SyncPy Documentation

Release 42

Giovanna Varni, Marie Avril

December 07, 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 :

- Python (v2.7)
- Matplotlib (v1.4.3)
- NetworkX (v1.10)
- Numpy and Scipy (v1.9)
- Pandas (v 0.16.0)
- Statsmodels and Patsy (v0.6.1)

To correctly use the interface, please download also this library:

- pyQt (v4.11.3)

Version numbers are just an indication to precise which versions have been used to test the current release.

2.2 Sources

The full source code is available in GitHub platform, please download the last version :

https://github.com/syncpy/SyncPy

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 (`Univariate` 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.

Univariate 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.Univariate.Categorical.Linear.BooleanTurnsActivity.**BooleanTurnsActivity**(*max_latency*, *min_pause_duration*, *ele_per_sec=1*, *duration=-1*)

It computes data turns statistics between two boolean univariate 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.Univariate.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 univariate 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_result(*result*)
It plots the resulting Q and q when atype is set to 'tsl' or 'asl'

param result Event Sync result from compute()

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.Univariate.Continuous.Linear.Coherence.**Coherence**(*fs=1.0*, *NFFT=256*, *detrend=0*, *noverlap=0*, *plot=False*)
It computes the linear correlation between two univariate 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_result (*result*)

It plots the coherence function

Parameters **result** (*dict*) – coherence and frequencies from compute()

Returns plt.figure – figure plot

Correlation module *Module author: Giovanna Varni*

class DataFrom2Persons.Univariate.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 univariate 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_result (*result*)

It plots the correlation function in the range specified.

Parameters **result** (*dict*) – Correlation result from compute()

Returns plt.figure – figure plot

WindowCrossCorrelation module *Module author: Marie Avril*

class DataFrom2Persons.Univariate.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 two univariate 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 dict – windowed cross correlation dictionary with (2 * tau_max + 1)/tau_inc rows and (length(x) - window - win_inc)/ win_inc columns

plot_result(*result*)

It plots the window cross correlation matrix

Parameters **result** (*dict*) – window cross correlation dictionary from compute()

Returns plt.figure – figure plot

GrangerCausality module *Module author: Adem Usta*

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

It computes a Granger causality test between two univariate 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.

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-value, p-value, ratio(G-magnitude), optimal_lag, predicted_signal_restricted, predicted_signal_unrestricted.

plot_result(*result*)

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

Parameters **result** (*dict*) – Granger Causality result from compute()

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

SpectralGrangerCausality module *Module author: Adem Usta*

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

It computes a Granger causality test between two univariate 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_result(*result*)

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

Parameters **result** (*dict*) – Spectral Granger Causality result from compute()

Returns plt.figure – A figure that contains the plot

MachineLearning package

Nonlinear package

NonlinearCorr module *Module author: Giovanna Varni*

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

It computes the nonparametric nonlinear regression coefficient h2 describing the dependency between two univariate 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.Univariate.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** (*int*) –

  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

- **var_resc** (*bool*) – Boolean value indicating if the input signals should be rescaled at unitary variance. Default: False
- **noise** (*bool*) – 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

compute (*x*, *y*)

It computes Mutual Information.

Parameters

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

Returns dict – Mutual Information

WindowMutualInformation module *Module author: Marie Avril*

class DataFrom2Persons.Univariate.Continuous.Nonlinear.WindowMutualInformation.**WindowMutualInformation**(*n_neighbours*, *my_type=1*, *var_resc=True*, *noise=True*, *tau_max=0*, *window=0*, *win_inc=1*, *tau_inc=1*, *plot=False*)

It computes Windowed 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** (*int*) –

  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
- **var_resc** (*bool*) – Boolean value indicating if the input signals should be rescaled at unitary variance. Default: False
- **noise** (*bool*) – 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
- **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

compute (*x*, *y*)
It computes Mutual Information.

Parameters

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

Returns dict – Windowed Mutual Information

plot_result (*result*)
It plots the window mutual information matrix

Parameters **result** (*dict*) – Windowed Mutual Information from compute()

Returns plt.figure – figure plot

PhaseSynchro_Strobo module *Module author: Giovanna Varni*

class DataFrom2Persons.Univariate.Continuous.Nonlinear.PhaseSynchro_Strobo.**PhaseSynchro_Strobo**(*n=1*, *m=1*, *nbins_mode='auto'*, *nbins=10*)

It computes n:m synchronization index lambda_nm by using a stroboscopic approach between two univariate signals x and y (in pandas DataFrame format).

Reference : M. Rosenblum, A. Pikovsky, J. Kurths, C. Schafer and P. A. Tass. Phase synchronizatio:from theory to practice. In Handbook of Biological Physics, Elsiever Science, Series Editor A.J. Hoff, Vol. , Neuro-Informatics, Editors: F. Moss and S. Gielen, Chapter 9.

Parameters

- **n** (*int*) – it is the integer for the order of synchronization
- **m** (*int*) – it is the integer for the order of synchronization
- **nbins_mode** (*str*) – It can be: 1. 'auto':the number of bis wil be automatically estimated 2. 'man': the number of bins (nbins) will take the value expressed in nbins parameter
- **nbins** (*int*) – it is the number of bins to be used to build phase distribution

compute(*x*, *y*)

It computes the synchronization index lambda_nm

Parameters

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

Returns dict – lambda_mn index

PhaseSynchro_Entropy module *Module author: Giovanna Varni*

class DataFrom2Persons.Univariate.Continuous.Nonlinear.PhaseSynchro_Entropy.**PhaseSynchro_Entropy**(*n=1*, *m=1*, *nbins_mode='auto'*, *nbins=10*, *dist_cyc_rel_phase=False*)

It computes n:m synchronization index rho_nm by using a Shannon entropy based approach between two univariate signals x and y (in pandas DataFrame format). Rho_nm ranges in [0,1] where 0 means no synchronization at all and 1 means perfect synchronization.

Reference : M. Rosenblum, A. Pikovsky, J. Kurths, C. Schafer and P. A. Tass. Phase synchronizatio:from theory to practice. In Handbook of Biological Physics, Elsiever Science, Series Editor A.J. Hoff, Vol. , Neuro-Informatics, Editors: F. Moss and S. Gielen, Chapter 9.

Parameters

- **n** (*int*) – it is the integer for the order of synchronization
- **m** (*int*) – it is the integer for the order of synchronization

- **nbins_mode** (*str*) – It can be: 1. 'auto':the number of bis wil be automatically estimated 2. 'man': the number of bins (nbins) will take the value expressed in nbins parameter
- **nbins** (*bool*) – it is the number of bins to be used to build phase distribution
- **dist_cyc_rel_phase** – if True the plot of the distribution of the cyclic relative phase is returned. Default: False

compute (*x*, *y*)
It computes the synchronization index lambda_nm

Parameters

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

Returns dict – rho_mn index

plot_dist_cyc_phase (*dist_psi_nm_hist*, *dist_psi_nm_bins*)
It plots the distribution of the cyclic relative phase.

Parameters

- **dist_psi_nm_hist** (*np.array*) – the values of the histogram
- **dist_psi_nm_bins** – the bin-edges

Returns plt.figure – figure plot

PhaseSynchro_Fourier module *Module author: Giovanna Varni*

class DataFrom2Persons.Univariate.Continuous.Nonlinear.PhaseSynchro_Fourier.**PhaseSynchro_Fourier** (*n=1*, *m=1*)
It computes n:m synchronization index gamma2_nm as the intensity of the first Fourier mode of the cyclic relative phase of two univariate signals x and y (in pandas DataFrame format). Gamma2_nm ranges in [0,1] where 0 means no synchronization at all and 1 means perfect synchronization.

Reference : M. Rosenblum, A. Pikovsky, J. Kurths, C. Schafer and P. A. Tass. Phase synchronizatio:from theory to practice. In Handbook of Biological Physics, Elsiever Science, Series Editor A.J. Hoff, Vol. , Neuro-Informatics, Editors: F. Moss and S. Gielen, Chapter 9.

Parameters

- **n** (*int*) – it is the integer for the order of synchronization
- **m** (*int*) – it is the integer for the order of synchronization

compute (*x*, *y*)
It computes the synchronization index lambda_nm

Parameters

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

Returns dict – gamma2_mn index

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

GSI module *Module author: Giovanna Varni*

class DataFrom2Persons.Multivariate.Continuous.Nonlinear.GSI.**GSI**(*m*, *t*, *rr*)

It computes the generalised synchronization index (GSI) between two uni/multi-variate signals x and y(in DataFrame format). GSI ranges in [0,1] where 0 means no synchronization and 1 perfect generalized synchronization.

Parameters

- **m** (*int*) – embedding dimension
- **t** (*float*) – embedding delay
- **rr** – recurrence rate

compute(*x*, *y*)

It computes GSI

Parameters

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

Returns dict – gsi

3.2.2 DataFromManyPersons package

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

Univariate 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 and Giovanna Varni*

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

It computes Conditional Granger Causality among signals. It assesses whether the interaction between two signals is direct or is mediated by other signals and wheter the causal influence is due to differential time delays in the driving inputs. The algorithm computes 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 the other signals. At the end of the test, a graph is shown to see the true links among the signals.

Reference :

•Xiaotong Wen, 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_result(*result*)
It plots the final result of the module : a graphic that shows the links between the signals.

Parameters **result** (*dict*) – Conditional Granger Causality result from compute()

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

MultipleGrangerCausality module *Module author: Adem Usta*

class DataFromManyPersons.Univariate.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_result(*result*)

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

Parameters **result** (*dict*) – Granger Causality result from compute()

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

Omega_Complexity module *Module author: Giovanna Varni*

class DataFromManyPersons.Univariate.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.Univariate.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.Univariate.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_result (*result*)

Plot surrogates signals

Parameters **result** (*dict*) – S-estimator result from compute() (only 'surrogate_signal' used)

Returns plt.figure – figure plot

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

Crqa module

Module author: Giovanna Varni

utils.Crqa.**Crqa**(*x*, *y*, *m*, *t*, *e*, *distance*, *standardization*, *window*, *window_size*, *step*, *lmin*, *thw*)

It computes the following (cross)recurrence measures from the (cross)recurrence plot of two uni/multi-variate signals x and y (in pandas DataFrame format): Recurrence Rate (RR), Determinism (DET), Average Diagonal Line Length (L), Maximum Diagonal Line Length (L_max), Entropy (ENT).

Parameters

- **x** (*pd.DataFrame*) – first input signal
- **y** (*pd.DataFrame*) – second input signal
- **m** (*int*) – embedding dimension
- **t** (*int*) – embedding delay
- **eps** (*float*) – threshold for recurrence
- **distance** (*str*) – It specifies which distance method is used. It can assumes the following values:
  1. 'euclidean';
  2. 'maximum';
  3. 'manhattan'
- **standardization** (*bool*) – if True data are nomalize to zero mean and unitary variance
- **window** (*bool*) – second input signal
- **window_size** (*int*) – embedding dimension
- **step** (*int*) – embedding delay
- **lmin** (*int*) – threshold
- **thw** (*int*) – distance method

utils.Crqa.**DET**(*crp_matrix*, *hist_P*, *rr*, *lmin*)

It computes the Determinism (DET)

utils.Crqa.**Entr**(*crp_matrix*, *hist_P*, *lmin*)

It computes the Entropy (ENTR)

utils.Crqa.**L**(*crp_matrix*, *hist_P*, *lmin*)

It computes the Average Diagonal Line Length (L)

utils.Crqa.**L_max**(*crp_matrix*, *hist_P*, *lmin*)
It computes the Maximum Diagonal Line Length (L)

utils.Crqa.**RR**(*crp_matrix*, *thw*)
It computes the Recurrence Rate (RR)

utils.Crqa.**length_ones_seq**(*diag_line*)
It computes the length of a sequence of ones

Crqa_diag module

Module author: Giovanna Varni

utils.Crqa_diag.**Crqa_diag**(*x*, *y*, *m*, *t*, *e*, *distance*, *standardization*, *window_size*, *lmin*)
It computes the following diagonalwise (cross) recurrence measures from the (cross)recurrence plot of two uni/multi-variate signals x and y (in pandas DataFrame format): Recurrence Rate (RR), Determinism (DET), Average Diagonal Line Length (L).

Reference :

- 14.Marwan, M. Carmen Romano, M. Thiel and J. Kurths. "Recurrence plots for the analysis of complex systems". Physics Reports 438(5), 2007.

Parameters

- **x** (*pd.DataFrame*) – first input signal
- **y** (*pd.DataFrame*) – second input signal
- **m** (*int*) – embedding dimension
- **t** (*int*) – embedding delay
- **eps** (*float*) – threshold for recurrence
- **distance** (*str*) – It specifies which distance method is used. It can assumes the following values:
  1. 'euclidean';
  2. 'maximum';
  3. 'manhattan'
- **standardization** (*bool*) – if True data are nomalize to zero mean and unitary variance
- **window_size** (*int*) – it is the size of the window around the main diagonal over which the measures will be computed
- **lmin** (*int*) – it is the minimum value of the diagonal length line will be used when measures will be computed

utils.Crqa_diag.**length_ones_seq**(*diag_line*)
It computes the length of a sequence of ones

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 – Univariate 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. If the signal is constant, the normalization converts it into the 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

PeakPicking module

Module author: Marie Avril

utils.PeakPicking.**PeakPicking**(*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 to a cross-matrix (computed by WindowCrossCorrelation or WindowMutualInformation for example)

Parameters

- **matrix** (*dict*) – cross matrix (from WindowCrossCorrelation or WindowMutualInformation for example)
- **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 magnitude acceptable for a peak. For maxima, compared to threshold, for minima, compared to (-threshold)
- **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

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)

utils.PeakPicking.**PeakPicking_plot**(*result*, *matrix*, *tau_max*, *ele_per_sec=1*, *plot_on_mat=False*)

It plots the peakpicking result. Works only with unsorted results

Parameters

- **result** (*pd.DataFrame*) – result of PeakPicking()
- **matrix** (*dict*) – cross matrix (from WindowCrossCorrelation or WindowMutualInformation for example)
- **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]
- **ele_per_sec** (*int*) – number of elements in one second
- **plot_on_mat** (*bool*) – if True the plot of peakpicking + correlation matrix function is returned. Default: False

Returns plt.figure – figure plot

utils.PeakPicking.**PeakPicking_sortResult**(*result*)

It organizes peakPicking result in order to compute statistics

Parameters **result** (*pd.DataFrame*) – result of PeakPicking()

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

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)

Embedding module

Module author: Giovanna Varni

utils.Embedding.**Embedding**(*x*, *m*, *t*)

It embeds the input signal x by using a dimension equal to m and a delay between data equal to t.

Parameters

- **x** (*pd.DataFrame*) – first input signal
- **m** (*int*) – the embedding dimension
- **t** (*int*) – the embedding delay expressed in samples

Returns pd.DataFrame – the embedded DataFrame

CrossRecurrencePlot module

Module author: Giovanna Varni

utils.CrossRecurrencePlot.**CrossRecurrencePlot**(*x*, *y*, *m*, *t*, *e*, *distance*, *standardization=False*, *plot=False*)

It computes and plots the (cross)recurrence plot of the uni/multivariate input signal(s) x and y (in pandas DataFrame format).

Reference :

- 14.Marwan, M. Carmen Romano, M. Thiel and J. Kurths. "Recurrence plots for the analysis of complex systems". Physics Reports 438(5), 2007.

Parameters

- **x** (*pd.DataFrame*) – first input signal
- **y** (*pd.DataFrame*) – second input signal
- **m** (*int*) – embedding dimension
- **t** (*int*) – embedding delay
- **eps** (*float*) – threshold for recurrence
- **distance** (*str*) – It specifies which distance method is used. It can assumes the following values:
  1. 'euclidean';
  2. 'maximum';
  3. 'manhattan'

4. ‘fixed distance maximum norm’

- **standardization** (*bool*) – if True data are nomalize to zero mean and unitary variance. Default: False
- **plot** – if True the plot of correlation function is returned. Default: False

JointRecurrencePlot module

Module author: Giovanna Varni

`utils.JointRecurrencePlot.`**`JointRecurrencePlot`**(*x*, *y*, *m*, *t*, *e*, *distance*, *standardization=False*, *plot=False*)

It computes and plots the joint recurrence plot of the uni/multivariate input signal(s) x and y (in pandas DataFrame format).

Reference :

- 14.Marwan, M. Carmen Romano, M. Thiel and J. Kurths. “Recurrence plots for the analysis of complex systems”. Physics Reports 438(5), 2007.

Parameters

- **x** (*pd.DataFrame*) – first input signal
- **y** (*pd.DataFrame*) – second input signal
- **m** (*int*) – embedding dimension
- **t** (*int*) – embedding delay
- **eps** (*float*) – threshold for recurrence
- **distance** (*str*) – It specifies which distance method is used. It can assumes the following values:
  1. ‘euclidean’;
  2. ‘maximum’;
  3. ‘manhattan’
  4. ‘fixed distance maximum norm’
- **standardization** (*bool*) – if True data are nomalize to zero mean and unitary variance. Default: False
- **plot** – if True the plot of correlation function is returned. Default: False

3.4 Indices and tables

- *genindex*
- *modindex*
- *search*

CHAPTER 4

SyncPy library Examples

Available source code examples

- DataFrom2Persons Univariate Categorical Linear module
  - Boolean Turn Activity example
- DataFrom2Persons Univariate Categorical Nonlinear module
  - Event Synchronization example
- DataFrom2Persons Univariate 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 Univariate Continuous NonLinear module
  - Nonlinear Correlation example
  - Mutual Information example
  - Window Mutual Information example
  - PhaseSynchro_Strobo example
  - PhaseSynchro_Entropy example
  - PhaseSynchro_Fourier example
- DataFromManyPersons Univariate Continuous Linear module
  - Conditional Granger Causality example
  - Omega complexity example
  - PartialCoherence example
  - S_Estimator example
  - Multiple Granger Causality example
- DataFrom2Persons Multivariate Continuous NonLinear module
  - GSI example

4.1 DataFrom2Persons Univariate Categorical Linear module

4.1.1 Boolean Turn Activity example

(Source code)

```
"""
BooleanTurnsActivity example :
Computes data turns statistics between two boolean univariate signals x and y (in DataFrame format):
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 univariate signals \n")
print("********************************************************************************")

""" Import wanted module with every parent packages """
import DataFrom2Persons.Univariate.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, 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

""" Instantiate 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 of 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 Univariate Categorical Nonlinear module

4.2.1 Event Synchronization example

(Source code)

```
"""
Event Synchronisation example:
```

```
It computes Event Synchornisation (ES) and time delay patterns between two monovariate signals x and y
(in pandas DataFrame format) in which events can be identified using Quian Quiroga's method.
"""

""" 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 compute the synchronisation and time delay pattern between two monovariate\n"+
      "signals (in pandas DataFrame format) showing events.")
print("********************************************************************************************")

""" Import wanted module with every parent packages """
import DataFrom2Persons.Univariate.Categorical.Nonlinear.EventSync as EventSync

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


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

#input signals
print("\n")
print("Generating the input signals...")

N=26

x = pd.DataFrame([0,0,1,0,0,0,0,0,0,1,0,0,0,0,0,0,1,0,0,0,0,0,0,1,0,0],np.arange(0,N))
y = pd.DataFrame([0,0,0,0,1,0,0,0,0,0,1,0,0,0,0,0,1,0,0,0,0,0,1,0,0,0],np.arange(0,N))


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

```
x = ExtractSignalFromCSV(filename, columns = ['0'])
y = ExtractSignalFromCSV(filename, columns = ['1'])


'''
s=np.arange(0,x.shape[0])

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].stem(s,x,label="x")
axarr[1].stem(s, y, label="y")
axarr[0].legend(loc='best')
axarr[1].legend(loc='best')


""" Define class attributes of the wanted method """
atype = 'tot'                       # algorithm to be used to compute Q and q
tau = 1                             # algorithm is used to estimates the delay
lag_tau = 2                         # number of samples will be used as delay
window = 1                          # size of the window to compute Q and q
plot= False                         # plot of Q and q

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

try :
    c=EventSync.EventSync(atype, tau, lag_tau, window, plot)
except TypeError, err :
    print("TypeError in EventSync constructor : \n" + str(err))
    sys.exit(-1)
except ValueError, err :
    print("ValueError in EventSync constructor : \n" + str(err))
    sys.exit(-1)
except Exception, e :
    print("Exception in EventSync constructor : \n" + str(e))
    sys.exit(-1)
```

```
print("An instance the class is now created with the following parameters:\n" +
      "type = " + str(atype) + "\n" +
      "tau = " + str(tau) + "\n" +
      "lag_tau = " + str(lag_tau) + "\n" +
      "window= " + str(window) + "\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 EventSync computation : \n" + str(err))
    sys.exit(-1)
except ValueError, err :
    print("ValueError in EventSync computation : \n" + str(err))
    sys.exit(-1)
except Exception, e :
    print("Exception in EventSync computation : \n" + str(e))
    sys.exit(-1)

""" Display result """
print("\n")
print("************************************ \n")
print('EventSync complete result :')
print("************************************\n")

print("Q:")
print(res['Q'])
print("q:")
print(res['q'])

""" OTHER EXAMPLE WITH TSL TYPE """
print("\n")
print("************************************************ \n")
print("************************************************ \n")

""" Define class attributes of the wanted method """
atype = 'tsl'                       # algorithm to be used to compute Q and q
```

```
tau = 1                             # algorithm is used to estimates the delay
lag_tau = 2                         # number of samples will be used as delay
window = 30                          # size of the window to compute Q and q
plot= True                          # plot of Q and q

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

try :
    c=EventSync.EventSync(atype, tau, lag_tau, window, plot)
except TypeError, err :
    print("TypeError in EventSync constructor : \n" + str(err))
    sys.exit(-1)
except ValueError, err :
    print("ValueError in EventSync constructor : \n" + str(err))
    sys.exit(-1)
except Exception, e :
    print("Exception in EventSync constructor : \n" + str(e))
    sys.exit(-1)


print("An instance the class is now created with the following parameters:\n" +
      "type = " + str(atype) + "\n" +
      "tau = " + str(tau) + "\n" +
      "lag_tau = " + str(lag_tau) + "\n" +
      "window= " + str(window) + "\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 EventSync computation : \n" + str(err))
    sys.exit(-1)
except ValueError, err :
    print("ValueError in EventSync computation : \n" + str(err))
    sys.exit(-1)
except Exception, e :
```

```
    print("Exception in EventSync computation : \n" + str(e))
    sys.exit(-1)

""" Display result """
print("\n")
print("************************************* \n")
print('EventSync complete result :')
print("*************************************\n")

print("Q:")
print(res['Q'])
print("q:")
print(res['q'])

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

4.3 DataFrom2Persons Univariate Continuous Linear module

4.3.1 Coherence example

(Source code)

```
"""
Coherence example :
It computes coherence function between two continuous univariate signals x and y (in pandas DataFrame format).
"""

""" 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 univariate 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.Univariate.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_Univariate_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 = 1     # remove constant detrending
noverlap = 80   # number of points of overlap between epochs
plot = True     # plot of the coherence function

""" Instantiate 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 of 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.")
```

4.3.2 Correlation example

(Source code)

```
"""
Correlation example:
It computes the linear correlation between two continuous univariate signals x and y (in pandas DataFrame format) as a function of the
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 univariate 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.Univariate.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_Univariate_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]

""" Instantiate 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 of 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'])

if corr_tau_max:
    print("Maximum value of the correlation %f and lag (in samples) %d:" %(res['max_corr'],res['t_max']))

if corr_coeff:
    print("Pearson's correlation coefficient %f:" %(res['corr_coeff']))



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

4.3.3 Granger Causality example

(Source code)

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

""" Import common python packages """
import sys
import numpy as np              # Mathematical 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 between two univariate 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.Univariate.Continuous.Linear.GrangerCausality as GC

""" Import signal from a .csv file """
filename = 'data_examples/2Persons_Univariate_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)

""" Plot input signals """
Signals = [x1, x2]
plt.ion()

nrows = len(Signals)
figure, ax = plt.subplots(nrows, sharex=True)
idx = 0
for col in  range(len(Signals)) :
```

```
    ax[idx].grid(True) # Display a grid
    ax[idx].set_title('Input signal : ' + str(Signals[col].columns[0]))
    ax[idx].plot(Signals[col].index, Signals[col].iloc[:,0])
    idx += 1

ax[idx-1].set_xlabel('Time')


""" 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

""" Instantiate 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",criterion,", the optimal number of lag estimated is :", results['optimal_lag'],"\n"
print "F_value =",results['F_value']," with p_value =",results['p_value'],"\n"
print "ratio value=",results['ratio'],"\n"

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

4.3.4 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.Univariate.Continuous.Linear.SpectralGrangerCausality as SGC

""" Import signal from a .csv file """
filename = 'data_examples/1Person_Multivariate_Continuous_data.csv'
print "\nLoading signals from csv files : ", filename,"\n"
x1 = ExtractSignalFromCSV(filename, columns = ['x2'])
x2 = ExtractSignalFromCSV(filename, columns = ['x3'])
""" 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.")
```

4.3.5 Window Cross Correlation example

(Source code)

```
"""
WindowCrossCorrelation example :
Computes the window cross correlation between two continuous univariate signals x and y (in pandas DataFrame format)
"""

""" 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 continuous monovariate signals \n" +
      "expressed as Python Pandas DataFrame.")
print("********************************************************************************")
```

```
""" Import wanted module with every parent packages """
import DataFrom2Persons.Univariate.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_Univariate_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

""" Instantiate 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 of 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.")
```

4.3.6 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.Univariate.Continuous.Linear.WindowCrossCorrelation as WindowCrossCorrelation


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

'''
""" 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_Univariate_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)

""" Get peaks of current result """
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 = False        #if True the plot of peakpicking function is returned. Default: False
plot_on_mat =False  # if True the plot of peakpicking + correlation matrix function is returned. Default: False
sorted_peak = False # if True the peaks found will be organized by type of Lag and Magnitude (positive or negative). Default: False

# Compute peakPicking util method
try :
    peak_res = PeakPicking.PeakPicking(cross_corr, tau_max, tau_inc, threshold, lookahead, delta, ele_per_sec, plot, plot_on_mat, sort
except TypeError, err :
    print("TypeError in PeakPicking method : \n" + str(err))
    sys.exit(-1)
except ValueError, err :
    print("ValueError in PeakPicking method : \n" + str(err))
    sys.exit(-1)
except Exception, e :
    print("Exception in PeakPicking method : \n" + str(e))
    sys.exit(-1)

# Get figure
try :
    peak_figure = PeakPicking.PeakPicking_plot(peak_res, cross_corr, tau_max, ele_per_sec, plot_on_mat = True)
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)

# Get stats
```

```
peak_sorted = PeakPicking.PeakPicking_sortResult(peak_res)# Get statistics

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

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

4.4 DataFrom2Persons Univariate Continuous NonLinear module

4.4.1 Nonlinear Correlation example

(Source code)

```
"""
Nonlinear Correlation example :
It computes the nonparametric nonlinear regression coefficient h2 describing the dependency
between two continouos univariate signals x and y (in pandas DataFrame format)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 univariate signals \n")
print("************************************************************")

""" Import wanted module with every parent packages """
import DataFrom2Persons.Univariate.Continuous.Nonlinear.NonlinearCorr as NonlinearCorr
```

```
from utils.ExtractSignal import ExtractSignalFromCSV
from DataFrom2Persons.Univariate.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_Univariate_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 of 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]

""" Instantiate 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 of 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 coefficients 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")
```

4.4.2 Mutual Information example

(Source code)

```
"""
Mutual Information example:
It Computes Mutual Information (MI) estimators starting from entropy estimates from k-nearest-neighbours distances.

"""
```

```
""" 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("********************************************************************************\n")
print("This scripts computes two Mutual Information estimators from signals \n" +
      "by using k-nearest-neighbours approach.\n")
print("********************************************************************************\n")

""" Import wanted module with every parent packages """
import DataFrom2Persons.Univariate.Continuous.Nonlinear.MutualInformation as MutualInformation

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


""" Define signals in pd.dataFrame format """
print ("Two independent uniformly distributed signals: Mi estimator should be \n" +
       "around 0 (negative values are due to statistical fluctuations)\n")
print("********************************************************************************\n")

# Create signals
n = 1000
x = pd.DataFrame(1.0*np.random.rand(n,1), range(0,n))
y = pd.DataFrame(1.0*np.random.rand(n,1), range(0,n))

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

x = ExtractSignalFromCSV(filename, columns = ['x1'])
y = ExtractSignalFromCSV(filename, columns = ['x2'])
n = 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(range(0,n), x, label="x")
axarr[1].plot(range(0,n), y, label="y", color='r')
axarr[0].legend(loc='best')
axarr[1].legend(loc='best')


""" Define class attributes of the wanted method """
n_neighbours = 10                   # the number of the nearest neighbours to be used
my_type = 1                         # the type of estimators
var_res = True                      # rescaling of the time series
noise = True                        # adding random noise to the time series


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

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

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

""" Compute the method and get the result """
```

```
print("\n")
print("Computing...")

try :
    res1= c.compute(x, y)
except TypeError, err :
    print("TypeError in MutualInformation computation : \n" + str(err))
    sys.exit(-1)
except ValueError, err :
    print("ValueError in MutualInformation computation : \n" + str(err))
    sys.exit(-1)
except Exception, e :
    print("Exception in MutualInformation computation : \n" + str(e))
    sys.exit(-1)

""" OTHER TRY WITH my_type = 2 """
my_type = 2

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

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

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

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

```
try :
    res2= c.compute(x, y)
except TypeError, err :
    print("TypeError in MutualInformation computation : \n" + str(err))
    sys.exit(-1)
except ValueError, err :
    print("ValueError in MutualInformation computation : \n" + str(err))
    sys.exit(-1)
except Exception, e :
    print("Exception in MutualInformation computation : \n" + str(e))
    sys.exit(-1)


""" Display results """
print("\n")
print("**************************************\n")
print('MutualInformation complete result 1:\n')
print("**************************************\n")
print("MutualInformation estimator 1:")
print(res1)
print("\n")
print("MutualInformation estimator 2:")
print(res2)

#mixing matrix
C=np.random.rand(2,2)
x1 = pd.DataFrame(C[0,0]*x.iloc[:,0] + C[0,1]*y.iloc[:,0], x.index)
y1 = pd.DataFrame(C[1,0]*x.iloc[:,0] + C[1,1]*y.iloc[:,0], y.index)

print ("\n")
print("*************************************************************************************************\n")
print ("We add some dependenty to the time series (new times series are linear combination of x and y):\n" +
       "Mi estimator should be now greater than 0 \n")
print("*************************************************************************************************\n")

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

n_neighbours = 10                   # the number of the nearest neighbours to be used
my_type = 1                         # the type of estimators
var_res = True                      # rescaling of the time series
noise = True                        # adding random noise to the time series
```

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

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

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

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

try :
    res1= c.compute(x1, y1)
except TypeError, err :
    print("TypeError in MutualInformation computation : \n" + str(err))
    sys.exit(-1)
except ValueError, err :
    print("ValueError in MutualInformation computation : \n" + str(err))
    sys.exit(-1)
except Exception, e :
    print("Exception in MutualInformation computation : \n" + str(e))
    sys.exit(-1)


my_type = 2
```

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

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

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

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

try :
    res2= c.compute(x1, y1)
except TypeError, err :
    print("TypeError in MutualInformation computation : \n" + str(err))
    sys.exit(-1)
except ValueError, err :
    print("ValueError in MutualInformation computation : \n" + str(err))
    sys.exit(-1)
except Exception, e :
    print("Exception in MutualInformation computation : \n" + str(e))
    sys.exit(-1)

""" Display result """
print("\n")
print("************************************ \n")
print('MutualInformation complete result 2:\n')
print("************************************\n")
```

```
print("MutualInformation estimator 1:")
print(res1)
print("\n")
print("MutualInformation estimator 2:")
print(res2)

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

^

4.4.3 Window Mutual Information example

(Source code)

```
"""
WindowMututalInformation example :
Computes the windowed mutual information between two continuous univariate signals x and y (in pandas DataFrame format)
"""

""" 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 mutual information between two monovariate time series \n" +
      "expressed as Python Pandas DataFrame.")
print("********************************************************************************")

""" Import wanted module with every parent packages """
import DataFrom2Persons.Univariate.Continuous.Nonlinear.WindowMutualInformation as WindowMutualInformation


""" Define signals in pd.dataFrame format """
# Create signals
n = 100
```

```
x = pd.DataFrame(1.0*np.random.rand(n,1), range(0,n))
y = pd.DataFrame(1.0*np.random.rand(n,1), range(0,n))

""" Plot input signals """
n = [float(i) 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('Time (s)')
ax.set_ylabel('signals')
ax.set_title('Input signals')
ax.plot(n , y, label='y')
ax.plot(n , x, label='x')
plt.legend(bbox_transform=plt.gcf().transFigure)

""" Define class attributes of the wanted method """
n_neighbours = 5          # the number of the nearest neighbours to be used
my_type = 1               # the type of estimators
var_res = True            # rescaling of the time series
noise = True              # adding random noise to the time series

tau_max = 10              # the maximum lag at which correlation should be computed. It is in the range [0; (lx+ly-1)/2] (in samples)
window = 10               # length of the windowed signals (in samples)
window_inc =  1           # 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


""" Instanciate the class with its attributes """
print("\n")
try :
    wmi = WindowMutualInformation.WindowMutualInformation(n_neighbours, my_type, var_res,noise, tau_max, window, window_inc, tau_inc,
except TypeError, err :
    print("TypeError in WindowMutualInformation constructor : \n" + str(err))
    sys.exit(-1)
except ValueError, err :
    print("ValueError in WindowMutualInformation constructor : \n" + str(err))
    sys.exit(-1)
except Exception, e :
    print("Exception in WindowMutualInformation constructor : \n" + str(e))
    sys.exit(-1)
```

```
print("An instance the class is now created with the following parameters:\n" +
      "n_neighbours = " + str(n_neighbours) + "\n" +
      "my_type = " + str(my_type) + "\n" +
      "var_res = " + str(var_res) + "\n" +
      "noise = " + str(noise) + "\n" +
      "tau max = " + str(tau_max) + "\n" +
      "window length = " + str(window) + "\n" +
      "window increment = " + str(window_inc) + "\n" +
      "tau increment = " + str(window_inc) + "\n" +
      "plot result = " + str(plot))

print("\n")

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

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


""" Display result """
print("\n")
print("********************************* \n")
print('Window Mutual Information result :')
print("********************************* \n")
'''print(win_MI)'''

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

4.4.4 PhaseSynchro_Strobo example

(Source code)

```
"""
PhaseSynchro_Strobo example :
It computes the n:m synchronization index lambda_nm by using a stroboscopic approach between two continuous univariate signals x and y
(in DataFrame format).
"""

""" 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 n:m synchronization index lambda_nm by \n" +
      "using a stroboscopic approach between two continuous univariate \n" +
      "signals x and y (in DataFrame format).\n")
print("************************************************************")

""" Import wanted module with every parent packages """
import DataFrom2Persons.Univariate.Continuous.Nonlinear.PhaseSynchro_Strobo as PhaseSynchro_Strobo


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

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


x=pd.DataFrame(np.sin(t), np.arange(0,N))
y=pd.DataFrame(np.sin(3*t+10), np.arange(0,N))


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


""" Define class attributes of the wanted method """
n = 3                   # integer of the order of synchronization
m  = 1                  # integer of the order of synchronization
nbins_mode = 'man'      # mode used to compute the nbins number
nbins = 10              # number of bins



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

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


print("An instance of the class is now created with the following parameters:\n" +
      "n = " + str(n) + "\n" +
      "m = " + str(m) + "\n" +
      "nbins_mode = " + str(nbins_mode) + "\n" +
      "nbins = " + 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 PhaseSynchro_Strobo computation : \n" + str(err))
    sys.exit(-1)
except ValueError, err :
    print("ValueError in PhaseSynchro_Strobo computation : \n" + str(err))
    sys.exit(-1)
except Exception, e :
    print("Exception in PhaseSynchro_Strobo computation : \n" + str(e))
    sys.exit(-1)



""" Display results """
print("\n")
print("****************************************\n")
print('PhaseSynchro_Strobo complete result:\n')
print("****************************************\n")
print("lambda_nm:")
print(res)
print("\n")

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

4.4.5 PhaseSynchro_Entropy example

(Source code)

```
"""
PhaseSynchro_Entropy example :
It computes the n:m synchronization index rho_nm by using a Shannon entropy based approach between two univariate signals x and y
(in pandas DataFrame format). Rho_nm ranges in [0,1] where 0 means no synchronization at all and 1 means perfect synchronization.
"""

""" 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 n:m synchronization index rho_nm by \n" +
      "using a Shannon entropy based approach between two univariate signals x and y\n" +
      "(in pandas DataFrame format). Rho_nm ranges in [0,1] where 0 means \n" +
      "no synchronization at all and 1 means perfect synchronization.")
print("********************************************************************************")

""" Import wanted module with every parent packages """
import DataFrom2Persons.Univariate.Continuous.Nonlinear.PhaseSynchro_Entropy as PhaseSynchro_Entropy


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

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


x=pd.DataFrame(np.sin(t), np.arange(0,N))
y=pd.DataFrame(np.sin(3*t+10), np.arange(0,N))


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

```
""" Define class attributes of the wanted method """
n =  3                      # integer of the order of synchronization
m  = 1                      # integer of the order of synchronization
nbins_mode = 'man'          # mode used to compute the nbins number
nbins = 50                  # number of bins
dist_cyc_rel_phase = False # plot of the distribution of the cyclci relative phase

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

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


print("An instance of the class is now created with the following parameters:\n" +
      "n = " + str(n) + "\n" +
      "m = " + str(m) + "\n" +
      "nbins_mode = " + str(nbins_mode) + "\n" +
      "nbins = " + str(nbins)+ "\n" +
      "dist_cyc_rel_phase =" + str(dist_cyc_rel_phase))


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

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

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



""" Display results """
print("\n")
print("****************************************\n")
print('PhaseSynchro_Entropy complete result:\n')
print("****************************************\n")
print("rho_nm:")
print(res)
print("\n")

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

4.4.6 PhaseSynchro_Fourier example

(Source code)

```
"""
PhaseSynchro_Fourier example :
It computes the n:m synchronization index gamma2_nm as the intensity of the first Fourier mode of the cyclic relative phase two contin
(in DataFrame format). Gamma2_nm ranges in [0,1] where 0 means no synchronization at all and 1 means perfect synchronization.
"""

""" 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 n:m synchronization index gamma2_nm as \n" +
```

```
      "the intensity of the first Fourier mode of the cyclic relative phase \n" +
      "of two continuous univariate signals x and y (in DataFrame format). \n" +
      "Gamma2_nm ranges in [0,1] where 0 means no synchronization at all and 1 \n" +
      "means perfect synchronization.")
print("*************************************************************************")

""" Import wanted module with every parent packages """
import DataFrom2Persons.Univariate.Continuous.Nonlinear.PhaseSynchro_Fourier as PhaseSynchro_Fourier


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

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


x=pd.DataFrame(np.sin(t), np.arange(0,N))
y=pd.DataFrame(np.sin(3*t+10), np.arange(0,N))


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


""" Define class attributes of the wanted method """
n =  2                      # integer of the order of synchronization
m  = 1                      # integer of the order of synchronization

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

try :
    c=c=PhaseSynchro_Fourier.PhaseSynchro_Fourier(n ,m)
```

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


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


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

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



""" Display results """
print("\n")
print("****************************************\n")
print('PhaseSynchro_Fourier complete result:\n')
print("****************************************\n")
print("gamma2_nm:")
print(res)
print("\n")
```

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

4.5 DataFromManyPersons Univariate Continuous Linear module

4.5.1 Conditional Granger Causality example

(Source code)

```
"""
ConditionalGrangerCausality example :
It computes Condtional Granger Causality among signals (in pandas DataFrame format) organized 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("=================================================================")
print("== Testing for Conditional Granger Causality between 4 signals ==")
print("=================================================================")

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

""" Import wanted module with every parent packages """
import DataFromManyPersons.Univariate.Continuous.Linear.ConditionalGrangerCausality as CGC

""" Import signal from a .csv file """
filename = 'data_examples/1Person_Multivariate_Continuous_data.csv'
print "\nLoading signals from csv files : ", filename,"\n"
x1 = ExtractSignalFromCSV(filename, columns = ['x1'])
x2 = ExtractSignalFromCSV(filename, columns = ['x2'])
x3 = ExtractSignalFromCSV(filename, columns = ['x3'])
```

```
x4 = ExtractSignalFromCSV(filename, columns = ['x4'])


""" 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

""" Instantiate the class with its attributes """
print("\n")
try :
        cgc = CGC.ConditionalGrangerCausality(max_lag = max_lag, criterion = criterion, plot = plot)
except TypeError, err :
        print("TypeError in ConditionalGrangerCausality constructor : \n" + str(err))
        sys.exit(-1)
except ValueError, err :
        print("ValueError in ConditionalGrangerCausality constructor : \n" + str(err))
        sys.exit(-1)
except Exception, e :
        print("Exception in ConditionalGrangerCausality 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 = cgc.compute(x1,x2,x3,x4)
except TypeError, err :
        print("TypeError in ConditionalGrangerCausality computation : \n" + str(err))
        sys.exit(-1)

except ValueError, err :
        print("ValueError in ConditionalGrangerCausality computation : \n" + str(err))
        sys.exit(-1)

except Exception, e :
        print("Exception in ConditionalGrangerCausality computation : \n" + str(e))
```

```
        sys.exit(-1)

raw_input("Press any key to exit")
```

4.5.2 Omega complexity example

(Source code)

```
"""
Omega_Complexity example :
Compute Omega_Complexity for multiple monovariate signals (organized as a list of pandas DataFrames).
"""

""" 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 Omega_Complexity for multiple monovariate signals \n" +
      "(organized as a list of pandas DataFrames) \n")
print("**********************************************************************")

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

""" Define signals in pd.dataFrame format """
# preparing the input time series
N = 1000             # 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)}, np.arange(0,N) )
y = pd.DataFrame({'Y':np.sin(4*3.14*f*n/Fs)}, np.arange(0,N) )
z = pd.DataFrame({'Z':np.cos(2*3.14*f*n/Fs)}, np.arange(0,N) )
w = pd.DataFrame(2.0*np.random.rand(N,1),np.arange(0,N))


'''
"""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'])
'''
''''
"""OR"""
""" Import signal from a .mat file """
filename = 'data_examples/data_example_MAT.mat'
z = ExtractSignalFromMAT(filename, columns_index =[0,2], columns_wanted_names=['Time', 'GlobalBodyActivity0'])
t = ExtractSignalFromMAT(filename, columns_index =[10], columns_wanted_names=['GlobalBodyActivity1'])
'''

signals = [x,y,z,w]

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 """

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

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

print("Instantiating the class...")

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

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

""" Display result """
print("\n")
print("*****************************\n")
print('Omega_Complexity result :')
print("*****************************\n")
print("\n")

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

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

4.5.3 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.Univariate.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.")
```

4.5.4 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.Univariate.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 = 1000             # 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.cos(2*3.14*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()

nrows = len(Signals)
figure, ax = plt.subplots(nrows, sharex=True)

idx = 0
for col in  range(len(Signals)) :
    ax[idx].grid(True) # Display a grid
    ax[idx].set_title('Standardized signal for : ' + Signals[col].columns[0] + ' variable')
    ax[idx].plot(n, Signals[col].iloc[:,0])
    idx += 1

ax[idx-1].set_xlabel('Samples')

""" 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.")
```

4.5.5 Multiple Granger Causality example

(Source code)

```
"""
MultipleGrangerCausality example :
Computes a Granger Causality test between some signals that are stored as a pandas 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 between some signals \n" +
      "in pandas DataFrame format.")
print("********************************************************************************")

""" Import wanted module with every parent packages """
import DataFromManyPersons.Univariate.Continuous.Linear.MultipleGrangerCausality as MGC
from utils.ExtractSignal import ExtractSignalFromCSV


""" Define signals in pd.dataFrame format """
# preparing the input time series
N = 1000                                                        # Size of signals
X  = pd.DataFrame({'X' :np.random.randn(N)})        # Signal to test
Y1 = pd.DataFrame({'Y1':np.random.randn(N)})        # Helping signal n1
Y2 = pd.DataFrame({'Y2':np.random.randn(N)})        # Helping signal n2
Y3 = pd.DataFrame({'Y3':np.random.randn(N)})        # Helping signal n3
signal = [X,Y1,Y2,Y3]

'''
"""OR"""
""" Import signal from a .csv file """
filename = 'data_examples/1Person_Multivariate_Continous_data.csv'
```

```
x1 = ExtractSignalFromCSV(filename, columns = ['x1'])
x2 = ExtractSignalFromCSV(filename, columns = ['x2'])
x3 = ExtractSignalFromCSV(filename, columns = ['x3'])
x4 = ExtractSignalFromCSV(filename, columns = ['x4'])
x5 = ExtractSignalFromCSV(filename, columns = ['x5'])
signal = [x1,x2,x3,x4,x5]
'''

""" Plot input signals """
Signals = [X, Y1, Y2, Y3]
plt.ion()

nrows = len(Signals)
figure, ax = plt.subplots(nrows, sharex=True)
idx = 0
for col in  range(len(Signals)) :
    ax[idx].grid(True) # Display a grid
    ax[idx].set_title('Input signal : ' + str(Signals[col].columns[0]))
    ax[idx].plot(Signals[col].index, Signals[col].iloc[:,0])
    idx += 1

ax[idx-1].set_xlabel('Time')

""" Define class attributes """
max_lag = 10                # Define the maximum lag acceptable to estimate autoregressive models
criterion = 'aic'         # 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 :
        mgc = MGC.MultipleGrangerCausality(max_lag = max_lag, criterion = criterion, plot = plot)
except TypeError, err :
        print("TypeError in MultipleGrangerCausality constructor : \n" + str(err))
        sys.exit(-1)
except ValueError, err :
        print("ValueError in MultipleGrangerCausality constructor : \n" + str(err))
        sys.exit(-1)
except Exception, e :
        print("Exception in MultipleGrangerCausality 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 = mgc.compute(*signal)
except TypeError, err :
        print("TypeError in MultipleGrangerCausality computation : \n" + str(err))
        sys.exit(-1)

except ValueError, err :
        print("ValueError in MultipleGrangerCausality computation : \n" + str(err))
        sys.exit(-1)

except Exception, e :
        print("Exception in MultipleGrangerCausality 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",criterion,", the optimal number of lag estimated is :", results['optimal_lag'],"\n"
print "F_value =",results['F_value']," with p_value =",results['p_value'],"\n"

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

4.6 DataFrom2Persons Multivariate Continuous NonLinear module

4.6.1 GSI example

(Source code)

```
"""
GSI example :
```

```
It computes the generalised synchronization index (GSI) between two uni/multi-variate signals x and y(in DataFrame format).
GSI ranges in [0,1] where 0 means no synchronization and 1 perfect generalized synchronization.
"""

""" 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 generalised synchronization index (GSI) between \n" +
      "two uni/multi-variate signals x and y(in DataFrame format).\n" +
      "GSI ranges in [0,1] where 0 means no synchronization and 1 perfect generalized synchronization.")
print("********************************************************************************")

""" Import wanted module  """
import DataFrom2Persons.Multivariate.Continuous.Nonlinear.GSI as GSI

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

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

#Create signals
x=pd.DataFrame(np.sin(t), np.arange(0,N))
y=pd.DataFrame(np.sin(3*t+10), np.arange(0,N))

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

```
""" Define parameters of the wanted method """
m=1                         # the mebedding dimension
t=1                         # the delay between data
rr=0.1                      # the threshold rate for recurrence


""" Call CrossRecurrencePlot utils method """
print("\n")

try :
    c =GSI.GSI(m, t, rr)
except TypeError, err :
    print("TypeError in GSI : \n" + str(err))
    sys.exit(-1)
except ValueError, err :
    print("ValueError in GSI : \n" + str(err))
    sys.exit(-1)
except Exception, e :
    print("Exception in GSI : \n" + str(e))
    sys.exit(-1)



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


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


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

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


""" Display results """
print("\n")
print("****************************************\n")
print('GSI complete result:\n')
print("****************************************\n")
print("GSI:")
print(res)
print("\n")



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

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.