculate the expected values for each variable and for the XY.
    EX = E(X, np.ones(np.size(X)))
    EY = E(Y, np.ones(np.size(Y)))
```

```
    EXY = E(XY, np.ones(np.size(XY)))

    # Calculate the covariance coefficient.
    return EXY - (EX * EY)

# Display matrix of the covariance coefficient values.
covMatrix = np.array([[covariance(X, X), covariance(X, Y)], [covariance(Y, X),
covariance(Y, Y)]])
print("My function:", covMatrix)

# Display standard numpy.cov() covariance coefficient matrix for ddof=0.
print("Numpy.cov() function:", np.cov([X, Y], ddof=0))
```

2 Variance

Variance is a special case of the covariance when the two variables are identical. We can express it by equation:$Var(X) = E[X^2] - (E[X])^2$, because$Var(X) = \sigma(X,X)$. It is a basic measure of the variability of observed values, and it tells us how big differences are in the values for a given set (variable). In other words, does values are concentrated around the mean, are there small differences between the average and the individual results, or maybe the values scattering arround the mean is big.

```
'''
Calculate the variance coefficient for one variable.
'''

import numpy as np

X = np.array([60, 60, 60, 60, 70, 70, 70, 80, 80, 80, 80])

# Expected value function.
def E(X, P):
    expectedValue = 0
    for i in np.arange(0, np.size(X)):
        expectedValue += X[i] * (P[i] / np.size(X))
    return expectedValue

# Variance coefficient function.
def variance(X):
    '''
    Raise to power of 2 each value from X variable.
    '''
    XX = X ** 2

    # Calculate the expected values for X and for the XX.
    EX = E(X, np.ones(np.size(X)))
    EXX = E(XX, np.ones(np.size(XX)))

    # Calculate the covariance coefficient.
    return EXX - (EX ** 2)
```

```
print(variance(X))
```

3 Correlation

The correlation coefficient can be simply described as normalized covariance coefficient. It can be expressed by equation:$\rho_{XY} = \frac{\sigma_{XY}}{\sigma_X \sigma_Y}$, where$\sigma_{XY}$is the covariance coefficient between$X$and$Y$variables, andσ_X,σ_X, are their standard deviations, where$\sigma_k = \sqrt{E[(k - \mu)^2]}forE[k] = \mu$.

```
'''
Calculate the correlation coefficient between two variables.
'''

import numpy as np

X = np.array([156, 173, 183, 187, 176, 168])
Y = np.array([54, 45, 89, 87, 76, 56])

# Expected value function.
def E(X, P):
    expectedValue = 0
    for i in np.arange(0, np.size(X)):
        expectedValue += X[i] * (P[i] / np.size(X))
    return expectedValue

# Covariance coefficient function.
def covariance(X, Y):
    '''
    Calculate the product of the multiplication for each pair of variables
    values.
    '''
    XY = X * Y

    # Calculate the expected values for each variable and for the XY.
    EX = E(X, np.ones(np.size(X)))
    EY = E(Y, np.ones(np.size(Y)))
    EXY = E(XY, np.ones(np.size(XY)))

    # Calculate the covariance coefficient.
    return EXY - (EX * EY)

# Correlation coefficient function.
def correlation(X, Y):
    # Calculate the covariance coefficient.
    cov = covariance(X, Y)

    # Calculate standard deviations.
    EX = E(X, np.ones(np.size(X)))
    SDX = (E((X - EX) ** 2, np.ones(np.size(X)))) ** (1/2)

    EY = E(Y, np.ones(np.size(Y)))
```

```
    SDY = (E((Y - EY) ** 2, np.ones(np.size(Y)))) ** (1/2)

    # Calculate correlation coefficient.
    return cov / (SDX * SDY)

# Display matrix of the correlation coefficient values.
corrMatrix = np.array([[correlation(X, X), correlation(X, Y)], [correlation(Y,
X), correlation(Y, Y)]])
print("My function:", corrMatrix)

# Display standard numpy.corrcoef() correlation coefficient matrix for ddof=0.
print("Numpy.corrcoef() function:", np.corrcoef([X, Y], ddof=0))
```

4 Autocovariance

The autocovariance is a function that gives the covariance of the process with itself at pairs of time points. The autocovariance is given by $\gamma(k) = \sigma(x(i), x(i-k)) = E[(x(i) - \mu)(x(i-k) - \mu)]$, where $x(i)$ is a given signal (e.g vector $[0,1,2,3,4,5,6]$), $\mu = E[x(i)]$, and k is "the time shift" of the signal $x(i)$. In the case of discrete signals, k stands for shift of a given amount of $x(i)$ signal samples, and N is the size of the signal measured in samples. Thus, one can estimate the autocovariance coeficient with use of the equation:

$$\gamma(k) = \frac{1}{N-1}\sum_{i=0}^{N-k}(x(i+k) - x_s)\,(x(i) - x_s), \text{ where } x(s) = \frac{\sum_{i=0}^{N} x(i)}{N}.$$

```
'''
Calculate the autocovarriance coefficient for discrete signal.
'''

import numpy as np

Xi = np.array([1, 2, 3, 4, 5, 1, 2, 3, 4, 5, 1, 2, 3, 4, 5])
N = np.size(Xi)
k = 5
Xs = np.average(Xi)

def autocovariance(Xi, N, k, Xs):
    autoCov = 0
    for i in np.arange(0, N-k):
        autoCov += ((Xi[i+k])-Xs)*(Xi[i]-Xs)
    return (1/(N-1))*autoCov

print("Autocovariance:", autocovariance(Xi, N, k, Xs))
```

5 Autocorrelation

The autocorrelation coefficient is just a normalized version of the autocovariance coefficient. In the

case of discrete signal one can express it as $\rho(k) = \frac{\gamma(k)}{\gamma(0)}$.

```
'''
Calculate the autocorrelation coefficient for discrete signal.
'''

import numpy as np

Xi = np.array([1, 2, 3, 4, 5, 1, 2, 3, 4, 5, 1, 2, 3, 4, 5])
N = np.size(Xi)
k = 5
Xs = np.average(Xi)

def autocovariance(Xi, N, k, Xs):
    autoCov = 0
    for i in np.arange(0, N-k):
        autoCov += ((Xi[i+k])-Xs)*(Xi[i]-Xs)
    return (1/(N-1))*autoCov

def autocorrelation():
    return autocovariance(Xi, N, k, Xs) / autocovariance(Xi, N, 0, Xs)

print("Autocorrelation:", autocorrelation())
```

6 Autocovariance and autocorrelation - the result in the form of a vector

In order to get a vector as a result of autocovariance or autocorrelation function, one have to calculate coefficients for the chosen function for each k value from the range of $\langle |-k|; k \rangle$, where $k \in \mathbb{N}_{positive} \cup \{0\}$. Code shown below, for input $x_i = [1,2,3]$ and $k = 2$, will return the vector of $[-0.5,0,1,0,-0.5]$.

```
'''
Autocovariance and autocorrelation functions - the result
in the form of a vector.
'''

import numpy as np

Xi = np.array([1, 2, 3])
N = np.size(Xi)
k = 2
Xs = np.average(Xi)

def autocovariance(Xi, N, k, Xs):
    autoCov = 0
    for i in np.arange(0, N-k):
```

```
        autoCov += (Xi[i+k]-Xs)*(Xi[i]-Xs)
    return (1/(N-1))*autoCov

def autocorrelation():
    return autocovariance(Xi, N, k, Xs) / autocovariance(Xi, N, 0, Xs)

def array(k, norm = True):
    # If norm = True, return array of autocorrelation coefficients.
    # If norm = False, return array of autocovariance coefficients.
    vector = np.array([])
    shifts = np.abs(np.arange(-k, k+1, 1))
    for i in shifts:
        if norm == True:
            vector = np.append(autocovariance(Xi, N, i, Xs) /
autocovariance(Xi, N, 0, Xs), vector)
        else:
            vector = np.append(autocovariance(Xi, N, i, Xs), vector)
    return vector

print("Array as a result:", array(k))
```

7 Autocorrelogram

Autocorrelogram is a visualisation of $\rho(k)$ function for $\langle |-k|; k \rangle$, where $k \in \mathbb{N}_{positive} \cup \{0\}$. The example autocorrelogram for the signal described as: $f(x) = \cos(4\pi x) + \cos(6\pi x) + \cos(8\pi x)$, for $x = np.linspace(0,10,2000)$ and $k = 2000$ will look like I shown on Fig. 2.

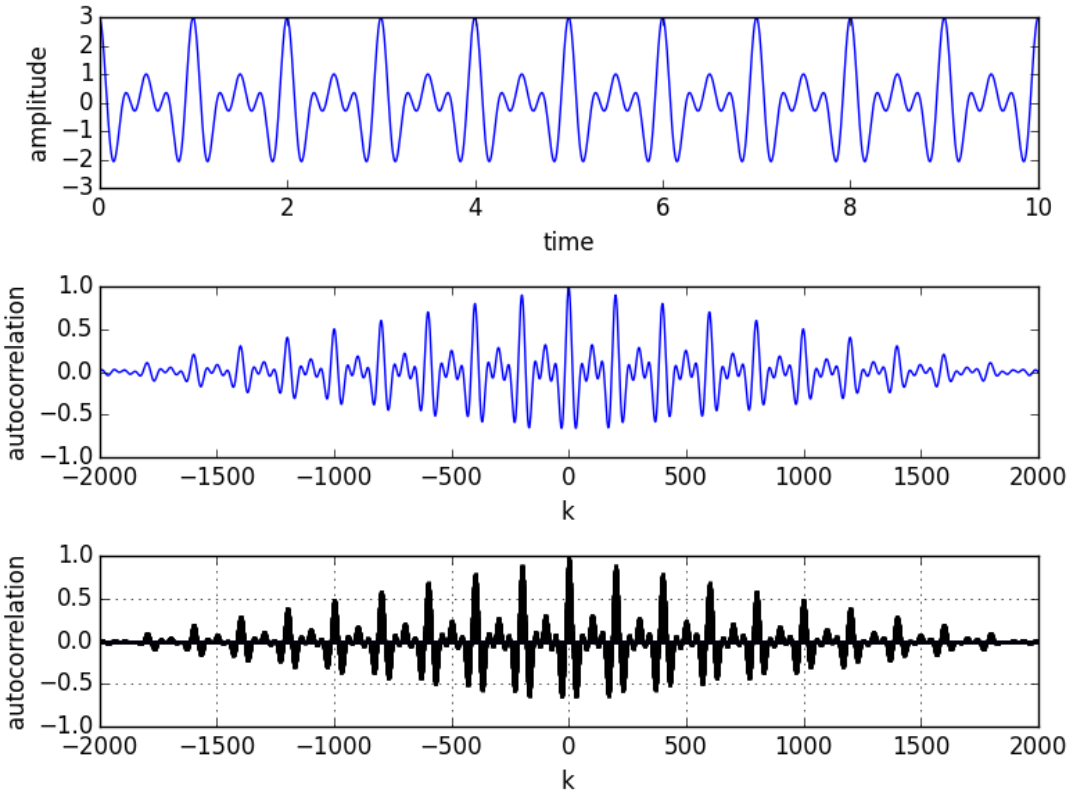

Figure 2: Description of the diagrams from the top: 1)$f(x)$function; 2) autocorrelogram of the$f(x)$function implemented by me; 3) autocorrelogram of the $f(x)$ function plotted with the use of the *matplotlib.pyplot.acorr* function.

```
import numpy as np
import pylab as py

# Define the signal.
x = np. linspace(0, 10, 2000)
signal = np.cos(4*np.pi*x) + np.cos(6*np.pi*x) + np.cos(8*np.pi*x)

# Plot the signal.
py.subplot(3, 1, 1)
py.plot(x, signal)
py.xlabel('time')
py.ylabel('amplitude')

# Autocorrelation function of the signal.
Xi = signal
N = np.size(Xi)
k = 2000
Xs = np.average(Xi)

def autocovariance(Xi, N, k, Xs):
    autoCov = 0
    for i in np.arange(0, N-k):
        autoCov += (Xi[i+k]-Xs)*(Xi[i]-Xs)
    return (1/(N-1))*autoCov
```

```
def autocorrelation():
    return autocovariance(Xi, N, k, Xs) / autocovariance(Xi, N, 0, Xs)

def array(k, norm = True):
    # If norm = True, return array of autocorrelation coefficients.
    # If norm = False, return array of autocovariance coefficients.
    vector = np.array([])
    shifts = np.abs(np.arange(-k, k+1, 1))
    for i in shifts:
        if norm == True:
            vector = np.append(autocovariance(Xi, N, i, Xs) /
autocovariance(Xi, N, 0, Xs), vector)
        else:
            vector = np.append(autocovariance(Xi, N, i, Xs), vector)
    return vector

def autocorrelogram(k, ki, norm = True):
    k = np.arange(-k, k+1, 1)
    py.subplot(3, 1, 2)
    py.plot(k, ki)
    py.xlabel('k')

    if norm == True:
        py.ylim([-1, 1])
        py.ylabel('autocorrelation')
    else:
        py.ylim([np.min(ki), np.max(ki)])
        py.ylabel('autocovariance')

autocorrelogram(k, array(k))

# Plot autocorrelation function with the use of matplotlib.pyplot.acorr.
ax = py.subplot(3, 1, 3)
py.ylabel('autocorrelation')
py.xlabel('k')
py.ylim([-1, 1])
ax.acorr(signal, usevlines = True, normed = True, maxlags = None, lw = 2)
ax.grid(True)
ax.axhline(0, color='black', lw=2)

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

8 Interdependence of two signals – cross-covariance & cross-correlation

In order to characterize the mutual dependence of two signals, a covariance function is used. In the case of discrete signals one can express it as: $\gamma_{xy} = \frac{1}{N-1}\sum_{i=0}^{N-k}(x(i+k) - x_s)\,(y(i) - y_s)$. The covariance function can be normalized: $\rho(k) = \frac{\gamma_{xy}}{\sigma_x \sigma_y}$. The resulting function is called the correlation

function.

```
'''
Calculate the cross-covariance and cross-correlation coefficients
for two random discrete signals.
'''

import numpy as np

# x(i) and y(i) must have equal size.
Xi = np.array([1, 2, 3, 4, 5])
Yi = np.array([5, 4, 3, 2, 1])
N = np.size(Xi)
k = 2
Xs = np.average(Xi)
Ys = np.average(Yi)

# Covariance coefficient functions.
def covariance(Xi, Yi, N, k, Xs, Ys, forCorrelation = False):
    autoCov = 0
    for i in np.arange(0, N-k):
        autoCov += ((Xi[i+k])-Xs)*(Yi[i]-Ys)
    if forCorrelation == True:
        return autoCov/N
    else:
        return (1/(N-1))*autoCov

# Expected value function.
def E(X, P):
    expectedValue = 0
    for i in np.arange(0, np.size(X)):
        expectedValue += X[i] * (P[i] / np.size(X))
    return expectedValue

# Correlation coefficient function.
def correlation(Xi, Yi, k):
    # Calculate the covariance coefficient.
    cov = covariance(Xi, Yi, N, k, Xs, Ys, forCorrelation = True)

    # Calculate standard deviations.
    EX = E(Xi, np.ones(np.size(Xi)))
    SDX = (E((Xi - EX) ** 2, np.ones(np.size(Xi)))) ** (1/2)

    EY = E(Yi, np.ones(np.size(Yi)))
    SDY = (E((Yi - EY) ** 2, np.ones(np.size(Yi)))) ** (1/2)

    # Calculate correlation coefficient.
    return cov / (SDX * SDY)

def array(k, norm = True):
    # If norm = True, return array of autocorrelation coefficients.
    # If norm = False, return array of autocovariance coefficients.
    vector = np.array([])
    shifts = np.abs(np.arange(-k, k+1, 1))
    for i in shifts:
```

```
        if norm == True:
            vector = np.append(correlation(Xi, Yi, i), vector)
        else:
            vector = np.append(covariance(Xi, Yi, N, i, Xs, Ys), vector)
    return vector

print("Covariance:", covariance(Xi, Yi, N, k, Xs, Ys))
print("Correlation:", correlation(Xi, Yi, k))
print("Array as a result:", array(k, norm = True))
```

8.1 Example

Say, one want to check interdependance of gathered raw EEG data (Fig. 3) from three given channels, i.e Fp1, Fp2, and F7, with total1000samples per channel, sampling frequency equal to$128Hz$, and signal shift equal to$k = 128$.

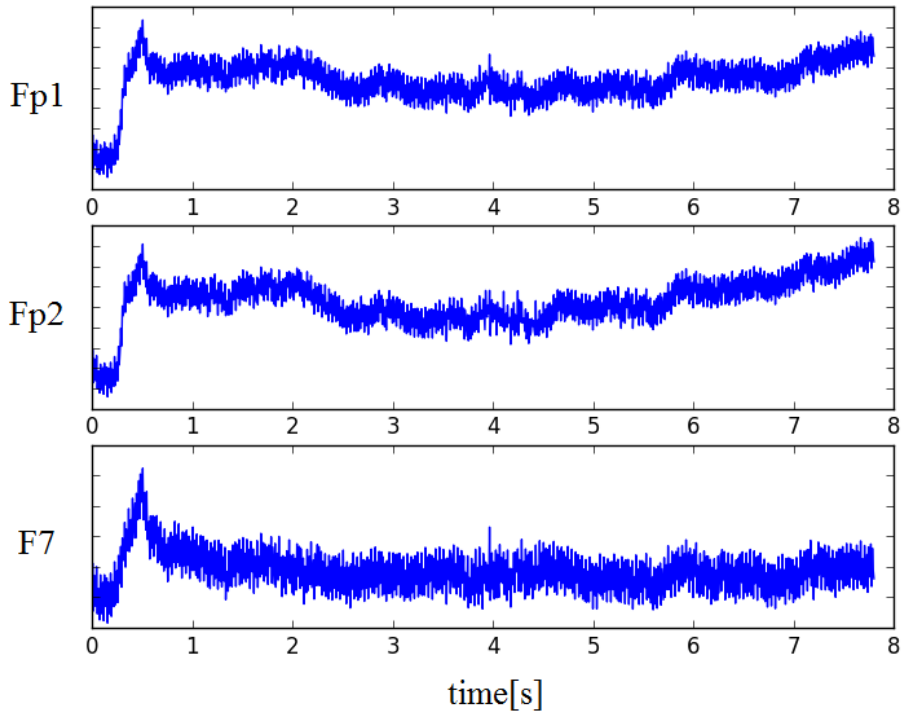

Figure 3: EEG raw signal data gathered from three channels - Fp1, Fp2, F7.

In this particular situation one could examine nine cases of the correlation between channels: Fp1/Fp1, Fp1/Fp2, Fp1/F7, Fp2/Fp1, Fp2/Fp2, Fp2/F7, F7/Fp1, F7/Fp2, F7/F7. As a result of such examination one can next plot 3x3 matrix of$\rho(k)$functions for each channel pair, i.e nine correlograms, as shown on Fig. 4.

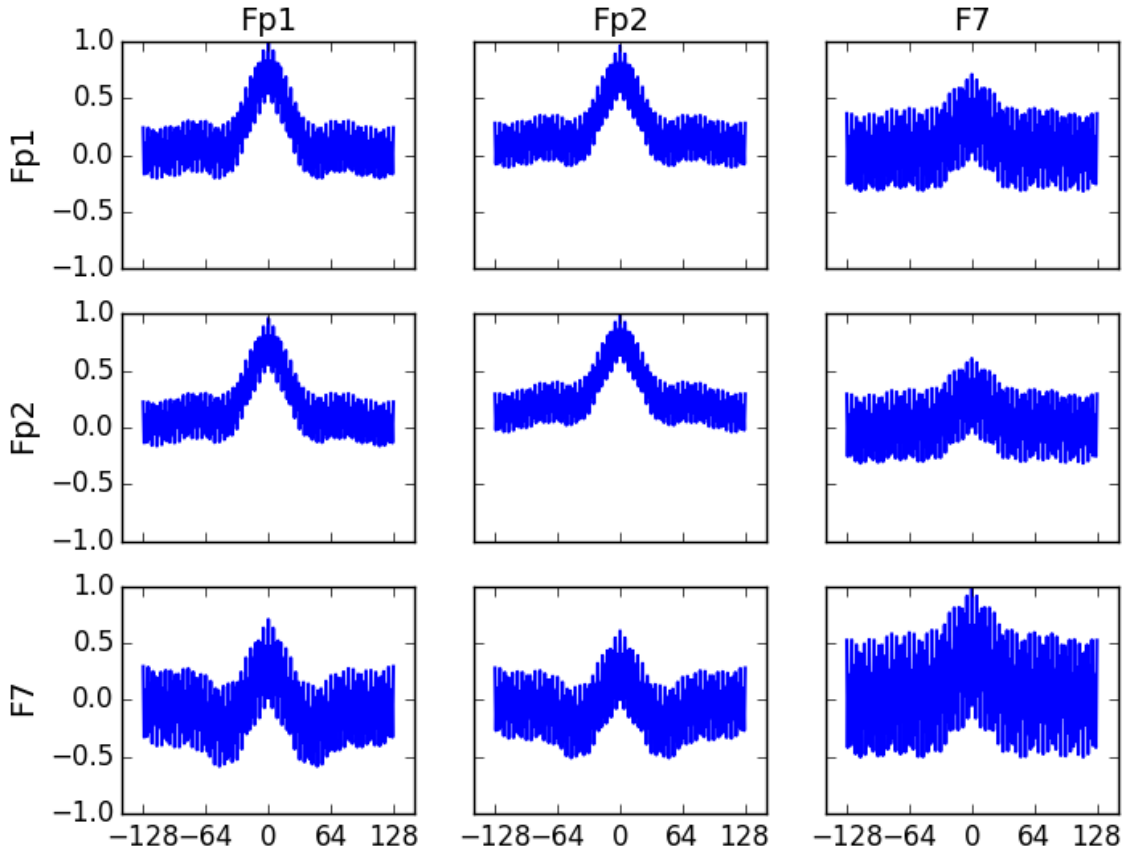

Figure 4: Correlograms for each channel pair.

```
'''
Calculate cros-channel correlation coefficients and plot the results.
'''

import numpy as np
import pylab as py

# Load *.raw file.
signal =
np.fromfile("data/artefakty_praceeg_cw1_grupa_kpm_10zamknieteoczy.raw", dtype =
"float32")

'''
Reshape and transpose loaded file. Remove chosen channels, eg. "T[:-2]"
will remove 2 last channels. Multiply by amplifier error, ie "0.0715".
Every amplifier can have different error.
'''
channelsTotalNumber = 23
amplifierError = 0.0715
signal = (signal.reshape((-1, channelsTotalNumber)).T[:-2]) * amplifierError

# Choose channels.
chosenChannels = np.array([signal[0, :], signal[1, :], signal[2, :]])
```

```
# Get other values.
samplingFrequency = 128.0

# Covariance coefficient functions.
def covariance(Xi, Yi, Xs, Ys, k, forCorrelation = False):
    N = np.size(Xi)
    autoCov = 0
    for i in np.arange(0, N-k):
        autoCov += ((Xi[i+k])-Xs)*(Yi[i]-Ys)
    if forCorrelation == True:
        return autoCov/N
    else:
        return (1/(N-1))*autoCov

# Expected value function.
def E(X, P):
    expectedValue = 0
    for i in np.arange(0, np.size(X)):
        expectedValue += X[i] * (P[i] / np.size(X))
    return expectedValue

# Correlation coefficient function.
def correlation(Xi, Yi, Xs, Ys, k):
    # Calculate the covariance coefficient.
    cov = covariance(Xi, Yi, Xs, Ys, k, forCorrelation = True)

    # Calculate standard deviations.
    EX = E(Xi, np.ones(np.size(Xi)))
    SDX = (E((Xi - EX) ** 2, np.ones(np.size(Xi)))) ** (1/2)

    EY = E(Yi, np.ones(np.size(Yi)))
    SDY = (E((Yi - EY) ** 2, np.ones(np.size(Yi)))) ** (1/2)

    # Calculate correlation coefficient.
    return cov / (SDX * SDY)

def array(Xi, Yi, Xs, Ys, k, norm = True):
    # If norm = True, return array of autocorrelation coefficients.
    # If norm = False, return array of autocovariance coefficients.
    vector = np.array([])
    shifts = np.abs(np.arange(-k, k+1, 1))
    for i in shifts:
        if norm == True:
            vector = np.append(correlation(Xi, Yi, Xs, Ys, i), vector)
        else:
            vector = np.append(covariance(Xi, Yi, Xs, Ys, i), vector)
    return vector

# Calculate cros-channel correlation coefficients and plot the results.
k = 128 # 1s
x = np.arange(-k, k+1, 1)
x_ticks = np.arange(-k, k+1, (k*2)/4)

Fp1 = chosenChannels[0, :1000]
Fp2 = chosenChannels[1, :1000]
F7 = chosenChannels[2, :1000]
```

```
Fp1_Fp1 = array(Fp1, Fp1, np.average(Fp1), np.average(Fp1), k)
print("Done Fp1_Fp1")
Fp1_Fp2 = array(Fp1, Fp2, np.average(Fp1), np.average(Fp2), k)
print("Done Fp1_Fp2")
Fp1_F7 = array(Fp1, F7, np.average(Fp1), np.average(F7), k)
print("Done Fp1_F7")

Fp2_Fp1 = array(Fp2, Fp1, np.average(Fp2), np.average(Fp1), k)
print("Done Fp2_Fp1")
Fp2_Fp2 = array(Fp2, Fp2, np.average(Fp2), np.average(Fp2), k)
print("Done Fp2_Fp2")
Fp2_F7 = array(Fp2, F7, np.average(Fp2), np.average(F7), k)
print("Done Fp2_F7")

F7_Fp1 = array(F7, Fp1, np.average(F7), np.average(Fp1), k)
print("Done F7_Fp1")
F7_Fp2 = array(F7, Fp2, np.average(F7), np.average(Fp2), k)
print("Done F7_Fp2")
F7_F7 = array(F7, F7, np.average(F7), np.average(F7), k)
print("Done F7_F7")

ax1 = py.subplot(3, 3, 1)
ax1.axes.xaxis.set_ticklabels([])
py.plot(x, Fp1_Fp1, linewidth=1.5)
py.xticks(x_ticks)
py.ylim([-1, 1])
ax1.set_ylabel("Fp1", fontsize=14)
ax1.set_xlabel("Fp1", fontsize=14)
ax1.xaxis.set_label_position('top')

ax2 = py.subplot(3, 3, 2)
ax2.axes.xaxis.set_ticklabels([])
ax2.axes.yaxis.set_ticklabels([])
py.plot(x, Fp1_Fp2, linewidth=1.5)
py.xticks(x_ticks)
py.ylim([-1, 1])
ax2.set_xlabel("Fp2", fontsize=14)
ax2.xaxis.set_label_position('top')

ax3 = py.subplot(3, 3, 3)
ax3.axes.xaxis.set_ticklabels([])
ax3.axes.yaxis.set_ticklabels([])
py.plot(x, Fp1_F7, linewidth=1.5)
py.xticks(x_ticks)
py.ylim([-1, 1])
ax3.set_xlabel("F7", fontsize=14)
ax3.xaxis.set_label_position('top')

ax4 = py.subplot(3, 3, 4)
ax4.axes.xaxis.set_ticklabels([])
py.plot(x, Fp2_Fp1, linewidth=1.5)
py.xticks(x_ticks)
py.ylim([-1, 1])
ax4.set_ylabel("Fp2", fontsize=14)
```

```
ax5 = py.subplot(3, 3, 5)
ax5.axes.xaxis.set_ticklabels([])
ax5.axes.yaxis.set_ticklabels([])
py.plot(x, Fp2_Fp2, linewidth=1.5)
py.xticks(x_ticks)
py.ylim([-1, 1])

ax6 = py.subplot(3, 3, 6)
ax6.axes.xaxis.set_ticklabels([])
ax6.axes.yaxis.set_ticklabels([])
py.plot(x, Fp2_F7, linewidth=1.5)
py.xticks(x_ticks)
py.ylim([-1, 1])

ax7 = py.subplot(3, 3, 7)
py.plot(x, F7_Fp1, linewidth=1.5)
py.xticks(x_ticks)
py.ylim([-1, 1])
ax7.set_ylabel("F7", fontsize=14)

ax8 = py.subplot(3, 3, 8)
ax8.axes.yaxis.set_ticklabels([])
py.plot(x, F7_Fp2, linewidth=1.5)
py.xticks(x_ticks)
py.ylim([-1, 1])

ax9 = py.subplot(3, 3, 9)
ax9.axes.yaxis.set_ticklabels([])
py.plot(x, F7_F7, linewidth=1.5)
py.xticks(x_ticks)
py.ylim([-1, 1])

py.show()
```

8.2 Savitzky-Golay filter

In order to make the correlograms from Fig. 4 more readable and easier to interpret, one can apply to the$\rho(k)$functions an appropiate low-pass filter. One of the possible available solutions is the Savitzky-Golay filter. It works by process of convolution, by fitting successive sub-sets of adjacent data points with a low-degree polynomial by the method of linear least squares. When the data points are equally spaced an analytical solution to the least-squares equations can be found, in the form of a single set of convolution coefficients that can be applied to all data sub-sets, to give estimates of the smoothed signal, at the central point of each sub-set. The application of Savitzky-Golay filter in practice is very easy. Say, the data consists of a set of$n\{x_j, y_j\}$points, where$x$is an independent variable and$y$is an observed value. They are treated with a set of m convolution coefficients, C_i according to the expression: $Y_j = \sum_{i=-(m-1)/2}^{i=(m-1)/2} C_i y_{j+i}$ for $\frac{m+1}{2} \leqslant j \leqslant n - \frac{m-1}{2}$. C_i is an elemets of the matrix $C = (J^T J)^{-1} J^T$, where $J = \frac{\partial Y}{\partial a}, Y = a_0 + a_1 z + a_2 z^2 + \ldots + a_k z^k, a = (J^T J)^{-1} J^T y, z = \frac{x_j - \overline{x_j}}{h}$, and h as an

constant interval between each$x_j$value.

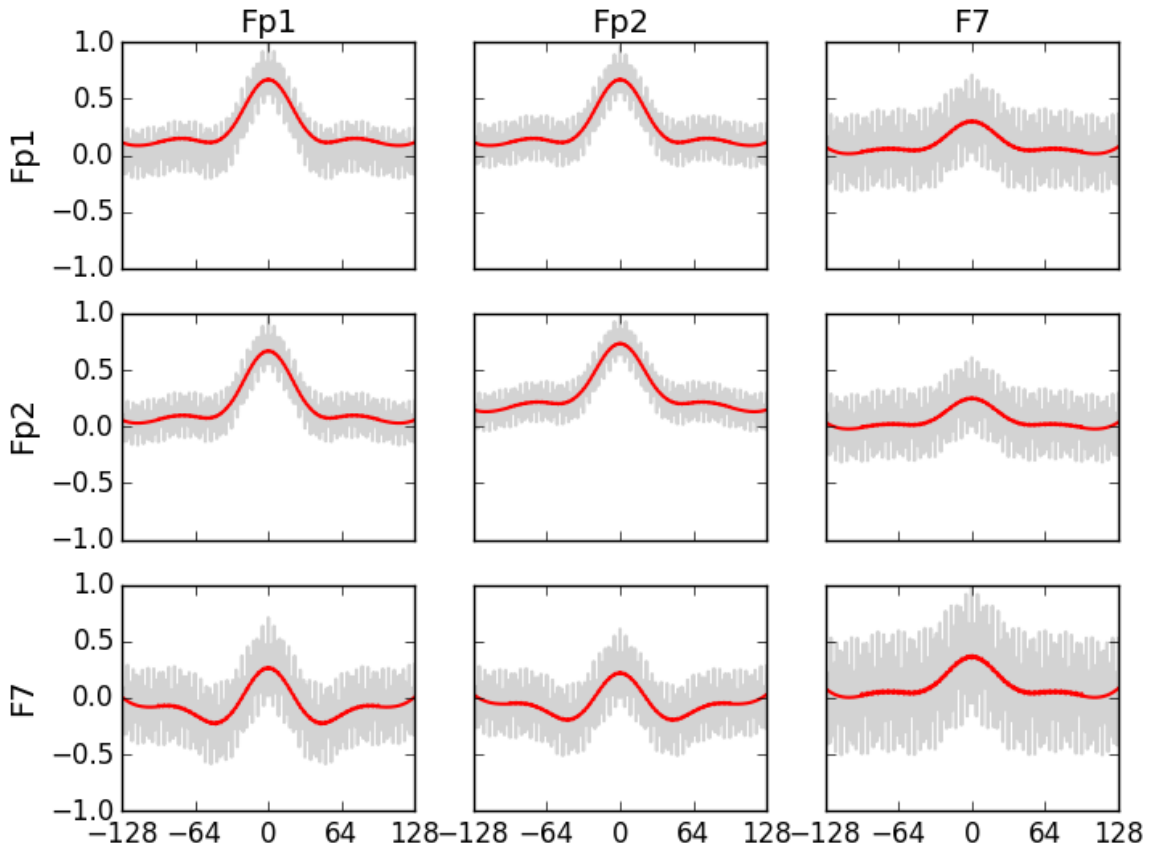

Figure 5: Savitzky-Golay filter applied to the correlograms from Fig. 4.

The Fig. 5 illustrates the application of the Savitzky-Golay filter to the correlograms from the Fig. 4 for the 3-point quadtratic polynomial and$m = 63$. The use of the above-mentioned filter in practice can look like this:

```
import numpy as np
import pylab as py
import scipy.signal as sig

x = np.arange(0, 250, 1)
y = np.random.rand(250)
py.plot(x, y, linewidth=1.5, color='lightgray')
py.plot(x, sig.savgol_filter(y, 61, 3), linewidth=1.5, color='red')

py.show()
```

And the scipy.signal.savgol_filter() implementation:

```
def savgol_filter(x, window_length, polyorder, deriv=0, delta=1.0, axis=-1,
```

```
mode='interp', cval=0.0):

    if mode not in ["mirror", "constant", "nearest", "interp", "wrap"]:
        raise ValueError("mode must be 'mirror', 'constant', 'nearest' "
                         "'wrap' or 'interp'.")

    x = np.asarray(x)
    # Ensure that x is either single or double precision floating point.
    if x.dtype != np.float64 and x.dtype != np.float32:
        x = x.astype(np.float64)

    coeffs = savgol_coeffs(window_length, polyorder, deriv=deriv, delta=delta)

    if mode == "interp":
        # Do not pad.  Instead, for the elements within `window_length // 2`
        # of the ends of the sequence, use the polynomial that is fitted to
        # the last `window_length` elements.
        y = convolve1d(x, coeffs, axis=axis, mode="constant")
        _fit_edges_polyfit(x, window_length, polyorder, deriv, delta, axis, y)
    else:
        # Any mode other than 'interp' is passed on to ndimage.convolve1d.
        y = convolve1d(x, coeffs, axis=axis, mode=mode, cval=cval)

    return y
```

9 Fourier series

Any periodic function$f(x+T)=f(x)$, where$T$is the period of this function, can be expressed as a series of simple sine and cosine waves, in which the increasing number of its components (n) also increases the accuracy with which the serries reproduce the function$f(x)$. This series is called the Fourier series and it can be determined as$Sf(x)$. It is expressed by the following formulas:

$$Sf(x)=\frac{a_0}{2}+\sum_{n=1}^{\infty}[a_n\cos\frac{n\pi x}{L}+b_n\sin\frac{n\pi x}{L}],\text{ where}L=\frac{T}{2},$$

$$a_n=\frac{1}{L}\int_{-L}^{L}\quad f(x)\cos\frac{n\pi x}{L}dx\text{and}b_n=\frac{1}{L}\int_{-L}^{L}\quad f(x)\sin\frac{n\pi x}{L}dx.$$

So lets say, that one want to evaluate the Fourier series for theoretical signal expressed by the$f(x)=\sin\pi x+\sin2\pi x+\sin5\pi x$function for$x=np.linspace(0,10,1000)$with the period of$T=2$and for then$n=5$. Figure 6 ilustrates the plotted signal function and its Fourier series representation.

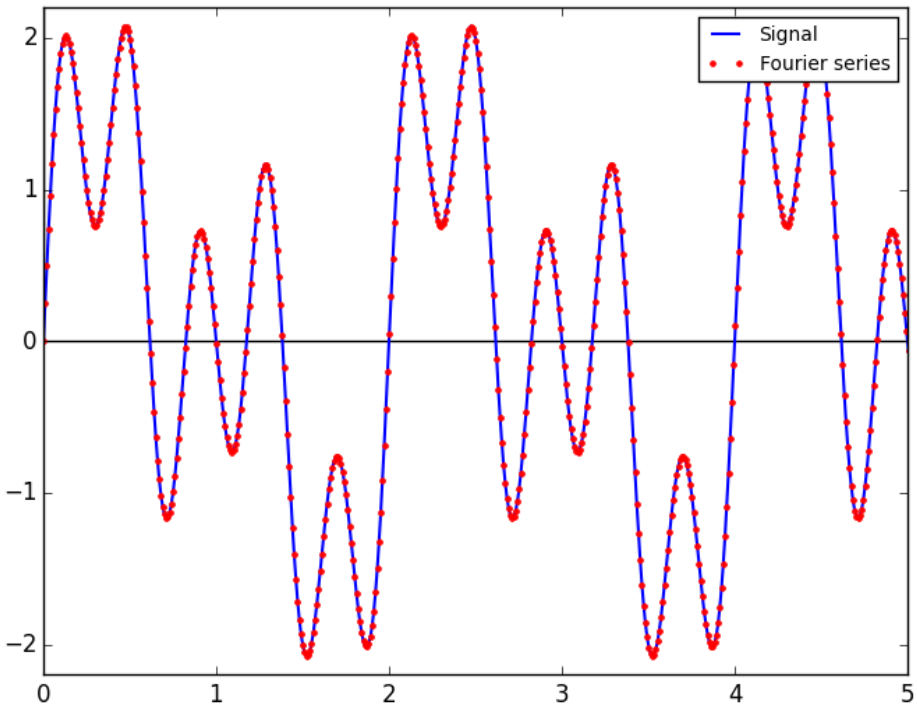

Figure 6: Theoretical signal for $f(x) = \sin\pi x + \sin 2\pi x + \sin 5\pi x$ and its Fourier series representation.

One can also dismantle the signal into pieces and see the elements of which it consists. In order to do this, just calculate the $Sf(x)$ function for individual n values (Fig. 7).

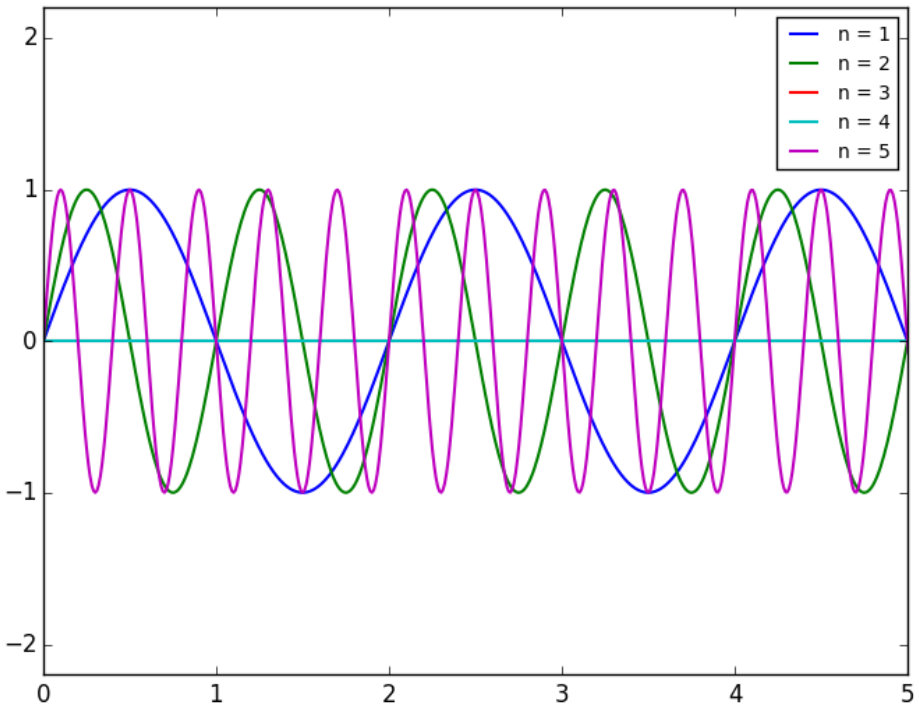

Figure 7: Individual components of the $f(x) = \sin\pi x + \sin 2\pi x + \sin 5\pi x$ function.

```
'''
Calculate the Fourier series with the trigonometry approach.
'''
from __future__ import division
import numpy as np
import pylab as py

# Define "x" range.
x = np.linspace(0, 10, 1000)

# Define "T", i.e functions' period.
T = 2
L = T / 2

# "f(x)" function definition.
def f(x):
    return np.sin((np.pi) * x) + np.sin((2 * np.pi) * x) + np.sin((5 * np.pi) *
x)

# "a" coefficient calculation.
def a(n, L, accuracy = 1000):
    a, b = -L, L
    dx = (b - a) / accuracy
    integration = 0
    for x in np.linspace(a, b, accuracy):
        integration += f(x) * np.cos((n * np.pi * x) / L)
    integration *= dx
```

```
    return (1 / L) * integration

# "b" coefficient calculation.
def b(n, L, accuracy = 1000):
    a, b = -L, L
    dx = (b - a) / accuracy
    integration = 0
    for x in np.linspace(a, b, accuracy):
        integration += f(x) * np.sin((n * np.pi * x) / L)
    integration *= dx
    return (1 / L) * integration

# Fourier series.
def Sf(x, L, n = 5):
    a0 = a(0, L)
    sum = np.zeros(np.size(x))
    for i in np.arange(1, n + 1):
        sum += ((a(i, L) * np.cos((i * np.pi * x) / L)) + (b(i, L) * np.sin((i
* np.pi * x) / L)))
    return (a0 / 2) + sum

# Individual component of the f(x) function.
def Sf_individualComponent(x, L, n):
    a0 = a(0, L)
    return (a0 / 2) + ((a(n, L) * np.cos((n * np.pi * x) / L)) + (b(n, L) *
np.sin((n * np.pi * x) / L)))

# x axis.
py.plot(x, np.zeros(np.size(x)), color = 'black')

# y axis.
py.plot(np.zeros(np.size(x)), x, color = 'black')

# Original signal.
py.plot(x, f(x), linewidth = 1.5, label = 'Signal')

# Approximation signal (Fourier series coefficients).
py.plot(x, Sf(x, L), '.', color = 'red', linewidth = 1.5, label = 'Fourier
series')

# Specify x and y axes limits.
py.xlim([0, 5])
py.ylim([-2.2, 2.2])

py.legend(loc = 'upper right', fontsize = '10')

py.show()
```

Additionaly, one can use above code to study other functions. Say, $f(x) = x^3$ for $x = np.arange(-1,1,0.01)$. In this case, the signal and the Fourier series representation for$n = 10$will look like shown on Fig. 8. Figure 9 also shows Fourier series decomposition on its individual components.

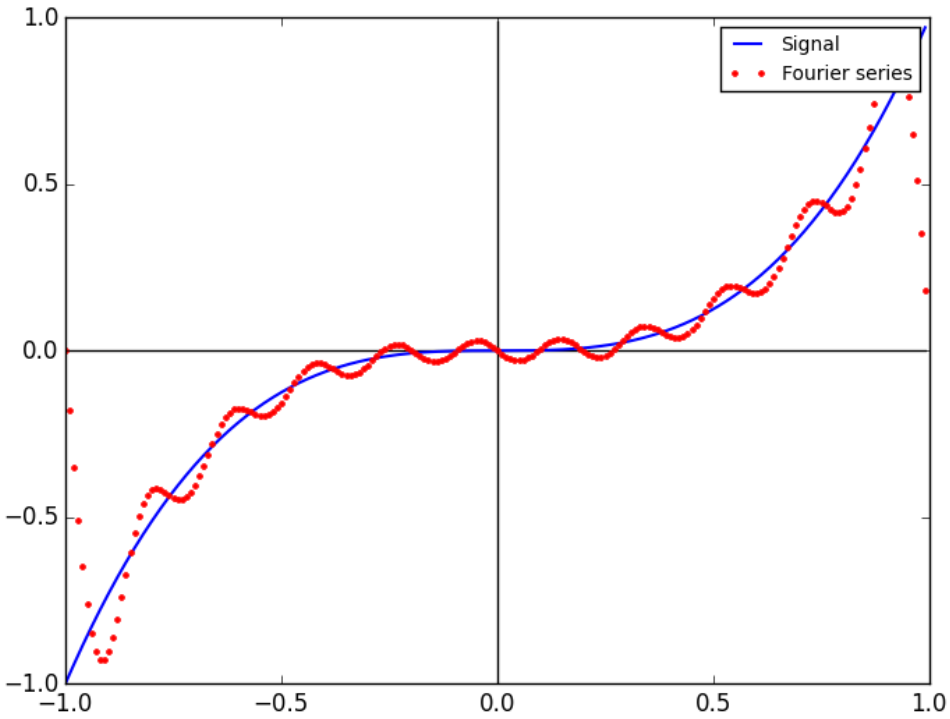

Figure 8: Theoretical signal for $f(x) = x^3$ and its Fourier series representation for $T = 2$ and $n = 10$.

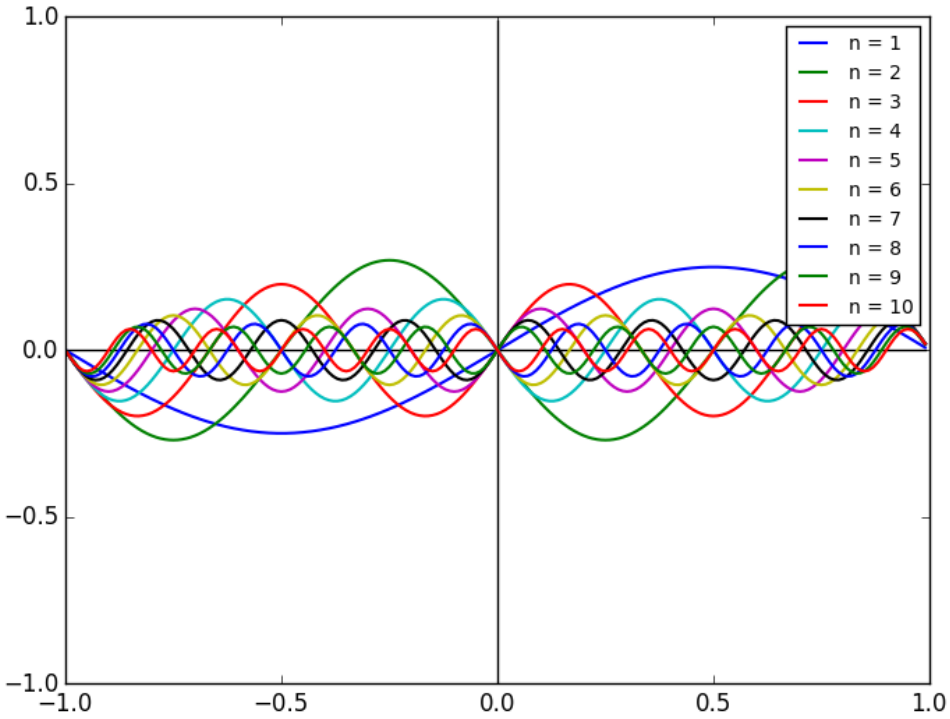

Figure 9: Individual components of the Fourier series calculated for $f(x) = x^3$ function where $n = 10$.

10 Fourier transform

Fourier series is a good tool for investigating the functions describing the periodic nature phenomena. The seek for the analogue of the Fourier series for the non-periodic (aperiodic) functions leads to the concept of the Fourier transform.

The Fourier transform decomposes a function of, say, time (a signal) into the frequencies that make it up (Fig. 10). The Fourier transform of a function of time itself is a complex-valued function of frequency, whose absolute value represents the amount of that frequency present in the original function, and whose complex argument is the phase offset of the basic sinusoid in that frequency. The Fourier transform is called the *frequency domain representation* of the original signal. Moreover, the term *Fourier transform* refers to both: the frequency domain representation and the mathematical operation that associates the frequency domain representation to a function of time.

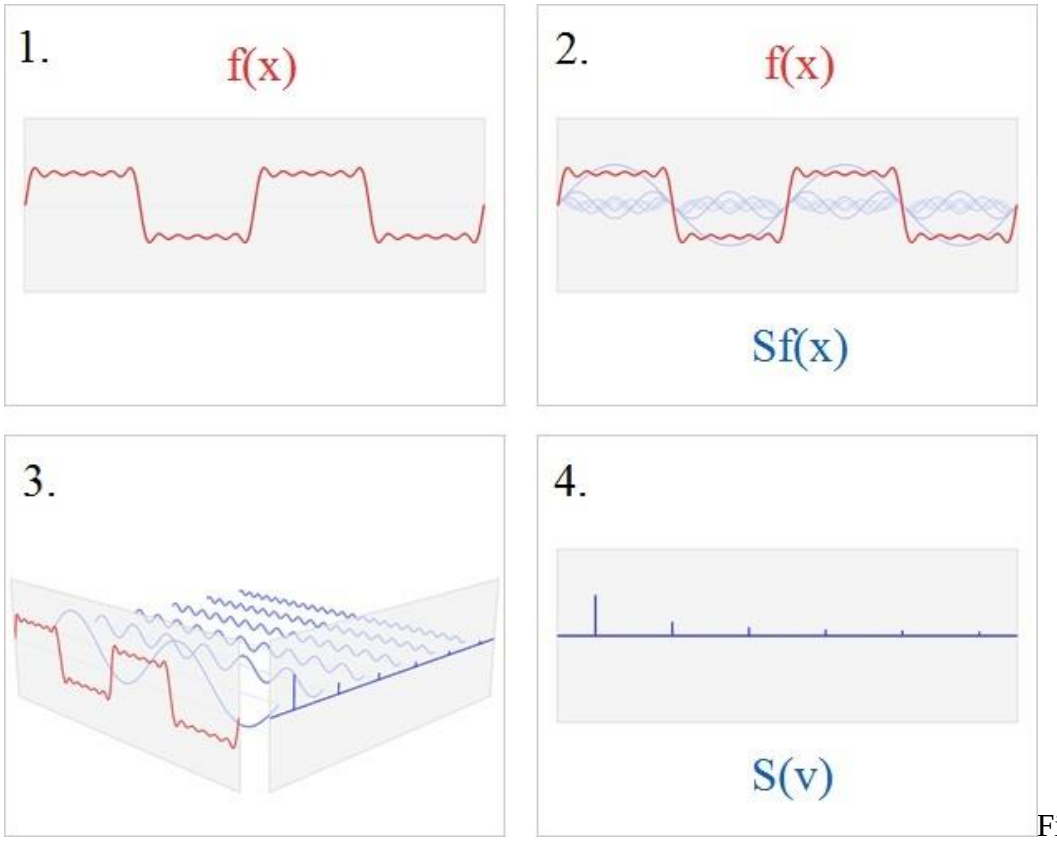

Figure 10: Diagram illustrates: 1. some function, 2. some functionwith its, i.e Fourier series individual components representation; 3. and 4. illustrates the Fourier transform of the function, i.e.

The Fourier transform can be expressed as:

$$S(v) = \int_{-\infty}^{\infty} \quad f(x)e^{-ivx}dx,$$

where$v$stands for particular transformation component, and$f(x)$is a specific function for which one want to perform the transform. But how can we calculate the Fourier transform? In the case of the discrete signal we could use the Discrete Fourier Transform (DFT) algorithm. However, DFT is very sluggish in comparison to the Fast Fourier Transform (FFT), which is a well optimized and a little bit modified version of the DFT (thanks to James W. Cooley and John W. Turkey). How sluggish DFT is in comparison to FFT we will see later on the working example. But now let's look at the two algorithms closer.

10.1 DFT

The DFT algorithm can be expressed by the following formula:

$$X_k = \sum_{n=0}^{N-1} \quad x_n e^{-i2\pi kn/N},$$

where$N$describes the number of samples of our signal,$x_n$is the n'th sample of it, and$k$stands for the k'th transform element of the X transformation. According to the above equation and

considering X and x as simple one dimensional vectors, we can express the DFT algorithm just in one line, i.e:

$$\vec{X} = M * \vec{x} \text{ with the matrix } M \text{ given by } M_{kn} = e^{-i2\pi kn/N}.$$

Thus, the source code for the DFT calculation can look like I shown below. Additionaly, its result is illustrated on Fig. 11.

```
import numpy as np
import pylab as py

# Define "x" range.
x = np.random.random(1024)

py.subplot(2, 1, 1)
py.plot(np.arange(np.size(x)), x)
py.xlim([0, 1024])

# Discrete Fourier Transform.
def DFT(x):
    x = np.asarray(x, dtype = float)
    N = np.size(x)
    n = np.arange(N)
    k = n.reshape((N, 1))
    M = np.exp(-(2j*np.pi*k*n)/N)

    return np.dot(M, x)

py.subplot(2, 1, 2)
py.plot(np.arange(np.size(x)), DFT(x))
py.xlim([0, 1024])
py.ylim([-50, 50])

py.show()
```

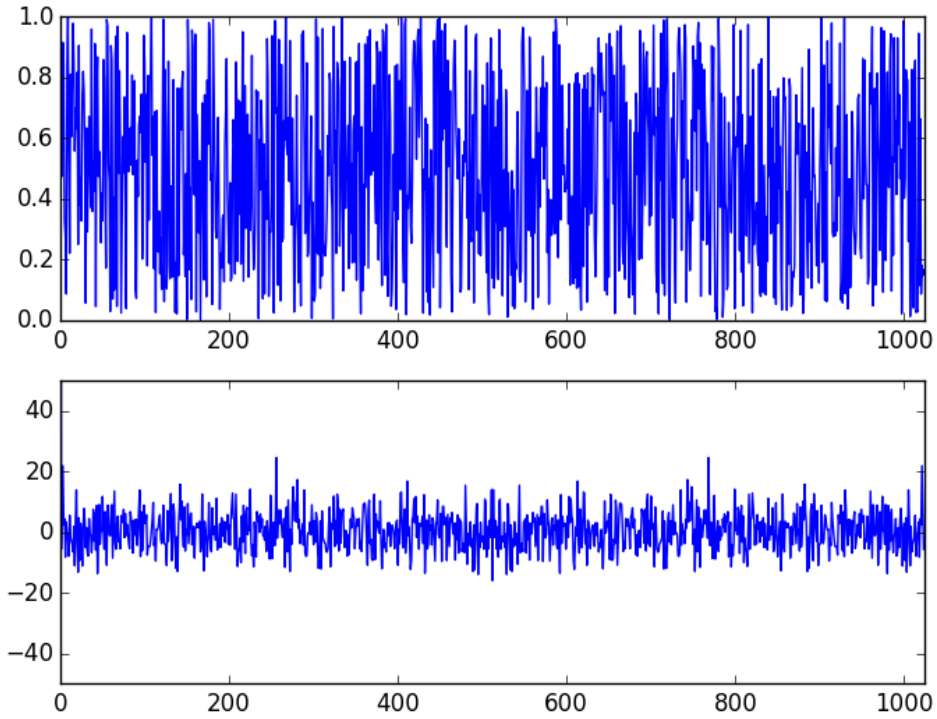

Figure 11: DFT in time domain performed for the "signal" builded of random values.

10.2 FFT

On the other hand we have the FFT algorithm, which one can implement as I shown below:

```
import numpy as np
import pylab as py

# Define "x" range.
x = np.random.random(1024)

py.subplot(2, 1, 1)
py.plot(np.arange(np.size(x)), x)
py.xlim([0, 1024])

# Discrete Fourier Transform.
def DFT(x):
    x = np.asarray(x, dtype = float)
    N = np.size(x)
    n = np.arange(N)
    k = n.reshape((N, 1))
    M = np.exp(-(2j*np.pi*k*n)/N)

    return np.dot(M, x)
```

```
# Fast Fourier Transform.
def FFT(x):
    x = np.asarray(x, dtype = float)
    N = np.size(x)

    if N % 2 > 0:
        raise ValueError("Size of x must be a power of 2")
    elif N <= 32: # this cutoff shoulb be optimized
        return DFT(x)
    else:
        X_even = FFT(x[::2])
        X_odd = FFT(x[1::2])
        factor = np.exp(-2j*np.pi*np.arange(N)/N)
        return np.concatenate([X_even + factor[:N / 2] * X_odd, X_even +
factor[N / 2:] * X_odd])

py.subplot(2, 1, 2)
py.plot(np.arange(np.size(x)), FFT(x))
py.xlim([0, 1024])
py.ylim([-50, 50])

py.show()
```

Additionaly, we can check the performance of both: the DFT and the FFT, in comparison to the built-in numpy's FFT function:

```
x = np.random.random(1024)

%timeit DFT(x)
10 loops, best of 3: 86.9 ms per loop

%timeit FFT(x)
100 loops, best of 3: 4.28 ms per loop

%timeit np.fft.fft(x)
100000 loops, best of 3: 18.6 µs per loop
```

Like we can see our FFT algorithm implementation is much more efficient than DFT, but certainly slower from the numpy function. Why is that? Well, this is because FFTPACK algorithm behind numpy's FFT is a Fortran implementation which has received years of tweaks and optimizations. Furthermore, our solution involves both Python-stack recursion and the allocation of many temporary arrays, which adds significant computation time. A good strategy to speed up code when working with numpy is to vectorize repeated computations where possible. We can do this, and in the process remove our recursive functions calls, and make our FFT even more efficient.

```
def FFTs(x):
    x = np.asarray(x, dtype=float)
    N = x.shape[0]
```

```
    if np.log2(N) % 1 > 0:
        raise ValueError("size of x must be a power of 2")

    N_min = min(N, 32)

    n = np.arange(N_min)
    k = n[:, None]
    M = np.exp(-2j * np.pi * n * k / N_min)
    X = np.dot(M, x.reshape((N_min, -1)))

    while X.shape[0] < N:
        X_even = X[:, :X.shape[1] / 2]
        X_odd = X[:, X.shape[1] / 2:]
        factor = np.exp(-1j * np.pi * np.arange(X.shape[0])
                        / X.shape[0])[:, None]
        X = np.vstack([X_even + factor * X_odd,
                       X_even - factor * X_odd])

    return X.ravel()
```

Now let's look at the performance of the DFT, FFT, FFTs and numpy built-in FFT function.

```
x = np.random.random(1024)

%timeit DFT(x)
10 loops, best of 3: 87.2 ms per loop

%timeit FFT(x)
100 loops, best of 3: 4.34 ms per loop

%timeit FFTs(x)
1000 loops, best of 3: 463 µs per loop

%timeit np.fft.fft(x)
100000 loops, best of 3: 18.7 µs per loop
```

Ok, we can see that our FFT algorithm accelerated, but how does numpy's built in function attain such realy good timings? The main reason is that, in low-level language like Fortran it's easier to control and minimize memory use.

10.3 FFT in the frequency domain

You may have noticed that the FFT is normally expressed in the time domain. In order to express it in the frequency domain one have to perform some additional operations.

```
'''
Express FFT in the frequency domain (Fig. 12).
'''
```

```
import numpy as np
import pylab as py
from pylab import xlabel, ylabel
from scipy.fftpack import fftfreq, rfft

# Define signal.
Fs = 128.0 # Sampling rate.
Ts = 1.0 / Fs # Sampling interval.
Time = np.arange(0, 10, Ts) # Time vector.
signal = np.cos(4*np.pi*Time) + np.cos(6*np.pi*Time) + np.cos(8*np.pi*Time)

# Plot the signal.
py.subplot(2, 1, 1)
py.plot(Time, signal)
xlabel('Time')
ylabel('Amplitude')
py.ylim([-4, 4])

# Express FFT in the frequency domain.
def spectrum(signal, Time):

    frq = fftfreq(signal.size, d = Time[1] - Time[0] )
    Y = rfft(signal)

    return frq, Y

frq, spectrum = spectrum(signal, Time)

# Plot frequencies of the signal.
py.subplot(2, 1, 2)
py.plot(frq, spectrum)
py.xlim([0, np.max(frq)])
xlabel('Freq (Hz)')
ylabel('|Y(freq)|')

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

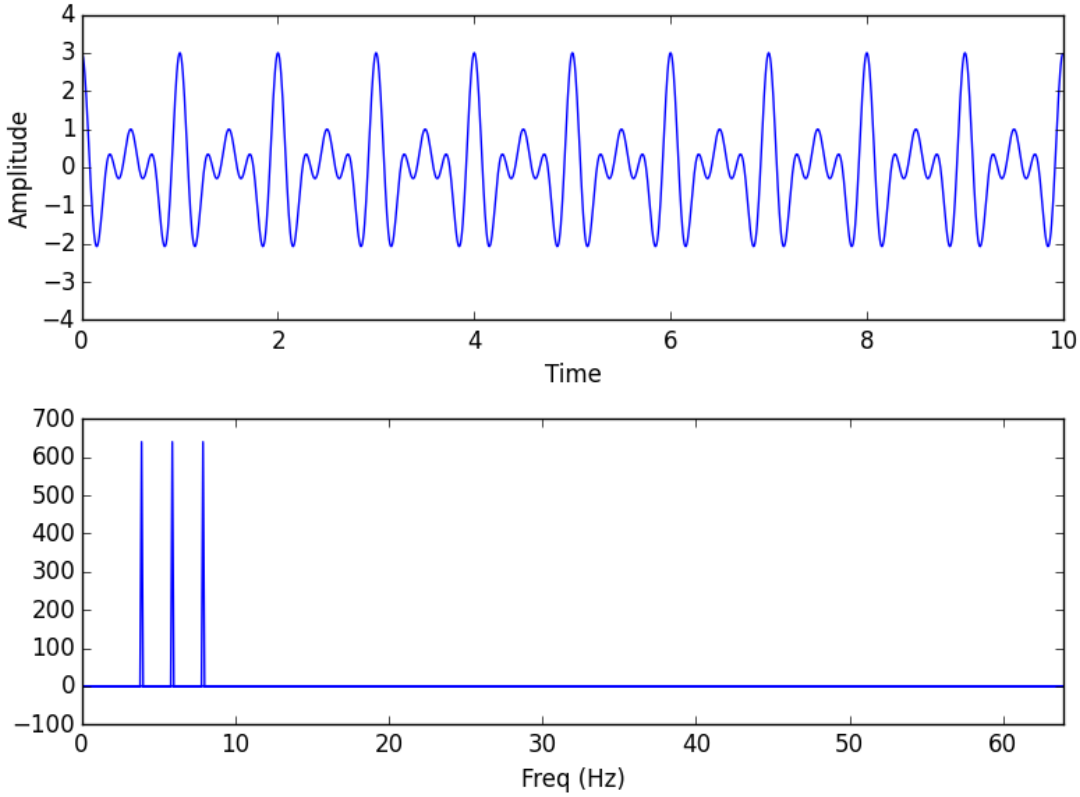

Figure 12: Some example signal and its FFT expressed in the frequency domain.

10.4 Quick example of the FFT use in bandpassing the signal

Before applying the appropriate smoothening filter as I shown in the section 8.2, one should consider to bandpass gathered signal in a specific range before performing any further actions. This will make the analyzed data much more fitted to the study assumptions, and also, much more easier to interpret and further analysis. One of the possible way to bandpass given signal is doing this with the use of FFT. In few words, it all comes down to the Fourier transformation of the signal, cutting unwanted frequencies and finally inverting the transformation function back (Fig. 13).

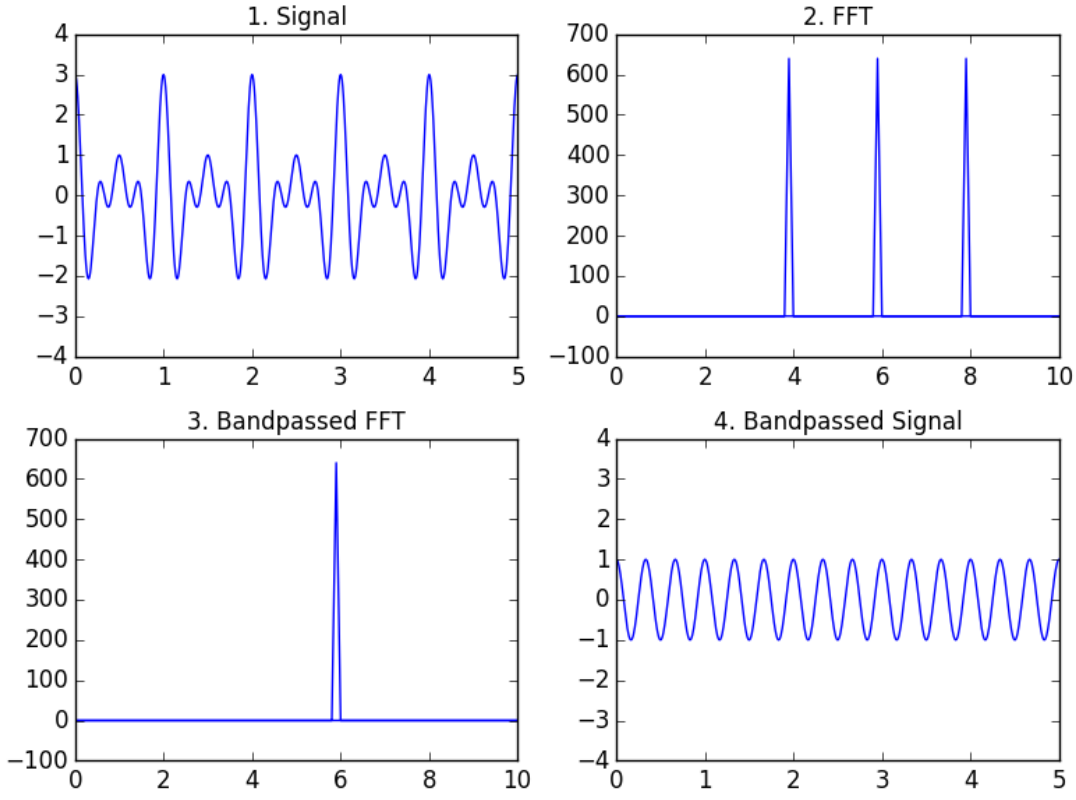

Figure 13: Bandpassing the signal for the range of > 5 and < 7.

```
'''
Simple bandpass based on the FFT frequency cut.
'''

import numpy as np
import pylab as py
from pylab import xlabel
from scipy.fftpack import fftfreq, rfft, irfft

# Define signal.
Fs = 128.0 # Sampling rate.
Ts = 1.0 / Fs # Sampling interval.
Time = np.arange(0, 10, Ts) # Time vector.
signal = np.cos(4*np.pi*Time) + np.cos(6*np.pi*Time) + np.cos(8*np.pi*Time)

# Plot the signal.
ax1 = py.subplot(2, 2, 1)
ax1.xaxis.set_label_position('top')
py.plot(Time, signal)
py.xlim([0, 5])
py.ylim([-4, 4])
xlabel("1. Signal")

# Express FFT in the frequency domain.
def spectrum(signal, Time):

    frq = fftfreq(signal.size, d = Time[1] - Time[0] )
```

```
    Y = rfft(signal)

    return frq, Y

frq, spectrum = spectrum(signal, Time)

# Plot frequencies of the signal.
ax2 = py.subplot(2, 2, 2)
ax2.xaxis.set_label_position('top')
py.plot(frq, spectrum)
py.xlim([0, 10])
xlabel('2. FFT')

# Bandpass the frequencies.
def bandpass(frq, spectrum, minFreq, maxFreq):
    spectrum[(frq < minFreq) | (frq > maxFreq)] = 0
    return spectrum

# Plot bandpassed frequencies of the signal.
ax3 = py.subplot(2, 2, 3)
ax3.xaxis.set_label_position('top')
py.plot(frq, bandpass(frq, spectrum, 5, 7))
py.xlim([0, 10])
xlabel("3. Bandpassed FFT")

# Inverse bandpassed FFT.
bandpassedSignal = irfft(bandpass(frq, spectrum, 4, 6))

# Plot bandpassed signal.
ax4 = py.subplot(2, 2, 4)
ax4.xaxis.set_label_position('top')
py.plot(Time, bandpassedSignal)
py.xlim([0, 5])
py.ylim([-4, 4])
xlabel("4. Bandpassed Signal")

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

When we will use the bandpassing method described above to our signal from Fp1, Fp2 and F7 channels, calculate for them the cross-correlation functions, and finally apply the Savitzky-Golay filter, than we will get the results as shown on the Fig. 14. Like you can see, there is now a big difference in the clarity of the results between Fig. 4-5 and Fig. 14.

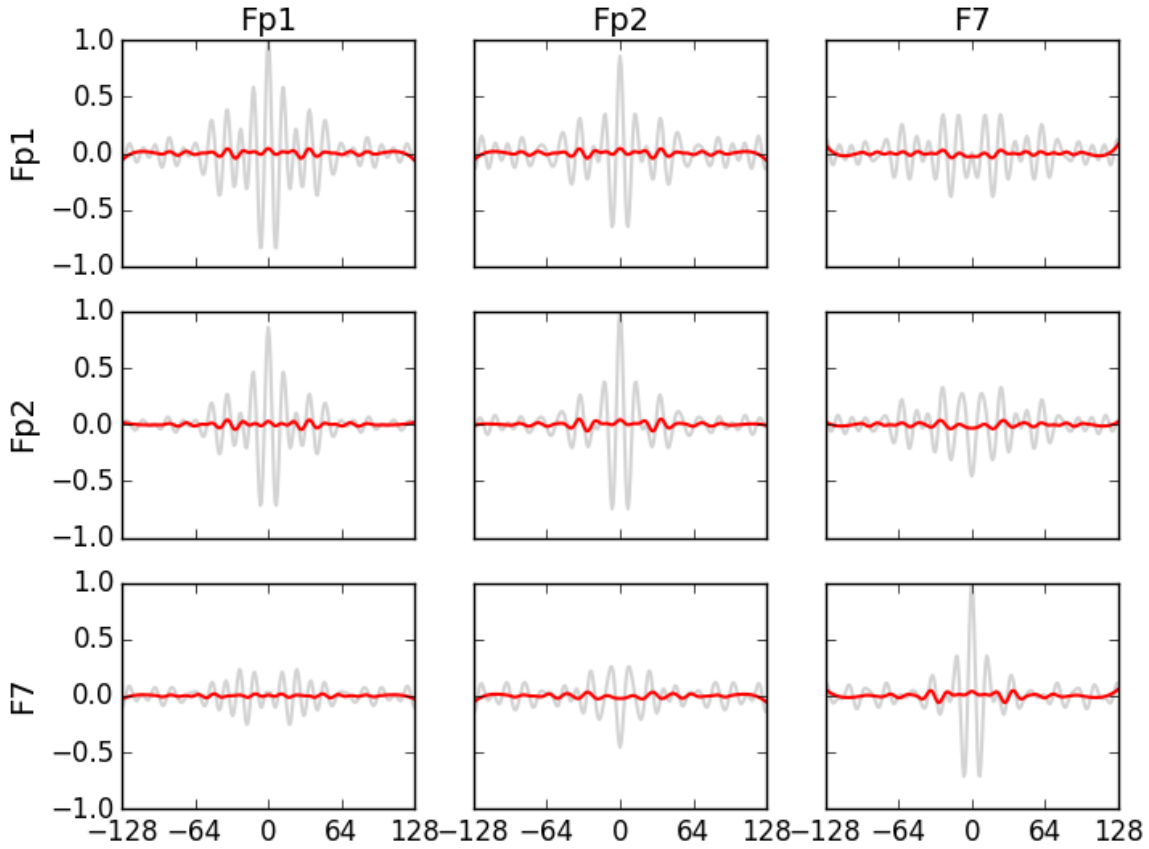

Figure 14: Correlograms for the bandpassed signals in the range of 7-13 Hz.

11 Power spectral density

Power spectral density can be described as the Fourier transform of the autocorrelation function of a given signal. Thus, in the context of the DFT, we can express the power spectral density as:

$$S(f) = \sum_{n=0}^{N-1} \quad \rho_n e^{-i2\pi f n/N},$$

where$\rho_n$is the n'th element of the autocorrelation function, i.e$\rho(k) = \frac{\gamma(k)}{\gamma(0)}$, which was described in sections 4-7.

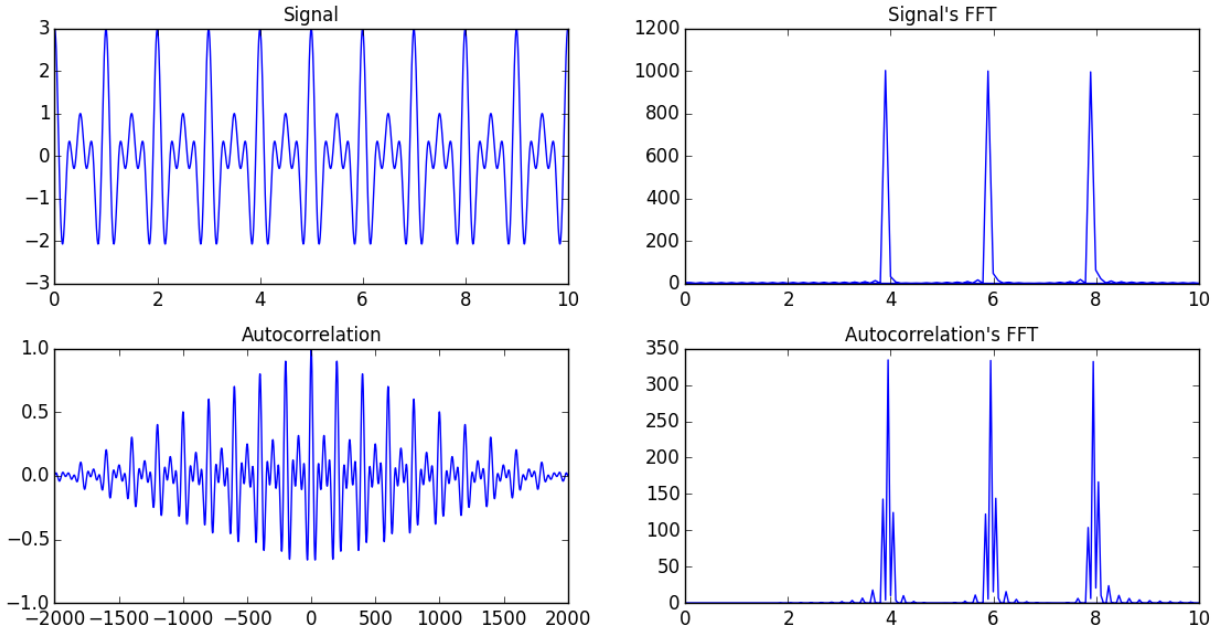

Figure 15: Calculate the power spectral density of the example signal.

```
import numpy as np
import pylab as py
from scipy.fftpack import fftfreq, rfft

# Define the signal.
x = np.linspace(0, 10, 2000)
signal = np.cos(4*np.pi*x) + np.cos(6*np.pi*x) + np.cos(8*np.pi*x)

# Plot the signal.
ax1 = py.subplot(2, 2, 1)
ax1.xaxis.set_label_position('top')
py.plot(x, signal)
py.xlabel('Signal')

# Plot signals FFT in the frequency domain.
def spectrum(signal, Time, absoluteFFT = True):

    frq = fftfreq(signal.size, d = Time[1] - Time[0] )
    Y = rfft(signal)

    if absoluteFFT == True:
        return frq, np.abs(Y)
    else:
        return frq, Y

frq, spec = spectrum(signal, x)

ax2 = py.subplot(2, 2, 2)
ax2.xaxis.set_label_position('top')
py.plot(frq, spec)
py.xlim([0, 10])
py.xlabel("Signal's FFT")
```

```
# Cross-correlation function of the signals.
def normalization(a, v):
    a = (a - np.mean(a)) / (np.std(a) * len(a))
    v = (v - np.mean(v)) /  np.std(v)

    return a, v

a, v = normalization(signal, signal)
RHO_signal = np.correlate(a, v, mode = 'full')

ax3 = py.subplot(2, 2, 3)
ax3.xaxis.set_label_position('top')
py.plot(np.linspace(-np.size(RHO_signal) / 2, np.size(RHO_signal) / 2,
np.size(RHO_signal)), RHO_signal)
py.xlabel("Autocorrelation")
py.ylim([-1, 1])

# Plot cross-correlation's FFT in the frequency domain.
# (i.e cross-power spectral density)
frq_signal, spectrum_signal = spectrum(RHO_signal, x)

ax4 = py.subplot(2, 2, 4)
ax4.xaxis.set_label_position('top')
py.plot(frq_signal, spectrum_signal)
py.xlim([0, 10])
py.xlabel("Autocorrelation's FFT")

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

12 Cross-power spectral density

In the case of the cross-power spectral density function we will use the formula from the section 11, i.e:$S_{xy}(f) = \sum_{n=0}^{N-1} \quad \rho_n e^{-i2\pi fn/N}$, but now$\rho(k) = \frac{\gamma_{xy}}{\sigma_x \sigma_y}$, which is due to the fact, that we want to verify the cross-power spectral density function based on the cross-correlation function.

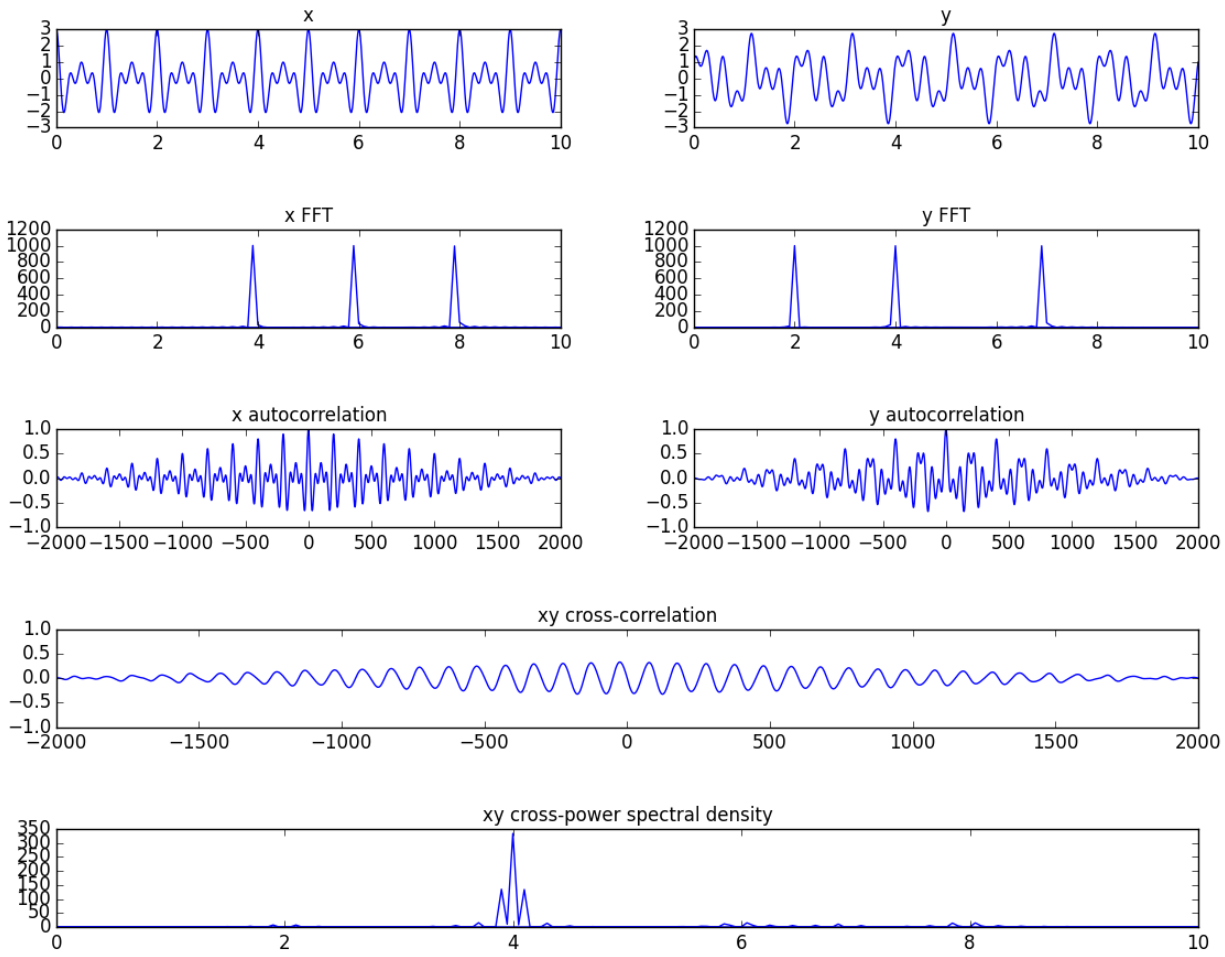

Figure 16: Calculate the cross-power spectral density for two example signals.

```
import numpy as np
import pylab as py
from scipy.fftpack import fftfreq, rfft

# Define signal x.
time = np.linspace(0, 10, 2000)
x = np.cos(4*np.pi*time) + np.cos(6*np.pi*time) + np.cos(8*np.pi*time)

# Define signal y.
y = np.sin(2*np.pi*time) + np.sin(4*np.pi*time) + np.cos(7*np.pi*time)

# Plot signal x.
ax1 = py.subplot2grid((5, 2), (0, 0), colspan=1)
ax1.xaxis.set_label_position('top')
py.plot(time, x)
py.xlabel('x')

# Plot signal y.
ax2 = py.subplot2grid((5, 2), (0, 1), colspan=1)
ax2.xaxis.set_label_position('top')
py.plot(time, y)
```

```
py.xlabel('y')

# Plot signals FFT in the frequency domain.
def spectrum(signal, Time, absoluteFFT = True):

    frq = fftfreq(signal.size, d = Time[1] - Time[0] )
    Y = rfft(signal)

    if absoluteFFT == True:
        return frq, np.abs(Y)
    else:
        return frq, Y

x_frq, x_spec = spectrum(x, time)

ax3 = py.subplot2grid((5, 2), (1, 0), colspan=1)
ax3.xaxis.set_label_position('top')
py.plot(x_frq, x_spec)
py.xlim([0, 10])
py.xlabel("x FFT")

y_frq, y_spec = spectrum(y, time)

ax4 = py.subplot2grid((5, 2), (1, 1), colspan=1)
ax4.xaxis.set_label_position('top')
py.plot(y_frq, y_spec)
py.xlim([0, 10])
py.xlabel("y FFT")

# Cross-correlation function of the signals.
def normalization(a, v):
    a = (a - np.mean(a)) / (np.std(a) * len(a))
    v = (v - np.mean(v)) /  np.std(v)

    return a, v

a, v = normalization(x, y)
RHO_xy = np.correlate(a, v, mode = 'full')

a, v = normalization(x, x)
RHO_x = np.correlate(a, v, mode = 'full')

a, v = normalization(y, y)
RHO_y = np.correlate(a, v, mode = 'full')

ax5 = py.subplot2grid((5, 2), (2, 0), colspan=1)
ax5.xaxis.set_label_position('top')
py.plot(np.linspace(-np.size(RHO_x) / 2, np.size(RHO_x) / 2, np.size(RHO_x)),
RHO_x)
py.xlabel("x autocorrelation")
py.ylim([-1, 1])

ax6 = py.subplot2grid((5, 2), (2, 1), colspan=1)
ax6.xaxis.set_label_position('top')
py.plot(np.linspace(-np.size(RHO_y) / 2, np.size(RHO_y) / 2, np.size(RHO_y)),
RHO_y)
```

```
py.xlabel("y autocorrelation")
py.ylim([-1, 1])

ax7 = py.subplot2grid((5, 2), (3, 0), colspan=2)
ax7.xaxis.set_label_position('top')
py.plot(np.linspace(-np.size(RHO_xy) / 2, np.size(RHO_xy) / 2,
np.size(RHO_xy)), RHO_xy)
py.xlabel("xy cross-correlation")
py.ylim([-1, 1])

# Plot cross-correlation's FFT in the frequency domain.
# (i.e cross-power spectral density)
frq, spec = spectrum(RHO_xy, time)

ax8 = py.subplot2grid((5, 2), (4, 0), colspan=2)
ax8.xaxis.set_label_position('top')
py.plot(frq, spec)
py.xlim([0, 10])
py.xlabel("xy cross-power spectral density")

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

13 Coherence

Coherence is nothing but normalized version of the cross-power spectral density function. One can express it as:$C_{xy}(f) = \frac{|S_{xy}(f)|^2}{S_{xx}(f)S_{yy}(f)}$.

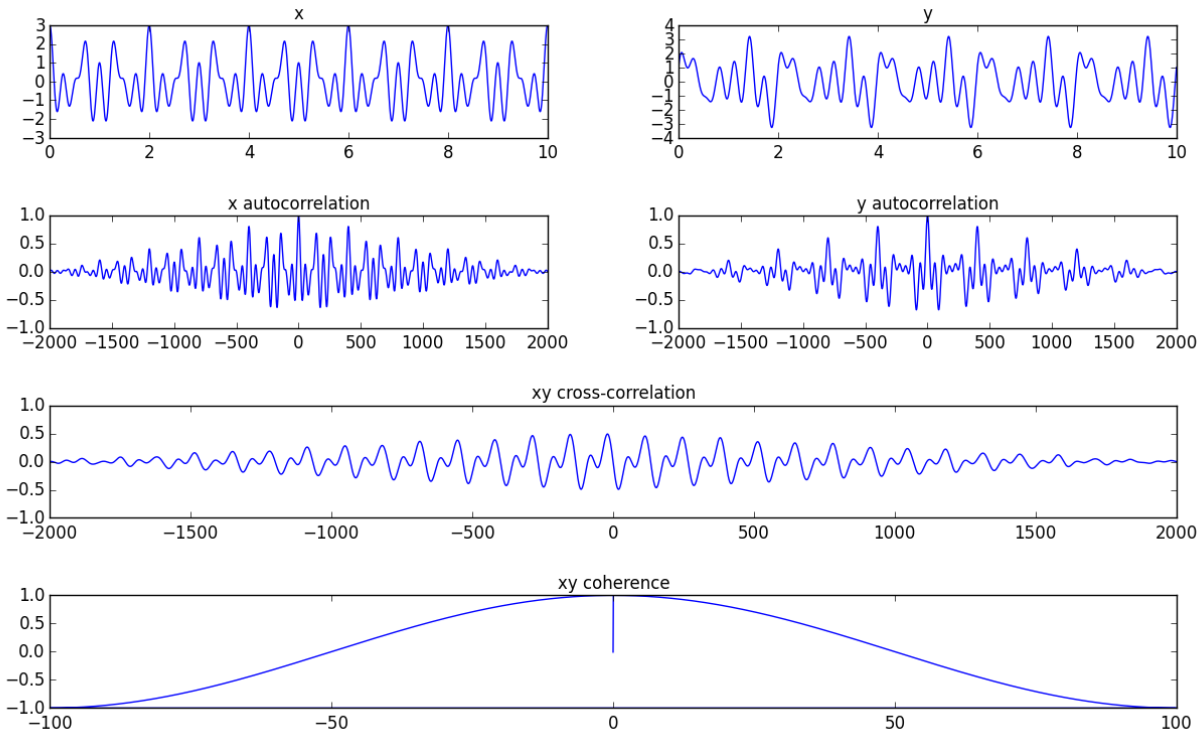

Figure 17: Calculate the coherence for two example signals.

```
import numpy as np
import pylab as py
from scipy.fftpack import fftfreq

# Define signal x.
time = np.linspace(0, 10, 2000)
x = np.cos(3*np.pi*time) + np.cos(6*np.pi*time) + np.cos(8*np.pi*time)

# Define signal y.
y = np.sin(2*np.pi*time) + np.sin(3*np.pi*time) + np.cos(7*np.pi*time) +
np.sin(6*np.pi*time)

# Plot the signal x.
ax1 = py.subplot2grid((4, 2), (0, 0), colspan=1)
ax1.xaxis.set_label_position('top')
py.plot(time, x)
py.xlabel('x')

# Plot the signal y.
ax2 = py.subplot2grid((4, 2), (0, 1), colspan=1)
ax2.xaxis.set_label_position('top')
py.plot(time, y)
py.xlabel('y')

# Cross-correlation function of the signals.
# Normalize x and y signals before performing the correlation.
def normalization(a, v):
    a = (a - np.mean(a)) / (np.std(a) * len(a))
    v = (v - np.mean(v)) /  np.std(v)
```

```
    return a, v

a, v = normalization(x, y)
RHO_xy = np.correlate(a, v, mode = 'full')

a, v = normalization(x, x)
RHO_x = np.correlate(a, v, mode = 'full')

a, v = normalization(y, y)
RHO_y = np.correlate(a, v, mode = 'full')

ax3 = py.subplot2grid((4, 2), (1, 0), colspan=1)
ax3.xaxis.set_label_position('top')
py.plot(np.linspace(-np.size(RHO_x) / 2, np.size(RHO_x) / 2, np.size(RHO_x)),
RHO_x)
py.xlabel("x autocorrelation")
py.ylim([-1, 1])

ax4 = py.subplot2grid((4, 2), (1, 1), colspan=1)
ax4.xaxis.set_label_position('top')
py.plot(np.linspace(-np.size(RHO_y) / 2, np.size(RHO_y) / 2, np.size(RHO_y)),
RHO_y)
py.xlabel("y autocorrelation")
py.ylim([-1, 1])

ax5 = py.subplot2grid((4, 2), (2, 0), colspan=2)
ax5.xaxis.set_label_position('top')
py.plot(np.linspace(-np.size(RHO_xy) / 2, np.size(RHO_xy) / 2,
np.size(RHO_xy)), RHO_xy)
py.xlabel("xy cross-correlation")
py.ylim([-1, 1])

# Calculate coherence.
def coherence(Pxy, Px, Py, Time):

    frq = fftfreq(Pxy.size, d = Time[1] - Time[0])

    Sxy = np.fft.fft(Pxy)
    Sx = np.fft.fft(Px)
    Sy = np.fft.fft(Py)

    Cxy = (np.abs(Sxy)**2) / (Sx*Sy)

    return frq, Cxy

coh_frq, Cxy = coherence(RHO_xy, RHO_x, RHO_y, time)

ax6 = py.subplot2grid((4, 2), (3, 0), colspan=2)
ax6.xaxis.set_label_position('top')
py.plot(coh_frq, Cxy)
py.xlabel("xy coherence")

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

14 Quantization noise

When you perform the recording of the analog signal with the use of analog to digital (A/D) converter hardware, you will get a discrete signal. Due to the finite perfection of the A/D converters it is impossible for discrete signals to be perfectly reproductions of the analog signals. The source of this imperfection lies in the way how A/D converters work.

A/D converter simplify the analog signal into a quantized (discrete) form, which is a process of replacing the continuous values of the analog signal by values changing by leaps in respect to the appropriate scale mapping (accuracy). A/D conversion in three steps can be described as: sampling, quantization, and coding. The number of discrete values that A/D converter can generate is determined by it's resolution, which is usually expressed in bits. For example, the A/D converter that can convert the signal sample to one of the 64 numerical values has a resolution equal to 6 bits, since$2^6 = 64$. The resolution can be also expressed in Volts. Voltage resolution of the A/D is equal to the full scale of measurement divided by the number of quantization levels. Example: 1) scale is defined by the range of 0 to 10 Volts; 2) A/D converter resolution is 6 bits; 3) so the voltage resolution is$(10 - 0)/64 = 0{,}15625$, which in Volts is$15{,}62mV$, and can be expressed by the equation$res_V = \frac{R}{2^N}$, where$R$is range of the voltage scale,$N$is the number of A/D converter bits, and$2^N$is the number of quantization levels.

So like we can see, the discrete signal will always be different from the analog signal. This difference is also known as the quantization noise. But we can try to minimalize this noise by the use of the A/D converters with larger resolution, i.e number of bits.

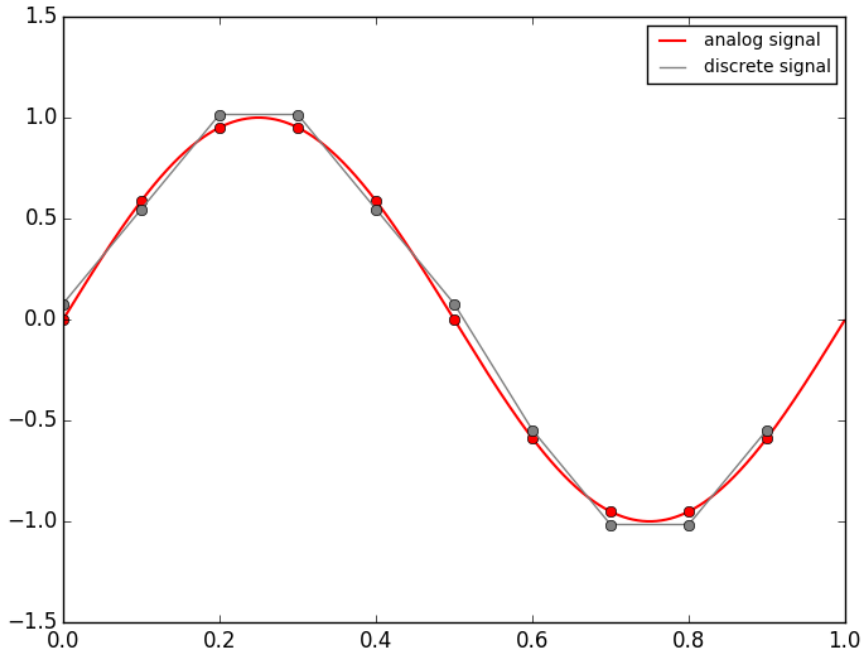

Figure 18: Illustration of the analog and quantized signals for A/D converter with 6 bit resolution and for 0-10 V range.

```
import numpy as np
import pylab as py

def sin(frequency = 1, time = 1, sampling = 128, phi = 0):
    '''
    Sinus function for a given frequency in Hz, length in time,
    sampling frequency, and phase.
    '''
    dt = 1.0 / sampling
    t = np.arange(0, time, dt)
    s = np.sin((2 * np.pi * frequency * t) + phi)
    return s, t

s_analog_1, t_analog_1 = sin(frequency = 1, sampling = 1000)
py.plot(t_analog_1, s_analog_1, linewidth = '1.5', color = 'red', label =
'analog signal')

s_analog_2, t_analog_2 = sin(frequency = 1, sampling = 10)
py.plot(t_analog_2, s_analog_2, 'o', color = 'red')

# Calculate the quantization noise, ie "R/2^N"
s_discrete, t_discrete = sin(frequency = 1, sampling = 10)
N = 6
R = 10
dy = R/2**N
s_quantized = np.floor(s_discrete / dy) * dy + 0.5 *dy

py.plot(t_discrete, s_quantized, color = 'gray', label = 'discrete signal')
py.plot(t_discrete, s_quantized, 'o', color = 'gray')
```

```
py.legend(loc = 'upper right', fontsize = '10')

py.show()
```

15 Generation of random signals

In a study of different methods of signal analysis, we may need signals with known properties. In particular, it is good to be able to give signals occurring in digital form, and artificial test signals, certain physical properties such as: sampling frequency, time duration, amplitude, and phase. Below you can find implementations of some famous functions.

15.1 Sinus

```
import numpy as np
import pylab as py

def sin(frequency = 1, time = 1, sampling = 128, phi = 0):
    '''
    Sinus function for a given frequency in Hz, length in time,
    sampling frequency, and phase.
    '''
    dt = 1.0 / sampling
    t = np.arange(0, time, dt)
    s = np.sin((2 * np.pi * frequency * t) + phi)
    return s, t

s, t = sin(frequency = 1, sampling = 1000)
py.plot(t, s)
py.show()
```

15.2 Delta

```
import numpy as np

import pylab as py

def delta(impulseTime = 0.5, time = 1, sampling = 128):
    '''
    Delta function for a given impulse time,
    length in time, and sampling frequency.
    '''
    dt = 1.0 / sampling
    t = np.arange(0, time, dt)
    d = np.zeros(len(t))
    d[np.ceil(impulseTime * sampling)] = 1
    return d, t
```

```
d, t = delta()
py.plot(t, d)
```