SyncPy Documentation

Release 1

Giovanna Varni, Marie Avril

September 02, 2015

CHAPTER 1

Introduction

SyncPy is a novel open-source analytic library for investigating synchrony in a fast and exhaustive way. It stems from work and discussions among researchers on synchrony in different domains as engineering, computer science and psychology. SyncPy is mainly aimed at helping researchers to explore, try and compare in an easy way and with a single tool synchrony methods starting from signals. Signals are synthetic or experimental time series organized as *pandas DataFrames*.

The library has been conceived to investigate synchrony in human-human/human machine interaction, however, although it focuses on *interpersonal synchrony*, all the methods are exploitable in other contexts.

SyncPy functionalities include three main components:

1. utils package
2. graphical interface
3. synchrony methods package

The utils package contains functionals of general utility directly used by the synchrony methods or to preprocess the input signals. All the functionals are described in the Section 3.3 of this document.

The graphical interface is a pyQT application conceived to assist users to choose and try several methods. More sepcifically, it allows users to:

1. load time series from files
2. visualize/modify these time series through the utils
3. choose a consistent method according to the data set
4. compute the selected method and
5. visualize and/or save the result in a file (.csv format).

The synchrony methods package will be described in details in the Section 3.2 of this document.

SyncPy library is currently under development in the framework of the SMART Labex Project (smart-labex)

CHAPTER 2

Getting the SyncPy library

2.1 Dependencies

For a complete use of the **SyncPy** library, please install these libraries first :

- Matplotlib
- NetworkX
- Numpy and Scipy
- Pandas
- Statsmodels and Patsy

2.2 Sources

2.3 Getting Started

Trying to use a specific method of the Toolbox ? Check out the *SyncPy library Examples* and the *Documentation* pages to help you.

CHAPTER 3

Documentation

3.1 Library tree organization

SyncPy structured over both a horizontal and a vertical sub-package tree: this allows to distinguish synchrony methods depending on input and method types, respectively. Four horizontal levels are identifiable:

1. Number of participants: two (`DataFrom2Persons` package) ore more than two involved in the interaction (`DataFromManyPersons` package)
2. Number of variables in input signals: one (`Monovariate` package) or more than one (`Multivariate` package) that are available for each participant
3. Type of data in input signals continuous (`Continuous` pacakge) or categorical (`Categorical` package)
4. Type of analysis will be carried out on signals (`Linear`, `NonLinear` or `MachineLearning` pacakges)

3.2 Toolbox complete tree

3.2.1 DataFrom2Persons package

This package allows to compute synchronisation between continuous/categorical monovariate and multivariate signals gathered from two persons.

Monovariate package

This package allows to compute synchronisation between continuous/categorical monovariate signals gathered from two persons. Each monovariate signal should be organized as a monovariate pandas DataFrame.

Categorical package

This package allows to compute synchronisation between categorical monovariate signals gathered from two persons.

Linear package

BooleanTurnsActivity module *Module author: Marie Avril*

class `DataFrom2Persons.Monovariate.Categorical.Linear.BooleanTurnsActivity.`**`BooleanTurnsActivity`**(*max_latency*, *min_pause_duration*, *ele_per_sec=1*, *duration=-1*)

It computes data turns statistics between two boolean monovariate signals (in pandas DataFrame format) x and y : x signal activity duration, y signal activity duration, pause duration, overlap duration, x signal pause duration, y signal pause duration, pause duration between x and y activity, synchrony ratios between x and y (defined by max_latency).

Parameters

- **`max_latency`** (*float*) – the maximal delay (in second) between the two signals activity to define synchrony
- **`min_pause_duration`** (*float*) – minimal time (in second) for defining a pause
- **`ele_per_sec`** (*int*) – number of elements in one second. Default: 1
- **`duration`** (*int*) – total activity duration (in second). If -1, duration = len(x)*ele_per_sec. Default : -1

`compute`(*x*, *y*)

Compute data turns activities

Parameters

- **`x`** (*pd.DataFrame*) – first input signal
- **`y`** (*pd.DataFrame*) – second input signal

Returns pd.DataFrame – duration for each type of activity

Returns pd.DataFrame – ratios for each type of activity

`diff`(*vector*)

Compute a diff vector

Parameters **`np.array`** – input vector to diff

Retuns np.array – list of differences between each two consecutive values

MachineLearning package

Nonlinear package

EventSync module

Module author: Giovanna Varni

class `DataFrom2Persons.Monovariate.Categorical.Nonlinear.EventSync.`**`EventSync`**(*atype='tot'*, *tau=0*, *lag_tau=0*, *window=1*, *plot=False*)

It computes synchronicity (Q) and time delay patterns (q) between two monovariate signals x and y (in pandas DataFrame format) in which events can be identified.

Reference :

- R.Quian Quiroga; T.Kreuz and P.Grassberger; "A simple and fast method to measure synchronicity and time delay patterns." (Phys.Rev. E; 66; 041904 (2002))

Parameters

- **`atype`** (*str*) – it can assumes the following values: 'tot', 'tsl', 'asl'

  1. 'tot': synchronicity (Q) and time delay pattern(q) are computed over all the time signals
  2. 'tsl': time resolved variants of Q and q
  3. 'asl': averaged time resolved variants of Q and q over a window whose size is specified by the window parameter

  Default: 'tot
- **`tau`** (*int*) – it is a binary value [0,1] indicating which type of algorithm is used to estimates the delay.

  1. 0 : a prefixed tau with value specified by lag_tau will be used. It should be smaller than half the minimum interevent distance
  2. 1 : an atuomatically estimated local tau for each pair of events in the series will be used. The local tau for a generic pair of events i and j is computed as the half of the minimum value in the following set

  [tau_x(i+1)- tau_x(i); tau_x(i)-tau_x(i-1); tau_y(j+i)-tau_y(j), tau_y(j)-tau_y(j-1)] Default: 1
- **`lag_tau`** (*int*) – it is the (positive) number of samples will be used as delay when tau is set to 0
- **`window`** (*int*) – it is the size of the window (in samples) used to compute Q and q when type is 'asl'.
- **`plot`** (*bool*) – if True the plot of Q and q is returned when atype is set to 'tsl' or 'asl'. Default: False

`Jfunct`(*peak_x*, *peak_y*)

It computes the terms of the conditional probabilities c_tau

Parameters

- **peak_x** (*array*) – peaks in x
- **peak_y** (*array*) – peaks in y

Returns tuple – terms of the conditional probabilities c_tau

QQ_tsl (*jay_out_xy*, *jay_out_yx*, *lx*, *peak_x*, *peak_y*, *delta_sample*)

It computes synchronicity (Q) and time delay pattern (q) when atype is set to 'tsl' or to 'asl'

Parameters

- **jay_out_xy** (*array*) – Jfunct for x given y
- **jay_out_yx** (*array*) – Jfunct for y given x
- **peak_x** (*array*) – peaks locations in x
- **peak_y** (*array*) – peaks locations in y
- **delta_sample** (*array*) – size of window (useful only for computing Q and q in 'asl')

Returns tuple – Q and q over the whole length time series

Qq_tot (*jay_out_xy*, *jay_out_yx*, *l_peak_x*, *l_peak_y*)

It computes synchronicity (Q) and time delay pattern(q) over all the time series ('tot' type)

Parameters

- **jay_out_xy** (*array*) – Jfunct for x given y
- **jay_out_yx** (*array*) – Jfunct for y given x
- **l_peak_x** (*array*) – peaks locations in x
- **l_peak_y** (*array*) – peaks locations in y

Returns tuple – Q and q over the whole length time series

compute (*x*, *y*)

It computes the wanted version of Q and q

Parameters

- **x** (*pd.DataFrame*) – first input signal
- **y** (*pd.DataFrame*) – second input signal

Returns dict – Q, q

heaviside (*x*)
It computes the heaviside function of x

Parameters **x** (*int*) – number on which computing heaviside

Returns int – heaviside function value

optimal_tau (*t_peak_x*, *t_peak_y*)
It estimates the value of tau as the half the minimum of all the intervent distances.

Parameters

- **t_peak_x** (*array*) – sequences of events in x
- **t_peak_y** (*array*) – sequences of events in y

Returns array – optimal value for tau

plot ()
It plots the resulting Q and q when atype is set to 'tsl' or 'asl'

Returns plt.figure – figure plot

Continuous package

This package allows to compute synchronisation between continuous monovariate signals gathered from two persons.

Linear package

Coherence module *Module author: Giovanna Varni*

class DataFrom2Persons.Monovariate.Continuous.Linear.Coherence.**Coherence** (*fs=1.0*, *NFFT=256*, *detrend=0*, *noverlap=0*, *plot=False*)
It computes the linear correlation between two monovariate signals x and y (in pandas DataFrame format) as a function of the frequency. It is the cross-spectral density function normalized by the autospectral density function of x and y.

Reference :

•Inspired by a John Hunter's Python code

Parameters

- **fs** (*float*) – sampling frequency (in Hz) of the input DataFrame . Default: 1.0

- **NFFT** (*int*) – length (in samples) of each epoch. Default: 256
- **detrend** (*int*) –

  it specifies which kind of detrending should be computed on the inout. It ranges in [0;2]:

  1. 0 no detrending;
  2. 1 constant detrending;
  3. 2 linear detrending.

  Default: 0
- **noverlap** (*int*) – number of sampels to overlap between epochs. Default: 0
- **plot** (*bool*) – if True the plot of coherence function is returned. Default: False

compute (*x*, *y*)
It computes the coherence function between x and y.

Parameters

- **x** (*pd.DataFrame*) – first input signal
- **y** (*pd.DataFrame*) – second input signal

Returns dict –coherence and frequencies over which the coherence is computed

plot ()
It plots the coherence function

Returns plt.figure – figure plot

Correlation module *Module author: Giovanna Varni*

class DataFrom2Persons.Monovariate.Continuous.Linear.Correlation.**Correlation**(*tau_max*, *plot=False*, *standardization=False*, *corr_tau_max=False*, *corr_coeff=False*, *scale=False*)
It computes the linear correlation between two monovariate signals x and y (in pandas DataFrame format) as a function of their delay tau. It computes autocorrelation when y coincides with x.

Parameters

- **tau_max** (*int*) – the maximum lag (in samples) at which correlation should be computed. It is in the range [0; (length(x)+length(y)-1)/2]
- **plot** (*bool*) – if True the plot of correlation function is returned. Default: False

- **standardization** (*bool*) – if True the inputs are standardize to mean 0 and variance 1. Default: False
- **corr_tau_max** (*bool*) – if True the maximum of correlation and its lag are returned. Default: False
- **corr_coeff** (*bool*) – if True the correlation coefficient (Pearson's version) is computed. It is enabled only if the parameter standardize is True. Default: False
- **scale** (*bool*) – if True the correlation function is scaled in the range [-1;1]

compute (*x*, *y*)
It computes the correlation function between x and y

Parameters

- **x** (*pd.DataFrame*) – first input signal
- **y** (*pd.DataFrame*) – second input signal

Returns dict – correlation function/maximum of correlation and its lag/Pearson's coefficient

compute_coeff (*corr_f*, *lmin*, *ly*)
It computes the Pearson's correlation coefficient.

Parameters

- **corr_f** (*numpy.array*) – correlation function
- **lmin** – the length of the shortest input
- **ly** (*int*) – length of the second input

Returns numpy.array – time/Pearson's correlation coefficient

compute_tau_range (*lx*, *ly*)
Computes the range of tau values the correlation function is returned for.

Parameters

- **self.lx** (*int*) – length of the first input signal
- **self.ly** (*int*) – length of the second input signal

Returns numpy.array – the range of tau values the correlation function is returned for

plot ()
It plots the correlation function in the range specified.

Returns plt.figure – figure plot

WindowCrossCorrelation module *Module author: Marie Avril*

class `DataFrom2Persons.Monovariate.Continuous.Linear.WindowCrossCorrelation.`**`WindowCrossCorrelation`**(*tau_max=0*, *window=0*, *win_inc=1*, *tau_inc=1*, *plot=False*, *ele_per_sec=1*)

It computes the window cross correlation between signals (in pandas DataFrame format) x and y

Parameters

- **`tau_max`** (*int*) – the maximum lag (in samples) at which correlation should be computed. It is in the range [0; (length(x)+length(y)-1)/2]
- **`window`** (*int*) – length (in samples) of the windowed signals
- **`win_inc`** (*int*) – amount of time (in samples) elapsed between two windows
- **`tau_inc`** (*int*) – amount of time (in samples) elapsed between two cross-correlation
- **`plot`** (*bool*) – if True the plot of correlation function is returned. Default: False
- **`ele_per_sec`** (*int*) – number of element in one second

`compute`(*x*, *y*)

it computes correlation function

Parameters

- **`x`** (*pd.DataFrame*) – first input signal
- **`y`** (*pd.DataFrame*) – second input signal

Returns pd.DataFrame – windowed cross correlation DataFrame with (2 * tau_max + 1)/tau_inc rows and (length(x) - window - win_inc)/win_inc columns

`plot`()

It plots the window cross correlation matrix

Returns plt.figure – figure plot

PeakPicking module *Module author: Marie Avril*

class `DataFrom2Persons.Monovariate.Continuous.Linear.PeakPicking.`**`PeakPicking`**(*corr_matrix*, *tau_max*, *tau_inc=0*, *threshold=0*, *lookahead=300*, *delta=0*, *ele_per_sec=1*, *plot=False*, *plot_on_mat=False*, *sorted_peak=False*)

It computes peak picking algorithm with window cross correlation results (see WindowCrossCorrelation)

Parameters

- **`corr_matrix`** (*pd.DataFrame*) – matrix of cross correlation computed by WindowCrossCorrelation module (windowed cross correlation DataFrame with (2 * tau_max + 1)/tau_inc rows and (length(x) - window - win_inc)/ win_inc colums)
- **`tau_max`** (*int*) – the maximum lag (in samples) at which correlation should be computed. It is in the range [0, (length(x)+length(y)-1)/2]
- **`tau_inc`** (*int*) – amount of time (in samples) elapsed between two cross-correlation
- **`threshold`** (*float*) – minimal correlation magnitude acceptable for a peak. It is in the range [-1;1]
- **`lookahead`** (*int*) – distance to look ahead from a peak candidate to determine if it is the actual peak. Default: 200 (sample / period) / f where 4 >= f >= 1.25 might be a good value
- **`delta`** (*int*) – it specifies a minimum difference between a peak and the following points, before a peak may be considered a peak. Useful to hinder the function from picking up false peaks towards to end of the signal. To work well delta should be set to delta >= RMSnoise * 5. Default: 0
- **`ele_per_sec`** (*int*) – number of elements in one second
- **`plot`** (*bool*) – if True the plot of peakpicking function is returned. Default: False
- **`plot_on_mat`** (*bool*) – if True the plot of peakpicking + correlation matrix function is returned. Default: False
- **`sorted_peak`** – if True the peaks found will be organized by type of Lag and Magnitude (positive or negative). Default: False

`compute`()

It computes peak picking from cross correlation matrix

Returns pd.DataFrame – if sorted_peak is False, peaks found organized per Maximin, Minimum and Extremum

Returns pd.DataFrame – if sorted_peak is True, peaks found organized by type of Lag and Magnitude (positive or negative)

`plot`()

It plots the peakpicking result

Returns plt.figure – figure plot

`sort_peakPickingResult`()

It organizes peakPicking result in order to compute statistics

Returns pd.DataFrame – peaks found organized by type of Lag and Magnitude (positive or negative)

GrangerCausality module *Module author: Adem Usta*

class `DataFrom2Persons.Monovariate.Continuous.Linear.GrangerCausality.`**`GrangerCausality`**(*max_lag=1*, *criterion='bic'*, *plot=False*)

It computes a Granger causality test between two monovariate signals x and y (in pandas DataFrame format). It computes unidirectionnal causality test with a bivariate autoregressive model and test if the unrestricted model is statistically significant compared to the restricted one. An F-test is computed and then the interpretation is up to the user.

Reference :

•Anil K. Seth. A MATLAB toolbox for Granger causal connectivity analysis. Journal of Neuroscience Methods, 186(2) :262-273, February 2010.

param max_lag The number of maximum lag (in samples) with which the autoregressive model will be computed. It ranges in [1;length(x)]. Default : 1.

type max_lag int

Parameters

- **`criterion`** (*str*) – A string that contains the name of the selected criterion to estimate optimal number of lags value. Two choices are possible :

  1.'bic' (Bayesian Information Criterion); 2.'aic' (Akaike Information Criterion)

  Default : 'bic'

- **`plot`** (*bool*) – if True the plot of correlation function is returned. Default: False

`compute`(*x*, *y*)

It computes restricted AR and unrestricted AR models, and evaluates whether the x signal could be forecasted by the y signal. F-value and p-value are computed, the interpretation of the results is up to the user.

Parameters

- **`x`** (*pd.DataFrame*) – first input signal - 'signal_to_predict'
- **`y`** (*pd.DataFrame*) – second input signal - 'helping_signal'

Returns dict – F-values and P-values.

plot()
It plots the results of AR process for both restricted and unrestricted models :

Returns plt.figure – A figure that contains all the subplots

SpectralGrangerCausality module *Module author: Adem Usta*

class DataFrom2Persons.Monovariate.Continuous.Linear.SpectralGrangerCausality.**SpectralGrangerCausality**(*max_lag=1*, *criterion='bic'*, *plot=False*)

It computes a Granger causality test between two monovariate signals x and y (in pandas DataFrame format), in the spectral domain.

Reference :

•Adam B. Barrett, Michael Murphy, Marie-Aurelie Bruno, Quentin Noirhomme, Melanie Boly, Steven Laureys, and Anil K. Seth. Granger Causality Analysis of Steady-State Electroencephalographic Signals during Propofol-Induced Anaesthesia. PLoS ONE, 7(1) :e29072, January 2012.

Parameters

- **max_lag** (*int*) – The number of maximum lag (in samples) with which the autoregressive model will be computed. It ranges in [1;length(x)]. Default : 1.
- **criterion** (*str*) – A string that contains the name of the selected criterion to estimate optimal number of lags value. Two choices are possible :

  1.'bic' (Bayesian Information Criterion); 2.'aic' (Akaike information criterion)

  Default : 'bic'
- **plot** (*bool*) – if True the plot of correlation function is returned. Default: False

compute(*x*, *y*)
It computes restricted AR and unrestricted AR models, and evaluates whether the x signal could be forecasted by the y signal. F-value and p-value are computed, the interpretation of the results is up to the user.

Parameters

- **x** (*pd.DataFrame*) – first input signal - 'signal_to_predict'
- **y** (*pd.DataFrame*) – second input signal - 'helping_signal'

Returns dict – F_xy

plot()

It plots the results of SpectralGrangerCausality Test : F y->x is computed for each frequency (Hz), and then plotted

Returns plt.figure – A figure that contains the plot

MachineLearning package

Nonlinear package

NonlinearCorr module *Module author: Giovanna Varni*

class DataFrom2Persons.Monovariate.Continuous.Nonlinear.NonlinearCorr.**NonlinearCorr**(*nbins*)

It computes the nonparametric nonlinear regression coefficient h2 describing the dependency between two signals x and y (in pandas DataFrame format) in the most general way. It is equal to 0 when the two signals are independent, 1 when they are perfectly dependent.

Reference :

- F.Lopes da Silva, P. J.P., and B.P. Interdependence of eeg signals: linear vs. nonlinear associations and the signifcance of time delays and phase shifts. BrainTopography,2:9-18, 1989.

Parameters **nbins** (*int*) – number of bins in which the time series is divided into.

compute(*x*, *y*)

It computes the nonlinear correlation coefficient h2.

Parameters

- **x** (*pd.DataFrame*) – first input signal
- **y** (*pd.DataFrame*) – second input signal

Returns dict – nonlinear coefficient h2

MutualInformation module *Module author: Giovanna Varni*

class DataFrom2Persons.Monovariate.Continuous.Nonlinear.MutualInformation.**MutualInformation**(*n_neighbours*, *my_type=1*, *var_resc=True*, *noise=True*)

It computes Mutual Information (MI) estimators starting from entropy estimates from k-nearest-neighbours distances.

Reference :

- A.Kraskov, H.Stogbauer, and P.Grassberger. Estimating mutual information. Physical Review E, 69(6):066138, 2004

Parameters

- **n_neighbours** (*int*) – number of nearest neighbours
- **my_type** –

  Type of the estimators will be used to compute MI. Two options (1 and 2) are available: 1. the number of the points nx and ny is computed by taking into account only the points whose distance is stricly less than the distance of the k-nearest neighbours; 2. the number of the points nx and ny is computed by taking into account only the points whose distance is equal to or less than the distance of the k-nearest neighbours;

  Default: 1

:type my_type:int

Parameters **var_resc** – Boolean value indicating if the input signals should be rescaled at unitary variance. Default: False

:type var_resc:bool

Parameters **noise** – Boolean value indicating if a very low amplitude random noise should be added to the signals. It is done to avoid that there are many signals points having identical coordinates. Default: True

: type noise:bool

compute (*x*, *y*)

It computes Mutual Information.

Parameters

- **x** (*pd.DataFrame*) – first input signal
- **y** (*pd.DataFrame*) – second input signal

Returns dict – Mutual Information

Multivariate package

This package allows to compute synchronisation between continuous/categorical multivariate signals gathered from two persons. Each multivariate signal should be organized as a multivariate pandas DataFrame.

Categorical package

This package allows to compute synchronisation between categorical multivariate signals gathered from two persons.

Linear package

MachineLearning package

Nonlinear package

Continuous package

This package allows to compute synchronisation between continuous multivariate signals gathered from two persons.

Linear package

MachineLearning package

Nonlinear package

3.2.2 DataFromManyPersons package

This package allows to compute synchronisation between continuous/categorical monovariate and multivariate signals gathered from many persons.

Monovariate package

This package allows to compute synchronisation between monovariate signals gathered from many persons.

Categorical package

This package allows to compute synchronisation between categorical monovariate signals gathered from many persons.

Linear package

Nonlinear package

MachineLearning package

Continuous package

This package allows to compute synchronisation between continuous monovariate signals gathered from many persons.

Linear package

ConditionalGrangerCausality module

Module author: Adem Usta

class `DataFromManyPersons.Monovariate.Continuous.Linear.ConditionalGrangerCausality.`**`ConditionalGrangerCausality`**(*max_lag=1*, *criterion='bic'*, *plot=False*)

It computes a bi-directional pairwise Granger causality test between all the signals, to detect temporary direct links. For each detected link, a conditional test is made to know if the link between the two signals is mediated by another signal. At the end of the test, a node graphic is shown to see the links between the signals.

Reference :

- XiaotongWen, Govindan Rangarajan, and Mingzhou Ding. Multivariate Granger causality : an estimation framework based on factorization of the spectral density matrix. Philosophical Transactions of the Royal Society of London A : Mathematical, Physical and Engineering Sciences, 371(1997) :20110610, August 2013.

Parameters

- **max_lag** (*int*) – The number of maximum lag (in samples) with which the autoregressive model will be computed. It ranges in [1;length(x)]. Default :1.
- **criterion** (*str*) – A string that contains the name of the selected criterion to estimate optimal number of lags value. Two choices are possible :

  1. 'bic' (Bayesian Information Criterion);

2. 'aic' (Akaike information criterion)

Default : 'bic'

- **plot** (*bool*) – if True the plot of correlation function is returned. Default: False

compute (**signals*)

This method computes the ConditionalGrangerCausality. At the end of the computation, a graph is made to show the links between the signals.

Parameters **signals** (*list[pd.DataFrame]*) – list of signals, one per person.

Returns dict – matrix of links between the signals.

plot ()

It plots the final result of the module : a graphic that shows the links between the signals.

Returns plt.figure – A figure that contains the nodes graph

MultipleGrangerCausality module *Module author: Adem Usta*

class DataFromManyPersons.Monovariate.Continuous.Linear.MultipleGrangerCausality.**MultipleGrangerCausality** (*max_lag=1*, *criterion='bic'*, *plot=False*)

It computes a Granger causality test between one signal and a couple of other signals (in pandas DataFrame format). It computes unidirectionnal causality test with a multivariate autoregressive model and test if the unrestricted model is statistically significant compared to the restricted one. An F-test is computed and then the interpretation is up to the user.

Reference :

•Anil K. Seth. A MATLAB toolbox for Granger causal connectivity analysis. Journal of Neuroscience Methods, 186(2) :262-273, February 2010.

Parameters

- **max_lag** (*int*) – The number of maximum lag (in samples) with which the autoregressive model will be computed. It ranges in [1;length(x)]. Default :1.
- **criterion** (*str*) – A string that contains the name of the selected criterion to estimate optimal number of lags value. Two choices are possible :

  1.'bic' (Bayesian Information Criterion); 2 'aic' (for Akaike information criterion)

  Default : 'bic'

- **plot** – if True the plot of correlation function is returned. Default: False :type plot: bool

compute (**signals*)
It computes restricted AR and unrestricted AR models, and evaluates whether the first signal (first parameter) could be forecasted by the others. F-value and p-value are computed, the interpretation of the results is up to the user.

Parameters **signals** (*list[pd.DataFrame]*) – list of signals, one per person.

Returns dict – F-values and P-values.

plot ()
It plots the results of AR process for both restricted and unrestricted models :

Returns plt.figure – A figure that contains all the subplots

Omega_Complexity module *Module author: Giovanna Varni*

class DataFromManyPersons.Monovariate.Continuous.Linear.Omega_Complexity.**Omega_Complexity**
It computes Omega complexity among many monovariate signals (organized as a list of pandas DataFrame). It is a measure based on spatial principal component analysis (SPCA) carried out on the covariance matrix of the DataFrame. It ranges in [0,N], where 1 stands for maximum synchrony, N minimum synchrony.

Reference :

•Wackermann, J. Beyond mapping: estimating complexityof multichannel EEG recordings. Acta Neurobiol. Exp., 1996, 56:197-208.

compute (**signals*)
It computes the Omega complexity for multiple monovariate signals (organized as a list). If input signals are multivariates, only the first column of the signal is considered

Parameters **signals** (*list[pd.DataFrame]*) – list of signals, one per person.

Returns dict – omega

PartialCoherence module *Module author: Marie Avril*

class DataFromManyPersons.Monovariate.Continuous.Linear.PartialCoherence.**PartialCoherence** (*fs=1.0*, *NFFT=256*, *detrend=0*, *noverlap=0*)

It computes the partial coherence in a list of signals, 3 signals at a time.

Reference :

•Pereda, E. and al., Nonlinear multivariate analysis of neurophysiological signals. Progress in Neurobiology 77 (2005) I-37.

Parameters

- **fs** (*float*) – sampling frequency (in Hz) of the input signal. Default: 1.0
- **NFFT** (*int*) – length of each epoch (in samples). Default: 256
- **detrend** (*int*) –

  it specifies which kind of detrending should be computed on data. Ranges in [0;1]:

  1. 0 constant detrending;
  2. 1 linear detrending.

  Default: 0
- **noverlap** (*int*) – number of samples to overlap between epochs. Default: 0

compute (**signals*)

It computes the partial coherence between each signals.

Parameters **signals** (*list[pd.DataFrame]*) – list of signals, one per person.

Returns dict – partial coherence between each signal, organized in a dict: {z : {(x,y): K_xy_z}} with K_xy_z the partial coherence betwen signals[x] and signals[y] given all the linear informaiton of signals[z]

compute_partial_cross_spectrum (*X*, *Y*, *Z*)

It computes partial cross-spectrum between X and Y given all the linear information of Z

Parameters

- **X** (*pd.DataFrame*) – first signal
- **Y** (*pd.DataFrame*) – second signal
- **Z** (*pd.DataFrame*) – third signal

Returns np.array – partial cross-spectrum

S-Estimator module *Module author: Marie Avril*

class DataFromManyPersons.Monovariate.Continuous.Linear.S_Estimator.**S_Estimator** (*surr_nb_iter=100*, *plot=False*)

S_Estimator computes the S-Estimator, Genuine and Random Synchronization index among multiple monovariate signals (organized as a list of pandas DataFrame).

Reference :

- Cui, D. and al., Estimation of genuine and random synchronization in multivariate neural series. Neural Networks 23 (2010) 698-704.
- Andrzejak, R. and al., Bivariate surrogate techniques: Necessity, strengths, and caveats. Physical Review E 68, 066202 (2003).
- Schreiber, T. and al., Surrogate time series. Physica D 142 (2000) 346-382.

Parameters

- **surr_nb_iter** (*int*) – Number of surrogate iterations. Default : 100
- **plot** (*bool*) – if True the plot of surrogates signals is returned. Default: False

AAFT_surrogates (*Xi*)

Computes amplitude adjusted Fourier Transform (AAFT) method to create a surrogate signal. Get starting point / initial conditions for R surrogates

Parameters **Xi** (*np.array*) – signal to surrogate

Returns np.array – surrogated signal

compute (**signals*)

Computes SSI for multiple monovariate signals (organized as a list). If input signals are multivariates, only the first column of the signal is considered

Parameters **signals** (*list[pd.DataFrame]*) – list of signals, one per person.

Returns dict – Synchronization indexes : S-Estimator (SSI), Genuine Synchronization Index (GSI) and Random Synchronization Index (RSI)

getSynchronizationIndex (*lambda_i*)

Compute Synchronization Index (SI)

Parameters **lambda_i** (*np.array*) – normalized eigenvalues (depending on the type of SI)

Returns float – Synchronization index

plot ()

Plot surrogates signals

refined_AAFT_surrogate (*X*)

Computes an Iteratively refined amplitude adjusted Fourier Transform (AAFT) method to create a surrogate signal.

Note: Assume that original signal X is already standardized.

Parameters `X` (*pd.DataFrame*) – signal to surrogate

Returns pd.DataFrame – surrogated signal

Returns np.array – average eigvalues of surrogate signal among all iterations

MachineLearning package

Nonlinear package

Multivariate package

This package allows to compute synchronisation between multivariate signals gathered from many persons. All the multivariate panda DataFrame describing each person should be then organized as a list.

Categorical package

This package allows to compute synchronisation between categorical multivariate data streams gathered from many persons.

Linear package

MachineLearning package

Nonlinear package

Continuous package

This package allows to compute synchronisation between continuous multivariate data streams gathered from many persons.

Linear package

MachineLearning package

Nonlinear package

3.3 Toolbox Utils methods

3.3.1 Utils package

This package contains functionals of general utility directly used by synchrony methods or to preprocess the input signals

Align module

`utils.Align.Align`(*signal_1*, *signal_2*, *how='inner'*)

It aligns two monovariate signals (in pandas DataFrame format) according to their times indexes

Parameters

- **signal_1** (*pd.DataFrame*) – first monovariate signal
- **signal_2** (*pd.DataFrame*) – second monovariate signal
- **how** (*str*) – {'left', 'right', 'outer', 'inner'} How to handle indexes of the two objects for joining on index, None otherwise. Default: 'inner'.
  - left: use calling frame's index
  - right: use input frame's index
  - outer: form union of indexes
  - inner: use intersection of indexes

Returns pd.DataFrame – first aligned signal

Returns pd.DataFrame – second aligned signal

ConvertContinueToBinary module

`utils.ConvertContinueToBinary.`**`ConvertContinueToBinary`**(*signal*, *threshold=0*, *maximize=True*)

It converts a continue signal (in pandas DataFrame format) into a binany signal according to a rule defined by a threshold and a type of filter.

Parameters

- **`signal`** (*pd.DataFrame*) – input signal
- **`threshold`** (*float*) – value of the threshold. Default: 0
- **`maximize`** (*bool*) – is True if the conversion is done for values higher than the threshold. Default: True

Returns pd.DataFrame – binarized signal

Cpsd module

Module author: Giovanna Varni

`utils.Cpsd.`**`Cpsd`**(*x*, *y*, *fs=1.0*, *NFFT=256*, *detrend=0*, *noverlap=0*, *plot=False*)

It computes the cross power spectral density of two monovariate signals x and y (in pandas DataFrame format) by Welch's. This density is as the average of the density through the epochs (segments) of x and y and it is corrected for the power leakage due to (the hanning) windowing.

Parameters

- **`x`** (*pd.DataFrame*) – first input signal
- **`y`** (*pd.DataFrame*) – second input signal
- **`fs`** (*float*) – it is the sampling frequency of x (in Hz);
- **`NFFT`** (*int*) – it is the length of each epoch (segment);
- **`detrend`** (*bool*) – it specifies how the data can be detrended. Three options are avaliable: 1. 0 none; 2. 1 mean detrending; and 3. 1 linear detrending
- **`noverlap`** (*bool*) – it is the number of samples to overlap between epochs (segments);
- **`plot`** (*bool*) – if it is True the plot of the absolute of the density function is returned. Default: False

Returns dict – the cross power spectral density and the frequencies over which the coherence is computed

Detrend module

utils.Detrend.**Detrend**(*signal*, *det_type*)

It removes constant or linear trending in a monoviarate/multivariate signal (in pandas DataFrame format). In case of multivariate signal, detrending is carried out on each column of the DataFrame

Parameters

- **signal** (*pd.DataFrame*) – input signal
- **det_type** (*str*) – {'mean' 'linear'}

Returns pd.DataFrame – detrended signal

Distance module

It allows to compute several distance measures between monovariate/multivariate signals (in pandas DataFrame format).

utils.Distance.**Mahalanobis**(*df1*, *df2*)

It computes the Mahalanobis distance

Parameters

- **df1** (*pd.DataFrame*) – first input signal
- **df2** (*pd.DataFrame*) – second input signal

Returns float – distance between the two signals

utils.Distance.**Minkowski**(*x*, *y*, *order*)

It computes the Minkowski distance of order p (p cannot be less than 1).

1. p = 1, Manhattan distance;
2. p = 2, Euclidean distance; and
3. p = np.inf, Cebysev distance

Parameters

- **x** (*pd.DataFrame*) – first input signal
- **y** (*pd.DataFrame*) – second input signal
- **order** – the order of the distance to be computed

Returns float – a pandas DataFrame with the p-order distance between the two signals

ExtractSignal module

utils.ExtractSignal.**ExtractSignalFromCSV**(*filename, separator=',', unit='ms', columns=['all']*)

It extracts a signal from a .csv file (organized by columns, with first one corresponding to time index)

Parameters

- **filename** (*str*) – complete path + filename to the csv file.
- **separator** (*str*) – separator between columns in the csv file. Default: ','
- **unit** (*str*) – Time unit for the index. Default = 'ms'
- **columns** (*list*) – array containing columns name of index wanted for the signal. Default: 'all'

Returns pd.DataFrame – Extracted signal

utils.ExtractSignal.**ExtractSignalFromELAN**(*filename, separator=',', unit='s', columns_name=['Actor', ' ', 't_begin', 't_end', 'duration', 'Action', 'video'], total_duration=0, ele_per_sec=1, Actor='', Action='all'*)

It extracts a boolean signal from ELAN output annotations. It returns a boolean signal, a DataFrame with milliseconds timestamps. The freqency of timestamps is defined by 'ele_per_sec'. The signal is True between two timestamps if in the file, the actor defined in 'Actor' pararameter is doing the action defined in 'Action'.

Parameters

- **filename** (*str*) – complete path + filename to the csv file out from ELAN
- **separator** (*str*) – separator between columns in the csv file. Default: ','
- **unit** (*str*) – Time unit for the index. Default = 's'
- **columns_name** (*list*) – array containing the names of each columns in ELAN File in the correct order It must contain at lest these exacts elements : 'Actor', 't_begin', 't_end', 'Action' if a colunm is empty, give '' as name. Default: ['Actor', ' ', 't_begin', 't_end', 'duration', 'Action', 'video']
- **total_duration** (*int*) – the total duration attempted for the signal, in time unit given by 'unit'. If zero is given, the total duration will be computed as the end of the last event recorded in ELAN file. Default: 0
- **ele_per_sec** (*int*) – Number of element wanted per second in the computed signal. Default = 1
- **Actor** (*str*) – Name of the Actor in the ELAN annotation file
- **Action** (*str*) – Name of the Action in the ELAN annotation file. Default ='all'

Returns pd.DataFrame – Monovariate boolean signal, with 1 at timestamps corresponding to the Action of the Actor, timestamps in ms `utils.ExtractSignal.`**`ExtractSignalFromMAT`**(*filename, columns_index=['all'], columns_wanted_names=['all'], unit='ms'*)

It extracts a signal from a .mat MATLAB file (organized by columns, with first one corresponding to time index)

Parameters

- **`filename`** (*str*) – complete path + filename to the mat file.
- **`columns_index`** (*list*) – array containing columns indexes of index wanted for the signal. Default: 'all'
- **`columns_wanted_names`** (*list*) – array containing columns names wanted for the signal. Default: 'all' ('0', '1' ...)
- **`unit`** (*str*) – Time unit for the index. Default = 'ms'

Returns pd.DataFrame – Extracted signal

Normalize module

`utils.Normalize.`**`Normalize`**(*signal, min_value=[0], max_value=[1]*)

It normalizes function normalizes signal between min_value and max_value

Parameters

- **`signal`** (*pd.DataFrame*) – input signal
- **`min_value`** (*array*) – minimal value desired. Default: [0]
- **`max_value`** (*array*) – maximal value desired. Default: [1]

Returns pd.DataFrame – normalized signal

PeakDetect module

`utils.PeakDetect.`**`peakdetect`**(*y_axis, x_axis=None, lookahead=300, delta=0*)

It discovers peaks by searching for values which are surrounded by lower or larger values for maxima and minima, respectively. This script is converted from/based on a MATLAB script at: http://billauer.co.il/peakdet.html

for detecting local maxima and minmia in a signal.

Parameters

- **`y_axis`** – a list containg the signal over which to find peaks
- **`x_axis`** – (optional) a x-axis whose values correspond to the y_axis list and is used in the return to specify the postion of the peaks. If omitted an index of the y_axis is used. Default: None

- **lookahead** – (optional) distance to look ahead from a peak candidate to determine if it is the actual peak (default: 200) '(sample / period) / f' where '4 >= f >= 1.25' might be a good value
- **delta** – (optional) this specifies a minimum difference between a peak and the following points, before a peak may be considered a peak. Useful to hinder the function from picking up false peaks towards to end of the signal. To work well delta should be set to delta >= RMSnoise * 5. Default: 0 delta function causes a 20% decrease in speed, when omitted. Correctly used it can double the speed of the function

Returns two lists [max_peaks, min_peaks] containing the positive and negative peaks, respectively. Each cell of the lists contains a tuple of: (position, peak_value) to get the average peak value do: np.mean(max_peaks, 0)[1] on the results to unpack one of the lists into x, y coordinates do: x, y = zip(*tab)

`utils.PeakDetect.`**`peakdetect_fft`**(*y_axis*, *x_axis*, *pad_len=5*)

It performs a FFT calculation on the data and zero-pads the results to increase the time domain resolution after performing the inverse fft and send the data to the 'peakdetect' function for peak detection.

Omitting the x_axis is forbidden as it would make the resulting x_axis value silly if it was returned as the index 50.234 or similar.

Will find at least 1 less peak then the 'peakdetect_zero_crossing' function, but should result in a more precise value of the peak as resolution has been increased. Some peaks are lost in an attempt to minimize spectral leakage by calculating the fft between two zero crossings for n amount of signal periods.

The biggest time eater in this function is the ifft and thereafter it's the 'peakdetect' function which takes only half the time of the ifft. Speed improvementd could include to check if 2**n points could be used for fft and ifft or change the 'peakdetect' to the 'peakdetect_zero_crossing', which is maybe 10 times faster than 'peakdetct'. The pro of 'peakdetect' is that it resutls in one less lost peak. It should also be noted that the time used by the ifft function can change greatly depending on the input.

Parameters

- **y_axis** – a list containg the signal over which to find peaks
- **x_axis** – a x-axis whose values correspond to the y_axis list and is used in the return to specify the postion of the peaks.
- **pad_len** – (optional) how many times the time resolution should be increased by, e.g. 1 doubles the resolution. The amount is rounded up to the nearest 2 ** n amount (default: 5)

Returns two lists [max_peaks, min_peaks] containing the positive and negative peaks respectively. Each cell of the lists contains a tupple of: (position, peak_value) to get the average peak value do: np.mean(max_peaks, 0)[1] on the results to unpack one of the lists into x, y coordinates do: x, y = zip(*tab)

`utils.PeakDetect.`**`peakdetect_parabole`**(*y_axis*, *x_axis*, *points=9*)

IT detects local maxima and minima in a signal. Discovers peaks by fitting the model function: y = k (x - tau) ** 2 + m to the peaks. The amount of points used in the fitting is set by the points argument.

Omitting the x_axis is forbidden as it would make the resulting x_axis value silly if it was returned as index 50.234 or similar.

will find the same amount of peaks as the 'peakdetect_zero_crossing' function, but might result in a more precise value of the peak.

Parameters

- **y_axis** – a list containg the signal over which to find peaks
- **x_axis** – a x-axis whose values correspond to the y_axis list and is used in the return to specify the postion of the peaks.
- **points** – (optional) How many points around the peak should be used during curve fitting, must be odd (default: 9)

Returns two lists [max_peaks, min_peaks] containing the positive and negative peaks respectively. Each cell of the lists contains a list of: (position, peak_value) to get the average peak value do: np.mean(max_peaks, 0)[1] on the results to unpack one of the lists into x, y coordinates do: x, y = zip(*max_peaks)

utils.PeakDetect.**peakdetect_sine**(*y_axis*, *x_axis*, *points=9*, *lock_frequency=False*)

It detects local maxima and minima in a signal. It discovers peaks by fitting the model function: y = A * sin(2 * pi * f * x - tau) to the peaks. The amount of points used in the fitting is set by the points argument.

Omitting the x_axis is forbidden as it would make the resulting x_axis value silly if it was returned as index 50.234 or similar.

will find the same amount of peaks as the 'peakdetect_zero_crossing' function, but might result in a more precise value of the peak.

The function might have some problems if the sine wave has a non-negligible total angle i.e. a k*x component, as this messes with the internal offset calculation of the peaks, might be fixed by fitting a k * x + m function to the peaks for offset calculation.

Parameters

- **y_axis** – a list containg the signal over which to find peaks
- **x_axis** – a x-axis whose values correspond to the y_axis list and is used in the return to specify the postion of the peaks.
- **points** – (optional) How many points around the peak should be used during curve fitting, must be odd (default: 9)
- **lock_frequency** – (optional) Specifies if the frequency argument of the model function should be locked to the value calculated from the raw peaks or if optimization process may tinker with it. (default: False)

Returns two lists [max_peaks, min_peaks] containing the positive and negative peaks respectively. Each cell of the lists contains a tupple of: (position, peak_value) to get the average peak value do: np.mean(max_peaks, 0)[1] on the results to unpack one of the lists into x, y coordinates do: x, y = zip(*tab)

utils.PeakDetect.**peakdetect_sine_locked**(*y_axis*, *x_axis*, *points=9*)

It is a convinience function for calling the 'peakdetect_sine' function with the lock_frequency argument as True.

Parameters

- **y_axis** – a list containg the signal over which to find peaks
- **x_axis** – a x-axis whose values correspond to the y_axis list and is used in the return to specify the postion of the peaks.
- **points** – (optional) how many points around the peak should be used during curve fitting, must be odd (default: 9)

Returns see 'peakdetect_sine'

`utils.PeakDetect.`**`peakdetect_zero_crossing`**(*y_axis*, *x_axis=None*, *window=11*)

It detects local maxima and minima in a signal. It discovers peaks by dividing the signal into bins and retrieving the maximum and minimum value of each the even and odd bins respectively. Division into bins is performed by smoothing the curve and finding the zero crossings.

Suitable for repeatable signals, where some noise is tolerated. Excecutes faster than 'peakdetect', although this function will break if the offset of the signal is too large. It should also be noted that the first and last peak will probably not be found, as this function only can find peaks between the first and last zero crossing.

Parameters

- **`y_axis`** – a list containg the signal over which to find peaks
- **`x_axis`** – (optional) a x-axis whose values correspond to the y_axis list and is used in the return to specify the postion of the peaks. If omitted an index of the y_axis is used. (default: None)
- **`window`** – the dimension of the smoothing window; should be an odd integer (default: 11)

Returns two lists [max_peaks, min_peaks] containing the positive and negative peaks respectively. Each cell of the lists contains a tupple of: (position, peak_value) to get the average peak value do: np.mean(max_peaks, 0)[1] on the results to unpack one of the lists into x, y coordinates do: x, y = zip(*tab)

`utils.PeakDetect.`**`zero_crossings`**(*y_axis*, *window=11*)

Algorithm to find zero crossings. Smoothes the curve and finds the zero-crossings by looking for a sign change.

Parameters

- **`y_axis`** – a list containg the signal over which to find zero-crossings
- **`window`** – the dimension of the smoothing window; should be an odd integer (default: 11)

Returns the index for each zero-crossing

ResampleAndInterpolate module

`utils.ResampleAndInterpolate.`**`ResampleAndInterpolate`**(*signal*, *rule='100ms'*, *limit=None*)

It resamples signal and does linear interpolation to values added by the resampling. Signal must have DateTime index.

Parameters

- **`signal`** (*pd.DataFrame*) – monovariate signal
- **`rule`** (*str*) – string with the resampling rule (ex: for 100ms resampling, rule='100ms'). Default: '100ms'
- **`limit`** (*int*) – for interpolation, maximum number of consecutive NaN values to fill. Default: None

Returns pd.DataFrame – resampled signal with linear interpolation of added data

Standardize module

`utils.Standardize.`**`Standardize`**(*signal*)

It standardizes a monovariate/multivariate signals (in pandas DataFrame format) so that it has mean equal to zero and unitary variance. In case of a multivariate signal, standardization is carried out on each column of the DataFrame.

Parameters **`signal`** (*pd.DataFrame*) – input signal

Returns pd.DataFrame – standardized signal

Trafo module

`utils.Trafo.`**`Trafo`**(*signal*, *sk*, *trafo_type*, *log_base=2*)

It transforms a monovariate/multivariate signals (in pandas DataFrame format) in a new signal by applying a square root or logaritmic or inverse transformation.

Parameters

- **`signal`** (*pd.DataFrame*) – input signal
- **`sk`** (*str*) – {'pos','neg'} the skewness of signal distribution.
- **`trafo_type`** (*str*) – {'sqrt','log','inv'} the kind of tranformation should be applied
- **`log_base`** (*int*) –

  The base of the log. Available options:

  1. 2.0;
  2. np.e; and
  3. 10.0.

  Default: 2

Returns pd.DataFrame – transformed signal

Welch_psd module

Module author: Giovanna Varni

`utils.Welch_psd.`**`Welch_psd`**(*x*, *fs=1.0*, *NFFT=256*, *detrend=0*, *noverlap=0*, *plot=False*)

It computes the Welch's power spectral density of a real signal x (in pandas DataFrame format). This density is as the average of the density through the epochs (segments) of x and it is corrected for the power leakage due to (the hanning) windowing.

Parameters

- **x** (*pd.DataFrame*) – input signal
- **fs** (*float*) – it is the sampling frequency of x (expressed in Hz);
- **NFFT** (*int*) – it is the length of each epoch (segment);
- **detrend** (*bool*) –

  it specifies how the data can be detrended. Three options are avaliable:

  1. 0, none detrending;
  2. 1, mean detrending; and
  3. 1, linear detrending
- **noverlap** (*bool*) – it is the number of samples to overlap between epochs (segments);
- **plot** (*bool*) – if it is True the plot of the density function is returned. Default: False

Returns dict – the power spectral density and the frequencies over which the coherence is computed (keys : psd, Frequency)

3.4 Indices and tables

- genindex
- modindex
- search

CHAPTER 4

SyncPy library Examples

Available source code examples

- *DataFrom2Persons Monovariate Categorical Linear module*
  - *Boolean Turn Activity example*
- *DataFrom2Persons Monovariate Continuous Linear module*
  - *Coherence example*
  - *Correlation example*
  - *Granger Causality example*
  - *Spectral Granger Causality example*
  - *Window Cross Correlation example*
  - *Window Cross Correlation with PeakPicking example*
- *DataFrom2Persons Monovariate Continuous NonLinear module*
  - *Nonlinear Correlation example*
- *DataFromManyPersons Monovariate Continuous Linear module*
  - *PartialCoherence example*
  - *S_Estimator example*
- *DataFromManyPersons Multivariate Continuous Linear module*
  - *Multivariate Granger Causality example*

4.1 DataFrom2Persons Monovariate Categorical Linear module

4.1.1 Boolean Turn Activity example

(Source code)

```
"""
BooleanTurnsActivity example :
Computes data turns statistics between two boolean monovariate signals (in DataFrame format) x and y :
x signal activity duration, y signal activity duration, pause duration, overlap duration,
x signal pause duration, y signal pause duration, pause duration between x and y activity,
synchrony ratios between x and y (defined by max_latency)
"""

""" Import common python packages """
import sys
import os
import numpy as np           # Mathematical package
import pandas as pd          # Time serie package
import matplotlib.pyplot as plt # Plotting package
sys.path.insert(0, '../src/')    # To be able to import packages from parent directory

print("\n")
print("**************************************************************************************")
print("This script computes the boolean turn activty of two categorical monovariate signals \n")
print("**************************************************************************************")

""" Import wanted module with every parent packages """
import DataFrom2Persons.Monovariate.Categorical.Linear.BooleanTurnsActivity as BooleanTurnsActivity

""" Import Utils modules """
from utils.ExtractSignal import ExtractSignalFromELAN

""" Define signals in pd.dataFrame format """
'''
# Create signals
user0_data = pd.DataFrame({'X':[0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0]})
user1_data = pd.DataFrame({'Y':[0, 0, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1]})
'''

# Import signals from .csv file, example with an ELAN data file
```

```
filename = 'data_examples/ELAN_2Persons.csv'
user0_data = ExtractSignalFromELAN(filename, separator=';', total_duration = 240,
                                    ele_per_sec = 5, Actor = 'Maman', Action = 'all')
user1_data = ExtractSignalFromELAN(filename, separator=';', total_duration = 240,
                                    ele_per_sec = 5, Actor = 'Bebe', Action = 'all')

# plot input signals
ele_per_sec = 5
n = [float(x)/ele_per_sec for x in range(user0_data.size)] # create x axis values
plt.ion()
fig = plt.figure()
ax = fig.add_subplot(111)
ax.grid(True)
ax.set_xlabel('Time (ms)')
ax.set_title('Input signals')
ax.set_ylim(0, 1.5)
ax.plot(n,[float(y)/2 for y in user0_data.values],'g.', label=user0_data.columns[0])
ax.plot(n, user1_data.values, 'b.', label=user1_data.columns[0])
plt.legend(bbox_transform=plt.gcf().transFigure)

""" Define class attributes of the wanted method """
max_latency = 3.0           # the maximal delay between the two signals activity to define synchrony (in second)
min_pause_duration = 0.01   # minimal time for defining a pause (in second)
ele_per_sec = 5             # number of element in one second. Default: 1
duration = -1               # total activity duration (in second). Default or -1 : len(x)*ele_per_sec

""" Instanciate the class with its attributes """
print("\n")

try :
    turns = BooleanTurnsActivity.BooleanTurnsActivity(max_latency, min_pause_duration, ele_per_sec, duration)
except TypeError, err :
    print("TypeError in BooleanTurnsActivity constructor : \n" + str(err))
    sys.exit(-1)
except ValueError, err :
    print("ValueError in BooleanTurnsActivity constructor : \n" + str(err))
    sys.exit(-1)
except Exception, e :
    print("Exception in BooleanTurnsActivity constructor : \n" + str(e))
    sys.exit(-1)

print("An instance the class is now created with the following parameters:\n" +
```

```
      "maximal latency = " + str(max_latency) + "\n" +
      "minimal pause duration = " + str(min_pause_duration) + "\n" +
      "number of element per second = " + str(ele_per_sec) + "\n" +
      "duration = " + str(duration))

""" Compute the method and get the result """
try :
    res_turns, turns_ratio = turns.compute(user0_data, user1_data)
except TypeError, err :
    print("TypeError in BooleanTurnsActivity computation : \n" + str(err))
    sys.exit(-1)
except ValueError, err :
    print("ValueError in BooleanTurnsActivity computation : \n" + str(err))
    sys.exit(-1)
except Exception, e :
    print("Exception in BooleanTurnsActivity computation : \n" + str(e))
    sys.exit(-1)

""" Display result """
print("\n")
print("**************************************** \n")
print('Boolean turns activity complete result :')
print("****************************************\n")
print(res_turns)
print(turns_ratio)

""" Get simple statistics of the result """
stats = res_turns.describe()

""" Display statistics result """
print("\n")
print("***********************************\n")
print('Boolean turns activity statistics :')
print("***********************************\n")
print(stats)

raw_input("Push ENTER key to exit.")
```

4.2 DataFrom2Persons Monovariate Continuous Linear module

4.2.1 Coherence example

(Source code)

```
"""
Coherence example :
It computes coherence fucntion between two monovariate signals (in DataFrame format) x and y.
"""

""" Import common python packages """

""" Import common python packages """
import sys
import os
import numpy as np          # Mathematical package
import pandas as pd         # Time serie package
import matplotlib.pyplot as plt # Plotting package
sys.path.insert(0, '../src/')   # To be able to import packages from parent directory

print("\n")
print("**************************************************************************************")
print("This script computes the coherence of two continuous monovariate signals. \n" +
      "First input is a sum among a sinewave of 100 Hz frequency,  a cosinewave \n" +
      "of 200 Hz frequency and unformly distributed random noise. The second one\n" +
      "is a sub-multiple of the first one.")
print("**************************************************************************************")

""" Import wanted module with every parent packages """

import DataFrom2Persons.Monovariate.Continuous.Linear.Coherence as Coherence


""" Import Utils modules """
from utils import Standardize
from utils.ExtractSignal import ExtractSignalFromCSV


""" Define signals in pd.dataFrame format """
Fs=1000.0 # sampling frequency (Hz)
```

```
t=np.arange(0,1-1.0/Fs,1.0/Fs) #number or samples

# Create signals
x = pd.DataFrame({'X':np.cos(2*np.pi*100*t)+np.sin(2*np.pi*200*t)+np.random.randn(t.size)})
y = pd.DataFrame({'Y':0.5*np.cos(2*np.pi*100*t-np.pi/4)+0.35*np.sin(2*np.pi*200*t-np.pi/2)+0.5*np.random.randn(t.size)})


'''
"""OR"""
""" Import signals from a .csv file """
#Data from files
filename = 'data_examples/2Persons_Monovariate_Continuous_data.csv'

x = ExtractSignalFromCSV(filename, columns = ['x1'])
y = ExtractSignalFromCSV(filename, columns = ['x2'])
t=np.arange(0,x.shape[0])
'''


""" Plot input signals """
plt.ion()
f, axarr = plt.subplots(2, sharex=True)
axarr[0].set_title('Input signals')
axarr[0].set_xlabel('Samples')
axarr[1].set_xlabel('Samples')
axarr[0].plot(t, x, label="x")
axarr[1].plot(t, y, label="y", color='r')
axarr[0].legend(loc='best')
axarr[1].legend(loc='best')

""" Define class attributes of the wanted method """
Fs=1000.0       # sampling frequency (Hz)
NFFT = 100      # length of each epoch
detrend = 0     # remove constant detrending
noverlap = 80   # number of points of overlap between epochs
plot = True     # plot of the coherence function

""" Instanciate the class with its attributes """
print("\n")

try :
```

```
    c = Coherence.Coherence(Fs, NFFT, detrend, noverlap, plot=True)
except TypeError, err :
    print("TypeError in Coherence constructor : \n" + str(err))
    sys.exit(-1)
except ValueError, err :
    print("ValueError in Coherence constructor : \n" + str(err))
    sys.exit(-1)
except Exception, e :
    print("Exception in Coherence constructor : \n" + str(e))
    sys.exit(-1)


print("An instance the class is now created with the following parameters:\n" +
      "NFFT = " + str(NFFT) + "\n" +
      "detrend = " + str(detrend) + "\n" +
      "noverlap= " + str(noverlap) + "\n" +
      "plot = " + str(plot))

""" Compute the method and get the result """
print("\n")
print("Computing...")

try :
    res = c.compute(x,y)
except TypeError, err :
    print("TypeError in Coherence computation : \n" + str(err))
    sys.exit(-1)
except ValueError, err :
    print("ValueError in Coherence computation : \n" + str(err))
    sys.exit(-1)
except Exception, e :
    print("Exception in Coherence computation : \n" + str(e))
    sys.exit(-1)

""" Display result """
print("\n")
print("**************************************** \n")
print('Coherence complete result :')
print("****************************************\n")
print(res['Coherence'])
print(res['Frequency'])
```

```
raw_input("Push ENTER key to exit.")
```

4.2.2 Correlation example

(Source code)

```
"""
Correlation example:
It computes the linear correlation between two monovariate signals x and y (in DataFrame format) as a function of their delay tau.
It computes autocorrelation when y coincides with x.
"""

""" Import common python packages """
import sys
import os
import numpy as np      # Mathematical package
import pandas as pd     # Time serie package
import matplotlib.pyplot as plt # Plotting package
sys.path.insert(0, '../src/')   # To be able to import packages from parent directory

print ("\n")
print("**************************************************************************************************************")
print("This scripts computes the correlation between two monovariate signals."
       "First input is a sinewave of 1 Hz frequency, the second one\n is the sum of this sinewave"
       "with a gaussian random process having zero mean and unitary\n variance.")
print("***************************************************************************************************************")

""" Import wanted module with every parent packages """
import DataFrom2Persons.Monovariate.Continuous.Linear.Correlation as Correlation

""" Import Utils modules """
from utils import Standardize
from utils.ExtractSignal import ExtractSignalFromCSV


""" Define signals in pd.dataFrame format """

#Define parameters
N=1024 # number of samples
f=1.0  # sinewave frequency (Hz)
```

```
Fs=200 # sampling frequency (Hz)

n=np.arange(0,N) #number of samples

# Create signals
x = pd.DataFrame({'X':np.sin(2*3.14*f*n/Fs)}, np.arange(0,N))
y = pd.DataFrame({'Y':np.sin(2*3.14*f*n/Fs)+10*np.random.randn(1,N)[0]},np.arange(0,N))

'''
"""OR"""
""" Import signals from a .csv file """
#Data from files
filename = 'data_examples/2Persons_Monovariate_Continuous_data.csv'

x = ExtractSignalFromCSV(filename, columns = ['x1'])
y = ExtractSignalFromCSV(filename, columns = ['x2'])
n=np.arange(0,x.shape[0])
'''



"""Plot input signals"""
plt.ion()
f, axarr = plt.subplots(2, sharex=True)
axarr[0].set_title('Input signals')
axarr[0].set_xlabel('Samples')
axarr[1].set_xlabel('Samples')

axarr[0].plot(n, x, label="x")
axarr[1].plot(n, y, label="y", color='r')
axarr[0].legend(loc='best')
axarr[1].legend(loc='best')



""" Define class attributes of the wanted method """

tau_max = 999                       # the maximum lag at which correlation should be computed (in samples)
plot=True                           # plot of the correlation fucntion
standardization = True              # standardization of the time series to mean 0 and variance 1
corr_tau_max = True                 # return of the maximum of correlation and its lag
corr_coeff = True                   # computation of the correlation coefficient (Pearson's version)
```

```
scale=True                          # scale factor to have correlaton in [-1,1]

""" Instanciate the class with its attributes """
print("\n")

try :
    c=Correlation.Correlation(tau_max, plot, standardization, corr_tau_max, corr_coeff, scale)
except TypeError, err :
    print("TypeError in Correlation constructor : \n" + str(err))
    sys.exit(-1)
except ValueError, err :
    print("ValueError in Correlation constructor : \n" + str(err))
    sys.exit(-1)
except Exception, e :
    print("Exception in Correlation constructor : \n" + str(e))
    sys.exit(-1)

print("An instance the class is now created with the following parameters:\n" +
      "tau max = " + str(tau_max) + "\n" +
      "plot = " + str(plot) + "\n" +
      "standardization= " + str(standardization) + "\n" +
      "corr_tau_max = " + str(corr_tau_max) + "\n" +
      "corr_coeff =" + str(corr_coeff) +"\n" +
      "scale =" + str(scale))

""" Compute the method and get the result """
print("\n")
print("Computing...")

try :
    res= c.compute(x, y)
except TypeError, err :
    print("TypeError in Correlation computation : \n" + str(err))
    sys.exit(-1)
except ValueError, err :
    print("ValueError in Correlation computation : \n" + str(err))
    sys.exit(-1)
except Exception, e :
    print("Exception in Correlation computation : \n" + str(e))
    sys.exit(-1)

""" Display result """
```

```
print("\n")
print("**************************************** \n")
print('Correlation complete result :')
print("****************************************\n")
print("Correlation function array:")
print(res['corr_funct'])
print("Maximum value of the correlation %f and lag (in samples) %d:" %(res['max_corr'],res['t_max']))
print("Pearson's correlation coefficient %f:" %(res['corr_coeff']))



raw_input("Push ENTER key to exit.")
plt.close("all")
```

4.2.3 Granger Causality example

(Source code)

```
"""
GrangerCausality example :
Computes a Granger Causality test between two signals x and y that are stored as a DataFrame
"""

""" Import common python packages """
import sys
import numpy as np              # Mathematical package
sys.path.insert(0, '../src/')       # To be able to import from parent directory

print("\n")
print("*************************************************************************************")
print("This script computes the Granger Causality test between two monovariate signals \n" +
      "in pandas DataFrame format.")
print("*************************************************************************************")

""" Import Utils modules """
from utils.ExtractSignal import ExtractSignalFromCSV
from utils.ResampleAndInterpolate import ResampleAndInterpolate

""" Import wanted module with every parent packages """
import DataFrom2Persons.Monovariate.Continuous.Linear.GrangerCausality as GC
```

```
""" Import signal from a .csv file """
filename = 'data_examples/2Persons_Monovariate_Continuous_data.csv'
print "\nLoading signals from csv files : ", filename,"\n"
x1 = ExtractSignalFromCSV(filename, columns = ['x1'])
x2 = ExtractSignalFromCSV(filename, columns = ['x2'])

# Resample and Interpolate data to have constant frequency
x1 = ResampleAndInterpolate(x1, rule='200ms', limit=5)
x2 = ResampleAndInterpolate(x2, rule='200ms', limit=5)

""" Define class attributes """
max_lag = 3                          # Define the maximum lag acceptable to estimate autoregressive models
criterion = 'bic'            # Define the criterion to estimate the optimal number of lags to estimate autoregressive models
plot = True                                # Authorize the plot of the results

""" Instanciate the class with its attributes """
print("\n")

try :
        gc = GC.GrangerCausality(max_lag = max_lag, criterion = criterion, plot = plot)
except TypeError, err :
        print("TypeError in GrangerCausality constructor : \n" + str(err))
        sys.exit(-1)
except ValueError, err :
        print("ValueError in GrangerCausality constructor : \n" + str(err))
        sys.exit(-1)
except Exception, e :
        print("Exception in GrangerCausality constructor : \n" + str(e))
        sys.exit(-1)

print("An instance of the class is now created with the following parameters:\n" +
                "max_lag = " + str(max_lag) + "\n" +
                "criterion = " + str(criterion) + "\n" +
                "plot = " + str(plot))

""" Compute the method and get the result """
print("\n")
print("Computing...\n")

try :
        results = gc.compute(x1,x2)
```

```
except TypeError, err :
        print("TypeError in GrangerCausality computation : \n" + str(err))
        sys.exit(-1)
except ValueError, err :
        print("ValueError in GrangerCausality computation : \n" + str(err))
        sys.exit(-1)
except Exception, e :
        print("Exception in GrangerCausality computation : \n" + str(e))
        sys.exit(-1)

# Displaying results :
print "Computing autoregressive model 'restricted' and 'unrestricted' via the 'Ordinary Least Squares' method\n"

print "According to",gc._criterion,", the optimal number of lag estimated is :", gc._olag,"\n"

print "Printing RESULTS  ...\n"

print "RESTRICTED model :\n"

print "        Coefficients :\n"
for i in range(0,gc._olag):
        print"       lag",i+1,":",gc._OLS_restricted.params[i]
print "\n"
print "        Variance of residual error :", np.var(gc._OLS_restricted.resid),"\n"

print "UNRESTRICTED model :\n"

print "        Coefficients of 'signal_to_predict' :\n"
for i in range(0,gc._olag):
        print"       lag",i+1,":",gc._OLS_unrestricted.params[i]
print "\n"

print "        Coefficients of 'helping_signal' :\n"
for i in range(0,gc._olag):
        print"       lag",i+1,":",gc._OLS_unrestricted.params[i+gc._olag]
print "\n"

print "        Variance of residual error :", np.var(gc._OLS_unrestricted.resid),"\n"

print "F_value =",gc._F_value," with p_value =",gc._p_value,"\n"

raw_input("Push ENTER key to exit.")
```

4.2.4 Spectral Granger Causality example

(Source code)

```
"""
SpectralGrangerCausality example :
Computes a Spectral Granger Causality test between two signals x and y that are stored as a DataFrame
"""

""" Import common python packages """
import sys
import os
import numpy as np              # Mathematical package
import pandas as pd             # Time serie package
import matplotlib.pyplot as plt # Plotting package
sys.path.insert(0, '../src/')       # To be able to import from parent directory

print("\n")
print("**********************************************************************************")
print("This script computes the Granger Causality test in the spectral domain between two monovariate signals \n" +
      "expressed as Python Pandas DataFrame.")
print("**********************************************************************************")

""" Import Utils modules """
from utils.ExtractSignal import ExtractSignalFromCSV
from utils.ResampleAndInterpolate import ResampleAndInterpolate

""" Import wanted module with every parent packages """
import DataFrom2Persons.Monovariate.Continuous.Linear.SpectralGrangerCausality as SGC

""" Import signal from a .csv file """
filename = 'data_examples/data_jouet_4.csv'
print "\nLoading signals from csv files : ", filename,"\n"
x1 = ExtractSignalFromCSV(filename, columns = ['x1'])
x2 = ExtractSignalFromCSV(filename, columns = ['x2'])
""" Define class attributes """
max_lag =10        # Define the maximum lag acceptable to estimate autoregressive models
criterion = 'bic'        # Define the criterion to estimate the optimal number of lags to estimate autoregressive models
plot = True                # Authorize the plot of the results

""" Instanciate the class with its attributes """
print("\n")
```

```
try :
        sgc = SGC.SpectralGrangerCausality(max_lag = max_lag, criterion = criterion, plot = plot)
except TypeError, err :
        print("TypeError in SpectralGrangerCausality constructor : \n" + str(err))
        sys.exit(-1)
except ValueError, err :
        print("ValueError in SpectralGrangerCausality constructor : \n" + str(err))
        sys.exit(-1)
except Exception, e :
        print("Exception in SpectralGrangerCausality constructor : \n" + str(e))
        sys.exit(-1)

print("An instance of the class is now created with the following parameters:\n" +
                "max_lag = " + str(max_lag) + "\n" +
                "criterion = " + str(criterion) + "\n" +
                "plot = " + str(plot))
""" Compute the method and get the result """
print("\n")
print("Computing...\n")
try :
        results = sgc.compute(x1,x2)
except TypeError, err :
        print("TypeError in SpectralGrangerCausality computation : \n" + str(err))
        sys.exit(-1)
except ValueError, err :
        print("ValueError in SpectralGrangerCausality computation : \n" + str(err))
        sys.exit(-1)
except Exception, e :
        print("Exception in SpectralGrangerCausality computation : \n" + str(e))
        sys.exit(-1)

# Displaying results :
print "Computing autoregressive model 'restricted' and 'unrestricted' via the 'Ordinary Least Squares' method\n"

print "According to",sgc._criterion,", the optimal number of lag estimated is :", sgc._olag,"\n"

print "Printing RESULTS  ...\n"

raw_input("Push ENTER key to exit.")
```

4.2.5 Window Cross Correlation example

(Source code)

```
"""
WindowCrossCorrelation example :
Computes the window cross correlation between times series (in DataFrame format) x and y
"""

""" Import common python packages """
import sys
import os
import numpy as np              # Mathematical package
import pandas as pd             # Time serie package
import matplotlib.pyplot as plt # Plotting package
sys.path.insert(0, '../src/')       # To be able to import from parent directory

print("\n")
print("**********************************************************************************")
print("This script computes the windowed cross-correlation between two monovariate time series \n" +
      "expressed as Python Pandas DataFrame.")
print("**********************************************************************************")

""" Import wanted module with every parent packages """
import DataFrom2Persons.Monovariate.Continuous.Linear.WindowCrossCorrelation as WindowCrossCorrelation

""" Import Utils modules """
from utils.ExtractSignal import ExtractSignalFromCSV
from utils.ExtractSignal import ExtractSignalFromMAT
from utils.ResampleAndInterpolate import ResampleAndInterpolate

'''
""" Define signals in pd.dataFrame format """
# preparing the input time series
N = 20              # number of samples
f = 1.0              # sinewave frequency (Hz)
Fs = 200             # sampling frequency (Hz)
n = np.arange(0,N)   # number of samples
# input time series
x = pd.DataFrame({'X':np.sin(2*3.14*f*n/Fs)})
y = pd.DataFrame({'Y':np.sin(2*3.14*2*f*n/Fs)})
'''
```

```
"""OR"""
""" Import signal from a .csv file """
filename = 'data_examples/2Persons_Monovariate_Continuous_data_2.csv'
x = ExtractSignalFromCSV(filename, columns = ['x'], unit = 's')
y = ExtractSignalFromCSV(filename, columns = ['y'], unit = 's')

# Resample and Interpolate data to have constant frequency
x = ResampleAndInterpolate(x, rule='500ms', limit=5)
y = ResampleAndInterpolate(y, rule='500ms', limit=5)

'''
"""OR"""
""" Import signal from a .mat file """
filename = 'data_examples/data_example_MAT.mat'
x = ExtractSignalFromMAT(filename, columns_index =[0,2], columns_wanted_names=['Time', 'GlobalBodyActivity0'])
y = ExtractSignalFromMAT(filename, columns_index =[10], columns_wanted_names=['GlobalBodyActivity1'])
'''

""" Plot input signals """
n = [float(i)/2 for i in range(x.size)] # create x axis values
plt.ion()
fig = plt.figure()
ax = fig.add_subplot(111)
ax.grid(True)
ax.set_xlabel('Samples')
ax.set_title('Input signals')
ax.plot(n, x, label=x.columns[0])
ax.plot(n, y, label=y.columns[0])
plt.legend(bbox_transform=plt.gcf().transFigure)

""" Define class attributes of the wanted method """
tau_max = 5 * 2      # the maximum lag at which correlation should be computed. It is in the range [0; (lx+ly-1)/2] (in samples)
window = 5 * 10      # length of the windowed signals (in samples)
window_inc = 5 * 2   # amount of time elapsed between two windows (in samples)
tau_inc= 1           # amount of time elapsed between two cross-correlation (in samples)
plot = True          # if True the plot of correlation function is returned. Default: False
ele_per_sec = 5      # number of element in one second

""" Instanciate the class with its attributes """
print("\n")
try :
```

```
    corr = WindowCrossCorrelation.WindowCrossCorrelation(tau_max, window, window_inc, tau_inc, plot, ele_per_sec)
except TypeError, err :
    print("TypeError in WindowCrossCorrelation constructor : \n" + str(err))
    sys.exit(-1)
except ValueError, err :
    print("ValueError in WindowCrossCorrelation constructor : \n" + str(err))
    sys.exit(-1)
except Exception, e :
    print("Exception in WindowCrossCorrelation constructor : \n" + str(e))
    sys.exit(-1)

print("An instance the class is now created with the following parameters:\n" +
      "tau max = " + str(tau_max) + "\n" +
      "window length = " + str(window) + "\n" +
      "window increment = " + str(window_inc) + "\n" +
      "tau increment = " + str(window_inc) + "\n" +
      "number of element per second = " + str(ele_per_sec) + "\n" +
      "plot result = " + str(plot))

""" Compute the method and get the result """
print("\n")
print("Computing...")

try :
    cross_corr = corr.compute(x,y)
except TypeError, err :
    print("TypeError in WindowCrossCorrelation computation : \n" + str(err))
    sys.exit(-1)
except ValueError, err :
    print("ValueError in WindowCrossCorrelation computation : \n" + str(err))
    sys.exit(-1)
except Exception, e :
    print("Exception in WindowCrossCorrelation computation : \n" + str(e))
    sys.exit(-1)

""" Display result """
print("\n")
print("********************************** \n")
print('Window Cross correlation result :')
print("********************************** \n")
print(cross_corr)
```

```
raw_input("Push ENTER key to exit.")
```

4.2.6 Window Cross Correlation with PeakPicking example

(Source code)

```
"""
PeakPicking example :
Computes peak picking algorithm with window cross correlation results
"""

""" Import common python packages """
import sys
import os
import numpy as np          # Mathematical package
import pandas as pd         # Time serie package
import matplotlib.pyplot as plt # Plotting package
sys.path.insert(0, '../src/')   # To be able to import from parent directory

print("\n")
print("****************************************************************************")
print("This script computes the peak picking selection of a cross correlation matrix \n")
print("****************************************************************************")

""" Import wanted modules with every parent packages """
import DataFrom2Persons.Monovariate.Continuous.Linear.WindowCrossCorrelation as WindowCrossCorrelation
import DataFrom2Persons.Monovariate.Continuous.Linear.PeakPicking as PeakPicking

""" Import Utils modules """
from utils.ExtractSignal import ExtractSignalFromCSV
from utils.ExtractSignal import ExtractSignalFromMAT
from utils.ResampleAndInterpolate import ResampleAndInterpolate

'''
""" Define signals in pd.dataFrame format """
# preparing the input signals
N = 20              # number of samples
f = 1.0              # sinewave frequency (Hz)
Fs = 200             # sampling frequency (Hz)
n = np.arange(0,N)   # number of samples
```

```
# input signals
x = pd.DataFrame({'X':np.sin(2*3.14*f*n/Fs)})
y = pd.DataFrame({'Y':np.sin(2*3.14*2*f*n/Fs)})
'''

"""OR"""
""" Import signals from a .csv file """
filename = 'data_examples/2Persons_Monovariate_Continuous_data_2.csv'
x = ExtractSignalFromCSV(filename, columns = ['x'], unit = 's')
y = ExtractSignalFromCSV(filename, columns = ['y'], unit = 's')

# Resample and Interpolate data to have constant frequency
x = ResampleAndInterpolate(x, rule='500ms', limit=5)
y = ResampleAndInterpolate(y, rule='500ms', limit=5)

'''
"""OR"""
""" Import signals from a .mat file """
filename = 'data_examples/data_example_MAT.mat'
x = ExtractSignalFromMAT(filename, columns_index =[0,2], columns_wanted_names=['Time', 'GlobalBodyActivity0'])
y = ExtractSignalFromMAT(filename, columns_index =[10], columns_wanted_names=['GlobalBodyActivity1'])
'''

""" Plot input signals """
n = [float(i)/2 for i in range(x.size)] # create x axis values
plt.ion()
fig = plt.figure()
ax = fig.add_subplot(111)
ax.grid(True)
ax.set_xlabel('Samples')
ax.set_title('Input signals')
ax.plot(n, x, label=x.columns[0])
ax.plot(n, y, label=y.columns[0])
plt.legend(bbox_transform=plt.gcf().transFigure)

""" Define class attributes of the wanted method """
tau_max = 5 * 2     # the maximum lag at which correlation should be computed. It is in the range [0; (lx+ly-1)/2] (in samples)
window = 5 * 10     # length of the windowed signals (in samples)
window_inc = 5 * 2  # amount of time elapsed between two windows (in samples)
tau_inc= 1          # amount of time elapsed between two cross-correlation (in samples)
plot = True         # if True the plot of correlation function is returned. Default: False
ele_per_sec = 5     # number of element in one second
```

```
""" Instanciate the class with its attributes """
try :
    corr = WindowCrossCorrelation.WindowCrossCorrelation(tau_max, window, window_inc, tau_inc, plot, ele_per_sec)
except TypeError, err :
    print("TypeError in WindowCrossCorrelation constructor : \n" + str(err))
    sys.exit(-1)
except ValueError, err :
    print("ValueError in WindowCrossCorrelation constructor : \n" + str(err))
    sys.exit(-1)
except Exception, e :
    print("Exception in WindowCrossCorrelation constructor : \n" + str(e))
    sys.exit(-1)

""" Compute the method and get the result """
try :
    cross_corr = corr.compute(x,y)
except TypeError, err :
    print("TypeError in WindowCrossCorrelation computation : \n" + str(err))
    sys.exit(-1)
except ValueError, err :
    print("ValueError in WindowCrossCorrelation computation : \n" + str(err))
    sys.exit(-1)
except Exception, e :
    print("Exception in WindowCrossCorrelation computation : \n" + str(e))
    sys.exit(-1)

""" Define class attributes of the wanted method """
tau_max = 10        # the maximum lag at which correlation should be computed. It is in the range [0; (lx+ly-1)/2] (in samples)
tau_inc= 1          # amount of time elapsed between two cross-correlation (in samples)
threshold = 0.5     # minimal correlation magnitude acceptable for a peak (between -1 and 1)
lookahead = 2       # distance to look ahead from a peak candidate to determine if it is the actual peak. Default: 200
delta = 0           # this specifies a minimum difference between a peak and the following points, before a peak may be considered a p
ele_per_sec = 2     # number of element in one second
plot = True         #i f True the plot of peakpicking function is returned. Default: False
plot_on_mat = True  # if True the plot of peakpicking + correlation matrix function is returned. Default: False
sorted_peak = True  # if True the peaks found will be organized by type of Lag and Magnitude (positive or negative). Default: False

""" Instanciate the class with its attributes """
try :
    peak = PeakPicking.PeakPicking(cross_corr, tau_max, tau_inc, threshold, lookahead, delta, ele_per_sec, plot, plot_on_mat, sorted_p
except TypeError, err :
```

```
    print("TypeError in PeakPicking constructor : \n" + str(err))
    sys.exit(-1)
except ValueError, err :
    print("ValueError in PeakPicking constructor : \n" + str(err))
    sys.exit(-1)
except Exception, e :
    print("Exception in PeakPicking constructor : \n" + str(e))
    sys.exit(-1)

""" Compute the method and get the result """
try :
    sorted_peaks = peak.compute()
except TypeError, err :
    print("TypeError in PeakPicking computation : \n" + str(err))
    sys.exit(-1)
except ValueError, err :
    print("ValueError in PeakPicking computation : \n" + str(err))
    sys.exit(-1)
except Exception, e :
    print("Exception in PeakPicking computation : \n" + str(e))
    sys.exit(-1)

""" Display result """
print("\n")
print("******************** \n")
print('Peak Picking result : ')
print("******************** \n")
print(sorted_peaks)

raw_input("Push ENTER key to exit.")
```

4.3 DataFrom2Persons Monovariate Continuous NonLinear module

4.3.1 Nonlinear Correlation example

(Source code)

```
"""
Nonlinear Correlation example :
```

```
It computes the nonparametric nonlinear regression coefficient h2 describing the dependency
between two signals (in DataFrame format) x and y in a most general way
"""

""" Import common python packages """
import sys
import os
import numpy as np          # Mathematical package
import pandas as pd         # Time serie package
import matplotlib.pyplot as plt # Plotting package
sys.path.insert(0, '../src/')   # To be able to import packages from parent directory

print("\n")
print("**************************************************************")
print("This script computes the nonlinear correlation coefficient \n"+
      "of two continouos monovariate signals \n")
print("**************************************************************")

""" Import wanted module with every parent packages """
import DataFrom2Persons.Monovariate.Continuous.Nonlinear.NonlinearCorr as NonlinearCorr
from utils.ExtractSignal import ExtractSignalFromCSV
from DataFrom2Persons.Monovariate.Continuous.Linear import Correlation



""" Define signals in pd.dataFrame format """

#Define parameters
N=1000                     # number of samples
t=np.linspace(0,4*np.pi,N) # number of samples


#Create signals
x=pd.DataFrame({'X':3.0*np.sin(t+0.0001)}, np.arange(0,N))
y=x**2

'''
"""OR"""
""" Import signals from a .csv file """
#Data from files
filename = 'data_examples/2Persons_Monovariate_Continuous_data.csv'
```

```
x = ExtractSignalFromCSV(filename, columns = ['x1'])
y = ExtractSignalFromCSV(filename, columns = ['x2'])
'''


""" Plot input signals"""

plt.ion()
f, axarr = plt.subplots(2, sharex=True)
axarr[0].set_title('Input signals')
axarr[0].set_xlabel('Samples')
axarr[1].set_xlabel('Samples')
axarr[0].plot(t, x, label="x")
axarr[1].plot(t, y, label="y", color='r')
axarr[0].legend(loc='best')
axarr[1].legend(loc='best')



""" Define class attributes of the wanted method """
nbins=100   # number of bins in which the time series is divided into


""" Instanciate the class with its attributes """
print("\n")

try :
    c = NonlinearCorr.NonlinearCorr(nbins)
except TypeError, err :
    print("TypeError in NonlinearCorr constructor : \n" + str(err))
    sys.exit(-1)
except ValueError, err :
    print("ValueError in NonlinearCorr constructor : \n" + str(err))
    sys.exit(-1)
except Exception, e :
    print("Exception in NonlinearCorr constructor : \n" + str(e))
    sys.exit(-1)


print("An instance the class is now created with the following parameters:\n" +
      "number of bin = " + str(nbins))
```

```
""" Compute the method and get the result """
print("\n")
print("Computing...")

try :
    res = c.compute(x,y)
except TypeError, err :
    print("TypeError in NonlinearCorr computation : \n" + str(err))
    sys.exit(-1)
except ValueError, err :
    print("ValueError in NonlinearCorr computation : \n" + str(err))
    sys.exit(-1)
except Exception, e :
    print("Exception in NonlinearCorr computation : \n" + str(e))
    sys.exit(-1)

""" Display result """
print("\n")
print("**************************************** \n")
print('NonlinearCorr complete result :\n')
print("****************************************\n")
print(res['h2 coefficient'])

""" Computing the linear correlation coeffcient"""
print("\n")
print ("Computing the linear correlation coefficient:")
print("\n")
""" Define class attributes of the wanted method """

tau_max = 999                    # the maximum lag at which correlation should be computed (in samples)
plot=False                       # plot of the correlation fucntion
standardization = True           # standardization of the time series to mean 0 and variance 1
corr_tau_max = False             # return of the maximum of correlation and its lag
corr_coeff = True                # computation of the correlation coefficient (Pearson's version)
scale= False                     # scale factor to have correlaton in [-1,1]

""" Instanciate the class with its attributes """
print("\n")

try :
    c=Correlation.Correlation(tau_max, plot, standardization, corr_tau_max, corr_coeff, scale)
except TypeError, err :
```

```
    print("TypeError in Correlation constructor : \n" + str(err))
    sys.exit(-1)
except ValueError, err :
    print("ValueError in Correlation constructor : \n" + str(err))
    sys.exit(-1)
except Exception, e :
    print("Exception in Correlation constructor : \n" + str(e))
    sys.exit(-1)

print("An instance the class is now created with the following parameters:\n" +
      "tau max = " + str(tau_max) + "\n" +
      "plot = " + str(plot) + "\n" +
      "standardization= " + str(standardization) + "\n" +
      "corr_tau_max = " + str(corr_tau_max) + "\n" +
      "corr_coeff =" + str(corr_coeff) +"\n" +
      "scale =" + str(scale))

""" Compute the method and get the result """
print("\n")
print("Computing...")

try :
    res= c.compute(x, y)
except TypeError, err :
    print("TypeError in Correlation computation : \n" + str(err))
    sys.exit(-1)
except ValueError, err :
    print("ValueError in Correlation computation : \n" + str(err))
    sys.exit(-1)
except Exception, e :
    print("Exception in Correlation computation : \n" + str(e))
    sys.exit(-1)

""" Display result """
print("\n")
print("**************************************** \n")
print('Correlation complete result :\n')
print("****************************************\n")
print("Pearson's correlation coefficient %f:" %(res['corr_coeff']))
print("\n")

print ("As expected the two coefficient provide different results, \n" +
```

```
        "that is high value of nonlinear correlation coefficient and \n" +
        "low value of linear correlation coefficient.")


raw_input("Push ENTER key to exit.")
plt.close("all")
```

4.4 DataFromManyPersons Monovariate Continuous Linear module

4.4.1 PartialCoherence example

(Source code)

```
"""
PartialCoherence example :
Compute Partial Coherence for multiple monovariate signals (orgainized as a list).
"""

""" Import common python packages """
import sys
import os
import numpy as np              # Mathematical package
import pandas as pd             # Time serie package
import matplotlib.pyplot as plt # Plotting package
sys.path.insert(0, '../src')       # To be able to import from parent directory

print("\n")
print("**********************************************************************")
print("This script computes Partial Coherence for multiple monovariate signals \n" +
      "(organized as a list) \n")
print("**********************************************************************")

""" Import wanted module with every parent packages """
import DataFromManyPersons.Monovariate.Continuous.Linear.PartialCoherence as PartialCoherence
from utils.ExtractSignal import ExtractSignalFromCSV
from utils.ExtractSignal import ExtractSignalFromMAT

'''
""" Define signals in pd.dataFrame format """
```

```
# preparing the input time series
N = 20            # number of samples
f = 1.0            # sinewave frequency (Hz)
Fs = 200           # sampling frequency (Hz)
n = np.arange(0,N)  # number of samples
# input time series
x = pd.DataFrame({'X':np.sin(2*3.14*f*n/Fs)})
y = pd.DataFrame({'Y':np.sin(2*3.14*2*f*n/Fs)})
'''

"""OR"""
""" Import signal from a .csv file """
filename = 'data_examples/2Persons_Multivariate_Continous_data.csv'
x = ExtractSignalFromCSV(filename, columns = ['Upper body mq'])
y = ExtractSignalFromCSV(filename, columns = ['Upper body mq.1'])
z = ExtractSignalFromCSV(filename, columns = ['Left Hand mq'])
a = ExtractSignalFromCSV(filename, columns = ['Left Hand mq.1'])

'''
"""OR"""
""" Import signal from a .mat file """
filename = 'data_examples/data_example_MAT.mat'
x = ExtractSignalFromMAT(filename, columns_index =[0,2], columns_wanted_names=['Time', 'GlobalBodyActivity0'])
y = ExtractSignalFromMAT(filename, columns_index =[10], columns_wanted_names=['GlobalBodyActivity1'])
'''

signals = [x,y,z,a]

N = signals[0].shape[0]
n = np.arange(0,N)

""" Plot input signals """
plt.ion()
fig = plt.figure()
ax = fig.add_subplot(111)
ax.grid(True)
ax.set_xlabel('Samples')
ax.set_title('Input signals')
for i in range(len(signals)) :
    ax.plot(n, signals[i].iloc[:,0], label=signals[i].columns[0])
plt.legend(bbox_transform=plt.gcf().transFigure)
```

```
""" Define class attributes of the wanted method """
fs = 1.0         # sampling frequency of the input DataFrame in Hz
NFFT = 256       # length of each epoch
detrend = 1      # specifies which kind of detrending should be computed on data. 0 stands for constant detrending; 1 stands for linear
noverlap = 0     # number of points to overlap between epochs

""" Instanciate the class with its attributes """
print("\n")
try :
    pc = PartialCoherence.PartialCoherence(fs, NFFT, detrend, noverlap)
except TypeError, err :
    print("TypeError in PartialCoherence constructor : \n" + str(err))
    sys.exit(-1)
except ValueError, err :
    print("ValueError in PartialCoherence constructor : \n" + str(err))
    sys.exit(-1)
except Exception, e :
    print("Exception in PartialCoherence constructor : \n" + str(e))
    sys.exit(-1)

print("An instance the class is now created with the following parameters:\n" +
      "fs = " + str(fs) + "\n" +
      "NFFT = " + str(NFFT) + "\n" +
      "detrend = " + str(detrend) + "\n" +
      "noverlap = " + str(noverlap) + "\n"
      )

""" Compute the method and get the result """
print("\n")
print("Computing...")

try :
    partial_coherence = pc.compute(*signals)
except TypeError, err :
    print("TypeError in PartialCoherence computation : \n" + str(err))
    sys.exit(-1)
except ValueError, err :
    print("ValueError in PartialCoherence computation : \n" + str(err))
    sys.exit(-1)
except Exception, e :
    print("Exception in PartialCoherence computation : \n" + str(e))
    sys.exit(-1)
```

```
""" Display result """
print("\n")
print('PartialCoherence computed for theses signals indexes :')
print("\n")

for i in partial_coherence.keys():
    print(str(sorted(partial_coherence[i].keys())) + " given " + str(i)+ ", ")
    print("\n")

raw_input("Push ENTER key to exit.")
```

4.4.2 S_Estimator example

(Source code)

```
"""
S_Estimator example :
Compute Synchronization Indexes for multiple monovariate signals (orgainized as a list).
"""

""" Import common python packages """
import sys
import os
import numpy as np              # Mathematical package
import pandas as pd             # Time serie package
import matplotlib.pyplot as plt # Plotting package
sys.path.insert(0, '../src/')       # To be able to import from parent directory

print("\n")
print("**********************************************************************************")
print("This script computes Synchronization indexes for multiple monovariate signals \n" +
      "(orgainized as a list) \n")
print("**********************************************************************************")

""" Import wanted module with every parent packages """
import DataFromManyPersons.Monovariate.Continuous.Linear.S_Estimator as S_Estimator
from utils.ExtractSignal import ExtractSignalFromCSV
from utils.ExtractSignal import ExtractSignalFromMAT
from utils.Standardize import Standardize
```

```
'''
""" Define signals in pd.dataFrame format """
# preparing the input signals
N = 20              # number of samples
f = 1.0              # sinewave frequency (Hz)
Fs = 200             # sampling frequency (Hz)
n = np.arange(0,N)   # number of samples
# input signals
x = pd.DataFrame({'X':np.sin(2*3.14*f*n/Fs)})
y = pd.DataFrame({'Y':np.sin(2*3.14*2*f*n/Fs)})
'''

"""OR"""
""" Import signals from a .csv file """
filename = 'data_examples/2Persons_Multivariate_Continous_data.csv'
x = ExtractSignalFromCSV(filename, columns = ['Upper body mq'])
y = ExtractSignalFromCSV(filename, columns = ['Upper body mq.1'])

'''
"""OR"""
""" Import signals from a .mat file """
filename = 'data_examples/data_example_MAT.mat'
x = ExtractSignalFromMAT(filename, columns_index =[0,2], columns_wanted_names=['Time', 'GlobalBodyActivity0'])
y = ExtractSignalFromMAT(filename, columns_index =[10], columns_wanted_names=['GlobalBodyActivity1'])
'''

signals = [x,y]

N = signals[0].shape[0]
n = np.arange(0,N)

""" Plot standardized input signals """
Signals = signals
plt.ion()
for i in range(len(signals)) :
    Signals[i] = Standardize(signals[i])
fig = plt.figure()
ax = fig.add_subplot(111)
ax.grid(True)
ax.set_xlabel('Samples')
ax.set_title('Standardized signals')
for i in range(len(Signals)) :
```

```
    ax.plot(n, Signals[i].iloc[:,0], label=Signals[i].columns[0])
plt.legend(bbox_transform=plt.gcf().transFigure)

""" Define class attributes of the wanted method """
surr_nb_iter = 100
plot_surrogate = True

""" Instanciate the class with its attributes """
print("\n")
try :
    s_estimator = S_Estimator.S_Estimator(surr_nb_iter, plot_surrogate)
except TypeError, err :
    print("TypeError in S_Estimator constructor : \n" + str(err))
    sys.exit(-1)
except ValueError, err :
    print("ValueError in S_Estimator constructor : \n" + str(err))
    sys.exit(-1)
except Exception, e :
    print("Exception in S_Estimator constructor : \n" + str(e))
    sys.exit(-1)

print("An instance the class is now created with the following parameters:\n" +
      "surr_nb_iter = " + str(surr_nb_iter) + "\n"
      )

""" Compute the method and get the result """
print("\n")
print("Computing...")

try :
    estimators = s_estimator.compute(*signals)
except TypeError, err :
    print("TypeError in S_Estimator computation : \n" + str(err))
    sys.exit(-1)
except ValueError, err :
    print("ValueError in S_Estimator computation : \n" + str(err))
    sys.exit(-1)
except Exception, e :
    print("Exception in S_Estimator computation : \n" + str(e))
    sys.exit(-1)

""" Display result """
```

```
print("\n")
print('S_Estimator result :')
print("\n")

for i in estimators.keys():
    print(i + " : " + str(estimators[i]))
print("\n")

raw_input("Push ENTER key to exit.")
```

4.5 DataFromManyPersons Multivariate Continuous Linear module

4.5.1 Multivariate Granger Causality example

(Source code)

CHAPTER 5

Publications

Please cite this paper if you are using SyncPy for your own research :

Giovanna Varni, Marie Avril, Adem Usta, Mohamed Chetouani. *SyncPy - A unified analytic library for synchrony.*

Accepted at First International Workshop on Modeling INTEPERsonal SynchrONy @ICMI 2015 Conference.