Nolds Documentation

Release 0.5.2

Christopher Schölzel

May 07, 2020

Contents

The acronym Nolds stands for 'NOnLinear measures for Dynamical Systems'. It is a small numpy-based library that provides an implementation and a learning resource for nonlinear measures for dynamical systems based on one-dimensional time series.

Nolds is hosted on GitHub. This documentation describes the latest version. A change log of the different versions can be found on GitHub.

For the impatient, here is a small example how you can calculate the lyapunov exponent of the logistic map with Nolds:

```
import nolds
import numpy as np
lm = nolds.logistic_map(0.1, 1000, r=4)
x = np.fromiter(lm, dtype="float32")
l = max(nolds.lyap_e(x))
```

Contents:

CHAPTER 1

nolds module

Nolds only consists of to single module called nolds which contains all relevant algorithms and helper functions.

Internally these functions are subdivided into different modules such as measures and datasets, but you should not need to import these modules directly unless you want access to some internal helper functions.

1.1 Algorithms

1.1.1 Lyapunov exponent (Rosenstein et al.)

nolds.**lyap_r**(*data*, *emb_dim=10*, *lag=None*, *min_tsep=None*, *tau=1*, *min_neighbors=20*, *trajectory_len=20*, *fit=u'RANSAC'*, *debug_plot=False*, *debug_data=False*, *plot_file=None*, *fit_offset=0*)

Estimates the largest Lyapunov exponent using the algorithm of Rosenstein et al. [lr_1].

Explanation of Lyapunov exponents: See lyap_e.

Explanation of the algorithm: The algorithm of Rosenstein et al. is only able to recover the largest Lyapunov exponent, but behaves rather robust to parameter choices.

The idea for the algorithm relates closely to the definition of Lyapunov exponents. First, the dynamics of the data are reconstructed using a delay embedding method with a lag, such that each value x_i of the data is mapped to the vector

X_i = [x_i, x_(i+lag), x_(i+2*lag), ..., x_(i+(emb_dim-1) * lag)]

For each such vector X_i, we find the closest neighbor X_j using the euclidean distance. We know that as we follow the trajectories from X_i and X_j in time in a chaotic system the distances between X_(i+k) and X_(j+k) denoted as d_i(k) will increase according to a power law d_i(k) = c * e^(lambda * k) where lambda is a good approximation of the highest Lyapunov exponent, because the exponential expansion along the axis associated with this exponent will quickly dominate the expansion or contraction along other axes.

To calculate lambda, we look at the logarithm of the distance trajectory, because log(d_i(k)) = log(c) + lambda * k. This gives a set of lines (one for each index i) whose slope is an approximation of lambda. We therefore extract the mean log trajectory d'(k) by taking the mean of log(d_i(k)) over all orbit vectors X_i.

We then fit a straight line to the plot of d'(k) versus k. The slope of the line gives the desired parameter lambda.

Method for choosing min_tsep: Usually we want to find neighbors between points that are close in phase space but not too close in time, because we want to avoid spurious correlations between the obtained trajectories that originate from temporal dependencies rather than the dynamic properties of the system. Therefore it is critical to find a good value for min_tsep. One rather plausible estimate for this value is to set min_tsep to the mean period of the signal, which can be obtained by calculating the mean frequency using the fast fourier transform. This procedure is used by default if the user sets min_tsep = None.

Method for choosing lag: Another parameter that can be hard to choose by instinct alone is the lag between individual values in a vector of the embedded orbit. Here, Rosenstein et al. suggest to set the lag to the distance where the autocorrelation function drops below 1 - 1/e times its original (maximal) value. This procedure is used by default if the user sets lag = None.

References:

Reference Code:

Args:

data (iterable of float): (one-dimensional) time series

Kwargs:

emb_dim (int): embedding dimension for delay embedding

lag (float): lag for delay embedding

min_tsep (float): minimal temporal separation between two "neighbors" (default: find a suitable value by calculating the mean period of the data)

tau (float): step size between data points in the time series in seconds (normalization scaling factor for exponents)

min_neighbors (int): if lag=None, the search for a suitable lag will be stopped when the number of potential neighbors for a vector drops below min_neighbors

trajectory_len (int): the time (in number of data points) to follow the distance trajectories between two neighboring points

fit (str): the fitting method to use for the line fit, either 'poly' for normal least squares polynomial fitting or 'RANSAC' for RANSAC-fitting which is more robust to outliers

debug_plot (boolean): if True, a simple plot of the final line-fitting step will be shown

debug_data (boolean): if True, debugging data will be returned alongside the result

plot_file (str): if debug_plot is True and plot_file is not None, the plot will be saved under the given file name instead of directly showing it through `plt.show()`

fit_offset (int): neglect the first fit_offset steps when fitting

Returns:

float: an estimate of the largest Lyapunov exponent (a positive exponent is a strong indicator for chaos)

(1d-vector, 1d-vector, list): only present if debug_data is True: debug data of the form `(ks, div_traj, poly)` where `ks` are the x-values of the line fit, `div_traj` are the y-values and `poly` are the line coefficients (`[slope, intercept]`).

1.1.2 Lyapunov exponent (Eckmann et al.)

nolds.**lyap_e**(*data*, *emb_dim=10*, *matrix_dim=4*, *min_nb=None*, *min_tsep=0*, *tau=1*, *debug_plot=False*, *debug_data=False*, *plot_file=None*)

Estimates the Lyapunov exponents for the given data using the algorithm of Eckmann et al. [le_1].

Recommendations for parameter settings by Eckmann et al.:

- long recording time improves accuracy, small tau does not
- use large values for emb_dim
- matrix_dim should be 'somewhat larger than the expected number of positive Lyapunov exponents'
- min_nb = min(2 * matrix_dim, matrix_dim + 4)

Explanation of Lyapunov exponents: The Lyapunov exponent describes the rate of separation of two infinitesimally close trajectories of a dynamical system in phase space. In a chaotic system, these trajectories diverge exponentially following the equation:

|X(t, X_0) - X(t, X_0 + eps)| = e^(lambda * t) * |eps|

In this equation X(t, X_0) is the trajectory of the system X starting at the point X_0 in phase space at time t. eps is the (infinitesimal) difference vector and lambda is called the Lyapunov exponent. If the system has more than one free variable, the phase space is multidimensional and each dimension has its own Lyapunov exponent. The existence of at least one positive Lyapunov exponent is generally seen as a strong indicator for chaos.

Explanation of the Algorithm: To calculate the Lyapunov exponents analytically, the Jacobian of the system is required. The algorithm of Eckmann et al. therefore tries to estimate this Jacobian by reconstructing the dynamics of the system from which the time series was obtained. For this, several steps are required:

- Embed the time series [x_1, x_2, ..., x_(N-1)] in an orbit of emb_dim dimensions (map each point x_i of the time series to a vector [x_i, x_(i+1), x_(i+2), ... x_(i+emb_dim-1)]).
- For each vector X_i in this orbit find a radius r_i so that at least min_nb other vectors lie within (chebyshev-)distance r_i around X_i. These vectors will be called "neighbors" of X_i.
- Find the Matrix T_i that sends points from the neighborhood of X_i to the neighborhood of X_(i+1). To avoid undetermined values in T_i, we construct T_i not with size (emb_dim x emb_dim) but with size (matrix_dim x matrix_dim), so that we have a larger "step size" m in the X_i, which are now defined as X'_i = [x_i, x_(i+m), x_(i+2m), ... x_(i+(matrix_dim-1)*m)]. This means that emb_dim-1 must be divisible by matrix_dim-1. The T_i are then found by a linear least squares fit, assuring that T_i (X_j - X_i) ~= X_(j+m) - X_(i+m) for any X_j in the neighborhood of X_i.
- Starting with i = 1 and Q_0 = identity successively decompose the matrix T_i * Q_(i-1) into the matrices Q_i and R_i by a QR-decomposition.
- Calculate the Lyapunov exponents from the mean of the logarithm of the diagonal elements of the matrices R_i. To normalize the Lyapunov exponents, they have to be divided by m and by the step size tau of the original time series.

References:

Reference code:

Args:

data (array-like of float): (scalar) data points

Kwargs:

emb_dim (int): embedding dimension

matrix_dim (int): matrix dimension (emb_dim - 1 must be divisible by matrix_dim - 1)

min_nb (int): minimal number of neighbors (default: min(2 * matrix_dim, matrix_dim + 4))

min_tsep (int): minimal temporal separation between two "neighbors"

tau (float): step size of the data in seconds (normalization scaling factor for exponents)

debug_plot (boolean): if True, a histogram matrix of the individual estimates will be shown

debug_data (boolean): if True, debugging data will be returned alongside the result

plot_file (str): if debug_plot is True and plot_file is not None, the plot will be saved under the given file name instead of directly showing it through `plt.show()`

Returns:

float array: array of matrix_dim Lyapunov exponents (positive exponents are indicators for chaos)

2d-array of floats: only present if debug_data is True: all estimates for the matrix_dim Lyapunov exponents from the x iterations of R_i. The shape of this debug data is (x, matrix_dim).

1.1.3 Sample entropy

`nolds.`**`sampen`**(*data*, *emb_dim=2*, *tolerance=None*, *dist=<function rowwise_chebyshev>*, *debug_plot=False*, *debug_data=False*, *plot_file=None*)

Computes the sample entropy of the given data.

Explanation of the sample entropy: The sample entropy of a time series is defined as the negative natural logarithm of the conditional probability that two sequences similar for emb_dim points remain similar at the next point, excluding self-matches.

A lower value for the sample entropy therefore corresponds to a higher probability indicating more self-similarity.

Explanation of the algorithm: The algorithm constructs all subsequences of length emb_dim [s_1, s_2, s_3, ...] and then counts each pair (s_i, s_j) with i != j where dist(s_i, s_j) < tolerance. The same process is repeated for all subsequences of length emb_dim + 1. The sum of similar sequence pairs with length emb_dim + 1 is divided by the sum of similar sequence pairs with length emb_dim. The result of the algorithm is the negative logarithm of this ratio/probability.

References:

Reference code:

Args:

data (array-like of float): input data

Kwargs:

emb_dim (int): the embedding dimension (length of vectors to compare)

tolerance (float): distance threshold for two template vectors to be considered equal (default: 0.2 * std(data))

dist (function (2d-array, 1d-array) -> 1d-array): distance function used to calculate the distance between template vectors. Sampen is defined using `rowwise_chebyshev`. You should only use something else, if you are sure that you need it.

debug_plot (boolean): if True, a histogram of the individual distances for m and m+1

debug_data (boolean): if True, debugging data will be returned alongside the result

plot_file (str): if debug_plot is True and plot_file is not None, the plot will be saved under the given file name instead of directly showing it through `plt.show()`

Returns:

float: the sample entropy of the data (negative logarithm of ratio between similar template vectors of length emb_dim + 1 and emb_dim)

[float list, float list]: Lists of lists of the form `[dists_m, dists_m1]` containing the distances between template vectors for m (dists_m) and for m + 1 (dists_m1).

1.1.4 Hurst exponent

nolds.**hurst_rs**(*data*, *nvals=None*, *fit=u'RANSAC'*, *debug_plot=False*, *debug_data=False*, *plot_file=None*, *corrected=True*, *unbiased=True*)

Calculates the Hurst exponent by a standard rescaled range (R/S) approach.

Explanation of Hurst exponent: The Hurst exponent is a measure for the "long-term memory" of a time series, meaning the long statistical dependencies in the data that do not originate from cycles.

It originates from H.E. Hursts observations of the problem of long-term storage in water reservoirs. If x_i is the discharge of a river in year i and we observe this discharge for N years, we can calculate the storage capacity that would be required to keep the discharge steady at its mean value.

To do so, we first substract the mean over all x_i from the individual x_i to obtain the departures x'_i from the mean for each year i. As the excess or deficit in discharge always carrys over from year i to year i+1, we need to examine the cumulative sum of x'_i, denoted by y_i. This cumulative sum represents the filling of our hypothetical storage. If the sum is above 0, we are storing excess discharge from the river, if it is below zero we have compensated a deficit in discharge by releasing water from the storage. The range (maximum - minimum) R of y_i therefore represents the total capacity required for the storage.

Hurst showed that this value follows a steady trend for varying N if it is normalized by the standard deviation sigma over the x_i. Namely he obtained the following formula:

R/sigma = (N/2)^K

In this equation, K is called the Hurst exponent. Its value is 0.5 for white noise, but becomes greater for time series that exhibit some positive dependency on previous values. For negative dependencies it becomes less than 0.5.

Explanation of the algorithm: The rescaled range (R/S) approach is directly derived from Hurst's definition. The time series of length N is split into non-overlapping subseries of length n. Then, R and S (S = sigma) are calculated for each subseries and the mean is taken over all subseries yielding (R/S)_n. This process is repeated for several lengths n. Finally, the exponent K is obtained by fitting a straight line to the plot of log((R/S)_n) vs log(n).

There seems to be no consensus how to chose the subseries lenghts n. This function therefore leaves the choice to the user. The module provides some utility functions for "typical" values:

- binary_n: N/2, N/4, N/8, ...
- logarithmic_n: min_n, min_n * f, min_n * f^2, ...

References:

Reference Code:

Args:

data (array-like of float): time series

Kwargs:

nvals (iterable of int): sizes of subseries to use (default: logmid_n(total_N, ratio=1/4.0, nsteps=15) , that is 15 logarithmically spaced values in the medium 25% of the logarithmic range)

Generally, the choice for n is a trade-off between the length and the number of the subsequences that are used for the calculation of the (R/S)_n. Very low values of n lead to high variance in the `r` and `s` while very high values may leave too few subsequences that the mean along them is still meaningful. Logarithmic spacing makes sense, because it translates to even spacing in the log-log-plot.

fit (str): the fitting method to use for the line fit, either 'poly' for normal least squares polynomial fitting or 'RANSAC' for RANSAC-fitting which is more robust to outliers

debug_plot (boolean): if True, a simple plot of the final line-fitting step will be shown

debug_data (boolean): if True, debugging data will be returned alongside the result

plot_file (str): if debug_plot is True and plot_file is not None, the plot will be saved under the given file name instead of directly showing it through `plt.show()`

corrected (boolean): if True, the Anis-Lloyd-Peters correction factor will be applied to the output according to the expected value for the individual (R/S)_n (see [h_3])

unbiased (boolean): if True, the standard deviation based on the unbiased variance (1/(N-1) instead of 1/N) will be used. This should be the default choice, since the true mean of the sequences is not known. This parameter should only be changed to recreate results of other implementations.

Returns:

float: estimated Hurst exponent K using a rescaled range approach (if K = 0.5 there are no long-range correlations in the data, if K < 0.5 there are negative long-range correlations, if K > 0.5 there are positive long-range correlations)

(1d-vector, 1d-vector, list): only present if debug_data is True: debug data of the form `(nvals, rsvals, poly)` where `nvals` are the values used for log(n), `rsvals` are the corresponding log((R/S)_n) and `poly` are the line coefficients (`[slope, intercept]`)

1.1.5 Correlation dimension

`nolds.`**`corr_dim`**(*data*, *emb_dim*, *rvals=None*, *dist=<function rowwise_euclidean>*, *fit=u'RANSAC'*, *debug_plot=False*, *debug_data=False*, *plot_file=None*)

Calculates the correlation dimension with the Grassberger-Procaccia algorithm

Explanation of correlation dimension: The correlation dimension is a characteristic measure that can be used to describe the geometry of chaotic attractors. It is defined using the correlation sum C(r) which is the fraction of pairs of points X_i in the phase space whose distance is smaller than r.

If the relation between C(r) and r can be described by the power law

C(r) ~ r^D

then D is called the correlation dimension of the system.

In a d-dimensional system, the maximum value for D is d. This value is obtained for systems that expand uniformly in each dimension with time. The lowest possible value is 0 for a system with constant C(r) (i.e. a system that visits just one point in the phase space). Generally if D is lower than d and the system has an attractor, this attractor is called "strange" and D is a measure of this "strangeness".

Explanation of the algorithm: The Grassberger-Procaccia algorithm calculates C(r) for a range of different r and then fits a straight line into the plot of log(C(r)) versus log(r).

This version of the algorithm is created for one-dimensional (scalar) time series. Therefore, before calculating C(r), a delay embedding of the time series is performed to yield emb_dim dimensional vectors Y_i = [X_i, X_(i+1), X_(i+2), ... X_(i+embd_dim-1)]. Choosing a higher value for emb_dim allows to reconstruct higher dimensional dynamics and avoids "systematic errors due to corrections to scaling".

References:

Reference Code:

Args:

data (array-like of float): time series of data points

emb_dim (int): embedding dimension

Kwargs:

rvals (iterable of float): list of values for to use for r (default: logarithmic_r(0.1 * std, 0.5 * std, 1.03))

dist (function (2d-array, 1d-array) -> 1d-array): row-wise difference function

fit (str): the fitting method to use for the line fit, either 'poly' for normal least squares polynomial fitting or 'RANSAC' for RANSAC-fitting which is more robust to outliers

debug_plot (boolean): if True, a simple plot of the final line-fitting step will be shown

debug_data (boolean): if True, debugging data will be returned alongside the result

plot_file (str): if debug_plot is True and plot_file is not None, the plot will be saved under the given file name instead of directly showing it through `plt.show()`

Returns:

float: correlation dimension as slope of the line fitted to log(r) vs log(C(r))

(1d-vector, 1d-vector, list): only present if debug_data is True: debug data of the form `(rvals, csums, poly)` where `rvals` are the values used for log(r), `csums` are the corresponding log(C(r)) and `poly` are the line coefficients (`[slope, intercept]`)

1.1.6 Detrended fluctuation analysis

`nolds.`**`dfa`**(*data*, *nvals=None*, *overlap=True*, *order=1*, *fit_trend=u'poly'*, *fit_exp=u'RANSAC'*, *debug_plot=False*, *debug_data=False*, *plot_file=None*)

Performs a detrended fluctuation analysis (DFA) on the given data

Recommendations for parameter settings by Hardstone et al.:

- nvals should be equally spaced on a logarithmic scale so that each window scale hase the same weight
- min(nvals) < 4 does not make much sense as fitting a polynomial (even if it is only of order 1) to 3 or less data points is very prone.
- max(nvals) > len(data) / 10 does not make much sense as we will then have less than 10 windows to calculate the average fluctuation
- use overlap=True to obtain more windows and therefore better statistics (at an increased computational cost)

Explanation of DFA: Detrended fluctuation analysis, much like the Hurst exponent, is used to find long-term statistical dependencies in time series.

The idea behind DFA originates from the definition of self-affine processes. A process X is said to be self-affine if the standard deviation of the values within a window of length n changes with the window length factor L in a power law:

std(X,L * n) = L^H * std(X, n)

where std(X, k) is the standard deviation of the process X calculated over windows of size k. In this equation, H is called the Hurst parameter, which behaves indeed very similar to the Hurst exponent.

Like the Hurst exponent, H can be obtained from a time series by calculating std(X,n) for different n and fitting a straight line to the plot of log(std(X,n)) versus log(n).

To calculate a single std(X,n), the time series is split into windows of equal length n, so that the ith window of this size has the form

W_(n,i) = [x_i, x_(i+1), x_(i+2), ... x_(i+n-1)]

The value std(X,n) is then obtained by calculating std(W_(n,i)) for each i and averaging the obtained values over i.

The aforementioned definition of self-affinity, however, assumes that the process is non-stationary (i.e. that the standard deviation changes over time) and it is highly influenced by local and global trends of the time series.

To overcome these problems, an estimate alpha of H is calculated by using a "walk" or "signal profile" instead of the raw time series. This walk is obtained by substracting the mean and then taking the cumulative sum of the original time series. The local trends are removed for each window separately by fitting a polynomial p_(n,i) to the window W_(n,i) and then calculating W'_(n,i) = W_(n,i) - p_(n,i) (element-wise substraction).

We then calculate std(X,n) as before only using the "detrended" window W'_(n,i) instead of W_(n,i). Instead of H we obtain the parameter alpha from the line fitting.

For alpha < 1 the underlying process is stationary and can be modelled as fractional Gaussian noise with H = alpha. This means for alpha = 0.5 we have no correlation or "memory", for 0.5 < alpha < 1 we have a memory with positive correlation and for alpha < 0.5 the correlation is negative.

For alpha > 1 the underlying process is non-stationary and can be modeled as fractional Brownian motion with H = alpha - 1.

References:

Reference code:

Args:

data (array-like of float): time series

Kwargs:

nvals (iterable of int): subseries sizes at which to calculate fluctuation (default: logarithmic_n(4, 0.1*len(data), 1.2))

overlap (boolean): if True, the windows W_(n,i) will have a 50% overlap, otherwise non-overlapping windows will be used

order (int): (polynomial) order of trend to remove

fit_trend (str): the fitting method to use for fitting the trends, either 'poly' for normal least squares polynomial fitting or 'RANSAC' for RANSAC-fitting which is more robust to outliers but also tends to lead to unstable results

fit_exp (str): the fitting method to use for the line fit, either 'poly' for normal least squares polynomial fitting or 'RANSAC' for RANSAC-fitting which is more robust to outliers

debug_plot (boolean): if True, a simple plot of the final line-fitting step will be shown

debug_data (boolean): if True, debugging data will be returned alongside the result

plot_file (str): if debug_plot is True and plot_file is not None, the plot will be saved under the given file name instead of directly showing it through `plt.show()`

Returns:

float: the estimate alpha for the Hurst parameter (alpha < 1: stationary process similar to fractional Gaussian noise with H = alpha, alpha > 1: non-stationary process similar to fractional Brownian motion with H = alpha - 1)

(1d-vector, 1d-vector, list): only present if debug_data is True: debug data of the form `(nvals, fluctuations, poly)` where `nvals` are the values used for log(n), `fluctuations` are the corresponding log(std(X,n)) and `poly` are the line coefficients (`[slope, intercept]`)

1.2 Helper functions

`nolds.`**`binary_n`**(*total_N*, *min_n=50*)

Creates a list of values by successively halving the total length total_N until the resulting value is less than min_n.

Non-integer results are rounded down.

Args:

total_N (int): total length

Kwargs:

min_n (int): minimal length after division

Returns:

list of integers: total_N/2, total_N/4, total_N/8, ... until total_N/2^i < min_n

`nolds.`**`logarithmic_n`**(*min_n*, *max_n*, *factor*)

Creates a list of values by successively multiplying a minimum value min_n by a factor > 1 until a maximum value max_n is reached.

Non-integer results are rounded down.

Args:

min_n (float): minimum value (must be < max_n)

max_n (float): maximum value (must be > min_n)

factor (float): factor used to increase min_n (must be > 1)

Returns:

list of integers: min_n, min_n * factor, min_n * factor^2, ... min_n * factor^i < max_n without duplicates

`nolds.`**`logarithmic_r`**(*min_n*, *max_n*, *factor*)

Creates a list of values by successively multiplying a minimum value min_n by a factor > 1 until a maximum value max_n is reached.

Args:

min_n (float): minimum value (must be < max_n)

max_n (float): maximum value (must be > min_n)

factor (float): factor used to increase min_n (must be > 1)

Returns:

list of floats: min_n, min_n * factor, min_n * factor^2, ... min_n * factor^i < max_n

`nolds.`**`expected_h`**(*nvals*, *fit=u'RANSAC'*)

Uses expected_rs to calculate the expected value for the Hurst exponent h based on the values of n used for the calculation.

Args:

nvals (iterable of int): the values of n used to calculate the individual (R/S)_n

KWargs:

fit (str): the fitting method to use for the line fit, either 'poly' for normal least squares polynomial fitting or 'RANSAC' for RANSAC-fitting which is more robust to outliers

Returns:

float: expected h for white noise

nolds.**expected_rs**(*n*)

Calculates the expected (R/S)_n for white noise for a given n.

This is used as a correction factor in the function hurst_rs. It uses the formula of Anis-Lloyd-Peters (see [h_3]).

Args:

n (int): the value of n for which the expected (R/S)_n should be calculated

Returns:

float: expected (R/S)_n for white noise

nolds.**logmid_n**(*max_n*, *ratio=0.25*, *nsteps=15*)

Creates an array of integers that lie evenly spaced in the "middle" of the logarithmic scale from 0 to log(max_n).

If max_n is very small and/or nsteps is very large, this may lead to duplicate values which will be removed from the output.

This function has benefits in hurst_rs, because it cuts away both very small and very large n, which both can cause problems, and still produces a logarithmically spaced sequence.

Args:

max_n (int): largest possible output value (should be the sequence length when used in hurst_rs)

Kwargs:

ratio (float): width of the "middle" of the logarithmic interval relative to log(max_n). For example, for ratio=1/2.0 the logarithm of the resulting values will lie between 0.25 * log(max_n) and 0.75 * log(max_n).

nsteps (float): (maximum) number of values to take from the specified range

Returns:

array of int: a logarithmically spaced sequence of at most nsteps values (may be less, because only unique values are returned)

nolds.**lyap_r_len**(***kwargs*)

Helper function that calculates the minimum number of data points required to use lyap_r.

Note that none of the required parameters may be set to None.

Kwargs:

kwargs(dict): arguments used for lyap_r (required: emb_dim, lag, trajectory_len and min_tsep)

Returns: minimum number of data points required to call lyap_r with the given parameters

nolds.**lyap_e_len**(***kwargs*)

Helper function that calculates the minimum number of data points required to use lyap_e.

Note that none of the required parameters may be set to None.

Kwargs:

kwargs(dict): arguments used for lyap_e (required: emb_dim, matrix_dim, min_nb and min_tsep)

Returns: minimum number of data points required to call lyap_e with the given parameters

1.3 Datasets

1.3.1 Benchmark dataset for hurst exponent

nolds.**brown72 = <sphinx.ext.autodoc.importer._MockObject object>**
Used by autodoc_mock_imports.

The brown72 dataset has a prescribed (uncorrected) Hurst exponent of 0.7270. It is a synthetic dataset from the book "Chaos and Order in the Capital markets"[b7_a].

It is included here, because the dataset can be found online [b7_b] and is used by other software packages such as the R-package `pracma` [b7_c]. As such it can be used to compare different implementations.

However, it should be noted that the idea that the "true" Hurst exponent of this series is indeed 0.7270 is problematic for several reasons:

1. This value does not take into account the Anis-Lloyd-Peters correction factor.
2. It was obtained using the biased version of the standard deviation.
3. It depends (as always for the Hurst exponent) on the choice of the length of the subsequences.

If you want to reproduce the prescribed value, you can use the following code:

```
nolds.hurst_rs(
   nolds.brown72,
   nvals=2**np.arange(3,11),
   fit="poly", corrected=False, unbiased=False
)
```

References:

1.3.2 Tent map

nolds.**tent_map**(*x*, *steps*, *mu=2*)
Generates a time series of the tent map.

Characteristics and Background: The name of the tent map is derived from the fact that the plot of x_i vs x_i+1 looks like a tent. For mu > 1 one application of the mapping function can be viewed as stretching the surface on which the value is located and then folding the area that is greater than one back towards the zero. This corresponds nicely to the definition of chaos as expansion in one dimension which is counteracted by a compression in another dimension.

Calculating the Lyapunov exponent: The lyapunov exponent of the tent map can be easily calculated as due to this stretching behavior a small difference delta between two neighboring points will indeed grow exponentially by a factor of mu in each iteration. We thus can assume that:

delta_n = delta_0 * mu^n

We now only have to change the basis to e to obtain the exact formula that is used for the definition of the lyapunov exponent:

delta_n = delta_0 * e^(ln(mu) * n)

Therefore the lyapunov exponent of the tent map is:

lambda = ln(mu)

References:

Args:

x (float): starting point

steps (int): number of steps for which the generator should run

Kwargs:

mu (int): parameter mu that controls the behavior of the map

Returns:

generator object: the generator that creates the time series

1.3.3 Logistic map

nolds.**logistic_map**(*x*, *steps*, *r=4*)

Generates a time series of the logistic map.

Characteristics and Background: The logistic map is among the simplest examples for a time series that can exhibit chaotic behavior depending on the parameter r. For r between 2 and 3, the series quickly becomes static. At r=3 the first bifurcation point is reached after which the series starts to oscillate. Beginning with r = 3.6 it shows chaotic behavior with a few islands of stability until perfect chaos is achieved at r = 4.

Calculating the Lyapunov exponent: To calculate the "true" Lyapunov exponent of the logistic map, we first have to make a few observations for maps in general that are repeated applications of a function to a starting value.

If we have two starting values that differ by some infinitesimal $delta_0$ then according to the definition of the lyapunov exponent we will have an exponential divergence:

$$|\delta_n| = |\delta_0| e^{\lambda n}$$

We can now write that:

$$e^{\lambda n} = \lim_{\delta_0 -> 0} |\frac{\delta_n}{\delta_0}|$$

This is the definition of the derivative $\frac{dx_n}{dx_0}$ of a point x_n in the time series with respect to the starting point x_0 (or rather the absolute value of that derivative). Now we can use the fact that due to the definition of our map as repetitive application of some f we have:

$$f^{n\prime}(x) = f(f(f(...f(x_0)...))) = f'(x_n - 1) \cdot f'(x_n - 2) \cdot ... \cdot f'(x_0)$$

with

$$e^{\lambda n} = |f^{n\prime}(x)|$$

we now have

$$\begin{aligned} e^{\lambda n} &= |f'(x_n - 1) \cdot f'(x_n - 2) \cdot ... \cdot f'(x_0)| \\ &\Leftrightarrow \\ \lambda n &= \ln |f'(x_n - 1) \cdot f'(x_n - 2) \cdot ... \cdot f'(x_0)| \\ &\Leftrightarrow \\ \lambda &= \frac{1}{n} \ln |f'(x_n - 1) \cdot f'(x_n - 2) \cdot ... \cdot f'(x_0)| \\ &= \frac{1}{n} \sum_{k=0}^{n-1} \ln |f'(x_k)| \end{aligned}$$

With this sum we can now calculate the lyapunov exponent for any map. For the logistic map we simply have to calculate $f'(x)$ and as we have

$$f(x) = rx(1-x) = rx - rx$$

we now get

$$f'(x) = r - 2rx$$

References:

Args:

x (float): starting point

steps (int): number of steps for which the generator should run

Kwargs:

r (int): parameter r that controls the behavior of the map

Returns:

generator object: the generator that creates the time series

1.3.4 Fractional brownian motion

nolds.**fbm**(*n*, *H=0.75*)

Generates fractional brownian motions of desired length.

Author: Christian Thomae

References:

Args:

n (int): length of sequence to generate

Kwargs:

H (float): hurst parameter

Returns:

array of float: simulated fractional brownian motion

1.3.5 Fractional gaussian noise

nolds.**fgn**(*n*, *H=0.75*)

Generates fractional gaussian noise of desired length.

References:

Args:

n (int): length of sequence to generate

Kwargs:

H (float): hurst parameter

Returns:

array of float: simulated fractional gaussian noise

1.3.6 Quantum random numbers

nolds.**qrandom**(*n*)
Creates an array of n true random numbers obtained from the quantum random number generator at qrng.anu.edu.au

This function requires the package quantumrandom and an internet connection.

Args:

n (int): length of the random array

Return:

array of ints: array of truly random unsigned 16 bit int values

nolds.**load_qrandom**()
Loads a set of 10000 random numbers generated by qrandom.

This dataset can be used when you want to do some limited tests with "true" random data without an internet connection.

Returns:

int array the dataset

CHAPTER 2

Nolds examples

You can run some examples for the functions in nolds with the command `python -m nolds.examples <key>` where `<key>` can be one of the following:

- `lyapunov-logistic` shows a bifurcation plot of the logistic map and compares the true lyapunov exponent to the estimates obtained with `lyap_e` and `lyap_r`.
- `lyapunov-tent` shows the same plot as `lyapunov-logistic`, but for the tent map.
- `profiling` runs a profiling test with the package `cProfile`.
- `hurst-weron2` plots a reconstruction of figure 2 of the weron 2002 paper about the hurst exponent.
- `hurst-hist` plots a histogram of hurst exponents obtained for random noise.
- `hurst-nvals` creates a plot that compares the results of different choices for nvals for the function `hurst_rs`.

These tests are also available as functions inside the module `nolds.examples`.

2.1 Functions in `nolds.examples`

`nolds.examples.`**`plot_lyap`**(*maptype=u'logistic'*)

Plots a bifurcation plot of the given map and superimposes the true lyapunov exponent as well as the estimates of the largest lyapunov exponent obtained by `lyap_r` and `lyap_e`. The idea for this plot is taken from [ll].

This function requires the package `matplotlib`.

References:

Kwargs:

maptype (str): can be either `"logistic"` for the logistic map or `"tent"` for the tent map.

`nolds.examples.`**`profiling`**()

Runs a profiling test for the function `lyap_e` (mainly used for development)

This function requires the package `cProfile`.

nolds.examples.**weron_2002_figure2**(*n=10000*)

Recreates figure 2 of [w] comparing the reported values by Weron to the values obtained by the functions in this package.

The experiment consists of n iterations where the hurst exponent of randomly generated gaussian noise is calculated. This is done with differing sequence lengths of 256, 512, 1024, ..., 65536. The average estimated hurst exponent over all iterations is plotted for the following configurations:

- `weron` is the Anis-Lloyd-corrected Hurst exponent calculated by Weron
- `rs50` is the Anis-Lloyd-corrected Hurst exponent calculated by Nolds with the same parameters as used by Weron
- `weron_raw` is the uncorrected Hurst exponent calculated by Weron
- `rs50_raw` is the uncorrected Hurst exponent calculated by Nolds with the same parameters as used by Weron
- `rsn` is the Anis-Lloyd-corrected Hurst exponent calculated by Nolds with the default settings of Nolds

The values reported by Weron are only measured from the plot in the PDF version of the paper and can therefore have some small inaccuracies.

This function requires the package `matplotlib`.

References:

Kwargs:

n (int): number of iterations of the experiment (Weron used 10000, but this takes a while)

nolds.examples.**plot_hurst_hist**()

Plots a histogram of values obtained for the hurst exponent of uniformly distributed white noise.

This function requires the package `matplotlib`.

nolds.examples.**hurst_compare_nvals**(*data*, *nvals=None*)

Creates a plot that compares the results of different choices for nvals for the function hurst_rs.

Args:

data (array-like of float): the input data from which the hurst exponent should be estimated

Kwargs:

nvals (array of int): a manually selected value for the nvals parameter that should be plotted in comparison to the default choices

CHAPTER 3

Nolds Unittests

Nolds includes a set of unittests that can be run with `python -m unittest nolds.test_measures`. Some of these tests are based on random numbers and can therefore fail in rare cases.

Please note that running all tests may take a few minutes.

CHAPTER 4

Indices and tables

- genindex
- search

Bibliography

[lr_1] M. T. Rosenstein, J. J. Collins, and C. J. De Luca, "A practical method for calculating largest Lyapunov exponents from small data sets," Physica D: Nonlinear Phenomena, vol. 65, no. 1, pp. 117–134, 1993.

[lr_a] mirwais, "Largest Lyapunov Exponent with Rosenstein's Algorithm", url: http://www.mathworks.com/matlabcentral/fileexchange/38424-largest-lyapunov-exponent-with-rosenstein-s-algorithm

[lr_b] Shapour Mohammadi, "LYAPROSEN: MATLAB function to calculate Lyapunov exponent", url: https://ideas.repec.org/c/boc/bocode/t741502.html

[le_1] J. P. Eckmann, S. O. Kamphorst, D. Ruelle, and S. Ciliberto, "Liapunov exponents from time series," Physical Review A, vol. 34, no. 6, pp. 4971–4979, 1986.

[le_a] Manfred Füllsack, "Lyapunov exponent", url: http://systems-sciences.uni-graz.at/etextbook/sw2/lyapunov.html

[le_b] Steve SIU, Lyapunov Exponents Toolbox (LET), url: http://www.mathworks.com/matlabcentral/fileexchange/233-let/content/LET/findlyap.m

[le_c] Rainer Hegger, Holger Kantz, and Thomas Schreiber, TISEAN, url: http://www.mpipks-dresden.mpg.de/~tisean/Tisean_3.0.1/index.html

[se_1] J. S. Richman and J. R. Moorman, "Physiological time-series analysis using approximate entropy and sample entropy," American Journal of Physiology-Heart and Circulatory Physiology, vol. 278, no. 6, pp. H2039–H2049, 2000.

[se_a] "sample_entropy" function in R-package "pracma", url: https://cran.r-project.org/web/packages/pracma/pracma.pdf

[h_1] H. E. Hurst, "The problem of long-term storage in reservoirs," International Association of Scientific Hydrology. Bulletin, vol. 1, no. 3, pp. 13–27, 1956.

[h_2] H. E. Hurst, "A suggested statistical model of some time series which occur in nature," Nature, vol. 180, p. 494, 1957.

[h_3] R. Weron, "Estimating long-range dependence: finite sample properties and confidence intervals," Physica A: Statistical Mechanics and its Applications, vol. 312, no. 1, pp. 285–299, 2002.

[h_a] "hurst" function in R-package "pracma", url: https://cran.r-project.org/web/packages/pracma/pracma.pdf

Note: Pracma yields several estimates of the Hurst exponent, which are listed below. Unless otherwise stated they use the divisors of the length of the sequence as n. The length is reduced by at most 1% to find the value that has the most divisors.

- The "Simple R/S" estimate is just log((R/S)_n) / log(n) for n = N.
- The "theoretical Hurst exponent" is the value that would be expected of an uncorrected rescaled range approach for random noise of the size of the input data.
- The "empirical Hurst exponent" is the uncorrected Hurst exponent obtained by the rescaled range approach.
- The "corrected empirical Hurst exponent" is the Anis-Lloyd-Peters corrected Hurst exponent, but with sqrt(1/2 * pi * n) added to the (R/S)_n before the log.
- The "corrected R over S Hurst exponent" uses the R-function "lm" instead of pracmas own "polyfit" and uses n = N/2, N/4, N/8, ... by successively halving the subsequences (which means that some subsequences may be one element longer than others). In contrast to its name it does not use the Anis-Lloyd-Peters correction factor.

If you want to compare the output of pracma to the output of nolds, the "empirical hurst exponent" is the only measure that exactly corresponds to the Hurst measure implemented in nolds (by choosing corrected=False, fit="poly" and employing the same strategy for choosing n as the divisors of the (reduced) sequence length).

[h_b] Rafael Weron, "HURST: MATLAB function to compute the Hurst exponent using R/S Analysis", url: https://ideas.repec.org/c/wuu/hscode/m11003.html

Note: When the same values for nvals are used and fit is set to "poly", nolds yields exactly the same results as this implementation.

[h_c] Bill Davidson, "Hurst exponent", url: http://www.mathworks.com/matlabcentral/fileexchange/9842-hurst-exponent

[h_d] Tomaso Aste, "Generalized Hurst exponent", url: http://de.mathworks.com/matlabcentral/fileexchange/30076-generalized-hurst-exponent

[cd_1] P. Grassberger and I. Procaccia, "Characterization of strange attractors," Physical review letters, vol. 50, no. 5, p. 346, 1983.

[cd_2] P. Grassberger and I. Procaccia, "Measuring the strangeness of strange attractors," Physica D: Nonlinear Phenomena, vol. 9, no. 1, pp. 189–208, 1983.

[cd_3] P. Grassberger, "Grassberger-Procaccia algorithm," Scholarpedia, vol. 2, no. 5, p. 3043. urL: http://www.scholarpedia.org/article/Grassberger-Procaccia_algorithm

[cd_a] "corrDim" function in R package "fractal", url: https://cran.r-project.org/web/packages/fractal/fractal.pdf

[cd_b] Peng Yuehua, "Correlation dimension", url: http://de.mathworks.com/matlabcentral/fileexchange/24089-correlation-dimension

[dfa_1] C.-K. Peng, S. V. Buldyrev, S. Havlin, M. Simons, H. E. Stanley, and A. L. Goldberger, "Mosaic organization of DNA nucleotides," Physical Review E, vol. 49, no. 2, 1994.

[dfa_2] R. Hardstone, S.-S. Poil, G. Schiavone, R. Jansen, V. V. Nikulin, H. D. Mansvelder, and K. Linkenkaer-Hansen, "Detrended fluctuation analysis: A scale-free view on neuronal oscillations," Frontiers in Physiology, vol. 30, 2012.

[dfa_a] Peter Jurica, "Introduction to MDFA in Python", url: http://bsp.brain.riken.jp/~juricap/mdfa/mdfaintro.html

[dfa_b] JE Mietus, "dfa", url: https://www.physionet.org/physiotools/dfa/dfa-1.htm

[dfa_c] "DFA" function in R package "fractal"

[b7_a] Edgar Peters, "Chaos and Order in the Capital Markets: A New View of Cycles, Prices, and Market Volatility", Wiley: Hoboken, 2nd Edition, 1996.

[b7_b] Ian L. Kaplan, "Estimating the Hurst Exponent", url: http://www.bearcave.com/misl/misl_tech/wavelets/hurst/

[b7_c] HwB, “Pracma: brown72”, url: https://www.rdocumentation.org/packages/pracma/versions/1.9.9/topics/brown72

[tm_1] https://en.wikipedia.org/wiki/Tent_map

[lm_1] https://en.wikipedia.org/wiki/Tent_map

[lm_2] https://blog.abhranil.net/2015/05/15/lyapunov-exponent-of-the-logistic-map-mathematica-code/

[fbm_1] https://en.wikipedia.org/wiki/Fractional_Brownian_motion#Method_1_of_simulation

[fgn_1] https://en.wikipedia.org/wiki/Fractional_Brownian_motion

[ll] Manfred Füllsack, “Lyapunov exponent”, url: http://systems-sciences.uni-graz.at/etextbook/sw2/lyapunov.html

[w] R. Weron, “Estimating long-range dependence: finite sample properties and confidence intervals,” Physica A: Statistical Mechanics and its Applications, vol. 312, no. 1, pp. 285–299, 2002.

Index