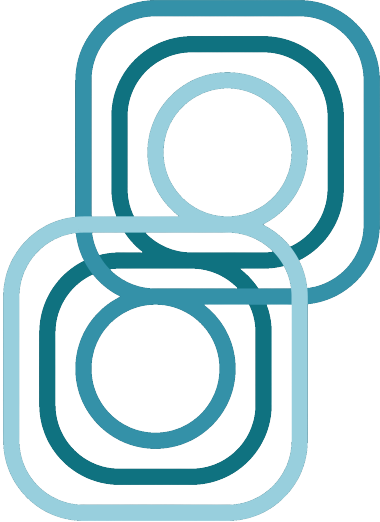
**EntropyHub Guide**

**v.0.0 (2021)**

**Preface**

Quote here from Savi / Boltzmann

Since the introduction of approximate entropy by Pincus three decades ago[[1]](#footnote-1), the use of information theoretic entropy measures to estimate the complexity, randomness or regularity of time series data has become ubiquitous in many research domains. Applications of entropy are ever-increasing (Fig. 1), as are the number of newly derived entropies that seek to improve on previously established methods, with less sensitivity to signal length, amplitude fluctuations, etc. (see Ribiero *et al*. [[2]](#footnote-2)).

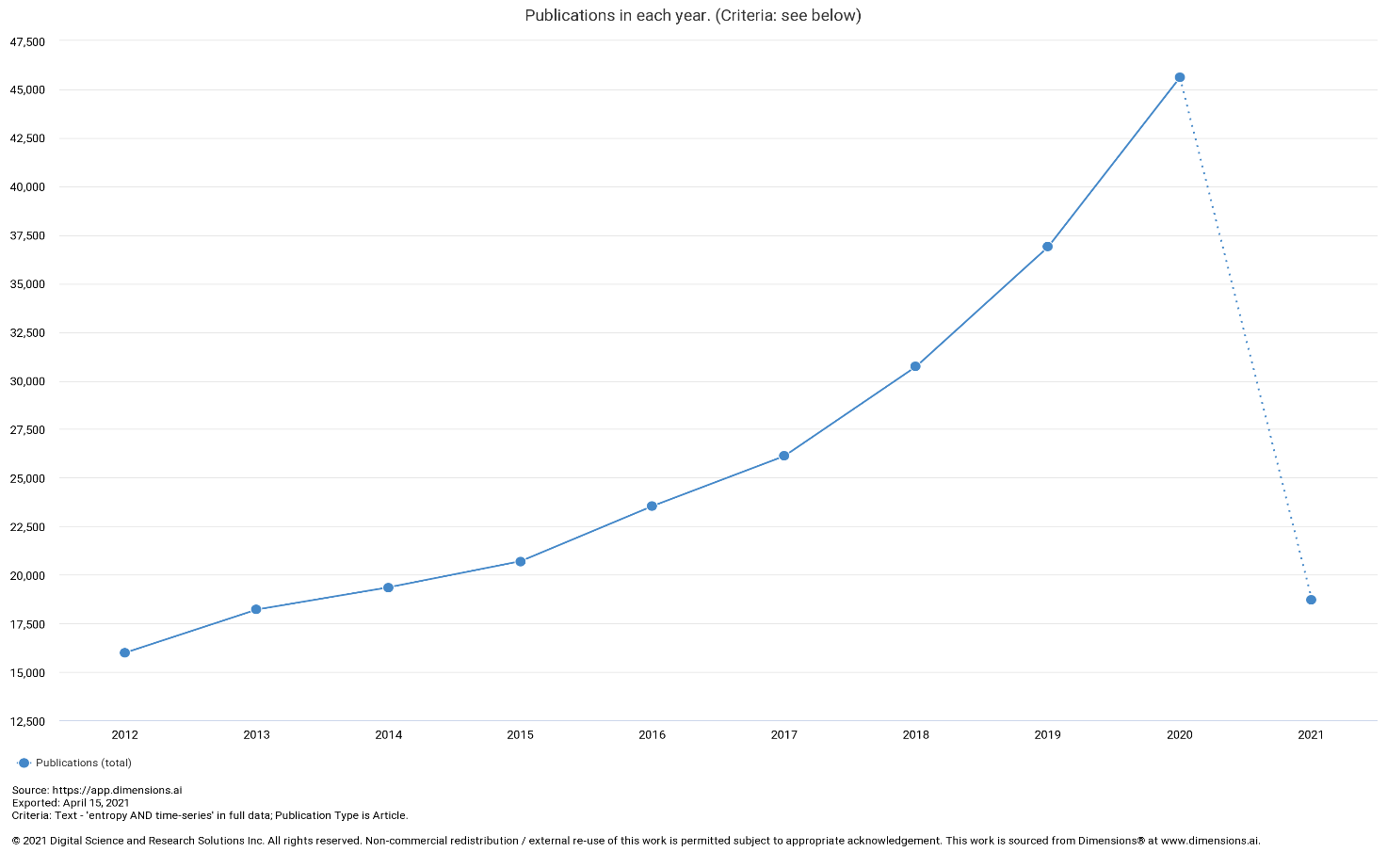


Figure 1.

Although many functions for estimating these entropies can be found separately online, there is currently no toolkit to perform entropic time-series analysis at the command line with reliable code, extensive documentation and consistent syntax, that is also accessible in multiple programming languages. Hence, the goal of EntropyHub is to integrate the many established entropy methods into one package that is available for users of Python, MatLab and Julia.

EntropyHub features multiscale variants of all base and cross-entropy methods, (including composite, refined and hierarchical multiscale approaches), in addition to bidimensional entropies for 2D matrix analysis. As the scientific community develops novel entropic measures, efforts will be made to incorporate them in later versions of the package.

EntropyHub is licensed under the Apache License (Version 2.0) and is free to use by all on condition that the following reference be cited on any outputs realized using the software:

Matthew W. Flood and Bernd Grimm,   
*EntropyHub: an Open-Source Toolkit for Entropic Time Series Analysis*,   
2021 (https://github.com/MattWillFlood/EntropyHub)

If you find this package useful, please consider starring it on [GitHub](https://github.com/MattWillFlood/EntropyHub/), MatLab File Exchange, PyPI or Julia Packages. This helps us to gauge user satisfaction.

Thanks,

Matt

[entropyhubproject@gmail.com](mailto:entropyhubproject@gmail.com)

The EntropyHub License and Terms of Use are available at: https://github.com/MattWillFlood/EntropyHub

**Contents**

1. **Introduction**

It is important to clarify at the outset that the term ‘[entropy](http://www.scholarpedia.org/article/Entropy)’ henceforth described refers to entropy in the context of probability theory and information theory as defined by Shannon[[3]](#footnote-3), and **not** thermodynamic or other entropies from classical physics.

EntropyHub functions fall into five categories:

* **Base** – functions for estimating the entropy of a single univariate time series.
* **Cross** – functions for estimating the entropy between two univariate time series.
* **Bidimensional** – functions for estimating the entropy of a two-dimensional univariate matrix.
* **Multiscale** – functions for estimating the multiscale entropy of a single univariate time series using any of the ***Base*** entropy functions.
* **Multiscale Cross** – functions for estimating the multiscale entropy between two univariate time series using any of the ***Cross-entropy*** functions.

[See Table 1 for a list of all functions]

While each function has its own unique keyword arguments, there are several keyword arguments (also known as *Name/Value* pairs in MatLab) that are common to most Base, Cross and Bidimensional entropies. These are:

* ***m*** embedding dimension
* ***tau*** time delay
* ***Logx*** base of the logarithm in Shannon’s formula for entropy.

This argument allows the entropy to be estimated in bits (base 2), nats (base *e*), dits (base 10), or whatever the user specifies.

* ***Norm*** normalisation of the entropy value as outlined in the source literature for that particular function

All keyword arguments for Multiscale and Multiscale Cross-entropy functions are identical.

One of the advantages of EntropyHub is the variety of keyword arguments available for many functions. For example, by specifying the *Typex* keyword argument when calling *PermEn*, one can calculate the edge, weighted, modified, amplitude-aware, fine-grained or UNIQUANT version of permutation entropy, in addition to the original defined by Bandt and Pope [3]. Similarly, one can employ different fuzzy functions to transform the state vector distances when calculating fuzzy entropy (*FuzzEn*) by specifying the *Fx* keyword argument. This ability to augment various parameters at the command line enables more advanced entropy methods to be called with no effort.

Although each function is complete with default arguments, no time series should be analysed without identifying the appropriate parameters that capture the underlying dynamics.

Each function has a helpful description of its usage in the docstrings, explaining input parameters, outputs values and the relevant source literature. To read the docstrings of a particular function, type:

(MatLab) help *function\_name* e.g. help PermEn  
(Python) help(*EntropyHub*.*function\_name)* e.g. help EntropyHub.PermEn  
(Julia) ? *function\_name* e.g. ? PermEn()

While the majority of multiscale and multiscale-cross functions available through EntropyHub have been previously published, options are available to call new multiscale variants, such as multiscale cross-spectral entropy.

|  |  |  |  |
| --- | --- | --- | --- |
| **Base Entropy** | ***Function*** | **Cross-Entropy** | ***Function*** |
| Approximate Entropy | *ApEn* | Cross Sample Entropy | *XSampEn* |
| Sample Entropy | *SampEn* | Cross Approximate Entropy | *XApEn* |
| Fuzzy Entropy | *FuzzEn* | Cross Fuzzy Entropy | *XFuzzEn* |
| Kolmogorov Entropy | *K2En* | Cross Permutation Entropy | *XPermEn* |
| Permutation Entropy | *PermEn* | Cross Conditional Entropy | *XCondEn* |
| Conditional Entropy | *CondEn* | Cross Distribution Entropy | *XDistEn* |
| Distribution Entropy | *DistEn* | Cross Spectral Entropy | *XSpecEn* |
| Spectral Entropy | *SpecEn* | Cross Kolmogorov Entropy | *XK2En* |
| Dispersion Entropy | *DispEn* |  |  |
| Symbolic Dynamic Entropy | *SyDyEn* |  |  |
| Increment Entropy | *IncrEn* | **Bidimensional Entropy** | ***Function*** |
| Cosine Similarity Entropy | *CoSiEn* | Bidimensional Sample Entropy | *SampEn2D* |
| Phase Entropy | *PhasEn* | Bidimensional Fuzzy Entropy | *FuzzEn2D* |
| Slope Entropy | *SlopEn* | Bidimensional Distribution Entropy | *DistEn2D* |
| Bubble Entropy | *BubbEn* |  |  |
| Grid Distribution Entropy | *GridEn* |  |  |
| Entropy of Entropy | *EnofEn* |  |  |
| Attention Entropy | *AttnEn* |  |  |
|  |  |  |  |
| **Multiscale Entropy** | ***Function*** | **Multiscale Cross-Entropy** | ***Function*** |
| Multiscale Entropy | *MSEn* | Multiscale Cross-Entropy | *XMSEn* |
| Composite Multiscale Entropy (+ Refined-Composite Multiscale Entropy | *CMSEn* | Composite Multiscale Cross-Entropy (+ Refined-Composite Multiscale Cross-Entropy | *cXMSEn* |
|  |  |
| Refined Multiscale Entropy | *rMSEn* | Refined Multiscale Cross-Entropy | *rXMSEn* |
| Hierarchical Multiscale Entropy | *hMSEn* | Hierarchical Multiscale Cross-Entropy | *hXMSEn* |

**Let’s get started!**

1. **Installation**

Stable releases of EntropyHub are available from the default package manager for MatLab, Python and Julia, while the latest version of EntropyHub can be downloaded or cloned from the [GitHub](https://github.com/MattWillFlood/EntropyHub/) repository.

* 1. **MatLab**There are 3 ways to install EntropyHub for Matlab. Method 1 is the most straightforward.

**Method 1.**

1. In MatLab, click the ‘Add-Ons’ button in the HOME tab. This should open the MatLab Add-On Explorer.
2. In the Add-On Explorer, search for ‘EntropyHub’.  
   **Figure 2 here – screenshot of EntropyHub page in explorer.**
3. In the top right corner, click the ‘Add’ button.  
   You may be asked to accept or decline the License Agreement to installation.

**Method 2.**

1. Visit the EntropyHub File Exchange page.   
   Note: you need to be logged in to your MathWorks account to continue.
2. Download the toolbox file (EntropyHub.mltbx) by clicking ‘Toolbox’ in the dropdown menu under the ‘Download’ button on the right hand side.  
   **Figure 3 here – screenshot of EntropyHub page dropdown box.**
3. With MatLab running, navigate to the *EntropyHub.mltbx* file in your system explorer and open it.  
   You may be asked to accept or decline the License Agreement to installation.

**Method 3.**

1. Go to the MatLab folder in the EntropyHub Github repository.
2. Download the toolbox file (EntropyHub.mltbx) by right clicking the link and selecting ‘Save Link As…’ (for Windows users).

**Figure 4 here – screenshot of right click saving.**

1. With MatLab running, navigate to the *EntropyHub.mltbx* file in your system explorer and open it.  
   You may be asked to accept or decline the License Agreement to installation.
   1. **Python**

There are 2 ways to install EntropyHub for Python. Method 1 is the most straightforward and is strongly recommended.

**Method 1.**

1. Python comes with an inbuilt package management system, *pip*. Pip can install, update, or delete any official package.

You can install packages via the command line by entering:

**Figure 5 here – screenshot of pip install command.**If using a Python IDE, it is recommended to restart after installing.

1. To use EntropyHub, import the module with the following command:

**Figure 6 here – screenshot of import command with and without abbreviation**

**Method 2.**\*Note: installation with Method 2 requires the latest version of ‘wheel’ to be previously installed in Python.

1. Visit the EntropyHub PyPI page and download the tar zip folder file from the ‘download files’ button on the left-hand side.  
   **Figure 7 here – screenshot of PyPI page**

Extract the files into a local directory.

1. Open a command or terminal window and navigate to the root directory where setup.py is located.
2. In the command line, enter: python setup.py install  
   \*Ensure that an up-to-date version of setuptools is installed:  
   python -m pip install --upgrade setuptools
   1. **Julia**There are 2 ways to install EntropyHub in Julia. Method 1 is recommended.

**Method 1.**

1. In Julia, enter the following:

] add EntropyHub

Alternatively,

using Pkg; Pkg.add("EntropyHub")

1. To use EntropyHub in julia, type:

using EntropyHub

or using an abbreviation:

using EntropyHub as EH

or import specific functions only:

using EntropyHub: SampEn, MSobject, MSEn

**Method 2.**

1. In Julia, enter the following:

] add https://github.com/MattWillFlood/EntropyHub/EntropyHub%20-%20Julia.git

1. To use EntropyHub in julia, type:

using EntropyHub

or using an abbreviation:

using EntropyHub as EH

or import specific functions only:

using EntropyHub: SampEn, MSobject, MSEn

1. **Functions**

Sections 3.1 – 3.5 outline the command line syntax of each function with descriptions of every argument and returned value. ~~Directly under each sub-heading is the function command.~~ **~~Unless otherwise stated~~**~~, the first command corresponds to MatLab syntax and the second to Python~~ **~~and~~** ~~Julia syntax.~~

**Numpy (np) before argumatent values in python text below**

**Note: For concision, function commands written below in the Python and ~~Julia~~ format exclude the module prefix which would otherwise be required, i.e. EntropyHub.SampEn() is written as SampEn().**

* 1. **Base Entropy Functions**
     1. **ApEn: Approximate Entropy**

[Ap, Phi] = ApEn(Sig, ‘m’, 2, ‘tau’, 1, ‘r’, 0.2\*std(Sig), ‘Logx’, exp(1))

Ap, Phi = ApEn(Sig, m = 2, tau = 1, r = 0.2\*np.std(Sig), Logx = np.exp(1))

Ap, Phi = ApEn(Sig, m = 2, tau = 1, r = 0.2\*std(Sig), Logx = exp(1))

Arguments

*Sig* - Time series signal, a vector of length > 10

*m* - Embedding Dimension, an integer > 1.

*tau* - Time Delay, a positive integer.

*r* - Radius threshold value, a positive scalar.

*Logx* - Logarithm base in Shannon’s entropy formula, a positive scalar.

Output

*Ap* - Approximate entropy estimates, a vector of length m+1.  
 \*\*The first value of Ap is the zeroth estimate, i.e. Log(N)/N – Phi(1),

and the last value of Ap is the estimate for the specified *m*.

*Phi*  - The number of matched state vectors for each embedding dimension from 0 – m+1.

References: [1]

* + 1. **SampEn: Sample Entropy**

[Samp, A, B] = SampEn(Sig, ‘m’, 2, ‘tau’, 1, ‘r’, 0.2\*std(Sig), ‘Logx’, exp(1))

Samp, A, B = SampEn(Sig, m = 2, tau = 1, r = 0.2\*np.std(Sig), Logx = np.exp(1))

Samp, A, B = SampEn(Sig, m = 2, tau = 1, r = 0.2\*std(Sig), Logx = exp(1))

Arguments

*Sig* - Time series signal, a vector of length > 10

*m* - Embedding Dimension, an integer > 1.

*tau* - Time Delay, a positive integer.

*r* - Radius threshold value, a positive scalar.

*Logx* - Logarithm base in Shannon’s entropy formula, a positive scalar.

Output

*Samp* - Sample entropy estimates, a vector of length m+1.  
 \*\*The first value of Samp is the zeroth estimate,   
 i.e. Log(N\*(N-1)/2) – Log(A(1)), and the last value of Samp is the  
 estimate for the specified *m*.

*A*  - The number of matched state vectors for each embedding dimension from 0 – m.

*B*  - The number of matched state vectors for each embedding dimension from 1 – m+1.

References: [2]

* + 1. **FuzzEn: Fuzzy Entropy**

[Fuzz, Ps1, Ps2] = FuzzEn(Sig, ‘m’, 2, ‘tau’, 1, ‘Fx’, ‘default, ‘r’, (0.2, 2), ‘Logx’, exp(1))

Fuzz, Ps1, Ps2 = FuzzEn(Sig, m = 2, tau = 1, Fx = “default”, r = (0.2, 2), Logx = np.exp(1))

Fuzz, Ps1, Ps2 = FuzzEn(Sig, m = 2, tau = 1, Fx = “default”, r = (0.2, 2), Logx = exp(1))

Arguments

*Sig* - Time series signal, a vector of length > 10

*m* - Embedding Dimension, an integer > 1.

*tau* - Time Delay, a positive integer.

*Fx* - Type of fuzzy function for, one of the following:

Sigmoid

Modsampen

Default

Gudermannian

Linear  
 **Figures here of each method**

*r* - Parameters of the fuzzy function given by Fx, a 1 element scalar or a 2 element tuple of positive values. The `r` parameters for each fuzzy

function are defined as follows:

sigmoid: r(1) = divisor of the exponential argument

r(2) = value subtracted from argument (pre-division)

modsampen: same as sigmoid

default: r(1) = divisor of the exponential argument

r(2) = argument exponent (pre-division)

gudermannian: r = a scalar whose value is the numerator of

argument to gudermannian function:

GD(x) = atan(tanh(r/x))

linear: r = an integer value. When r = 0, the argument of the exponential function is normalised between [0 1]. When r = 1,

the minimum value of the exponential argument is set to 0.

*Logx* - Logarithm base in Shannon’s entropy formula, a positive scalar.

Output

*Fuzz* - Fuzzy entropy estimates for each embedding dimension from 1:m,

a vector of length m.  
*Ps1*  - The average fuzzy distances for embedding dimensions 1:m.

*Ps12*  - The average fuzzy distances for embedding dimensions 2:m+1.

References: [3], [4], [5], [6], [7], [8]

* + 1. **K2En: Kolmogorov Entropy**

[K2, Ci] = K2En(Sig, ‘m’, 2, ‘tau’, 1, ‘r’, 0.2\*std(Sig), ‘Logx’, exp(1))

K2, Ci = K2En(Sig, m = 2, tau = 1, r = 0.2\*np.std(Sig), Logx = np.exp(1))

K2, Ci = K2En(Sig, m = 2, tau = 1, r = 0.2\*std(Sig), Logx = exp(1))

Arguments

*Sig* - Time series signal, a vector of length > 10

*m* - Embedding Dimension, an integer > 1.

*tau* - Time Delay, a positive integer.

*r* - Radius threshold value, a positive scalar.

*Logx* - Logarithm base in Shannon’s entropy formula, a positive scalar.

Output

*K2* - Kolmogorov entropy estimates estimates for each embedding dimension from 1:m.  
*Ci*  - The correlation integral/sum for each embedding dimension from 1:m.

References: [9]

* + 1. **PermEn: Permutation Entropy**

[Perm, Pnorm, cPE] = PermEn(Sig, ‘m’, 2, ‘tau’, 1, ‘Typex’, ’none’, ‘tpx’, [], ‘Logx’, 2, ‘Norm’, true)

Perm, Pnorm, cPE = PermEn(Sig, m = 2, tau = 1, Typex = ‘none’, tpx = -1, Norm = True ,Logx = 2)

Perm, Pnorm, cPE = PermEn(Sig, m = 2, tau = 1, Typex = “none”, tpx = nothing, Norm = true, Logx = 2)

Arguments

*Sig* - Time series signal, a vector of length > 10

*m* - Embedding Dimension, an integer > 1.

*tau* - Time Delay, a positive integer.

*Typex* - Variant of permutation entropy, one of the following strings:

*'uniquant'* uniform quantization [11]

*'finegrain'* fine-grained permutation entropy [12]

*'modified'* modified permutation entropy [13]

*'ampaware'* amplitude-aware permutation entropy [14]

*'weighted'* weighted permutation entropy [15]

*'edge'* edge permutation entropy [16]

**More descriptions of these**

*tpx* - Tuning parameter for the permutation entropy specified by Typex.

[uniquant] 'tpx' is the L parameter, an integer > 1 (default = 4).

[finegrain] 'tpx' is the alpha parameter, a positive scalar (default = 1)

[ampaware] 'tpx' is the A parameter, a value in range [0 1] (default = 0.5)

[edge] 'tpx' is the r sensitivity parameter, a scalar > 0 (default = 1)

Describe these in more detail

*Norm* - Normalisation of Perm value, a boolean operator:

When True - normalises w.r.t log(# of permutation symbols [m]) - default

When false - normalises w.r.t log(# of all possible permutations [m!])

\* Note: Normalised permutation entropy is undefined for m = 1.

\*\* Note: When Typex = *uniquant* and Norm = true, normalisation

is calculated w.r.t. log(tpx^m)

*Logx* - Logarithm base in Shannon’s entropy formula, a positive scalar.

Output

*Perm* - Permutation entropy estimates for embedding dimensions 1:m.  
*Pnorm*  - Normalised Permutation entropy estimates.

*cPE - Conditional permutation entropy – reference here [17]*

References: [10], [11], [12], [13], [14], [15], [16], [17]

* + 1. **CondEn: Conditional Entropy**

[Cond, SEw, SEz] = CondEn(Sig, ‘m’, 2, ‘tau’, 1, ‘c’, 6, ‘Logx’, exp(1), ‘Norm’, false)

Cond, SEw, SEz = CondEn(Sig, m = 2, tau = 1, c = 6, Logx = np.exp(1), Norm = False)

Cond, SEw, SEz = CondEn(Sig, m = 2, tau = 1, c = 6, Logx = exp(1), Norm = false)

Arguments

*Sig* - Time series signal, a vector of length > 10

*m* - Embedding Dimension, an integer > 1.

*tau* - Time Delay, a positive integer.

*c* - number of symbols in symbolic transformation, in integer > 1

*Logx* - Logarithm base in Shannon’s entropy formula, a positive scalar.

*Norm* - Normalisation of Cond value:

[false] no normalisation - default

[true] normalises w.r.t Shannon entropy of data sequence `Sig`

Output

*Cond* - Corrected conditional entropy estimate

*SEw* - Shannon entropy estimate for m

*SEz* - Shannon entropy estimate for m+1

References: [18], [19]

* + 1. **DistEn: Distribution Entropy**

[Dist, Ppi] = DistEn(Sig, ‘m’, 2, ‘tau’, 1, ‘Bins’, ‘sturges’, ‘Logx’, 2, ‘Norm’, true)

Dist, Ppi = DistEn(Sig, m = 2, tau = 1, Bins = ‘sturges’, Logx = 2, Norm = True)

Dist, Ppi = DistEn(Sig, m = 2, tau = 1, Bins = “sturges”, Logx = 2, Norm = true)

Arguments

*Sig* - Time series signal, a vector of length > 10

*m* - Embedding Dimension, an integer > 1.

*tau* - Time Delay, a positive integer.

*Bins* - Histogram binning method, in integer > 1 indicating the number of bins, or one of the following strings

{'sturges','sqrt','rice','doanes'} [default: 'sturges']

*Logx* - Logarithm base in Shannon’s entropy formula, a positive scalar.

(Enter 0 for natural logarithm)

*Norm* - Normalisation of Dist value:

[false] no normalisation

[true] normalises w.r.t number of bins (default)

Output

*Dist* - Distribution entropy estimate

*Ppi* - Probability of each histogram bin

References [20]

* + 1. **SpecEn: Spectral Entropy**

[Spec, BandEn] = SpecEn(Sig, ‘N’, 2\*length(Sig)+1, ‘Freqs’, [0,1], ‘Logx’, exp(1), ‘Norm’, true)

Spec, BandEn = SpecEn(Sig, N = 2\*len(Sig) + 1, ‘Freqs’, (0,1), Logx = np.exp(1), Norm = True)

Spec, BandEn = SpecEn(Sig, N = 2\*length(Sig) + 1, Freqs = (0,1), Logx = exp(1), Norm = true)

Arguments

*Sig* - Time series signal, a vector of length > 10

*N* - Resolution of the N-point fft, an integer > 1.

*Freqs* - Normalised band edge-frequencies for calculating the band entropy (BandEn), a 2 element tuple with values in range [0,1] where 1 is the Nyquist frequency. **Figure to demonstrate**

\* When no edge frequencies are provided, BandEn==SpecEn

*Logx* - Logarithm base in Shannon’s entropy formula, a positive scalar.

*Norm* - Normalisation of Spec value:

[false] no normalisation

[true] normalises Spec w.r.t number of Nyquist frequency value, and BandEn w.r.t. range of frequencies in the band given by Freqs.

Output

*Spec* - Spectral entropy estimate

*BandEn* - Spectral Band entropy estimate

References [21], [22]

* + 1. **DispEn: Dispersion Entropy**

[Dispx, RDE] = DispEn(Sig, ‘m’, 2, ‘tau’, 1, ‘Typex’, ‘ncdf’, ‘Logx’, exp(1), ‘Fluct’, false, ‘Norm’, false, ‘rho’, 1)

Dispx, RDE = DispEn(Sig, m = 2, tau = 1, Typex = ‘ncdf’, Logx = exp(1), Fluct = False, Norm = False, rho = 1)

Dispx, RDE = DispEn(Sig, m = 2, tau = 1, Typex = “ncdf”, Logx = exp(1), Fluct = false, Norm = false, rho = 1)

Arguments

*Sig* - Time series signal, a vector of length > 10

*m* - Embedding Dimension, an integer > 1.

*tau* - Time Delay, a positive integer.

*Typex* - Type of symbolic sequence transform, one of the following strings:

"linear": ….

"kmeans": ….

"ncdf": …

"finesort" …

"equal" …

*Logx* - Logarithm base in Shannon’s entropy formula, a positive scalar.

*Fluct -* When true, returns the fluctuation-based dispersion entropy

*Norm* - Normalisation of Dispx value:

[false] no normalisation

[true] normalises w.r.t number of bins (default)

*rho*  - \*If Typex = 'finesort', rho is the tuning parameter, a positive scalar (default = 1)

Output

*Dispx* - Dispersion entropy estimate

*RDE* - Reverse Dispersion entropy estimate

References [23], [24], [25], [26]

* 1. **Cross-Entropy Functions**
  2. **Multiscale Entropy Functions**
  3. **Multiscale Cross-Entropy Functions**
  4. **Bidimensional Entropy Functions  
       
     put note in textbox about locking matrix size**

1. **Tutorial**

1. Steven M. Pincus, *Approximate entropy as a measure of system complexity*

   Proceedings of the National Academy of Sciences (1991); 88.6: 2297-2301. [↑](#footnote-ref-1)
2. Ribeiro M, Henriques T, Castro L, Souto A, Antunes L, Costa-Santos C, Teixeira A.,

   *The Entropy Universe*,

   Entropy (2021); 23(2):222. [↑](#footnote-ref-2)
3. Claude E. Shannon,   
   *A Mathematical Theory of Communication* Bell System Technical Journal (1948), 27 (3): 379–423. [↑](#footnote-ref-3)