pyunicorn Documentation

Release 0.5.1

Jonathan F. Donges and pyunicorn authors

April 17, 2016

Contents

1 Introduction 1

2 Download 3
2.1 Code . . . 3
2.2 Documentation . . . 3
2.3 Reference . . . 3
2.4 Platforms and Compatibility . . . 3
2.5 Dependencies . . . 3
2.6 Installing . . . 4

3 Tutorials 5
3.1 Constructing and analyzing a climate network . . . 5
3.2 Recurrence network analysis of the logistic map . . . 8

4 Methods 11
4.1 General complex networks . . . 11
4.2 Spatially embedded networks . . . 11
4.3 Interacting/interdependent/multiplex networks / networks of networks . . . 11
4.4 Node-weighted network measures / node-splitting invariance . . . 11
4.5 Climate networks / Coupled climate networks . . . 12
4.6 Recurrence networks / recurrence quantification analysis / recurrence plots . . . 12
4.7 Visibility graph analysis . . . 12
4.8 Surrogate time series . . . 13

5 API 15
5.1 core . . . 15
5.2 climate . . . 92
5.3 timeseries . . . 127
5.4 funcnet . . . 155
5.5 utils . . . 163

6 Development 169
6.1 Test suite . . . 169
6.2 Mailing list . . . 169

7 Changelog 171

8 Publications 173
8.1 General complex networks . . . 173
8.2 Spatially embedded networks . . . 173
8.3 Interacting/interdependent networks / networks of networks . . . 173
8.4 Node-weighted network measures / node-splitting invariance . . . 173
8.5 Climate data analysis (general) . . . 174

8.6 Climate networks / Coupled climate networks 174
8.7 Power Grids/Power Networks 174
8.8 Time series analysis and synchronization (general) 175
8.9 Recurrence networks / quantification analysis / plots 175
8.10 Visibility graph analysis 176

9 License **177**

10 Contact **179**
10.1 Reference 179

Bibliography **181**

Python Module Index **187**

Index **189**

CHAPTER 1

Introduction

pyunicorn (**Uni**fied **Co**mplex Network and **R**ecurre**N**ce analysis toolbox) is a fully object-oriented Python package for the advanced analysis and modeling of complex networks. Above the standard measures of complex network theory such as degree, betweenness and clustering coefficient it provides some **uncommon but interesting statistics** like Newman's random walk betweenness. pyunicorn features novel **node-weighted (node splitting invariant)** network statistics as well as measures designed for analyzing **networks of interacting/interdependent networks**.

Moreover, pyunicorn allows to easily **construct networks from uni- and multivariate time series data** (functional (climate) networks and recurrence networks). This involves linear and nonlinear measures of time series analysis for constructing functional networks from multivariate data as well as modern techniques of nonlinear analysis of single time series like recurrence quantification analysis (RQA) and recurrence network analysis.

For example, to generate a recurrence network with 1000 nodes from a sinusoidal signal and compute its network transitivity you simply need to type

```
import numpy as np
from pyunicorn.timeseries import RecurrenceNetwork

x = np.sin(np.linspace(0, 10 * np.pi, 1000))
net = RecurrenceNetwork(x, recurrence_rate=0.05)
print net.transitivity()
```

The package provides special tools to analyze and model **spatially embedded** complex networks.

pyunicorn is **fast** because all costly computations are performed in compiled C, C++ and Fortran code. It can handle **large networks** through the use of sparse data structures. The package can be used interactively, from any python script and even for parallel computations on large cluster architectures.

CHAPTER 2

Download

2.1 Code

Stable releases (https://github.com/pik-copan/pyunicorn/releases), Development version (https://github.com/pik-copan/pyunicorn)

2.2 Documentation

For extensive HTML documentation, jump right to the pyunicorn homepage (http://www.pik-potsdam.de/ donges/pyunicorn/). Recent PDF versions (http://www.pik-potsdam.de/ donges/pyunicorn/docs/) are also available.

On a local development version, HTML and PDF documentation can be generated using `Sphinx`:

```
$> pip install --user -e .
$> cd docs; make clean html latexpdf
```

2.3 Reference

Please acknowledge and cite the use of this software and its authors when results are used in publications or published elsewhere. You can use the following reference:

> J.F. Donges, J. Heitzig, B. Beronov, M. Wiedermann, J. Runge, Q.-Y. Feng, L. Tupikina, V. Stolbova, R.V. Donner, N. Marwan, H.A. Dijkstra, and J. Kurths, Unified functional network and nonlinear time series analysis for complex systems science: The pyunicorn package, Chaos 25, 113101 (2015), doi:10.1063/1.4934554, (http://dx.doi.org/10.1063/1.4934554) Preprint: arxiv.org:1507.01571 [physics.data-an]. (http://arxiv.org/abs/1507.01571)

2.4 Platforms and Compatibility

`pyunicorn` is written in Python 2.7. The software is quite flexible, we have it running on Linux and MacOSX machines, the institute's IBM iDataPlex cluster and even on Windows.

2.5 Dependencies

`pyunicorn` relies on the following open source or freely available packages which have to be installed on your machine.

Required:

- Numpy (http://numpy.scipy.org/) 1.8+
- Scipy (http://www.scipy.org/) 0.14+
- Weave (https://github.com/scipy/weave) 0.15+
- igraph, python-igraph (http://igraph.sourceforge.net/) 0.7+

Optional ***(used only in certain classes and methods)*****:**

- PyNGL (http://www.pyngl.ucar.edu/Download/) (for class NetCDFDictionary)
- netcdf4-python (http://code.google.com/p/netcdf4-python/) (for classes Data and NetCDFDictionary)
- Matplotlib (http://matplotlib.sourceforge.net) 1.3+
- Matplotlib Basemap Toolkit (http://matplotlib.org/basemap/) (for drawing maps)
- mpi4py (http://code.google.com/p/mpi4py/) (for parallelizing costly computations)
- Sphinx (http://sphinx-doc.org/) (for generating documentation)
- Cython (http://cython.org/) 0.21+ (for compiling code during development)

Numpy, Scipy, Matplotlib, igraph and other packages should be available via a package management system on Linux or MacOSX. All packages can be downloaded, compiled and installed following the instructions on their homepages.

An easy way to go may be a Python distribution like Anaconda (https://store.continuum.io/cshop/anaconda/) or Enthought (http://www.enthought.com) that already include many libraries.

2.6 Installing

Stable release Via the Python Package Index:

```
$> pip install pyunicorn
```

Development version For a simple system-wide installation:

```
$> pip install .
```

Depending on your system, you may need root privileges. On UNIX-based operating systems (Linux, Mac OS X etc.) this is achieved with `sudo`.

For development, especially if you want to test `pyunicorn` from within the source directory:

```
$> pip install --user -e .
```

CHAPTER 3

Tutorials

The tutorials are designed to be self-explanatory. For further details on the used classes and methods please refer to the API.

3.1 Constructing and analyzing a climate network

This tutorials illustrates the use of `climate` for constructing a climate network from given data in a commonly used format, performing a statistical analysis of the network and finally plotting the results on a map.

For example, our software can handle data from the NCEP/NCAR reanalysis 1 project like this monthly surface air temperature data set (a NetCDF file): ftp://ftp.cdc.noaa.gov/Datasets/ncep.reanalysis.derived/surface/air.mon.mean.nc

You can use `PyNgl` for plotting the results on maps (http://www.pyngl.ucar.edu/Download/). Alternatively, the tutorial saves the results as well as the grid information in text files which can be used for plotting in your favorite software.

```
#!/usr/bin/python
# -*- coding: utf-8 -*-

"""
Tutorial on analyzing climate networks using Python.

Uses the Python packages ``core`` and ``climate`` providing all kinds of tools
related to climate networks. Written as part of a diploma / phd thesis in
Physics by Jonathan F. Donges (donges@pik-potsdam.de) at University of Potsdam
/ Humboldt University Berlin and Potsdam Institute of Climate Impact Research
(PIK),

Copyright 2008-2015.
"""

import numpy as np

from pyunicorn import climate


#
#  Settings
#

#  Related to data

#  Path and filename of NetCDF file containing climate data
DATA_FILENAME = "../../../Daten/Reanalysis/NCEP-NCAR/air.mon.mean.nc"
```

```
#  Type of data file ("NetCDF" indicates a NetCDF file with data on a regular
#  lat-lon grid, "iNetCDF" allows for arbitrary grids - > see documentation).
#  For example, the "NetCDF" FILE_TYPE is compatible with data from the IPCC
#  AR4 model ensemble or the reanalysis data provided by NCEP/NCAR.
FILE_TYPE = "NetCDF"

#  Indicate data source (optional)
DATA_SOURCE = "ncep_ncar_reanalysis"

#  Name of observable in NetCDF file ("air" indicates surface air temperature
#  in NCEP/NCAR reanalysis data)
OBSERVABLE_NAME = "air"

#  Select a subset in time and space from the data (e.g., a particular region
#  or a particular time window, or both)
WINDOW = {"time_min": 0., "time_max": 0., "lat_min": 0, "lon_min": 0,
          "lat_max": 30, "lon_max": 0}  # selects the whole data set

#  Indicate the length of the annual cycle in the data (e.g., 12 for monthly
#  data). This is used for calculating climatological anomaly values
#  correctly.
TIME_CYCLE = 12

#  Related to climate network construction

#  For setting fixed threshold
THRESHOLD = 0.5

#  For setting fixed link density
LINK_DENSITY = 0.005

#  Indicates whether to use only data from winter months (DJF) for calculating
#  correlations
WINTER_ONLY = False

#
#  Print script title
#

print "\n"
print "Tutorial on how to use climate"
print "-------------------------------"
print "\n"

#
#  Create a ClimateData object containing the data and print information
#

data = climate.ClimateData.Load(
    file_name=DATA_FILENAME, observable_name=OBSERVABLE_NAME,
    data_source=DATA_SOURCE, file_type=FILE_TYPE,
    window=WINDOW, time_cycle=TIME_CYCLE)

#  Print some information on the data set
print data

#
#  Create a MapPlots object to manage 2D-plotting on the sphere
#

#  Comment this if you are not using pyngl for plotting!!!
map_plots = climate.MapPlots(data.grid, DATA_SOURCE)
```

```
#
#  Generate climate network using various procedures
#

#  One of several alternative similarity measures and construction mechanisms
#  may be chosen here

#  Create a climate network based on Pearson correlation without lag and with
#  fixed threshold
net = climate.TsonisClimateNetwork(
    data, threshold=THRESHOLD, winter_only=WINTER_ONLY)

#  Create a climate network based on Pearson correlation without lag and with
#  fixed link density
# net = climate.TsonisClimateNetwork(
#     data, link_density=LINK_DENSITY, winter_only=WINTER_ONLY)

#  Create a climate network based on Spearman's rank order correlation without
#  lag and with fixed threshold
# net = climate.SpearmanClimateNetwork(
#     data, threshold=THRESHOLD, winter_only=WINTER_ONLY)

#  Create a climate network based on mutual information without lag and with
#  fixed threshold
# net = climate.MutualInfoClimateNetwork(
#     data, threshold=THRESHOLD, winter_only=WINTER_ONLY)

#
#  Some calculations
#

print "Link density:", net.link_density

#  Get degree
degree = net.degree()
#  Get closeness
closeness = net.closeness()
#  Get betweenness
betweenness = net.betweenness()
#  Get local clustering coefficient
clustering = net.local_clustering()
#  Get average link distance
ald = net.average_link_distance()
#  Get maximum link distance
mld = net.max_link_distance()

#
#  Save results to text file
#

#  Save the grid (mainly vertex coordinates) to text files
data.grid.save_txt(filename="grid.txt")

#  Save the degree sequence. Other measures may be saved similarly.
np.savetxt("degree.txt", degree)

#
#  Plotting
#

#  Comment everything below if you are not using pyngl for plotting!

#  Add network measures to the plotting queue
```

```
map_plots.add_dataset("Degree", degree)
map_plots.add_dataset("Closeness", closeness)
map_plots.add_dataset("Betweenness (log10)", np.log10(betweenness + 1))
map_plots.add_dataset("Clustering", clustering)
map_plots.add_dataset("Average link distance", ald)
map_plots.add_dataset("Maximum link distance", mld)

#  Change the map projection
map_plots.resources.mpProjection = "Robinson"
map_plots.resources.mpCenterLonF = 0

#  Change the levels of contouring
map_plots.resources.cnLevelSelectionMode = "EqualSpacedLevels"
map_plots.resources.cnMaxLevelCount = 20

# map_plots.resources.cnRasterSmoothingOn = True
# map_plots.resources.cnFillMode = "AreaFill"

map_plots.generate_map_plots(file_name="climate_network_measures",
                             title_on=False, labels_on=True)
```

3.2 Recurrence network analysis of the logistic map

This tutorial demonstrates how to use `timeseries` for a nonlinear time series analysis of a realization of the chaotic logistic map. It also features `weave` for including fast inline C++ code.

```
#!/usr/bin/python
# -*- coding: utf-8 -*-

"""
Tutorial on how to handle recurrence plots and recurrence networks using
Python, based on the timeseries package.

Written as part of a PhD thesis in Physics by Jonathan F. Donges
(donges@pik-potsdam.de) at the Potsdam Institute of Climate Impact Research
(PIK) and Humboldt University Berlin,

Copyright 2008-2015.
"""

# array object and fast numerics
import numpy as np
# C++ inline code
import weave

# plotting facilities
import pylab

from pyunicorn.timeseries import RecurrencePlot, RecurrenceNetwork


#
#  Functions
#

def logistic_map(x0, r, T):
    """
    Returns a time series of length T using the logistic map
    x_(n+1) = r*x_n(1-x_n) at parameter r and using the initial condition x0.
    INPUT: x0 - Initial condition, 0 <= x0 <= 1
            r - Bifurcation parameter, 0 <= r <= 4
            T - length of the desired time series
    """
    #  Initialize the time series array
    timeSeries = np.empty(T)

    r = float(r)

    code = r"""
    int i;
    double xn;

    // Set initial condition
    timeSeries(0) = x0;

    for (i = 1; i < T; i++) {
        xn = timeSeries(i-1);
        timeSeries(i) = r * xn * (1 - xn);
    }
    """
    args = ['x0', 'r', 'T', 'timeSeries']
    weave.inline(code, arg_names=args, type_converters=weave.converters.blitz,
                 compiler='gcc', extra_compile_args=['-O3'])

    return timeSeries


def logistic_map_lyapunov_exponent(timeSeries, r):
    """
    Returns the Lyapunov exponent of the logistic map for different r.

    INPUT: timeSeries - The time series generated by a logistic map
                    r - the bifurcation parameter
    """
    lyap = np.log(r) + (np.log(np.abs(1 - 2 * timeSeries))).mean()

    return lyap

#
#  Settings
#

#  Parameters of logistic map
r = 3.679   # Bifurcation parameter
x0 = 0.7    # Initial value

#  Length of the time series
T = 150

#  Settings for the embedding
DIM = 1   # Embedding dimension
TAU = 0   # Embedding delay

#  Settings for the recurrence plot
EPS = 0.05   # Fixed threshold
RR = 0.05    # Fixed recurrence rate
# Distance metric in phase space ->
# Possible choices ("manhattan","euclidean","supremum")
METRIC = "supremum"

#
#  Main script
```

```
#

#  Create a time series using the logistic map
time_series = logistic_map(x0, r, T)

#  Print the time series
print time_series

#  Plot the time series
pylab.plot(time_series, "r")
#  You can include LaTex labels...
pylab.xlabel("$n$")
pylab.ylabel("$x_n$")

#  Generate a recurrence plot object with fixed recurrence threshold EPS
rp = RecurrencePlot(time_series, dim=DIM, tau=TAU, metric=METRIC,
                    normalize=False, threshold=EPS)

#  Show the recurrence plot
pylab.matshow(rp.recurrence_matrix())
pylab.xlabel("$n$")
pylab.ylabel("$n$")
pylab.show()

#  Calculate and print the recurrence rate
print "Recurrence rate:", rp.recurrence_rate()

#  Calculate some standard RQA measures
DET = rp.determinism(l_min=2)
LAM = rp.laminarity(v_min=2)

print "Determinism:", DET
print "Laminarity:", LAM

#  Generate a recurrence plot object with fixed recurrence rate RR
rp = RecurrencePlot(time_series, dim=DIM, tau=TAU, metric=METRIC,
                    normalize=False, recurrence_rate=RR)

#  Calculate and print the recurrence rate again to check if it worked...
RR = rp.recurrence_rate()
print "Recurrence rate:", RR

#  Calculate some standard RQA measures
DET = rp.determinism(l_min=2)
LAM = rp.laminarity(v_min=2)

print "Determinism:", DET
print "Laminarity:", LAM

#  Generate a recurrence network at fixed recurrence rate
rn = RecurrenceNetwork(time_series, dim=DIM, tau=TAU, metric=METRIC,
                       normalize=False, recurrence_rate=RR)

#  Calculate average path length, transitivity and assortativity
L = rn.average_path_length()
T = rn.transitivity()
C = rn.global_clustering()
R = rn.assortativity()

print "Average path length:", L
print "Transitivity:", T
print "Global clustering:", C
print "Assortativity:", R
```

CHAPTER 4

Methods

A brief introduction to the methods, measures and algorithms provided by `pyunicorn`.

4.1 General complex networks

Many standard complex network measures, network models and algorithms are supported, most of them inherited from the `igraph` package, e.g., degree, closeness and betweenness centralities, clustering coefficient and transitivity or commmunity detection algorithms and network models such as Erdos-Renyi or Barabasi-Albert. Moreover, a number of less common network statistics like Newman's or Arenas' random walk betweenness can be computed. Reading and saving network data from and to many common data formats is possible.

- core.network

4.2 Spatially embedded networks

`pyunicorn` includes measures and models specifically designed for spatially embedded networks (or simply spatial networks) via the GeoNetwork and Grid classes.

- core.geo_network
- core.grid

4.3 Interacting/interdependent/multiplex networks / networks of networks

The InteractingNetworks class provides a rich collection of network measures and models specifically designed for investigating the structure of networks of networks (also called interacting networks, interdependent networks or multiplex networks in different contexts). Examples include the cross-link density of connections between different subnetworks or the cross-shortest path betweenness quantifying the importance of nodes for mediating interactions between different subnetworks. Models of interacting networks allow to assess the degree of organization of the cross-connectivity between subnetworks.

- core.interacting_networks

4.4 Node-weighted network measures / node-splitting invariance

Node-weighted networks measures derived following the node-splitting invariance approach are useful for studying systems with nodes representing subsystems of heterogeneous size, weight, area, volume or importance, e.g., nodes representing grid cells of widely different area in climate networks or voxels of differing volume in functional brain networks. `pyunicorn` provides node-weighted variants of most standard and non-standard measures for networks as well as interacting networks.

- core.network
- core.interacting_networks

4.5 Climate networks / Coupled climate networks

`pyunicorn` provides classes for the easy construction and analysis of the statistical interdependency structure within and between fields of time series (functional networks) using various similarity measures such as Pearson and Spearman correlation, lagged linear correlation, mutual information and event synchronization. Climate networks allow the analysis of single fields of time series, whereas coupled climate networks focus on studying the interrelationships between two fields of time series. While there is a historical focus on applications to climate data, those methods can also be applied to other sources of time series data such as neuroscientific (e.g., FMRI and EEG data) or financial data (e.g., stock market indices).

- climate.climate_network
- climate.coupled_climate_network
- climate.climate_data

4.6 Recurrence networks / recurrence quantification analysis / recurrence plots

Recurrence analysis is a powerful method for studying nonlinear systems, particularly based on univariate and multivariate time series data. Recurrence quantification analysis (RQA) and recurrence network analysis (RNA) allow to classify different dynamical regimes in time series and to detect regime shifts, dynamical transitions or tipping points, among many other applications. Bivariate methods such as joint recurrence plots/networks, cross recurrence plots or inter system recurrence networks allow to investigate the coupling structure between two dynamical systems based on time series, including methods to detect the directionality of coupling. Recurrence analysis is applicable to general time series data from many fields such as climatology, paleoclimatology, medicine, neuroscience or economics.

- timeseries.recurrence_plot
- timeseries.recurrence_network
- timeseries.joint_recurrence_plot
- timeseries.joint_recurrence_network
- timeseries.cross_recurrence_plot
- timeseries.inter_system_recurrence_network

4.7 Visibility graph analysis

Visibility graph analysis is an alternative approach to nonlinear time series analysis, allowing to study among others fractal properties and long-term memory in time series. As a special feature, `pyunicorn` provides time-directed measures such as advanced and retarded degree/clustering that can be used for designing tests for time-irreversibility (time-reversal asymmetry) of processes.

- timeseries.visibility_graph

4.8 Surrogate time series

Surrogate time series are useful for testing hypothesis on observed time series properties, e.g., on what features of a time series are expected to arise with high probability for randomized time series with the same autocorrelation structure. `pyunicorn` can be used to generate various types of time series surrogates, including white noise surrogates, Fourier surrogates, amplitude adjusted Fourier (AAFT) surrogates or twin surrogates (conserving the recurrence structure of the underlying time series).

- timeseries.surrogates

API

Release 0.5.1

Date April 17, 2016

`pyunicorn` consists of five subpackages, where the `core` and `utils.mpi` namespaces are to be accessed by calling `import pyunicorn`. All subpackages except for `utils` directly export the classes defined in their submodules.

5.1 core

General network analysis and modeling.

5.1.1 core.data

Provides classes for analyzing spatially embedded complex networks, handling multivariate data and generating time series surrogates.

class `pyunicorn.core.data.`**`Data`**(*observable*, *grid*, *observable_name=None*, *observable_long_name=None*, *window=None*, *silence_level=0*)

Bases: `object`

Encapsulates general spatio-temporal data.

Also contains methods to load data from various file formats (currently NetCDF and ASCII).

Mainly an abstract class.

classmethod **`Load`**(*file_name*, *observable_name*, *file_type*, *dimension_names=None*, *window=None*, *vertical_level=None*, *silence_level=0*)

Initialize an instance of Data.

Supported file types `file_type` are:

- "NetCDF" for regular (rectangular) grids
- "iNetCDF" for irregular (e.g. geodesic) grids or station data.

The spatio-temporal window is described by the following dictionary:

```
window = {"time_min": 0., "time_max": 0., "lat_min": 0.,
          "lat_max": 0., "lon_min": 0., "lon_max": 0.}
```

Note: It is assumed that the NetCDF file to be loaded uses the following dimension names: lat, lon, time (e.g., as is the case for NCEP/NCAR reanalysis 1 data (http://www.esrl.noaa.gov/psd/data/gridded/data.ncep.reanalysis.html)). These standard dimension names can be modified using the dimension_names argument. Alternatively, the standard class constructor `__init__()` needs to be used after loading the data manually, e.g., employing netcdf4-python or scipy.io.netcdf functionality.

Parameters

- **`file_name`** (*str*) – The name of the data file.
- **`observable_name`** (*str*) – The short name of the observable within data file (particularly relevant for NetCDF).
- **`file_type`** (*str*) – The type of the data file.
- **`dimension_names`** (*dict*) – The names of the dimensions as used in the NetCDF file. Default: {"lat": "lat", "lon": "lon", "time": "time"}
- **`window`** (*dict*) – Spatio-temporal window to select a view on the data.
- **`vertical_level`** (*int*) – The vertical level to be extracted from the data file. Is ignored for horizontal data sets. If None, the first level in the data file is chosen.
- **`silence_level`** (*int*) – The inverse level of verbosity of the object.

static `SmallTestData`()

Return test data set of 6 time series with 10 sampling points each.

Example:

```
>>> Data.SmallTestData().observable()
array([[  0.00000000e+00,   1.00000000e+00,   1.22464680e-16,
         -1.00000000e+00,  -2.44929360e-16,   1.00000000e+00],
       [  3.09016994e-01,   9.51056516e-01,  -3.09016994e-01,
         -9.51056516e-01,   3.09016994e-01,   9.51056516e-01],
       [  5.87785252e-01,   8.09016994e-01,  -5.87785252e-01,
         -8.09016994e-01,   5.87785252e-01,   8.09016994e-01],
       [  8.09016994e-01,   5.87785252e-01,  -8.09016994e-01,
         -5.87785252e-01,   8.09016994e-01,   5.87785252e-01],
       [  9.51056516e-01,   3.09016994e-01,  -9.51056516e-01,
         -3.09016994e-01,   9.51056516e-01,   3.09016994e-01],
       [  1.00000000e+00,   1.22464680e-16,  -1.00000000e+00,
         -2.44929360e-16,   1.00000000e+00,   3.67394040e-16],
       [  9.51056516e-01,  -3.09016994e-01,  -9.51056516e-01,
          3.09016994e-01,   9.51056516e-01,  -3.09016994e-01],
       [  8.09016994e-01,  -5.87785252e-01,  -8.09016994e-01,
          5.87785252e-01,   8.09016994e-01,  -5.87785252e-01],
       [  5.87785252e-01,  -8.09016994e-01,  -5.87785252e-01,
          8.09016994e-01,   5.87785252e-01,  -8.09016994e-01],
       [  3.09016994e-01,  -9.51056516e-01,  -3.09016994e-01,
          9.51056516e-01,   3.09016994e-01,  -9.51056516e-01]])
```

Return type Data instance

Returns a Data instance for testing purposes.

`__init__`(*observable*, *grid*, *observable_name=None*, *observable_long_name=None*, *window=None*, *silence_level=0*)

Initialize an instance of Data.

The spatio-temporal window is described by the following dictionary:

```
window = {"time_min": 0., "time_max": 0., "lat_min": 0.,
          "lat_max": 0., "lon_min": 0., "lon_max": 0.}
```

Parameters

- **observable** (*2D array [time, index]*) – The array of time series to be represented by the `Data` instance.
- **grid** (`Grid` instance) – The Grid representing the spatial coordinates associated to the time series and their temporal sampling.
- **observable_name** (*str*) – A short name for the observable.
- **observable_long_name** (*str*) – A long name for the observable.
- **window** (*dict*) – Spatio-temporal window to select a view on the data.
- **silence_level** (*int*) – The inverse level of verbosity of the object.

__str__()
Return a string representation of the object.

__weakref__
list of weak references to the object (if defined)

classmethod _get_netcdf_data (*file_name*, *file_type*, *observable_name*, *dimension_names*, *vertical_level=None*, *silence_level=0*)
Import data from a NetCDF file with a regular and rectangular grid.

Supported file types `file_type` are:

- "NetCDF" for regular (rectangular) grids
- "iNetCDF" for irregular (e.g. geodesic) grids or station data

Parameters

- **file_name** (*str*) – The name of the data file.
- **file_type** (*str*) – The format of the data file.
- **observable_name** (*str*) – The short name of the observable within data file (particularly relevant for NetCDF).
- **dimension_names** (*dict*) – The names of the dimensions as used in the NetCDF file. E.g., dimension_names = {"lat": "lat", "lon": "lon", "time": "time"}.
- **vertical_level** (*int*) – The vertical level to be extracted from the data file. Is ignored for horizontal data sets. If None, the first level in the data file is chosen.
- **silence_level** (*int*) – The inverse level of verbosity of the object.

classmethod _load_data (*file_name*, *file_type*, *observable_name*, *dimension_names*, *vertical_level=None*, *silence_level=0*)
Load data into a Numpy array and create a corresponding Grid object.

Supported file types `file_type` are:

- "NetCDF" for regular (rectangular) grids
- "iNetCDF" for irregular (e.g. geodesic) grids or station data

Parameters

- **file_name** (*str*) – The name of the data file.
- **file_type** (*str*) – The format of the data file.
- **observable_name** (*str*) – The short name of the observable within data file (particularly relevant for NetCDF).
- **dimension_names** (*dict*) – The names of the dimensions as used in the NetCDF file. E.g., dimension_names = {"lat": "lat", "lon": "lon", "time": "time"}.
- **vertical_level** (*int*) – The vertical level to be extracted from the data file. Is ignored for horizontal data sets. If None, the first level in the data file is chosen.

- **silence_level** (*int*) – The inverse level of verbosity of the object.

clear_cache()

Clean up cache.

Is reversible, since all cached information can be recalculated from basic data.

static cos_window(*data*, *gamma*)

Return a cosine window fitting the shape of the data argument.

The window is one for most of the time and goes to zero at the boundaries of each time series in the data array.

The width of the cosine shaped decay region is controlled by the shape parameter gamma:

- Gamma=1 means, that each of the two decay regions extends over half of the time series.
- Gamma=0 means, that the decay regions vanish and the window transformation becomes the identity.

Example:

```
>>> ts = np.arange(24).reshape(12,2)
>>> Data.cos_window(data=ts, gamma=0.75)
array([[ 0.        ,  0.        ], [ 0.14644661,  0.14644661],
       [ 0.5       ,  0.5       ], [ 0.85355339,  0.85355339],
       [ 1.        ,  1.        ], [ 1.        ,  1.        ],
       [ 1.        ,  1.        ], [ 1.        ,  1.        ],
       [ 0.85355339,  0.85355339], [ 0.5       ,  0.5       ],
       [ 0.14644661,  0.14644661], [ 0.        ,  0.        ]])
```

Parameters

- **data** (*2D Numpy array [time, index]*) – The data array to be fitted by cosine window.
- **gamma** (*number (float)*) – The cosine window shape parameter.

Return type 2D Numpy array [time, index]

Returns the cosine window fitting data array.

grid = None

The *Grid* object associated with the data.

static next_power_2(*i*)

Return the power of two 2^n, that is greater or equal than i.

Example:

```
>>> Data.next_power_2(253)
256
```

Parameters **i** (*number (float)*) – Some real number.

Return type number (float)

Returns the power of two greater of equal than a given value.

static normalize_time_series_array(*time_series_array*)

Normalize an array of time series to zero mean and unit variance individually for each individual time series.

Works also for complex valued time series.

Modifies the given array in place!

Example:

```
>>> ts = np.arange(16).reshape(4,4).astype("float")
>>> Data.normalize_time_series_array(ts)
>>> ts.mean(axis=0)
array([ 0.,  0.,  0.,  0.])
>>> ts.std(axis=0)
array([ 1.,  1.,  1.,  1.])
>>> ts[:,0]
array([-1.34164079, -0.4472136 ,  0.4472136 ,  1.34164079])
```

Parameters **time_series_array** (*2D Numpy array [time, index]*) – The time series array to be normalized.

observable()

Return the current spatio-temporal view on the data.

Example:

```
>>> Data.SmallTestData().observable()[0,:]
array([  0.00000000e+00,   1.00000000e+00,   1.22464680e-16,
        -1.00000000e+00,  -2.44929360e-16,   1.00000000e+00])
```

Return type 2D Numpy array [time, space]

Returns the current spatio-temporal view on the data.

observable_long_name = None

(str) - The long name of the observable within data file.

observable_name = None

(str) - The short name of the observable within data file (particularly relevant for NetCDF).

print_data_info()

Print information on the data encapsulated by the Data object.

static rescale(*array*, *var_type*)

Rescale an array to a given data type.

Returns the tuple (scaled_array, scale_factor, add_offset, actual_range). Allows flexible handling of final amount of used storage volume for the file.

Parameters

- **array** –
- **var_type** (*str*) – Determines the desired final data type of the array.

set_global_window()

Set the view on the whole data set.

Select the full data set and creates a data array as well as a corresponding Grid object to access this window from outside.

Example (Set smaller window and subsequently restore global window):

```
>>> data = Data.SmallTestData()
>>> data.set_window(window={"time_min": 0., "time_max": 4.,
...                 "lat_min": 10., "lat_max": 20., "lon_min": 5.,
...                 "lon_max": 10.})
>>> data.grid.grid()["lat"]
array([ 10.,  15.], dtype=float32)
>>> data.set_global_window()
>>> data.grid.grid()["lat"]
array([  0.,   5.,  10.,  15.,  20.,  25.], dtype=float32)
```

set_silence_level(*silence_level*)
Set the silence level.

Includes dependent objects such as *grid*.

Parameters **silence_level** (*number (int)*) – The inverse level of verbosity of the object.

set_window(*window*)
Select a rectangular spatio-temporal region from the data set.

Create a data array as well as a corresponding Grid object to access this window.

The time axis of the underlying raw data is assumed to be ordered and increasing. The latitude and longitude sequences can be arbitrarily chosen, i.e., no ordering and no regular grid is required.

The spatio-temporal window is described by the following dictionary:

```
window = {"time_min": 0., "time_max": 0., "lat_min": 0.,
          "lat_max": 0., "lon_min": 0., "lon_max": 0.}
```

If the temporal boundaries are equal, the data's full time range is selected. If any of the two corresponding spatial boundaries are equal, the data's full spatial extension is included.

Example:

```
>>> data = Data.SmallTestData()
>>> data.set_window(window={
...     "time_min": 0., "time_max": 4., "lat_min": 10.,
...     "lat_max": 20., "lon_min": 5., "lon_max": 10.})
>>> data.observable()
array([[  1.22464680e-16,  -1.00000000e+00],
       [ -3.09016994e-01,  -9.51056516e-01],
       [ -5.87785252e-01,  -8.09016994e-01],
       [ -8.09016994e-01,  -5.87785252e-01],
       [ -9.51056516e-01,  -3.09016994e-01]])
```

Parameters **window** (*dictionary*) – A spatio-temporal window to select a view on the data.

silence_level = None
(int) - The inverse level of verbosity of the object.

window()
Return the current spatio-temporal window.

Examples:

```
>>> Data.SmallTestData().window()["lon_min"]
2.5
```

```
>>> Data.SmallTestData().window()["lon_max"]
15.0
```

Return type dictionary

Returns the current spatio-temporal window.

static **zero_pad_data**(*data*)
Return zero padded data, such that the length of individual time series is a power of 2.

Example:

```
>>> ts = np.arange(20).reshape(5,4)
>>> Data.zero_pad_data(ts)
array([[  0.,   0.,   0.,   0.], [  0.,   1.,   2.,   3.],
       [  4.,   5.,   6.,   7.], [  8.,   9.,  10.,  11.],
       [ 12.,  13.,  14.,  15.], [ 16.,  17.,  18.,  19.],
       [  0.,   0.,   0.,   0.], [  0.,   0.,   0.,   0.]])
```

Parameters **data** (*2D Numpy array [time, index]*) – The data array to be zero padded.

Return type 2D Numpy array [time, index]

Returns the zero padded data array.

5.1.2 core.geo_network

Provides classes for analyzing spatially embedded complex networks, handling multivariate data and generating time series surrogates.

class pyunicorn.core.geo_network.**GeoNetwork** (*grid*, *adjacency=None*, *edge_list=None*, *directed=False*, *node_weight_type='surface'*, *silence_level=0*)

Bases: *pyunicorn.core.network.Network*

Encapsulates a network embedded on a spherical surface.

Particularly adds more network measures and statistics based on the spatial embedding.

Variables **node_weight_type** – (string) - The type of geographical node weight to be used.

static BarabasiAlbert (*n_nodes*, *n_links*, *grid*, *node_weight_type='surface'*, *silence_level=0*)

Generates an undirected and spatially embedded Barabasi-Albert network.

Parameters

- **n_nodes** (*int*) – The number of nodes.
- **n_links** (*int*) – The number of links of the node that is added at each step of the growth process.
- **grid** (*Grid object*) – The Grid object describing the network's spatial embedding.
- **node_weight_type** (*str*) – The type of geographical node weight to be used (see *set_node_weight_type()*).
- **silence_level** (*int*) – The inverse level of verbosity of the object.

Return type *GeoNetwork*

Returns the Barabasi-Albert network.

static ConfigurationModel (*grid*, *degrees*, *node_weight_type='surface'*, *silence_level=0*)

Generates an undirected and spatially embedded configuration model graph.

The configuration model gives a fully random graph with a given degree sequence *degrees*.

Note: The configuration model network is simplified to eliminate self-loops and multiple edges. This results in a model degree sequence differing (slightly) from the original one. To fully conserve the degree sequence, distribution, link density etc., random rewiring should be used (*Network.randomly_rewire()*).

Example (Repeat creation of configuration model network from SmallTestNetwork until the number of links is the same as in the original network):

```
>>> n = 0
>>> while n != 7:
...     net = GeoNetwork.ConfigurationModel(
...         grid=Grid.SmallTestGrid(),
...         degrees=GeoNetwork.SmallTestNetwork().degree(),
...         silence_level=2)
...     n = net.n_links
>>> print net.link_density
0.466666666667
```

Parameters

- **degrees** (*1D array [index]*) – The original degree sequence.
- **grid** (*Grid object*) – The Grid object describing the network's spatial embedding.
- **node_weight_type** (*str*) – The type of geographical node weight to be used (see *set_node_weight_type()*).
- **silence_level** (*int*) – The inverse level of verbosity of the object.

Return type *GeoNetwork*

Returns the configuration model network.

static ErdosRenyi (*grid*, *n_nodes*, *link_probability=None*, *n_links=None*, *node_weight_type='surface'*, *silence_level=0*)

Generates an undirected and spatially embedded Erdos-Renyi random graph

Any pair of nodes is connected with probability p.

Example:

```
>>> print GeoNetwork.ErdosRenyi(
...     grid=Grid.SmallTestGrid(), n_nodes=6, n_links=5)
Generating Erdos-Renyi random graph with 6 nodes and 5 links...
Setting area weights according to type surface...
GeoNetwork:
Network: undirected, 6 nodes, 5 links, link density 0.333.
Geographical boundaries:
           time      lat      lon
   min     0.0     0.00     2.50
   max     9.0    25.00    15.00
```

Parameters

- **grid** (*Grid* object) – The *Grid* object describing the network's spatial embedding.
- **n_nodes** (*number > 0 (int)*) – Number of nodes.
- **link_probability** (*number from 0 to 1 (float), or None*) – If not None, each pair of nodes is independently linked with this probability. (Default: None)
- **n_links** (*number > 0 (int), or None*) – If not None, this many links are assigned at random. Must be None if link_probability is not None. (Default: None)
- **node_weight_type** (*str*) – The type of geographical node weight to be used (see *set_node_weight_type()*).
- **silence_level** (*int*) – The inverse level of verbosity of the object.

Return type *GeoNetwork*

Returns the Erdos-Renyi random graph.

static Load (*filename_network*, *filename_grid*, *fileformat=None*, *silence_level=0*, **args*, ***kwds*)

Return a GeoNetwork object stored in files.

Unified reading function for graphs. Relies on and partially extends the corresponding igraph function. Refer to igraph documentation for further details on the various reader methods for different formats.

This method tries to identify the format of the graph given in the first parameter and calls the corresponding reader method.

Existing node and link attributes/weights are also restored depending on the chosen file format. E.g., the formats GraphML and gzipped GraphML are able to store both node and link weights.

The remaining arguments are passed to the reader method without any changes.

Parameters

- **`filename_network`** (*str*) – The name of the file where the Network object is to be stored.
- **`filename_grid`** (*str*) – The name of the file where the Grid object is to be stored (including ending).
- **`fileformat`** (*str*) – the format of the file (if known in advance) `None` means auto-detection. Possible values are: `"ncol"` (NCOL format), `"lgl"` (LGL format), `"graphml"`, `"graphmlz"` (GraphML and gzipped GraphML format), `"gml"` (GML format), `"net"`, `"pajek"` (Pajek format), `"dimacs"` (DIMACS format), `"edgelist"`, `"edges"` or `"edge"` (edge list), `"adjacency"` (adjacency matrix), `"pickle"` (Python pickled format).
- **`silence_level`** (*int*) – The inverse level of verbosity of the object.

Return type GeoNetwork object

Returns `GeoNetwork` instance.

static **`SmallTestNetwork`**()

Return a 6-node undirected geographically embedded test network.

The test network consists of the SmallTestNetwork of the Network class with node coordinates given by the SmallTestGrid of the Grid class.

The network looks like this:

```
          3 - 1
          |   | \
      5 - 0 - 4 - 2
```

Return type GeoNetwork instance

Returns an instance of GeoNetwork for testing purposes.

`__init__`(*grid*, *adjacency=None*, *edge_list=None*, *directed=False*, *node_weight_type='surface'*, *silence_level=0*)

Initialize an instance of GeoNetwork.

Parameters

- **`grid`** (`Grid`) – The Grid object describing the network's spatial embedding.
- **`adjacency`** (*2D array (int8) [index, index]*) – The network's adjacency matrix.
- **`edge_list`** (*array-like list of lists*) – Edge list of the new network. Entries [i,0], [i,1] contain the end-nodes of an edge.
- **`directed`** (*bool*) – Determines, whether the network is treated as directed.
- **`node_weight_type`** (*str*) – The type of geographical node weight to be used.
- **`silence_level`** (*int*) – The inverse level of verbosity of the object.

Possible choices for `node_weight_type`:

- None (constant unit weights)
- "surface" (cos lat)
- "irrigation" ($\cos^2$ lat)

`__str__`()

Return a string representation of the GeoNetwork object.

`_calculate_general_average_link_distance`(*adjacency*, *degrees*, *geometry_corrected=False*)

Return general average link distances (ALD).

This general method is called to calculate undirected average link distance, average in-link distance and average out-link distance.

The resulting sequence can optionally be corrected for biases in average link distance arising due to the grid geometry. E.g., for regional networks, nodes on the boundaries may have a bias towards larger values of ALD, while nodes in the center have a bias towards smaller values of ALD.

Parameters

- **adjacency** (*2D array [index, index]*) – The adjacency matrix.
- **degrees** (*1D array [index]*) – The degree sequence.
- **geometry_corrected** (*bool*) – Toggles geometry correction.

Return type 1D array [index]

Returns the general average link distance sequence.

_calculate_general_connectivity_weighted_distance(*adjacency*, *degrees*)

Return general connectivity weighted link distances (CWD).

This method is called to calculate undirected CWD, in-CWD and out-CWD.

Parameters

- **adjacency** (*2D array [index, index]*) – The adjacency matrix.
- **degrees** (*1D array [index]*) – The degree sequence.

Return type 1D array [index]

Returns the general connectivity weighted distance sequence.

angular_distance(**args*, ***kwargs*)

Return the angular great circle distance matrix.

area_weighted_connectivity()

Return area weighted connectivity (AWC).

It gives the fractional area of the network, a node is connected to. AWC is closely related to node splitting invariant degree `Network.nsi_degree()` with area as node weight.

Example:

```
>>> r(GeoNetwork.SmallTestNetwork().area_weighted_connectivity())
array([ 0.4854,  0.499 ,  0.3342,  0.3446,  0.5146,  0.1726])
```

Return type 1D Numpy array [index]

Returns the area weighted connectivity sequence.

area_weighted_connectivity_cumulative_distribution(*n_bins*)

Return the cumulative area weighted connectivity distribution.

Also return estimated statistical error and lower bin boundaries.

Example:

```
>>> r(GeoNetwork.SmallTestNetwork().            area_weighted_connectivity_cumulative
...          n_bins=4)[0])
array([ 1.  ,  0.8435,  0.5068,  0.1622])
```

Parameters **n_bins** (*number (int)*) – The number of bins for histogram.

Return type tuple of three 1D Numpy arrays [bin]

Returns the cumulative AWC distribution, statistical error, and lower bin boundaries.

area_weighted_connectivity_distribution(*n_bins*)
Return the area weighted connectivity frequency distribution.

Also return estimated statistical error and lower bin boundaries.

Example:

```
>>> r(GeoNetwork.SmallTestNetwork().                area_weighted_connectivity_distributi
array([ 0.1565, 0.3367, 0.3446, 0.1622])
```

Parameters **n_bins** (*number (int)*) – The number of bins for histogram.

Return type tuple of three 1D Numpy arrays [bin]

Returns the AWC distribution, statistical error, and lower bin boundaries.

average_distance_weighted_path_length()
Return average distance weighted path length.

Returns the average path length link-weighted by the angular great circle distance between nodes.

Example:

```
>>> r(GeoNetwork.SmallTestNetwork().                average_distance_weighted_path_length
0.4985
```

Return type number (float)

Returns the average distance weighted path length.

average_link_distance(*geometry_corrected=False*)
Return average link distances (undirected).

Note: Does not use directionality information.

Examples:

```
>>> r(GeoNetwork.SmallTestNetwork().                average_link_distance(geometry_correc
array([ 0.3885, 0.1943, 0.1456, 0.2433, 0.2912, 0.4847])
>>> r(GeoNetwork.SmallTestNetwork().                average_link_distance(geometry_correc
array([ 1.5988, 1.0921, 1.0001, 1.6708, 1.6384, 2.0041])
```

Parameters **geometry_corrected** (*bool*) – Toggles geometry correction.

Return type 1D array [index]

Returns the average link distance sequence (undirected).

average_neighbor_area_weighted_connectivity()
Return average neighbor area weighted connectivity.

Note: Does not use directionality information.

Example:

```
>>> r(GeoNetwork.SmallTestNetwork().                average_neighbor_area_weighted_connec
array([ 0.3439, 0.3978, 0.5068, 0.4922, 0.4395, 0.4854])
```

Return type 1D Numpy array [index]

Returns the average neighbor area weighted connectivity sequence.

boundary (*nodes*, *geodesic=True*, *gap=0.0*)
Return a list of ordered lists of nodes on the connected parts of the boundary of a subset of nodes and a list of ordered lists of (lat,lon) coordinates of the corresponding polygons

- EXPERIMENTAL! *

clear_cache ()
Clean up cache.

Is reversible, since all cached information can be recalculated from basic data.

connectivity_weighted_distance ()
Return undirected connectivity weighted link distances (CWD).

Note: Does not use directionality information.

Example:

```
>>> r(GeoNetwork.SmallTestNetwork().            connectivity_weighted_distance())
array([ 0.0625, 0.0321, 0.0241, 0.0419, 0.05 , 0.0837])
```

Return type 1D Numpy array [index]

Returns the undirected connectivity weighted distance sequence.

distance_weighted_closeness ()
Return distance weighted closeness.

Returns the sequence of closeness centralities link-weighted by the angular great circle distance between nodes.

Example:

```
>>> r(GeoNetwork.SmallTestNetwork().            distance_weighted_closeness())
array([ 2.2378, 2.4501, 2.2396, 2.4501, 2.2396, 1.1982])
```

Return type 1D Numpy array [index]

Returns the distance weighted closeness sequence.

geographical_cumulative_distribution (*sequence*, *n_bins*)
Return a normalized geographical cumulative distribution.

Also return estimated statistical error and the lower bin boundaries.

This function counts which percentage of total surface area has a value of sequence larger or equal than the one bounded by a specific bin and NOT which number of nodes does so.

Note: Be aware that this method only returns meaningful results for regular rectangular grids, where the representative area of each node is proportional to the cosine of its latitude.

Example:

```
>>> net = GeoNetwork.SmallTestNetwork()
>>> r(net.geographical_cumulative_distribution(
...     sequence=net.degree(), n_bins=3)[0])
array([ 1. , 0.8435, 0.5068])
```

Parameters

- **sequence** (*1D Numpy array [index]*) – The input sequence (e.g., some local network measure).
- **n_bins** (*number (int)*) – The number of bins for histogram.

Return type tuple of three 1D Numpy arrays [bin]

Returns the cumulative geographical distribution, statistical error, and lower bin boundaries.

geographical_distribution(*sequence*, *n_bins*)

Return a normalized geographical frequency distribution.

Also return the estimated statistical error and lower bin boundaries.

This function counts which percentage of total surface area falls into a bin and NOT which number of nodes does so.

Note: Be aware that this method only returns meaningful results for regular rectangular grids, where the representative area of each node is proportional to the cosine of its latitude.

Example:

```
>>> net = GeoNetwork.SmallTestNetwork()
>>> r(net.geographical_distribution(
...     sequence=net.degree(), n_bins=3)[0])
array([ 0.1565, 0.3367, 0.5068])
```

Parameters

- **sequence** (*1D Numpy array [index]*) – The input sequence (e.g., some local network measure).
- **n_bins** (*number (int)*) – The number of bins for histogram.

Return type tuple of three 1D Numpy arrays [bin]

Returns the geographical distribution, statistical error, and lower bin boundaries.

grid = None

(Grid) - Grid object describing the network's spatial embedding

inarea_weighted_connectivity()

Return in-area weighted connectivity.

It gives the fractional area of the netwerk that connects to a given node. For undirected networks, it calculates total area weighted connectivity.

Example:

```
>>> r(GeoNetwork.SmallTestNetwork().inarea_weighted_connectivity())
array([ 0.4854, 0.499 , 0.3342, 0.3446, 0.5146, 0.1726])
```

Return type 1D Numpy array [index]

Returns the in-area weighted connectivity sequence.

inarea_weighted_connectivity_cumulative_distribution(*n_bins*)

Return the cumulative in-area weighted connectivity distribution.

Also return estimated statistical error and lower bin boundaries.

Example:

```
>>> r(GeoNetwork.SmallTestNetwork().            inarea_weighted_connectivity_cumulati
...         n_bins=4)[0])
array([ 1. , 0.8435, 0.5068, 0.1622])
```

Parameters **n_bins** (*number (int)*) – The number of bins for histogram.

Return type tuple of three 1D Numpy arrays [bin]

Returns the cumulative in-AWC distribution, statistical error, and lower bin boundaries.

inarea_weighted_connectivity_distribution(*n_bins*)
Return the in-area weighted connectivity frequency distribution.

Also return estimated statistical error and lower bin boundaries.

Example:

```
>>> r(GeoNetwork.SmallTestNetwork().                inarea_weighted_connectivity_distribu
array([ 0.1565, 0.3367, 0.3446, 0.1622])
```

Parameters **n_bins** (*number (int)*) – The number of bins for histogram.

Return type tuple of three 1D Numpy arrays [bin]

Returns the in-AWC distribution, statistical error, and lower bin boundaries.

inaverage_link_distance(*geometry_corrected=False*)
Return in-average link distances.

Return regular average link distance for undirected networks.

Example:

```
>>> r(GeoNetwork.SmallTestNetwork().                inaverage_link_distance(geometry_corr
array([ 0.3885, 0.1943, 0.1456, 0.2433, 0.2912, 0.4847])
```

Parameters **geometry_corrected** (*bool*) – Toggles geometry correction.

Return type 1D array [index]

Returns the in-average link distance sequence.

inconnectivity_weighted_distance()
Return in-connectivity weighted link distances (CWD).

Example:

```
>>> r(GeoNetwork.SmallTestNetwork().                inconnectivity_weighted_distance())
array([ 0.0625, 0.0321, 0.0241, 0.0419, 0.05 , 0.0837])
```

Return type 1D Numpy array [index]

Returns the in-connectivity weighted distance sequence.

intotal_link_distance(*geometry_corrected=False*)
Return the sequence of in-total link distances for all nodes.

Example:

```
>>> r(GeoNetwork.SmallTestNetwork().                intotal_link_distance(geometry_correc
array([ 0.1886, 0.097 , 0.0486, 0.0838, 0.1498, 0.0837])
```

Parameters **geometry_corrected** (*bool*) – Toggles geometry correction.

Return type 1D array [index]

Returns the in-total link distance sequence.

link_distance_distribution(*n_bins*, *grid_type='spherical'*, *geometry_corrected=False*)
Return the normalized link distance distribution.

Correct for the geometry of the embedding space by default.

Examples:

```
>>> GeoNetwork.SmallTestNetwork().link_distance_distribution(
...     n_bins=4, geometry_corrected=False)[0]
array([ 0.14285714,  0.28571429,  0.28571429,  0.28571429])
>>> GeoNetwork.SmallTestNetwork().link_distance_distribution(
...     n_bins=4, geometry_corrected=True)[0]
array([ 0.09836066,  0.24590164,  0.32786885,  0.32786885])
```

Parameters

- **n_bins** (*int*) – The number of bins for histogram.
- **grid_type** (*str*) – Type of grid, used for distance calculation, can take values "euclidean" and "spherical".
- **geometry_corrected** (*bool*) – Toggles correction for grid geometry.

Return type tuple of three 1D arrays [bin]

Returns the link distance distribution, statistical error, and lower bin boundaries.

local_distance_weighted_vulnerability()

Return local distance weighted vulnerability.

Return the sequence of vulnerabilities link-weighted by the angular great circle distance between nodes.

Example:

```
>>> r(GeoNetwork.SmallTestNetwork().                local_distance_weighted_vulnerability
array([ 0.0325, 0.3137, 0.2056, 0.028 , -0.0283, -0.288 ])
```

Return type 1D Numpy array [index]

Returns the local distance weighted vulnerability sequence.

local_geographical_clustering()

Return local geographical clustering.

Returns the sequence of local clustering coefficients weighted by the inverse angular great circle distance between nodes. This guarantees, that short links between spatially neighboring nodes in a triangle are weighted higher than long links between nodes that are spatially far away.

Uses a definition of weighted clustering coefficient introduced in *[Holme2007]*.

Note: Experimental measure!

Example:

```
>>> r(GeoNetwork.SmallTestNetwork().local_geographical_clustering())
Calculating local weighted clustering coefficient...
array([ 0. , 0.0998, 0.1489, 0. , 0.2842, 0. ])
```

Return type 1D Numpy array (index)

Returns the local geographical clustering sequence.

local_tsonis_clustering()

Return local Tsonis clustering.

This measure of local clustering was introduced in *[Tsonis2008a]*.

Return type 1D Numpy array (index)

Returns the local Tsonis clustering sequence.

max_link_distance()

Return maximum angular geodesic link distances.

Note: Does not use directionality information.

Example:

```
>>> r(GeoNetwork.SmallTestNetwork().max_link_distance())
array([ 0.4847, 0.2911, 0.1938, 0.292 , 0.3887, 0.4847])
```

Return type 1D Numpy array [index]

Returns the maximum link distance sequence.

max_neighbor_area_weighted_connectivity()

Return maximum neighbor area weighted connectivity.

Note: Does not use directionality information.

```
>>> r(GeoNetwork.SmallTestNetwork().                max_neighbor_area_weighted_connectivi
array([ 0.5146, 0.5146, 0.5146, 0.499 , 0.499 , 0.4854])
```

Return type 1D Numpy array [index]

Returns the maximum neighbor area weighted connectivity sequence.

nsi_connected_hamming_cluster_tree(*lon_closed=True*, *lat_closed=False*, *alpha=0.01*)

Perform NSI agglomerative clustering.

Minimize in each step the Hamming distance between the original and the clustered network, but only joins connected clusters.

Return c,h where c[i,j] = i iff node j is in cluster no. i, and 0 otherwise, and h is the corresponding list of total resulting relative Hamming distance between 0 and 1. The cluster numbers for all nodes and a k clusters solution is then c[:2*N-k,:].max(axis=0)

Parameters

- **lon_closed** (*bool*) – TODO
- **lat_closed** (*bool*) – TODO
- **alpha** (*float*) – TODO

Return type TODO

Returns TODO

outarea_weighted_connectivity()

Return out-area weighted connectivity.

It gives the fractional area of the netwerk that a given node connects to. For undirected networks, it calculates total area weighted connectivity.

Example:

```
>>> r(GeoNetwork.SmallTestNetwork().                outarea_weighted_connectivity())
array([ 0.4854, 0.499 , 0.3342, 0.3446, 0.5146, 0.1726])
```

Return type 1D Numpy array [index]

Returns the out-area weighted connectivity sequence.

outarea_weighted_connectivity_cumulative_distribution(*n_bins*)
Return the cumulative out-area weighted connectivity distribution.

Also return estimated statistical error and lower bin boundaries.

Example:

```
>>> r(GeoNetwork.SmallTestNetwork().                outarea_weighted_connectivity_cumulat
...          n_bins=4)[0])
array([ 1. , 0.8435, 0.5068, 0.1622])
```

Parameters **n_bins** (*number (int)*) – The number of bins for histogram.

Return type tuple of three 1D Numpy arrays [bin]

Returns the cumulative out-AWC distribution, statistical error, and lower bin boundaries.

outarea_weighted_connectivity_distribution(*n_bins*)
Return the out-area weighted connectivity frequency distribution.

Also return estimated statistical error and lower bin boundaries.

Example:

```
>>> r(GeoNetwork.SmallTestNetwork().                outarea_weighted_connectivity_distrib
array([ 0.1565, 0.3367, 0.3446, 0.1622])
```

Parameters **n_bins** (*number (int)*) – The number of bins for histogram.

Return type tuple of three 1D Numpy arrays [bin]

Returns the out-AWC distribution, statistical error, and lower bin boundaries.

outaverage_link_distance(*geometry_corrected=False*)
Return out-average link distances.

Return regular average link distance for undirected networks.

Example:

```
>>> r(GeoNetwork.SmallTestNetwork().                outaverage_link_distance(geometry_cor
array([ 0.3885, 0.1943, 0.1456, 0.2433, 0.2912, 0.4847])
```

Parameters **geometry_corrected** (*bool*) – Toggles geometry correction.

Return type 1D array [index]

Returns the out-average link distance sequence.

outconnectivity_weighted_distance()
Return out-connectivity weighted link distances (CWD).

Example:

```
>>> r(GeoNetwork.SmallTestNetwork().                outconnectivity_weighted_distance())
array([ 0.0625, 0.0321, 0.0241, 0.0419, 0.05 , 0.0837])
```

Return type 1D Numpy array [index]

Returns the out-connectivity weighted distance sequence.

outtotal_link_distance(*geometry_corrected=False*)
Return the sequence of out-total link distances for all nodes.

Example:

```
>>> r(GeoNetwork.SmallTestNetwork().            outtotal_link_distance(geometry_corre
array([ 0.1886, 0.097 , 0.0486, 0.0838, 0.1498, 0.0837])
```

Parameters **`geometry_corrected`** (*bool*) – Toggles geometry correction.

Return type 1D array [index]

Returns the out-total link distance sequence.

`randomly_rewire_geomodel_I` (*distance_matrix*, *iterations*, *inaccuracy*)
Randomly rewire the current network in place using geographical model I.

Geographical model I preserves the degree sequence (exactly) and the link distance distribution $p(l)$ (approximately).

A higher `inaccuracy` in the conservation of $p(l)$ will lead to

•less deterministic links in the network and, hence,

•more degrees of freedom for the random graph and

•a shorter runtime of the algorithm, since more pairs of nodes eligible for rewiring can be found.

Example (The degree sequence should be the same after rewiring):

```
>>> net = GeoNetwork.SmallTestNetwork()
>>> net.randomly_rewire_geomodel_I(
...     distance_matrix=net.grid.angular_distance(),
...     iterations=100, inaccuracy=1.0)
>>> net.degree()
array([3, 3, 2, 2, 3, 1])
```

Parameters

- **`distance_matrix`** (*2D Numpy array [index, index]*) – Suitable distance matrix between nodes.
- **`iterations`** (*number (int)*) – The number of rewirings to be performed.
- **`inaccuracy`** (*number (float)*) – The inaccuracy with which to conserve $p(l)$.

`randomly_rewire_geomodel_II` (*distance_matrix*, *iterations*, *inaccuracy*)
Randomly rewire the current network in place using geographical model II.

Geographical model II preserves the degree sequence k_v (exactly), the link distance distribution $p(l)$ (approximately), and the average link distance sequence $< l >_v$ (approximately).

A higher `inaccuracy` in the conservation of $p(l)$ and $< l >_v$ will lead to:

•less deterministic links in the network and, hence,

•more degrees of freedom for the random graph and

•a shorter runtime of the algorithm, since more pairs of nodes eligible for rewiring can be found.

Parameters

- **`distance_matrix`** (*2D Numpy array [index, index]*) – Suitable distance matrix between nodes.
- **`iterations`** (*number (int)*) – The number of rewirings to be performed.
- **`inaccuracy`** (*number (float)*) – The inaccuracy with which to conserve $p(l)$.

`randomly_rewire_geomodel_III` (*distance_matrix*, *iterations*, *inaccuracy*)
Randomly rewire the current network in place using geographical model III.

Geographical model III preserves the degree sequence k_v (exactly), the link distance distribution $p(l)$ (approximately), and the average link distance sequence $< l >_v$ (approximately). Moreover, degree-degree correlations are also conserved exactly.

A higher `inaccuracy` in the conservation of $p(l)$ and $< l >_v$ will lead to:

•less deterministic links in the network and, hence,

•more degrees of freedom for the random graph and

•a shorter runtime of the algorithm, since more pairs of nodes eligible for rewiring can be found.

Parameters

- **`distance_matrix`** (*2D Numpy array [index, index]*) – Suitable distance matrix between nodes.
- **`iterations`** (*number (int)*) – The number of rewirings to be performed.
- **`inaccuracy`** (*number (float)*) – The inaccuracy with which to conserve $p(l)$.

`save` (*filename_network*, *filename_grid=None*, *fileformat=None*, **args*, ***kwds*)

Save the GeoNetwork object to files.

Unified writing function for graphs. Relies on and partially extends the corresponding igraph function. Refer to igraph documentation for further details on the various writer methods for different formats.

This method tries to identify the format of the graph given in the first parameter (based on extension) and calls the corresponding writer method.

Existing node and link attributes/weights are also stored depending on the chosen file format. E.g., the formats GraphML and gzipped GraphML are able to store both node and link weights.

The grid is not stored if the corresponding filename is None.

The remaining arguments are passed to the writer method without any changes.

Parameters

- **`filename_network`** (*str*) – The name of the file where the Network object is to be stored.
- **`filename_grid`** (*str*) – The name of the file where the Grid object is to be stored (including ending).
- **`fileformat`** (*str*) – the format of the file (if one wants to override the format determined from the filename extension, or the filename itself is a stream). `None` means auto-detection. Possible values are: `"ncol"` (NCOL format), `"lgl"` (LGL format), `"graphml"`, `"graphmlz"` (GraphML and gzipped GraphML format), `"gml"` (GML format), `"dot"`, `"graphviz"` (DOT format, used by GraphViz), `"net"`, `"pajek"` (Pajek format), `"dimacs"` (DIMACS format), `"edgelist"`, `"edges"` or `"edge"` (edge list), `"adjacency"` (adjacency matrix), `"pickle"` (Python pickled format), `"svg"` (Scalable Vector Graphics).

`save_for_cgv` (*filename*, *fileformat='graphml'*)

Save the GeoNetwork and its attributes for the CGV visualization software.

The node coordinates are stored as node attributes by default, likewise angular link distances are stored as edge attributes by default. All additional node and link properties are also stored for visualization.

This format is intended for being used by the spatial graph visualization software CGV developed in Rostock (contact Thomas Nocke, nocke@pik-potsdam.de (nocke@pik-potsdam.de)). By default, the file includes the latitude and longitude vectors as node properties, as well as the geodesic angular distance as an link property.

Parameters

- **`file_name`** (*str*) – The file name should end with ".dot" or ".gml".

- **fileformat** (*str*) – The file format: "graphml" - GraphML format "graphmlz" - gzipped GraphML format "graphviz" - GraphViz format

set_node_weight_type (*node_weight_type*)
Set node weights for calculation of n.s.i. measures according to requested type.

Possible choices for node_weight_type:

- None (constant unit weights)
- "surface" (cos lat)
- "irrigation" ($\cos^2$ lat)

Parameters node_weight_type (*str*) – The type of geographical node weight to be used.

set_random_links_by_distance (*a, b*)
Reassign links independently with link probability = $exp(a + b * angulardistance)$.

Note: Modifies network in place, creates an undirected network!

Example (Repeat until a network with 5 links is created):

```
>>> net = GeoNetwork.SmallTestNetwork()
>>> while (net.n_links != 5):
...     net.set_random_links_by_distance(a=0., b=-4.)
>>> net.n_links
5
```

Parameters

- **a** (*number (float)*) – The a parameter.
- **b** (*number (float)*) – The b parameter.

shuffled_by_distance_copy ()
Return a copy of the network where all links in each node-distance class have been randomly re-assigned.

In other words, the result is a random network in which the link probability only depends on the nodes' distance and is the same as in the original network.

Return type *GeoNetwork*

Returns the distance shuffled copy.

total_link_distance (*geometry_corrected=False*)
Return the sequence of total link distances for all nodes.

Note: Does not use directionality information.

Example:

```
>>> r(GeoNetwork.SmallTestNetwork().          total_link_distance(geometry_correcte
array([ 0.1886, 0.097 , 0.0486, 0.0838, 0.1498, 0.0837])
```

Parameters geometry_corrected (*bool*) – Toggles geometry correction.

Return type 1D array [index]

Returns the total link distance sequence.

5.1.3 core.grid

Provides classes for analyzing spatially embedded complex networks, handling multivariate data and generating time series surrogates.

class `pyunicorn.core.grid.`**`Grid`**(*time_seq*, *lat_seq*, *lon_seq*, *silence_level=0*)

Bases: `object`

Encapsulates a horizontal spatio-temporal grid on the sphere.

The spatial grid points can be arbitrarily distributed, which is useful for representing station data or geodesic grids.

static **`Load`**(*filename*)

Return a Grid object stored in a cPickle file.

Parameters **`filename`** (*str*) – The name of the file where Grid object is stored (including ending).

Return type Grid object

Returns `Grid` instance.

static **`LoadTXT`**(*filename*)

Return a Grid object stored in text files.

The latitude, longitude and time sequences are loaded from three separate text files.

Parameters **`filename`** (*str*) – The name of the files where Grid object is stored (excluding endings).

Return type Grid object

Returns `Grid` instance.

`N` **= None**

(number (int)) - The number of spatial grid points / nodes.

static **`RegularGrid`**(*time_seq*, *lat_grid*, *lon_grid*, *silence_level=0*)

Initialize an instance of a regular grid.

Examples:

```
>>> Grid.RegularGrid(
...      time_seq=np.arange(2), lat_grid=np.array([0.,5.]),
...      lon_grid=np.array([1.,2.]), silence_level=2).lat_sequence()
array([ 0.,   0.,   5.,   5.], dtype=float32)
>>> Grid.RegularGrid(
...      time_seq=np.arange(2), lat_grid=np.array([0.,5.]),
...      lon_grid=np.array([1.,2.]), silence_level=2).lon_sequence()
array([ 1.,   2.,   1.,   2.], dtype=float32)
```

Parameters

- **`time_seq`** (*1D Numpy array [time]*) – The increasing sequence of temporal sampling points.
- **`lat_grid`** (*1D Numpy array [n_lat]*) – The latitudinal grid.
- **`lon_grid`** (*1D Numpy array [n_lon]*) – The longitudinal grid.
- **`silence_level`** (*number (int)*) – The inverse level of verbosity of the object.

Return type Grid object

Returns `Grid` instance.

static **`SmallTestGrid`**()

Return test grid of 6 spatial grid points with 10 temporal sampling points each.

Return type Grid instance

Returns a Grid instance for testing purposes.

`__init__`(*time_seq*, *lat_seq*, *lon_seq*, *silence_level=0*)
Initialize an instance of Grid.

Parameters

- **`time_seq`** (*1D Numpy array [time]*) – The increasing sequence of temporal sampling points.
- **`lat_seq`** (*1D Numpy array [index]*) – The sequence of latitudinal sampling points.
- **`lon_seq`** (*1D Numpy array [index]*) – The sequence of longitudinal sampling points.
- **`silence_level`** (*number (int)*) – The inverse level of verbosity of the object.

`__str__`()
Return a string representation of the Grid object.

`__weakref__`
list of weak references to the object (if defined)

`_calculate_angular_distance`()
Calculate and return the angular great circle distance matrix.

No normalization applied anymore! Return values are in the range 0 to Pi.

Return type 2D Numpy array [index, index]

Returns the angular great circle distance matrix (unit radians).

`angular_distance`()
Return the angular great circle distance matrix.

No normalization applied anymore! Return values are in the range 0 to Pi.

Example:

```
>>> rr(Grid.SmallTestGrid().angular_distance())
[['0'      '0.0975' '0.1949' '0.292'  '0.3887' '0.4847']
 ['0.0975' '0'      '0.0974' '0.1945' '0.2911' '0.3872']
 ['0.1949' '0.0974' '0.0003' '0.0971' '0.1938' '0.2898']
 ['0.292'  '0.1945' '0.0971' '0.0005' '0.0967' '0.1928']
 ['0.3887' '0.2911' '0.1938' '0.0967' '0'      '0.0961']
 ['0.4847' '0.3872' '0.2898' '0.1928' '0.0961' '0.0005']]
```

Return type 2D Numpy array [index, index]

Returns the angular great circle distance matrix.

`boundaries`()
Return the spatio-temporal grid boundaries.

Structure of the returned dictionary:

- **self._boundaries = {"time_min": time_seq.min(),** "time_max": time_seq.max(), "lat_min": lat_seq.min(), "lat_max": lat_seq.max(), "lon_min": lon_seq.min(), "lon_max": lon_seq.max()}

Example:

```
>>> print Grid.SmallTestGrid().print_boundaries()
         time      lat      lon
   min    0.0     0.00     2.50
   max    9.0    25.00    15.00
```

Return type dictionary

Returns the spatio-temporal grid boundaries.

clear_cache()
Clean up cache.

Is reversible, since all cached information can be recalculated from basic data.

convert_lon_coordinates(*lon_seq*)
Return longitude coordinates in the system -180 deg W <= lon <= +180 deg O for all nodes.

Accepts longitude coordinates in the system 0 deg <= lon <= 360 deg. 0 deg corresponds to Greenwich, England.

Example:

```
>>> Grid.SmallTestGrid().convert_lon_coordinates(
...     np.array([10.,350.,20.,340.,170.,190.]))
array([  10.,  -10.,   20.,  -20.,  170., -170.])
```

Parameters **lon_seq** (*1D Numpy array [index]*) – Sequence of longitude coordinates.

Return type 1D Numpy array [index]

Returns the converted longitude coordinates for all nodes.

static **coord_sequence_from_rect_grid**(*lat_grid*, *lon_grid*)
Return the sequences of latitude and longitude for a regular and rectangular grid.

Example:

```
>>> Grid.coord_sequence_from_rect_grid(
...     lat_grid=np.array([0.,5.]), lon_grid=np.array([1.,2.]))
(array([ 0.,  0.,  5.,  5.]), array([ 1.,  2.,  1.,  2.]))
```

Parameters

- **lat_grid** (*1D Numpy array [lat]*) – The grid's latitudinal sampling points.
- **lon_grid** (*1D Numpy array [lon]*) – The grid's longitudinal sampling points.

Return type tuple of two 1D Numpy arrays [index]

Returns the coordinates of all nodes in the grid.

cos_lat()
Return the sequence of cosines of latitude for all nodes.

Example:

```
>>> r(Grid.SmallTestGrid().cos_lat()[:2])
array([ 1. , 0.9962])
```

Return type 1D Numpy array [index]

Returns the cosine of latitudes for all nodes.

cos_lon()
Return the sequence of cosines of longitude for all nodes.

Example:

```
>>> r(Grid.SmallTestGrid().cos_lon()[:2])
array([ 0.999 , 0.9962])
```

Return type 1D Numpy array [index]

Returns the cosine of longitudes for all nodes.

euclidean_distance()

Return the euclidean distance matrix between grid points.

So far assumes that the given latitude and longitude coordinates are planar, euclidean coordinates. Approximates great circle distance well for points with a separation much smaller than the Earth's radius.

Return type 2D Numpy array [index, index]

Returns the euclidean distance matrix.

geometric_distance_distribution(*n_bins*)

Return the distribution of angular great circle distances between all pairs of grid points.

Examples:

```
>>> r(Grid.SmallTestGrid().geometric_distance_distribution(3)[0])
array([ 0.3333,  0.4667,  0.2  ])
>>> r(Grid.SmallTestGrid().geometric_distance_distribution(3)[1])
array([ 0.  ,  0.1616,  0.3231,  0.4847])
```

Parameters **n_bins** (*number (int)*) – The number of histogram bins.

Return type tuple of two 1D Numpy arrays [bin]

Returns the normalized histogram and lower bin boundaries of angular great circle distances.

grid()

Return the grid's spatio-temporal sampling points.

Structure of the returned dictionary:

- **self._grid = {"time": time_seq.astype("float32"),** "lat": lat_seq.astype("float32"), "lon": lon_seq.astype("float32")}

Examples:

```
>>> Grid.SmallTestGrid().grid()["lat"]
array([  0.,   5.,  10.,  15.,  20.,  25.], dtype=float32)
>>> Grid.SmallTestGrid().grid()["lon"][5]
15.0
```

Return type dictionary

Returns the grid's spatio-temporal sampling points.

grid_size()

Return the sizes of the grid's spatial and temporal dimensions.

Structure of the returned dictionary:

- **self._grid_size = {"time": len(time_seq),** "space": len(lat_seq)}

Example:

```
>>> print Grid.SmallTestGrid().print_grid_size()
   space    time
       6      10
```

Return type dictionary

Returns the sizes of the grid's spatial and temporal dimensions.

lat_sequence()

Return the sequence of latitudes for all nodes.

Example:

```
>>> Grid.SmallTestGrid().lat_sequence()
array([  0.,    5.,  10.,  15.,  20.,  25.], dtype=float32)
```

Return type 1D Numpy array [index]

Returns the sequence of latitudes for all nodes.

lon_sequence()

Return the sequence of longitudes for all nodes.

Example:

```
>>> Grid.SmallTestGrid().lon_sequence()
array([  2.5,   5. ,   7.5,  10. ,  12.5,  15. ], dtype=float32)
```

Return type 1D Numpy array [index]

Returns the sequence of longitudes for all nodes.

n_grid_points = None

(number (int)) - The total number of data points / samples.

node_coordinates(*index*)

Return the geographical latitude and longitude of node `index`.

Example:

```
>>> Grid.SmallTestGrid().node_coordinates(3)
(15.0, 10.0)
```

Parameters **index** (*number (int)*) – The node index as used in node sequences.

Return type tuple of number (float)

Returns the node's latitude and longitude coordinates.

node_number(*lat_node*, *lon_node*)

Return the index of the closest node given geographical coordinates.

Example:

```
>>> Grid.SmallTestGrid().node_number(lat_node=14., lon_node=9.)
3
```

Parameters

- **lat_node** (*number (float)*) – The latitude coordinate.
- **lon_node** (*number (float)*) – The longitude coordinate.

Return type number (int)

Returns the closest node's index.

print_boundaries()

Pretty print the spatio-temporal grid boundaries.

print_grid_size()

Pretty print the sizes of the grid's spatial and temporal dimensions.

static region(*name*)

Return some standard regions.

region_indices(*region*)

Returns a boolean array of nodes with True values when the node is inside the region.

Example:

```
>>> Grid.SmallTestGrid().region_indices(
...     np.array([0.,0.,0.,11.,11.,11.,11.,0.]))
array([False,  True,  True, False, False, False], dtype=bool)
```

Parameters **region** (*1D Numpy array [n_polygon_nodes]*) – array of lon, lat, lon, lat, ... [-80.2, 5., -82.4, 5.3, ...] as copied from Google Earth Polygon file

Return type 1D bool array [index]

Returns bool array with True for nodes inside region

save(*filename*)

Save the Grid object to a cPickle file.

Parameters **filename** (*str*) – The name of the file where Grid object is stored (including ending).

save_txt(*filename*)

Save the Grid object to text files.

The latitude, longitude and time sequences are stored in three separate text files.

Parameters **filename** (*str*) – The name of the files where Grid object is stored (excluding ending).

silence_level = None

(number (int)) - The inverse level of verbosity of the object.

sin_lat()

Return the sequence of sines of latitude for all nodes.

Example:

```
>>> r(Grid.SmallTestGrid().sin_lat()[:2])
array([ 0.  ,  0.0872])
```

Return type 1D Numpy array [index]

Returns the sine of latitudes for all nodes.

sin_lon()

Return the sequence of sines of longitude for all nodes.

Example:

```
>>> r(Grid.SmallTestGrid().sin_lon()[:2])
array([ 0.0436,  0.0872])
```

Return type 1D Numpy array [index]

Returns the sine of longitudes for all nodes.

5.1.4 core.interacting_networks

Provides classes for analyzing spatially embedded complex networks, handling multivariate data and generating time series surrogates.

class `pyunicorn.core.interacting_networks.`**`InteractingNetworks`**(*adjacency*, *directed=False*, *node_weights=None*, *silence_level=0*)

Bases: *`pyunicorn.core.network.Network`*

Encapsulates an ensemble of interacting networks.

Provides measures to analyze the interaction topology of different pairs of subnetworks (groups of vertices).

So far, most methods only give meaningful results for undirected networks!

The idea of interacting networks and measures for their analysis are described in *[Donges2011a]*.

Consistently node-weighted measures for interacting network topologies are derived, described and applied in *[Wiedermann2011]*.

static **`RandomlyRewireCrossLinks`**(*network*, *node_list1*, *node_list2*, *swaps*)

Randomize the cross links between two subnetworks under preservation of cross degree centrality of both subnetworks.

Chooses randomly two cross links and swaps their ending points in subnetwork 2.

Implementation:

Stores the coordinates of the "1"-entries of the cross adjacency matrix in a tuple. Chooses randomly two entries of the tuple (ergo two cross links) allowing for the constraints that

1. the chosen links have distinct starting points in subnetwork 1 and distinct ending points in subnetwork 2
2. there do not exist intermediate links such that starting point of link 1 is connected to ending point of link 2 and vice versa.

[In case two links have the same starting point or / and the same ending point, condition (2) is never satisfied. Therefore only condition (2) is implemented.]

Swaps the ending points of the links in subnetwork 2 and overwrites the coordinates of the initial links in the tuple. The number of permutation procedures is determined by the "swaps" argument and the initial number of cross links. Creates a new adjacency matrix out of the altered tuple of coordinates.

Example (Degree and cross degree sequences should be the same after rewiring):

```
>>> net = InteractingNetworks.SmallTestNetwork()
>>> print "Degree:", net.degree()
Degree: [3 3 2 2 3 1]
>>> print "Cross degree:", net.cross_degree(
...     node_list1=[0,3,5], node_list2=[1,2,4])
Cross degree: [1 1 0]
>>> rewired_net = net.RandomlyRewireCrossLinks(
...     network=net, node_list1=[0,3,5],
...     node_list2=[1,2,4], swaps=10.)
>>> print "Degree:", rewired_net.degree()
Degree: [3 3 2 2 3 1]
>>> print "Cross degree:", rewired_net.cross_degree(
...     node_list1=[0,3,5], node_list2=[1,2,4])
Cross degree: [1 1 0]
```

Parameters

- **`network`** (*`InteractingNetworks`* instance) – The base network for rewiring cross links.
- **`node_list1`** (*[int]*) – list of node indices describing the first subnetwork
- **`node_list2`** (*[int]*) – list of node indices describing the second subnetwork
- **`internal`** (*float*) – Gives the fraction number_swaps / number_cross_links.

Return type *InteractingNetworks*

Returns The initial InteractingNetworks with swapped cross links

static RandomlySetCrossLinks(*network*, *node_list1*, *node_list2*, *cross_link_density=None*, *number_cross_links=None*)

Creates a set of random cross links between the considered interacting subnetworks. The number of cross links to be set can be chosen either explicitly or via a predefined cross link density. By not choosing any of either, a null model is created under preservation of the cross link density of the initial network.

Implementation:

Determines the number of cross links to be set. Creates an empty cross adjacency matrix. Randomly picks the coordinates of an entry and sets it to one. Repeats the procedure until the desired cross link density is reached.

Parameters

- **network** (*InteractingNetworks instance*) – The base network for setting random cross links.
- **node_list1** (*[int]*) – list of node indices describing the first subnetwork
- **node_list2** (*[int]*) – list of node indices describing the second subnetwork

Return type *InteractingNetworks*

Returns The initial InteractingNetworks with random cross links

static RandomlySetCrossLinks_sparse(*network*, *node_list1*, *node_list2*, *cross_link_density=None*, *number_cross_links=None*)

Creates a set of random cross links between the considered interacting subnetworks. The number of cross links to be set can be chosen either explicitly or via a predefined cross link density. By not choosing any of either, a null model is created under preservation of the cross link density of the initial network.

Implementation:

Determines the number of cross links to be set. Creates an empty cross adjacency matrix. Randomly picks the coordinates of an entry and sets it to one. Repeats the procedure until the desired cross link density is reached.

Parameters

- **network** (*InteractingNetworks instance*) – The base network for setting random cross links.
- **node_list1** (*[int]*) – list of node indices describing the first subnetwork
- **node_list2** (*[int]*) – list of node indices describing the second subnetwork

Return type *InteractingNetworks*

Returns The initial InteractingNetworks with random cross links

static SmallTestNetwork()

Return a 6-node undirected test network.

The network looks like this:

```
    3 - 1
    |   | \
5 - 0 - 4 - 2
```

Return type InteractingNetworks instance

Returns an InteractingNetworks instance for testing purposes.

__init__(*adjacency*, *directed=False*, *node_weights=None*, *silence_level=0*)

Initialize an instance of InteractingNetworks.

Parameters

- **adjacency** (*square numpy array or list [node,node] of 0s and 1s*) – Adjacency matrix of the new network. Entry [i,j] indicates whether node i links to node j. Its diagonal must be zero. Must be symmetric if directed=False.
- **directed** (*bool*) – Indicates whether the network shall be considered as directed. If False, adjacency must be symmetric.
- **node_weights** (*1d numpy array or list [node] of floats >= 0*) – Optional array or list of node weights to be used for node splitting invariant network measures. Entry [i] is the weight of node i. (Default: list of ones)
- **silence_level** (*int*) – The inverse level of verbosity of the object.

__str__()

Return a string representation of InteractingNetworks object.

static **_calculate_general_average_path_length**(*path_lengths*, *internal=False*)

Calculate general average path length for interacting networks.

Parameters

- **path_lengths** (*2D array [index, index]*) – The path length matrix.
- **internal** (*bool*) – Indicates, whether internal or cross average path length shall be calculated.

Return float the general average path length.

_calculate_general_closeness(*path_lengths*, *internal=True*)

Calculate general closeness sequence for interacting networks.

Parameters

- **path_lengths** (*2D array [node,node] of floats*) – Path lengths to use
- **internal** (*bool*) – Indicates, whether internal or cross closeness shall be calculated.

Return type 1D array [index]

Returns the general closeness sequence.

cross_adjacency(*node_list1*, *node_list2*)

Return cross adjacency matrix describing the interaction of two subnetworks.

The cross adjacency matrix entry $CA_{ij} = 1$ describes that node i in the first subnetwork is linked to node j in the second subnetwork. Vice versa, $CA_{ji} = 1$ indicates that node j in the first subnetwork is linked to node i in the second subnetwork.

Note: The Cross adjacency matrix is NEITHER square NOR symmetric in general!

Examples:

```
>>> r(InteractingNetworks.SmallTestNetwork().            cross_adjacency([1,2,4], [0,
array([[0, 1, 0], [0, 0, 0], [1, 0, 0]])
>>> r(InteractingNetworks.SmallTestNetwork().            cross_adjacency([1,2,3,4], [
array([[0, 0], [0, 0], [1, 0], [1, 0]])
```

Parameters

- **node_list1** (*[int]*) – list of node indices describing the first subnetwork
- **node_list2** (*[int]*) – list of node indices describing the second subnetwork

Return type 2D array [node index_1, node index_2]

Returns the cross adjacency matrix.

cross_adjacency_sparse(*node_list1*, *node_list2*)
Return cross adjacency matrix describing the interaction of two subnetworks.

The cross adjacency matrix entry M{CA_ij = 1} describes that node i in the first subnetwork is linked to node j in the second subnetwork. Vice versa, M{CA_ji = 1} indicates that node j in the first subnetwork is linked to node i in the second subnetwork.

Note: The Cross adjacency matrix is NEITHER square NOR symmetric in general!

Examples:

```
>>> print InteractingNetworks.SmallTestNetwork().              cross_adjacency_sparse([
[[0 1 0] [0 0 0] [1 0 0]]
```

Parameters

- **node_list1** (*[int]*) – list of node indices describing the first subnetwork
- **node_list2** (*[int]*) – list of node indices describing the second subnetwork

Return type 2D array [node index_1, node index_2]

Returns the cross adjacency matrix.

cross_average_path_length(*node_list1*, *node_list2*, *link_attribute=None*)
Return cross average path length.

Return the average (weighted) shortest path length between two induced subnetworks.

Examples:

```
>>> InteractingNetworks.SmallTestNetwork().              cross_average_path_length([0,3
2.0
>>> InteractingNetworks.SmallTestNetwork().              cross_average_path_length([0,5
2.0
```

Parameters

- **node_list1** (*[int]*) – list of node indices describing the first subnetwork
- **node_list2** (*[int]*) – list of node indices describing the second subnetwork
- **link_attribute** (*str*) – Optional name of the link attribute to be used as the links' length. If None, links have length 1. (Default: None)

Return float the cross average path length between a pair of subnetworks.

cross_betweenness(*node_list1*, *node_list2*)
Return the cross betweenness sequence for the whole network with respect to a pair of subnetworks.

Gives the normalized number of shortest paths only between nodes from **two** subnetworks, in which a node i is contained. This is equivalent to the inter-regional / inter-group betweenness with respect to subnetwork 1 and subnetwork 2.

Examples:

```
>>> InteractingNetworks.SmallTestNetwork().              cross_betweenness([2], [3,5])
array([ 1.,  1.,  0.,  0.,  1.,  0.])
>>> InteractingNetworks.SmallTestNetwork().              cross_betweenness(range(0,6),
array([ 9.,  3.,  0.,  2.,  6.,  0.])
```

Parameters

- **node_list1** (*[int]*) – list of node indices describing the first subnetwork

- **node_list2** (*[int]*) – list of node indices describing the second subnetwork

Return type 1D arrays [node index]

Returns the cross betweenness sequence for the whole network with respect to two subnetworks.

cross_closeness(*node_list1*, *node_list2*, *link_attribute=None*)

Return cross closeness sequence for a pair of induced subnetworks.

Gives the inverse average geodesic distance from a node in subnetwork 1 to all nodes in subnetwork 2.

Examples:

```
>>> InteractingNetworks.SmallTestNetwork().            cross_closeness([0,3,5], [1,2,
array([ 0.6  ,  0.6  ,  0.375])
>>> InteractingNetworks.SmallTestNetwork().            cross_closeness([0,5], [1,2,3,
array([ 0.66666667,  0.4       ])
```

Parameters

- **node_list1** (*[int]*) – list of node indices describing the first subnetwork
- **node_list2** (*[int]*) – list of node indices describing the second subnetwork
- **link_attribute** (*str*) – Optional name of the link attribute to be used as the links' length. If None, links have length 1. (Default: None)

Return type 1D arrays [index]

Returns the cross closeness sequence.

cross_degree(*node_list1*, *node_list2*)

Return the cross degree sequence for one subnetwork with respect to a second subnetwork.

Gives the number of links from a specific node in the first subnetwork projecting to the second subnetwork.

Examples:

```
>>> InteractingNetworks.SmallTestNetwork().            cross_degree([0,3,5], [1,2,4])
array([1, 1, 0])
>>> InteractingNetworks.SmallTestNetwork().            cross_degree([1,2,4], [0,3,5])
array([1, 0, 1])
>>> InteractingNetworks.SmallTestNetwork().            cross_degree([1,2,3,4], [0,5])
array([0, 0, 1, 1])
```

Parameters

- **node_list1** (*[int]*) – list of node indices describing the first subnetwork
- **node_list2** (*[int]*) – list of node indices describing the second subnetwork

Return type 1D array [node index]

Returns the cross degree sequence.

cross_global_clustering(*node_list1*, *node_list2*)

Return global cross clustering for a pair of subnetworks.

The global cross clustering coefficient C_v gives the average probability, that two randomly drawn neighbors in subnetwork 2 of node v in subnetwork 1 are also neighbors and vice versa. It counts triangles having one vertex in subnetwork 1 and two vertices in subnetwork 2 and vice versa.

Examples:

```
>>> InteractingNetworks.SmallTestNetwork().              cross_global_clustering([0,3,5
0.0
>>> InteractingNetworks.SmallTestNetwork().              cross_global_clustering([2],  [
1.0
>>> InteractingNetworks.SmallTestNetwork().              cross_global_clustering([3,4],
0.5
```

Parameters

- **node_list1** (*[int]*) – list of node indices describing the first subnetwork
- **node_list2** (*[int]*) – list of node indices describing the second subnetwork

Return float the cross global clustering coefficient for a pair of subnetworks.

cross_global_clustering_sparse(*node_list1*, *node_list2*)
Return global cross clustering for a pair of subnetworks.

The global cross clustering coefficient C_v gives the average probability, that two randomly drawn neighbors in subnetwork 2 of node v in subnetwork 1 are also neighbors and vice versa. It counts triangles having one vertex in subnetwork 1 and two vertices in subnetwork 2 and vice versa.

Examples:

```
>>> InteractingNetworks.SmallTestNetwork().cross_global_clustering(
...                                                  [0,3,5], [1,2,4])
0.0
```

```
>>> InteractingNetworks.SmallTestNetwork().cross_global_clustering(
...                                                      [2], [1,3,4])
1.0
```

```
>>> InteractingNetworks.SmallTestNetwork().cross_global_clustering(
...                                                      [3,4], [1,2])
0.5
```

Parameters

- **node_list1** (*[int]*) – list of node indices describing the first subnetwork
- **node_list2** (*[int]*) – list of node indices describing the second subnetwork

Return float the cross global clustering coefficient for a pair of subnetworks.

cross_link_density(*node_list1*, *node_list2*)
Return the density of links between two subnetworks.

Examples:

```
>>> r(InteractingNetworks.SmallTestNetwork().              cross_link_density([0,3,5],
0.2222
>>> InteractingNetworks.SmallTestNetwork().              cross_link_density([0,5], [1,2
0.25
```

Parameters

- **node_list1** (*[int]*) – list of node indices describing the first subnetwork
- **node_list2** (*[int]*) – list of node indices describing the second subnetwork

Return float the density of links between two subnetworks.

cross_local_clustering(*node_list1*, *node_list2*)
Return local cross clustering for a pair of subnetworks.

The local cross clustering coefficient C_v gives the probability, that two randomly drawn neighbors in subnetwork 1 of node v in subnetwork 1 are also neighbors. It counts triangles having one vertex in subnetwork 1 and two vertices in subnetwork 2.

Examples:

```
>>> InteractingNetworks.SmallTestNetwork().                cross_local_clustering([0,3,5]
array([ 0.,  0.,  0.])
>>> InteractingNetworks.SmallTestNetwork().                cross_local_clustering([2], [1
array([ 1.])
>>> InteractingNetworks.SmallTestNetwork().                cross_local_clustering([3,4],
array([ 0.,  1.])
```

Parameters

- **`node_list1`** (*[int]*) – list of node indices describing the first subnetwork
- **`node_list2`** (*[int]*) – list of node indices describing the second subnetwork

Return type 1D array [node index]

Returns the cross local clustering coefficient.

`cross_local_clustering_sparse`(*node_list1*, *node_list2*)

Return local cross clustering for a pair of subnetworks.

The local cross clustering coefficient C_v gives the probability, that two randomly drawn neighbors in subnetwork 1 of node v in subnetwork 1 are also neighbors. It counts triangles having one vertex in subnetwork 1 and two vertices in subnetwork 2.

Examples:

```
>>> InteractingNetworks.SmallTestNetwork().                cross_local_clustering_sparse(
array([ 0.,  0.,  0.])
```

```
>>> InteractingNetworks.SmallTestNetwork().                cross_local_clustering_sparse(
array([ 1.])
```

```
>>> InteractingNetworks.SmallTestNetwork().                cross_local_clustering_sparse(
array([ 0.,  1.])
```

Parameters

- **`node_list1`** (*[int]*) – list of node indices describing the first subnetwork
- **`node_list2`** (*[int]*) – list of node indices describing the second subnetwork

Return type 1D array [node index]

Returns the cross local clustering coefficient.

`cross_path_lengths`(*node_list1*, *node_list2*, *link_attribute=None*)

Return cross path length matrix for a pair of subnetworks.

Contains the path length between nodes from different subnetworks. The paths may generally contain nodes from the full network.

Examples:

```
>>> InteractingNetworks.SmallTestNetwork().                cross_path_lengths([0,3,5], [1
array([[ 2.,  2.,  1.], [ 1.,  2.,  2.], [ 3.,  3.,  2.]])
>>> InteractingNetworks.SmallTestNetwork().                cross_path_lengths([0,5], [1,2
array([[ 2.,  2.,  1.,  1.], [ 3.,  3.,  2.,  2.]])
```

Parameters

- **`node_list1`** (*[int]*) – list of node indices describing the first subnetwork

- **node_list2** (*[int]*) – list of node indices describing the second subnetwork
- **link_attribute** (*str*) – Optional name of the link attribute to be used as the links' length. If None, links have length 1. (Default: None)

Return type 2D array [index1, index2]

Returns the cross path length matrix for a pair of subnetworks.

cross_transitivity(*node_list1*, *node_list2*)
Return cross transitivity for a pair of subnetworks.

The cross transitivity is the probability, that two randomly drawn neighbors in subnetwork 2 of node v in subnetwork 1 are also neighbors. It counts triangles having one vertex in subnetwork 1 and two vertices in subnetwork 2. Cross transitivity tends to weight low cross degree vertices less strongly when compared to the global cross clustering coefficient (see [Newman2003]).

Examples:

```
>>> InteractingNetworks.SmallTestNetwork().                cross_transitivity([0,3,5], [1
0.0
>>> InteractingNetworks.SmallTestNetwork().                cross_transitivity([2], [1,3,4
1.0
>>> InteractingNetworks.SmallTestNetwork().                cross_transitivity([3,4], [1,2
1.0
```

Parameters

- **node_list1** (*[int]*) – list of node indices describing the first subnetwork
- **node_list2** (*[int]*) – list of node indices describing the second subnetwork

Return float the cross transitivity for a pair of subnetworks.

cross_transitivity_sparse(*node_list1*, *node_list2*)
Return cross transitivity for a pair of subnetworks.

The cross transitivity is the probability, that two randomly drawn neighbors in subnetwork 2 of node v in subnetwork 1 are also neighbors. It counts triangles having one vertex in subnetwork 1 and two vertices in subnetwork 2. Cross transitivity tends to weight low cross degree vertices less strongly when compared to the global cross clustering coefficient (see Newman, SIAM Review, 2003).

Examples:

```
>>> InteractingNetworks.SmallTestNetwork().                cross_transitivity_sparse([0,3
0.0
>>> InteractingNetworks.SmallTestNetwork().                cross_transitivity_sparse([3,4
1.0
```

Parameters

- **node_list1** (*[int]*) – list of node indices describing the first subnetwork
- **node_list2** (*[int]*) – list of node indices describing the second subnetwork

Return float the cross transitivity for a pair of subnetworks.

internal_adjacency(*node_list*)
Return the adjacency matrix of a subnetwork induced by a subset of nodes.

Examples:

```
>>> InteractingNetworks.SmallTestNetwork().                internal_adjacency([0,3,5])
array([[0, 1, 1], [1, 0, 0], [1, 0, 0]], dtype=int8)
>>> InteractingNetworks.SmallTestNetwork().                internal_adjacency([1,2,4])
array([[0, 1, 1], [1, 0, 1], [1, 1, 0]], dtype=int8)
```

Parameters **node_list** (*[int]*) – list of node indices describing the subnetwork

Return type 2D array [node index, node index]

Returns the subnetwork's adjacency matrix.

internal_average_path_length(*node_list*, *link_attribute=None*)
Return internal average path length for an induced subnetwork.

Return the average (weighted) shortest path length between all pairs of nodes within a subnetwork separately for which a path exists. Paths between nodes from different subnetworks are not included in the average!

However, even if the end points lie within the same layer, the paths themselves will generally contain nodes from the whole network. To avoid this and only consider paths lying within the subnetwork, do the following:

```
>>> r(InteractingNetworks.SmallTestNetwork().                subnetwork([0,3,5]).average_
1.3333
```

Examples:

```
>>> r(InteractingNetworks.SmallTestNetwork().                internal_average_path_length
1.3333
>>> r(InteractingNetworks.SmallTestNetwork().                internal_average_path_length
1.0
```

Parameters

- **node_list** (*[int]*) – list of node indices describing the subnetwork
- **link_attribute** (*str*) – Optional name of the link attribute to be used as the links' length. If None, links have length 1. (Default: None)

Return float the internal average path length.

internal_betweenness(*node_list*)
Return the internal betweenness sequence for an induced subnetwork.

Gives the normalized number of shortest paths only between nodes from subnetwork 1, in which a node i from the whole network is contained. This is equivalent to the inter-regional / inter-group betweenness with respect to subnetwork 1 and subnetwork 1.

Examples:

```
>>> InteractingNetworks.SmallTestNetwork().                internal_betweenness(range(0,6
array([ 9.,  3.,  0.,  2.,  6.,  0.])
```

Parameters **node_list** (*[int]*) – list of node indices describing the subnetwork

Return type 1D array [node index]

Returns the internal betweenness sequence for layer 1.

internal_closeness(*node_list*, *link_attribute=None*)
Return internal closeness sequence for an induced subnetwork.

Gives the inverse average geodesic distance from a node to all other nodes in the same induced subnetwork.

However, the included paths will generally contain nodes from the whole network. To avoid this, do the following:

```
>>> r(InteractingNetworks.SmallTestNetwork().                subnetwork([0,3,5]).closenes
array([ 1. , 0.6667, 0.6667])
```

Examples:

```
>>> InteractingNetworks.SmallTestNetwork().                internal_closeness([0,3,5], No
array([ 1.        ,  0.66666667,  0.66666667])
>>> InteractingNetworks.SmallTestNetwork().                internal_closeness([1,2,4], No
array([ 1.,  1.,  1.])
```

Parameters

- **node_list** (*[int]*) – list of node indices describing the subnetwork
- **link_attribute** (*str*) – Optional name of the link attribute to be used as the links' length. If None, links have length 1. (Default: None)

Return type 1D array [index]

Returns the internal closeness sequence.

internal_degree(*node_list*)

Return the internal degree sequence of one induced subnetwork.

Gives the number of links from a specific node to other nodes within the same induced subnetwork.

Examples:

```
>>> InteractingNetworks.SmallTestNetwork().internal_degree([0,3,5])
array([2, 1, 1])
>>> InteractingNetworks.SmallTestNetwork().internal_degree([1,2,4])
array([2, 2, 2])
```

Parameters **node_list** (*[int]*) – list of node indices describing the subnetwork

Return type 1D array [node index]

Returns the internal degree sequence.

internal_global_clustering(*node_list*)

Return internal global clustering coefficients for an induced subnetwork.

Internal global clustering coefficients are calculated as mean values from the local clustering sequence of the whole network. This implies that triangles spanning different subnetworks will generally contribute to the internal clustering coefficient.

To avoid this and consider only triangles lying within the subnetwork:

```
>>> r(InteractingNetworks.SmallTestNetwork().                subnetwork([0,3,5]).global_c
0.0
```

Examples:

```
>>> r(InteractingNetworks.SmallTestNetwork().                internal_global_clustering([
0.0
>>> r(InteractingNetworks.SmallTestNetwork().                internal_global_clustering([
0.5556
```

Parameters **node_list** (*[int]*) – list of node indices describing the subnetwork

Return float the internal global clustering coefficient for a subnetwork.

internal_link_density(*node_list*)

Return the density of links within an induced subnetwork.

Examples:

```
>>> r(InteractingNetworks.SmallTestNetwork().            internal_link_density([0,3,5
0.6667
>>> r(InteractingNetworks.SmallTestNetwork().            internal_link_density([1,2,3
0.6667
```

Parameters **node_list** (*[int]*) – list of node indices describing the subnetwork

Return float the density of links within a subnetwork.

internal_path_lengths(*node_list*, *link_attribute=None*)

Return internal path length matrix of an induced subnetwork.

Contains the paths length between all pairs of nodes within the subnetwork. However, the paths themselves will generally contain nodes from the full network. To avoid this and only consider paths lying within the subnetwork, do the following:

```
>>> InteractingNetworks.SmallTestNetwork().            subnetwork([0,3,5]).path_lengt
array([[ 0., 1., 1.], [ 1., 0., 2.], [ 1., 2., 0.]])
```

Examples:

```
>>> InteractingNetworks.SmallTestNetwork().            internal_path_lengths([0,3,5],
array([[ 0., 1., 1.], [ 1., 0., 2.], [ 1., 2., 0.]])
>>> InteractingNetworks.SmallTestNetwork().            internal_path_lengths([1,2,4],
array([[ 0., 1., 1.], [ 1., 0., 1.], [ 1., 1., 0.]])
```

Parameters

- **node_list** (*[int]*) – list of node indices describing the subnetwork
- **link_attribute** (*str*) – Optional name of the link attribute to be used as the links' length. If None, links have length 1. (Default: None)

Return type 2D array [node index, node index]

Returns the internal path length matrix of an induced subnetwork.

nsi_cross_average_path_length(*node_list1*, *node_list2*)

Return n.s.i. cross average path length between two induced subnetworks.

Examples:

```
>>> net = InteractingNetworks.SmallTestNetwork()
>>> r(net.nsi_cross_average_path_length([0,5],[1,2,4]))
3.3306
>>> r(net.nsi_cross_average_path_length([1,3,4,5],[2]))
0.376
```

Parameters

- **node_list1** (*[int]*) – list of node indices describing the first subnetwork
- **node_list2** (*[int]*) – list of node indices describing the second subnetwork

Return float the n.s.i. cross-average path length between a pair of subnetworks.

nsi_cross_betweenness(*node_list1*, *node_list2*)

Return the n.s.i. cross betweenness sequence for the whole network with respect to a pair of subnetworks.

Examples:

```
>>> InteractingNetworks.SmallTestNetwork().            nsi_cross_betweenness([0,4,5],
array([ 6.53333333,  1.2       ,  0.        ,
        0.67692308,  0.67692308,  0.        ])
```

```
>>> InteractingNetworks.SmallTestNetwork().                nsi_cross_betweenness([0,1],[2
array([ 2.13333333,  0.        ,  0.        ,
        0.49230769,  0.92087912,  0.        ])
```

Parameters

- **node_list1** (*[int]*) – list of node indices describing the first subnetwork
- **node_list2** (*[int]*) – list of node indices describing the second subnetwork

Return type 1D arrays [node index]

Returns the n.s.i. cross betweenness sequence for the whole network with respect to two subnetworks.

nsi_cross_closeness_centrality(*node_list1*, *node_list2*)

Return the n.s.i. cross-closeness centrality for a pair of induced subnetworks.

Examples:

```
>>> InteractingNetworks.SmallTestNetwork().                nsi_cross_closeness_centrality
array([ 1.        ,  0.56756757,  0.48837209])
>>> InteractingNetworks.SmallTestNetwork().                nsi_cross_closeness_centrality
array([ 0.73333333,  1.        ,  0.42307692])
```

Parameters

- **node_list1** (*[int]*) – list of node indices describing the subnetwork 1
- **node_list2** (*[int]*) – list of node indices describing the subnetwork 2

Return type 1D array [node index]

Returns the n.s.i. cross-closeness for layer 1.

nsi_cross_degree(*node_list1*, *node_list2*)

Return the n.s.i. cross-degree for a pair of induced subnetworks.

Gives an estimation about the quota of the whole domain of interest of the subnetwork 2 any node in the subnetwork 1 is connected to.

Examples:

```
>>> InteractingNetworks.SmallTestNetwork().                nsi_cross_degree([0,1,2],[3,4,
array([ 4.2,  2.6,  1.4])
>>> InteractingNetworks.SmallTestNetwork().                nsi_cross_degree([0,2,5],[1,4]
array([ 1.4,  2.2,  0. ])
```

Parameters

- **node_list1** (*[int]*) – list of node indices describing the subnetwork 1
- **node_list2** (*[int]*) – list of node indices describing the subnetwork 2

Return type 1D array [node index]

Returns the n.s.i. cross-degree for layer 1.

nsi_cross_edge_density(*node_list1*, *node_list2*)

Return the n.s.i. density of edges between two subnetworks.

Examples:

```
>>> r(InteractingNetworks.SmallTestNetwork().                nsi_cross_edge_density([1,2,
0.1091
>>> r(InteractingNetworks.SmallTestNetwork().                nsi_cross_edge_density([0],[
0.7895
```

Parameters

- **node_list1** (*[int]*) – list of node indices describing the first subnetwork
- **node_list2** (*[int]*) – list of node indices describing the second subnetwork

Return float the n.s.i. cross density of edges between two subnetworks 1 and 2.

nsi_cross_global_clustering(*node_list1*, *node_list2*)

Return the n.s.i. cross-global clustering coefficient for an induced subnetwork 1 with regard to a second induced subnetwork 2.

Examples:

```
>>> InteractingNetworks.SmallTestNetwork().                nsi_cross_global_clustering([0
0.6687866480862498
```

Parameters

- **node_list1** (*[int]*) – list of node indices describing the subnetwork 1
- **node_list2** (*[int]*) – list of node indices describing the subnetwork 2

Return float the n.s.i. cross-global clustering coefficient for the subnetwork 1 with regard to subnetwork 2.

nsi_cross_local_clustering(*node_list1*, *node_list2*)

Return the n.s.i. cross-local clustering coefficient for a pair of induced subnetworks.

Examples:

```
>>> InteractingNetworks.SmallTestNetwork().                nsi_cross_local_clustering([0,
array([ 0.33786848,  0.50295858,  1.  ])
>>> InteractingNetworks.SmallTestNetwork().                nsi_cross_local_clustering([0,
array([ 1.,  1.,  0.])
```

Parameters

- **node_list1** (*[int]*) – list of node indices describing the subnetwork 1
- **node_list2** (*[int]*) – list of node indices describing the subnetwork 2

Return type 1D array [node index]

Returns the n.s.i. cross-local clustering coefficient for layer 1.

nsi_cross_mean_degree(*node_list1*, *node_list2*)

Return the n.s.i. cross-mean degree for a pair of induced subnetworks.

Examples:

```
>>> InteractingNetworks.SmallTestNetwork().                nsi_cross_mean_degree([0,1,2],
2.5
>>> InteractingNetworks.SmallTestNetwork().                nsi_cross_mean_degree([0,2,5],
0.94999999999999996
```

Parameters

- **node_list1** (*[int]*) – list of node indices describing the subnetwork 1
- **node_list2** (*[int]*) – list of node indices describing the subnetwork 2

Return float the n.s.i. cross-mean degree for layer 1.

nsi_cross_transitivity(*node_list1*, *node_list2*)
Return n.s.i. cross-transitivity for a pair of subnetworks.

Examples:

```
>>> r(InteractingNetworks.SmallTestNetwork().                nsi_cross_transitivity([1,2]
0.6352
>>> InteractingNetworks.SmallTestNetwork().                nsi_cross_transitivity([0,2,3]
1.0
```

Parameters

- **node_list1** (*[int]*) – list of node indices describing the first subnetwork
- **node_list2** (*[int]*) – list of node indices describing the second subnetwork

Return float the n.s.i. cross transitivity for a pair of subnetworks 1 and 2.

nsi_internal_closeness_centrality(*node_list*)
Return the n.s.i. internal closeness sequence of one induced subnetwork.

Examples:

```
>>> InteractingNetworks.SmallTestNetwork().                nsi_internal_closeness_central
array([ 1.        ,  0.68      ,  0.73913043])
>>> InteractingNetworks.SmallTestNetwork().                nsi_internal_closeness_central
array([ 0.84      ,  0.525     ,  0.72413793,  0.6       ])
```

Parameters **node_list** (*[int]*) – list of node indices describing the subnetwork

Return type 1D array [node index]

Returns the n.s.i. internal closeness sequence

nsi_internal_degree(*node_list*)
Return the n.s.i. internal degree sequence of one induced subnetwork.

Examples:

```
>>> InteractingNetworks.SmallTestNetwork().                nsi_internal_degree([0,3,5])
array([ 3.4,  1.8,  2.2])
>>> InteractingNetworks.SmallTestNetwork().                nsi_internal_degree([0,1,3,5])
array([ 3.4,  2. ,  2.6,  2.2])
```

Parameters **node_list** (*[int]*) – list of node indices describing the subnetwork

Return type 1D array [node index]

Returns the n.s.i. internal degree sequence

nsi_internal_local_clustering(*node_list*)
Return the n.s.i. internal cross-local clustering coefficient for an induced subnetwork.

Examples:

```
>>> InteractingNetworks.SmallTestNetwork().                nsi_internal_local_clustering(
array([ 0.73333333,  1.        ,  1.        ,  1.        ])
>>> InteractingNetworks.SmallTestNetwork().                nsi_internal_local_clustering(
array([ 1.        ,  1.        ,  0.86666667])
```

Parameters **node_list** (*[int]*) – list of node indices describing the subnetwork

Return type 1D numpy array [node_index]

Returns the n.s.i. internal-local clustering coefficient for all nodes within the induced subnetwork

number_cross_links (*node_list1*, *node_list2*)

Return the number of links connecting the two subnetworks.

Examples:

```
>>> InteractingNetworks.SmallTestNetwork().            number_cross_links([0,3,5],  [1
2
>>> InteractingNetworks.SmallTestNetwork().            number_cross_links([0,5],  [1,2
2
```

Parameters

- **node_list1** (*[int]*) – list of node indices describing the first subnetwork
- **node_list2** (*[int]*) – list of node indices describing the second subnetwork

Return int the number of links between nodes from different subnetworks.

number_internal_links (*node_list*)

Return the number of links within an induced subnetwork.

Examples:

```
>>> InteractingNetworks.SmallTestNetwork().            number_internal_links([0,3,5])
2
>>> InteractingNetworks.SmallTestNetwork().            number_internal_links([1,2,4])
3
```

Parameters **node_list** (*[int]*) – list of node indices describing the subnetwork

Return int the number of links within a given subnetwork.

subnetwork (*node_list*)

Return the subnetwork induced by a subset of nodes as a Network object.

This can be used to conveniently analyze the subnetwork separately, e.g., for calculation network measures solely this subnetwork.

Parameters **node_list** (*[int]*) – list of node indices describing the subnetwork

Return type *Network*

Returns the subnetwork induced by the nodes in node_list.

5.1.5 core.netcdf_dictionary

Provides classes for saving and loading NetCDF files from and to appropriate Python dictionaries, allowing NetCDF4 compression methods.

class pyunicorn.core.netcdf_dictionary.**NetCDFDictionary** (*data_dict=None*, *silence_level=0*)

Bases: object

Encapsulates appropriate dictionary following NetCDF conventions.

Also contains methods to load data from NetCDF and NetCDF4 files.

__init__ (*data_dict=None*, *silence_level=0*)

Return a NetCDF object containing an appropriately structured dictionary.

If no data_dict is given, a default quasi-empty dictionary is created.

Parameters

- **data_dict** (*dictionary*) – Contains data in a structure following NetCDF conventions: {“global_attributes”: {}, “dimensions”: {}, “variables”: {“obs”: {“array”: (), “dims”: (), “attributes”: ()}}}

- **`silence_level`** (*int >= 0*) – The higher, the less progress info is output.

`__str__`()
Return a string representation of the object.

`__weakref__`
list of weak references to the object (if defined)

static `from_file`(*file_name*, *with_array='all'*)
Load NetCDF4 file into a dictionary.

Supported file types `file_type` are:

- "NetCDF"
- "NetCDF4"

Parameters

- **`file_name`** (*str*) – The name of the data file.
- **`with_array`** (*[str]*) – Names of data arrays to be loaded completely.

Return type NetCDF instance

`silence_level` = None
(int >= 0) The higher, the less progress info is output.

`to_file`(*file_name*, *compress=False*, *comp_level=6*, *least_significant_digit=10*)
Write NetCDF4 file by using appropriate dictionary.

Parameters

- **`file_name`** (*str*) – The name of the data file.
- **`compress`** (*bool*) – Determines whether the data should be compressed.
- **`comp_level`** (*int*) – Level of compression, between 0 (no compression, fastest) and 9 (strongest compression, slowest).
- **`least_significant_digit`** (*int*) – Last precise digit.

5.1.6 core.network

Provides classes for analyzing spatially embedded complex networks, handling multivariate data and generating time series surrogates.

class `pyunicorn.core.network.`**`Network`**(*adjacency=None*, *edge_list=None*, *directed=False*, *node_weights=None*, *silence_level=0*)
Bases: `object`

A Network is a simple, undirected or directed graph with optional node and/or link weights. This class encapsulates data structures and methods to represent, generate and analyze such structures.

Network relies on the package igraph for many of its features, but also implements new functionality. Highlights include weighted and directed statistical network measures, measures based on random walks, and node splitting invariant network measures.

Examples:

Create an undirected network given the adjacency matrix:

```
>>> net = Network(adjacency=[[0,1,0,0,0,0], [1,0,1,0,0,1],
...                          [0,1,0,1,1,0], [0,0,1,0,1,0],
...                          [0,0,1,1,0,1], [0,1,0,0,1,0]])
```

Create an Erdos-Renyi random graph:

```
>>> net = Network.ErdosRenyi(n_nodes=100, link_probability=0.05)
Generating Erdos-Renyi random graph with 100 nodes and probability 0.05...
```

static BarabasiAlbert (*n_nodes=100*, *n_links_each=5*, *silence_level=0*)

Return a new undirected Barabasi-Albert random graph with exactly n_links_each * (n_nodes-n_links_each) links.

Parameters **silence_level** (*int >= 0*) – The higher, the less progress info is output.

static BarabasiAlbert_igraph (*n_nodes=100*, *n_links_each=5*, *silence_level=0*)

Return a new undirected Barabasi-Albert random graph generated by igraph.

CAUTION: actual no. of new links can be smaller than n_links_each because neighbours are drawn with replacement and graph is then simplified.

The given number of nodes are added in turn to the initially empty node set, and each new node is linked to the given number of existing nodes. The resulting link density is approx. 2 * `n_links_each/n_nodes`.

Example: Generating a random tree:

```
>>> net = Network.BarabasiAlbert_igraph(n_nodes=100, n_links_each=1)
>>> print net.link_density
0.02
```

Parameters

- **n_nodes** (*int > 0*) – Number of nodes. (Default: 100)
- **n_links_each** (*int > 0*) – Number of links to existing nodes each new node gets during construction. (Default: 5)
- **silence_level** (*int >= 0*) – The higher, the less progress info is output.

Return type *Network* instance

static ConfigurationModel (*degrees*, *silence_level=0*)

Return a new configuration model random graph with a given degree sequence.

Example: Generate a network of 1000 nodes with degree 3 each:

```
>>> net = Network.ConfigurationModel([3 for _ in xrange(0,1000)])
Generating configuration model random graph
from given degree sequence...
>>> print net.degree()[0]
3
```

Parameters

- **degrees** (*1d numpy array or list [node]*) – Array or list of degrees wanted.
- **silence_level** (*int >= 0*) – The higher, the less progress info is output.

Return type *Network* instance

static ErdosRenyi (*n_nodes=100*, *link_probability=None*, *n_links=None*, *silence_level=0*)

Return a new undirected Erdos-Renyi random graph with a given number of nodes and linking probability.

The expected link density equals this probability.

Example:

```
>>> print Network.ErdosRenyi(n_nodes=10, n_links=18)
Generating Erdos-Renyi random graph with 10 nodes and 18 links...
Network: undirected, 10 nodes, 18 links, link density 0.400.
```

Parameters

- **n_nodes** (*int > 0*) – Number of nodes. (Default: 100)
- **link_probability** (*float from 0 to 1, or None*) – If not None, each pair of nodes is independently linked with this probability. (Default: None)
- **n_links** (*int > 0, or None*) – If not None, this many links are assigned at random. Must be None if link_probability is not None. (Default: None)
- **silence_level** (*int >= 0*) – The higher, the less progress info is output.

Return type `Network` instance

static FromIGraph (*graph*, *silence_level=0*)
Return a `Network` object given an igraph Graph object.

Parameters

- **graph** (*igraph Graph object*) – The igraph Graph object to be converted.
- **silence_level** (*int >= 0*) – The higher, the less progress info is output.

Return type `Network` instance

Returns `Network` object.

static GrowPreferentially (*n_nodes=100*, *n_growths=1*, *n_links_new=1*, *n_links_old=1*, *nsi=True*, *preferential_exponent=1*, *n_initials=1*, *silence_level=0*)
EXPERIMENTAL: Return a random network grown with preferential weight increase and n.s.i. preferential attachment.

Return a random network grown as follows: Starting with a clique of 2*n_links_new+1 unit weight nodes, iteratively add a unit weight node, connect it with n_links_new different existing nodes chosen with probabilities proportional to their current n.s.i. degree, then increase the weights of n_growths nodes chosen with probabilities proportional to their current weight (with replacement), then add n_links_old new links between pairs of nodes chosen with probabilities proportional to their current weight.

Parameters **silence_level** (*int >= 0*) – The higher, the less progress info is output.

static GrowPreferentially_old (*n_nodes=100*, *m=2*, *silence_level=0*)
EXPERIMENTAL: Return a random network grown with preferential weight increase and preferential attachment.

Return a random network grown as follows: Starting with a clique of m+1 unit weight nodes, iteratively add a unit weight node and then m times increase the weight of an existing node by one unit, for n=m+2...N. Choose the growing node with probabilities proportional to the node's weight. After each node addition or weight increase, add one link from the respective node to another node, chosen with probability proportional to that node's n.s.i. degree.

Parameters **silence_level** (*int >= 0*) – The higher, the less progress info is output.

static GrowWeights (*n_nodes=100*, *n_initials=1*, *exponent=1*, *mode='exp'*, *split_prob=0.01*, *split_weight=100*, *beta=1.0*, *n_increases=1e+100*)
EXPERIMENTAL

static Load (*filename*, *fileformat=None*, *silence_level=0*, **args*, ***kwds*)
Return a Network object stored in a file.

Unified reading function for graphs. Relies on and partially extends the corresponding igraph function. Refer to igraph documentation for further details on the various reader methods for different formats.

This method tries to identify the format of the graph given in the first parameter and calls the corresponding reader method.

Existing node and link attributes/weights are also restored depending on the chosen file format. E.g., the formats GraphML and gzipped GraphML are able to store both node and link weights.

The remaining arguments are passed to the reader method without any changes.

Parameters

- **`filename`** (*str*) – The name of the file containing the Network object.
- **`fileformat`** (*str*) – the format of the file (if known in advance). `None` means auto-detection. Possible values are: `"ncol"` (NCOL format), `"lgl"` (LGL format), `"graphml"`, `"graphmlz"` (GraphML and gzipped GraphML format), `"gml"` (GML format), `"net"`, `"pajek"` (Pajek format), `"dimacs"` (DIMACS format), `"edgelist"`, `"edges"` or `"edge"` (edge list), `"adjacency"` (adjacency matrix), `"pickle"` (Python pickled format).
- **`silence_level`** (*int >= 0*) – The higher, the less progress info is output.

Return type Network object

Returns *`Network`* instance.

`N` **= None**

(int>0) number of nodes

static `SmallTestNetwork``()`

Return a 6-node undirected test network with node weights.

The network looks like this:

```
     3 - 1
     |   | \
 5 - 0 - 4 - 2
```

The node weights are [1.5, 1.7, 1.9, 2.1, 2.3, 2.5], a typical node weight for corrected n.s.i. measures would be 2.0.

Return type Network instance

static `WattsStrogatzGraph``(`*N*, *k*, *p*`)`

Return a Watt-Strogatz random graph.

Reference: *[Watts1998]*

Parameters

- **`N`** (*int > 0*) – Number of nodes.
- **`k`** (*int > 0*) – Each node is connected to k nearest neighbors in ring topology.
- **`p`** (*float > 0*) – Probability of rewiring each edge.

`__init__``(`*adjacency=None*, *edge_list=None*, *directed=False*, *node_weights=None*, *silence_level=0*`)`

Return a new directed or undirected Network object with given adjacency matrix and optional node weights.

Parameters

- **`adjacency`** (*square array-like [node,node], or pysparse matrix of 0s and 1s*) – Adjacency matrix of the new network. Entry [i,j] indicates whether node i links to node j. Its diagonal must be zero. Must be symmetric if directed=False.
- **`edge_list`** (*array-like list of lists*) – Edge list of the new network. Entries [i,0], [i,1] contain the end-nodes of an edge.
- **`directed`** (*bool*) – Indicates whether the network shall be considered as directed. If False, adjacency must be symmetric.
- **`node_weights`** (*1d numpy array or list [node] of floats >= 0*) – Optional array or list of node weights to be used for node splitting invariant network measures. Entry [i] is the weight of node i. (Default: list of ones)
- **`silence_level`** (*int >= 0*) – The higher, the less progress info is output.

Return type `Network` instance

Returns The new network.

`__len__`()

Return the number of nodes as the 'length'.

Example:

```
>>> len(Network.SmallTestNetwork())
6
```

Return type int > 0

`__str__`()

Return a short summary of the network.

Example:

```
>>> print Network.SmallTestNetwork()
Network: undirected, 6 nodes, 7 links, link density 0.467.
```

Return type string

`__weakref__`

list of weak references to the object (if defined)

static `_cum_histogram`(*values*, *n_bins*, *interval=None*)

Return a normalized cumulative histogram of a list of values, and the lower bin boundaries.

Example: Get the relative frequencies only:

```
>>> r(Network._cum_histogram(
...     values=[1,2,13], n_bins=3, interval=(0,30))[0])
array([ 1. ,  0.3333,   0. ])
```

Parameters

- **`values`** (*1d array or list of floats*) – The values whose distribution is wanted.
- **`n_bins`** (*int > 0*) – Number of bins to be used for the histogram.
- **`interval`** (*tuple (float,float), or None*) – Optional range to use. If None, the minimum and maximum values are used. (Default: None)

Return type tuple (list,list)

Returns A list of cumulative relative bin frequencies (entry [i] is the sum of the frequencies of all bins j >= i), and a list of lower bin boundaries.

`_eigenvector_centrality_slow`(*link_attribute=None*)

For each node, return its (weighted) eigenvector centrality.

This is the load on this node from the eigenvector corresponding to the largest eigenvalue of the (weighted) adjacency matrix, normalized to a maximum of 1.

Parameters **`link_attribute`** (*str*) – Optional name of the link attribute to be used as the links' weight. If None, links have weight 1. (Default: None)

Return type 1d numpy array [node] of floats

static `_histogram`(*values*, *n_bins*, *interval=None*)

Return a normalized histogram of a list of values, its statistical error, and the lower bin boundaries.

Example: Get the relative frequencies only:

```
>>> r(Network._histogram(
...     values=[1,2,13], n_bins=3, interval=(0,30))[0])
array([ 0.6667,  0.3333,  0. ])
```

Parameters

- **values** (*1d array or list of floats*) – The values whose distribution is wanted.
- **n_bins** (*int > 0*) – Number of bins to be used for the histogram.
- **interval** (*tuple (float,float), or None*) – Optional interval to use. If None, the minimum and maximum values are used. (Default: None)

Return type tuple (list,list,list)

Returns A list of relative bin frequencies, a list of estimated statistical errors, and a list of lower bin boundaries.

_set_adjacency(*adjacency*)
Set a new adjacency matrix.

Example:

```
>>> net = Network.SmallTestNetwork(); print net
Network: undirected, 6 nodes, 7 links, link density 0.467.
>>> net.adjacency = [[0,1],[1,0]]; print net
Network: undirected, 2 nodes, 1 links, link density 1.000.
```

Parameters **adjacency** (*square array-like [[0|1]]*) – Entry [i,j] indicates whether node i links to node j. Its diagonal must be zero. Symmetric if the network is undirected.

_set_node_weights(*weights*)
Set the node weights to be used for node splitting invariant network measures.

Example:

```
>>> net = Network.SmallTestNetwork(); print net.node_weights
[ 1.5  1.7  1.9  2.1  2.3  2.5]
>>> net.node_weights = [1,1,1,1,1,1]; print net.node_weights
[ 1.  1.  1.  1.  1.  1.]
```

Parameters **weights** (*array-like [float>=0]*) – array-like [node] of weights (default: [1...1])

adjacency
Return the (possibly non-symmetric) adjacency matrix as a dense matrix.

Example:

```
>>> r(Network.SmallTestNetwork().adjacency)
array([[0, 0, 0, 1, 1, 1], [0, 0, 1, 1, 1, 0], [0, 1, 0, 0, 1, 0],
       [1, 1, 0, 0, 0, 0], [1, 1, 1, 0, 0, 0], [1, 0, 0, 0, 0, 0]])
```

Return type square numpy array [node,node] of 0s and 1s

arenas_betweenness(**args*, ***kwargs*)
For each node, return its Arenas-type random walk betweenness.

This measures how often a random walk search for a random target node from a random source node is expected to pass this node. (see *[Arenas2003]*)

Example:

```
>>> r(Network.SmallTestNetwork().arenas_betweenness())
Calculating Arenas-type random walk betweenness...
   (giant component size: 6 (1.0))
array([ 50.1818, 50.1818, 33.4545, 33.4545, 50.1818, 16.7273])
```

Return type 1d numpy array [node] of floats >= 0

assortativity()

Return the assortativity coefficient.

This follows *[Newman2002]*.

Example:

```
>>> r(Network.SmallTestNetwork().assortativity())
-0.4737
```

Return type float between 0 and 1

average_link_attribute(*attribute_name*)

For each node, return the average of a link attribute over all links of that node.

Parameters **attribute_name** (*str*) – Name of link attribute to be used.

Return type 1d numpy array [node] of floats

average_neighbors_degree(**args*, ***kwargs*)

For each node, return the average degree of its neighbors.

(Does not use directionality information.)

Example:

```
>>> r(Network.SmallTestNetwork().average_neighbors_degree())
Calculating average neighbours' degrees...
array([ 2. ,  2.3333,  3. , 3. ,  2.6667,  3. ])
```

Return type 1d numpy array [node] of floats >= 0

average_path_length(*link_attribute=None*)

Return the average (weighted) shortest path length between all pairs of nodes for which a path exists.

Example:

```
>>> print r(Network.SmallTestNetwork().average_path_length())
Calculating average (weighted) shortest path length...
1.6667
```

Parameters **link_attribute** (*str*) – Optional name of the link attribute to be used as the links' length. If None, links have length 1. (Default: None)

Return type float

betweenness(**args*, ***kwargs*)

For each node, return its betweenness.

This measures roughly how many shortest paths pass through the node.

Example:

```
>>> Network.SmallTestNetwork().betweenness()
Calculating node betweenness...
array([ 4.5,  1.5,  0. ,  1. ,  3. ,  0. ])
```

Parameters **no_big_int** (*bool*) – Toggles use of big integer calculation (slow if False).

Return type 1d numpy array [node] of floats >= 0

cache = None
(dict) cache of re-usable computation results

clear_cache()
Clear cache of information that can be recalculated from basic data.

clear_link_attribute(*attribute_name*)
Clear cache of a link attribute.

Parameters **attribute_name** (*str*) – name of link attribute

clear_nsi_cache()
Clear cache of information that can be recalculated from basic data and depends on the node weights.

clear_paths_cache()
Clear cache of path legths for link attributes.

closeness(*link_attribute=None*)
For each node, return its (weighted) closeness.

This is the inverse of the mean shortest path length from the node to all other nodes.

Example:

```
>>> r(Network.SmallTestNetwork().closeness())
Calculating closeness...
array([ 0.7143, 0.625 , 0.5556, 0.625 , 0.7143, 0.4545])
```

Parameters **link_attribute** (*str*) – Optional name of the link attribute to be used as the links' length. If None, links have length 1. (Default: None)

Return type 1d numpy array [node] of floats between 0 and 1

copy()
Return a copy of the network.

coreness(**args*, ***kwargs*)
For each node, return its coreness.

The k-core of a network is a maximal subnetwork in which each node has at least degree k. (Degree here means the degree in the subnetwork of course). The coreness of a node is k if it is a member of the k-core but not a member of the (k+1)-core.

Example:

```
>>> Network.SmallTestNetwork().coreness()
Calculating coreness...
array([2, 2, 2, 2, 2, 1])
```

Return type 1d numpy array [node] of floats

degree(**args*, ***kwargs*)
Return list of degrees.

Example:

```
>>> Network.SmallTestNetwork().degree()
array([3, 3, 2, 2, 3, 1])
```

Return type array([int>=0])

degree_cdf (**args*, ***kwargs*)
Return the cumulative degree frequency distribution.

Example:

```
>>> r(Network.SmallTestNetwork().degree_cdf())
Calculating the cumulative degree distribution...
array([ 1. , 0.8333,  0.5 ])
```

Return type 1d numpy array [k] of ints >= 0

Returns Entry [k] is the number of nodes having degree k or more.

degree_distribution (**args*, ***kwargs*)
Return the degree frequency distribution.

Example:

```
>>> r(Network.SmallTestNetwork().degree_distribution())
Calculating the degree frequency distribution...
array([ 0.1667, 0.3333, 0.5 ])
```

Return type 1d numpy array [k] of ints >= 0

Returns Entry [k] is the number of nodes having degree k.

del_link_attribute (*attribute_name*)
Delete a link attribute.

Parameters **attribute_name** (*str*) – name of link attribute to be deleted

del_node_attribute (*attribute_name*)
Delete a node attribute.

Parameters **attribute_name** (*str*) – Name of node attribute to be deleted.

diameter (*directed=True*, *only_connected=True*)
Return the diameter (largest shortest path length between any nodes).

Example:

```
>>> print Network.SmallTestNetwork().diameter()
3
```

Parameters

- **directed** (*bool*) – Indicates whether to respect link directions if the network is directed. (Default: True)
- **only_connected** (*bool*) – Indicates whether to use only pairs of nodes with a connecting path. If False and the network is unconnected, the number of all nodes is returned. (Default: True)

Return type int >= 0

directed = None
(bool) Indicates whether the network is directed.

distance_based_measures (*replace_inf_by=None*)
Return a dictionary of local and global measures that are based on shortest path lengths.

This is useful for large graphs for which the matrix of all shortest path lengths cannot be stored.

EXPERIMENTAL!

Parameters **replace_inf_by** (*float/inf/None*) – If None, the number of nodes is used. (Default: None)

Return type dictionary with keys "closeness", "harmonic_closeness", "exponential_closeness", "average_path_length", "global_efficiency", "nsi_closeness", "nsi_harmonic_closeness", "nsi_exponential_closeness", "nsi_average_path_length", "nsi_global_efficiency"

do_nsi_clustering (*d0=None*, *tree_dotfile=None*, *distances=None*, *candidates=None*)
Perform agglomerative clustering based on representation accuracy.

This minimizes in each step the mean squared error of representing the pairwise node distances by their cluster averages.

Note: This is still EXPERIMENTAL!

See the code for arguments and return value.

Clusters 0...n-1 are the singletons (cluster i containing just node i). Clusters n...2n-2 are numbered in the order in which clusters are joined (a cluster with id c is a union of two earlier clusters with ids c1,c2 < c). In particular, cluster 2n-2 is the full set of nodes.

Return type dictionary

Returns

A dictionary containing the following keys:

- "min_clusters": int > 0. Smallest number of clusters generated.
- "error": array(n+1). Entry [k] is the representation error for the solution with k clusters.
- "node2cluster": array(n,n+1). Entry [i,k] is the id of the cluster that contains node i in the solution with k clusters.
- "cluster_weight": array(2n-1). Entry [c] is the total weight of cluster c.
- "cluster2rank": array(2n-1,n+1). Entry [c,k] is the descending order rank of cluster c in the k-cluster solution, i.e., the number of larger clusters in that solution. Use this to convert cluster ids in 0...2n-1 to cluster ids in 0...k-1.
- "node_in_cluster": array(n,2n-1). Entry [i,c] indicates whether node i is in the cluster with id c.
- "children": array(2n-1,2). Entries [c,0] and [c,1] are the ids of the two clusters that were joined to give cluster c.
- "sibling": array(2n-2). Entry [c] is the id of the cluster with which cluster c is joined.
- "parent": array(2n-2). Entry [c] is the id of the cluster that results from joining cluster c with its sibling.

do_nsi_hamming_clustering (*admissible_joins=None*, *alpha=0.01*, *tree_dotfile=None*)
Perform agglomerative clustering based on Hamming distances.

This minimizes in each step the Hamming distance between the original and the "clustered" network.

Note: This is still EXPERIMENTAL!

See the code for arguments and return value.

Clusters 0...n-1 are the singletons (cluster i containing just node i). Clusters n...2n-2 are numbered in the order in which clusters are joined (a cluster with id c is a union of two earlier clusters with ids c1,c2 < c). In particular, cluster 2n-2 is the full set of nodes.

Return type dictionary

Returns

A dictionary containing the following keys:

- "error": array(n+1). Entry [k] is the representation error for the solution with k clusters.
- "node2cluster": array(n,n+1). Entry [i,k] is the id of the cluster that contains node i in the solution with k clusters.
- "cluster_weight": array(2n-1). Entry [c] is the total weight of cluster c.
- "cluster2rank": array(2n-1,n+1). Entry [c,k] is the descending order rank of cluster c in the k-cluster solution, i.e., the number of larger clusters in that solution. Use this to convert cluster ids in 0...2n-1 to cluster ids in 0...k-1.
- "node_in_cluster": array(n,2n-1). Entry [i,c] indicates whether node i is in the cluster with id c.
- "children": array(2n-1,2). Entries [c,0] and [c,1] are the ids of the two clusters that were joined to give cluster c.
- "sibling": array(2n-2). Entry [c] is the id of the cluster with which cluster c is joined.
- "parent": array(2n-2). Entry [c] is the id of the cluster that results from joining cluster c with its sibling.

do_nsi_pca_clustering(*max_n_clusters=None*)

Perform a clustering of the nodes using principal components analysis.

Perform a PCA for the columns of the adjacency matrix, extract the largest eigenvalues, and assign each node to that eigenvalue whose eigenvector explains the largest amount of the node's column's variance, i.e. the one that maximizes the value of eigenvalue * corresponding factor load on that node's column.

Note: This is still EXPERIMENTAL!

Parameters **max_n_clusters** (*int* >= *1*) – Number of clusters to find at most. (Default: ceil(sqrt(N)))

Return type tuple (list[node], list[node], list[cluster], 2d numpy array)

Returns A list of cluster indices for each node, a list with the fraction of the node's column's variance explained by chosen eigenvector, for each node, a list of eigenvalues corresponding to each cluster, and an array whose columns are the corresponding eigenvectors

edge_betweenness()

For each link, return its betweenness.

Alias to `link_betweenness()`. This measures on how likely the link is on a randomly chosen shortest path in the network.

(Does not respect directionality of links.)

Example:

```
>>> print Network.SmallTestNetwork().edge_betweenness()
Calculating link betweenness...
[[ 0.   0.   0.   3.5  5.5  5. ] [ 0.   0.   2.   3.5  2.5  0. ]
 [ 0.   2.   0.   0.   3.   0. ] [ 3.5  3.5  0.   0.   0.   0. ]
 [ 5.5  2.5  3.   0.   0.   0. ] [ 5.   0.   0.   0.   0.   0. ]]
```

Return type square numpy array [node,node] of floats between 0 and 1

Returns Entry [i,j] is the betweenness of the link between i and j, or 0 if i is not linked to j.

edge_list()

Return the network's edge list.

Example:

```
>>> print Network.SmallTestNetwork().edge_list()[:8]
[[0 3] [0 4] [0 5] [1 2] [1 3] [1 4] [2 1] [2 4]]
```

Return type array-like (numpy matrix or list of lists/tuples)

eigenvector_centrality(**args*, ***kwargs*)
For each node, return its eigenvector centrality.

This is the load on this node from the eigenvector corresponding to the largest eigenvalue of the adjacency matrix, normalized to a maximum of 1.

Example:

```
>>> r(Network.SmallTestNetwork().eigenvector_centrality())
Calculating eigenvector centrality...
array([ 0.7895, 0.973 , 0.7769, 0.6941, 1. , 0.3109])
```

Return type 1d numpy array [node] of floats

global_clustering(**args*, ***kwargs*)
Return the global (Watts-Strogatz) clustering coefficient.

This is the mean of the local clustering coefficients. *[Newman2003]* refers to this measure as C_2.

Example:

```
>>> r(Network.SmallTestNetwork().global_clustering())
Calculating global clustering coefficient (C_2)...
Calculating local clustering coefficients...
0.2778
```

Return type float between 0 and 1

global_efficiency(*link_attribute=None*)
Return the global (weighted) efficiency. (see *[Costa2007]*)

Example:

```
>>> r(Network.SmallTestNetwork().global_efficiency())
Calculating all shortest path lengths...
Calculating global (weighted) efficiency...
0.7111
```

Parameters **link_attribute** (*str*) – Optional name of the link attribute to be used as the links' length. If None, links have length 1. (Default: None)

Return type float

graph = None
(igraph.Graph) Embedded graph object providing some standard network measures.

hamming_distance_from(*other_network*)
Return the normalized hamming distance between this and another network.

This is the percentage of links that have to be changed to transform this network into the other. Hamming distance is only defined for networks with an equal number of nodes.

Return type float between 0 and 1

higher_order_transitivity(*order*, *estimate=False*)
Return transitivity of a certain order.

The transitivity of order n is defined as:

- (n x Number of cliques of n nodes) / (Number of stars of n nodes)

It is a generalization of the standard network transitivity, which is included as a special case for n = 3.

Parameters

- **order** (*int*) – The order (number of nodes) of cliques to be considered.
- **estimate** (*bool*) – Toggles random sampling for estimating higher order transitivity (much faster than exact calculation).

Return type number (float) between 0 and 1

indegree (**args*, ***kwargs*)

Return list of in-degrees.

Example:

```
>>> Network.SmallTestNetwork().indegree()
array([3, 3, 2, 2, 3, 1])
```

Return type array([int>=0])

indegree_cdf (**args*, ***kwargs*)

Return the cumulative in-degree frequency distribution.

Example:

```
>>> r(Network.SmallTestNetwork().indegree_cdf())
Calculating the cumulative in-degree distribution...
array([ 1. , 0.8333, 0.8333, 0.5 ])
```

Return type 1d numpy array [k] of ints >= 0

Returns Entry [k] is the number of nodes having in-degree k or more.

indegree_distribution (**args*, ***kwargs*)

Return the in-degree frequency distribution.

Example:

```
>>> r(Network.SmallTestNetwork().indegree_distribution())
Calculating in-degree frequency distribution...
array([ 0.1667, 0.3333, 0.5 ])
```

Return type 1d numpy array [k] of ints >= 0

Returns Entry [k] is the number of nodes having in-degree k.

interregional_betweenness (**args*, ***kwargs*)

For each node, return its interregional betweenness for given sets of source and target nodes.

This measures roughly how many shortest paths from one of the sources to one of the targets pass through the node.

Examples:

```
>>> Network.SmallTestNetwork().interregional_betweenness(
...     sources=[2], targets=[3,5])
Calculating interregional betweenness...
array([ 1.,  1.,  0.,  0.,  1.,  0.])
>>> Network.SmallTestNetwork().interregional_betweenness(
...     sources=range(0,6), targets=range(0,6))
Calculating interregional betweenness...
array([ 9.,  3.,  0.,  2.,  6.,  0.])
```

as compared to

```
>>> Network.SmallTestNetwork().betweenness()
Calculating node betweenness...
array([ 4.5,  1.5,  0. ,  1. ,  3. ,  0. ])
```

Parameters

- **sources** (*1d numpy array or list of ints from 0 to n_nodes-1*) – Set of source node indices.
- **targets** (*1d numpy array or list of ints from 0 to n_nodes-1*) – Set of target node indices.

Return type 1d numpy array [node] of floats between 0 and 1

laplacian (*direction='out'*, *link_attribute=None*)
Return the (possibly non-symmetric) dense Laplacian matrix.

Example:

```
>>> r(Network.SmallTestNetwork().laplacian())
array([[ 3,  0,  0, -1, -1, -1], [ 0,  3, -1, -1, -1,  0],
       [ 0, -1,  2,  0, -1,  0], [-1, -1,  0,  2,  0,  0],
       [-1, -1, -1,  0,  3,  0], [-1,  0,  0,  0,  0,  1]])
```

Parameters

- **direction** (*str*) – This argument is ignored for undirected graphs. "out" - out-degree on diagonal of laplacian "in" - in-degree on diagonal of laplacian
- **link_attribute** (*str*) – name of link attribute to be used

Return type square array [node,node] of ints

link_attribute (*attribute_name*)
Return the values of a link attribute.

Parameters **attribute_name** (*str*) – Name of link attribute to be used.

Return type square numpy array [node,node]

Returns Entry [i,j] is the attribute of the link from i to j.

link_betweenness (**args*, ***kwargs*)
For each link, return its betweenness.

This measures on how likely the link is on a randomly chosen shortest path in the network.

(Does not respect directionality of links.)

Example:

```
>>> print Network.SmallTestNetwork().link_betweenness()
Calculating link betweenness...
[[ 0.   0.   0.   3.5  5.5  5. ] [ 0.   0.   2.   3.5  2.5  0. ]
 [ 0.   2.   0.   0.   3.   0. ] [ 3.5  3.5  0.   0.   0.   0. ]
 [ 5.5  2.5  3.   0.   0.   0. ] [ 5.   0.   0.   0.   0.   0. ]]
```

Return type square numpy array [node,node] of floats between 0 and 1

Returns Entry [i,j] is the betweenness of the link between i and j, or 0 if i is not linked to j.

link_density = None
(0<float<1) proportion of linked node pairs

local_cliquishness(*order*)
Return local cliquishness of a certain order.

The local cliquishness measures the relative number of cliques (fully connected subgraphs) of a certain order that a node participates in.

Local cliquishness is not defined for orders 1 and 2. For order 3, it is equivalent to the local clustering coefficient `local_clustering()`, since cliques of order 3 are triangles.

Local cliquishness is always bounded by 0 and 1 and set to zero for nodes with degree smaller than order - 1.

Parameters **order** (*number (int)*) – The order (number of nodes) of cliques to be considered.

Return type 1d numpy array [node] of floats between 0 and 1

local_clustering(**args*, ***kwargs*)
For each node, return its (Watts-Strogatz) clustering coefficient.

This is the proportion of all pairs of its neighbors which are themselves interlinked.

(Uses directionality information, if available)

Example:

```
>>> r(Network.SmallTestNetwork().local_clustering())
Calculating local clustering coefficients...
array([ 0. , 0.3333, 1. , 0. , 0.3333, 0. ])
```

Return type 1d numpy array [node] of floats between 0 and 1

local_vulnerability(*link_attribute=None*)
For each node, return its vulnerability. (see [Costa2007])

Example:

```
>>> r(Network.SmallTestNetwork().local_vulnerability())
Calculating all shortest path lengths...
Calculating global (weighted) efficiency...
Calculating (weighted) node vulnerabilities...
array([ 0.2969, 0.0625, -0.0313, -0.0078, 0.0977, -0.125 ])
```

Parameters **link_attribute** (*str*) – Optional name of the link attribute to be used as the links' length. If None, links have length 1. (Default: None)

Return type 1d numpy array [node] of floats

matching_index(**args*, ***kwargs*)
For each pair of nodes, return their matching index.

This is the ratio of the number of common neighbors and the number of nodes linked to at least one of the two nodes.

Example:

```
>>> print r(Network.SmallTestNetwork().matching_index())
Calculating matching index matrix...
[[ 1.    0.5   0.25    0.      0.      0.    ]
 [ 0.5   1.    0.25    0.      0.2     0.    ]
 [ 0.25  0.25  1.      0.3333  0.25    0.    ]
 [ 0.    0.    0.3333  1.      0.6667  0.5   ]
 [ 0.    0.2   0.25    0.6667  1.      0.3333]
 [ 0.    0.    0.      0.5     0.3333  1.    ]]
```

Return type array([[0<=float<=1,0<=float<=1]])

max_neighbors_degree(**args*, ***kwargs*)
For each node, return the maximal degree of its neighbors.

(Does not use directionality information.)

Example:

```
>>> Network.SmallTestNetwork().max_neighbors_degree()
Calculating maximum neighbours' degree...
array([3, 3, 3, 3, 3, 3])
```

Return type 1d numpy array [node] of ints >= 0

mean_node_weight = None
mean node weight

msf_synchronizability(**args*, ***kwargs*)
Return the synchronizability in the master stability function framework.

This is equal to the largest eigenvalue of the graph Laplacian divided by the smallest non-zero eigenvalue. A smaller value indicates higher synchronizability and vice versa. This function makes sense for undirected climate networks (with symmetric laplacian matrix). For directed networks, the undirected laplacian matrix is used.

(see *[Pecora1998]*)

Note: Only defined for undirected networks.

Example:

```
>>> r(Network.SmallTestNetwork().msf_synchronizability())
Calculating master stability function synchronizability...
6.7784
```

Return type float

n_links = None
(int>0) number of links

newman_betweenness(**args*, ***kwargs*)
For each node, return Newman's random walk betweenness.

This measures how often a random walk search for a random target node from a random source node is expected to pass this node, not counting when the walk returns along a link it took before to leave the node. (see *[Newman2005]*)

Example:

```
>>> r(Network.SmallTestNetwork().newman_betweenness())
Calculating Newman's random walk betweenness...
    (giant component size: 6 (1.0))
array([ 4.1818, 3.4182, 2.5091, 3.0182, 3.6 , 2. ])
```

Return type 1d numpy array [node] of floats >= 0

node_attribute(*attribute_name*)
Return a node attribute.

Examples for node attributes/weights are degree or betweenness.

Parameters **attribute_name** (*str*) – The name of the node attribute.

Return type 1D Numpy array [node]

Returns The node attribute sequence.

node_weights
(array([int>=0])) array of node weights

nsi_arenas_betweenness (*exclude_neighbors=True*, *stopping_mode='neighbors'*)
For each node, return its n.s.i. Arenas-type random walk betweenness.

This measures how often a random walk search for a random target node from a random source node is expected to pass this node. (see *[Arenas2003]*)

Examples:

```
>>> net = Network.SmallTestNetwork()
>>> r(net.nsi_arenas_betweenness())
Calculating n.s.i. Arenas-type random walk betweenness...
   (giant component size: 6 (1.0))
Calculating n.s.i. degree...
array([ 20.5814, 29.2103, 27.0075, 19.5434, 25.2849, 24.8483])
>>> r(net.splitted_copy().nsi_arenas_betweenness())
Calculating n.s.i. Arenas-type random walk betweenness...
   (giant component size: 7 (1.0))
Calculating n.s.i. degree...
array([ 20.5814, 29.2103, 27.0075, 19.5434, 25.2849, 24.8483, 24.8483])
>>> r(net.nsi_arenas_betweenness(exclude_neighbors=False))
Calculating n.s.i. Arenas-type random walk betweenness...
   (giant component size: 6 (1.0))
Calculating n.s.i. degree...
array([ 44.5351, 37.4058, 27.0075, 21.7736, 31.3256, 24.8483])
>>> r(net.nsi_arenas_betweenness(stopping_mode="twinness"))
Calculating n.s.i. Arenas-type random walk betweenness...
   (giant component size: 6 (1.0))
Calculating n.s.i. degree...
Calculating n.s.i. degree...
array([ 22.6153, 41.2314, 38.6411, 28.6195, 38.5824, 30.2994])
```

as compared to its unweighted version:

```
>>> net = Network.SmallTestNetwork()
>>> r(net.arenas_betweenness())
Calculating Arenas-type random walk betweenness...
   (giant component size: 6 (1.0))
array([ 50.1818, 50.1818, 33.4545, 33.4545, 50.1818, 16.7273])
>>> r(net.splitted_copy().arenas_betweenness())
Calculating Arenas-type random walk betweenness...
   (giant component size: 7 (1.0))
array([ 90.4242, 67.8182, 45.2121, 45.2121, 67.8182, 45.2121, 45.2121])
```

Parameters

- **exclude_neighbors** (*bool*) – Indicates whether to use only source and target nodes that are not linked to the node of interest. (Default: True)
- **stopping_mode** (*str*) – Specifies when the random walk is stopped. If "neighbors", the walk stops as soon as it reaches a neighbor of the target node. If "twinnness", the stopping probability at each step is the twinnness of the current and target nodes as given by `nsi_twinness()`. (Default: "neighbors")

Return type 1d numpy array [node] of floats >= 0

nsi_average_neighbors_degree (**args*, ***kwargs*)
For each node, return the average n.s.i. degree of its neighbors.

(not yet implemented for directed networks.)

Example:

```
>>> net = Network.SmallTestNetwork()
>>> r(net.nsi_average_neighbors_degree())
Calculating n.s.i. average neighbours' degree...
Calculating n.s.i. degree...
array([ 6.0417, 6.62 , 7.0898, 7.0434, 7.3554, 5.65 ])
>>> r(net.splitted_copy().nsi_average_neighbors_degree())
Calculating n.s.i. average neighbours' degree...
Calculating n.s.i. degree...
array([ 6.0417, 6.62 , 7.0898, 7.0434, 7.3554, 5.65 , 5.65 ])
```

as compared to the unweighted version:

```
>>> net = Network.SmallTestNetwork()
>>> r(net.average_neighbors_degree())
Calculating average neighbours' degrees...
array([ 2. , 2.3333, 3. , 3. , 2.6667, 3. ])
>>> r(net.splitted_copy().average_neighbors_degree())
Calculating average neighbours' degrees...
array([ 2.25 , 2.3333, 3. , 3.5 , 3. , 3. , 3. ])
```

Return type 1d numpy array [node] of floats >= 0

nsi_average_path_length(**args*, ***kwargs*)

Return the n.s.i. average shortest path length between all pairs of nodes for which a path exists.

The path length from a node to itself is considered to be 1 to achieve node splitting invariance.

Example:

```
>>> net = Network.SmallTestNetwork()
>>> r(net.nsi_average_path_length())
Calculating n.s.i. average shortest path length...
Calculating all shortest path lengths...
1.6003
>>> r(net.splitted_copy().nsi_average_path_length())
Calculating n.s.i. average shortest path length...
Calculating all shortest path lengths...
1.6003
```

as compared to the unweighted version:

```
>>> net = Network.SmallTestNetwork()
>>> r(net.average_path_length())
Calculating average (weighted) shortest path length...
1.6667
>>> r(net.splitted_copy().average_path_length())
Calculating average (weighted) shortest path length...
1.7619
```

Return type float

nsi_betweenness(***kwargs*)

For each node, return its n.s.i. betweenness.

This measures roughly how many shortest paths pass through the node, taking node weights into account.

Example:

```
>>> net = Network.SmallTestNetwork()
>>> r(net.nsi_betweenness())
Calculating n.s.i. betweenness...
array([ 29.6854, 7.7129, 0. , 3.0909, 9.6996, 0. ])
>>> r(net.splitted_copy().nsi_betweenness())
```

```
Calculating n.s.i. betweenness...
array([ 29.6854, 7.7129, 0. , 3.0909, 9.6996, 0. , 0. ])
```

as compared to the unweighted version:

```
>>> net = Network.SmallTestNetwork()
>>> net.betweenness()
Calculating node betweenness...
array([ 4.5,  1.5,  0. ,  1. ,  3. ,  0. ])
>>> net.splitted_copy().betweenness()
Calculating node betweenness...
array([ 8.5,  1.5,  0. ,  1.5,  4.5,  0. ,  0. ])
```

Return type 1d numpy array [node] of floats between 0 and 1

nsi_closeness(**args*, ***kwargs*)

For each node, return its n.s.i. closeness.

This is the inverse of the mean shortest path length from the node to all other nodes. If the network is not connected, the result is 0.

Example:

```
>>> net = Network.SmallTestNetwork()
>>> r(net.nsi_closeness())
Calculating n.s.i. closeness...
Calculating all shortest path lengths...
array([ 0.7692, 0.6486, 0.5825, 0.6417, 0.7229, 0.5085])
>>> r(net.splitted_copy().nsi_closeness())
Calculating n.s.i. closeness...
Calculating all shortest path lengths...
array([ 0.7692, 0.6486, 0.5825, 0.6417, 0.7229, 0.5085, 0.5085])
```

as compared to the unweighted version:

```
>>> net = Network.SmallTestNetwork()
>>> r(net.closeness())
Calculating closeness...
array([ 0.7143, 0.625 , 0.5556, 0.625 , 0.7143, 0.4545])
>>> r(net.splitted_copy().closeness())
Calculating closeness...
array([ 0.75 , 0.5455, 0.5 , 0.6 , 0.6667, 0.5 , 0.5 ])
```

Return type 1d numpy array [node] of floats between 0 and 1

nsi_degree(*typical_weight=None*)

For each node, return its uncorrected or corrected n.s.i. degree.

Examples:

```
>>> net = Network.SmallTestNetwork()
>>> net.nsi_degree()
Calculating n.s.i. degree...
array([ 8.4,  8. ,  5.9,  5.3,  7.4,  4. ])
>>> net.splitted_copy().nsi_degree()
Calculating n.s.i. degree...
array([ 8.4,  8. ,  5.9,  5.3,  7.4,  4. ,  4. ])
>>> net.nsi_degree(typical_weight=2.0)
array([ 3.2 ,  3.  ,  1.95,  1.65,  2.7 ,  1.  ])
>>> net.splitted_copy().nsi_degree(typical_weight=2.0)
Calculating n.s.i. degree...
array([ 3.2 ,  3.  ,  1.95,  1.65,  2.7 ,  1.  ,  1.  ])
```

as compared to the unweighted version:

```
>>> net = Network.SmallTestNetwork()
>>> r(net.degree())
array([3, 3, 2, 2, 3, 1])
>>> r(net.splitted_copy().degree())
array([4, 3, 2, 2, 3, 2, 2])
```

Parameters **`typical_weight`** (*float* > *0*) – Optional typical node weight to be used for correction. If None, the uncorrected measure is returned. (Default: None)

Return type array([float])

`nsi_degree_cumulative_histogram`(**args*, ***kwargs*)
Return a cumulative frequency (!) histogram of n.s.i. degree.

Example:

```
>>> r(Network.SmallTestNetwork().nsi_degree_cumulative_histogram())
Calculating a cumulative n.s.i. degree frequency histogram...
Calculating n.s.i. degree...
(array([ 1.  , 0.6667, 0.5  ]), array([ 4.  , 5.4667, 6.9333]))
```

Return type tuple (list,list)

Returns List of cumulative frequencies and list of lower bin bounds.

`nsi_degree_histogram`(**args*, ***kwargs*)
Return a frequency (!) histogram of n.s.i. degree.

Example:

```
>>> r(Network.SmallTestNetwork().nsi_degree_histogram())
Calculating a n.s.i. degree frequency histogram...
Calculating n.s.i. degree...
(array([ 0.3333, 0.1667, 0.5  ]), array([ 0.1179, 0.1667, 0.0962]),
 array([ 4.  , 5.4667, 6.9333]))
```

Return type tuple (list,list)

Returns List of frequencies and list of lower bin bounds.

`nsi_degree_uncorr`(**args*, ***kwargs*)
For each node, return its uncorrected n.s.i. degree.

Return type array([float])

`nsi_eigenvector_centrality`(**args*, ***kwargs*)
For each node, return its n.s.i. eigenvector centrality.

This is the load on this node from the eigenvector corresponding to the largest eigenvalue of the n.s.i. adjacency matrix, divided by sqrt(node weight) and normalized to a maximum of 1.

Example:

```
>>> net = Network.SmallTestNetwork()
>>> r(net.nsi_eigenvector_centrality())
Calculating n.s.i. eigenvector centrality...
array([ 0.8045, 1.  , 0.8093, 0.6179, 0.9867, 0.2804])
>>> r(net.splitted_copy().nsi_eigenvector_centrality())
Calculating n.s.i. eigenvector centrality...
array([ 0.8045, 1.  , 0.8093, 0.6179, 0.9867, 0.2804, 0.2804])
```

as compared to the unweighted version:

```
>>> r(net.eigenvector_centrality())
Calculating eigenvector centrality...
array([ 0.7895, 0.973 , 0.7769, 0.6941, 1. , 0.3109])
>>> r(net.splitted_copy().eigenvector_centrality())
Calculating eigenvector centrality...
array([ 1. , 0.8008, 0.6226, 0.6625, 0.8916, 0.582 , 0.582 ])
```

Return type 1d numpy array [node] of floats

nsi_exponential_closeness(**args*, ***kwargs*)

For each node, return its n.s.i. exponential harmonic closeness.

This is the mean of 2**(- shortest path length) from the node to all other nodes. If the network is not connected, the result is not necessarily 0.

Example:

```
>>> net = Network.SmallTestNetwork()
>>> r(net.nsi_exponential_closeness())
Calculating n.s.i. exponential closeness centrality...
Calculating all shortest path lengths...
array([ 0.425 , 0.3906, 0.3469, 0.3604, 0.4042, 0.2958])
>>> r(net.splitted_copy().nsi_exponential_closeness())
Calculating n.s.i. exponential closeness centrality...
Calculating all shortest path lengths...
array([ 0.425 , 0.3906, 0.3469, 0.3604, 0.4042, 0.2958, 0.2958])
```

Return type 1d numpy array [node] of floats between 0 and 1

nsi_global_clustering(**args*, ***kwargs*)

Return the n.s.i. global clustering coefficient.

(not yet implemented for directed networks.)

Example:

```
>>> r(Network.SmallTestNetwork().nsi_global_clustering())
Calculating n.s.i. global topological clustering coefficient...
Calculating n.s.i. degree...
0.8353
```

as compared to the unweighted version:

```
>>> r(Network.SmallTestNetwork().global_clustering())
Calculating global clustering coefficient (C_2)...
Calculating local clustering coefficients...
0.2778
```

Return type float between 0 and 1

nsi_global_efficiency(**args*, ***kwargs*)

Return the n.s.i. global efficiency.

Example:

```
>>> r(Network.SmallTestNetwork().nsi_global_efficiency())
Calculating n.s.i. global efficiency...
Calculating all shortest path lengths...
0.7415
```

Return type float

nsi_harmonic_closeness(**args*, ***kwargs*)

For each node, return its n.s.i. harmonic closeness.

This is the inverse of the harmonic mean shortest path length from the node to all other nodes. If the network is not connected, the result is not necessarily 0.

Example:

```
>>> net = Network.SmallTestNetwork()
>>> r(net.nsi_harmonic_closeness())
Calculating n.s.i. harmonic closeness...
Calculating all shortest path lengths...
array([ 0.85 , 0.7986, 0.7111, 0.7208, 0.8083, 0.6167])
>>> r(net.splitted_copy().nsi_harmonic_closeness())
Calculating n.s.i. harmonic closeness...
Calculating all shortest path lengths...
array([ 0.85 , 0.7986, 0.7111, 0.7208, 0.8083, 0.6167, 0.6167])
```

Return type 1d numpy array [node] of floats between 0 and 1

nsi_interregional_betweenness(**args*, ***kwargs*)

For each node, return its n.s.i. interregional betweenness for given sets of source and target nodes.

This measures roughly how many shortest paths from one of the sources to one of the targets pass through the node, taking node weights into account.

Example:

```
>>> r(Network.SmallTestNetwork().nsi_interregional_betweenness(
...     sources=[2], targets=[3,5]))
Calculating n.s.i. interregional betweenness...
array([ 3.1667, 2.3471, 0. , 0. , 2.0652, 0. ])
```

as compared to the unweighted version:

```
>>> Network.SmallTestNetwork().interregional_betweenness(
...     sources=[2], targets=[3,5])
Calculating interregional betweenness...
array([ 1.,  1.,  0.,  0.,  1.,  0.])
```

Return type 1d numpy array [node] of floats between 0 and 1

nsi_laplacian()

Return the n.s.i. Laplacian matrix (undirected networks only!).

Example:

```
>>> Network.SmallTestNetwork().nsi_laplacian()
Calculating n.s.i. degree...
array([[ 6.9,  0. ,  0. , -2.1, -2.3, -2.5],
       [ 0. ,  6.3, -1.9, -2.1, -2.3,  0. ],
       [ 0. , -1.7,  4. ,  0. , -2.3,  0. ],
       [-1.5, -1.7,  0. ,  3.2,  0. ,  0. ],
       [-1.5, -1.7, -1.9,  0. ,  5.1,  0. ],
       [-1.5,  0. ,  0. ,  0. ,  0. ,  1.5]])
```

Return type square array([[float]])

nsi_local_clustering(*typical_weight=None*)

For each node, return its uncorrected (between 0 and 1) or corrected (at most 1 / negative / NaN) n.s.i. clustering coefficient.

(not yet implemented for directed networks)

Example:

```
>>> net = Network.SmallTestNetwork()
>>> r(net.nsi_local_clustering())
Calculating n.s.i. degree...
array([ 0.5513, 0.7244, 1. , 0.8184, 0.8028, 1. ])
>>> r(net.splitted_copy().nsi_local_clustering())
Calculating n.s.i. degree...
array([ 0.5513, 0.7244, 1. , 0.8184, 0.8028, 1. , 1. ])
```

as compared to the unweighted version:

```
>>> net = Network.SmallTestNetwork()
>>> r(net.local_clustering())
Calculating local clustering coefficients...
array([ 0. , 0.3333, 1. , 0. , 0.3333, 0. ])
>>> r(net.splitted_copy().local_clustering())
Calculating local clustering coefficients...
array([ 0.1667, 0.3333, 1. ,  0. , 0.3333, 1. , 1. ])
```

Parameters **`typical_weight`** (*float > 0*) – Optional typical node weight to be used for correction. If None, the uncorrected measure is returned. (Default: None)

Return type array([float])

`nsi_local_clustering_uncorr`(**args*, ***kwargs*)

For each node, return its uncorrected n.s.i. clustering coefficient (between 0 and 1).

(not yet implemented for directed networks)

Return type array([float])

`nsi_local_soffer_clustering`(**args*, ***kwargs*)

For each node, return its n.s.i. clustering coefficient with bias-reduction following *[Soffer2005]*.

(not yet implemented for directed networks.)

Example:

```
>>> net = Network.SmallTestNetwork()
>>> r(net.nsi_local_soffer_clustering())
Calculating n.s.i. local Soffer clustering coefficients...
Calculating n.s.i. degree...
array([ 0.7665, 0.8754, 1. , 0.8184, 0.8469, 1. ])
>>> r(net.splitted_copy().nsi_local_soffer_clustering())
Calculating n.s.i. local Soffer clustering coefficients...
Calculating n.s.i. degree...
array([ 0.7665, 0.8754, 1. , 0.8184, 0.8469, 1. , 1. ])
```

as compared to the version without bias-reduction:

```
>>> r(Network.SmallTestNetwork().nsi_local_clustering())
Calculating n.s.i. degree...
array([ 0.5513, 0.7244, 1. , 0.8184, 0.8028, 1. ])
```

Return type 1d numpy array [node] of floats between 0 and 1

`nsi_max_neighbors_degree`(**args*, ***kwargs*)

For each node, return the maximal n.s.i. degree of its neighbors.

(not yet implemented for directed networks.)

Example:

```
>>> Network.SmallTestNetwork().nsi_max_neighbors_degree()
Calculating n.s.i. maximum neighbour degree...
Calculating n.s.i. degree...
array([ 8.4,  8. ,  8. ,  8.4,  8.4,  8.4])
```

as compared to the unweighted version:

```
>>> print Network.SmallTestNetwork().max_neighbors_degree()
Calculating maximum neighbours' degree...
[3 3 3 3 3 3]
```

Return type 1d numpy array [node] of floats >= 0

nsi_newman_betweenness(*add_local_ends=False*)

For each node, return its n.s.i. Newman-type random walk betweenness.

This measures how often a random walk search for a random target node from a random source node is expected to pass this node, not counting when the walk returns along a link it took before to leave the node. (see *[Newman2005]*)

In this n.s.i. version, node weights are taken into account, and only random walks are used that do not start or end in neighbors of the node.

Example:

```
>>> net = Network.SmallTestNetwork()
>>> r(net.nsi_newman_betweenness())
Calculating n.s.i. Newman-type random walk betweenness...
   (giant component size: 6 (1.0))
Calculating n.s.i. degree...
array([ 0.4048, 0. , 0.8521, 3.3357, 1.3662, 0. ])
>>> r(net.splitted_copy().nsi_newman_betweenness())
Calculating n.s.i. Newman-type random walk betweenness...
   (giant component size: 7 (1.0))
Calculating n.s.i. degree...
array([ 0.4048, 0. , 0.8521, 3.3357, 1.3662, 0. , 0. ])
>>> r(net.nsi_newman_betweenness(add_local_ends=True))
Calculating n.s.i. Newman-type random walk betweenness...
   (giant component size: 6 (1.0))
Calculating n.s.i. degree...
array([ 131.4448, 128. , 107.6421, 102.4457, 124.2062, 80. ])
>>> r(net.splitted_copy().nsi_newman_betweenness(
...     add_local_ends=True))
Calculating n.s.i. Newman-type random walk betweenness...
   (giant component size: 7 (1.0))
Calculating n.s.i. degree...
array([ 131.4448, 128. , 107.6421, 102.4457, 124.2062, 80. , 80. ])
```

as compared to its unweighted version:

```
>>> net = Network.SmallTestNetwork()
>>> r(net.newman_betweenness())
Calculating Newman's random walk betweenness...
   (giant component size: 6 (1.0))
array([ 4.1818, 3.4182, 2.5091, 3.0182, 3.6 , 2. ])
>>> r(net.splitted_copy().newman_betweenness())
Calculating Newman's random walk betweenness...
   (giant component size: 7 (1.0))
array([ 5.2626, 3.5152, 2.5455, 3.2121, 3.8182, 2.5556, 2.5556])
```

Parameters **add_local_ends** (*bool*) – Indicates whether to add a correction for the fact that walks starting or ending in neighbors are not used. (Default: false)

Return type array [float>=0]

nsi_spreading(*alpha=None*)
For each node, return its n.s.i. "spreading" value.

Note: This is still EXPERIMENTAL!

Return type 1d numpy array [node] of floats

nsi_transitivity(**args*, ***kwargs*)
Return the n.s.i. transitivity.

Warning: Not yet implemented!

Return type float between 0 and 1

nsi_twinness()
For each pair of nodes, return an n.s.i. measure of 'twinness'.

This varies from 0.0 for unlinked nodes to 1.0 for linked nodes having exactly the same neighbors (called twins).

Example:

```
>>> net = Network.SmallTestNetwork()
>>> print r(net.nsi_twinness())
Calculating n.s.i. degree...
[[ 1.      0.      0.      0.4286  0.4524  0.4762]
 [ 0.      1.      0.7375  0.475   0.7375  0.    ]
 [ 0.      0.7375  1.      0.      0.7973  0.    ]
 [ 0.4286  0.475   0.      1.      0.      0.    ]
 [ 0.4524  0.7375  0.7973  0.      1.      0.    ]
 [ 0.4762  0.      0.      0.      0.      1.    ]]
>>> print r(net.splitted_copy().nsi_twinness())
Calculating n.s.i. degree...
[[ 1.      0.      0.      0.4286  0.4524  0.4762  0.4762]
 [ 0.      1.      0.7375  0.475   0.7375  0.      0.    ]
 [ 0.      0.7375  1.      0.      0.7973  0.      0.    ]
 [ 0.4286  0.475   0.      1.      0.      0.      0.    ]
 [ 0.4524  0.7375  0.7973  0.      1.      0.      0.    ]
 [ 0.4762  0.      0.      0.      0.      1.      1.    ]
 [ 0.4762  0.      0.      0.      0.      1.      1.    ]]
```

Return type square array [node,node] of floats between 0 and 1

outdegree(**args*, ***kwargs*)
Return list of out-degrees.

Example:

```
>>> Network.SmallTestNetwork().outdegree()
array([3, 3, 2, 2, 3, 1])
```

Return type array([int>=0])

outdegree_cdf(**args*, ***kwargs*)
Return the cumulative out-degree frequency distribution.

Example:

```
>>> r(Network.SmallTestNetwork().outdegree_cdf())
Calculating the cumulative out-degree distribution...
array([ 1. , 0.8333, 0.8333, 0.5 ])
```

Return type 1d numpy array [k] of ints >= 0

Returns Entry [k] is the number of nodes having out-degree k or more.

outdegree_distribution(**args*, ***kwargs*)

Return the out-degree frequency distribution.

Example:

```
>>> r(Network.SmallTestNetwork().outdegree_distribution())
Calculating out-degree frequency distribution...
array([ 0.1667, 0. , 0.3333, 0.5 ])
```

Return type 1d numpy array [k] of ints >= 0

Returns Entry [k] is the number of nodes having out-degree k.

pagerank(*link_attribute=None*, *use_directed=True*)

For each node, return its (weighted) PageRank.

This is the load on this node from the eigenvector corresponding to the largest eigenvalue of a modified adjacency matrix, normalized to a maximum of 1.

Example:

```
>>> r(Network.SmallTestNetwork().pagerank())
Calculating PageRank...
array([ 0.2184, 0.2044, 0.1409, 0.1448, 0.2047, 0.0869])
```

Parameters **link_attribute** (*str*) – Optional name of the link attribute to be used as the links' weight. If None, links have weight 1. (Default: None)

Return type 1d numpy array [node] of

path_lengths(*key=None*, ***kwargs*)

For each pair of nodes i,j, return the (weighted) shortest path length from i to j (also called the distance from i to j).

This is the shortest length of a path from i to j along links, or infinity if there is no such path.

The length of links can be specified by an optional link attribute.

Example:

```
>>> print Network.SmallTestNetwork().path_lengths()
Calculating all shortest path lengths...
[[ 0.  2.  2.  1.  1.  1.]
 [ 2.  0.  1.  1.  1.  3.]
 [ 2.  1.  0.  2.  1.  3.]
 [ 1.  1.  2.  0.  2.  2.]
 [ 1.  1.  1.  2.  0.  2.]
 [ 1.  3.  3.  2.  2.  0.]]
```

Parameters **link_attribute** (*str*) – Optional name of the link attribute to be used as the links' length. If None, links have length 1. (Default: None)

Return type square array [[float]]

permuted_copy(*permutation*)

Return a copy of the network with node numbers rearranged. This operation should not change topological information and network measures.

Parameters **permutation** (*array-like [int]*) – desired permutation of nodes

Return type `Network` instance

randomly_rewire (*iterations*)

Randomly rewire the network, preserving the degree sequence.

Example: Generate a network of 100 nodes with degree 5 each:

```
>>> net = Network.SmallTestNetwork(); print net
Network: undirected, 6 nodes, 7 links, link density 0.467.
>>> net.randomly_rewire(iterations=10); print net
Randomly rewiring the network,preserving the degree sequence...
Network: undirected, 6 nodes, 7 links, link density 0.467.
```

Parameters **iterations** (*int > 0*) – Number of iterations. In each iteration, two randomly chosen links a–b and c–d for which {a,c} and {b,d} are not linked, are replaced by the links a–c and b–d.

save (*filename*, *fileformat=None*, **args*, ***kwds*)

Save the Network object to a file.

Unified writing function for graphs. Relies on and partially extends the corresponding igraph function. Refer to igraph documentation for further details on the various writer methods for different formats.

This method tries to identify the format of the graph given in the first parameter (based on extension) and calls the corresponding writer method.

Existing node and link attributes/weights are also stored depending on the chosen file format. E.g., the formats GraphML and gzipped GraphML are able to store both node and link weights.

The remaining arguments are passed to the writer method without any changes.

Parameters

- **filename** (*str*) – The name of the file where the Network object is to be stored.
- **fileformat** (*str*) – the format of the file (if one wants to override the format determined from the filename extension, or the filename itself is a stream). `None` means auto-detection. Possible values are: `"ncol"` (NCOL format), `"lgl"` (LGL format), `"graphml"`, `"graphmlz"` (GraphML and gzipped GraphML format), `"gml"` (GML format), `"dot"`, `"graphviz"` (DOT format, used by GraphViz), `"net"`, `"pajek"` (Pajek format), `"dimacs"` (DIMACS format), `"edgelist"`, `"edges"` or `"edge"` (edge list), `"adjacency"` (adjacency matrix), `"pickle"` (Python pickled format), `"svg"` (Scalable Vector Graphics).

set_edge_list (*edge_list*)

Reset network from an edge list representation.

Note: Assumes that nodes are numbered by natural numbers from 0 to N-1 without gaps!

Example:

Parameters **edge_list** (*array-like [[int>=0,int>=0]]*) – [[i,j]] for edges i -> j

set_link_attribute (*attribute_name*, *values*)

Set the values of some link attribute.

These can be used as weights in measures requiring link weights.

Note: The attribute/weight matrix should be symmetric for undirected networks.

Parameters

- **attribute_name** (*str*) – name of link attribute to be set
- **values** (*square numpy array [node,node]*) – Entry [i,j] is the attribute of the link from i to j.

set_node_attribute(*attribute_name*, *values*)
Add a node attribute.

Examples for node attributes/weights are degree or betweenness.

Parameters

- **attribute_name** (*str*) – The name of the node attribute.
- **values** (*1D Numpy array [node]*) – The node attribute sequence.

silence_level = None
(int>=0) higher -> less progress info

sp_A = None
(sparse.csc_matrix([[int,int]]) with entries 0,1) Adjacency matrix. A[i,j]=1 indicates a link i -> j. Symmetric if the network is undirected.

sp_Aplus()
A^+ = A + Id. matrix used in n.s.i. measures

sp_diag_sqrt_w()
Sparse diagonal matrix of square roots of node weights

sp_diag_w()
Sparse diagonal matrix of node weights

sp_diag_w_inv()
Sparse diagonal matrix of inverse node weights

sp_nsi_diag_k()
Sparse diagonal matrix of n.s.i. degrees

sp_nsi_diag_k_inv()
Sparse diagonal matrix of inverse n.s.i. degrees

splitted_copy(*node=-1*, *proportion=0.5*)
Return a copy of the network with one node splitted.

The specified node is split in two interlinked nodes which are linked to the same nodes as the original node, and the weight is splitted according to the given proportion.

(This method is useful for testing the node splitting invariance of measures since a n.s.i. measure will be the same before and after the split.)

Example:

```
>>> net = Network.SmallTestNetwork(); print net
Network: undirected, 6 nodes, 7 links, link density 0.467.
>>> net2 = net.splitted_copy(node=5, proportion=0.2); print net2
Network: undirected, 7 nodes, 9 links, link density 0.429.
>>> print net.node_weights; print net2.node_weights
[ 1.5  1.7  1.9  2.1  2.3  2.5]
[ 1.5  1.7  1.9  2.1  2.3  2.   0.5]
```

Parameters

- **node** (*int*) – The index of the node to be splitted. If negative, N + index is used. The new node gets index N. (Default: -1)
- **proportion** (*float from 0 to 1*) – The splitted node gets a new weight of (1-proportion) * (weight of splitted node), and the new node gets a weight of proportion * (weight of splitted node). (Default: 0.5)

Return type `Network`

spreading(*alpha=None*)
For each node, return its "spreading" value.

Note: This is still EXPERIMENTAL!

Return type 1d numpy array [node] of floats

total_node_weight = None
total node weight

transitivity(**args, **kwargs*)
Return the transitivity (coefficient).

This is the ratio of three times the number of triangles to the number of connected triples of vertices. [Newman2003] refers to this measure as C_1.

Example:

```
>>> r(Network.SmallTestNetwork().transitivity())
Calculating transitivity coefficient (C_1)...
0.2727
```

Return type float between 0 and 1

undirected_adjacency()
Return the adjacency matrix of the undirected version of the network as a dense numpy array. Entry [i,j] is 1 if i links to j or j links to i.

Example:

```
>>> net = Network(adjacency=[[0,1],[0,0]], directed=True)
>>> print net.undirected_adjacency().A
[[0 1] [1 0]]
```

Return type array([[0|1]])

undirected_copy()
Return an undirected copy of the network.

Nodes i and j are linked in the copy if, in the current network, i links to j or j links to i or both.

Example:

```
>>> net = Network(adjacency=[[0,1],[0,0]], directed=True); print net
Network: directed, 2 nodes, 1 links, link density 0.500.
>>> print net.undirected_copy()
Network: undirected, 2 nodes, 1 links, link density 1.000.
```

Return type `Network` instance

static **weighted_local_clustering**(*weighted_A*)
For each node, return its weighted clustering coefficient, given a weighted adjacency matrix.

This follows [Holme2007].

Example:

```
>>> print r(Network.weighted_local_clustering(weighted_A=[
...     [ 0.  , 0.  , 0.  , 0.55, 0.65, 0.75],
...     [ 0.  , 0.  , 0.63, 0.77, 0.91, 0.  ],
...     [ 0.  , 0.63, 0.  , 0.  , 1.17, 0.  ],
...     [ 0.55, 0.77, 0.  , 0.  , 0.  , 0.  ],
...     [ 0.65, 0.91, 1.17, 0.  , 0.  , 0.  ],
...     [ 0.75, 0.  , 0.  , 0.  , 0.  , 0.  ]]))
Calculating local weighted clustering coefficient...
[ 0.  0.2149  0.3539  0.  0.1538  0. ]
```

as compared to the unweighted version:

```
>>> print r(Network.SmallTestNetwork().local_clustering())
Calculating local clustering coefficients...
[ 0.  0.3333  1.  0.  0.3333  0. ]
```

Parameters **weighted_A** (*square numpy array [node,node] of floats >= 0*) – Entry [i,j] is the link weight from i to j. A value of 0 means there is no link.

Return type 1d numpy array [node] of floats between 0 and 1

exception pyunicorn.core.network.**NetworkError**(*value*)
Bases: exceptions.Exception

Used for all exceptions raised by Network.

__weakref__
list of weak references to the object (if defined)

pyunicorn.core.network.**cache_helper**(*self*, *cat*, *key*, *msg*, *func*, **args*, ***kwargs*)
Cache result of a function in a subdict of self.cache.

Parameters

- **cat** (*str*) – cache category
- **key** (*str*) – cache key
- **msg** (*str*) – message to be displayed during first calculation
- **func** (*func*) – function to be cached

pyunicorn.core.network.**cached_const**(*cat*, *key*, *msg=None*)
Cache result of decorated method in a fixed subdict of self.cache.

pyunicorn.core.network.**cached_var**(*cat*, *msg=None*)
Cache result of decorated method in a variable subdict of self.cache, specified as first argument to the decorated method.

pyunicorn.core.network.**nz_coords**(*matrix*)
Find coordinates of all non-zero entries in a sparse matrix.

Returns list of coordinates [row,col]

Return type array([[int>=0,int>=0]])

pyunicorn.core.network.**weave_inline**(*local_dict*, *code*, *args*, *blitz=True*, ***kwargs*)
Default configuration for C inline code.

5.1.7 core.resistive_network

Module contains class ResNetwork.

Provides function for computing resistance based networks. It is subclassed from GeoNetwork and provides most GeoNetwork's functions/properties.

The class has the following instance variables:

```
(bool) flagDebug      : flag for debugging mode
(bool) flagWeave      : flag for switching between python/C code parts
(bool) flagComplex    : flag for complex input
(ndarray) resistances: array of resistances (complex or real)
```

Overriden inherited methods:

```
(str) __str__          : extended description
(ndarray) get_adjacency: returns complex adjacency if needed
```

class `pyunicorn.core.resistive_network.`**`ResNetwork`**(*resistances*, *grid=None*, *adjacency=None*, *edge_list=None*, *directed=False*, *node_weight_type=None*, *silence_level=2*)

Bases: *pyunicorn.core.geo_network.GeoNetwork*

A resistive network class

ResNetwork, provides methods for an extended analysis of resistive/resistance-based networks.

Examples:

```
>>> print ResNetwork.SmallTestNetwork()
ResNetwork:
GeoNetwork:
Network: undirected, 5 nodes, 5 links, link density 0.500.
Geographical boundaries:
         time      lat      lon
   min    0.0     0.00 -180.00
   max    9.0    90.00  180.00
Average resistance: 2.4
```

static **`SmallComplexNetwork`**()

A test network with complex resistances analogue to SmallTestNetwork()

Return type Resistive Network instance

Returns an ResNetwork instance with complex resistances

Examples:

```
>>> res = ResNetwork.SmallComplexNetwork()
>>> isinstance(res, ResNetwork)
True
>>> res.flagComplex
True
>>> adm = res.get_admittance()
>>> print adm.real
[[ 0.      0.1     0.      0.      0.    ]
 [ 0.1     0.      0.0625  0.25    0.    ]
 [ 0.      0.0625  0.      0.0625  0.    ]
 [ 0.      0.25    0.0625  0.      0.05  ]
 [ 0.      0.      0.      0.05    0.    ]]
```

```
>>> print adm.imag
[[ 0.     -0.2     0.      0.      0.    ]
 [-0.2     0.     -0.0625 -0.25    0.    ]
 [ 0.     -0.0625  0.     -0.0625  0.    ]
 [ 0.     -0.25   -0.0625  0.     -0.05  ]
 [ 0.      0.      0.     -0.05    0.    ]]
```

static **`SmallTestNetwork`**()

Create a small test network with unit resistances of the following topology:

```
0------1--------3------4
        \      /
         \    /
          \  /
           \/
           2
```

Return type Resistive Network instance

Returns an ResNetwork instance for testing purposes.

Examples:

```
>>> res = ResNetwork.SmallTestNetwork()
>>> isinstance(res, ResNetwork)
True
```

`__init__`(*resistances*, *grid=None*, *adjacency=None*, *edge_list=None*, *directed=False*, *node_weight_type=None*, *silence_level=2*)
Initialize an instance of ResNetwork.

Parameters

- **`resistances`** (*2D NumPy array*) – A matrix with the resistances
- **`grid`** (*Grid object*) – The Grid object describing the network's spatial embedding.
- **`adjacency`** (*2D NumPy array (int8) [index, index]*) – The network's adjacency matrix.
- **`edge_list`** (*array-like list of lists*) – Edge list of the new network. Entries [i,0], [i,1] contain the end-nodes of an edge.
- **`directed`** (*boolean*) – Determines, whether the network is treated as directed.
- **`node_weight_type`** (*string*) – The type of geographical node weight to be used.
- **`silence_level`** (*number (int)*) – The inverse level of verbosity of the object.

`__str__`()
Return a short summary of the resistive network.

`_edge_current_flow_betweenness_python`()
Python version of ECFB

`_edge_current_flow_betweenness_weave`()
Weave/C version of ECFB

`_vertex_current_flow_betweenness_python`(*i*)
Python version of VCFB

`_vertex_current_flow_betweenness_weave`(*i*)
C Version of VCFB

`admittance_lapacian`()
Return the (possibly non-symmetric) dense Laplacian matrix of the admittance.

Return type square NumPy matrix [node,node] of

Examples:

```
>>> print ResNetwork.SmallTestNetwork().admittance_lapacian()
[[ 0.5   -0.5    0.     0.     0.   ]
 [-0.5    1.125 -0.125 -0.5    0.   ]
 [ 0.    -0.125  0.25  -0.125  0.   ]
 [ 0.    -0.5   -0.125  0.725 -0.1  ]
 [ 0.     0.     0.    -0.1    0.1  ]]
>>> print type( ResNetwork.SmallTestNetwork().admittance_lapacian() )
<type 'numpy.ndarray'>
```

`admittive_degree`()
admittive degree of the network

The admittive (or effective) degree of the resistive network, which is the counterpart to the traditional degree.

Return type 1D NumPy array

Examples:

```
>>> print ResNetwork.SmallTestNetwork().admittive_degree()
[ 0.5    1.125  0.25   0.725  0.1  ]
>>> print type( ResNetwork.SmallTestNetwork().admittive_degree())
<type 'numpy.ndarray'>
```

average_effective_resistance()

Return the average effective resistance (<ER>) of the resistive network, the average resistances for all "paths" (connections)

Return type float

Examples:

```
>>> res = ResNetwork.SmallTestNetwork()
>>> print "%.5f" % res.average_effective_resistance()
7.28889
>>> print type( res.average_effective_resistance() )
<type 'numpy.float64'>
```

average_neighbors_admittive_degree()

Average neighbour effective degree

Return type 1D NumPy array

Examples:

```
>>> print ResNetwork.SmallTestNetwork().                average_neighbors_admittive_degre
[ 2.25   1.31111111  7.4   2.03448276  7.25   ]
>>> print type(ResNetwork.SmallTestNetwork().admittive_degree())
<type 'numpy.ndarray'>
```

diameter_effective_resistance()

Return the diameter (the highest resistance path between any nodes).

Return type float

Examples:

```
>>> res = ResNetwork.SmallTestNetwork()
>>> print "%.3f" % res.diameter_effective_resistance()
Re-computing all effective resistances
14.444
>>> print type(res.diameter_effective_resistance())
<type 'numpy.float64'>
```

```
>>> res = ResNetwork.SmallTestNetwork()
>>> x = res.average_effective_resistance()
>>> print "%.3f" % res.diameter_effective_resistance()
14.444
```

edge_current_flow_betweenness()

The electrial version of Newmann's edge betweeness

Return type NumPy float

Examples:

```
>>> res = ResNetwork.SmallTestNetwork()
>>> print res.edge_current_flow_betweenness()
[[ 0.          0.4         0.          0.          0.        ]
 [ 0.4         0.          0.24444444  0.53333333  0.        ]
 [ 0.          0.24444444  0.          0.24444444  0.        ]
 [ 0.          0.53333333  0.24444444  0.          0.4       ]
 [ 0.          0.          0.          0.4         0.        ]]
>>> # update to unit resistances
>>> res.update_resistances(res.adjacency)
>>> print res.edge_current_flow_betweenness()
[[ 0.          0.4         0.          0.          0.        ]
 [ 0.4         0.          0.33333333  0.4         0.        ]
 [ 0.          0.33333333  0.          0.33333333  0.        ]
 [ 0.          0.4         0.33333333  0.          0.4       ]
 [ 0.          0.          0.          0.4         0.        ]]
```

effective_resistance(*a*, *b*)

Return the effective resistance (ER) between two nodes a and b. The ER is the electrical analogue to the shortest path where a is considered as "source" and b as the "sink"

Parameters

- **a** (*int*) – index of the "source" node
- **b** (*int*) – index of the "sink" node

Return type NumPy float

Examples:

```
>>> res = ResNetwork.SmallTestNetwork()
>>> print res.effective_resistance(1,1)
0.0
>>> print type( res.effective_resistance(1,1) )
<type 'float'>
>>> print "%.3f" % res.effective_resistance(1,2)
4.444
>>> print type( res.effective_resistance(1,1) )
<type 'float'>
```

effective_resistance_closeness_centrality(*a*)

The effective resistance closeness centrality (ERCC) of node a

Parameters **a** (*int*) – index of the "source" node

Return type NumPy float

Examples:

```
>>> res = ResNetwork.SmallTestNetwork()
>>> print "%.3f" % res.effective_resistance_closeness_centrality(0)
0.154
>>> print "%.3f" % res.effective_resistance_closeness_centrality(4)
0.080
```

get_R()

Return the pseudo inverse of of the admittance Laplacian

The pseudoinverse is used of the novel betweeness measures such as *vertex_current_flow_betweenness()* and *edge_current_flow_betweenness()* It is computed on instantiation and on change of the resistances/admittance

Returns the pseudoinverse of the admittange Laplacian

Return type ndarray (float)

Examples:

```
>>> res = ResNetwork.SmallTestNetwork();print res.get_R()
[[ 2.28444444  0.68444444 -0.56       -0.20444444 -2.20444444]
 [ 0.68444444  1.08444444 -0.16        0.19555556 -1.80444444]
 [-0.56       -0.16        3.04       -0.16       -2.16      ]
 [-0.20444444  0.19555556 -0.16        1.08444444 -0.91555556]
 [-2.20444444 -1.80444444 -2.16       -0.91555556  7.08444444]]
```

get_admittance()

Return the (possibly non-symmetric) dense admittance matrix

Return type square NumPy matrix [node,node] of ints

Examples:

```
>>> res = ResNetwork.SmallTestNetwork();print res.get_admittance()
[[ 0.      0.5     0.      0.      0.    ]
 [ 0.5     0.      0.125   0.5     0.    ]
 [ 0.      0.125   0.      0.125   0.    ]
 [ 0.      0.5     0.125   0.      0.1   ]
 [ 0.      0.      0.      0.1     0.    ]]
>>> print type( res.get_admittance() )
<type 'numpy.ndarray'>
```

global_admittive_clustering()

Return node wise admittive clustering coefficient.

Return type NumPy float

Examples:

```
>>> res =  ResNetwork.SmallTestNetwork()
>>> print "%.3f" % res.global_admittive_clustering()
0.016
>>> print type(res.global_admittive_clustering())
<type 'numpy.float64'>
```

local_admittive_clustering()

Return node wise admittive clustering coefficient (AC).

The AC is the electrical analogue of the clustering coefficient for regular network (see `get_admittive_ws_clustering()` and `get_local_clustering()` and sometimes called Effective Clustering (EC))

The admittive clustering (ac) of node i is defined as:

$$\text{ac}_i = \frac{\sum_{j,k}^{N} \alpha_{i,j}, \alpha_{i,k}, \alpha_{j,k}}{\text{ad}_i(\text{d}_i - 1)}$$

where

- α is the admittance matrix
- ad_i is the admittive degree of the node i
- d_i is the degree of the node i

Return type 1d NumPy array (float)

Examples:

```
>>> res =  ResNetwork.SmallTestNetwork()
>>> print res.local_admittive_clustering()
[ 0.  0.00694444  0.0625  0.01077586  0. ]
>>> print type(res.local_admittive_clustering())
<type 'numpy.ndarray'>
```

update_R()

Updates R, the pseudo inverse of the admittance Laplacian

This function is run, whenever the admittance is changed.

Return type none

Examples:

```
>>> res = ResNetwork.SmallTestNetwork();print res.get_admittance()
[[ 0.     0.5    0.     0.     0.   ]
 [ 0.5    0.     0.125  0.5    0.   ]
 [ 0.     0.125  0.     0.125  0.   ]
 [ 0.     0.5    0.125  0.     0.1  ]
 [ 0.     0.     0.     0.1    0.   ]]
>>> print type( res.get_admittance() )
<type 'numpy.ndarray'>
```

update_admittance()

Updates admittance matrix which is inverse the resistances

Return type none

Examples:

```
>>> res = ResNetwork.SmallTestNetwork();print res.get_admittance()
[[ 0.     0.5    0.     0.     0.   ]
 [ 0.5    0.     0.125  0.5    0.   ]
 [ 0.     0.125  0.     0.125  0.   ]
 [ 0.     0.5    0.125  0.     0.1  ]
 [ 0.     0.     0.     0.1    0.   ]]
>>> print type(res.get_admittance())
<type 'numpy.ndarray'>
```

update_resistances(*resistances*)

Update the resistance matrix

This function is called to changed the resistance matrix. It sets the property and the calls the `update_admittance()` and `update_R()` functions.

Return type None

Examples:

```
>>> # test network with given resistances
>>> res = ResNetwork.SmallTestNetwork()
>>> print res.resistances
[[ 0  2  0  0  0]
 [ 2  0  8  2  0]
 [ 0  8  0  8  0]
 [ 0  2  8  0 10]
 [ 0  0  0 10  0]]
>>> # print admittance and admittance Laplacian
>>> print res.get_admittance()
[[ 0.     0.5    0.     0.     0.   ]
 [ 0.5    0.     0.125  0.5    0.   ]
 [ 0.     0.125  0.     0.125  0.   ]
 [ 0.     0.5    0.125  0.     0.1  ]
 [ 0.     0.     0.     0.1    0.   ]]
>>> print res.admittance_lapacian()
[[ 0.5   -0.5    0.     0.     0.   ]
 [-0.5    1.125 -0.125 -0.5    0.   ]
 [ 0.    -0.125  0.25  -0.125  0.   ]
 [ 0.    -0.5   -0.125  0.725 -0.1  ]
 [ 0.     0.     0.    -0.1    0.1  ]]
>>> # now update to unit resistance
>>> res.update_resistances(res.adjacency)
>>> # and check new admittance/admittance Laplacian
>>> print res.get_admittance()
[[ 0.  1.  0.  0.  0.]
 [ 1.  0.  1.  1.  0.]
 [ 0.  1.  0.  1.  0.]
 [ 0.  1.  1.  0.  1.]
 [ 0.  0.  0.  1.  0.]]
```

```
>>> print res.admittance_lapacian()
[[ 1. -1.  0.  0.  0.]
 [-1.  3. -1. -1.  0.]
 [ 0. -1.  2. -1.  0.]
 [ 0. -1. -1.  3. -1.]
 [ 0.  0.  0. -1.  1.]]
```

vertex_current_flow_betweenness(*i*)

Vertex Current Flow Betweeness (VCFB) of a node i.

The electrial version of Newmann's node betweeness is here defined as the Vertex Current Flow Betweeness (VCGB) of a node

$$VCFB_i := \frac{2}{n\,(n-1)} \sum_{s<t} I_i^{st}$$

where

$$\begin{aligned} I_i^{st} &= \frac{1}{2} \sum_j \Gamma_{i,j} |V_i - V_j| \\ &= \frac{1}{2} \sum_j \Gamma_{i,j} |I_s(R_{i,s} - R_{j,s}) + I_t(R_{j,t} - R_{i,t})| \end{aligned}$$

and further:

- $I_s^{st} := I_s$
- $I_t^{st} := I_t$
- Γ is the admittance matrix
- R is the pseudoinverse of the admittance Laplacian

Parameters **a** (*int*) – index of the "source" node

Return type NumPy float

Examples:

```
>>> res = ResNetwork.SmallTestNetwork()
>>> print "%.3f" % res.vertex_current_flow_betweenness(1)
0.389
>>> print "%.3f" % res.vertex_current_flow_betweenness(2)
0.044
```

5.2 climate

Constructing and analysing climate networks, related climate data analysis.

5.2.1 climate.climate_data

Provides classes for generating and analyzing complex climate networks.

class pyunicorn.climate.climate_data.**ClimateData**(*observable*, *grid*, *time_cycle*, *anomalies=False*, *observable_name=''*, *observable_long_name=None*, *window=None*, *silence_level=0*)

Bases: *pyunicorn.core.data.Data*

Encapsulates spatio-temporal climate data.

Provides methods to manipulate this data, i.e. calculate daily (monthly) mean values and anomaly values.

@ivar data_source: (string) - The name of the data source (model, reanalysis, station)

classmethod Load (*file_name*, *observable_name*, *time_cycle*, *time_name='time'*, *latitude_name='lat'*, *longitude_name='lon'*, *data_source=None*, *file_type='NetCDF'*, *window=None*, *vertical_level=None*, *silence_level=0*)
Initialize an instance of ClimateData.

Supported file types `file_type` are:

- "NetCDF" for regular (rectangular) grids
- "iNetCDF" for irregular (e.g. geodesic) grids or station data.

The spatio-temporal window is described by the following dictionary:

```
window = {"time_min": 0., "time_max": 0., "lat_min": 0.,
          "lat_max": 0., "lon_min": 0., "lon_max": 0.}
```

Parameters

- **`file_name`** (*str*) – The name of the data file.
- **`observable_name`** (*str*) – The short name of the observable within data file (particularly relevant for NetCDF).
- **`time_cycle`** (*int*) – The annual cycle length of the data (units of samples).
- **`time_name`** (*str*) – The name of the time variable within data file.
- **`latitude_name`** (*str*) – The name of the latitude variable within data file.
- **`longitude_name`** (*str*) – The name of longitude variable within data file.
- **`data_source`** (*str*) – The name of the data source (model, reanalysis, station).
- **`file_type`** (*str*) – The format of the data file.
- **`window`** (*dict*) – Spatio-temporal window to select a view on the data.
- **`vertical_level`** (*int*) – The vertical level to be extracted from the data file. Is ignored for horizontal data sets. If None, the first level in the data file is chosen.
- **`silence_level`** (*int*) – The inverse level of verbosity of the object.

static SmallTestData ()
Return test data set of 6 time series with 10 sampling points each.

Example:

```
>>> r(Data.SmallTestData().observable())
array([[ 0.    ,  1.    ,  0.    , -1.    , -0.    ,  1.    ],
       [ 0.309 ,  0.9511, -0.309 , -0.9511,  0.309 ,  0.9511],
       [ 0.5878,  0.809 , -0.5878, -0.809 ,  0.5878,  0.809 ],
       [ 0.809 ,  0.5878, -0.809 , -0.5878,  0.809 ,  0.5878],
       [ 0.9511,  0.309 , -0.9511, -0.309 ,  0.9511,  0.309 ],
       [ 1.    ,  0.    , -1.    , -0.    ,  1.    ,  0.    ],
       [ 0.9511, -0.309 , -0.9511,  0.309 ,  0.9511, -0.309 ],
       [ 0.809 , -0.5878, -0.809 ,  0.5878,  0.809 , -0.5878],
       [ 0.5878, -0.809 , -0.5878,  0.809 ,  0.5878, -0.809 ],
       [ 0.309 , -0.9511, -0.309 ,  0.9511,  0.309 , -0.9511]])
```

Return type ClimateData instance

Returns a ClimateData instance for testing purposes.

__init__ (*observable*, *grid*, *time_cycle*, *anomalies=False*, *observable_name=''*, *observable_long_name=None*, *window=None*, *silence_level=0*)
Initialize an instance of ClimateData.

The spatio-temporal window is described by the following dictionary:

```
window = {"time_min": 0., "time_max": 0., "lat_min": 0.,
          "lat_max": 0., "lon_min": 0., "lon_max": 0.}
```

Parameters

- **observable** (*2D array [time, index]*) – The array of time series to be represented by the Data instance.
- **grid** (Grid instance) – The Grid representing the spatial coordinates associated to the time series and their temporal sampling.
- **time_cycle** (*int*) – The annual cycle length of the data (units of samples).
- **anomalies** (*bool*) – Indicates whether the data are climatological anomaly values.
- **observable_name** (*str*) – A short name for the observable.
- **observable_long_name** (*str*) – A long name for the observable.
- **window** (*dict*) – Spatio-temporal window to select a view on the data.
- **silence_level** (*int*) – The inverse level of verbosity of the object.

__str__()

Returns a string representation.

_calculate_anomaly()

Calculate anomaly time series from observable.

To obtain climatological anomaly time series, the climatological means are subtracted from each sample in the original time series. This procedure is also known as phase averaging.

Note: Only the currently selected spatio-temporal window is considered.

Return type 2D Numpy array [time, node index]

Returns the anomalized time series.

_calculate_phase_mean()

Calculate mean values of observable for each phase of the annual cycle.

This is also commonly referred to as climatological mean, e.g., the mean temperature for all Januaries in the data set for monthly time resolution (time_cycle=12).

Note: Only the currently selected spatio-temporal window is considered.

Return type 2D Numpy array [cycle index, node index]

Returns the mean values of observable for each phase of the annual cycle.

anomaly()

Return anomaly time series from observable.

For further comments, see *_calculate_anomaly()*.

Note: Only the currently selected spatio-temporal window is considered.

Example:

```
>>> r(ClimateData.SmallTestData().anomaly()[:,0])
array([-0.5 , -0.321 , -0.1106,  0.1106,  0.321 ,
        0.5 ,  0.321 ,  0.1106, -0.1106, -0.321 ])
```

Return type 2D Numpy array [time, node index]

Returns the anomalized time series.

anomaly_selected_months(*selected_months*)

Return anomaly time series from observable for selected months.

For further comments, see `_calculate_anomaly()`.

Note: Only the currently selected spatio-temporal window is considered.

Parameters **selected_months** (*[number]*) – The selected months.

Return type 2D array [time, node index]

Returns the anomalized time series for selected months.

clear_cache()

Clean up cache.

Is reversible, since all cached information can be recalculated from basic data.

indices_selected_months(*selected_months*)

Return sorted time indices associated to certain months.

Currently, only cycle lengths of 12 (monthly data) and 360 (standardized daily data) are supported.

Note: Only the currently selected spatio-temporal window is considered.

Parameters **selected_months** (*[number]*) – The selected months.

Return type 1D array (int)

Returns the sorted time indices corresponding to chosen months.

indices_selected_phases(*selected_phases*)

Return sorted time indices associated to certain phase indices.

Note: Only the currently selected spatio-temporal window is considered.

Example:

```
>>> ClimateData.SmallTestData().indices_selected_phases([0,1,4])
array([0, 1, 4, 5, 6, 9])
```

Parameters **selected_phases** (*[int]*) – The selected phase indices.

Return type 1D array (int)

Returns the sorted time indices corresponding to chosen phase indices.

phase_indices()

Return time indices associated to all phases in the annual cycle.

In other words, provides all time indices falling into a particular day, month etc. of the year.

Just includes measurements from years for which complete data exists.

Note: Only the currently selected spatio-temporal window is considered.

Note: Only the currently selected spatio-temporal window is considered.

Example:

```
>>> ClimateData.SmallTestData().phase_indices()
array([[0, 5], [1, 6], [2, 7], [3, 8], [4, 9]])
```

Return type 2D Numpy array (int) [phase index, year]

Returns the time indices associated to all phases of the annual cycle.

phase_mean()

Return mean values of observable for each phase of the annual cycle.

For further comments, see *_calculate_phase_mean()*.

Note: Only the currently selected spatio-temporal window is considered.

Example:

```
>>> r(ClimateData.SmallTestData().phase_mean())
array([[ 0.5   ,  0.5   , -0.5   , -0.5   ,  0.5   ,  0.5   ],
       [ 0.63  ,  0.321 , -0.63  , -0.321 ,  0.63  ,  0.321 ],
       [ 0.6984,  0.1106, -0.6984, -0.1106,  0.6984,  0.1106],
       [ 0.6984, -0.1106, -0.6984,  0.1106,  0.6984, -0.1106],
       [ 0.63  , -0.321 , -0.63  ,  0.321 ,  0.63  , -0.321 ]])
```

Return type 2D Numpy array [cycle index, node index]

Returns the mean values of observable for each phase of the annual cycle.

set_global_window()

Set the view on the whole data set.

Select the full data set and creates a data array as well as a corresponding Grid object to access this window from outside.

Example (Set smaller window and subsequently restore global window):

```
>>> data = ClimateData.SmallTestData()
>>> data.set_window(window={"time_min": 0., "time_max": 4.,
...                 "lat_min": 10., "lat_max": 20.,
...                 "lon_min": 5.,  "lon_max": 10.})
>>> data.grid.grid()["lat"]
array([ 10.,  15.], dtype=float32)
>>> data.set_global_window()
>>> data.grid.grid()["lat"]
array([  0.,   5.,  10.,  15.,  20.,  25.], dtype=float32)
```

set_window(*window*)

Set spatio-temporal window.

Calls set_window method of parent class Data and additionally sets flags, so that measures derived from data (mean, anomaly) will be recalculated for new window.

The spatio-temporal window is described by the following dictionary:

```
window = {"time_min": 0., "time_max": 0., "lat_min": 0.,
          "lat_max": 0., "lon_min": 0., "lon_max": 0.}
```

If the temporal boundaries are equal, the data's full time range is selected. If any of the two corresponding spatial boundaries are equal, the data's full spatial extension is included.

For more information see `pyunicorn.Data.set_window()`.

Example:

```
>>> data = ClimateData.SmallTestData()
>>> data.set_window(window={"time_min": 0., "time_max": 0.,
...                 "lat_min": 10., "lat_max": 20.,
...                 "lon_min": 5.,  "lon_max": 10.})
>>> r(data.anomaly())
array([[ 0.5   , -0.5   ], [ 0.321 , -0.63  ], [ 0.1106, -0.6984],
       [-0.1106, -0.6984], [-0.321 , -0.63  ], [-0.5   ,  0.5   ],
       [-0.321 ,  0.63  ], [-0.1106,  0.6984], [ 0.1106,  0.6984],
       [ 0.321 ,  0.63  ]])
```

Parameters **window** (*dictionary*) – The spatio-temporal window to select a view on the data.

shuffled_anomaly()

Return the randomly shuffled anomaly time series.

Each anomaly time series is shuffled individually.

Note: Only the currently selected spatio-temporal window is considered.

Example (Anomaly with and without temporal shuffling should have the same standard deviation along time axis):

```
>>> r(ClimateData.SmallTestData().anomaly().std(axis=0))
array([ 0.31 , 0.6355, 0.31 , 0.6355, 0.31 , 0.6355])
>>> r(ClimateData.SmallTestData().shuffled_anomaly().std(axis=0))
array([ 0.31 , 0.6355, 0.31 , 0.6355, 0.31 , 0.6355])
```

Return type 2D Numpy array [time, node index]

Returns the anomalized and shuffled time series.

time_cycle = None

(number (int)) - The annual cycle length of the data (units of samples).

5.2.2 climate.climate_network

Provides classes for generating and analyzing complex climate networks.

class pyunicorn.climate.climate_network.**ClimateNetwork**(*grid*, *similarity_measure*, *threshold=None*, *link_density=None*, *non_local=False*, *directed=False*, *node_weight_type='surface'*, *silence_level=0*)

Bases: *pyunicorn.core.geo_network.GeoNetwork*

Encapsulates a similarity network embedded on a spherical surface.

Particularly provides functionality to generate a complex network from the matrix of a similarity measure of time series.

The analysis of climate time series based on similarity networks was first introduced in *[Tsonis2004]*.

static Load(*filename_network*, *filename_grid*, *filename_similarity_measure*, *fileformat=None*, **args*, ***kwds*)

Return a ClimateNetwork object stored in files.

Unified reading function for graphs. Relies on and partially extends the corresponding igraph function. Refer to igraph documentation for further details on the various reader methods for different formats.

This method tries to identify the format of the graph given in the first parameter and calls the corresponding reader method.

Existing node and link attributes/weights are also restored depending on the chosen file format. E.g., the formats GraphML and gzipped GraphML are able to store both node and link weights.

The remaining arguments are passed to the reader method without any changes.

Parameters

- **`filename_network`** (*str*) – The name of the file where the Network object is to be stored.
- **`filename_grid`** (*str*) – The name of the file where the Grid object is to be stored (including ending).
- **`filename_similarity_measure`** (*str*) – The name of the file where the similarity measure matrix is to be stored.
- **`fileformat`** (*str*) – the format of the file (if known in advance) `None` means auto-detection. Possible values are: `"ncol"` (NCOL format), `"lgl"` (LGL format), `"graphml"`, `"graphmlz"` (GraphML and gzipped GraphML format), `"gml"` (GML format), `"net"`, `"pajek"` (Pajek format), `"dimacs"` (DIMACS format), `"edgelist"`, `"edges"` or `"edge"` (edge list), `"adjacency"` (adjacency matrix), `"pickle"` (Python pickled format).

Returns `ClimateNetwork` instance.

static **`SmallTestNetwork`**()

Return a 6-node undirected test climate network from a similarity matrix.

The network looks like this:

```
    3 - 1
    |   | \
5 - 0 - 4 - 2
```

Example:

```
>>> r(ClimateNetwork.SmallTestNetwork().adjacency)
array([[0, 0, 0, 1, 1, 1], [0, 0, 1, 1, 1, 0], [0, 1, 0, 0, 1, 0],
       [1, 1, 0, 0, 0, 0], [1, 1, 1, 0, 0, 0], [1, 0, 0, 0, 0, 0]])
```

Return type `Network` instance

`__init__`(*grid*, *similarity_measure*, *threshold=None*, *link_density=None*, *non_local=False*, *directed=False*, *node_weight_type='surface'*, *silence_level=0*)

Initialize an instance of `ClimateNetwork`.

Note: Either threshold **OR** link_density have to be given!

Possible choices for `node_weight_type`:

- None (constant unit weights)
- “surface” (cos lat)
- “irrigation” (cos**2 lat)

Parameters

- **`grid`** (`Grid`) – The Grid object describing the network’s spatial embedding.
- **`similarity_measure`** (*2D array [index, index]*) – The similarity measure for all pairs of nodes.

- **threshold** (*float*) – The threshold of similarity measure, above which two nodes are linked in the network.
- **link_density** (*float*) – The networks's desired link density.
- **non_local** (*bool*) – Determines, whether links between spatially close nodes should be suppressed.
- **directed** (*bool*) – Determines, whether the network is treated as directed.
- **node_weight_type** (*str*) – The type of geographical node weight to be used.
- **silence_level** (*int*) – The inverse level of verbosity of the object.

__str__()

Return a string representation of the ClimateNetwork object.

Example:

```
>>> print ClimateNetwork.SmallTestNetwork()
ClimateNetwork:
GeoNetwork:
Network: undirected, 6 nodes, 7 links, link density 0.467.
Geographical boundaries:
         time     lat     lon
   min    0.0    0.00    2.50
   max    9.0   25.00   15.00
Threshold: 0.5
Local connections filtered out: False
```

_calculate_non_local_adjacency(*similarity_measure*, *threshold*, *a=20*, *d_min=0.05*)

Return the adjacency matrix with suppressed spatially local links.

Physically trivial links between geographically close nodes are removed.

For large a, d_min corresponds to the minimum distance for which links are allowed to exist.

Example:

```
>>> net = ClimateNetwork.SmallTestNetwork()
>>> net._calculate_non_local_adjacency(
...     similarity_measure=net.similarity_measure(),
...     threshold=0.5, a=30, d_min=0.20)
array([[0, 0, 0, 1, 1, 1], [0, 0, 0, 0, 1, 0],
       [0, 0, 0, 0, 0, 0], [1, 0, 0, 0, 0, 0],
       [1, 1, 0, 0, 0, 0], [1, 0, 0, 0, 0, 0]], dtype=int8)
```

Parameters

- **similarity_measure** (*2D Numpy array [index, index]*) – The similarity measure for all pairs of nodes.
- **threshold** (*number (float)*) – The threshold of similarity measure, above which two nodes are linked in the network.
- **a** (*number (float)*) – The steepness parameter of the distance weighting function in the transition region from not including any links (weight=0) to including all links (weight=1).
- **d_min** (*number (float)*) – The parameter controlling the minimum distance, above which links can be included in the network (unit radians).

Return type 2D Numpy array (int8) [index, index]

Returns the network's adjacency matrix.

_calculate_threshold_adjacency(*similarity_measure*, *threshold*)
Extract the network's adjacency matrix by thresholding.

The resulting network is a simple graph, i.e., self-loops and multiple links are not allowed.

Example (Threshold zero should yield a fully connected network given the test similarity matrix):

```
>>> net = ClimateNetwork.SmallTestNetwork()
>>> net._calculate_threshold_adjacency(
...     similarity_measure=net.similarity_measure(), threshold=0.0)
array([[0, 1, 1, 1, 1, 1], [1, 0, 1, 1, 1, 1],
       [1, 1, 0, 1, 1, 1], [1, 1, 1, 0, 1, 1],
       [1, 1, 1, 1, 0, 1], [1, 1, 1, 1, 1, 0]], dtype=int8)
```

Parameters

- **similarity_measure** (*2D Numpy array [index, index]*) – The similarity measure for all pairs of nodes.
- **threshold** (*number (float)*) – The threshold of similarity measure, above which two nodes are linked in the network.

Return type 2D Numpy array (int8) [index, index]

Returns the network's adjacency matrix.

_regenerate_network()
Regenerate the current climate network according to new similarity measure.

clear_cache(*irreversible=False*)
Clean up cache.

If irreversible=True, the network cannot be recalculated using a different threshold, or link density.

Parameters **irreversible** (*bool*) – The irreversibility of clearing the cache.

correlation_distance(**args*, ***kwargs*)
Return correlation weighted distances between nodes.

Defined as the elementwise product of the correlation measure and angular great circle distance matrices.

This is a useful measure of the relative importance of links, since links with high geographical distance and high correlation (teleconnections) get the highest weight. Trivial correlations with small geographical distance and high correlation get a lower weight.

Correlation distance appears to be the simplest functional form of combining geographical distance and correlation measure that yields meaningful results.

Example:

```
>>> r(ClimateNetwork.SmallTestNetwork().correlation_distance())
array([[ 0.    , 0.0098, 0.039 , 0.1752, 0.2721, 0.2666],
       [ 0.0098, 0.    , 0.0536, 0.175 , 0.2911, 0.1162],
       [ 0.039 , 0.0536, 0.0003, 0.0194, 0.155 , 0.029 ],
       [ 0.1752, 0.175 , 0.0097, 0.0005, 0.0097, 0.0578],
       [ 0.2721, 0.2911, 0.155 , 0.0097, 0.    , 0.0384],
       [ 0.2666, 0.1162, 0.029 , 0.0578, 0.0384, 0.0005]])
```

Return type 2D matrix [index, index]

Returns the correlation distance matrix.

correlation_distance_weighted_closeness()
Return correlation distance weighted closeness.

Calculates the sequence of closeness centralities link-weighted by the inverse of correlation distance between nodes. For closeness centrality calculation, the inverse of correlation distance is used, because high values of this measure should correspond to short distances in the graph and vice versa when weighted shortest paths are calculated.

Example:

```
>>> r(ClimateNetwork.SmallTestNetwork().          correlation_distance_weighted_clo
array([ 0.1646,  0.1351,  0.0894,  0.1096,  0.1659,  0.1102])
```

Return type 1D Numpy array [index]

Returns the correlation distance weighted closeness sequence.

inv_correlation_distance(**args*, ***kwargs*)
Return correlation weighted distances between nodes.

Return type 2D matrix [index, index]

link_density_function(*n_bins*)
Return the network's link density as a function of the threshold.

Example:

```
>>> r(ClimateNetwork.SmallTestNetwork().          link_density_function(3)[0])
array([ 0.  ,  0.3889,  0.6667])
>>> r(ClimateNetwork.SmallTestNetwork().          link_density_function(3)[1])
array([ 0.1,  0.4,  0.7,  1. ])
```

Parameters **n_bins** (*number (int)*) – The number of bins.

Return type tuple of two 1D Numpy arrays [bin]

Returns the network's link density in dependence on threshold.

local_correlation_distance_weighted_vulnerability()
Return local correlation distance weighted vulnerability.

Calculates the sequence of vulnerabilities link-weighted by the inverse of correlation distance between nodes. For vulnerability calculation, the inverse of correlation distance is used, because high values of this measure should correspond to short distances in the graph and vice versa when weighted shortest paths are calculated.

Example:

```
>>> r(ClimateNetwork.SmallTestNetwork().          local_correlation_distance_weight
array([ 0.4037,  0.035 , -0.1731, -0.081 ,  0.3121, -0.0533])
```

Return type 1D Numpy array

Returns the local correlation distance weighted vulnerability sequence.

non_local()
Indicate if links between spatially close nodes were suppressed.

Example:

```
>>> ClimateNetwork.SmallTestNetwork().non_local()
False
```

Return bool Determines, whether links between spatially close nodes should be suppressed.

save (*filename_network*, *filename_grid=None*, *filename_similarity_measure=None*, *fileformat=None*, **args*, ***kwds*)

Save the ClimateNetwork object to files.

Unified writing function for graphs. Relies on and partially extends the corresponding igraph function. Refer to igraph documentation for further details on the various writer methods for different formats.

This method tries to identify the format of the graph given in the first parameter (based on extension) and calls the corresponding writer method.

Existing node and link attributes/weights are also stored depending on the chosen file format. E.g., the formats GraphML and gzipped GraphML are able to store both node and link weights.

Note: The similarity measure matrix and grid are not stored if the corresponding filenames are None.

The remaining arguments are passed to the writer method without any changes.

Parameters

- **filename_network** (*str*) – The name of the file where the Network object is to be stored.
- **filename_grid** (*str*) – The name of the file where the Grid object is to be stored (including ending).
- **filename_similarity_measure** (*str*) – The name of the file where the similarity measure matrix is to be stored.
- **fileformat** (*str*) – the format of the file (if one wants to override the format determined from the filename extension, or the filename itself is a stream). `None` means auto-detection. Possible values are: `"ncol"` (NCOL format), `"lgl"` (LGL format), `"graphml"`, `"graphmlz"` (GraphML and gzipped GraphML format), `"gml"` (GML format), `"dot"`, `"graphviz"` (DOT format, used by GraphViz), `"net"`, `"pajek"` (Pajek format), `"dimacs"` (DIMACS format), `"edgelist"`, `"edges"` or `"edge"` (edge list), `"adjacency"` (adjacency matrix), `"pickle"` (Python pickled format), `"svg"` (Scalable Vector Graphics).

set_link_density (*link_density*)

Generate climate network by thresholding with prescribed link density.

Note: The desired link density can only be achieved approximately in most cases.

Example:

```
>>> net = ClimateNetwork.SmallTestNetwork()
>>> r(net.link_density)
0.4667
>>> net.set_link_density(link_density=0.7)
>>> r(net.link_density)
0.6667
```

Parameters **link_density** (*number (float)*) – The networks's desired link density.

set_non_local (*non_local*)

Toggle suppression of links between spatially close nodes.

Example:

```
>>> net = ClimateNetwork.SmallTestNetwork()
>>> net.set_non_local(non_local=True)
>>> r(net.adjacency)
array([[0, 0, 0, 1, 1, 1], [0, 0, 0, 1, 1, 0], [0, 0, 0, 0, 1, 0],
       [1, 1, 0, 0, 0, 0], [1, 1, 1, 0, 0, 0], [1, 0, 0, 0, 0, 0]])
```

Parameters **`non_local`** (*bool*) – Determines, whether links between spatially close nodes should be suppressed.

`set_threshold`(*threshold*)

Generate climate network by thresholding similarity matrix.

Example (Number of links decreases as threshold increases):

```
>>> net = ClimateNetwork.SmallTestNetwork()
>>> net.n_links
7
>>> net.set_threshold(threshold=0.7)
>>> net.n_links
3
```

Parameters **`threshold`** (*number (float)*) – the threshold used to generate the current climate network.

`similarity_measure`()

Return the similarity measure used for network construction.

Example:

```
>>> r(ClimateNetwork.SmallTestNetwork().similarity_measure()[0,:])
array([ 1.  , 0.1 , 0.2 , 0.6 , 0.7 , 0.55])
```

Return type 2D Numpy array [index, index]

Returns The similarity measure for all pairs of nodes.

`threshold`()

Return the threshold used to generate the current climate network.

Example:

```
>>> ClimateNetwork.SmallTestNetwork().threshold()
0.5
```

Return type number (float)

Returns the threshold used to generate the current climate network.

`threshold_from_link_density`(*link_density*)

Return the threshold for network construction given link density.

Example:

```
>>> r(ClimateNetwork.SmallTestNetwork().                threshold_from_link_density(link_
0.4
```

Parameters **`link_density`** (*number (float)*) – The networks's desired link density.

Return type number (float)

Returns The threshold of similarity measure, above which two nodes are linked in the network.

5.2.3 climate.coupled_climate_network

Provides classes for generating and analyzing complex coupled climate networks.

class `pyunicorn.climate.coupled_climate_network.`**`CoupledClimateNetwork`**(*grid_1, grid_2, similarity_measure, threshold=None, link_density=None, non_local=False, directed=False, node_weight_type='surface', silence_level=0*)

Bases: *`pyunicorn.core.interacting_networks.InteractingNetworks`*, *`pyunicorn.climate.climate_network.ClimateNetwork`*

Encapsulates a coupled similarity network embedded on a spherical surface.

Particularly provides functionality to generate a complex network from the matrix of a similarity measure of time series from two different observables (temperature, pressure), vertical levels etc.

So far, most methods only give meaningful results for undirected networks!

The idea of coupled climate networks is based on the concept of coupled patterns, for a review refer to *[Bretherton1992]*.

Note: The two observables (layers) need to have the same time grid (temporal sampling points).

`N` **= None**
(number (int)) - The total number of nodes in both layers.

`N_1` **= None**
(number (int)) - The number of nodes in the first layer.

`N_2` **= None**
(number (int)) - The number of nodes in the second layer.

`__init__`(*grid_1, grid_2, similarity_measure, threshold=None, link_density=None, non_local=False, directed=False, node_weight_type='surface', silence_level=0*)
Initialize an instance of CoupledClimateNetwork.

Note: Either threshold **OR** link_density have to be given!

Possible choices for `node_weight_type`:

- None (constant unit weights)
- "surface" (cos lat)
- "irrigation" (cos**2 lat)

Parameters

- **`grid_1`** (*`Grid`*) – The Grid object describing the first layer's spatial embedding.
- **`grid_2`** (*`Grid`*) – The Grid object describing the second layer's spatial embedding.
- **`similarity_measure`** (*2D array [index, index]*) – The similarity measure for all pairs of nodes.
- **`threshold`** (*float*) – The threshold of similarity measure, above which two nodes are linked in the network.
- **`link_density`** (*float*) – The networks's desired link density.

- **non_local** (*bool*) – Determines, whether links between spatially close nodes should be suppressed.
- **directed** (*bool*) – Determines, whether the network is treated as directed.
- **strnode_weight_type** – The type of geographical node weight to be used.
- **silence_level** (*int*) – The inverse level of verbosity of the object.

__str__()
Return a string representation of CoupledClimateNetwork object.

adjacency_1()
Return internal adjacency matrix of first layer.

Return type 2D Numpy array [index_1, index_1]

Returns the internal adjacency matrix of first layer.

adjacency_2()
Return internal adjacency matrix of second layer.

Return type 2D Numpy array [index_2, index_2]

Returns the internal adjacency matrix of second layer.

cross_average_path_length(*link_attribute=None*)
Return cross average path length.

Return the average (weighted) shortest path length between all pairs of nodes from different layers only.

Parameters **link_attribute** (*str*) – Optional name of the link attribute to be used as the links' length. If None, links have length 1. (Default: None)

Return float the cross average path length.

cross_betweenness()
Return the cross betweenness sequence.

Gives the normalized number of shortest paths only between nodes from **different** layers, in which a node i is contained. This is equivalent to the inter-regional / inter-group betweenness with respect to layer 1 and layer 2.

Return type tuple of two 1D arrays [index]

Returns the cross betweenness sequence.

cross_closeness(*link_attribute=None*)
Return cross closeness sequence.

Gives the inverse average geodesic distance from a node in one layer to all nodes in the other layer.

Parameters **link_attribute** (*str*) – Optional name of the link attribute to be used as the links' length. If None, links have length 1. (Default: None)

Return type tuple of two 1D arrays [index]

Returns the cross closeness sequence.

cross_degree()
Return the cross degree sequences.

Gives the number of links from a specific node in one layer to the other layer.

Return type tuple of two 1D arrays [index]

Returns the cross degree sequences.

`cross_global_clustering`()

Return global cross clustering for coupled network.

The global cross clustering coefficient C_v gives the average probability, that two randomly drawn neighbors in layer 2 of node v in layer 1 are also neighbors and vice versa. It counts triangles having one vertex in layer 1 and two vertices in layer 2 and vice versa.

Return type (float, float)

Returns the cross global clustering coefficients.

`cross_layer_adjacency`()

Return cross adjacency matrix of the coupled network.

The cross adjacency matrix entry $CA_{ij} = 1$ describes that node i in the first layer is linked to node j in the second layer. Vice versa, $CA_{ji} = 1$ indicates that node j in the first layer is linked to node i in the second layer.

Note: Cross adjacency matrix is **NEITHER** square **NOR** symmetric in general!

Return type 2D Numpy array [index_1, index_2]

Returns the cross adjacency matrix.

`cross_link_density`()

Return the density of links between the two layers.

Return float the density of links between the two layers.

`cross_local_clustering`()

Return local cross clustering for coupled network.

The local cross clustering coefficient C_v gives the probability, that two randomly drawn neighbors in layer 2 of node v in layer 1 are also neighbors and vice versa. It counts triangles having one vertex in layer 1 and two vertices in layer 2 and vice versa.

Return type tuple of two 1D arrays [index]

Returns the cross local clustering coefficients.

`cross_path_lengths`(*link_attribute=None*)

Return cross path length matrix.

Contains the path length between nodes from different layers. The paths contain nodes from both layers.

Parameters **`link_attribute`** (*str*) – Optional name of the link attribute to be used as the links' length. If None, links have length 1. (Default: None)

Return type 2D array [index_1, index_2]

Returns the cross path length matrix.

`cross_similarity_measure`()

Return cross similarity measure matrix.

Note: Cross similarity measure matrix is NEITHER square NOR symmetric in general!

Return type 2D Numpy array [index_1, index_2]

Returns the cross similarity measure matrix.

`cross_transitivity`()

Return cross transitivity for coupled network.

The cross transitivity is the probability, that two randomly drawn neighbors in layer 2 of node v in layer 1 are also neighbors and vice versa. It counts triangles having one vertex in layer 1 and two vertices in layer 2 and vice versa. Cross transitivity tends to weight low cross degree vertices less strongly when compared to the global cross clustering coefficient (see *[Newman2003]*).

Return type (float, float)

Returns the cross transitivities.

`grid_1` **= None**

(Grid) - The Grid object describing the first layer's spatial embedding.

`grid_2` **= None**

(Grid) - The Grid object describing the second layer's spatial embedding.

`internal_average_path_length`(*link_attribute=None*)

Return internal average path length.

Return the average (weighted) shortest path length between all pairs of nodes within each layer separately for which a path exists. Paths between nodes from different layers are not included in the averages!

However, even if the end points lie within the same layer, the paths themselves will generally contain nodes from both layers. To avoid this and only consider paths lying within layer i, do the following:

```
net_i = coupled_network.network_i()
path_lengths_i = net_i.path_lengths(link_attribute)
```

Parameters **`link_attribute`** (*str*) – Optional name of the link attribute to be used as the links' length. If None, links have length 1. (Default: None)

Return type (float, float)

Returns the internal average path length.

`internal_betweenness_1`()

Return the internal betweenness sequences for layer 1.

Gives the normalized number of shortest paths only between nodes from layer 1, in which a node i is contained. i can be member of any of the two layers. This is equivalent to the inter-regional / inter-group betweenness with respect to layer 1 and layer 1.

Return type tuple of two 1D arrays [index]

Returns the internal betweenness sequence for layer 1.

`internal_betweenness_2`()

Return the internal betweenness sequences for layer 2.

Gives the normalized number of shortest paths only between nodes from layer 2, in which a node i is contained. i can be member of any of the two layers. This is equivalent to the inter-regional / inter-group betweenness with respect to layer 2 and layer 2.

Return type tuple of two 1D arrays [index]

Returns the internal betweenness sequence for layer 2.

`internal_closeness`(*link_attribute=None*)

Return internal closeness sequence.

Gives the inverse average geodesic distance from a node to all other nodes in the same layer.

However, the included paths will generally contain nodes from both layers. To avoid this, do the following:

```
net_i = coupled_network.network_i()
closeness_i = net_i.closeness(link_attribute)
```

Parameters **link_attribute** (*str*) – Optional name of the link attribute to be used as the links' length. If None, links have length 1. (Default: None)

Return type tuple of two 1D arrays [index]

Returns the internal closeness sequence.

internal_degree()
Return the internal degree sequences.

Gives the number of links from a specific node to other nodes within the same layer.

Return type tuple of two 1D arrays [index]

Returns the internal degree sequences.

internal_global_clustering()
Return global clustering coefficients for each layer separately.

Internal global clustering coefficients are calculated as mean values from the local clustering sequence of the whole coupled network. This implies that triangles spanning both layers will generally contribute to the internal clustering coefficients.

To avoid this and consider only triangles lying within each layer:

```
net_1 = coupled_network.network_1()
clustering_1 = net_1.global_clustering()
net_2 = coupled_network.network_2()
clustering_2 = net_2.global_clustering()
```

Return type (float, float)

Returns the internal global clustering coefficients.

internal_link_density()
Return the density of links within the two layers.

Return type (float, float)

Returns the density of links within the two layers.

network_1()
Return network consisting of layer 1 nodes and their internal links.

This can be used to conveniently analyze the layer 1 separately, e.g., for calculation network measures solely for layer 1.

Return type *GeoNetwork*

Returns the network consisting of layer 1 nodes and their internal links.

network_2()
Return network consisting of layer 2 nodes and their internal links.

This can be used to conveniently analyze the layer 2 separately, e.g., for calculation network measures solely for layer 2.

Return type *GeoNetwork*

Returns the network consisting of layer 2 nodes and their internal links.

nodes_1 = None
(list (int)) - List of node indices for first layer

nodes_2 = None
(list (int)) - List of node indices for second layer

number_cross_layer_links()
Return the number of links between the two layers.

Return int the number of links between nodes from different layers.

number_internal_links()

Return the number of links within each layer.

Return type (int, int)

Returns the number of links within each layer.

path_lengths_1(*link_attribute=None*)

Return internal path length matrix of first layer.

Contains the paths length between all pairs of nodes within layer 1. However, the paths themselves will generally contain nodes from both layers. To avoid this and only consider paths lying within layer 1, do the following:

```
net_1 = coupled_network.network_1()
path_lengths_1 = net_1.path_lengths(link_attribute)
```

Parameters **link_attribute** (*str*) – Optional name of the link attribute to be used as the links' length. If None, links have length 1. (Default: None)

Return type 2D array [index_1, index_1]

Returns the internal path length matrix of first layer.

path_lengths_2(*link_attribute=None*)

Return internal path length matrix of second layer.

Contains the path lengths between all pairs of nodes within layer 2. However, the paths themselves will generally contain nodes from both layers. To avoid this and only consider paths lying within layer 2, do the following:

```
net_2 = coupled_network.network_2()
path_lengths_2 = net_2.path_lengths(link_attribute)
```

Parameters **link_attribute** (*str*) – Optional name of the link attribute to be used as the links' length. If None, links have length 1. (Default: None)

Return type 2D array [index_2, index_2]

Returns the internal path length matrix of second layer.

similarity_measure_1()

Return internal similarity measure matrix of first layer.

Return type 2D Numpy array [index_1, index_1]

Returns the internal similarity measure matrix of first layer.

similarity_measure_2()

Return internal similarity measure matrix of second layer.

Return type 2D Numpy array [index_2, index_2]

Returns the internal similarity measure matrix of first layer.

5.2.4 climate.coupled_tsonis

Provides classes for generating and analyzing complex coupled climate networks.

class `pyunicorn.climate.coupled_tsonis.`**`CoupledTsonisClimateNetwork`**(*data_1, data_2, threshold=None, link_density=None, non_local=False, node_weight_type='surface', selected_months=None, silence_level=0*)

Bases: `pyunicorn.climate.coupled_climate_network.CoupledClimateNetwork`

Encapsulates a coupled similarity network embedded on a spherical surface.

Particularly provides functionality to generate a complex network from the Pearson correlation matrix of time series from two different observables (temperature, pressure), vertical levels etc.

Construct a static climate network following Tsonis et al. from the Pearson correlation matrix at zero lag *[Tsonis2004]*.

Hence, coupled Tsonis climate networks are undirected due to the symmetry of the correlation matrix.

The idea of coupled climate networks is based on the concept of coupled patterns, for a review refer to *[Bretherton1992]*.

Note: The two observables (layers) need to have the same time grid (temporal sampling points).

`__init__`(*data_1, data_2, threshold=None, link_density=None, non_local=False, node_weight_type='surface', selected_months=None, silence_level=0*)

Initialize an instance of `CoupledTsonisClimateNetwork`.

Note: Either threshold **OR** link_density have to be given!

Possible choices for `node_weight_type`:

- `None` (constant unit weights)
- `"surface"` (cos lat)
- `"irrigation"` (cos**2 lat)

Parameters

- **`data_1`** (`ClimateData`) – The climate data for the first layer.
- **`data_2`** (`ClimateData`) – The climate data for the second layer.
- **`threshold`** (*float*) – The threshold of similarity measure, above which two nodes are linked in the network.
- **`link_density`** (*float*) – The networks's desired link density.
- **`non_local`** (*bool*) – Determines, whether links between spatially close nodes should be suppressed.
- **`node_weight_type`** (*str*) – The type of geographical node weight to be used.
- **`]selected_months`** (*[int*) – The months for which to calculate the correlation matrix. The full time series are used for default value None.
- **`silence_level`** (*int*) – The inverse level of verbosity of the object.

`__str__`()

Return a string representation of CoupledClimateNetwork object.

`_calculate_correlation`(*anomaly_1*, *anomaly_2*)
Return the correlation matrix at zero lag.

Parameters

- **`anomaly_1`** (*2D Numpy array (time, index_1)*) – the first set of anomaly time series from which to calculate the correlation matrix at zero lag.
- **`anomaly_2`** (*2D Numpy array (time, index_2)*) – the second set of anomaly time series from which to calculate the correlation matrix at zero lag.

Return type 2D Numpy array (index, index)

Returns the correlation matrix at zero lag.

`calculate_similarity_measure`(*anomaly_1*, *anomaly_2*)
Encapsulate the calculation of the correlation matrix at zero lag.

Parameters

- **`anomaly_1`** (*2D Numpy array (time, index_1)*) – the first set of anomaly time series from which to calculate the correlation matrix at zero lag.
- **`anomaly_2`** (*2D Numpy array (time, index_2)*) – the second set of anomaly time series from which to calculate the correlation matrix at zero lag.

Return type 2D Numpy array (index, index)

Returns the correlation matrix at zero lag.

`correlation`()
Return the coupled correlation matrix at zero lag.

Return type 2D Numpy array (index, index)

Returns the correlation matrix at zero lag.

`silence_level` = None
(string) - The inverse level of verbosity of the object.

5.2.5 climate.havlin

Provides classes for generating and analyzing complex climate networks.

class `pyunicorn.climate.havlin.`**`HavlinClimateNetwork`**(*data*, *max_delay*, *threshold=None*, *link_density=None*, *non_local=False*, *node_weight_type='surface'*, *silence_level=0*)

Bases: *`pyunicorn.climate.climate_network.ClimateNetwork`*

Encapsulates a Havlin climate network.

The similarity matrix associated with a Havlin climate network is the maximum-lag correlation matrix with each entry normalized by the cross-correlation function's standard deviation.

Havlin climate networks are undirected so far.

Havlin climate networks were studied for daily data in *[Yamasaki2008]*, *[Gozolchiani2008]*, *[Yamasaki2009]*.

Note: So far, the cross-correlation functions are estimated using convolution in Fourier space (FFT). This may not be reliable for larger delays.

`__init__`(*data*, *max_delay*, *threshold=None*, *link_density=None*, *non_local=False*, *node_weight_type='surface'*, *silence_level=0*)
Initialize an instance of HavlinClimateNetwork.

Note: Either threshold **OR** link_density have to be given!

Possible choices for `node_weight_type`:

- None (constant unit weights)
- "surface" (cos lat)
- "irrigation" (cos**2 lat)

Parameters

- **`data`** (*ClimateData*) – The climate data used for network construction.
- **`threshold`** (*float*) – The threshold of similarity measure, above which two nodes are linked in the network.
- **`link_density`** (*float*) – The networks's desired link density.
- **`max_delay`** (*int*) – Maximum delay for cross-correlation functions.
- **`non_local`** (*bool*) – Determines, whether links between spatially close nodes should be suppressed.
- **`node_weight_type`** (*str*) – The type of geographical node weight to be used.
- **`silence_level`** (*int*) – The inverse level of verbosity of the object.

`__str__`()

Return a string version of the instance of HavlinClimateNetwork.

`_calculate_correlation_strength`(*anomaly*, *max_delay*, *gamma=0.2*)

Calculate correlation strength and maximum lag matrices.

Follows the method described in *[Yamasaki2008]*.

Also returns the time lag at maximum correlation for each link.

Parameters

- **`anomaly`** (*2D array [time, index]*) – The anomaly data for network construction.
- **`max_delay`** (*int*) – The maximum delay for cross-correlation functions.
- **`gamma`** (*float*) – The width of decay region in cosine shaped window used for FFT cross-correlation estimation.

Return type tuple of two 2D arrays [index, index]

Returns the correlation strength and maximum lag matrices.

`_set_max_delay`(*max_delay*)

Set the maximum lag time used for cross-correlation estimation.

Parameters **`max_delay`** (*int*) – The maximum delay for cross-correlation functions.

`clear_cache`(*irreversible=False*)

Clean up cache.

If irreversible=True, the network cannot be recalculated using a different threshold, or link density.

Parameters **`irreversible`** (*bool*) – The irreversibility of clearing the cache.

`correlation_lag`()

Return the lag at maximum cross-correlation matrix.

Return type 2D array [index, index]

Returns the lag at maximum cross-correlation matrix.

`correlation_lag_weighted_average_path_length`()
Return correlation lag weighted average path length.

Return float the correlation lag weighted average path length.

`correlation_lag_weighted_closeness`()
Return correlation lag weighted closeness.

Return type 1D array [index]

Returns the correlation lag weighted closeness sequence.

`correlation_strength`()
Return the correlation strength matrix.

Return type 2D array [index, index]

Returns the correlation strength matrix.

`correlation_strength_weighted_average_path_length`()
Return correlation strength weighted average path length.

Return float the correlation strength weighted average path length.

`correlation_strength_weighted_closeness`()
Return correlation strength weighted closeness.

Return type 1D array [index]

Returns the correlation strength weighted closeness sequence.

`data` = None
(ClimateData) - The climate data used for network construction.

`get_max_delay`()
Return the maximum delay used for cross-correlation estimation.

Return float the maximum delay used for cross-correlation estimation.

`local_correlation_lag_weighted_vulnerability`()
Return correlation lag weighted vulnerability.

Return type 1D array [index]

Returns the correlation lag weighted vulnerability sequence.

`local_correlation_strength_weighted_vulnerability`()
Return correlation strength weighted vulnerability.

Return type 1D array [index]

Returns the correlation strength weighted vulnerability sequence.

`set_max_delay`(*max_delay*)
Set the maximum lag time used for cross-correlation estimation.

(Re)generates the current Havlin climate network accordingly.

Parameters **`max_delay`** (*int*) – The maximum delay for cross-correlation functions.

5.2.6 climate.hilbert

Provides classes for generating and analyzing complex climate networks.

class `pyunicorn.climate.hilbert.`**`HilbertClimateNetwork`**(*data*, *threshold=None*, *link_density=None*, *non_local=False*, *directed=True*, *node_weight_type='surface'*, *silence_level=0*)

Bases: `pyunicorn.climate.climate_network.ClimateNetwork`

Encapsulates a Hilbert climate network.

The associated similarity matrix is based on Hilbert coherence between time series.

Hilbert climate networks can be directed and undirected. Optional directionality is based on the average phase difference between time series.

A preliminary study of Hilbert climate networks is presented in *[Donges2009c]*.

`__init__`(*data*, *threshold=None*, *link_density=None*, *non_local=False*, *directed=True*, *node_weight_type='surface'*, *silence_level=0*)

Initialize an instance of HilbertClimateNetwork.

Note: Either threshold **OR** link_density have to be given!

Possible choices for `node_weight_type`:

- None (constant unit weights)
- "surface" (cos lat)
- "irrigation" (cos**2 lat)

Parameters

- **`data`** (`ClimateData`) – The climate data used for network construction.
- **`threshold`** (*float*) – The threshold of similarity measure, above which two nodes are linked in the network.
- **`link_density`** (*float*) – The networks's desired link density.
- **`non_local`** (*bool*) – Determines, whether links between spatially close nodes should be suppressed.
- **`directed`** (*bool*) – Determines, whether the network is constructed as directed.
- **`node_weight_type`** (*str*) – The type of geographical node weight to be used.
- **`silence_level`** (*int*) – The inverse level of verbosity of the object.

`__str__`()

Return a string representation.

`_calculate_hilbert_correlation`(*anomaly*)

Calculate Hilbert coherence and phase matrices.

Output corresponds to modulus and argument of the complex correlation coefficients between all pairs of analytic signals calculated from anomaly data, as described in *[Bergner2008]*.

Parameters **`anomaly`** (*2D Numpy array [time, index]*) – The anomaly data for network construction.

Return type tuple of two 2D Numpy matrices [index, index]

Returns the Hilbert coherence and phase matrices.

`_set_directed`(*directed*, *calculate_coherence=True*)

Switch between directed and undirected Hilbert climate network.

Parameters

- **`directed`** (*bool*) – Determines whether the network is constructed as directed.
- **`calculate_coherence`** (*bool*) – Determines whether coherence and phase are calculated from data or the directed adjacency matrix is constructed from coherence and phase information.

`clear_cache`(*irreversible=False*)

Clean up cache.

If irreversible=True, the network cannot be recalculated using a different threshold, or link density.

Parameters **`irreversible`** (*bool*) – The irreversibility of clearing the cache.

`coherence`()

Return the Hilbert coherence matrix.

Return type 2D Numpy array [index, index]

Returns the Hilbert coherence matrix.

`data` **= None**

(ClimateData) - The climate data used for network construction.

`phase_shift`()

Return the average phase shift matrix.

Return type 2D Numpy array [index, index]

Returns the average phase shift matrix.

`set_directed`(*directed*)

Switch between directed and undirected Hilbert climate network.

Also performs the complete network generation.

Parameters **`directed`** (*bool*) – Determines whether the network is constructed as directed.

5.2.7 climate.map_plots

Provides classes for analyzing spatially embedded complex networks, handling multivariate data and generating time series surrogates.

class `pyunicorn.climate.map_plots.`**`MapPlots`**(*grid*, *title*)

Bases: `object`

Encapsulates map plotting functions.

Provides functionality to easily bundle multiple geo-datasets into a single file.

`__init__`(*grid*, *title*)

Initialize an instance of MapPlots.

Plotting of maps is powered by PyNGL.

Parameters

- **`grid`** (*`Grid`*) – The Grid object describing the map data to be plotted.
- **`title`** (*str*) – The title describing the map data.

`__weakref__`

list of weak references to the object (if defined)

`add_dataset`(*title*, *data*)

Add a map data set for plotting.

Data sets are stored as dictionaries in the *`map_data`* list.

Parameters

- **`title`** (*str*) – The string describing the data set.
- **`data`** (*1D array [index]*) – The numpy array containing the map to be drawn

`add_multiple_datasets` (*map_number*, *title*, *data*)

Add a map-dataset consisting of a title and the dataset itself to the *`map_data`* list of dictionaries (pure dictionaries have no order) and reshapes data array for plotting.

INPUT: title a string describing the dataset data a numpy array containing the map to be drawn

`add_multiple_datasets_npy` (*map_number*, *title*, *data*)

Method for very large data sets (RAM issues) and useful for PARALLEL code. Data is copied to npy files (titles still in the list) that can be loaded afterwards.

INPUT: title a string describing the data set data a Numpy array containing the map to be drawn

`generate_map_plots` (*file_name*, *title_on=True*, *labels_on=True*)

Generate and save map plots.

Store the plots in the file indicated by `file_name` in the current directory.

Map plots are stored in a PDF file, with each map occupying its own page.

Parameters

- **`file_name`** (*str*) – The name for the PDF file containing map plots.
- **`title_on`** (*bool*) – Determines, whether main title is plotted.
- **`labels_on`** (*bool*) – Determines whether individual map titles are plotted.

`generate_multiple_map_plots` (*map_names*, *map_scales*, *title_on=True*, *labels_on=True*)

Generate map plots from the datasets stored in the *`map_data`* list of dictionaries. Stores the plots in the file indicated by filename in the current directory.

`generate_multiple_map_plots_npy` (*map_names*, *map_scales*, *title_on=True*, *labels_on=True*)

Method for very large datasets (RAM issues) and useful for PARALLEL code. Generates map plots from the datasets stored in the npy files and the list of titles. The data is sorted as parallel computation mixes it up. Stores the plots in the file indicated by filename in the current directory.

`grid` = **None**

(Grid) - The Grid object describing the map data to be plotted.

`map_data` = **None**

(list) - The list storing map data and titles.

`map_mult_data` = **None**

(list) - The list storing map data and titles for multiple maps.

`resources` = **None**

The PyNGL resources allow fine tuning of plotting options.

`save_ps_map` (*title*, *data*, *labels_on=True*)

Directly create a PS file of data with filename=title. Assumes normalized data between 0 and 1.

INPUT: title a string describing the dataset data a numpy array containing the map to be drawn

`title` = **None**

(string) - The title describing the map data.

5.2.8 climate.mutual_info

Provides classes for generating and analyzing complex climate networks.

class `pyunicorn.climate.mutual_info.`**`MutualInfoClimateNetwork`**(*data*, *threshold=None*, *link_density=None*, *non_local=False*, *node_weight_type='surface'*, *winter_only=True*, *silence_level=0*)

Bases: `pyunicorn.climate.climate_network.ClimateNetwork`

Represents a mutual information climate network.

Constructs a static climate network based on mutual information at zero lag, as in *[Ueoka2008]*.

Mutual information climate networks are undirected, since mutual information is a symmetrical measure. In contrast to Pearson correlation used in `TsonisClimateNetwork`, mutual information has the potential to detect nonlinear statistical interdependencies.

`__init__`(*data*, *threshold=None*, *link_density=None*, *non_local=False*, *node_weight_type='surface'*, *winter_only=True*, *silence_level=0*)

Initialize an instance of MutualInfoClimateNework.

Note: Either threshold **OR** link_density have to be given!

Possible choices for `node_weight_type`:

- None (constant unit weights)
- "surface" (cos lat)
- "irrigation" (cos**2 lat)

Parameters

- **`data`** (`ClimateData`) – The climate data used for network construction.
- **`threshold`** (*float*) – The threshold of similarity measure, above which two nodes are linked in the network.
- **`link_density`** (*float*) – The networks's desired link density.
- **`non_local`** (*bool*) – Determines, whether links between spatially close nodes should be suppressed.
- **`node_weight_type`** (*str*) – The type of geographical node weight to be used.
- **`winter_only`** (*bool*) – Determines, whether only data points from the winter months (December, January and February) should be used for analysis. Possibly, this further suppresses the annual cycle in the time series.
- **`silence_level`** (*int*) – The inverse level of verbosity of the object.

`__str__`()

Return a string representation of MutualInfoClimateNetwork.

`_calculate_mutual_information`(*anomaly*, *n_bins=32*)

Calculate the mutual information matrix at zero lag.

Note: Slow since solely based on Python and Numpy!

Parameters

- **`anomaly`** (*2D array (time, index)*) – The anomaly time series.
- **`n_bins`** (*int*) – The number of bins for estimating probability distributions.

Return type 2D array (index, index)

Returns the mutual information matrix at zero lag.

`_set_winter_only`(*winter_only*, *dump=False*)
Toggle use of exclusively winter data points for network generation.

Parameters

- **`winter_only`** (*bool*) – Indicates whether only winter months were used for network generation.
- **`dump`** (*bool*) – Store MI in data file.

`_weave_calculate_mutual_information`(*anomaly*, *n_bins=32*, *fast=True*)
Calculate the mutual information matrix at zero lag.

The weave code is adopted from the Tisean 3.0.1 mutual.c module.

Parameters

- **`anomaly`** (*2D Numpy array (time, index)*) – The anomaly time series.
- **`n_bins`** (*int*) – The number of bins for estimating probability distributions.
- **`fast`** (*bool*) – Indicates, whether fast or slow algorithm should be used.

Return type 2D array (index, index)

Returns the mutual information matrix at zero lag.

`calculate_similarity_measure`(*anomaly*)
Calculate the mutual information matrix.

Encapsulates calculation of mutual information with standard parameters.

Parameters **`anomaly`** (*2D Numpy array (time, index)*) – The anomaly time series.

Return type 2D Numpy array (index, index)

Returns the mutual information matrix at zero lag.

`data` **= None**
(ClimateData) - The climate data used for network construction.

`eval_weave_calculate_mutual_information`(*anomaly*)
Compare the fast and slow weave code to calculate mutual information.

Parameters **`anomaly`** (*2D Numpy array (time, index)*) – The anomaly time series.

Return type tuple of two 2D Numpy arrays (index, index)

Returns the mutual information matrices from fast and slow algorithm.

`local_mutual_information_weighted_vulnerability`()
Return mutual information weighted vulnerability.

Return type 1D Numpy array [index]

Returns the mutual information weighted vulnerability sequence.

`mi_file` **= None**
(string) - The name of the file for storing the mutual information matrix.

`mutual_information`(*anomaly=None*, *dump=True*)
Return mutual information matrix at zero lag.

Check if mutual information matrix (MI) was already calculated before:

- If yes, return MI from a data file.
- If not, return MI from calculation and store in file.

Parameters

- **anomaly** (*2D Numpy array (time, index)*) – The anomaly time series.
- **dump** (*bool*) – Store MI in data file.

Return type 2D Numpy array (index, index)

Returns the mutual information matrix at zero lag.

mutual_information_weighted_average_path_length()
Return mutual information weighted average path length.

Return float the mutual information weighted average path length.

mutual_information_weighted_closeness()
Return mutual information weighted closeness.

Return type 1D Numpy array [index]

Returns the mutual information weighted closeness sequence.

set_winter_only(*winter_only*, *dump=True*)
Toggle use of exclusively winter data points for network generation.

Also explicitly regenerates the instance of MutualInfoClimateNetwork.

Parameters

- **winter_only** (*bool*) – Indicates whether only winter months were used for network generation.
- **dump** (*bool*) – Store MI in data file.

winter_only()
Indicate, if only winter months were used for network generation.

Return bool whether only winter months were used for network generation.

5.2.9 climate.partial_correlation

Provides classes for generating and analyzing complex climate networks.

class pyunicorn.climate.partial_correlation.**PartialCorrelationClimateNetwork**(*data*, *thresh-old=None*, *link_density=None* *non_local=False*, *node_weight_type=* *win-ter_only=True*, *si-lence_level=0*)

Bases: *pyunicorn.climate.tsonis.TsonisClimateNetwork*

Encapsulates a partial correlation climate network.

Constructs a static climate network based on partial correlation, as in *[Ueoka2008]*.

__init__(*data*, *threshold=None*, *link_density=None*, *non_local=False*, *node_weight_type='surface'*, *winter_only=True*, *silence_level=0*)
Initialize an instance of PartialCorrelationClimateNetwork.

Note: Either threshold **OR** link_density have to be given!

Possible choices for node_weight_type:

- None (constant unit weights)

- "surface" (cos lat)
- "irrigation" (cos**2 lat)

Parameters

- **data** (*ClimateData*) – The climate data used for network construction.
- **threshold** (*float*) – The threshold of similarity measure, above which two nodes are linked in the network.
- **link_density** (*float*) – The networks's desired link density.
- **non_local** (*bool*) – Determines, whether links between spatially close nodes should be suppressed.
- **node_weight_type** (*str*) – The type of geographical node weight to be used.
- **winter_only** (*bool*) – Determines, whether only data points from the winter months (December, January and February) should be used for analysis. Possibly, this further suppresses the annual cycle in the time series.
- **silence_level** (*int*) – The inverse level of verbosity of the object.

__str__()

Return a string representation of PartialCorrelationClimateNetwork.

_calculate_correlation(*anomaly*)

Return the partial correlation matrix at zero lag.

Parameters **anomaly** (*2D Numpy array (time, index)*) – the anomaly time series from to calculate the partial correlation matrix at zero lag.

Return type 2D Numpy array (index, index)

Returns the partial correlation matrix at zero lag.

5.2.10 climate.rainfall

Provides classes for generating and analyzing complex climate networks.

class pyunicorn.climate.rainfall.**RainfallClimateNetwork**(*data*, *threshold=None*, *link_density=None*, *non_local=False*, *node_weight_type='surface'*, *event_threshold=(0, 1)*, *scale_fac=37265*, *offset=1e-07*, *silence_level=0*)

Bases: *pyunicorn.climate.climate_network.ClimateNetwork*

Encapsulate a Rainfall climate network.

The Rainfall climate network is constructed from the Spearman rank order correlation matrix (Spearman's rho) but without considering "zeros" in the dataset, which represent the time at which there is no rainfall. Spearman's rho is more robust with respect to outliers and non-gaussian data distributions than the Pearson correlation coefficient.

Rainfall climate networks are undirected due to the symmetry of the Spearman's rho matrix.

Class RainfallClimateNetwork was created by Marc Wiedermann (marcw@physik.hu-berlin.de) during an internship at PIK in March 2010.

__init__(*data*, *threshold=None*, *link_density=None*, *non_local=False*, *node_weight_type='surface'*, *event_threshold=(0, 1)*, *scale_fac=37265*, *offset=1e-07*, *silence_level=0*)

Initialize an instance of RainfallClimateNetwork.

Note: Either threshold **OR** link_density have to be given!

Possible choices for `node_weight_type`:

- None (constant unit weights)
- "surface" (cos lat)
- "irrigation" (cos**2 lat)

Parameters

- **`data`** (*`ClimateData`*) – The climate data used for network construction.
- **`threshold`** (*float*) – The threshold of similarity measure, above which two nodes are linked in the network.
- **`link_density`** (*float*) – The networks's desired link density.
- **`non_local`** (*bool*) – Determines, whether links between spatially close nodes should be suppressed.
- **`node_weight_type`** (*str*) – The type of geographical node weight to be used.
- **`event_threshold`** (*list of two numbers between 0 and 1.*) – The quantiles of the rainfall distribution at each location between which rainfall events should be considered for calculating correlations.
- **`scale_fac`** (*float*) – Scale factor for rescaling data.
- **`offset`** (*float*) – Offset for rescaling data.
- **`silence_level`** (*int*) – The inverse level of verbosity of the object.

`__str__`()

Returns a string representation of RainfallClimateNetwork.

`_calculate_correlation`(*event_threshold*, *scale_fac*, *offset*, *time_cycle*)

Returns the Spearman Rho correlation matrix.

An event_threshold can be given to extract a percentage of the given dataset, i.e. [0.9,1] extracts the ten percent of heaviest rainfall events. [0,1] selects the whole dataset.

Parameters

- **`event_threshold`** (*list of two numbers between 0 and 1.*) – The quantiles of the rainfall distribution at each location between which rainfall events should be considered for calculating correlations.
- **`scale_fac`** (*number (float)*) – Scale factor for rescaling data.
- **`offset`** (*number (float)*) – Offset for rescaling data.
- **`time_cycle`** (*number (int)*) – Length of annual cycle in given data (monthly: 12, daily: 365 etc.)

Return type 2D Numpy array (index, index)

Returns the Spearman's rho matrix at zero lag.

`calculate_corr`(*final_mask*, *anomaly*)

Return the Spearman Correlation Matrix at zero lag.

Parameters

- **`final_mask`** (*2D Numpy array (index, time)*) – A bool array with False for every value in the rainfall data, which are zero or outside the top_event interval.

- **`anomaly`** (*2D Numpy array (index, time)*) – The rainfall anomaly time series for each measuring point.

Return type 2D Numpy array (index, index)

Returns the Spearman correlation matrix.

static `calculate_rainfall`(*observable*, *scale_fac*, *offset*)
Returns the rainfall in mm on each measuring point.

Parameters

- **`observable`** (*2D Numpy array (time, index)*) – The observable time series from the data source.
- **`scale_fac`** (*number (float)*) – Scale factor for rescaling data.
- **`offset`** (*number (float)*) – Offset for rescaling data.

Return type 2D Numpy array (time, index)

Returns the rainfall for each time and location

static `calculate_top_events`(*rainfall*, *event_threshold*)
Returns a mask with boolean values. The entries are false, when the rainfall of one day is zero, or when the rainfall is not inside the event_treshold

Parameters

- **`rainfall`** (*2D Numpy array (index, time)*) – the rainfall time series for each measuring point
- **`event_threshold`** (*list of two numbers between 0 and 1.*) – The quantiles of the rainfall distribution at each location between which rainfall events should be considered for calculating correlations.

Return type 2D Numpy array (index, time)

Returns A bool array with False for every value in the rainfall data, which are zero or outside the top_event Interval.

`data` = None
(ClimateData) - The climate data used for network construction.

static `rank_time_series`(*anomaly*)
Return rank time series.

Parameters **`anomaly`** (*2D Numpy array (index, time)*) – the rainfall anomaly time series for each measuring point

Return type 2D Numpy array (index, time)

Returns The ranked time series for each gridpoint

5.2.11 climate.spearman

Provides classes for generating and analyzing complex climate networks.

class `pyunicorn.climate.spearman.`**`SpearmanClimateNetwork`**(*data*, *threshold=None*, *link_density=None*, *non_local=False*, *node_weight_type='surface'*, *winter_only=True*, *silence_level=0*)

Bases: *`pyunicorn.climate.tsonis.TsonisClimateNetwork`*

Encapsulate a Spearman climate network.

The Spearman climate network is constructed from the Spearman rank order correlation matrix (Spearman's rho). Spearman's rho is more robust with respect to ouliers and non-gaussian data distributions than the Pearson correlation coefficient used in `TsonisClimateNetwork`.

Hence, Spearman climate networks are undirected due to the symmetry of the Spearman's rho matrix.

`__init__`(*data*, *threshold=None*, *link_density=None*, *non_local=False*, *node_weight_type='surface'*, *winter_only=True*, *silence_level=0*)

Initialize an instance of `SpearmanClimateNetwork`.

Note: Either threshold **OR** link_density have to be given!

Possible choices for `node_weight_type`:

- None (constant unit weights)
- "surface" (cos lat)
- "irrigation" (cos**2 lat)

Parameters

- **`data`** (`ClimateData`) – The climate data used for network construction.
- **`threshold`** (*float*) – The threshold of similarity measure, above which two nodes are linked in the network.
- **`link_density`** (*float*) – The networks's desired link density.
- **`non_local`** (*bool*) – Determines, whether links between spatially close nodes should be suppressed.
- **`node_weight_type`** (*str*) – The type of geographical node weight to be used.
- **`winter_only`** (*bool*) – Determines, whether only data points from the winter months (December, January and February) should be used for analysis. Possibly, this further suppresses the annual cycle in the time series.
- **`silence_level`** (*int*) – The inverse level of verbosity of the object.

`__str__`()

Returns a string representation of SpearmanClimateNetwork.

`_calculate_correlation`(*anomaly*)

Return Spearman's rho matrix at zero lag.

Parameters **`anomaly`** (*2D Numpy array (time, index)*) – the anomaly time series from to calculate the correlation matrix at zero lag.

Return type 2D Numpy array (index, index)

Returns the Spearman's rho matrix at zero lag.

static `rank_time_series`(*anomaly*)

Return rank time series.

Ranks are generated individually for each time series.

Parameters **`anomaly`** (*2D Numpy array [time, index]*) – The anomaly time series to be converted into ranks.

Return type 2D Numpy array [time, index]

Returns the rank time series.

5.2.12 climate.tsonis

Provides classes for generating and analyzing complex climate networks.

class `pyunicorn.climate.tsonis.`**`TsonisClimateNetwork`**(*data*, *threshold=None*, *link_density=None*, *non_local=False*, *node_weight_type='surface'*, *winter_only=True*, *silence_level=0*)

Bases: *`pyunicorn.climate.climate_network.ClimateNetwork`*

Encapsulates a Tsonis climate network.

Construct a static climate network following Tsonis et al. from the Pearson correlation matrix at zero lag.

Hence, Tsonis climate networks are undirected due to the symmetry of the correlation matrix.

References: *[Tsonis2004]*, *[Tsonis2006]*, *[Tsonis2008b]*, *[Tsonis2008c]*.

static **`SmallTestNetwork`**()

Return a 6-node undirected test climate network from a test data set.

Example:

```
>>> r(TsonisClimateNetwork.SmallTestNetwork().adjacency)
array([[0, 0, 1, 0, 1, 0], [0, 0, 0, 1, 0, 1], [1, 0, 0, 0, 1, 0],
       [0, 1, 0, 0, 0, 1], [1, 0, 1, 0, 0, 0], [0, 1, 0, 1, 0, 0]])
```

Return type Network instance

`__init__`(*data*, *threshold=None*, *link_density=None*, *non_local=False*, *node_weight_type='surface'*, *winter_only=True*, *silence_level=0*)

Initialize an instance of TsonisClimateNetwork.

Note: Either threshold **OR** link_density have to be given!

Possible choices for `node_weight_type`:

- None (constant unit weights)
- "surface" (cos lat)
- "irrigation" (cos**2 lat)

Parameters

- **`data`** (*:classL'.ClimateData'*) – The climate data used for network construction.
- **`threshold`** (*float*) – The threshold of similarity measure, above which two nodes are linked in the network.
- **`link_density`** (*float*) – The networks's desired link density.
- **`non_local`** (*bool*) – Determines, whether links between spatially close nodes should be suppressed.
- **`ode_weight_type`** (*str*) – The type of geographical node weight to be used.
- **`winter_only`** (*bool*) – Determines, whether only data points from the winter months (December, January and February) should be used for analysis. Possibly, this further suppresses the annual cycle in the time series.
- **`silence_level`** (*int*) – The inverse level of verbosity of the object.

__str__()

Return a string representation of TsonisClimateNetwork.

Example:

```
>>> print TsonisClimateNetwork.SmallTestNetwork()
TsonisClimateNetwork:
ClimateNetwork:
GeoNetwork:
Network: undirected, 6 nodes, 6 links, link density 0.400.
Geographical boundaries:
         time     lat     lon
   min    0.0    0.00    2.50
   max    9.0   25.00   15.00
Threshold: 0.5
Local connections filtered out: False
Use only data points from winter months: False
```

_calculate_correlation(*anomaly*)

Return the correlation matrix at zero lag.

Parameters **anomaly** (*2D Numpy array (time, index)*) – the anomaly time series from which to calculate the correlation matrix at zero lag.

Return type 2D Numpy array (index, index)

Returns the correlation matrix at zero lag.

_set_winter_only(*winter_only*)

Toggle use of exclusively winter data points for network generation.

Parameters **winter_only** (*bool*) – Indicates, whether only winter months were used for network generation.

calculate_similarity_measure(*anomaly*)

Encapsulate the calculation of the correlation matrix at zero lag.

Example:

```
>>> TsonisClimateNetwork.SmallTestNetwork()._calculate_correlation(
...     anomaly=ClimateData.SmallTestData().anomaly())
array([[ 1.        , -0.25377226, -1.        ,
         0.25377226,  1.        , -0.25377226],
       [-0.25377226,  1.        ,  0.25377226,
        -1.        , -0.25377226,  1.        ],
       [-1.        ,  0.25377226,  1.        ,
        -0.25377226, -1.        ,  0.25377226],
       [ 0.25377226, -1.        , -0.25377226,
         1.        ,  0.25377226, -1.        ],
       [ 1.        , -0.25377226, -1.        ,
         0.25377226,  1.        , -0.25377226],
       [-0.25377226,  1.        ,  0.25377226,
        -1.        , -0.25377226,  1.        ]], dtype=float32)
```

Parameters **anomaly** (*2D Numpy array (time, index)*) – the anomaly time series from which to calculate the correlation matrix at zero lag.

Return type 2D Numpy array (index, index)

Returns the correlation matrix at zero lag.

correlation(**args*, ***kwargs*)

Return the correlation matrix at zero lag.

Example:

```
>>> TsonisClimateNetwork.SmallTestNetwork().correlation()
array([[ 1.        ,  0.25377226,  1.        ,
         0.25377226,  1.        ,  0.25377226],
       [ 0.25377226,  1.        ,  0.25377226,
         1.        ,  0.25377226,  1.        ],
       [ 1.        ,  0.25377226,  1.        ,
         0.25377226,  1.        ,  0.25377226],
       [ 0.25377226,  1.        ,  0.25377226,
         1.        ,  0.25377226,  1.        ],
       [ 1.        ,  0.25377226,  1.        ,
         0.25377226,  1.        ,  0.25377226],
       [ 0.25377226,  1.        ,  0.25377226,
         1.        ,  0.25377226,  1.        ]], dtype=float32)
```

Return type 2D Numpy array (index, index)

Returns the correlation matrix at zero lag.

`correlation_weighted_average_path_length`()

Return correlation weighted average path length.

Example:

```
>>> TsonisClimateNetwork.SmallTestNetwork().                correlation_weighted_average_
1.0
```

Return float the correlation weighted average path length.

`correlation_weighted_closeness`()

Return correlation weighted closeness.

Example:

```
>>> TsonisClimateNetwork.SmallTestNetwork().                correlation_weighted_closenes
array([ 0.25,  0.25,  0.25,  0.25,  0.25,  0.25])
```

Return type 1D Numpy array [index]

Returns the correlation weighted closeness sequence.

`data` = **None**

(ClimateData) - The climate data used for network construction.

`local_correlation_weighted_vulnerability`()

Return correlation weighted vulnerability.

Example:

```
>>> TsonisClimateNetwork.SmallTestNetwork().                local_correlation_weighted_vu
array([ 0.,  0.,  0.,  0.,  0.,  0.])
```

Return type 1D Numpy array [index]

Returns the correlation weighted vulnerability sequence.

`set_winter_only`(*winter_only*)

Toggle use of exclusively winter data points for network generation.

Also explicitly re(generates) the instance of TsonisClimateNetwork.

Example:

```
>>> net = TsonisClimateNetwork.SmallTestNetwork()
>>> net.set_winter_only(winter_only=False)
>>> net.n_links
6
```

Parameters **winter_only** (*bool*) – Indicates, whether only winter months were used for network generation.

winter_only()

Indicate, if only winter months were used for network generation.

Example:

```
>>> TsonisClimateNetwork.SmallTestNetwork().winter_only()
False
```

Return bool whether only winter months were used for network generation.

5.3 timeseries

Recurrence plots, recurrence networks, multivariate extensions and visibility graph analysis of time series. Time series surrogates for significance testing.

5.3.1 timeseries.cross_recurrence_plot

Provides classes for the analysis of dynamical systems and time series based on recurrence plots, including measures of recurrence quantification analysis (RQA) and recurrence network analysis.

class pyunicorn.timeseries.cross_recurrence_plot.**CrossRecurrencePlot**(*x*, *y*, *metric='supremum'*, *normalize=False*, *sparse_rqa=False*, *silence_level=0*, ***kwds*)

Bases: *pyunicorn.timeseries.recurrence_plot.RecurrencePlot*

Class CrossRecurrencePlot for generating and quantitatively analyzing cross recurrence plots.

The CrossRecurrencePlot class supports the construction of cross recurrence plots from two multi-dimensional time series, optionally using embedding. Currently, manhattan, euclidean and supremum norms are provided for measuring distances in phase space.

Methods for calculating commonly used measures of recurrence quantification analysis (RQA) are provided, e.g., determinism, maximum diagonal line length and laminarity. The definitions of these measures together with a review of the theory and applications of cross recurrence plots are given in *[Marwan2007]*.

Examples:

- Create an instance of CrossRecurrencePlot from time series x and y with a fixed recurrence threshold and without embedding:

```
CrossRecurrencePlot(x, y, threshold=0.1)
```

- Create an instance of CrossRecurrencePlot at a fixed recurrence rate and using time delay embedding:

```
    CrossRecurrencePlot(x, y, dim=3, tau=2,
                        recurrence_rate=0.05).recurrence_rate()
```

CR = None
The cross recurrence matrix.

M = None
The length of the embedded time series y.

N = None
The length of the embedded time series x.

`__init__`(*x*, *y*, *metric='supremum'*, *normalize=False*, *sparse_rqa=False*, *silence_level=0*, ***kwds*)
Initialize an instance of CrossRecurrencePlot.

Note: For a cross recurrence plot, time series x and y generally do **not** need to have the same length!

Either recurrence threshold `threshold` or recurrence rate `recurrence_rate` have to be given as keyword arguments.

Embedding is only supported for scalar time series. If embedding dimension `dim` and delay `tau` are **both** given as keyword arguments, embedding is applied. Multidimensional time series are processed as is by default. The same delay embedding is applied to both time series x and y.

Parameters

- **x** (*2D array (time, dimension)*) – One of the time series to be analyzed, can be scalar or multi-dimensional.
- **y** (*2D array (time, dimension)*) – One of the time series to be analyzed, can be scalar or multi-dimensional.
- **metric** (*str*) – The metric for measuring distances in phase space ("manhattan", "euclidean", "supremum").
- **normalize** (*bool*) – Decide whether to normalize both time series to zero mean and unit standard deviation.
- **silence_level** (*number*) – Inverse level of verbosity of the object.
- **threshold** (*number*) – The recurrence threshold keyword for generating the cross recurrence plot using a fixed threshold.
- **recurrence_rate** (*number*) – The recurrence rate keyword for generating the cross recurrence plot using a fixed recurrence rate.
- **dim** (*number*) – The embedding dimension.
- **tau** (*number*) – The embedding delay.

`__str__`()
Returns a string representation.

balance()
Return balance of the cross recurrence plot.

Might be useful for detecting the direction of coupling between systems using cross recurrence analysis.

cross_recurrence_rate()
Return cross recurrence rate.

Return type number (float)

Returns the cross recurrence rate.

distance_matrix(*x_embedded*, *y_embedded*, *metric*)
Return phase space cross distance matrix D according to the chosen metric.

Parameters

- **x_embedded** (*2D array (time, embedding dimension)*) – The phase space trajectory x.
- **y_embedded** (*2D array (time, embedding dimension)*) – The phase space trajectory y.
- **metric** (*str*) – The metric for measuring distances in phase space ("manhattan", "euclidean", "supremum").

Return type 2D square array

Returns the phase space cross distance matrix D

euclidean_distance_matrix(*x_embedded*, *y_embedded*)

Return the euclidean distance matrix from two (embedded) time series.

Parameters

- **x_embedded** (*2D Numpy array (time, embedding dimension)*) – The phase space trajectory x.
- **y_embedded** (*2D Numpy array (time, embedding dimension)*) – The phase space trajectory y.

Return type 2D rectangular Numpy array ("float32")

Returns the euclidean distance matrix.

manhattan_distance_matrix(*x_embedded*, *y_embedded*)

Return the manhattan distance matrix from two (embedded) time series.

Parameters

- **x_embedded** (*2D Numpy array (time, embedding dimension)*) – The phase space trajectory x.
- **y_embedded** (*2D Numpy array (time, embedding dimension)*) – The phase space trajectory y.

Return type 2D rectangular Numpy array ("float32")

Returns the manhattan distance matrix.

recurrence_matrix()

Return the current cross recurrence matrix CR.

Return type 2D square Numpy array

Returns the current cross recurrence matrix CR.

recurrence_rate()

Return cross recurrence rate.

Alias to *cross_recurrence_rate()*, since `RecurrencePlot.recurrence_rate()` would give incorrect results here.

Return type number (float)

Returns the cross recurrence rate.

set_fixed_recurrence_rate(*recurrence_rate*)

Set the cross recurrence plot to a fixed recurrence rate.

Modifies / sets the class variables *CR*, *N* and *M* accordingly.

Parameters **recurrence_rate** (*number*) – The recurrence rate.

set_fixed_threshold(*threshold*)

Set the cross recurrence plot to a fixed threshold.

Modifies / sets the class variables *CR*, *N* and *M* accordingly.

Parameters **`threshold`** (*number*) – The recurrence threshold.

`supremum_distance_matrix` (*x_embedded*, *y_embedded*)

Return the supremum distance matrix from two (embedded) time series.

Parameters

- **`x_embedded`** (*2D Numpy array (time, embedding dimension)*) – The phase space trajectory x.
- **`y_embedded`** (*2D Numpy array (time, embedding dimension)*) – The phase space trajectory y.

Return type 2D rectangular Numpy array ("float32")

Returns the supremum distance matrix.

`x` **= None**

The time series x.

`x_embedded` **= None**

The embedded time series x.

`y` **= None**

The time series y.

`y_embedded` **= None**

The embedded time series y.

5.3.2 timeseries.inter_system_recurrence_network

Provides classes for the analysis of dynamical systems and time series based on recurrence plots, including measures of recurrence quantification analysis (RQA) and recurrence network analysis.

class `pyunicorn.timeseries.inter_system_recurrence_network.`**`InterSystemRecurrenceNetwork`**(*x*, *y*, *me* *ric* *no* *ma* *ize* *si-* *len* ****

Bases: *`pyunicorn.core.interacting_networks.InteractingNetworks`*

Generating and quantitatively analyzing inter-system recurrence networks.

For a inter-system recurrence network, time series x and y do not need to have the same length! Formally, nodes are identified with state vectors in the common phase space of both time series. Hence, the time series need to have the same number of dimensions and identical physical units. Undirected links are added to describe recurrences within x and y as well as cross-recurrences between x and y. Self-loops are excluded in this undirected network representation.

More information on the theory and applications of inter system recurrence networks can be found in *[Feldhoff2012]*.

Examples:

•Create an instance of InterSystemRecurrenceNetwork with fixed recurrence thresholds and without embedding:

```
InterSystemRecurrenceNetwork(x, y, threshold=(0.1, 0.2, 0.1))
```

•Create an instance of InterSystemRecurrenceNetwork at a fixed recurrence rate and using time delay embedding:

```
    InterSystemRecurrenceNetwork(
        x, y, dim=3, tau=(2, 1), recurrence_rate=(0.05, 0.05, 0.02))
```

N = None
Total number of nodes of ISRN.

N_x = None
Number of nodes in subnetwork x.

N_y = None
Number of nodes in subnetwork y.

__init__(*x*, *y*, *metric='supremum'*, *normalize=False*, *silence_level=0*, ***kwds*)
Initialize an instance of InterSystemRecurrenceNetwork (ISRN).

Note: For an inter system recurrence network, time series x and y need to have the same number of dimensions!

Creates an embedding of the given time series x and y, calculates a inter system recurrence matrix from the embedding and then creates an InteractingNetwork object from this matrix, interpreting the inter system recurrence matrix as the adjacency matrix of an undirected complex network.

Either recurrence thresholds `threshold` or recurrence rates `recurrence_rate` have to be given as keyword arguments.

Embedding is only supported for scalar time series. If embedding dimension `dim` and delay `tau` are **both** given as keyword arguments, embedding is applied. Multidimensional time series are processed as is by default.

Parameters

- **x** (*2D Numpy array (time, dimension)*) – The time series x to be analyzed, can be scalar or multi-dimensional.
- **y** (*2D Numpy array (time, dimension)*) – The time series y to be analyzed, can be scalar or multi-dimensional.
- **metric** (*tuple of string*) – The metric for measuring distances in phase space ("manhattan", "euclidean", "supremum").
- **normalize** (*bool*) – Decide whether to normalize the time series to zero mean and unit standard deviation.
- **silence_level** (*int*) – The inverse level of verbosity of the object.
- **kwds** – Additional options.
- **threshold** (*tuple of number (three numbers)*) – The recurrence threshold keyword for generating the recurrence plot using fixed thresholds. Give for each time series and the cross recurrence plot separately.
- **recurrence_rate** (*tuple of number (three numbers)*) – The recurrence rate keyword for generating the recurrence plot using a fixed recurrence rate. Give separately for each time series.
- **dim** (*int*) – The embedding dimension. Must be the same for both time series.
- **tau** (*tuple of int*) – The embedding delay. Give separately for each time series.

__str__()
Returns a string representation.

clear_cache()
Clean up memory by deleting information that can be recalculated from basic data.

Extends the clean up methods of the parent classes.

cross_global_clustering_xy()
Return cross global clustering of x with respect to y.

See *[Feldhoff2012]* for definition, further explanation and applications.

Return type number (float)

Returns the cross global clustering of x with respect to y.

cross_global_clustering_yx()
Return cross global clustering of y with respect to x.

See *[Feldhoff2012]* for definition, further explanation and applications.

Return type number (float)

Returns the cross global clustering of y with respect to x.

cross_recurrence_rate()
Return cross recurrence rate between subnetworks x and y.

Return type number (float)

Returns the cross recurrence rate between subnetworks x and y.

cross_transitivity_xy()
Return cross transitivity of x with respect to y.

See *[Feldhoff2012]* for definition, further explanation and applications.

Return type number (float)

Returns the cross transitivity of x with respect to y.

cross_transitivity_yx()
Return cross transitivity of y with respect to x.

See *[Feldhoff2012]* for definition, further explanation and applications.

Return type number (float)

Returns the cross transitivity of y with respect to x.

inter_system_recurrence_matrix()
Return the current inter system recurrence matrix $ISRM$.

Return type 2D square Numpy array

Returns the current inter system recurrence matrix $ISRM$.

internal_recurrence_rates()
Return internal recurrence rates of subnetworks x and y.

Return type tuple of number (float)

Returns the internal recurrence rates of subnetworks x and y.

metric = None
The metric used for measuring distances in phase space.

set_fixed_recurrence_rate(*density*)
Create a inter system recurrence network at fixed link densities (recurrence rates).

Parameters **density** (*tuple of number (three numbers)*) – The three recurrence rate parameters. Give for each time series and the cross recurrence plot separately.

set_fixed_threshold(*threshold*)
Create a inter system recurrence network at fixed thresholds.

Parameters **threshold** (*tuple of number (three numbers)*) – The three threshold parameters. Give for each time series and the cross recurrence plot separately.

`silence_level` = None
The inverse level of verbosity of the object.

`x` = None
The time series x.

`x_embedded` = None
The embedded time series x.

`y` = None
The time series y.

`y_embedded` = None
The embedded time series y.

5.3.3 timeseries.joint_recurrence_network

Provides classes for the analysis of dynamical systems and time series based on recurrence plots, including measures of recurrence quantification analysis (RQA) and recurrence network analysis.

class `pyunicorn.timeseries.joint_recurrence_network.`**`JointRecurrenceNetwork`**(*x, y, metric=('supremum', 'supremum'), normalize=False, lag=0, silence_level=0, **kwds*)

Bases: *`pyunicorn.timeseries.joint_recurrence_plot.JointRecurrencePlot`*, *`pyunicorn.core.network.Network`*

Class JointRecurrenceNetwork for generating and quantitatively analyzing joint recurrence networks.

For a joint recurrence network, time series x and y need to have the same length! Formally, nodes are identified with sampling points in time, while an undirected link (i,j) is introduced if x at time i is recurrent to x at time j and also y at time i is recurrent to y at time j. Self-loops are excluded in this undirected network representation.

More information on the theory and applications of joint recurrence networks can be found in *[Feldhoff2013]*.

Examples:

- Create an instance of JointRecurrenceNetwork with a fixed recurrence threshold and without embedding:

```
JointRecurrenceNetwork(x, y, threshold=(0.1,0.2))
```

- Create an instance of JointRecurrenceNetwork with a fixed recurrence threshold in units of STD and without embedding:

```
JointRecurrenceNetwork(x, y, threshold_std=(0.03,0.05))
```

- Create an instance of JointRecurrenceNetwork at a fixed recurrence rate and using time delay embedding:

```
JointRecurrenceNetwork(
    x, y, dim=(3,5), tau=(2,1),
    recurrence_rate=(0.05,0.04)).recurrence_rate()
```

`__init__`(*x*, *y*, *metric=('supremum', 'supremum')*, *normalize=False*, *lag=0*, *silence_level=0*, ***kwds*)

Initialize an instance of JointRecurrenceNetwork.

Note: For a joint recurrence network, time series x and y need to have the same length!

Creates an embedding of the given time series x and y, calculates a joint recurrence plot from the embedding and then creates a Network object from the joint recurrence plot, interpreting the joint recurrence matrix as the adjacency matrix of an undirected complex network.

Either recurrence thresholds `threshold`/`threshold_std` or recurrence rates `recurrence_rate` have to be given as keyword arguments.

Embedding is only supported for scalar time series. If embedding dimension `dim` and delay `tau` are **both** given as keyword arguments, embedding is applied. Multidimensional time series are processed as is by default.

Parameters

- **`x`** (*2D Numpy array (time, dimension)*) – The time series x to be analyzed, can be scalar or multi-dimensional.
- **`y`** (*2D Numpy array (time, dimension)*) – The time series y to be analyzed, can be scalar or multi-dimensional.
- **`metric`** (*tuple of string*) – The metric for measuring distances in phase space ("manhattan", "euclidean", "supremum"). Give separately for each time series.
- **`normalize`** (*tuple of bool*) – Decide whether to normalize the time series to zero mean and unit standard deviation. Give separately for each time series.
- **`lag`** (*number*) – To create a delayed version of the JRP.
- **`silence_level`** (*number*) – Inverse level of verbosity of the object.
- **`threshold`** (*tuple of number*) – The recurrence threshold keyword for generating the recurrence plot using a fixed threshold. Give separately for each time series.
- **`threshold_std`** (*tuple of number*) – The recurrence threshold keyword for generating the recurrence plot using a fixed threshold in units of the time series' STD. Give separately for each time series.
- **`recurrence_rate`** (*tuple of number*) – The recurrence rate keyword for generating the recurrence plot using a fixed recurrence rate. Give separately for each time series.
- **`dim`** (*tuple of number*) – The embedding dimension. Give separately for each time series.
- **`tau`** (*tuple of number*) – The embedding delay. Give separately for each time series.

`__str__`()

Returns a string representation.

`clear_cache`()

Clean up memory by deleting information that can be recalculated from basic data.

Extends the clean up methods of the parent classes.

`set_fixed_recurrence_rate`(*density*)

Create a joint recurrence network at fixed link densities (recurrence rates).

Parameters **`density`** (*tuple of number*) – The link density / recurrence rate. Give for each time series separately.

`set_fixed_threshold`(*threshold*)

Create a joint recurrence network at fixed thresholds.

Parameters **`threshold`** (*tuple of number*) – The threshold. Give for each time series separately.

set_fixed_threshold_std(*threshold_std*)

Create a joint recurrence network at fixed thresholds in units of the standard deviation of the time series.

Parameters **threshold_std** (*tuple of number*) – The threshold in units of standard deviation. Give for each time series separately.

5.3.4 timeseries.joint_recurrence_plot

Provides classes for the analysis of dynamical systems and time series based on recurrence plots, including measures of recurrence quantification analysis (RQA) and recurrence network analysis.

class pyunicorn.timeseries.joint_recurrence_plot.**JointRecurrencePlot**(*x*, *y*, *metric=('supremum', 'supremum')*, *normalize=False*, *lag=0*, *silence_level=0*, ***kwds*)

Bases: *pyunicorn.timeseries.recurrence_plot.RecurrencePlot*

Class JointRecurrencePlot for generating and quantitatively analyzing joint recurrence plots.

The JointRecurrencePlot class supports the construction of joint recurrence plots from two multi-dimensional time series, optionally using embedding. Currently, manhattan, euclidean and supremum norms are provided for measuring distances in phase space.

Methods for calculating commonly used measures of recurrence quantification analysis (RQA) are provided, e.g., determinism, maximum diagonal line length and laminarity. The definitions of these measures together with a review of the theory and applications of joint recurrence plots are given in *[Marwan2007]*.

Examples:

- Create an instance of JointRecurrencePlot with a fixed recurrence threshold and without embedding:

```
JointRecurrencePlot(x, y, threshold=(0.1,0.2))
```

- Create an instance of JointRecurrencePlot with a fixed recurrence threshold in units of STD and without embedding:

```
JointRecurrencePlot(x, y, threshold_std=(0.03,0.05))
```

- Create an instance of JointRecurrencePlot at a fixed recurrence rate and using time delay embedding:

```
JointRecurrencePlot(
    x, y, dim=(3,5), tau=(2,1),
    recurrence_rate=(0.05,0.04)).recurrence_rate()
```

JR = None

The joint recurrence matrix.

N = None

The length of both embedded time series x and y.

__init__(*x*, *y*, *metric=('supremum', 'supremum')*, *normalize=False*, *lag=0*, *silence_level=0*, ***kwds*)

Initialize an instance of JointRecurrencePlot.

Note: For a joint recurrence plot, time series x and y need to have the same length!

Either recurrence thresholds `threshold`/`threshold_std` or recurrence rates `recurrence_rate` have to be given as keyword arguments.

Embedding is only supported for scalar time series. If embedding dimension `dim` and delay `tau` are **both** given as keyword arguments, embedding is applied. Multidimensional time series are processed as is by default.

Parameters

- **x** (*2D array (time, dimension)*) – Time series x to be analyzed (scalar/multi-dimensional).
- **y** (*2D array (time, dimension)*) – Time series y to be analyzed (scalar/multi-dimensional).
- **metric** (*tuple of string*) – The metric for measuring distances in phase space ("manhattan", "euclidean", "supremum"). Give separately for each time series.
- **normalize** (*tuple of bool*) – Decide whether to normalize the time series to zero mean and unit standard deviation. Give separately for each time series.
- **lag** (*number*) – To create a delayed version of the JRP.
- **silence_level** (*number*) – Inverse level of verbosity of the object.
- **threshold** (*tuple of number*) – The recurrence threshold keyword for generating the recurrence plot using a fixed threshold. Give separately for each time series.
- **threshold_std** (*tuple of number*) – The recurrence threshold keyword for generating the recurrence plot using a fixed threshold in units of the time series' STD. Give separately for each time series.
- **recurrence_rate** (*tuple of number*) – The recurrence rate keyword for generating the recurrence plot using a fixed recurrence rate. Give separately for each time series.
- **dim** (*tuple of number*) – Embedding dimension. Give separately for each time series.
- **tau** (*tuple of number*) – Embedding delay. Give separately for each time series.

__str__()

Returns a string representation.

lag = **None**

Used to create a delayed JRP.

recurrence_matrix()

Return the current joint recurrence matrix JR.

Return type 2D square Numpy array

Returns the current joint recurrence matrix JR.

set_fixed_recurrence_rate(*recurrence_rate*)

Set the joint recurrence plot to fixed recurrence rates.

Modifies / sets the class variables *JR* and *N* accordingly.

Parameters **recurrence_rate** (*tuple of number*) – The recurrence rate. Give for both time series separately.

set_fixed_threshold(*threshold*)

Set the joint recurrence plot to fixed thresholds.

Modifies / sets the class variables *JR* and *N* accordingly.

Parameters **threshold** (*tuple of number*) – The recurrence threshold. Give for both time series separately.

set_fixed_threshold_std(*threshold_std*)

Set the joint recurrence plot to fixed thresholds in units of the standard deviation of the time series.

Calculates the absolute thresholds and calls *set_fixed_threshold()*.

Parameters **threshold_std** (*tuple of number*) – The recurrence threshold in units of the standard deviation of the time series. Give for both time series separately.

x = None

The time series x.

x_embedded = None

The embedded time series x.

y = None

The time series y.

y_embedded = None

The embedded time series y.

5.3.5 timeseries.recurrence_network

Provides classes for the analysis of dynamical systems and time series based on recurrence plots, including measures of recurrence quantification analysis (RQA) and recurrence network analysis.

class pyunicorn.timeseries.recurrence_network.**RecurrenceNetwork**(*time_series, metric='supremum', normalize=False, missing_values=False, silence_level=0, **kwds*)

Bases: *pyunicorn.timeseries.recurrence_plot.RecurrencePlot*, *pyunicorn.core.network.Network*

Class RecurrenceNetwork for generating and quantitatively analyzing recurrence networks.

More information on the theory and applications of recurrence networks can be found in *[Marwan2009]*, *[Donner2010b]*.

Examples:

- Create an instance of RecurrenceNetwork with a fixed recurrence threshold and without embedding:

```
RecurrenceNetwork(time_series, threshold=0.1)
```

- Create an instance of RecurrenceNetwork at a fixed (global) recurrence rate and using time delay embedding:

```
RecurrenceNetwork(time_series, dim=3, tau=2,
                  recurrence_rate=0.05).recurrence_rate()
```

__init__(*time_series, metric='supremum', normalize=False, missing_values=False, silence_level=0, **kwds*)

Initialize an instance of RecurrenceNetwork.

Creates an embedding of the given time series, calculates a recurrence plot from the embedding and then creates a Network object from the recurrence plot, interpreting the recurrence matrix as the adjacency matrix of a complex network.

Either recurrence threshold `threshold`/`threshold_std`, recurrence rate `recurrence_rate` or local recurrence rate `local_recurrence_rate` have to be given as keyword arguments.

Embedding is only supported for scalar time series. If embedding dimension `dim` and delay `tau` are **both** given as keyword arguments, embedding is applied. Multidimensional time series are processed as is by default.

Parameters

- **`time_series`** (*2D array (time, dimension)*) – The time series to be analyzed, can be scalar or multi-dimensional.
- **`metric`** (*str*) – The metric for measuring distances in phase space ("manhattan", "euclidean", "supremum").
- **`normalize`** (*bool*) – Decide whether to normalize the time series to zero mean and unit standard deviation.
- **`missing_values`** (*bool*) – Toggle special treatment of missing values in *`RecurrencePlot.time_series`*.
- **`silence_level`** (*number*) – Inverse level of verbosity of the object.
- **`threshold`** (*number*) – The recurrence threshold keyword for generating the recurrence network using a fixed threshold.
- **`threshold_std`** (*number*) – The recurrence threshold keyword for generating the recurrence plot using a fixed threshold in units of the time series' STD.
- **`recurrence_rate`** (*number*) – The recurrence rate keyword for generating the recurrence network using a fixed recurrence rate.
- **`local_recurrence_rate`** (*number*) – The local recurrence rate keyword for generating the recurrence plot using a fixed local recurrence rate (same number of recurrences for each state vector).
- **`adaptive_neighborhood_size`** (*number*) – The adaptive neighborhood size parameter for generating recurrence plots based on the algorithm in *[Xu2008]*.
- **`dim`** (*number*) – The embedding dimension.
- **`tau`** (*number*) – The embedding delay.
- **`node_weights`** (*1D array (time)*) – The sequence of weights associated with each node for calculating n.s.i. network measures.

`__str__`()
Returns a string representation.

`clear_cache`()
Clean up memory by deleting information that can be recalculated from basic data.

Extends the clean up methods of the parent classes.

`local_clustering_dim_single_scale`()
Return local clustering dimension for a single scale.

The single scale local clustering dimension can be interpreted as a local measureof the dimensionality of the set of points underlying the recurrence network (*[Donner2011b]*.). The scale is determined by the chosen recurrence threshold. Note that the maxima and minima of the single scale local clustering dimension when varying the scale give a more meaningful measure of dimensionality as is explained in *[Donner2011b]*.

Attention: currently only works correctly for supremum norm.

Return type 1d numpy array [node] of float

Returns the single scale transitivity dimension.

`set_adaptive_neighborhood_size`(*adaptive_neighborhood_size*, *order=None*)
Create a recurrence network using the adaptive neighborhood size algorithm used in *[Xu2008]*.

The exact algorithm was deduced from private correspondence with the authors. It leads to an undirected network with mean degree $< k >= 2 * m$, where m is the adaptive_neighborhood_size. The degree k_v of single nodes may vary, but $k_v >= m$ holds!

Parameters

- **`adaptive_neighborhood_size`** (*number*) – The number of adaptive nearest neighbors (recurrences) assigned to each state vector.
- **`order`** (*1D array (int32)*) – The indices of state vectors in the order desired for processing by the algorithm.

`set_fixed_local_recurrence_rate`(*local_recurrence_rate*)

Create a recurrence network at a fixed local recurrence rate.

This leads to a directed recurrence network with identical out-degree $int(N * local_recurrence_rate)$, but variable in-degree. The associated recurrence plot coincides with the original Eckmann definition.

Parameters **`local_recurrence_rate`** (*number*) – The local recurrence rate.

`set_fixed_recurrence_rate`(*recurrence_rate*)

Create a recurrence network at a fixed link density (recurrence rate).

Parameters **`recurrence_rate`** (*number*) – The link density / recurrence rate.

`set_fixed_threshold`(*threshold*)

Create a recurrence network at a fixed threshold.

Parameters **`threshold`** (*number*) – The threshold.

`set_fixed_threshold_std`(*threshold_std*)

Set the recurrence network to a fixed threshold in units of the standard deviation of the time series.

Parameters **`threshold_std`** (*number*) – The recurrence threshold in units of the standard deviation of the time series.

`transitivity_dim_single_scale`()

Return transitivity dimension for a single scale.

The single scale transitivity dimension can be interpreted as a global measure of the dimensionality of the set of points underlying the recurrence network (*[Donner2011b]*.). The scale is determined by the chosen recurrence threshold. Note that the maxima and minima of the single scale transitivity dimension when varying the scale give a more meaningful measure of dimensionality as is explained in *[Donner2011b]*.

Attention: currently only works correctly for supremum norm.

Return type float

Returns the single scale transitivity dimension.

5.3.6 timeseries.recurrence_plot

Provides classes for the analysis of dynamical systems and time series based on recurrence plots, including measures of recurrence quantification analysis (RQA) and recurrence network analysis.

class `pyunicorn.timeseries.recurrence_plot.`**`RecurrencePlot`**(*time_series*, *metric='supremum'*, *normalize=False*, *missing_values=False*, *sparse_rqa=False*, *silence_level=0*, ***kwds*)

Bases: `object`

Class RecurrencePlot for generating and quantitatively analyzing recurrence plots.

The RecurrencePlot class supports the construction of recurrence plots from multi-dimensional time series, optionally using embedding. Currently, manhattan, euclidean and supremum norms are provided for measuring distances in phase space.

Methods for calculating commonly used measures of recurrence quantification analysis (RQA) are provided, e.g., determinism, maximum diagonal line length and laminarity. The definitions of these measures together with a review of the theory and applications of recurrence plots are given in *[Marwan2007]*.

Examples:

- Create an instance of RecurrencePlot with a fixed recurrence threshold and without embedding:

```
RecurrencePlot(time_series, threshold=0.1)
```

- Create an instance of RecurrencePlot with a fixed recurrence threshold in units of STD and without embedding:

```
RecurrencePlot(time_series, threshold_std=0.03)
```

- Create an instance of RecurrencePlot at a fixed (global) recurrence rate and using time delay embedding:

```
RecurrencePlot(time_series, dim=3, tau=2,
               recurrence_rate=0.05).recurrence_rate()
```

`N` = None
The number of state vectors (number of lines and rows) of the RP.

`R` = None
The recurrence matrix.

`__init__`(*time_series*, *metric='supremum'*, *normalize=False*, *missing_values=False*, *sparse_rqa=False*, *silence_level=0*, ***kwds*)
Initialize an instance of RecurrencePlot.

Either recurrence threshold `threshold`/`threshold_std`, recurrence rate `recurrence_rate` or local recurrence rate `local_recurrence_rate` have to be given as keyword arguments.

Embedding is only supported for scalar time series. If embedding dimension `dim` and delay `tau` are **both** given as keyword arguments, embedding is applied. Multidimensional time series are processed as is by default.

Attention The sparse_rqa feature is experimental and currently only works for fixed threshold and the supremum metric.

Parameters

- **`time_series`** (*2D array (time, dimension)*) – The time series to be analyzed, can be scalar or multi-dimensional.
- **`metric`** (*str*) – The metric for measuring distances in phase space ("manhattan", "euclidean", "supremum").
- **`normalize`** (*bool*) – Decide whether to normalize the time series to zero mean and unit standard deviation.
- **`missing_values`** (*bool*) – Toggle special treatment of missing values in *`RecurrencePlot.time_series`*.
- **`sparse_rqa`** (*bool*) – Toggles sequential RQA computation using less memory for use with long time series.
- **`silence_level`** (*int*) – Inverse level of verbosity of the object.
- **`threshold`** (*number*) – The recurrence threshold keyword for generating the recurrence plot using a fixed threshold.
- **`threshold_std`** (*number*) – The recurrence threshold keyword for generating the recurrence plot using a fixed threshold in units of the time series' STD.

- **recurrence_rate** (*number*) – The recurrence rate keyword for generating the recurrence plot using a fixed recurrence rate.
- **local_recurrence_rate** (*number*) – The local recurrence rate keyword for generating the recurrence plot using a fixed local recurrence rate (same number of recurrences for each state vector).
- **adaptive_neighborhood_size** (*number*) – The adaptive neighborhood size parameter for generating recurrence plots based on the algorithm in *[Xu2008]*.
- **dim** (*number*) – The embedding dimension.
- **tau** (*number*) – The embedding delay.

__str__()
Returns a string representation.

__weakref__
list of weak references to the object (if defined)

average_diaglength(*l_min=2*, *resampled_dist=None*)
Return diagonal line-based RQA measure average diagonal line length L.

L is defined as the average length of diagonal lines (of at least length l_min).

Parameters

- **l_min** (*number*) – The minimum diagonal line length.
- **resampled_dist** (*1D array (integer)*) – resampled frequency distribution of diagonal lines

Return number the average diagonal line length L.

average_vertlength(*v_min=2*, *resampled_dist=None*)
Return vertical line-based RQA measure average vertical line length TT.

TT is defined as the average vertical line length (of at least length v_min) and is also called trapping time TT.

Parameters

- **v_min** (*number*) – The minimal vertical line length.
- **resampled_dist** (*1D array (integer)*) – resampled frequency distribution of vertical lines

Return number the trapping time TT.

average_white_vertlength(*w_min=1*)
Return white vertical line-based RQA measure average white vertical line length.

It is defined as the average white vertical line length (of at least length w_min) and is also called mean recurrence time.

Reference: *[Ngamga2007]*.

Parameters **w_min** (*number*) – The minimal white vertical line length.

Return number the mean recurrence time.

static **bootstrap_distance_matrix**(*embedding*, *metric*, *M*)
Return bootstrap samples from distance matrix.

Parameters

- **embedding** (*2D array (time, embedding dimension)*) – The phase space trajectory.
- **metric** (*str*) – The metric for measuring distances in phase space (“manhattan”, “euclidean”, “supremum”).
- **M** (*int*) – Number of bootstrap samples

Return type 1D array ("float32")

Returns the bootstrap samples from distance matrix.

clear_cache(*irreversible=False*)
Clean up memory.

determinism(*l_min=2*, *resampled_dist=None*)
Return diagonal line-based RQA measure determinism DET.

DET is defined as the ratio of recurrence points that form diagonal structures (of at least length l_min) to all recurrence points.

Parameters

- **l_min** (*number*) – The minimum diagonal line length.
- **resampled_dist** (*1D array (integer)*) – resampled frequency distribution of diagonal lines

Return number the determinism DET.

diag_entropy(*l_min=2*, *resampled_dist=None*)
Return diagonal line-based RQA measure diagonal line entropy $ENTR$.

$ENTR$ is defined as the entropy of the probability to find a diagonal line of exactly length l in the RP - reflects the complexity of the RP with respect to diagonal lines.

Parameters

- **l_min** (*number*) – The minimal diagonal line length.
- **resampled_dist** (*1D array (integer)*) – resampled frequency distribution of diagonal lines

Return number the diagonal line-based entropy $ENTR$.

diagline_dist()
Return the frequency distribution of diagonal line lengths $P(l)$.

The l th entry of $P(l)$ contains the number of diagonal lines of length l.

Note: Experimental handling of missing values. Diagonal lines touching lines and blocks of missing entries in the recurrence matrix are not counted.

Return type 1D array (int32)

Returns the frequency distribution of diagonal line lengths $P(l)$.

distance_matrix(*embedding*, *metric*)
Return phase space distance matrix D according to the chosen metric.

Parameters

- **embedding** (*2D array (time, embedding dimension)*) – The phase space trajectory.
- **metric** (*str*) – The metric for measuring distances in phase space ("manhattan", "euclidean", "supremum").

Return type 2D square array

Returns the phase space distance matrix D

static **embed_time_series**(*time_series*, *dim*, *tau*)
Return a time series' delay embedding.

Returns a Numpy array containing a delay embedding of the time series using embedding dimension dim and time delay tau.

Parameters

- **`time_series`** (*1D array*) – The scalar time series to be embedded.
- **`dim`** (*int*) – The embedding dimension.
- **`tau`** (*int*) – The embedding delay.

Return type 2D array (time, dimension)

Returns the embedded phase space trajectory.

`embedding` = **None**

The embedded time series.

`euclidean_distance_matrix`(*embedding*)

Return the euclidean distance matrix from an embedding of a time series.

Parameters **`embedding`** (*2D array (time, embedding dimension)*) – The phase space trajectory.

Return type 2D square array ("float32")

Returns the euclidean distance matrix.

`laminarity`(*v_min=2*, *resampled_dist=None*)

Return vertical line-based RQA measure laminarity LAM.

LAM is defined as the ratio of recurrence points that form vertical structures (of at least length v_min) to all recurrence points.

Parameters

- **`v_min`** (*number*) – The minimal vertical line length.
- **`resampled_dist`** (*1D array (integer)*) – resampled frequency distribution of vertical lines

Return number the laminarity LAM.

static `legendre_coordinates`(*x*, *dim=3*, *t=None*, *p=None*, *tau_w='est'*)

Return a phase space trajectory reconstructed using orthogonal polynomial filters.

The reconstructed state vector components are the zero-th to (dim-1)-th derivatives of the (possibly irregularly spaced) time series x as estimated by folding with the orthogonal polynomial filters that correspond to the sequence of measurement time points t.

This is a generalization for irregularly spaced time series of the "Legendre coordinates" introduced in Gibson et al. (1992).

Parameters

- **`x`** (*array-like*) – Time series values
- **`dim`** (*int*) – Dimension > 0 of reconstructed phase space. Default: 3
- **`t`** (*array-like or None*) – Optional array of measurement time points corresponding to the values in x. Default: [0,...,x.size-1]
- **`p`** (*int > 0 or None*) – No. of past and future time points to use for the estimation. Default: dim or determined by tau_w if given
- **`tau_w`** (*float > 0 or "est" or None*) – Optional (average) window width to use in determining p when p = None. Following Gibson et al. (1992), this should be about sqrt(3/< x**2 >) * std(x), or about a quarter period. If "est", this is estimated iteratively, starting with 4 * (max(t)-min(t)) / (N-1) and estimating x' from that.

Return type 2D array [observation index, dimension index]

Returns Estimated derivatives. Rows are reconstructed state vectors.

`manhattan_distance_matrix`(*embedding*)

Return the manhattan distance matrix from an embedding of a time series.

Parameters **embedding** (*2D array (time, embedding dimension)*) – The phase space trajectory.

Return type 2D square array ("float32")

Returns the manhattan distance matrix.

max_diaglength()

Return diagonal line-based RQA measure maximum diagonal line length L_max.

L_max is defined as the maximal length of a diagonal line in the recurrence matrix.

Return number the maximal diagonal line length L_max.

max_vertlength()

Return vertical line-based RQA measure maximal vertical line length V_max.

V_max is defined as the maximal length of a vertical line of the recurrence matrix.

Return number the maximal vertical line length V_max.

max_white_vertlength()

Return white vertical line-based RQA measure maximal white vertical line length.

It is defined as the maximal length of a white vertical line of the recurrence matrix and corresponds to the maximum recurrence time occuring in the time series.

Return number the maximal white vertical line length.

mean_recurrence_time(*w_min=1*)

Alias for `average_white_vertlength()` (see description there).

metric = None

The metric used for measuring distances in phase space.

missing_values = None

Controls special treatment of missing values in `RecurrencePlot.time_series`.

static **normalize_time_series**(*time_series*)

Normalize each component of a time series **in place**.

Works also for complex valued time series.

Note: Modifies the given array in place!

Parameters **time_series** (*2D array (time, dimension)*) – The time series to be normalized.

recurrence_matrix()

Return the current recurrence matrix R.

Return type 2D square Numpy array

Returns the current recurrence matrix R.

recurrence_probability(*lag=0*)

Return the recurrence probability. This is the probability, that the trajectory is recurrent after 'lag' time steps.

Contributed by Jan H. Feldhoff.

Return number the recurrence probability

recurrence_rate()

Return the recurrence rate RR.

RR gives the percentage of black dots in the recurrence plot.

Return number the recurrence rate RR.

static rejection_sampling(*dist*, *M*)

Rejection sampling of discrete frequency distribution.

Use simple rejection sampling algorithm for computing a resampled version of a given frequency distribution with discrete support.

Parameters

- **dist** (*1D array (integer)*) – discrete frequency distribution
- **M** (*int*) – number of resamplings

Return type 1D array (integer)

Returns the resampled frequency distribution.

resample_diagline_dist(*M*)

Return resampled frequency distribution of diagonal lines.

The resampled frequency distribution can be used for obtaining confidence bounds on diagonal line based RQA measures. This is described in detail in *[Schinkel2009]*.

Concerning the choice of the number of resamplings, Schinkel et al. write: "The number of resamplings is not generally agreed upon but common guidelines suggest values between 800 and 1500."

Parameters **M** (*int*) – number of resamplings

Return type 1D array (integer)

Returns the resampled frequency distribution of diagonal lines.

resample_vertline_dist(*M*)

Return resampled frequency distribution of vertical lines.

The resampled frequency distribution can be used for obtaining confidence bounds on vertical line based RQA measures. This is described in detail in *[Schinkel2009]*.

Concerning the choice of the number of resamplings, Schinkel et al. write: "The number of resamplings is not generally agreed upon but common guidelines suggest values between 800 and 1500."

Parameters **M** (*int*) – number of resamplings

Return type 1D array (integer)

Returns the resampled frequency distribution of vertical lines.

rqa_summary(*l_min=2*, *v_min=2*)

Return a selection of RQA measures.

The selection consists of the recurrence rate RR, the determinism DET, the average diagonal line length L and the laminarity LAM.

Parameters

- **l_min** (*int*) – The minimum diagonal line length.
- **v_min** (*int*) – The minimum vertical line length.

Return type Python dictionary

Returns a selection of RQA measures.

set_adaptive_neighborhood_size(*adaptive_neighborhood_size*, *order=None*)

Construct recurrence plot using the adaptive neighborhood size algorithm introduced in *[Xu2008]*.

The exact algorithm was deduced from private correspondence with the authors, as the description given in the above mentioned is not correct or at least ambiguous.

Modifies / sets the class variables R and N accordingly.

Parameters

- **`adaptive_neighborhood_size`** (*number*) – The number of adaptive nearest neighbors (recurrences) assigned to each state vector.
- **`order`** (*1D array of int32*) – The indices of state vectors in the order desired for processing by the algorithm. The standard order is $1, ..., N$.

`set_fixed_local_recurrence_rate`(*local_recurrence_rate*)
Set the recurrence plot to a fixed local recurrence rate.

This results in a fixed number of recurrences for each state vector, i.e., all state vectors have the same number of recurrences. Modifies / sets the class variables `R` and `N` accordingly.

Note: The resulting recurrence matrix R is generally asymmetric!

Parameters **`local_recurrence_rate`** (*number*) – The local recurrence rate.

`set_fixed_recurrence_rate`(*recurrence_rate*)
Set the recurrence plot to a fixed recurrence rate.

Modifies / sets the class variables `R` and `N` accordingly.

Parameters **`recurrence_rate`** (*number*) – The recurrence rate.

`set_fixed_threshold`(*threshold*)
Set the recurrence plot to a fixed threshold.

Modifies / sets the class variables `R` and `N` accordingly.

Parameters **`threshold`** (*number*) – The recurrence threshold.

`set_fixed_threshold_std`(*threshold_std*)
Set the recurrence plot to a fixed threshold in units of the standard deviation of the time series.

Calculates the absolute threshold and calls `set_fixed_threshold()`.

Parameters **`threshold_std`** (*number*) – The recurrence threshold in units of the standard deviation of the time series.

`silence_level` **= None**
The inverse level of verbosity of the object.

`sparse_rqa` **= None**
Controls sequential calculation of RQA measures.

`supremum_distance_matrix`(*embedding*)
Return the supremum distance matrix from an embedding of a time series.

Parameters **`embedding`** (*2D Numpy array (time, embedding dimension)*) – The phase space trajectory.

Return type 2D square Numpy array ("float32")

Returns the supremum distance matrix.

static **`threshold_from_recurrence_rate`**(*distance*, *recurrence_rate*)
Return the threshold for recurrence plot construction given the recurrence rate.

Be aware, that the returned threshold can only approximately give the desired recurrence rate. The accuracy depends on the distribution of values in the given distance matrix D.

Parameters

- **`distance`** (*2D square array.*) – The phase space distance matrix D.
- **`recurrence_rate`** (*number*) – The desired recurrence rate.

Return number the recurrence threshold corresponding to the desired recurrence rate.

static **threshold_from_recurrence_rate_fast** (*distance*, *recurrence_rate*, *rr_precision=0.001*)
Return the threshold for recurrence plot construction given the recurrence rate.

The threshold yielding a given recurrence_rate is approximated using a randomly selected rr_precision percent of the distance matrix' entries. Hence, the expected accuracy is lower than that achieved by using threshold_from_recurrence_rate.

Parameters

- **distance** (*2D square array.*) – The phase space distance matrix D.
- **recurrence_rate** (*number*) – The desired recurrence rate.
- **rr_precision** (*number*) – The desired precision of recurrence rate estimation.

Return number the recurrence threshold corresponding to the desired recurrence rate.

time_series = None
The time series from which the recurrence plot is constructed.

trapping_time (*v_min=2*, *resampled_dist=None*)
Alias for `average_vertlength()` (see description there).

twin_surrogates (*n_surrogates=1*, *min_dist=7*)
Generate surrogates based on the current (embedded) time series `embedding` using the twin surrogate method.

The twins surrogates have the same dimensionality as the (embedded) trajectory used for constructing the recurrence plot. If scalar surrogate time series are desired, any component of the twin surrogate trajectory may be isolated.

Twin surrogates share linear and nonlinear properties with the original time series, since they correspond to realizations of trajectories of the same dynamical systems with different initial conditions.

References: *[Thiel2006]* [*], *[Marwan2007]*.

Parameters

- **min_dist** (*number*) – The minimum temporal distance for twins.
- **n_surrogates** (*int*) – The number of twin surrogate trajectories to be returned.

Return type 3D array (surrogate number, time, dimension)

Returns the twin surrogate trajectories.

twins (*min_dist=7*)
Return list of the twins of each state vector based on the recurrence matrix.

Two state vectors are said to be twins if they share the same recurrences, i.e., if the corresponding rows or columns in the recurrence plot are identical.

References: *[Thiel2006]*, *[Marwan2007]*.

Parameters **min_dist** (*number*) – The minimum temporal distance for twins.

Return [[number]] the list of twins for each state vector in the time series.

vert_entropy (*v_min=2*, *resampled_dist=None*)
Return vertical line-based RQA measure vertical line entropy.

It is defined as the entropy of the probability to find a vertical line of exactly length l in the RP - reflects the complexity of the RP with respect to vertical lines.

Parameters

- **v_min** (*int*) – The minimal vertical line length.
- **resampled_dist** (*1D array (integer)*) – resampled frequency distribution of vertical lines

Return number the vertical line-based entropy.

vertline_dist()

Return the frequency distribution of vertical line lengths $P(v)$.

The v th entry of $P(v)$ contains the number of vertical lines of length v.

Return type 1D array (int32)

Returns the frequency distribution of vertical line lengths $P(v)$.

white_vert_entropy(*w_min=1*)

Return white vertical line-based RQA measure white vertical line entropy.

It is defined as the entropy of the probability to find a white vertical line of exactly length l in the RP - reflects the complexity of the RP with respect to white vertical lines (recurrence times).

Parameters **w_min** (*int*) – Minimal white vertical line length (recurrence time).

Return number the white vertical line-based entropy.

white_vertline_dist()

Return the frequency distribution of white vertical line lengths $P(w)$.

The w th entry of $P(w)$ contains the number of white vertical lines of length w.

The length of a white vertical line in a recurrence plot corresponds to the time the system takes to return close to an earlier state.

Return type 1D array (int32)

Returns the frequency distribution of white vertical line lengths $P(w)$.

5.3.7 timeseries.surrogates

Provides classes for analyzing spatially embedded complex networks, handling multivariate data and generating time series surrogates.

class `pyunicorn.timeseries.surrogates.`**Surrogates**(*original_data*, *silence_level=1*)

Bases: `object`

Encapsulates structures and methods related to surrogate time series.

Provides data structures and methods to generate surrogate data sets from a set of time series and to evaluate the significance of various correlation measures using these surrogates.

More information on time series surrogates can be found in [Schreiber2000] and [Kantz2006].

AAFT_surrogates(*original_data*)

Return surrogates using the amplitude adjusted Fourier transform method.

Reference: [Schreiber2000]

Parameters **original_data** (*2D array [index, time]*) – The original time series.

Return type 2D array [index, time]

Returns The surrogate time series.

static SmallTestData()

Return Surrogates instance representing test a data set of 6 time series.

Return type Surrogates instance

Returns a Surrogates instance for testing purposes.

__init__(*original_data*, *silence_level=1*)

Initialize an instance of Surrogates.

Note: The order of array dimensions is different from the standard of `core`. Here it is [index, time] for reasons of computational speed!

Parameters

- **`original_data`** (*2D array [index, time]*) – The original time series for surrogate generation.
- **`silence_level`** (*int*) – The inverse level of verbosity of the object.

`__str__`()
Returns a string representation.

`__weakref__`
list of weak references to the object (if defined)

`clear_cache`()
Clean up cache.

`correlated_noise_surrogates`(*original_data*)
Return Fourier surrogates.

Generate surrogates by Fourier transforming the *`original_data`* time series (assumed to be real valued), randomizing the phases and then applying an inverse Fourier transform. Correlated noise surrogates share their power spectrum and autocorrelation function with the original_data time series.

The Fast Fourier transforms of all time series are cached to facilitate a faster generation of several surrogates for each time series. Hence, *`clear_cache()`* has to be called before generating surrogates from a different set of time series!

Note: The amplitudes are not adjusted here, i.e., the individual amplitude distributions are not conserved!

Examples:

The power spectrum is conserved up to small numerical deviations:

```
>>> ts = Surrogates.SmallTestData().original_data
>>> surrogates = Surrogates.                SmallTestData().correlated_noise_surrogates(t
>>> all(r(np.abs(np.fft.fft(ts,         axis=1))[0,1:10]) ==                 r(np.abs(np.
True
```

However, the time series amplitude distributions differ:

```
>>> all(np.histogram(ts[0,:])[0] == np.histogram(surrogates[0,:])[0])
False
```

Parameters **`original_data`** (*2D array [index, time]*) – The original time series.

Return type 2D array [index, time]

Returns The surrogate time series.

`embed_time_series_array`(*time_series_array*, *dimension*, *delay*)
Return a delay embedding of all time series.

Note: Only works for scalar time series!

Example:

```
>>> ts = Surrogates.SmallTestData().original_data
>>> Surrogates.SmallTestData().embed_time_series_array(
...     time_series_array=ts, dimension=3, delay=2)[0,:6,:]
array([[ 0.        ,  0.61464833,  1.14988147],
       [ 0.31244015,  0.89680225,  1.3660254 ],
       [ 0.61464833,  1.14988147,  1.53884177],
       [ 0.89680225,  1.3660254 ,  1.6636525 ],
       [ 1.14988147,  1.53884177,  1.73766672],
       [ 1.3660254 ,  1.6636525 ,  1.76007351]])
```

Parameters

- **`time_series_array`** (*2D array [index, time]*) – The time series array to be normalized.
- **`dimension`** (*int*) – The embedding dimension.
- **`delay`** (*int*) – The embedding delay.

Return type 3D array [index, time, dimension]

Returns the embedded time series.

static **`eval_fast_code`** (*function*, *original_data*, *surrogates*)

Evaluate performance of fast and slow versions of algorithms.

Designed for evaluating fast and dirty C code against cleaner code using Blitz arrays. Does some profiling and returns the total error between the results.

Parameters

- **`function`** (*Python function*) – The function to be evaluated.
- **`original_data`** (*2D array [index, time]*) – The original time series.
- **`surrogates`** (*2D array [index, time]*) – The surrogate time series.

Return float The total squared difference between resulting matrices.

static **`normalize_time_series_array`** (*time_series_array*)

Normalize an array of time series to zero mean and unit variance individually for each individual time series.

Modifies the given array in place!

Examples:

```
>>> ts = Surrogates.SmallTestData().original_data
>>> Surrogates.SmallTestData().normalize_time_series_array(ts)
>>> r(ts.mean(axis=1))
array([ 0.,  0.,  0.,  0.,  0.,  0.])
>>> r(ts.std(axis=1))
array([ 1.,  1.,  1.,  1.,  1.,  1.])
```

Parameters **`time_series_array`** (*2D array [index, time]*) – The time series array to be normalized.

`original_data` **= None**

The original time series for surrogate generation.

`original_distribution` (*test_function*, *original_data*, *n_bins=100*)

Return a normalized histogram of a similarity measure matrix.

The absolute value of the similarity measure is used, since only the degree of similarity was of interest originally.

Parameters

- **`test_function`** (*Python function*) – The function implementing the similarity measure.
- **`original_data`** (*2D array [index, time]*) – The original time series.

- **n_bins** (*int*) – The number of bins for estimating prob. distributions.

Return type tuple of 1D arrays ([bins],[bins])

Returns the similarity measure histogram and lower bin boundaries.

recurrence_plot (*embedding*, *threshold*)

Return the recurrence plot from an embedding of a time series.

Uses supremum norm.

Parameters

- **embedding** (*2D array [time, dimension]*) – The embedded time series.
- **threshold** (*float*) – The recurrence threshold.

Return type 2D array [time, time]

Returns the recurrence matrix.

refined_AAFT_surrogates (*original_data*, *n_iterations*, *output='true_amplitudes'*)

Return surrogates using the iteratively refined amplitude adjusted Fourier transform method.

A set of AAFT surrogates (`AAFT_surrogates()`) is iteratively refined to produce a closer match of both amplitude distribution and power spectrum of surrogate and original data.

Reference: *[Schreiber2000]*

Parameters

- **original_data** (*2D array [index, time]*) – The original time series.
- **n_iterations** (*int*) – Number of iterations / refinement steps
- **output** (*str*) – Type of surrogate to return. “true_amplitudes”: surrogates with correct amplitude distribution, “true_spectrum”: surrogates with correct power spectrum, “both”: return both outputs of the algorithm.

Return type 2D array [index, time]

Returns The surrogate time series.

silence_level = None

(string) - The inverse level of verbosity of the object.

static test_mutual_information (*original_data*, *surrogates*, *n_bins=32*, *fast=True*)

Return a test matrix of mutual information (zero lag).

The test matrix’s entry (i, j) contains the mutual information between original time series i and surrogate time series j at zero lag. The resulting matrix is useful for significance tests based on the mutual information matrix of the original data.

Note: Assumes, that original_data and surrogates are already normalized.

Parameters

- **original_data** (*2D array [index, time]*) – The original time series.
- **surrogates** (*2D Numpy array [index, time]*) – The surrogate time series.
- **n_bins** (*int*) – Number of bins for estimating prob. distributions.
- **fast** (*bool*) – fast or slow algorithm to be used.

Return type 2D array [index, index]

Returns the mutual information test matrix.

static `test_pearson_correlation`(*original_data*, *surrogates*, *fast=True*)

Return a test matrix of the Pearson correlation coefficient (zero lag).

The test matrix's entry (i, j) contains the Pearson correlation coefficient between original time series i and surrogate time series j at lag zero. The resulting matrix is useful for significance tests based on the Pearson correlation matrix of the original data.

Note: Assumes, that original_data and surrogates are already normalized.

Parameters

- **`original_data`** (*2D array [index, time]*) – The original time series.
- **`surrogates`** (*2D array [index, time]*) – The surrogate time series.

Return type 2D array [index, index]

Returns the Pearson correlation test matrix.

`test_threshold_significance`(*surrogate_function*, *test_function*, *realizations=1*, *n_bins=100*, *interval=(-1, 1)*)

Return a test distribution for a similarity measure.

Perform a significance test on the values of a correlation measure based on original_data time series and surrogate data. Returns a density estimate (histogram) of the absolute value of the correlation measure over all realizations.

The resulting distribution of the values of similarity measure from original and surrogate time series is of use for testing the statistical significance of a selected threshold value for climate network generation.

Parameters

- **`surrogate_function`** (*Python function*) – The function implementing the surrogates.
- **`test_function`** (*Python function*) – The function implementing the similarity measure.
- **`realizations`** (*int*) – The number of surrogates to be created for each time series.
- **`n_bins`** (*int*) – The number of bins for estimating probability distribution of test similarity measure.
- **`interval`** (*(float, float)*) – The range over which to estimate similarity measure distribution.

Return type tuple of 1D arrays ([bins],[bins])

Returns similarity measure test histogram and lower bin boundaries.

`twin_surrogates`(*original_data*, *dimension*, *delay*, *threshold*, *min_dist=7*)

Return surrogates using the twin surrogate method.

Scalar twin surrogates are created by isolating the first component (dimension) of the twin surrogate trajectories.

Twin surrogates share linear and nonlinear properties with the original time series, since they correspond to realizations of trajectories of the same dynamical systems with different initial conditions.

References: *[Thiel2006]* [*], *[Marwan2007]*.

The twin lists of all time series are cached to facilitate a faster generation of several surrogates for each time series. Hence, `clear_cache()` has to be called before generating twin surrogates from a different set of time series!

Parameters

- **`original_data`** (*2D array [index, time]*) – The original time series.

- **dimension** (*int*) – The embedding dimension.
- **delay** (*int*) – The embedding delay.
- **threshold** (*float*) – The recurrence threshold.
- **min_dist** (*number*) – The minimum temporal distance for twins.

Return type 2D array [index, time]

Returns the twin surrogates.

twins (*embedding_array*, *threshold*, *min_dist=7*)

Return list of the twins of each state vector for all time series.

Two state vectors are said to be twins if they share the same recurrences, i.e., if the corresponding rows or columns in the recurrence plot are identical.

References: *[Thiel2006]*, *[Marwan2007]*.

Parameters

- **embedding_array** (*3D array [index, time, dimension]*) – The embedded time series array.
- **threshold** (*float*) – The recurrence threshold.
- **min_dist** (*number*) – The minimum temporal distance for twins.

Return type [[number]]

Returns the list of twins for each state vector in the time series.

white_noise_surrogates (*original_data*)

Return a shuffled copy of a time series array.

Each time series is shuffled individually. The surrogates correspond to realizations of white noise consistent with the original_data time series' amplitude distribution.

Example (Distributions of white noise surrogates should the same as for the original data):

```
>>> ts = Surrogates.SmallTestData().original_data
>>> surrogates = Surrogates.                    SmallTestData().white_noise_surrogates(ts)
>>> np.histogram(ts[0,:])[0]
array([21, 12,  9, 15, 34, 35, 18, 12, 16, 28])
>>> np.histogram(surrogates[0,:])[0]
array([21, 12,  9, 15, 34, 35, 18, 12, 16, 28])
```

Parameters **original_data** (*2D array [index, time]*) – The original time series.

Return type 2D array [index, time]

Returns The surrogate time series.

5.3.8 timeseries.visibility_graph

Provides classes for the analysis of dynamical systems and time series based on recurrence plots, including measures of recurrence quantification analysis (RQA) and recurrence network analysis.

class pyunicorn.timeseries.visibility_graph.**VisibilityGraph**(*time_series*, *timings=None*, *missing_values=False*, *horizontal=False*, *silence_level=0*)

Bases: pyunicorn.core.interacting_networks.InteractingNetworks

Class VisibilityGraph for generating and analyzing visibility graphs of time series.

Visibility graphs were initially applied for time series analysis by *[Lacasa2008]*.

`__init__`(*time_series*, *timings=None*, *missing_values=False*, *horizontal=False*, *silence_level=0*)
Missing values are handled as infinite values, effectively separating the visibility graph into different disconnected components.

Note: Missing values have to be marked by the Numpy NaN flag!

Parameters

- **`time_series`** (*2D array (time, dimension)*) – The time series to be analyzed, can be scalar or multi-dimensional.
- **`timings`** (*str*) – Timings of the observations in `time_series`.
- **`missing_values`** (*bool*) – Toggle special treatment of missing values in `time_series`.
- **`horizontal`** (*bool*) – Indicates whether a horizontal visibility relation is used.
- **`silence_level`** (*number*) – Inverse level of verbosity of the object.

`__str__`()
Returns a string representation.

`advanced_betweenness`()
Return betweenness of a node with respect to all pairs of nodes in its future.

`advanced_closeness`()
Return average path length to nodes in the future of a node.

`advanced_degree`()
Return number of neighbors in the future of a node.

`advanced_local_clustering`()
Return probability that two neighbors of a node in its future are connected.

`boundary_corrected_closeness`()
Return a weighted closeness corrected for trivial boundary effects.

`boundary_corrected_degree`()
Return a weighted degree corrected for trivial boundary effects.

`missing_values` = **None**
Controls special treatment of missing values in `time_series`.

`retarded_betweenness`()
Return betweenness of a node with respect to all pairs of nodes in its past.

`retarded_closeness`()
Return average path length to nodes in the past of a node.

`retarded_degree`()
Return number of neighbors in the past of a node.

`retarded_local_clustering`()
Return probability that two neighbors of a node in its past are connected.

`silence_level` = **None**
The inverse level of verbosity of the object.

`time_series` = **None**
The time series from which the visibility graph is constructed.

`timings` = **None**
The timimgs of the time series data points.

trans_betweenness()
Return betweenness of a node with respect to all pairs of nodes with one node the past and one node in the future, respectively.

visibility_relations()
TODO

visibility_relations_horizontal()
TODO

5.4 funcnet

Constructing and analysing general functional networks.

5.4.1 funcnet.coupling_analysis

Provides classes for analyzing spatially embedded complex networks, handling multivariate data. Written by Jakob Runge.

class `pyunicorn.funcnet.coupling_analysis.`**CouplingAnalysis**(*data*, *silence_level=0*)
Bases: `object`

Contains methods to calculate coupling matrices from large arrays of scalar time series. Comprises linear and information-theoretic measures, lagged and directed couplings.

__init__(*data*, *silence_level=0*)
Initialize an instance of CouplingAnalysis from data array.

Parameters

- **data** (*multidimensional numpy array*) – The time series array with time in first dimension.
- **silence_level** (*int >= 0*) – The higher, the less progress info is output.

__str__()
Return a string representation of the CouplingAnalysis object.

__weakref__
list of weak references to the object (if defined)

static _get_nearest_neighbors(*array*, *xyz*, *k*, *standardize=True*)
Returns nearest-neighbors for conditional mutual information estimator.

Reference: *[Kraskov2004]*

Parameters

- **array** (*array (float)*) – data array.
- **xyz** (*array [int(0|1|2)]*) – identifier of X, Y, Z in CMI
- **k** (*int [int>=1]*) – nearest-neighbor MI estimation parameter.
- **standardize** (*bool*) – standardize array before estimation. (default: True)

Return type tuple of arrays

Returns nearest neighbors for each sample point.

static _par_corr_to_cmi(*par_corr*)
Transformation of partial correlation to conditional mutual information scale using the (multivariate) Gaussian assumption.

Parameters **par_corr** (*float or array*) – partial correlation

Return type float

Returns transformed partial correlation.

static _quantile_bin_array (*array*, *bins=6*)

Returns symbolified array with aequi-quantile binning.

This partition results in a uniform distribution of the marginals.

Parameters

- **array** (*array*) – data
- **bins** (*int*) – number of bins

Return type array

Returns converted data

static bincount_hist (*symb_array*)

Computes histogram from symbolic array.

Parameters **symb_array** (*array of integers*) – symbolic data

Return type array

Returns (unnormalized) histogram

static create_plogp (*T*)

Precalculation of p*log(p) needed for entropies.

Parameters **T** (*int*) – sample length

Return type array

Returns p*log(p) array from p=1 to p=T

cross_correlation (*tau_max=0*, *lag_mode='max'*)

Return cross correlation between all pairs of nodes.

Two lag-modes are available (default: lag_mode='max'):

lag_mode = 'all': Return 3-dimensional array of lagged cross correlations between all pairs of nodes. An entry (i, j, τ) corresponds to $\rho(X_t^i - \tau, X_t^j)$ for positive lags tau, i.e., the direction i –> j for $\tau \neq 0$.

lag_mode = 'max': Return matrix of absolute maxima and corresponding lags of lagged cross correlation (CC) between all pairs of nodes. Returns two usually asymmetric matrices of CC values and lags: In each matrix, an entry (i, j) corresponds to the (positive or negative) value and lag, respectively, at absolute maximum of $\rho(X_t^i - \tau, X_t^j)$ for positive lags tau, i.e., the direction i –> j for $\tau > 0$. The matrices are, thus, asymmetric. The function *symmetrize_by_absmax()* can be used to obtain a symmetric matrix.

Example:

```
>>> coup_ana = CouplingAnalysis(CouplingAnalysis.test_data())
>>> similarity_matrix, lag_matrix = coup_ana.cross_correlation(
...     tau_max=5, lag_mode='max')
>>> r((similarity_matrix, lag_matrix))
(array([[ 1.    ,  0.757 ,  0.779 ,  0.7536],
        [ 0.4847,  1.    ,  0.4502,  0.5197],
        [ 0.6219,  0.5844,  1.    ,  0.5992],
        [ 0.4827,  0.5509,  0.4996,  1.    ]]),
 array([[0, 4, 1, 2], [0, 0, 0, 0], [0, 3, 0, 1], [0, 2, 0, 0]]))
```

Parameters

- **tau_max** (*int [int>=0]*) – maximum lag of cross correlation lag function.
- **lag_mode** (*str [('max'|'all')]*) – lag-mode of cross correlations to return.

Return type 3D-array or tuple of matrices

Returns all-lag array or matrices of value and lag at the absolute maximum.

information_transfer(*tau_max=0, estimator='knn', knn=10, past=1, cond_mode='ity', lag_mode='max'*)

Return bivariate information transfer between all pairs of nodes.

Two condition modes of information transfer are available as described in *[Runge2012b]*.

Information transfer to Y (ITY):

$$I(X_t^i - \tau, X_t^j | X_t^j - 1, ..., X_t^j - past)$$

Momentary information transfer (MIT):

$$I(X_t^i - \tau, X_t^j | X_t^j - 1, ..., X_t^j - past, X_t^i - \tau - 1, ..., X_t^j - \tau - past)$$

Two estimators are available:

estimator = 'knn' (Recommended): Based on k-nearest-neighbors *[Kraskov2004]*, version 1 in their paper. Larger k have smaller variance, but larger (typically negative) bias, and vice versa.

estimator = 'gauss': Captures only linear part of association. Essentially estimates a transformed partial correlation.

Two lag-modes are available (default: lag_mode='max'):

lag_mode = 'all': Return 3-dimensional array of lag-functions between all pairs of nodes. An entry (i, j, τ) corresponds to $I(X_t^i - \tau, X_t^j | ...)$ for positive lags tau, i.e., the direction i –> j for $\tau \neq 0$.

lag_mode = 'max': Return matrix of absolute maxima and corresponding lags of lag-functions between all pairs of nodes. Returns two usually asymmetric matrices of values and lags: In each matrix, an entry (i, j) corresponds to the value and lag, respectively, at absolute maximum of $I(X_t^i - \tau, X_t^j | ...)$ for positive lags tau, i.e., the direction i –> j for $\tau > 0$. The matrices are, thus, asymmetric. The function `symmetrize_by_absmax()` can be used to obtain a symmetric matrix.

Example:

```
>>> coup_ana = CouplingAnalysis(CouplingAnalysis.test_data())
>>> similarity_matrix, lag_matrix = coup_ana.information_transfer(
...     tau_max=5, estimator='knn', knn=10)
>>> r((similarity_matrix, lag_matrix))
(array([[ 0.    ,  0.1544,  0.3261,  0.3047],
       [ 0.0218,  0.    ,  0.0394,  0.0976],
       [ 0.0134,  0.0663,  0.    ,  0.1502],
       [ 0.0066,  0.0694,  0.0401,  0.    ]]),
array([[0, 2, 1, 2], [5, 0, 0, 0], [5, 1, 0, 1], [5, 0, 0, 0]]))
```

Parameters

- **tau_max** (*int [int>=0]*) – maximum lag of ITY lag function.
- **past** (*int [int>=1]*) – maximum lag of past history.
- **knn** (*int [int>=1]*) – nearest-neighbor ITY estimation parameter. (default: 10)
- **bins** (*int [int>=2]*) – binning ITY estimation parameter. (default: 6)
- **estimator** (*str [('knn'|'gauss')]*) – ITY estimator. (default: 'knn')
- **cond_mode** (*str [('ity'|'mit')]*) – condition mode. (default: 'ity')
- **lag_mode** (*str [('max'|'all')]*) – lag-mode of ITY to return.

Return type 3D-array or tuple of matrices

Returns all-lag array or matrices of value and lag at the absolute maximum.

mutual_information(*tau_max=0*, *estimator='knn'*, *knn=10*, *bins=6*, *lag_mode='max'*)

Return mutual information (MI) between all pairs of nodes.

Three estimators are available:

estimator = 'knn' (Recommended): Based on k-nearest-neighbors *[Kraskov2004]*, version 1 in their paper. Larger k have smaller variance, but larger (typically negative) bias, and vice versa.

estimator = 'binning': Binning estimator based on equal-quantile binning.

estimator = 'gauss': Captures only linear part of association. Essentially estimates a transformed partial correlation.

Two lag-modes are available (default: lag_mode='max'):

lag_mode = 'all': Return 3-dimensional array of lagged MI between all pairs of nodes. An entry (i, j, τ) corresponds to $I(X_t^i - \tau, X_t^j)$ for positive lags tau, i.e., the direction i –> j for $\tau \neq 0$.

lag_mode = 'max': Return matrix of absolute maxima and corresponding lags of lagged MI between all pairs of nodes. Returns two usually asymmetric matrices of MI values and lags: In each matrix, an entry (i, j) corresponds to the value and lag, respectively, at absolute maximum of $I(X_t^i - \tau, X_t^j)$ for positive lags tau, i.e., the direction i –> j for $\tau > 0$. The matrices are, thus, asymmetric. The function `symmetrize_by_absmax()` can be used to obtain a symmetric matrix.

Reference: *[Kraskov2004]*

Example:

```
>>> coup_ana = CouplingAnalysis(CouplingAnalysis.test_data())
>>> similarity_matrix, lag_matrix = coup_ana.mutual_information(
...     tau_max=5, knn=10, estimator='knn')
>>> similarity_matrix, lag_matrix
(array([[ 4.65048742,  0.43874303,  0.46520019,  0.41257444],
       [ 0.14704162,  4.65048742,  0.10645443,  0.16393046],
       [ 0.24829103,  0.2125767 ,  4.65048742,  0.22044939],
       [ 0.12093173,  0.19902836,  0.14530452,  4.65048742]],
       dtype=float32),
array([[0, 4, 1, 2],
       [0, 0, 0, 0],
       [0, 2, 0, 1],
       [0, 2, 0, 0]], dtype=int8))
```

Parameters

- **tau_max** (*int [int>=0]*) – maximum lag of MI lag function.
- **knn** (*int [int>=1]*) – nearest-neighbor MI estimation parameter. (default: 10)
- **bins** (*int [int>=2]*) – binning MI estimation parameter. (default: 6)
- **estimator** (*str [('knn'|'binning'|'gauss')]*) – MI estimator. (default: 'knn')
- **lag_mode** (*str [('max'|'all')]*) – lag-mode of MI to return.

Return type 3D-array or tuple of matrices

Returns all-lag array or matrices of value and lag at the absolute maximum.

silence_level = None

(int>=0) higher -> less progress info

symmetrize_by_absmax(*similarity_matrix*, *lag_matrix*)

Returns symmetrized similarity matrix.

Computes the largest absolute value for each pair (i,j) and (j,i) and returns the in-place changed matrices of measures and lags. A negative lag for an entry (i,j) in the lag_matrix then indicates a 'direction' j –> i regarding the peak of the lag function, and vice versa for a positive lag.

Example:

```
>>> coup_ana = CouplingAnalysis(CouplingAnalysis.test_data())
>>> similarity_matrix, lag_matrix = coup_ana.cross_correlation(
...     tau_max=2)
>>> r((similarity_matrix, lag_matrix))
(array([[ 1.    , 0.698 , 0.7788, 0.7535],
        [ 0.4848, 1.    , 0.4507, 0.52  ],
        [ 0.6219, 0.5704, 1.    , 0.5996],
        [ 0.4833, 0.5503, 0.5002, 1.    ]]),
 array([[0, 2, 1, 2], [0, 0, 0, 0],
        [0, 2, 0, 1], [0, 2, 0, 0]]))
>>> r(coup_ana.symmetrize_by_absmax(similarity_matrix, lag_matrix))
(array([[ 1.    , 0.698 , 0.7788, 0.7535],
        [ 0.698 , 1.    , 0.5704, 0.5503],
        [ 0.7788, 0.5704, 1.    , 0.5996],
        [ 0.7535, 0.5503, 0.5996, 1.    ]]),
 array([[ 0, 2, 1, 2], [-2, 0, -2, -2],
        [-1, 2, 0, 1], [-2, 2, -1, 0]]))
```

Parameters

- **`similarity_matrix`** (*array-like [float]*) – array-like [node, node] matrix of similarity estimates
- **`lag_matrix`** (*array-like [int>=0]*) – array-like [node, node] matrix of lags

Return type tuple of arrays

Returns the value at the absolute maximum and the (pos or neg) lag.

static `test_data``()`
Return example test data as discussed in pyunicorn description paper.

5.4.2 funcnet.coupling_analysis_pure_python

Provides classes for analyzing spatially embedded complex networks, handling multivariate data and generating time series surrogates.

Written by Jakob Runge. CMSI Method Reference: *[Pompe2011]*

class `pyunicorn.funcnet.coupling_analysis_pure_python.`**`CouplingAnalysisPurePython`**`(`*dataarray, only_tri=Fal si- lence_level=*

Bases: `object`

Contains methods to calculate coupling matrices from large arrays of scalar time series.

Comprises linear and information theoretic measures, lagged and directed (causal) couplings.

`__init__``(`*dataarray*, *only_tri=False*, *silence_level=0*`)`
Initialize an instance of CouplingAnalysisPurePython.

Possible choices for only_tri:

- "True" will calculate only the upper triangle of the coupling matrix, excluding the diagonal, assuming symmetry (not for directed measures)
- **"False" will calculate the whole matrix (asymmetry somes from** different integration ranges)

Parameters

- **`dataarray`** (*4D, 3D or 2D Numpy array [time, index, index] or [time, index]*) – The time series array with time in first dimension

- **only_tri** (*bool*) – Symmetric/asymmetric assumption on coupling matrix.
- **silence_level** (*int*) – The inverse level of verbosity of the object.

__str__()
Return a string representation of the CouplingAnalysisPurePython object.

__weakref__
list of weak references to the object (if defined)

_calculate_cc(*array*, *corr_range*, *tau_max*, *lag_mode*)
Returns the CC matrix.

Parameters

- **tau_max** (*int*) – maximum lag in both directions, including last lag
- **lag_mode** (*str*) – output mode

Return type 3D numpy array (float) [index, index, index]

Returns correlation matrix with different lag_mode choices

## lag_mode dict mode = self.lag_modi[lag_mode]

_calculate_mi(*array*, *corr_range*, *bins*, *tau_max*, *lag_mode*)
Returns the mi matrix.

Parameters

- **bins** (*int*) – number of bins for estimating MI
- **tau_max** (*int*) – maximum lag in both directions, including last lag
- **lag_mode** (*str*) – output mode

Return type 3D numpy array (float) [index, index, index]

Returns correlation matrix with different lag_mode choices

correlatedNoiseSurrogates(*original*)
Generates surrogates by Fourier transforming the original time series, randomizing the phases and then applying an inverse Fourier transform. Correlated noise surrogates share their power spectrum and autocorrelation function with the original time series.

Parameters **original** (*2D array*) – dim. 0 is index of time series, dim. 1 is time

Returns surrogate time series (same dimensions as original)

cross_correlation(*tau_max=0*, *lag_mode='all'*)
Returns the normalized cross correlation from all pairs of nodes from a range of time lags.

The calculation ranges are shown below:

```
(------------------------total_time--------------------------)
(---tau_max---)(---------corr_range------------)(---tau_max---)
```

CC is calculated about corr_range and with the other time series shifted by tau

Possible choices for lag_mode:

- "all" will return the full function for all lags, possible large memory need if only_tri is True, only the upper triangle contains the values, the lower one is zeros
- "sum" will return the sum over positive and negative lags seperatly, each inclunding tau=0 corrmat[0] is the positive sum, corrmat[1] the negative sum
- "max" will return only the maximum coupling (in corrmat[0]) and its lag (in corrmat[1])

Parameters

- **tau_max** (*int*) – maximum lag in both directions, including last lag

- **`lag_mode`** (*str*) – the output mode

Return type 3D numpy array (float) [index, index, index]

Returns correlation matrix with different lag_mode choices

`mutual_information` (*bins=16*, *tau_max=0*, *lag_mode='all'*)

Returns the normalized mutual information from all pairs of nodes from a range of time lags.

MI = H_x + H_y - H_xy

Uses adaptive bins, where each marginal bin contains the same number of samples. Then the marginal entropies have equal probable distributions H_x = H_y = log(bins)

The calculation ranges are shown below:

```
(------------------------total_time--------------------------)
(---tau_max---)(---------corr_range------------)(---tau_max---)
```

MI is calculated about corr_range and with the other time series shifted by tau

Possible choices for lag_mode:

- "all" will return the full function for all lags, possible large memory need if only_tri is True, only the upper triangle contains the values, the lower one is zeros
- "sum" will return the sum over positive and negative lags seperatly, each inclunding tau=0 corrmat[0] is the positive sum, corrmat[1] the negative sum
- "max" will return only the maximum coupling (in corrmat[0]) and its lag (in corrmat[1])

Parameters

- **`bins`** (*int*) – number of bins for estimating MI
- **`tau_max`** (*int*) – maximum lag in both directions, including last lag
- **`lag_mode`** (*str*) – output mode

Return type 3D numpy array (float) [index, index, index]

Returns correlation matrix with different lag_mode choices

`mutual_information_edges` (*bins=16*, *tau=0*, *lag_mode='all'*)

Returns the normalized mutual information from all pairs of nodes from a range of time lags.

MI = H_x + H_y - H_xy

Uses adaptive bins, where each marginal bin contains the same number of samples. Then the marginal entropies have equal probable distributions H_x = H_y = log(bins)

The calculation ranges are shown below:

```
(------------------------total_time--------------------------)
(---tau_max---)(---------corr_range------------)(---tau_max---)
```

MI is calculated about corr_range and with the other time series shifted by tau

Possible choices for lag_mode:

- "all" will return the full function for all lags, possible large memory need if only_tri is True, only the upper triangle contains the values, the lower one is zeros
- "sum" will return the sum over positive and negative lags seperatly, each inclunding tau=0 corrmat[0] is the positive sum, corrmat[1] the negative sum
- "max" will return only the maximum coupling (in corrmat[0]) and its lag (in corrmat[1])

Parameters

- **bins** (*int*) – number of bins for estimating MI
- **tau_max** (*int*) – maximum lag in both directions, including last lag
- **lag_mode** (*str*) – output mode

Return type 2D numpy array (float) [index, index]

Returns bin edges for zero lag

shuffled_surrogate_for_cc (*fourier=False*, *tau_max=1*, *lag_mode='all'*)

Returns a correlation matrix calculated with an independently shuffled surrogate of the dataarray of length corr_range for all taus.

Parameters

- **corr_range** (*int*) – length of sample
- **tau_max** (*int*) – maximum lag in both directions, including last lag
- **lag_mode** (*str*) – output mode

Return type 3D numpy array (float) [index, index, index]

Returns correlation matrix with different lag_mode choices

shuffled_surrogate_for_mi (*fourier=False*, *bins=16*, *tau_max=0*, *lag_mode='all'*)

Returns a shuffled surrogate of normalized mutual information from all pairs of nodes from a range of time lags.

Parameters

- **bins** (*int*) – number of bins for estimating MI
- **tau_max** (*int*) – maximum lag in both directions, including last lag
- **lag_mode** (*str*) – output mode

Return type 3D numpy array (float) [index, index, index]

Returns correlation matrix with different lag_mode choices

time_surrogate_for_cc (*sample_range=100*, *tau_max=1*, *lag_mode='all'*)

Returns a joint shuffled surrogate of the full dataarray of length sample_range for all taus.

Used for time evolution analysis. First one initializes the CouplingAnalysis class with the full dataarray and then this function is called for every single surrogate.

Parameters

- **sample_range** (*int*) – length of sample
- **tau_max** (*int*) – maximum lag in both directions, including last lag
- **lag_mode** (*str*) – output mode

Return type 3D numpy array (float) [index, index, index]

Returns correlation matrix with different lag_mode choices

time_surrogate_for_mi (*bins=16*, *sample_range=100*, *tau_max=1*, *lag_mode='all'*)

Returns a joint shuffled surrogate of the full dataarray of length sample_range for all taus.

Used for time evolution analysis. First one initializes the CouplingAnalysis class with the full dataarray and then this function is called for every single surrogate.

Parameters

- **sample_range** (*int*) – length of sample
- **bins** (*int*) – number of bins for estimating MI
- **tau_max** (*int*) – maximum lag in both directions, including last lag

- **lag_mode** (*str*) – output mode

Return type 3D numpy array (float) [index, index, index]

Returns correlation matrix with different lag_mode choices

5.5 utils

Parallelization, interactive network navigator, helpers.

5.5.1 utils.mpi

Module for parallelization using mpi4py.

Allows for easy parallelization in master/slaves mode with one master submitting function or method calls to slaves. Uses mpi4py if available, otherwise processes calls sequentially in one process.

Examples:

Save the following lines in `demo_mpi.py` and run:

```
> mpirun -n 10 python demo_mpi.py
```

1. Use master/slaves parallelization with the Network class:

```
from pyunicorn import Network, mpi


def master():
    net = Network.BarabasiAlbert(n_nodes=1000, n_links_each=10)
    print net.newman_betweenness()
    mpi.info()

mpi.run()
```

2. Do a Monte Carlo simulation as master/slaves:

```
from pyunicorn import Network, mpi


def do_one():
    net = Network.BarabasiAlbert(n_nodes=100, n_links_each=10)
    return net.global_clustering()


def master():
    n = 1000
    for i in range(0, n):
        mpi.submit_call("do_one", ())
    s = 0
    for i in range(0, n):
        s += mpi.get_next_result()
    print s/n
    mpi.info()

mpi.run()
```

3. Do a parameter scan without communication with a master, and just save the results in files:

```
import numpy
from pyunicorn import Network, mpi

offset = 10
n_max = 1000
s = 0
n = mpi.rank + offset
while n <= n_max + offset:
    s += Network.BarabasiAlbert(n_nodes=n).global_clustering()
    n += mpi.size

numpy.save("s"+str(mpi.rank), s)
```

`pyunicorn.utils.mpi.`**`abort`**`()`

Abort execution on all MPI nodes immediately.

Can be called by master and slaves.

`pyunicorn.utils.mpi.`**`am_master`** **= True**

(bool) indicates that this MPI node is the master.

`pyunicorn.utils.mpi.`**`am_slave`** **= False**

(bool) indicates that this MPI node is a slave.

`pyunicorn.utils.mpi.`**`assigned`** **= {}**

(dictionary) assigned[id] is the slave assigned to the call with that id.

`pyunicorn.utils.mpi.`**`available`** **= False**

(bool) indicates that slaves are available.

`pyunicorn.utils.mpi.`**`get_next_result`**`()`

Return result of next earlier submitted call whose result has not yet been got.

Can only be called by the master.

If the call is not yet finished, waits for it to finish.

Return type object

Returns return value of call, or None of there are no more calls in the queue.

`pyunicorn.utils.mpi.`**`get_result`**`(`*id*`)`

Return result of earlier submitted call.

Can only be called by the master.

If the call is not yet finished, waits for it to finish. Results should be collected in the same order as calls were submitted. For each slave, the results of calls assigned to that slave must be collected in the same order as those calls were submitted. Can only be called once per call.

Parameters **`id`** (*object*) – id of an earlier submitted call, as provided to or returned by submit_call().

Return type object

Returns return value of call.

`pyunicorn.utils.mpi.`**`info`**`()`

Print processing statistics.

Can only be called by the master.

`pyunicorn.utils.mpi.`**`n_processed`** **= array([0])**

(list of ints) n_processed[rank] is the total number of calls processed by MPI node rank. On slave i, only total_time[i] is available.

`pyunicorn.utils.mpi.`**`n_slaves`** **= 0**

(int) no. of slaves available.

`pyunicorn.utils.mpi.`**`queue`** **= []**
(list) ids of submitted calls

`pyunicorn.utils.mpi.`**`rank`** **= 0**
(int) rank of this MPI node (0 is the master).

`pyunicorn.utils.mpi.`**`results`** **= {}**
(dictionary) if mpi is not available, the result of submit_call(..., id=a) will be cached in results[a] until get_result(a).

`pyunicorn.utils.mpi.`**`run`**(*verbose=False*)
Run in master/slaves mode until master() finishes.

Must be called on all MPI nodes after function master() was defined.

On the master, run() calls master() and returns when master() returns.

On each slave, run() calls slave() if that is defined, or calls serve() otherwise, and returns when slave() returns, or when master() returns on the master, or when master calls terminate().

Parameters **`verbose`** (*bool*) – whether processing information should be printed.

`pyunicorn.utils.mpi.`**`size`** **= 1**
(int) number of MPI nodes (master and slaves).

`pyunicorn.utils.mpi.`**`slave_queue`** **= [[]]**
(list of lists) slave_queue[i] contains the ids of calls assigned to slave i.

`pyunicorn.utils.mpi.`**`start_time`** **= 1460909287.886224**
(float) starting time of this MPI node.

`pyunicorn.utils.mpi.`**`stats`** **= []**
(list of dictionaries) stats[id] contains processing statistics for the last call with this id. Keys:

- "id": id of the call
- "rank": MPI node who processed the call
- "this_time": wall time for processing the call
- "time_over_est": quotient of actual over estimated wall time
- "n_processed": no. of calls processed so far by that slave, including this
- "total_time": total wall time until this call was finished

`pyunicorn.utils.mpi.`**`submit_call`**(*name_to_call*, *args=()*, *kwargs={}*, *module='__main__'*, *time_est=1*, *id=None*, *slave=None*)
Submit a call for parallel execution.

If called by the master and slaves are available, the call is submitted to a slave for asynchronous execution.

If called by a slave or if no slaves are available, the call is instead executed synchronously on this MPI node.

Examples:

1. Provide ids and time estimate explicitly:

```
for n in range(0,10):
    mpi.submit_call("doit", (n,A[n]), id=n, time_est=n**2)

for n in range(0,10):
    result[n] = mpi.get_result(n)
```

2. Use generated ids stored in a list:

```
for n in range(0,10):
    ids.append(mpi.submit_call("doit", (n,A[n])))
```

```
for n in range(0,10):
    results.append(mpi.get_result(ids.pop()))
```

3.Ignore ids altogether:

```
for n in range(0,10):
    mpi.submit_call("doit", (n,A[n]))

for n in range(0,10):
    results.append(mpi.get_next_result())
```

4.Call a module function and use keyword arguments:

```
mpi.submit_call("solve", (), {"a":a, "b":b},
    module="numpy.linalg")
```

5.Call a static class method from a package:

```
mpi.submit_call("Network._get_histogram", (values, n_bins),
    module="pyunicorn")
```

Note that it is module=”pyunicorn” and not module=”pyunicorn.network” here.

Parameters

- **name_to_call** (*str*) – name of callable object (usually a function or static method of a class) as contained in the namespace specified by module.
- **args** (*tuple*) – the positional arguments to provide to the callable object. Tuples of length 1 must be written (arg,). Default: ()
- **kwargs** (*dict*) – the keyword arguments to provide to the callable object. Default: {}
- **module** (*str*) – optional name of the imported module or submodule in whose namespace the callable object is contained. For objects defined on the script level, this is "__main__", for objects defined in an imported package, this is the package name. Must be a key of the dictionary sys.modules (check there after import if in doubt). Default: "__main__"
- **time_est** (*float*) – estimated relative completion time for this call; used to find a suitable slave. Default: 1
- **id** (*object or None*) – unique id for this call. Must be a possible dictionary key. If None, a random id is assigned and returned. Can be re-used after get_result() for this is. Default: None
- **slave** (*int > 0 and < mpi.size, or None*) – optional no. of slave to assign the call to. If None, the call is assigned to the slave with the smallest current total time estimate. Default: None

Return object id of call, to be used in get_result().

pyunicorn.utils.mpi.**terminate**()

Tell all slaves to terminate.

Can only be called by the master.

pyunicorn.utils.mpi.**total_time** = **array([0.])**

(list of floats) total_time[rank] is the total wall time until that node finished its last call. On slave i, only total_time[i] is available.

pyunicorn.utils.mpi.**total_time_est** = **array([inf])**

(numpy array of ints) total_time_est[i] is the current estimate of the total time MPI slave i will work on already submitted calls. On slave i, only total_time_est[i] is available.

5.5.2 utils.navigator

Development

For development, especially if you want to test `pyunicorn` from within the source directory:

```
    $> pip install --user -e .
```

6.1 Test suite

Before committing changes to the code base, please make sure that all tests pass. The test suite is managed by tox (https://testrun.org/tox/) and configured to use system-wide packages when available. Thus to avoid frequent waiting, we recommend you to install the current versions of the following packages:

```
$> pip install tox nose networkx Sphinx
$> pip install pylint pytest pytest-xdist pytest-flakes pytest-pep8
```

The test suite can be run from anywhere in the project tree by issuing:

```
$> tox
```

To expose the defined test environments and target them independently:

```
$> tox -l
$> tox -e py27-units,py27-pylint
```

To test single files:

```
$> tests/test_doctests.py core.network      # doctests
$> nosetests -vs tests/core/TestNetwork.py  # unit tests
$> pylint pyunicorn/core/network.py         # code analysis
$> py.test pyunicorn/core/network.py        # style
```

6.2 Mailing list

Not implemented yet.

Changelog

A summary of major changes made in each release of `pyunicorn`:

0.5.1

- Added reference to pyunicorn description paper published in the journal Chaos.

0.5.0

- Substantial update of `CouplingAnalysis`.
- New methods in `RecurrenceNetwork`: `transitivity_dim_single_scale`, `local_clustering_dim_single_scale`.
- Renamed time-directed measures in `VisibilityGraph`: `left/right` -> `retarded/advanced`.
- Improved documentation and extended publication list.
- Began transition from `Weave` to `Cython`.
- Added unit tests and improved Pylint compliance.
- Set up continuous testing with Travis CI.
- Fixed some minor bugs.

0.4.1

- Removed a whole lot of `get_` s from the API. For example, `Network.get_degree()` is now `Network.degree()`.
- Fixed some minor bugs.

0.4.0

- Restructured package (subpackages: `core`, `climate`, `timeseries`, `funcnet`, `utils`).
- Removed dependencies: `Pysparse`, `PyNio`, `progressbar`.
- Added a module for resistive networks.
- Switched to `tox` for test suite management.
- Ensured PEP8 and PyFlakes compliance.

0.3.2

- Fixed some minor bugs.
- Switched to `Sphinx` documentation system.

0.3.1

- First public release of `pyunicorn`.

CHAPTER 8

Publications

References to peer-reviewed publications, theses and reports describing in detail and applying the methods implemented in the `pyunicorn` package.

8.1 General complex networks

8.1.1 *Review papers*

[Newman2003], *[Boccaletti2006]*, *[Costa2007]*.

8.1.2 *Further network papers*

[Watts1998], *[Newman2001]*, *[Newman2002]*, *[Arenas2003]*, *[Newman2005]*, *[Soffer2005]*, *[Holme2007]*, *[Tsonis2008a]*, *[Ueoka2008]*.

8.2 Spatially embedded networks

[Bartelemy2011].

8.3 Interacting/interdependent networks / networks of networks

8.3.1 *Introduction to structural analysis of interacting networks*

[Donges2011a].

8.4 Node-weighted network measures / node-splitting invariance

8.4.1 *Introduction*

[Heitzig2012].

8.4.2 *Random graph models and network surrogates for interacting networks*

[Schultz2010].

8.4.3 *Analysis of node-weighted interacting networks*

[Wiedermann2011], *[Wiedermann2013]*.

8.5 Climate data analysis (general)

[Bretherton1992].

8.6 Climate networks / Coupled climate networks

8.6.1 *Comparing linear and nonlinear construction of climate networks*

[Donges2009a].

8.6.2 *Studying the dynamical structure of the surface air temperature field*

[Donges2009b], *[Radebach2010]*.

8.6.3 *Introduction to coupled climate networks and applications*

[Schultz2010], *[Donges2011a]*, *[Wiedermann2011]*.

8.6.4 *Review of climate network analysis (in Chinese!)*

[Zou2011].

8.6.5 *Visualization of climate networks*

[Tominski2011].

8.6.6 *Evolving climate networks*

[Radebach2013].

8.6.7 *General*

[Tsonis2004], *[Tsonis2006]*, *[Gozolchiani2008]*, *[Tsonis2008b]*, *[Tsonis2008c]*, *[Yamasaki2008]*, *[Donges2009c]*, *[Yamasaki2009]*.

8.7 Power Grids/Power Networks

8.7.1 *Resistance based networks*

[Schultz2014], *[Schultz2014a]*.

8.8 Time series analysis and synchronization (general)

[Pecora1998], *[Schreiber2000]*, *[Kraskov2004]*, *[Kantz2006]*, *[Thiel2006]*, *[Bergner2008]*, *[Pompe2011]*, *[Runge2012b]*.

8.9 Recurrence networks / quantification analysis / plots

8.9.1 *Review of recurrence plots and RQA*

[Marwan2007].

8.9.2 *Introduction and application of recurrence networks in the context of RQA*

[Marwan2009].

8.9.3 *A thorough introduction to recurrence network analysis*

[Donner2010b].

8.9.4 *Discussion of choosing an appropriate recurrence threshold*

[Donner2010a], *[Zou2010]*.

8.9.5 *Review of various methods for network-based time series analysis*

[Donner2011a].

8.9.6 *Introduction to measures of (fractal) transitivity dimensions*

[Donner2011b].

8.9.7 *Applications of recurrence network analysis to paleoclimate data*

[Donges2011b], *[Donges2011c]*, *[Feldhoff2012]*.

8.9.8 *Theory of recurrence networks*

[Donges2012], *[Zou2012]*.

8.9.9 *Multivariate extensions of recurrence network analysis*

[Feldhoff2012], *[Feldhoff2013]*.

8.9.10 *General*

[Ngamga2007], *[Xu2008]*, *[Schinkel2009]*.

8.10 Visibility graph analysis

8.10.1 *Introduction*

[Lacasa2008].

8.10.2 *Application to geophysical time series*

[Donner2012].

8.10.3 *Tests for time series irreversibility*

[Donges2013].

License

CHAPTER 10

Contact

10.1 Reference

Please acknowledge and cite the use of this software and its authors when results are used in publications or published elsewhere. You can use the following reference:

> J.F. Donges, J. Heitzig, B. Beronov, M. Wiedermann, J. Runge, Q.-Y. Feng, L. Tupikina, V. Stolbova, R.V. Donner, N. Marwan, H.A. Dijkstra, and J. Kurths, Unified functional network and nonlinear time series analysis for complex systems science: The pyunicorn package, Chaos 25, 113101 (2015), doi:10.1063/1.4934554, (http://dx.doi.org/10.1063/1.4934554) Preprint: arxiv.org:1507.01571 [physics.data-an]. (http://arxiv.org/abs/1507.01571)

URL http://www.pik-potsdam.de/members/donges/software

Mail Jonathan Donges, Potsdam Institute for Climate Impact Research, P.O. Box 60 12 03, D-14412 Potsdam, Germany

Authors Written as part of a diploma/PhD thesis in physics by Jonathan F. Donges (donges@pik-potsdam.de) at Humboldt University Berlin and the Potsdam Institute for Climate Impact Research (PIK) and completed at the University of Potsdam, Germany. Substantially extended by Jobst Heitzig (heitzig@pik-potsdam.de).

Contributors

- Jakob Runge (extended `core` and `climate`)
- Alexander Radebach
- Hanna Schultz
- Marc Wiedermann (extended `core` and `climate`)
- Alraune Zech (alrauni@web.de) (extended `timeseries` during an internship at PIK)
- Jan Feldhoff (feldhoff@pik-potsdam.de) (extended `timeseries`)
- Aljoscha Rheinwalt
- Hannes Kutza
- Boyan Beronov (beronov@pik-potsdam.de) (restructured, consolidated and updated codebase during an internship at PIK)
- Paul Schultz (pschultz@pik-potsdam.de), Stefan Schinkel (mail@dreeg.org) (supplied `resistive_network` and corresponding tests)
- Wolfram Barfuss (barfuss@pik-potsdam.de)

Bibliography

[Newman2003] M.E.J. Newman. “The structure and function of complex networks”. In *SIAM Review*, vol. 45 (no. 2), p167-256 (2003) doi:10.1137/S003614450342480 (http://dx.doi.org/10.1137/S003614450342480)

[Boccaletti2006] S. Boccaletti, V. Latora, Y. Moreno, M. Chavez, D.U. Hwang. “Complex networks: structure and dynamics”. In *Physics Reports*, vol. 424 (no. 4-5), p175-308 (2006) doi:10.1016/j.physrep.2005.10.009 (http://dx.doi.org/10.1016/j.physrep.2005.10.009)

[Costa2007] L.D.F. Costa, F.A. Rodrigues, G. Travieso, P.R. Villas Boas. “Characterization of complex networks: A survey of measurements”. In *Advances in Physics*, vol. 56(1), 167-242 (2007) doi:10.1080/00018730601170527 (http://dx.doi.org/10.1080/00018730601170527)

[Watts1998] D.J. Watts and S.H. Strogatz. “Collective dynamics of small-world networks”. In *Nature* vol. 393, 440–442 (1998) doi:10.1038/30918 (http://dx.doi.org/10.1038/30918)

[Newman2001] M.E.J. Newman. “Scientific collaboration networks. II. Shortest paths, weighted networks, and centrality”. In *Physical Review E* vol. 64.1, 016132 (2001) doi:10.1103/PhysRevE.64.016132 (http://dx.doi.org/10.1103/PhysRevE.64.016132)

[Newman2002] M.E.J. Newman. “Assortative mixing in networks”. In *Physical Review Letters*, vol. 89.20, 208701 (2002) doi:10.1103/PhysRevLett.112.068103 (http://dx.doi.org/10.1103/PhysRevLett.112.068103)

[Arenas2003] A. Arenas, A. Cabrales, A. Díaz-Guilera, R. Guimerà, F. Vega-Redondo. “Search and Congestion in Complex Networks”. In “Statistical Mechanics of Complex Networks”, In *Lecture Notes in Physics*, vol. 625, p175-194 (2003) doi:10.1007/978-3-540-44943-0_11 (http://dx.doi.org/10.1007/978-3-540-44943-0_11)

[Newman2005] M.E.J. Newman. “A measure of betweenness centrality based on random walks”. In *Social Networks*, vol 27 (no. 1), p39–54 (2005) doi:10.1016/j.socnet.2004.11.009 (http://dx.doi.org/10.1016/j.socnet.2004.11.009)

[Soffer2005] S.N. Soffer and A. Vázquez “Network clustering coefficient without degree-correlation biases”. In *Physical Review E*, vol. 71, 057101 (2005) doi:10.1103/PhysRevE.71.057101 (http://dx.doi.org/10.1103/PhysRevE.71.057101)

[Holme2007] P. Holme, S.M. Park, B.J. Kim, C.R. Edling. “Korean university life in a network perspective: Dynamics of a large affiliation network”. In *Physica A: Statistical Mechanics and its Applications*, vol. 373, p821-830 (2007) doi:10.1016/j.physa.2006.04.066 (http://dx.doi.org/10.1016/j.physa.2006.04.066)

[Tsonis2008a] A.A. Tsonis, K.L. Swanson, G. Wang. “Estimating the clustering coefficient in scale-free networks on lattices with local spatial correlation structure”. In *Physica A: Statistical Mechanics and its Applications*, vol. 387 (no. 21) p5287-5294 (2008) doi:10.1016/j.physa.2008.05.048 (http://dx.doi.org/10.1016/j.physa.2008.05.048)

[Ueoka2008] Y. Ueoka, T. Suzuki, T. Ikeguchi, Y. Horio. “Efficiency of Statistical Measures to Estimate Network Structure of Chaos Coupled Systems”. Proceedings of NOLTA (2008) http://tsuzuki.ise.ibaraki.ac.jp/MyPaper/Meeting/08NOLTA.pdf

[Bartelemy2011] M. Barthelemy. “Spatial networks”. In *Physics Reports*, vol. 499 (no. 1-3), p1-101 (2011) doi:10.1016/j.physrep.2010.11.002 (http://dx.doi.org/10.1016/j.physrep.2010.11.002)

[Donges2011a] J.F. Donges, H.C.H. Schultz, N. Marwan, Y. Zou, J. Kurths. “Investigating the topology of interacting networks - Theory and application to coupled climate subnetworks”. In *European Physical Journal B: Condensed Matter and Complex Systems*, vol. 84 (no. 4) p635-652 (2011) doi:10.1140/epjb/e2011-10795-8 (http://dx.doi.org/10.1140/epjb/e2011-10795-8)

[Heitzig2012] J. Heitzig, J. F. Donges, Y. Zou, N. Marwan, J. Kurths. “Node-weighted measures for complex networks with spatially embedded, sampled, or differently sized nodes”. In *European Physical Journal B: Condensed Matter and Complex Systems*, vol. 85 p38 (2012) doi:10.1140/epjb/e2011-20678-7 (http://dx.doi.org/10.1140/epjb/e2011-20678-7)

[Schultz2010] H.C.H. Schultz. “Coupled climate networks: Investigating the terrestrial atmosphere’s dynamical structure”. Diploma thesis, Free University, Berlin (2010)

[Wiedermann2011] M. Wiedermann. “Coupled climate network analysis of multidecadal dynamics in the Arctic”. Bachelor’s thesis, Humboldt University, Berlin (2011)

[Wiedermann2013] M. Wiedermann, J.F. Donges, J. Heitzig, J. Kurths. “Node-weighted interacting network measures improve the representation of real-world complex systems”. In *Europhysics Letters*, vol. 102.2, 28007 (2013) doi:10.1209/0295-5075/102/28007 (http://dx.doi.org/10.1209/0295-5075/102/28007)

[Bretherton1992] C.S. Bretherton, C. Smith, J.M. Wallace. “An intercomparison of methods for finding coupled patterns in climate data”. In *Journal of Climate*, vol. 5, p541-560 (1992) doi:10.1175/1520-0442(1992)005<0541:AIOMFF>2.0.CO;2 (http://dx.doi.org/10.1175/1520-0442(1992)005%3C0541%3AAIOMFF%3E2.0.CO%3B2)

[Donges2009a] J.F. Donges, Y. Zou, N. Marwan, J. Kurths. “Complex networks in climate dynamics”. In *European Physical Journal Special Topics*, vol. 174 (no. 1), p157-179 (2009) doi:10.1140/epjst/e2009-01098-2 (http://dx.doi.org/10.1140/epjst/e2009-01098-2)

[Donges2009b] J.F. Donges, Y. Zou, N. Marwan, J. Kurths. “The backbone of the climate network”. In *Europhysics Letters*, vol. 87 (no. 4), 48007 (2009) doi:10.1209/0295-5075/87/48007 (http://dx.doi.org/10.1209/0295-5075/87/48007)

[Radebach2010] A. Radebach. “Evolving climate networks: Investigating the evolution of correlation structure of the Earth’s climate system”. Diploma thesis, Humboldt University, Berlin (2010)

[Zou2011] Y. Zou, J.F. Donges, J. Kurths. “Recent advances in complex climate network analysis”. In *Complex Systems and Complexity Science*, vol. 8 (no. 1), p27-38 (2011)

[Tominski2011] C. Tominski, J.F. Donges, T. Nocke. “Information Visualization in Climate Research”. In *Proceedings of the International Conference on Information Visualisation (IV), London*, p298-305 (2011) doi:10.1109/IV.2011.12 (http://dx.doi.org/10.1109/IV.2011.12)

[Radebach2013] A. Radebach, R.V. Donner, J. Runge, J.F. Donges, J. Kurths. “Disentangling different types of El Nino episodes by evolving climate network analysis”. In *Physical Review E*, vol. 88, 052807 (2013) doi:10.1103/PhysRevE.88.052807 (http://dx.doi.org/10.1103/PhysRevE.88.052807)

[Tsonis2004] A.A. Tsonis and P.J. Roebber. “The architecture of the climate network”. In *Physica A: Statistical Mechanics and its Applications*, vol. 333, p497-504 (2004) doi:10.1016/j.physa.2003.10.045 (http://dx.doi.org/10.1016/j.physa.2003.10.045)

[Tsonis2006] A.A. Tsonis, K.L. Swanson, P.J. Roebber. “What do networks have to do with climate?”. In *Bull. Amer. Meteor. Soc.* vol. 87 p585-595 (2006) doi:10.1175/BAMS-87-5-585 (http://dx.doi.org/10.1175/BAMS-87-5-585)

[Gozolchiani2008] A. Gozolchiani, K. Yamasaki, O. Gazit, S. Havlin. “Pattern of climate network blinking links follows El Niño events”. In *Europhysics Letters*, vol. 83 (no. 2), 28005 (2008) doi:10.1209/0295-5075/83/28005 (http://dx.doi.org/10.1209/0295-5075/83/28005)

[Tsonis2008b] A. A. Tsonis and K. L. Swanson. “Topology and Predictability of El Niño and La Niña Networks”. In *Physical Review Letters* vol 100, 228502 (2008) doi:10.1103/PhysRevLett.100.228502 (http://dx.doi.org/10.1103/PhysRevLett.100.228502)

[Tsonis2008c] A. A. Tsonis, K. L. Swanson, G. Wang. "On the role of atmospheric teleconnections in climate". In *Journal of Climate* vol. 21, p2990-3001 (2008) doi:10.1175/2007JCLI1907.1 (http://dx.doi.org/10.1175/2007JCLI1907.1)

[Yamasaki2008] K. Yamasaki, A. Gozolchiani, S. Havlin. "Climate Networks around the Globe are Significantly Affected by El Niño". In *Physical Review Letters*, vol. 100, 228501 (2008) doi:10.1103/PhysRevLett.100.228501 (http://dx.doi.org/10.1103/PhysRevLett.100.228501)

[Donges2009c] J.F. Donges "Complex networks in the climate system". Diploma thesis, University of Potsdam (2009) Advisor: Prof. Dr. Dr. h.c. Juergen Kurths. URN: `urn:nbn:de:kobv:517-opus-49775`.

[Yamasaki2009] K. Yamasaki, A. Gozolchiani, S. Havlin. "Climate Networks Based on Phase Synchronization Analysis Track El-Niño". In *Progress Of Theoretical Physics Supplement*, vol. 179, p178-188 (2009) doi:10.1143/PTPS.179.178 (http://dx.doi.org/10.1143/PTPS.179.178)

[Schultz2014] P. Schultz "Stability Analysis of Power Grid Networks". *M.Sc. Thesis*, Humboldt-Universität zu Berlin (2014)

[Schultz2014a] P. Schultz, J. Heitzig, J. Kurths A Random Growth Model for Power Grids and Other Spatially Embedded Infrastructure Networks". In *Eur. Phys. J. Special Topics: Resilient Power Grids and Extreme Events* (2014)

[Pecora1998] L.M. Pecora and T.L. Carroll. "Master Stability Functions for Synchronized Coupled Systems". In *Physical Review Letters*, vol. 80, 2109 (1998) doi:10.1103/PhysRevLett.80.2109 (http://dx.doi.org/10.1103/PhysRevLett.80.2109)

[Schreiber2000] T. Schreiber and A. Schmitz. "Surrogate time series". In *Physica D* vol. 142 (no. 3-4), p346-382 (2000) doi:10.1016/S0167-2789(00)00043-9 (http://dx.doi.org/10.1016/S0167-2789(00)00043-9)

[Kraskov2004] A. Kraskov, H. Stögbauer, P. Grassberger. "Estimating mutual information". In *Physical Review E*, vol. 69(6), 066138 (2004) doi:10.1103/PhysRevE.69.066138 (http://dx.doi.org/10.1103/PhysRevE.69.066138)

[Kantz2006] H. Kantz and T. Schreiber. "Nonlinear Time Series Analysis". Cambridge University Press, Cambridge, 2nd edition (2006)

[Thiel2006] M. Thiel, M.C. Romano, J. Kurths, M. Rolfs, R. Kliegl. "Twin surrogates to test for complex synchronization". In *Europhysics Letters*, vol. 75, p535-541 (2006) doi:10.1209/epl/i2006-10147-0 (http://dx.doi.org/10.1209/epl/i2006-10147-0)

[Bergner2008] A. Bergner, R. Meucci, K. Al Naimee, M.C. Romano, M. Thiel, J. Kurths, and F. T. Arecchi. "Continuous wavelet transform in the analysis of burst synchronization in a coupled laser system". In *Physical Review E*, vol. 78, 016211 (2008) doi:10.1103/PhysRevE.78.016211 (http://dx.doi.org/10.1103/PhysRevE.78.016211)

[Pompe2011] B. Pompe, J. Runge. "Momentary information transfer as a coupling measure of time series". In *Physical Review E* vol. 83, 051122 (2011) doi:10.1103/PhysRevE.83.051122 (http://dx.doi.org/10.1103/PhysRevE.83.051122)

[Runge2012b] J. Runge, J. Heitzig, N. Marwan, J. Kurths. "Quantifying causal coupling strength: A lag-specific measure for multivariate time series related to transfer entropy". In *Physical Review E*, vol. 86(6), 1-15 (2012) doi:10.1103/PhysRevE.86.061121 (http://dx.doi.org/10.1103/PhysRevE.86.061121)

[Marwan2007] N. Marwan, M.C. Romano, M. Thiel, J. Kurths. "Recurrence plots for the analysis of complex systems". In *Physics Reports*, vol. 438 (no. 5–6), p237-329 (2007) doi:10.1016/j.physrep.2006.11.001 (http://dx.doi.org/10.1016/j.physrep.2006.11.001)

[Marwan2009] N. Marwan, J.F. Donges, Y. Zou, R.V. Donner, J. Kurths. "Complex network approach for recurrence analysis of time series". In *Physics Letters A*, vol. 373 (no. 46), p4246-4254 (2009) doi:10.1016/j.physleta.2009.09.042 (http://dx.doi.org/10.1016/j.physleta.2009.09.042)

[Donner2010b] R.V. Donner, Y. Zou, J.F. Donges, N. Marwan, J. Kurths. "Recurrence networks – A novel paradigm for nonlinear time series analysis". In *New Journal of Physics*, vol. 12 (no. 3), 033205 (2010) doi:10.1088/1367-2630/12/3/033025 (http://dx.doi.org/10.1088/1367-2630/12/3/033025)

[Donner2010a] R.V. Donner, Y. Zou, J.F. Donges, N. Marwan, J. Kurths. “Ambiguities in recurrence-based complex network representations of time series”. In *Physical Review E*, vol. 81 (no. 1), 015101(R) (2010) doi:10.1103/PhysRevE.81.015101 (http://dx.doi.org/10.1103/PhysRevE.81.015101)

[Zou2010] Y. Zou, R.V. Donner, J.F. Donges, N. Marwan, J. Kurths. “Identifying complex periodic windows in continuous-time dynamical systems using recurrence-based methods”. In *Chaos*, vol. 20 (no. 4), 043130 (2010) doi:10.1063/1.3523304 (http://dx.doi.org/10.1063/1.3523304)

[Donner2011a] R.V. Donner, M. Small, J.F. Donges, N. Marwan, Y. Zou, R. Xiang, J. Kurths. “Recurrence-based time series analysis by means of complex network methods”. In *International Journal of Bifurcation and Chaos*, vol. 21 (no. 4), p1019-1046 (2011) doi:10.1142/S0218127411029021 (http://dx.doi.org/10.1142/S0218127411029021)

[Donner2011b] R.V. Donner, J. Heitzig, J.F. Donges, Y. Zou, J. Kurths. “The geometry of chaotic dynamics – A complex network perspective”. In *European Physical Journal B: Condensed Matter and Complex Systems*, vol. 84 (no. 4), p653-672 (2011) doi:10.1140/epjb/e2011-10899-1 (http://dx.doi.org/10.1140/epjb/e2011-10899-1)

[Donges2011b] J.F. Donges, R.V. Donner, K. Rehfeld, N. Marwan, M.H. Trauth, J. Kurths. “Identification of dynamical transitions in marine palaeoclimate records by recurrence network analysis”. In *Nonlinear Processes in Geophysics*, vol. 18 (no. 5), p545-562 (2011) doi:10.5194/npg-18-545-2011 (http://dx.doi.org/10.5194/npg-18-545-2011)

[Donges2011c] J.F. Donges, R.V. Donner, M.H. Trauth, N. Marwan, H.J. Schellnhuber, J. Kurths. “Nonlinear detection of paleoclimate-variability transitions possibly related to human evolution”. In *Proceedings of the National Academy of Sciences of the United States of America*, vol. 108 (no. 51), p20422-20427 (2011) doi:10.1073/pnas.1117052108 (http://dx.doi.org/10.1073/pnas.1117052108)

[Donges2012] J.F. Donges, J. Heitzig, R.V. Donner, J. Kurths. “Analytical framework for recurrence network analysis of time series”. In *Physical Review E: Statistical, Nonlinear, and Soft Matter Physics*, vol. 85, 046105 (2012) doi:10.1103/PhysRevE.85.046105 (http://dx.doi.org/10.1103/PhysRevE.85.046105)

[Zou2012] Y. Zou, J. Heitzig, R.V. Donner, J.F. Donges, J.D. Farmer, R. Meucci, S. Euzzor, N. Marwan, J. Kurths. “Power-laws in recurrence networks from dynamical systems”. In *Europhysics Letters*, vol. 98, 48001 (2012) doi:10.1209/0295-5075/98/48001 (http://dx.doi.org/10.1209/0295-5075/98/48001)

[Feldhoff2012] J.H. Feldhoff, R.V. Donner, J.F. Donges, N. Marwan, J. Kurths. “Geometric detection of coupling directions by means of inter-system recurrence networks”. In *Physics Letters A*, vol. 376, 3504-3513 (2012), doi:10.1016/j.physleta.2012.10.008 (http://dx.doi.org/10.1016/j.physleta.2012.10.008)

[Feldhoff2013] J.H. Feldhoff, R.V. Donner, J.F. Donges, N. Marwan, J. Kurths. “Geometric signature of complex synchronisation scenarios”. In *Europhysics Letters* vol. 102, 30007 (2013), doi:10.1209/0295-5075/102/30007 (http://dx.doi.org/10.1209/0295-5075/102/30007)

[Ngamga2007] E.J. Ngamga, A. Nandi, R. Ramaswamy, M.C. Romano, M. Thiel, J. Kurths. “Recurrence analysis of strange nonchaotic dynamics”. In *Physical Review E*, vol. 75, 036222 (2007) doi:10.1103/PhysRevE.75.036222 (http://dx.doi.org/10.1103/PhysRevE.75.036222)

[Xu2008] X. Xu, J. Zhang, M. Small. “Superfamily phenomena and motifs of networks induced from time series”. In *Proceedings of the National Academy of Sciences of the United States of America*, vol. 105 (no. 50) p19601-19605 (2008) doi:10.1073/pnas.0806082105 (http://dx.doi.org/10.1073/pnas.0806082105)

[Schinkel2009] S. Schinkel, N. Marwan, O. Dimigen, J. Kurths. “Confidence bounds of recurrence-based complexity measures”. In *Physics Letters A*, vol. 373 (no. 26) p2245–2250 (2009) doi:10.1016/j.physleta.2009.04.045 (http://dx.doi.org/10.1016/j.physleta.2009.04.045)

[Lacasa2008] L. Lacasa, B. Luque, F. Ballesteros, J. Luque, J.C. Nuno. “From time series to complex networks: The visibility graph”. In *Proceedings of the National Academy of Sciences of the United States of America*, vol. 105 (no. 13), p4972-4975 (2008) doi:10.1073/pnas.0709247105 (http://dx.doi.org/10.1073/pnas.0709247105)

[Donner2012] R.V. Donner and J.F. Donges. “Visibility graph analysis of geophysical time series: Potentials and possible pitfalls”. In *Acta Geophysica*, vol. 60 p589-623 (2012) doi:10.2478/s11600-012-0032-x (http://dx.doi.org/10.2478/s11600-012-0032-x)

[Donges2013] J.F. Donges, R.V. Donner, J. Kurths. “Testing time series irreversibility using complex network methods”. In *Europhysics Letters*, vol. 102.1, 10004 (2013) doi:10.1209/0295-5075/102/10004 (http://dx.doi.org/10.1209/0295-5075/102/10004)

c

pyunicorn.climate.climate_data, 92
pyunicorn.climate.climate_network, 97
pyunicorn.climate.coupled_climate_network, 103
pyunicorn.climate.coupled_tsonis, 109
pyunicorn.climate.havlin, 111
pyunicorn.climate.hilbert, 113
pyunicorn.climate.map_plots, 115
pyunicorn.climate.mutual_info, 116
pyunicorn.climate.partial_correlation, 119
pyunicorn.climate.rainfall, 120
pyunicorn.climate.spearman, 122
pyunicorn.climate.tsonis, 124
pyunicorn.core.data, 15
pyunicorn.core.geo_network, 21
pyunicorn.core.grid, 35
pyunicorn.core.interacting_networks, 40
pyunicorn.core.netcdf_dictionary, 55
pyunicorn.core.network, 56
pyunicorn.core.resistive_network, 85

f

pyunicorn.funcnet.coupling_analysis, 155
pyunicorn.funcnet.coupling_analysis_pure_python, 159

t

pyunicorn.timeseries.cross_recurrence_plot, 127
pyunicorn.timeseries.inter_system_recurrence_network, 130
pyunicorn.timeseries.joint_recurrence_network, 133
pyunicorn.timeseries.joint_recurrence_plot, 135
pyunicorn.timeseries.recurrence_network, 137
pyunicorn.timeseries.recurrence_plot, 139
pyunicorn.timeseries.surrogates, 148
pyunicorn.timeseries.visibility_graph, 153

u

pyunicorn.utils.mpi, 163

Symbols

__init__() (pyunicorn.climate.climate_data.ClimateData method), 93
__init__() (pyunicorn.climate.climate_network.ClimateNetwork method), 98
__init__() (pyunicorn.climate.coupled_climate_network.CoupledClimateNetwork method), 104
__init__() (pyunicorn.climate.coupled_tsonis.CoupledTsonisClimateNetwork method), 110
__init__() (pyunicorn.climate.havlin.HavlinClimateNetwork method), 111
__init__() (pyunicorn.climate.hilbert.HilbertClimateNetwork method), 114
__init__() (pyunicorn.climate.map_plots.MapPlots method), 115
__init__() (pyunicorn.climate.mutual_info.MutualInfoClimateNetwork method), 117
__init__() (pyunicorn.climate.partial_correlation.PartialCorrelationClimateNetwork method), 119
__init__() (pyunicorn.climate.rainfall.RainfallClimateNetwork method), 120
__init__() (pyunicorn.climate.spearman.SpearmanClimateNetwork method), 123
__init__() (pyunicorn.climate.tsonis.TsonisClimateNetwork method), 124
__init__() (pyunicorn.core.data.Data method), 16
__init__() (pyunicorn.core.geo_network.GeoNetwork method), 23
__init__() (pyunicorn.core.grid.Grid method), 36
__init__() (pyunicorn.core.interacting_networks.InteractingNetworks method), 42
__init__() (pyunicorn.core.netcdf_dictionary.NetCDFDictionary method), 55
__init__() (pyunicorn.core.network.Network method), 59
__init__() (pyunicorn.core.resistive_network.ResNetwork method), 87
__init__() (pyunicorn.funcnet.coupling_analysis.CouplingAnalysis method), 155
__init__() (pyunicorn.funcnet.coupling_analysis_pure_python.CouplingAnalysisPurePython method), 159
__init__() (pyunicorn.timeseries.cross_recurrence_plot.CrossRecurrencePlot method), 128
__init__() (pyunicorn.timeseries.inter_system_recurrence_network.InterSystemRecurrenceNetwork method), 131
__init__() (pyunicorn.timeseries.joint_recurrence_network.JointRecurren method), 133
__init__() (pyunicorn.timeseries.joint_recurrence_plot.JointRecurrencePl method), 135
__init__() (pyunicorn.timeseries.recurrence_network.RecurrenceNetwork method), 137
__init__() (pyunicorn.timeseries.recurrence_plot.RecurrencePlot method), 140
__init__() (pyunicorn.timeseries.surrogates.Surrogates method), 148
__init__() (pyunicorn.timeseries.visibility_graph.VisibilityGraph method), 153
__len__() (pyunicorn.core.network.Network method), 60
__str__() (pyunicorn.climate.climate_data.ClimateData method), 94
__str__() (pyunicorn.climate.climate_network.ClimateNetwork method), 99
__str__() (pyunicorn.climate.coupled_climate_network.CoupledClimateN method), 105
__str__() (pyunicorn.climate.coupled_tsonis.CoupledTsonisClimateNetw method), 110
__str__() (pyunicorn.climate.havlin.HavlinClimateNetwork method), 112
__str__() (pyunicorn.climate.hilbert.HilbertClimateNetwork method), 114
__str__() (pyunicorn.climate.mutual_info.MutualInfoClimateNetwork method), 117
__str__() (pyunicorn.climate.partial_correlation.PartialCorrelationClimat method), 120
__str__() (pyunicorn.climate.rainfall.RainfallClimateNetwork method), 121
__str__() (pyunicorn.climate.spearman.SpearmanClimateNetwork method), 123
__str__() (pyunicorn.climate.tsonis.TsonisClimateNetwork method), 124
__str__() (pyunicorn.core.data.Data method), 17
__str__() (pyunicorn.core.geo_network.GeoNetwork method), 23
__str__() (pyunicorn.core.grid.Grid method), 36
__str__() (pyunicorn.core.interacting_networks.InteractingNetworks method), 43
__str__() (pyunicorn.core.netcdf_dictionary.NetCDFDictionary method), 56

__str__() (pyunicorn.core.network.Network method), 60
__str__() (pyunicorn.core.resistive_network.ResNetwork method), 87
__str__() (pyunicorn.funcnet.coupling_analysis.CouplingAnalysis method), 155
__str__() (pyunicorn.funcnet.coupling_analysis_pure_python.CouplingAnalysisPurePython method), 160
__str__() (pyunicorn.timeseries.cross_recurrence_plot.CrossRecurrencePlot method), 128
__str__() (pyunicorn.timeseries.inter_system_recurrence_network.InterSystemRecurrenceNetwork method), 131
__str__() (pyunicorn.timeseries.joint_recurrence_network.JointRecurrenceNetwork method), 134
__str__() (pyunicorn.timeseries.joint_recurrence_plot.JointRecurrencePlot method), 136
__str__() (pyunicorn.timeseries.recurrence_network.RecurrenceNetwork method), 138
__str__() (pyunicorn.timeseries.recurrence_plot.RecurrencePlot method), 141
__str__() (pyunicorn.timeseries.surrogates.Surrogates method), 149
__str__() (pyunicorn.timeseries.visibility_graph.VisibilityGraph method), 154
__weakref__ (pyunicorn.climate.map_plots.MapPlots attribute), 115
__weakref__ (pyunicorn.core.data.Data attribute), 17
__weakref__ (pyunicorn.core.grid.Grid attribute), 36
__weakref__ (pyunicorn.core.netcdf_dictionary.NetCDFDictionary attribute), 56
__weakref__ (pyunicorn.core.network.Network attribute), 60
__weakref__ (pyunicorn.core.network.NetworkError attribute), 85
__weakref__ (pyunicorn.funcnet.coupling_analysis.CouplingAnalysis attribute), 155
__weakref__ (pyunicorn.funcnet.coupling_analysis_pure_python.CouplingAnalysisPurePython attribute), 160
__weakref__ (pyunicorn.timeseries.recurrence_plot.RecurrencePlot attribute), 141
__weakref__ (pyunicorn.timeseries.surrogates.Surrogates attribute), 149
_calculate_angular_distance() (pyunicorn.core.grid.Grid method), 36
_calculate_anomaly() (pyunicorn.climate.climate_data.ClimateData method), 94
_calculate_cc() (pyunicorn.funcnet.coupling_analysis_pure_python.CouplingAnalysisPurePython method), 160
_calculate_correlation() (pyunicorn.climate.coupled_tsonis.CoupledTsonisClimateNetwork method), 110
_calculate_correlation() (pyunicorn.climate.partial_correlation.PartialCorrelationClimateNetwork method), 120
_calculate_correlation() (pyunicorn.climate.rainfall.RainfallClimateNetwork method), 121
_calculate_correlation() (pyunicorn.climate.spearman.SpearmanClimateNetwork method), 123
_calculate_correlation() (pyunicorn.climate.tsonis.TsonisClimateNetwork method), 125
_calculate_correlation_strength() (pyunicorn.climate.havlin.HavlinClimateNetwork method), 112
_calculate_general_average_link_distance() (pyunicorn.core.geo_network.GeoNetwork method), 23
_calculate_general_average_path_length() (pyunicorn.core.interacting_networks.InteractingNetworks static method), 43
_calculate_general_closeness() (pyunicorn.core.interacting_networks.InteractingNetworks method), 43
_calculate_general_connectivity_weighted_distance() (pyunicorn.core.geo_network.GeoNetwork method), 24
_calculate_hilbert_correlation() (pyunicorn.climate.hilbert.HilbertClimateNetwork method), 114
_calculate_mi() (pyunicorn.funcnet.coupling_analysis_pure_python.CouplingAnalysisPurePython method), 160
_calculate_mutual_information() (pyunicorn.climate.mutual_info.MutualInfoClimateNetwork method), 117
_calculate_non_local_adjacency() (pyunicorn.climate.climate_network.ClimateNetwork method), 99
_calculate_phase_mean() (pyunicorn.climate.climate_data.ClimateData method), 94
_calculate_threshold_adjacency() (pyunicorn.climate.climate_network.ClimateNetwork method), 99
_cum_histogram() (pyunicorn.core.network.Network static method), 60
_edge_current_flow_betweenness_python() (pyunicorn.core.resistive_network.ResNetwork method), 87
_edge_current_flow_betweenness_weave() (pyunicorn.core.resistive_network.ResNetwork method), 87
_eigenvector_centrality_slow() (pyunicorn.core.network.Network method), 60
_get_nearest_neighbors() (pyunicorn.funcnet.coupling_analysis.CouplingAnalysis static method), 155
_get_netcdf_data() (pyunicorn.core.data.Data class method), 17
_histogram() (pyunicorn.core.network.Network static method), 60
_load_data() (pyunicorn.core.data.Data class method),

17
_par_corr_to_cmi() (pyunicorn.funcnet.coupling_analysis.CouplingAnalysis static method), 155
_quantile_bin_array() (pyunicorn.funcnet.coupling_analysis.CouplingAnalysis static method), 156
_regenerate_network() (pyunicorn.climate.climate_network.ClimateNetwork method), 100
_set_adjacency() (pyunicorn.core.network.Network method), 61
_set_directed() (pyunicorn.climate.hilbert.HilbertClimateNetwork method), 114
_set_max_delay() (pyunicorn.climate.havlin.HavlinClimateNetwork method), 112
_set_node_weights() (pyunicorn.core.network.Network method), 61
_set_winter_only() (pyunicorn.climate.mutual_info.MutualInfoClimateNetwork method), 118
_set_winter_only() (pyunicorn.climate.tsonis.TsonisClimateNetwork method), 125
_vertex_current_flow_betweenness_python() (pyunicorn.core.resistive_network.ResNetwork method), 87
_vertex_current_flow_betweenness_weave() (pyunicorn.core.resistive_network.ResNetwork method), 87
_weave_calculate_mutual_information() (pyunicorn.climate.mutual_info.MutualInfoClimateNetwork method), 118

A

AAFT_surrogates() (pyunicorn.timeseries.surrogates.Surrogates method), 148
abort() (in module pyunicorn.utils.mpi), 164
adaptive neighborhood size
  recurrence network, 138
  recurrence plot, 145
add_dataset() (pyunicorn.climate.map_plots.MapPlots method), 115
add_multiple_datasets() (pyunicorn.climate.map_plots.MapPlots method), 116
add_multiple_datasets_npy() (pyunicorn.climate.map_plots.MapPlots method), 116
adjacency (pyunicorn.core.network.Network attribute), 61
adjacency_1() (pyunicorn.climate.coupled_climate_network.CoupledClimateNetwork method), 105
adjacency_2() (pyunicorn.climate.coupled_climate_network.CoupledClimateNetwor method), 105
admittance_lapacian() (pyunicorn.core.resistive_network.ResNetwork method), 87
admittive_degree() (pyunicorn.core.resistive_network.ResNetwork method), 87
advanced_betweenness() (pyunicorn.timeseries.visibility_graph.VisibilityGraph method), 154
advanced_closeness() (pyunicorn.timeseries.visibility_graph.VisibilityGraph method), 154
advanced_degree() (pyunicorn.timeseries.visibility_graph.VisibilityGraph method), 154
advanced_local_clustering() (pyunicorn.timeseries.visibility_graph.VisibilityGraph method), 154
am_master (in module pyunicorn.utils.mpi), 164
am_slave (in module pyunicorn.utils.mpi), 164
angular_distance() (pyunicorn.core.geo_network.GeoNetwork method), 24
angular_distance() (pyunicorn.core.grid.Grid method), 36
anomaly() (pyunicorn.climate.climate_data.ClimateData method), 94
anomaly_selected_months() (pyunicorn.climate.climate_data.ClimateData method), 95
area_weighted_connectivity() (pyunicorn.core.geo_network.GeoNetwork method), 24
area_weighted_connectivity_cumulative_distribution() (pyunicorn.core.geo_network.GeoNetwork method), 24
area_weighted_connectivity_distribution() (pyunicorn.core.geo_network.GeoNetwork method), 24
arenas_betweenness() (pyunicorn.core.network.Network method), 61
assigned (in module pyunicorn.utils.mpi), 164
assortativity() (pyunicorn.core.network.Network method), 62
available (in module pyunicorn.utils.mpi), 164
average
  diagonal line length, 141
  vertical line length, 141
  white vertical line length, 141
average_diaglength() (pyunicorn.timeseries.recurrence_plot.RecurrencePlot method), 141
average_distance_weighted_path_length() (pyunicorn.core.geo_network.GeoNetwork method), 25
average_effective_resistance() (pyuni-

corn.core.resistive_network.ResNetwork method), 88
average_link_attribute() (pyunicorn.core.network.Network method), 62
average_link_distance() (pyunicorn.core.geo_network.GeoNetwork method), 25
average_neighbor_area_weighted_connectivity() (pyunicorn.core.geo_network.GeoNetwork method), 25
average_neighbors_admittive_degree() (pyunicorn.core.resistive_network.ResNetwork method), 88
average_neighbors_degree() (pyunicorn.core.network.Network method), 62
average_path_length() (pyunicorn.core.network.Network method), 62
average_vertlength() (pyunicorn.timeseries.recurrence_plot.RecurrencePlot method), 141
average_white_vertlength() (pyunicorn.timeseries.recurrence_plot.RecurrencePlot method), 141

B

balance() (pyunicorn.timeseries.cross_recurrence_plot.CrossRecurrencePlot method), 128
BarabasiAlbert() (pyunicorn.core.geo_network.GeoNetwork static method), 21
BarabasiAlbert() (pyunicorn.core.network.Network static method), 57
BarabasiAlbert_igraph() (pyunicorn.core.network.Network static method), 57
betweenness() (pyunicorn.core.network.Network method), 62
bincount_hist() (pyunicorn.funcnet.coupling_analysis.CouplingAnalysis static method), 156
bootstrap_distance_matrix() (pyunicorn.timeseries.recurrence_plot.RecurrencePlot static method), 141
boundaries() (pyunicorn.core.grid.Grid method), 36
boundary() (pyunicorn.core.geo_network.GeoNetwork method), 25
boundary_corrected_closeness() (pyunicorn.timeseries.visibility_graph.VisibilityGraph method), 154
boundary_corrected_degree() (pyunicorn.timeseries.visibility_graph.VisibilityGraph method), 154

C

cache (pyunicorn.core.network.Network attribute), 63
cache_helper() (in module pyunicorn.core.network), 85
cached_const() (in module pyunicorn.core.network), 85
cached_var() (in module pyunicorn.core.network), 85
calculate_corr() (pyunicorn.climate.rainfall.RainfallClimateNetwork method), 121
calculate_rainfall() (pyunicorn.climate.rainfall.RainfallClimateNetwork static method), 122
calculate_similarity_measure() (pyunicorn.climate.coupled_tsonis.CoupledTsonisClimateNetwork method), 111
calculate_similarity_measure() (pyunicorn.climate.mutual_info.MutualInfoClimateNetwork method), 118
calculate_similarity_measure() (pyunicorn.climate.tsonis.TsonisClimateNetwork method), 125
calculate_top_events() (pyunicorn.climate.rainfall.RainfallClimateNetwork static method), 122
clear_cache() (pyunicorn.climate.climate_data.ClimateData method), 95
clear_cache() (pyunicorn.climate.climate_network.ClimateNetwork method), 100
clear_cache() (pyunicorn.climate.havlin.HavlinClimateNetwork method), 112
clear_cache() (pyunicorn.climate.hilbert.HilbertClimateNetwork method), 115
clear_cache() (pyunicorn.core.data.Data method), 18
clear_cache() (pyunicorn.core.geo_network.GeoNetwork method), 26
clear_cache() (pyunicorn.core.grid.Grid method), 37
clear_cache() (pyunicorn.core.network.Network method), 63
clear_cache() (pyunicorn.timeseries.inter_system_recurrence_network.InterSystemR method), 131
clear_cache() (pyunicorn.timeseries.joint_recurrence_network.JointRecurrenceNetw method), 134
clear_cache() (pyunicorn.timeseries.recurrence_network.RecurrenceNetwork method), 138
clear_cache() (pyunicorn.timeseries.recurrence_plot.RecurrencePlot method), 142
clear_cache() (pyunicorn.timeseries.surrogates.Surrogates method), 149
clear_link_attribute() (pyunicorn.core.network.Network method), 63
clear_nsi_cache() (pyunicorn.core.network.Network method), 63
clear_paths_cache() (pyunicorn.core.network.Network method), 63

ClimateData (class in pyunicorn.climate.climate_data), 92
ClimateNetwork (class in pyunicorn.climate.climate_network), 97
closeness() (pyunicorn.core.network.Network method), 63
coherence() (pyunicorn.climate.hilbert.HilbertClimateNetwork method), 115
ConfigurationModel() (pyunicorn.core.geo_network.GeoNetwork static method), 21
ConfigurationModel() (pyunicorn.core.network.Network static method), 57
connectivity_weighted_distance() (pyunicorn.core.geo_network.GeoNetwork method), 26
convert_lon_coordinates() (pyunicorn.core.grid.Grid method), 37
coord_sequence_from_rect_grid() (pyunicorn.core.grid.Grid static method), 37
copy() (pyunicorn.core.network.Network method), 63
coreness() (pyunicorn.core.network.Network method), 63
correlated_noise_surrogates() (pyunicorn.timeseries.surrogates.Surrogates method), 149
correlatedNoiseSurrogates() (pyunicorn.funcnet.coupling_analysis_pure_python.CouplingAnalysisPurePython method), 160
correlation() (pyunicorn.climate.coupled_tsonis.CoupledTsonisClimateNetwork method), 111
correlation() (pyunicorn.climate.tsonis.TsonisClimateNetwork method), 125
correlation_distance() (pyunicorn.climate.climate_network.ClimateNetwork method), 100
correlation_distance_weighted_closeness() (pyunicorn.climate.climate_network.ClimateNetwork method), 100
correlation_lag() (pyunicorn.climate.havlin.HavlinClimateNetwork method), 112
correlation_lag_weighted_average_path_length() (pyunicorn.climate.havlin.HavlinClimateNetwork method), 112
correlation_lag_weighted_closeness() (pyunicorn.climate.havlin.HavlinClimateNetwork method), 113
correlation_strength() (pyunicorn.climate.havlin.HavlinClimateNetwork method), 113
correlation_strength_weighted_average_path_length() (pyunicorn.climate.havlin.HavlinClimateNetwork method), 113
correlation_strength_weighted_closeness() (pyunicorn.climate.havlin.HavlinClimateNetwork method), 113
correlation_weighted_average_path_length() (pyunicorn.climate.tsonis.TsonisClimateNetwork method), 126
correlation_weighted_closeness() (pyunicorn.climate.tsonis.TsonisClimateNetwork method), 126
cos_lat() (pyunicorn.core.grid.Grid method), 37
cos_lon() (pyunicorn.core.grid.Grid method), 37
cos_window() (pyunicorn.core.data.Data static method), 18
CoupledClimateNetwork (class in pyunicorn.climate.coupled_climate_network), 103
CoupledTsonisClimateNetwork (class in pyunicorn.climate.coupled_tsonis), 109
CouplingAnalysis (class in pyunicorn.funcnet.coupling_analysis), 155
CouplingAnalysisPurePython (class in pyunicorn.funcnet.coupling_analysis_pure_python), 159
CR (pyunicorn.timeseries.cross_recurrence_plot.CrossRecurrencePlot attribute), 128
create_plogp() (pyunicorn.funcnet.coupling_analysis.CouplingAnalysis static method), 156
cross recurrence plot, 127
    RQA, 127
cross_adjacency() (pyunicorn.core.interacting_networks.InteractingNetworks method), 43
cross_adjacency_sparse() (pyunicorn.core.interacting_networks.InteractingNetworks method), 44
cross_average_path_length() (pyunicorn.climate.coupled_climate_network.CoupledClimateNetwor method), 105
cross_average_path_length() (pyunicorn.core.interacting_networks.InteractingNetworks method), 44
cross_betweenness() (pyunicorn.climate.coupled_climate_network.CoupledClimateNetwor method), 105
cross_betweenness() (pyunicorn.core.interacting_networks.InteractingNetworks method), 44
cross_closeness() (pyunicorn.climate.coupled_climate_network.CoupledClimateNetwor method), 105
cross_closeness() (pyunicorn.core.interacting_networks.InteractingNetworks method), 45
cross_correlation() (pyunicorn.funcnet.coupling_analysis.CouplingAnalysis method), 156
cross_correlation() (pyunicorn.funcnet.coupling_analysis_pure_python.CouplingAnalysis method), 160
cross_degree() (pyuni-

corn.climate.coupled_climate_network.CoupledClimateNetwork method), 105
cross_degree() (pyunicorn.core.interacting_networks.InteractingNetworks method), 45
cross_global_clustering() (pyunicorn.climate.coupled_climate_network.CoupledClimateNetwork method), 105
cross_global_clustering() (pyunicorn.core.interacting_networks.InteractingNetworks method), 45
cross_global_clustering_sparse() (pyunicorn.core.interacting_networks.InteractingNetworks method), 46
cross_global_clustering_xy() (pyunicorn.timeseries.inter_system_recurrence_network.InterSystemRecurrenceNetwork method), 131
cross_global_clustering_yx() (pyunicorn.timeseries.inter_system_recurrence_network.InterSystemRecurrenceNetwork method), 132
cross_layer_adjacency() (pyunicorn.climate.coupled_climate_network.CoupledClimateNetwork method), 106
cross_link_density() (pyunicorn.climate.coupled_climate_network.CoupledClimateNetwork method), 106
cross_link_density() (pyunicorn.core.interacting_networks.InteractingNetworks method), 46
cross_local_clustering() (pyunicorn.climate.coupled_climate_network.CoupledClimateNetwork method), 106
cross_local_clustering() (pyunicorn.core.interacting_networks.InteractingNetworks method), 46
cross_local_clustering_sparse() (pyunicorn.core.interacting_networks.InteractingNetworks method), 47
cross_path_lengths() (pyunicorn.climate.coupled_climate_network.CoupledClimateNetwork method), 106
cross_path_lengths() (pyunicorn.core.interacting_networks.InteractingNetworks method), 47
cross_recurrence_rate() (pyunicorn.timeseries.cross_recurrence_plot.CrossRecurrencePlot method), 128
cross_recurrence_rate() (pyunicorn.timeseries.inter_system_recurrence_network.InterSystemRecurrenceNetwork method), 132
cross_similarity_measure() (pyunicorn.climate.coupled_climate_network.CoupledClimateNetwork method), 106
cross_transitivity() (pyunicorn.climate.coupled_climate_network.CoupledClimateNetwork method), 106
cross_transitivity() (pyunicorn.core.interacting_networks.InteractingNetworks method), 48
cross_transitivity_sparse() (pyunicorn.core.interacting_networks.InteractingNetworks method), 48
cross_transitivity_xy() (pyunicorn.timeseries.inter_system_recurrence_network.InterSystemR method), 132
cross_transitivity_yx() (pyunicorn.timeseries.inter_system_recurrence_network.InterSystemR method), 132
CrossRecurrencePlot (class in pyunicorn.timeseries.cross_recurrence_plot), 127

D

Data (class in pyunicorn.core.data), 15
data (pyunicorn.climate.havlin.HavlinClimateNetwork attribute), 113
data (pyunicorn.climate.hilbert.HilbertClimateNetwork attribute), 115
data (pyunicorn.climate.mutual_info.MutualInfoClimateNetwork attribute), 118
data (pyunicorn.climate.rainfall.RainfallClimateNetwork attribute), 122
data (pyunicorn.climate.tsonis.TsonisClimateNetwork attribute), 126
degree() (pyunicorn.core.network.Network method), 63
degree_cdf() (pyunicorn.core.network.Network method), 63
degree_distribution() (pyunicorn.core.network.Network method), 64
del_link_attribute() (pyunicorn.core.network.Network method), 64
del_node_attribute() (pyunicorn.core.network.Network method), 64
delay embedding, 149
determinism
    RQA, 142
determinism() (pyunicorn.timeseries.recurrence_plot.RecurrencePlot method), 142
diag_entropy() (pyunicorn.timeseries.recurrence_plot.RecurrencePlot method), 142
diagline_dist() (pyunicorn.timeseries.recurrence_plot.RecurrencePlot method), 142
diagonal
    line entropy, 142
    line length, average, 141
    line length, frequency distribution, 142
    line length, maximum, 144
    lines, 142
diameter() (pyunicorn.core.network.Network method), 64
diameter_effective_resistance() (pyunicorn.core.resistive_network.ResNetwork method), 88

directed (pyunicorn.core.network.Network attribute), 64
distance_based_measures() (pyunicorn.core.network.Network method), 64
distance_matrix() (pyunicorn.timeseries.cross_recurrence_plot.CrossRecurrencePlot method), 128
distance_matrix() (pyunicorn.timeseries.recurrence_plot.RecurrencePlot method), 142
distance_weighted_closeness() (pyunicorn.core.geo_network.GeoNetwork method), 26
do_nsi_clustering() (pyunicorn.core.network.Network method), 65
do_nsi_hamming_clustering() (pyunicorn.core.network.Network method), 65
do_nsi_pca_clustering() (pyunicorn.core.network.Network method), 66

E

edge_betweenness() (pyunicorn.core.network.Network method), 66
edge_current_flow_betweenness() (pyunicorn.core.resistive_network.ResNetwork method), 88
edge_list() (pyunicorn.core.network.Network method), 66
effective_resistance() (pyunicorn.core.resistive_network.ResNetwork method), 89
effective_resistance_closeness_centrality() (pyunicorn.core.resistive_network.ResNetwork method), 89
eigenvector_centrality() (pyunicorn.core.network.Network method), 67
embed_time_series() (pyunicorn.timeseries.recurrence_plot.RecurrencePlot static method), 142
embed_time_series_array() (pyunicorn.timeseries.surrogates.Surrogates method), 149
embedding (pyunicorn.timeseries.recurrence_plot.RecurrencePlot attribute), 143
ErdosRenyi() (pyunicorn.core.geo_network.GeoNetwork static method), 22
ErdosRenyi() (pyunicorn.core.network.Network static method), 57
euclidean_distance() (pyunicorn.core.grid.Grid method), 38
euclidean_distance_matrix() (pyunicorn.timeseries.cross_recurrence_plot.CrossRecurrencePlot method), 129
euclidean_distance_matrix() (pyunicorn.timeseries.recurrence_plot.RecurrencePlot method), 143
eval_fast_code() (pyunicorn.timeseries.surrogates.Surrogates static method), 150
eval_weave_calculate_mutual_information() (pyunicorn.climate.mutual_info.MutualInfoClimateNetwork method), 118

F

frequency distribution
    diagonal line length, 142
    vertical line length, 148
    white vertical line length, 148
from_file() (pyunicorn.core.netcdf_dictionary.NetCDFDictionary static method), 56
FromIGraph() (pyunicorn.core.network.Network static method), 58

G

generate_map_plots() (pyunicorn.climate.map_plots.MapPlots method), 116
generate_multiple_map_plots() (pyunicorn.climate.map_plots.MapPlots method), 116
generate_multiple_map_plots_npy() (pyunicorn.climate.map_plots.MapPlots method), 116
geographical_cumulative_distribution() (pyunicorn.core.geo_network.GeoNetwork method), 26
geographical_distribution() (pyunicorn.core.geo_network.GeoNetwork method), 27
geometric_distance_distribution() (pyunicorn.core.grid.Grid method), 38
GeoNetwork (class in pyunicorn.core.geo_network), 21
get_admittance() (pyunicorn.core.resistive_network.ResNetwork method), 89
get_max_delay() (pyunicorn.climate.havlin.HavlinClimateNetwork method), 113
get_next_result() (in module pyunicorn.utils.mpi), 164
get_R() (pyunicorn.core.resistive_network.ResNetwork method), 89
get_result() (in module pyunicorn.utils.mpi), 164
global_admittive_clustering() (pyunicorn.core.resistive_network.ResNetwork method), 90
global_clustering() (pyunicorn.core.network.Network method), 67
global_efficiency() (pyunicorn.core.network.Network method), 67
graph (pyunicorn.core.network.Network attribute), 67
Grid (class in pyunicorn.core.grid), 35
grid (pyunicorn.climate.map_plots.MapPlots attribute), 116
grid (pyunicorn.core.data.Data attribute), 18

grid (pyunicorn.core.geo_network.GeoNetwork attribute), 27
grid() (pyunicorn.core.grid.Grid method), 38
grid_1 (pyunicorn.climate.coupled_climate_network.CoupledClimateNetwork attribute), 107
grid_2 (pyunicorn.climate.coupled_climate_network.CoupledClimateNetwork attribute), 107
grid_size() (pyunicorn.core.grid.Grid method), 38
GrowPreferentially() (pyunicorn.core.network.Network static method), 58
GrowPreferentially_old() (pyunicorn.core.network.Network static method), 58
GrowWeights() (pyunicorn.core.network.Network static method), 58

H

hamming_distance_from() (pyunicorn.core.network.Network method), 67
HavlinClimateNetwork (class in pyunicorn.climate.havlin), 111
higher_order_transitivity() (pyunicorn.core.network.Network method), 67
HilbertClimateNetwork (class in pyunicorn.climate.hilbert), 113

I

inarea_weighted_connectivity() (pyunicorn.core.geo_network.GeoNetwork method), 27
inarea_weighted_connectivity_cumulative_distribution() (pyunicorn.core.geo_network.GeoNetwork method), 27
inarea_weighted_connectivity_distribution() (pyunicorn.core.geo_network.GeoNetwork method), 28
inaverage_link_distance() (pyunicorn.core.geo_network.GeoNetwork method), 28
inconnectivity_weighted_distance() (pyunicorn.core.geo_network.GeoNetwork method), 28
indegree() (pyunicorn.core.network.Network method), 68
indegree_cdf() (pyunicorn.core.network.Network method), 68
indegree_distribution() (pyunicorn.core.network.Network method), 68
indices_selected_months() (pyunicorn.climate.climate_data.ClimateData method), 95
indices_selected_phases() (pyunicorn.climate.climate_data.ClimateData method), 95
info() (in module pyunicorn.utils.mpi), 164
information_transfer() (pyunicorn.funcnet.coupling_analysis.CouplingAnalysis method), 157
inter_system_recurrence_matrix() (pyunicorn.timeseries.inter_system_recurrence_network.InterSystemR method), 132
InteractingNetworks (class in pyunicorn.core.interacting_networks), 40
internal_adjacency() (pyunicorn.core.interacting_networks.InteractingNetworks method), 48
internal_average_path_length() (pyunicorn.climate.coupled_climate_network.CoupledClimateNetwor method), 107
internal_average_path_length() (pyunicorn.core.interacting_networks.InteractingNetworks method), 49
internal_betweenness() (pyunicorn.core.interacting_networks.InteractingNetworks method), 49
internal_betweenness_1() (pyunicorn.climate.coupled_climate_network.CoupledClimateNetwor method), 107
internal_betweenness_2() (pyunicorn.climate.coupled_climate_network.CoupledClimateNetwor method), 107
internal_closeness() (pyunicorn.climate.coupled_climate_network.CoupledClimateNetwor method), 107
internal_closeness() (pyunicorn.core.interacting_networks.InteractingNetworks method), 49
internal_degree() (pyunicorn.climate.coupled_climate_network.CoupledClimateNetwor method), 108
internal_degree() (pyunicorn.core.interacting_networks.InteractingNetworks method), 50
internal_global_clustering() (pyunicorn.climate.coupled_climate_network.CoupledClimateNetwor method), 108
internal_global_clustering() (pyunicorn.core.interacting_networks.InteractingNetworks method), 50
internal_link_density() (pyunicorn.climate.coupled_climate_network.CoupledClimateNetwor method), 108
internal_link_density() (pyunicorn.core.interacting_networks.InteractingNetworks method), 50
internal_path_lengths() (pyunicorn.core.interacting_networks.InteractingNetworks method), 51
internal_recurrence_rates() (pyunicorn.timeseries.inter_system_recurrence_network.InterSystemR method), 132
interregional_betweenness() (pyunicorn.core.network.Network method), 68
InterSystemRecurrenceNetwork (class in pyunicorn.timeseries.inter_system_recurrence_network),

130
intotal_link_distance() (pyunicorn.core.geo_network.GeoNetwork method), 28
inv_correlation_distance() (pyunicorn.climate.climate_network.ClimateNetwork method), 101

J

joint recurrence plot, 135
    RQA, 135
JointRecurrenceNetwork (class in pyunicorn.timeseries.joint_recurrence_network), 133
JointRecurrencePlot (class in pyunicorn.timeseries.joint_recurrence_plot), 135
JR (pyunicorn.timeseries.joint_recurrence_plot.JointRecurrencePlot attribute), 135

L

lag (pyunicorn.timeseries.joint_recurrence_plot.JointRecurrencePlot attribute), 136
laminarity, 143
laminarity() (pyunicorn.timeseries.recurrence_plot.RecurrencePlot method), 143
laplacian() (pyunicorn.core.network.Network method), 69
lat_sequence() (pyunicorn.core.grid.Grid method), 38
legendre_coordinates() (pyunicorn.timeseries.recurrence_plot.RecurrencePlot static method), 143
line entropy
    diagonal, 142
    vertical, 147
    white vertical, 148
line length
    average diagonal, 141
    average vertical, 141
    average white vertical, 141
    frequency distribution diagonal, 142
    frequency distribution vertical, 148
    frequency distribution white vertical, 148
    maximum diagonal, 144
    maximum vertical, 144
    maximum white vertical, 144
lines
    diagonal, 142
    vertical, 148
    white vertical, 148
link_attribute() (pyunicorn.core.network.Network method), 69
link_betweenness() (pyunicorn.core.network.Network method), 69
link_density (pyunicorn.core.network.Network attribute), 69
link_density_function() (pyunicorn.climate.climate_network.ClimateNetwork method), 101
link_distance_distribution() (pyunicorn.core.geo_network.GeoNetwork method), 28
Load() (pyunicorn.climate.climate_data.ClimateData class method), 93
Load() (pyunicorn.climate.climate_network.ClimateNetwork static method), 97
Load() (pyunicorn.core.data.Data class method), 15
Load() (pyunicorn.core.geo_network.GeoNetwork static method), 22
Load() (pyunicorn.core.grid.Grid static method), 35
Load() (pyunicorn.core.network.Network static method), 58
LoadTXT() (pyunicorn.core.grid.Grid static method), 35
local_admittive_clustering() (pyunicorn.core.resistive_network.ResNetwork method), 90
local_cliquishness() (pyunicorn.core.network.Network method), 69
local_clustering() (pyunicorn.core.network.Network method), 70
local_clustering_dim_single_scale() (pyunicorn.timeseries.recurrence_network.RecurrenceNetwork method), 138
local_correlation_distance_weighted_vulnerability() (pyunicorn.climate.climate_network.ClimateNetwork method), 101
local_correlation_lag_weighted_vulnerability() (pyunicorn.climate.havlin.HavlinClimateNetwork method), 113
local_correlation_strength_weighted_vulnerability() (pyunicorn.climate.havlin.HavlinClimateNetwork method), 113
local_correlation_weighted_vulnerability() (pyunicorn.climate.tsonis.TsonisClimateNetwork method), 126
local_distance_weighted_vulnerability() (pyunicorn.core.geo_network.GeoNetwork method), 29
local_geographical_clustering() (pyunicorn.core.geo_network.GeoNetwork method), 29
local_mutual_information_weighted_vulnerability() (pyunicorn.climate.mutual_info.MutualInfoClimateNetwork method), 118
local_tsonis_clustering() (pyunicorn.core.geo_network.GeoNetwork method), 29
local_vulnerability() (pyunicorn.core.network.Network method), 70
lon_sequence() (pyunicorn.core.grid.Grid method), 39

M

M (pyunicorn.timeseries.cross_recurrence_plot.CrossRecurrencePlot attribute), 128

manhattan_distance_matrix() (pyunicorn.timeseries.cross_recurrence_plot.CrossRecurrencePlot method), 129
manhattan_distance_matrix() (pyunicorn.timeseries.recurrence_plot.RecurrencePlot method), 143
map_data (pyunicorn.climate.map_plots.MapPlots attribute), 116
map_mult_data (pyunicorn.climate.map_plots.MapPlots attribute), 116
MapPlots (class in pyunicorn.climate.map_plots), 115
matching_index() (pyunicorn.core.network.Network method), 70
max_diaglength() (pyunicorn.timeseries.recurrence_plot.RecurrencePlot method), 144
max_link_distance() (pyunicorn.core.geo_network.GeoNetwork method), 29
max_neighbor_area_weighted_connectivity() (pyunicorn.core.geo_network.GeoNetwork method), 30
max_neighbors_degree() (pyunicorn.core.network.Network method), 71
max_vertlength() (pyunicorn.timeseries.recurrence_plot.RecurrencePlot method), 144
max_white_vertlength() (pyunicorn.timeseries.recurrence_plot.RecurrencePlot method), 144
maximum
    diagonal line length, 144
    vertical line length, 144
    white vertical line length, 144
mean recurrence time, 141
mean_node_weight (pyunicorn.core.network.Network attribute), 71
mean_recurrence_time() (pyunicorn.timeseries.recurrence_plot.RecurrencePlot method), 144
metric (pyunicorn.timeseries.inter_system_recurrence_network.InterSystemRecurrenceNetwork attribute), 132
metric (pyunicorn.timeseries.recurrence_plot.RecurrencePlot attribute), 144
mi_file (pyunicorn.climate.mutual_info.MutualInfoClimateNetwork attribute), 118
missing_values (pyunicorn.timeseries.recurrence_plot.RecurrencePlot attribute), 144
missing_values (pyunicorn.timeseries.visibility_graph.VisibilityGraph attribute), 154
msf_synchronizability() (pyunicorn.core.network.Network method), 71
mutual_information() (pyunicorn.climate.mutual_info.MutualInfoClimateNetwork method), 118
mutual_information() (pyunicorn.funcnet.coupling_analysis.CouplingAnalysis method), 157
mutual_information() (pyunicorn.funcnet.coupling_analysis_pure_python.CouplingAnalysis method), 161
mutual_information_edges() (pyunicorn.funcnet.coupling_analysis_pure_python.CouplingAnalysis method), 161
mutual_information_weighted_average_path_length() (pyunicorn.climate.mutual_info.MutualInfoClimateNetwork method), 119
mutual_information_weighted_closeness() (pyunicorn.climate.mutual_info.MutualInfoClimateNetwork method), 119
MutualInfoClimateNetwork (class in pyunicorn.climate.mutual_info), 116

N

N (pyunicorn.climate.coupled_climate_network.CoupledClimateNetwork attribute), 104
N (pyunicorn.core.grid.Grid attribute), 35
N (pyunicorn.core.network.Network attribute), 59
N (pyunicorn.timeseries.cross_recurrence_plot.CrossRecurrencePlot attribute), 128
N (pyunicorn.timeseries.inter_system_recurrence_network.InterSystemRe attribute), 131
N (pyunicorn.timeseries.joint_recurrence_plot.JointRecurrencePlot attribute), 135
N (pyunicorn.timeseries.recurrence_plot.RecurrencePlot attribute), 140
N_1 (pyunicorn.climate.coupled_climate_network.CoupledClimateNetwo attribute), 104
N_2 (pyunicorn.climate.coupled_climate_network.CoupledClimateNetwo attribute), 104
n_grid_points (pyunicorn.core.grid.Grid attribute), 39
n_links (pyunicorn.core.network.Network attribute), 71
n_processed (in module pyunicorn.utils.mpi), 164
n_slaves (in module pyunicorn.utils.mpi), 164
N_x (pyunicorn.timeseries.inter_system_recurrence_network.InterSystem attribute), 131
N_y (pyunicorn.timeseries.inter_system_recurrence_network.InterSystem attribute), 131
NetCDFDictionary (class in pyunicorn.core.netcdf_dictionary), 55
Network (class in pyunicorn.core.network), 56
network_1() (pyunicorn.climate.coupled_climate_network.CoupledClima method), 108
network_2() (pyunicorn.climate.coupled_climate_network.CoupledClima method), 108
NetworkError, 85
newman_betweenness() (pyunicorn.core.network.Network method), 71
next_power_2() (pyunicorn.core.data.Data static method), 18
node_attribute() (pyunicorn.core.network.Network method), 71

node_coordinates() (pyunicorn.core.grid.Grid method), 39
node_number() (pyunicorn.core.grid.Grid method), 39
node_weights (pyunicorn.core.network.Network attribute), 71
nodes_1 (pyunicorn.climate.coupled_climate_network.CoupledClimateNetwork attribute), 108
nodes_2 (pyunicorn.climate.coupled_climate_network.CoupledClimateNetwork attribute), 108
non_local() (pyunicorn.climate.climate_network.ClimateNetwork method), 101
normalize
 time series, 144
 time series array, 18, 150
normalize_time_series() (pyunicorn.timeseries.recurrence_plot.RecurrencePlot static method), 144
normalize_time_series_array() (pyunicorn.core.data.Data static method), 18
normalize_time_series_array() (pyunicorn.timeseries.surrogates.Surrogates static method), 150
nsi_arenas_betweenness() (pyunicorn.core.network.Network method), 72
nsi_average_neighbors_degree() (pyunicorn.core.network.Network method), 72
nsi_average_path_length() (pyunicorn.core.network.Network method), 73
nsi_betweenness() (pyunicorn.core.network.Network method), 73
nsi_closeness() (pyunicorn.core.network.Network method), 74
nsi_connected_hamming_cluster_tree() (pyunicorn.core.geo_network.GeoNetwork method), 30
nsi_cross_average_path_length() (pyunicorn.core.interacting_networks.InteractingNetworks method), 51
nsi_cross_betweenness() (pyunicorn.core.interacting_networks.InteractingNetworks method), 51
nsi_cross_closeness_centrality() (pyunicorn.core.interacting_networks.InteractingNetworks method), 52
nsi_cross_degree() (pyunicorn.core.interacting_networks.InteractingNetworks method), 52
nsi_cross_edge_density() (pyunicorn.core.interacting_networks.InteractingNetworks method), 52
nsi_cross_global_clustering() (pyunicorn.core.interacting_networks.InteractingNetworks method), 53
nsi_cross_local_clustering() (pyunicorn.core.interacting_networks.InteractingNetworks method), 53
nsi_cross_mean_degree() (pyunicorn.core.interacting_networks.InteractingNetworks method), 53
nsi_cross_transitivity() (pyunicorn.core.interacting_networks.InteractingNetworks method), 53
nsi_degree() (pyunicorn.core.network.Network method), 74
nsi_degree_cumulative_histogram() (pyunicorn.core.network.Network method), 75
nsi_degree_histogram() (pyunicorn.core.network.Network method), 75
nsi_degree_uncorr() (pyunicorn.core.network.Network method), 75
nsi_eigenvector_centrality() (pyunicorn.core.network.Network method), 75
nsi_exponential_closeness() (pyunicorn.core.network.Network method), 76
nsi_global_clustering() (pyunicorn.core.network.Network method), 76
nsi_global_efficiency() (pyunicorn.core.network.Network method), 76
nsi_harmonic_closeness() (pyunicorn.core.network.Network method), 76
nsi_internal_closeness_centrality() (pyunicorn.core.interacting_networks.InteractingNetworks method), 54
nsi_internal_degree() (pyunicorn.core.interacting_networks.InteractingNetworks method), 54
nsi_internal_local_clustering() (pyunicorn.core.interacting_networks.InteractingNetworks method), 54
nsi_interregional_betweenness() (pyunicorn.core.network.Network method), 77
nsi_laplacian() (pyunicorn.core.network.Network method), 77
nsi_local_clustering() (pyunicorn.core.network.Network method), 77
nsi_local_clustering_uncorr() (pyunicorn.core.network.Network method), 78
nsi_local_soffer_clustering() (pyunicorn.core.network.Network method), 78
nsi_max_neighbors_degree() (pyunicorn.core.network.Network method), 78
nsi_newman_betweenness() (pyunicorn.core.network.Network method), 79
nsi_spreading() (pyunicorn.core.network.Network method), 80
nsi_transitivity() (pyunicorn.core.network.Network method), 80
nsi_twinness() (pyunicorn.core.network.Network method), 80
number_cross_layer_links() (pyunicorn.climate.coupled_climate_network.CoupledClimateNetwor method), 108
number_cross_links() (pyunicorn.core.interacting_networks.InteractingNetworks method), 54
number_internal_links() (pyuni-

corn.climate.coupled_climate_network.CoupledClimateNetwork method), 109
number_internal_links() (pyunicorn.core.interacting_networks.InteractingNetworks method), 55
nz_coords() (in module pyunicorn.core.network), 85

O

observable() (pyunicorn.core.data.Data method), 19
observable_long_name (pyunicorn.core.data.Data attribute), 19
observable_name (pyunicorn.core.data.Data attribute), 19
original_data (pyunicorn.timeseries.surrogates.Surrogates attribute), 150
original_distribution() (pyunicorn.timeseries.surrogates.Surrogates method), 150
outarea_weighted_connectivity() (pyunicorn.core.geo_network.GeoNetwork method), 30
outarea_weighted_connectivity_cumulative_distribution() (pyunicorn.core.geo_network.GeoNetwork method), 30
outarea_weighted_connectivity_distribution() (pyunicorn.core.geo_network.GeoNetwork method), 31
outaverage_link_distance() (pyunicorn.core.geo_network.GeoNetwork method), 31
outconnectivity_weighted_distance() (pyunicorn.core.geo_network.GeoNetwork method), 31
outdegree() (pyunicorn.core.network.Network method), 80
outdegree_cdf() (pyunicorn.core.network.Network method), 80
outdegree_distribution() (pyunicorn.core.network.Network method), 81
outtotal_link_distance() (pyunicorn.core.geo_network.GeoNetwork method), 31

P

pagerank() (pyunicorn.core.network.Network method), 81
PartialCorrelationClimateNetwork (class in pyunicorn.climate.partial_correlation), 119
path_lengths() (pyunicorn.core.network.Network method), 81
path_lengths_1() (pyunicorn.climate.coupled_climate_network.CoupledClimateNetwork method), 109
path_lengths_2() (pyunicorn.climate.coupled_climate_network.CoupledClimateNetwork method), 109
permuted_copy() (pyunicorn.core.network.Network method), 81
phase_indices() (pyunicorn.climate.climate_data.ClimateData method), 95
phase_mean() (pyunicorn.climate.climate_data.ClimateData method), 96
phase_shift() (pyunicorn.climate.hilbert.HilbertClimateNetwork method), 115
print_boundaries() (pyunicorn.core.grid.Grid method), 39
print_data_info() (pyunicorn.core.data.Data method), 19
print_grid_size() (pyunicorn.core.grid.Grid method), 39
pyunicorn.climate.climate_data (module), 92
pyunicorn.climate.climate_network (module), 97
pyunicorn.climate.coupled_climate_network (module), 103
pyunicorn.climate.coupled_tsonis (module), 109
pyunicorn.climate.havlin (module), 111
pyunicorn.climate.hilbert (module), 113
pyunicorn.climate.map_plots (module), 115
pyunicorn.climate.mutual_info (module), 116
pyunicorn.climate.partial_correlation (module), 119
pyunicorn.climate.rainfall (module), 120
pyunicorn.climate.spearman (module), 122
pyunicorn.climate.tsonis (module), 124
pyunicorn.core.data (module), 15
pyunicorn.core.geo_network (module), 21
pyunicorn.core.grid (module), 35
pyunicorn.core.interacting_networks (module), 40
pyunicorn.core.netcdf_dictionary (module), 55
pyunicorn.core.network (module), 56
pyunicorn.core.resistive_network (module), 85
pyunicorn.funcnet.coupling_analysis (module), 155
pyunicorn.funcnet.coupling_analysis_pure_python (module), 159
pyunicorn.timeseries.cross_recurrence_plot (module), 127
pyunicorn.timeseries.inter_system_recurrence_network (module), 130
pyunicorn.timeseries.joint_recurrence_network (module), 133
pyunicorn.timeseries.joint_recurrence_plot (module), 135
pyunicorn.timeseries.recurrence_network (module), 137
pyunicorn.timeseries.recurrence_plot (module), 139
pyunicorn.timeseries.surrogates (module), 148
pyunicorn.timeseries.visibility_graph (module), 153
pyunicorn.utils.mpi (module), 163

Q

queue (in module pyunicorn.utils.mpi), 164

R

R (pyunicorn.timeseries.recurrence_plot.RecurrencePlot attribute), 140
RainfallClimateNetwork (class in pyunicorn.climate.rainfall), 120
randomly_rewire() (pyunicorn.core.network.Network method), 81
randomly_rewire_geomodel_I() (pyunicorn.core.geo_network.GeoNetwork method), 32
randomly_rewire_geomodel_II() (pyunicorn.core.geo_network.GeoNetwork method), 32
randomly_rewire_geomodel_III() (pyunicorn.core.geo_network.GeoNetwork method), 32
RandomlyRewireCrossLinks() (pyunicorn.core.interacting_networks.InteractingNetworks static method), 41
RandomlySetCrossLinks() (pyunicorn.core.interacting_networks.InteractingNetworks static method), 42
RandomlySetCrossLinks_sparse() (pyunicorn.core.interacting_networks.InteractingNetworks static method), 42
rank (in module pyunicorn.utils.mpi), 165
rank_time_series() (pyunicorn.climate.rainfall.RainfallClimateNetwork static method), 122
rank_time_series() (pyunicorn.climate.spearman.SpearmanClimateNetwork static method), 123
recurrence plot, 139
  RQA, 140
  time series, 151
  twins, 147
recurrence rate, 144
recurrence_matrix() (pyunicorn.timeseries.cross_recurrence_plot.CrossRecurrencePlot method), 129
recurrence_matrix() (pyunicorn.timeseries.joint_recurrence_plot.JointRecurrencePlot method), 136
recurrence_matrix() (pyunicorn.timeseries.recurrence_plot.RecurrencePlot method), 144
recurrence_plot() (pyunicorn.timeseries.surrogates.Surrogates method), 151
recurrence_probability() (pyunicorn.timeseries.recurrence_plot.RecurrencePlot method), 144
recurrence_rate() (pyunicorn.timeseries.cross_recurrence_plot.CrossRecurrencePlot method), 129
recurrence_rate() (pyunicorn.timeseries.recurrence_plot.RecurrencePlot method), 144
RecurrenceNetwork (class in pyunicorn.timeseries.recurrence_network), 137
RecurrencePlot (class in pyunicorn.timeseries.recurrence_plot), 139
refined_AAFT_surrogates() (pyunicorn.timeseries.surrogates.Surrogates method), 151
region() (pyunicorn.core.grid.Grid static method), 39
region_indices() (pyunicorn.core.grid.Grid method), 40
RegularGrid() (pyunicorn.core.grid.Grid static method), 35
rejection_sampling() (pyunicorn.timeseries.recurrence_plot.RecurrencePlot static method), 144
resample_diagline_dist() (pyunicorn.timeseries.recurrence_plot.RecurrencePlot method), 145
resample_vertline_dist() (pyunicorn.timeseries.recurrence_plot.RecurrencePlot method), 145
rescale() (pyunicorn.core.data.Data static method), 19
ResNetwork (class in pyunicorn.core.resistive_network), 86
resources (pyunicorn.climate.map_plots.MapPlots attribute), 116
results (in module pyunicorn.utils.mpi), 165
retarded_betweenness() (pyunicorn.timeseries.visibility_graph.VisibilityGraph method), 154
retarded_closeness() (pyunicorn.timeseries.visibility_graph.VisibilityGraph method), 154
retarded_degree() (pyunicorn.timeseries.visibility_graph.VisibilityGraph method), 154
retarded_local_clustering() (pyunicorn.timeseries.visibility_graph.VisibilityGraph method), 154
RQA
  cross recurrence plot, 127
  determinism, 142
  joint recurrence plot, 135
  recurrence plot, 140
rqa_summary() (pyunicorn.timeseries.recurrence_plot.RecurrencePlot method), 145
run() (in module pyunicorn.utils.mpi), 165

S

save() (pyunicorn.climate.climate_network.ClimateNetwork method), 101
save() (pyunicorn.core.geo_network.GeoNetwork method), 33
save() (pyunicorn.core.grid.Grid method), 40
save() (pyunicorn.core.network.Network method), 82
save_for_cgv() (pyunicorn.core.geo_network.GeoNetwork method), 33

save_ps_map() (pyunicorn.climate.map_plots.MapPlots method), 116
save_txt() (pyunicorn.core.grid.Grid method), 40
set_adaptive_neighborhood_size() (pyunicorn.timeseries.recurrence_network.RecurrenceNetwork method), 138
set_adaptive_neighborhood_size() (pyunicorn.timeseries.recurrence_plot.RecurrencePlot method), 145
set_directed() (pyunicorn.climate.hilbert.HilbertClimateNetwork method), 115
set_edge_list() (pyunicorn.core.network.Network method), 82
set_fixed_local_recurrence_rate() (pyunicorn.timeseries.recurrence_network.RecurrenceNetwork method), 139
set_fixed_local_recurrence_rate() (pyunicorn.timeseries.recurrence_plot.RecurrencePlot method), 146
set_fixed_recurrence_rate() (pyunicorn.timeseries.cross_recurrence_plot.CrossRecurrencePlot method), 129
set_fixed_recurrence_rate() (pyunicorn.timeseries.inter_system_recurrence_network.InterSystemRecurrenceNetwork method), 132
set_fixed_recurrence_rate() (pyunicorn.timeseries.joint_recurrence_network.JointRecurrenceNetwork method), 134
set_fixed_recurrence_rate() (pyunicorn.timeseries.joint_recurrence_plot.JointRecurrencePlot method), 136
set_fixed_recurrence_rate() (pyunicorn.timeseries.recurrence_network.RecurrenceNetwork method), 139
set_fixed_recurrence_rate() (pyunicorn.timeseries.recurrence_plot.RecurrencePlot method), 146
set_fixed_threshold() (pyunicorn.timeseries.cross_recurrence_plot.CrossRecurrencePlot method), 129
set_fixed_threshold() (pyunicorn.timeseries.inter_system_recurrence_network.InterSystemRecurrenceNetwork method), 132
set_fixed_threshold() (pyunicorn.timeseries.joint_recurrence_network.JointRecurrenceNetwork method), 134
set_fixed_threshold() (pyunicorn.timeseries.joint_recurrence_plot.JointRecurrencePlot method), 136
set_fixed_threshold() (pyunicorn.timeseries.recurrence_network.RecurrenceNetwork method), 139
set_fixed_threshold() (pyunicorn.timeseries.recurrence_plot.RecurrencePlot method), 146
set_fixed_threshold_std() (pyunicorn.timeseries.joint_recurrence_network.JointRecurrenceNetw
method), 135
set_fixed_threshold_std() (pyunicorn.timeseries.joint_recurrence_plot.JointRecurrencePlot method), 136
set_fixed_threshold_std() (pyunicorn.timeseries.recurrence_network.RecurrenceNetwork method), 139
set_fixed_threshold_std() (pyunicorn.timeseries.recurrence_plot.RecurrencePlot method), 146
set_global_window() (pyunicorn.climate.climate_data.ClimateData method), 96
set_global_window() (pyunicorn.core.data.Data method), 19
set_link_attribute() (pyunicorn.core.network.Network method), 82
set_link_density() (pyunicorn.climate.climate_network.ClimateNetwork method), 102
set_max_delay() (pyunicorn.climate.havlin.HavlinClimateNetwork method), 113
set_node_attribute() (pyunicorn.core.network.Network method), 82
set_node_weight_type() (pyunicorn.core.geo_network.GeoNetwork method), 34
set_non_local() (pyunicorn.climate.climate_network.ClimateNetwork method), 102
set_random_links_by_distance() (pyunicorn.core.geo_network.GeoNetwork method), 34
set_silence_level() (pyunicorn.core.data.Data method), 19
set_threshold() (pyunicorn.climate.climate_network.ClimateNetwork method), 103
set_window() (pyunicorn.climate.climate_data.ClimateData method), 96
set_window() (pyunicorn.core.data.Data method), 20
set_winter_only() (pyunicorn.climate.mutual_info.MutualInfoClimateNetwork method), 119
set_winter_only() (pyunicorn.climate.tsonis.TsonisClimateNetwork method), 126
shuffled_anomaly() (pyunicorn.climate.climate_data.ClimateData method), 97
shuffled_by_distance_copy() (pyunicorn.core.geo_network.GeoNetwork method), 34
shuffled_surrogate_for_cc() (pyunicorn.funcnet.coupling_analysis_pure_python.CouplingAnalysis

method), 162
shuffled_surrogate_for_mi() (pyunicorn.funcnet.coupling_analysis_pure_python.CouplingAnalysisPurePython method), 162
silence_level (pyunicorn.climate.coupled_tsonis.CoupledTsonisClimateNetwork attribute), 111
silence_level (pyunicorn.core.data.Data attribute), 20
silence_level (pyunicorn.core.grid.Grid attribute), 40
silence_level (pyunicorn.core.netcdf_dictionary.NetCDFDictionary attribute), 56
silence_level (pyunicorn.core.network.Network attribute), 83
silence_level (pyunicorn.funcnet.coupling_analysis.CouplingAnalysis attribute), 158
silence_level (pyunicorn.timeseries.inter_system_recurrence_network.InterSystemRecurrenceNetwork attribute), 132
silence_level (pyunicorn.timeseries.recurrence_plot.RecurrencePlot attribute), 146
silence_level (pyunicorn.timeseries.surrogates.Surrogates attribute), 151
silence_level (pyunicorn.timeseries.visibility_graph.VisibilityGraph attribute), 154
similarity_measure() (pyunicorn.climate.climate_network.ClimateNetwork method), 103
similarity_measure_1() (pyunicorn.climate.coupled_climate_network.CoupledClimateNetwork method), 109
similarity_measure_2() (pyunicorn.climate.coupled_climate_network.CoupledClimateNetwork method), 109
sin_lat() (pyunicorn.core.grid.Grid method), 40
sin_lon() (pyunicorn.core.grid.Grid method), 40
size (in module pyunicorn.utils.mpi), 165
slave_queue (in module pyunicorn.utils.mpi), 165
SmallComplexNetwork() (pyunicorn.core.resistive_network.ResNetwork static method), 86
SmallTestData() (pyunicorn.climate.climate_data.ClimateData static method), 93
SmallTestData() (pyunicorn.core.data.Data static method), 16
SmallTestData() (pyunicorn.timeseries.surrogates.Surrogates static method), 148
SmallTestGrid() (pyunicorn.core.grid.Grid static method), 35
SmallTestNetwork() (pyunicorn.climate.climate_network.ClimateNetwork static method), 98
SmallTestNetwork() (pyunicorn.climate.tsonis.TsonisClimateNetwork static method), 124
SmallTestNetwork() (pyunicorn.core.geo_network.GeoNetwork static method), 23
SmallTestNetwork() (pyunicorn.core.interacting_networks.InteractingNetworks static method), 42
SmallTestNetwork() (pyunicorn.core.network.Network static method), 59
SmallTestNetwork() (pyunicorn.core.resistive_network.ResNetwork static method), 86
sp_A (pyunicorn.core.network.Network attribute), 83
sp_Aplus() (pyunicorn.core.network.Network method), 83
sp_diag_sqrt_w() (pyunicorn.core.network.Network method), 83
sp_diag_w() (pyunicorn.core.network.Network method), 83
sp_diag_w_inv() (pyunicorn.core.network.Network method), 83
sp_nsi_diag_k() (pyunicorn.core.network.Network method), 83
sp_nsi_diag_k_inv() (pyunicorn.core.network.Network method), 83
sparsity (pyunicorn.timeseries.recurrence_plot.RecurrencePlot attribute), 146
spatio-temporal window, 15, 93
SpearmanClimateNetwork (class in pyunicorn.climate.spearman), 122
splitted_copy() (pyunicorn.core.network.Network method), 83
spreading() (pyunicorn.core.network.Network method), 83
start_time (in module pyunicorn.utils.mpi), 165
stats (in module pyunicorn.utils.mpi), 165
submit_call() (in module pyunicorn.utils.mpi), 165
subnetwork() (pyunicorn.core.interacting_networks.InteractingNetworks method), 55
supremum_distance_matrix() (pyunicorn.timeseries.cross_recurrence_plot.CrossRecurrencePlot method), 130
supremum_distance_matrix() (pyunicorn.timeseries.recurrence_plot.RecurrencePlot method), 146
surrogates
    twins, 153
Surrogates (class in pyunicorn.timeseries.surrogates), 148
symmetrize_by_absmax() (pyunicorn.funcnet.coupling_analysis.CouplingAnalysis method), 158

T

terminate() (in module pyunicorn.utils.mpi), 166
test_data() (pyunicorn.funcnet.coupling_analysis.CouplingAnalysis static method), 159
test_mutual_information() (pyunicorn.timeseries.surrogates.Surrogates static method), 151
test_pearson_correlation() (pyunicorn.timeseries.surrogates.Surrogates static

method), 151
test_threshold_significance() (pyunicorn.timeseries.surrogates.Surrogates method), 152
threshold() (pyunicorn.climate.climate_network.ClimateNetwork method), 103
threshold_from_link_density() (pyunicorn.climate.climate_network.ClimateNetwork method), 103
threshold_from_recurrence_rate, 147
threshold_from_recurrence_rate() (pyunicorn.timeseries.recurrence_plot.RecurrencePlot static method), 146
threshold_from_recurrence_rate_fast() (pyunicorn.timeseries.recurrence_plot.RecurrencePlot static method), 146
time series
    normalize, 144
    recurrence plot, 151
time series array
    normalize, 18, 150
time_cycle (pyunicorn.climate.climate_data.ClimateData attribute), 97
time_series (pyunicorn.timeseries.recurrence_plot.RecurrencePlot attribute), 147
time_series (pyunicorn.timeseries.visibility_graph.VisibilityGraph attribute), 154
time_surrogate_for_cc() (pyunicorn.funcnet.coupling_analysis_pure_python.CouplingAnalysisPurePython method), 162
time_surrogate_for_mi() (pyunicorn.funcnet.coupling_analysis_pure_python.CouplingAnalysisPurePython method), 162
timings (pyunicorn.timeseries.visibility_graph.VisibilityGraph attribute), 154
title (pyunicorn.climate.map_plots.MapPlots attribute), 116
to_file() (pyunicorn.core.netcdf_dictionary.NetCDFDictionary method), 56
total_link_distance() (pyunicorn.core.geo_network.GeoNetwork method), 34
total_node_weight (pyunicorn.core.network.Network attribute), 84
total_time (in module pyunicorn.utils.mpi), 166
total_time_est (in module pyunicorn.utils.mpi), 166
trans_betweenness() (pyunicorn.timeseries.visibility_graph.VisibilityGraph method), 154
transitivity() (pyunicorn.core.network.Network method), 84
transitivity_dim_single_scale() (pyunicorn.timeseries.recurrence_network.RecurrenceNetwork method), 139
trapping time, 141
trapping_time() (pyunicorn.timeseries.recurrence_plot.RecurrencePlot method), 147
TsonisClimateNetwork (class in pyunicorn.climate.tsonis), 124
twin surrogate, 147
twin_surrogates() (pyunicorn.timeseries.recurrence_plot.RecurrencePlot method), 147
twin_surrogates() (pyunicorn.timeseries.surrogates.Surrogates method), 152
twins
    recurrence plot, 147
    surrogates, 153
twins() (pyunicorn.timeseries.recurrence_plot.RecurrencePlot method), 147
twins() (pyunicorn.timeseries.surrogates.Surrogates method), 153

U

undirected_adjacency() (pyunicorn.core.network.Network method), 84
undirected_copy() (pyunicorn.core.network.Network method), 84
update_admittance() (pyunicorn.core.resistive_network.ResNetwork method), 91
update_R() (pyunicorn.core.resistive_network.ResNetwork method), 90
update_resistances() (pyunicorn.core.resistive_network.ResNetwork method), 91

V

vert_entropy() (pyunicorn.timeseries.recurrence_plot.RecurrencePlot method), 147
vertex_current_flow_betweenness() (pyunicorn.core.resistive_network.ResNetwork method), 92
vertical
    line entropy, 147
    line length, average, 141
    line length, frequency distribution, 148
    line length, maximum, 144
    lines, 148
vertline_dist() (pyunicorn.timeseries.recurrence_plot.RecurrencePlot method), 148
visibility_relations() (pyunicorn.timeseries.visibility_graph.VisibilityGraph method), 155
visibility_relations_horizontal() (pyunicorn.timeseries.visibility_graph.VisibilityGraph method), 155
VisibilityGraph (class in pyunicorn.timeseries.visibility_graph), 153

W

WattsStrogatzGraph() (pyuni-

corn.core.network.Network static method), 59
weave_inline() (in module pyunicorn.core.network), 85
weighted_local_clustering() (pyunicorn.core.network.Network static method), 84
white vertical
  line entropy, 148
  line length, average, 141
  line length, frequency distribution, 148
  line length, maximum, 144
  lines, 148
white_noise_surrogates() (pyunicorn.timeseries.surrogates.Surrogates method), 153
white_vert_entropy() (pyunicorn.timeseries.recurrence_plot.RecurrencePlot method), 148
white_vertline_dist() (pyunicorn.timeseries.recurrence_plot.RecurrencePlot method), 148
window() (pyunicorn.core.data.Data method), 20
winter_only() (pyunicorn.climate.mutual_info.MutualInfoClimateNetwork method), 119
winter_only() (pyunicorn.climate.tsonis.TsonisClimateNetwork method), 127

X

x (pyunicorn.timeseries.cross_recurrence_plot.CrossRecurrencePlot attribute), 130
x (pyunicorn.timeseries.inter_system_recurrence_network.InterSystemRecurrenceNetwork attribute), 133
x (pyunicorn.timeseries.joint_recurrence_plot.JointRecurrencePlot attribute), 137
x_embedded (pyunicorn.timeseries.cross_recurrence_plot.CrossRecurrencePlot attribute), 130
x_embedded (pyunicorn.timeseries.inter_system_recurrence_network.InterSystemRecurrenceNetwork attribute), 133
x_embedded (pyunicorn.timeseries.joint_recurrence_plot.JointRecurrencePlot attribute), 137

Y

y (pyunicorn.timeseries.cross_recurrence_plot.CrossRecurrencePlot attribute), 130
y (pyunicorn.timeseries.inter_system_recurrence_network.InterSystemRecurrenceNetwork attribute), 133
y (pyunicorn.timeseries.joint_recurrence_plot.JointRecurrencePlot attribute), 137
y_embedded (pyunicorn.timeseries.cross_recurrence_plot.CrossRecurrencePlot attribute), 130
y_embedded (pyunicorn.timeseries.inter_system_recurrence_network.InterSystemRecurrenceNetwork attribute), 133
y_embedded (pyunicorn.timeseries.joint_recurrence_plot.JointRecurrencePlot attribute), 137

Z

zero padded data, 20
zero_pad_data() (pyunicorn.core.data.Data static method), 20