

Entropy and Lorenz Model

December 2021

1 Entropy

1.1 Theory svd entropy

We can find online the documentation of the function svd-entropy [¹].

$$S_{svd} = S(\mathbf{x}(t), order, delay) \quad (1)$$

We denote with \mathbf{x} the dataset that we want to study.

$$\begin{aligned} \mathbf{x}(t_0) &= \mathbf{x}_0, \\ \mathbf{x}(t) &= (x_1, x_2, x_3, \dots, x_N) \end{aligned} \quad (2)$$

This function creates a matrix Y with the dataset.

$$\mathbf{y}(i) = (x_i, x_{i+delay}, \dots, x_{i+(order-1)delay}) \quad (3)$$

$$Y = \begin{pmatrix} \mathbf{y}(1) \\ \mathbf{y}(2) \\ \vdots \\ \mathbf{y}(N - (ord - 1)del) \end{pmatrix}$$

If we use $order = 3$ and $delay = 1$ we obtain the following matrix:

$$Y = \begin{pmatrix} x_1 & x_2 & x_3 \\ x_2 & x_3 & x_4 \\ \vdots & \vdots & \vdots \\ \vdots & \vdots & \vdots \\ \vdots & x_{N-1} & x_N \end{pmatrix}$$

The next step is to write this matrix using the SVD decomposition.

$$Y = U\Sigma V^* \quad (4)$$

¹https://raphaelvallat.com/entropy/build/html/generated/entropy.svd_entropy.html

Where U is an unitary matrix, V is also an unitary matrix, and Σ is a diagonal matrix. Now we denotes the eigenvalues of the matrix Σ with σ_i . Now we can compute the average eigenvalues:

$$\bar{\sigma}_k = \frac{\sigma_k}{\sum_j^M \sigma_j} \quad (5)$$

Where M is the number of eigenvalues.

After this we can compute the SVD Entropy:

$$S_{svd} = - \sum_k^M \bar{\sigma}_k \log_2(\bar{\sigma}_k) \quad (6)$$

1.2 Fisher Information

We can define the Fisher Information in the following way^[2]:

$$I_F = \sum_k^{M-1} \frac{(\bar{\sigma}_{k+1} - \bar{\sigma}_k)^2}{\bar{\sigma}_{k-1}} \quad (7)$$

The quantities are exactly the same that we have defined in the previous section.

1.3 Shannon Entropy

We analyze the Shannon Entropy function ^[3].

```

1 def shannon_entropy(time_series):
2     """Return the Shannon Entropy of the sample data.
3     Args:
4         time_series: Vector or string of the sample data
5     Returns:
6         The Shannon Entropy as float value
7     """
8
9     # Check if string
10    if not isinstance(time_series, str):
11        time_series = list(time_series)
12
13    # Create a frequency data
14    data_set = list(set(time_series))
15    freq_list = []
16    for entry in data_set:
17        counter = 0.
18        for i in time_series:
19            if i == entry:
20                counter += 1
21        freq_list.append(float(counter) / len(time_series))
22
23    # Shannon entropy

```

²for reference see here:<https://www.mdpi.com/1099-4300/23/11/1424>

³<https://github.com/nikdon/pyEntropy/blob/master/pyentrp/entropy.py>

```

24     ent = 0.0
25     for freq in freq_list:
26         ent += freq * np.log2(freq)
27     ent = -ent
28     return ent

```

$$S_{sh} = S_{sh}(\mathbf{x}) \quad (8)$$

Here we find the definition of Shannon Entropy:

$$S_{sh} = - \sum_i P(x_i) \log_2(P(x_i)) \quad (9)$$

Where $P(x_i)$ is the frequency of x_i in the dataset \mathbf{x} .

1.3.1 Technical Issues

Lorenz's Orbit trajectory is a timeseries with 'float64' numbers.

A classic Shannon Entropy algorithm has to compute the relative frequency of each number in the array.

The issue is that because of this type of vector is almost impossible to find two identical numbers in the timeseries.

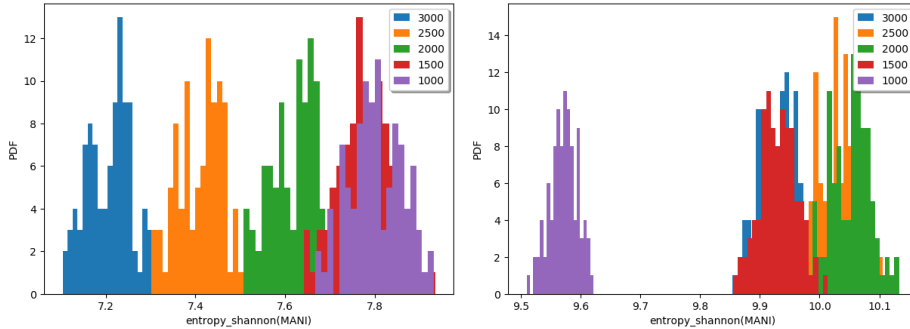
Indeed the function "torch.load()" ⁴ creates a vector of 'float64' arrays with a precision of 16 decimals. That's why to compute entropy is useful to use another definition of equality.

$$\begin{aligned} x(t_a) &= a_0, a_1 a_2 \dots a_{16} \\ x(t_b) &= b_0, b_1 b_2 \dots b_{16} \end{aligned} \quad (10)$$

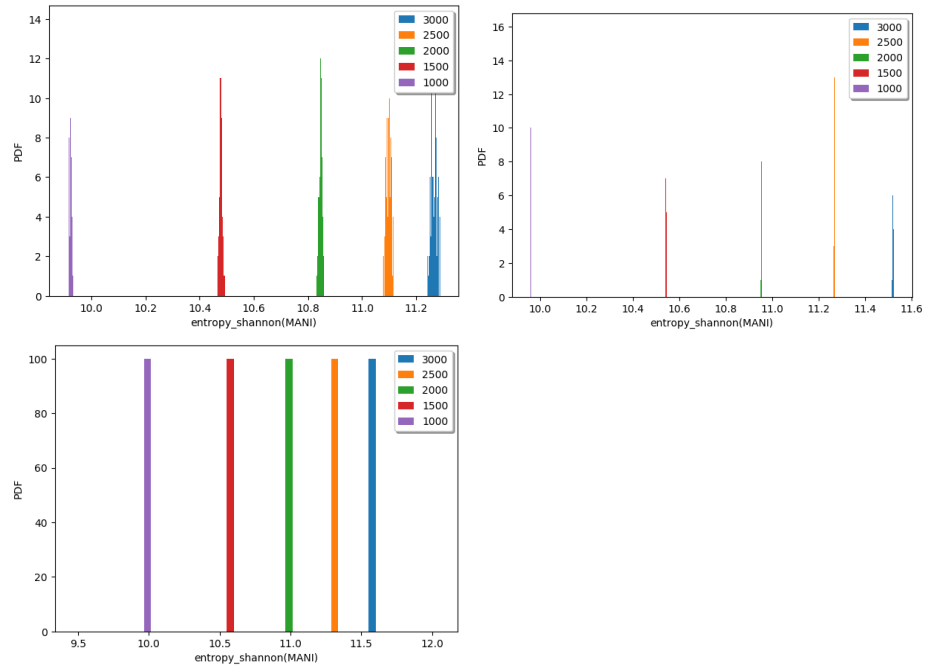
We say that $x(t_a) = x(t_b)$ with precision p if :

$$a_0 = b_0, a_1 = b_1, \dots, a_p = b_p \quad (11)$$

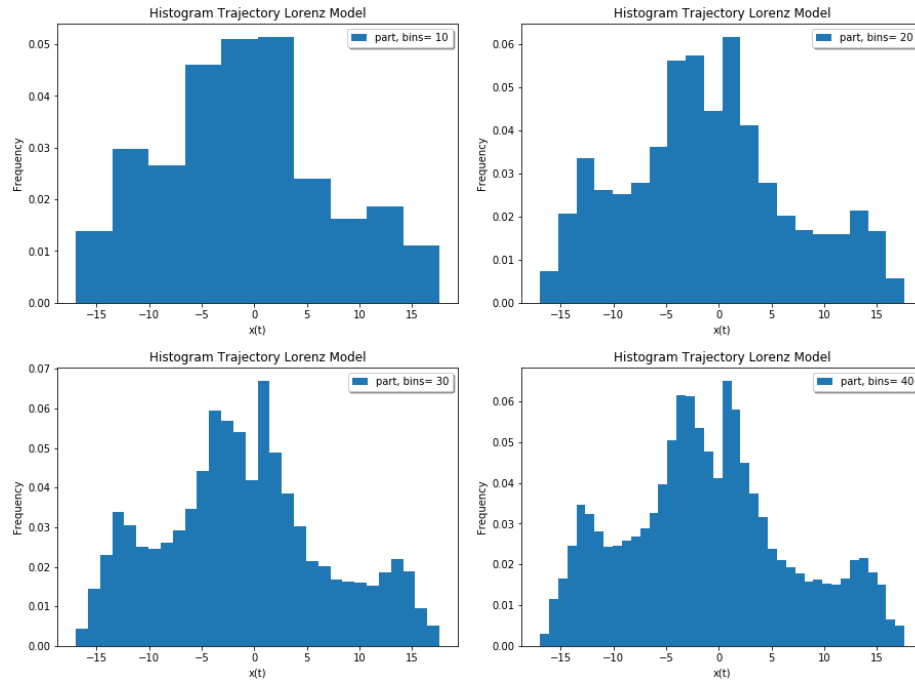
Following this idea we can make different plots of Shannon Entropy of the trajectories close to fixed points. With $p = 1, 2, 3, 4, 16$

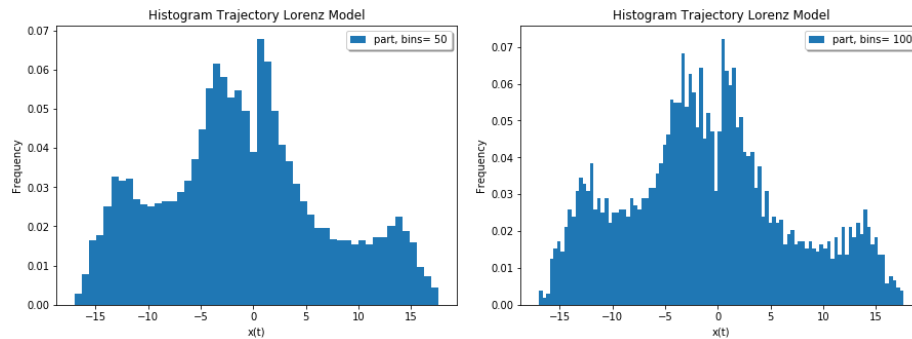


⁴Used to analyze the entropy of the trajectories



1.3.2 histograms





1.4 Approximate Entropy

```

1 def app_entropy(x, order=2, metric='chebyshev'):
2
3     phi = _app_samp_entropy(x, order=order, metric=metric,
4                             approximate=True) # it defines the vector phi
5     return np.subtract(phi[0], phi[1])
6
7 def _app_samp_entropy(x, order, metric='chebyshev', approximate=
8     True):
9     """Utility function for 'app_entropy' and 'sample_entropy'.
10
11     #technical stuff
12
13     _all_metrics = KDTree.valid_metrics
14     if metric not in _all_metrics:
15         raise ValueError('The given metric (%s) is not valid. The
16         valid '
17
18         'metric names are: %s' % (metric,
19         _all_metrics))
20
21     phi = np.zeros(2) #vector phi
22     r = 0.2 * np.std(x, ddof=0) #radius to define the concept of
23     neighbours (0.2* standard deviation with 0 DF of the
24     timeseries)
25
26     # compute phi(order, r)
27     _emb_data1 = _embed(x, order, 1) #creates a matrix with the
28     timeseries Nx2
29     if approximate: # TRUE
30         emb_data1 = _emb_data1
31     else:
32         emb_data1 = _emb_data1[:-1]
33     count1 = KDTree(emb_data1, metric=metric).query_radius(
34         emb_data1, r,
35
36         count_only=True
37
38         ).astype
39     (np.float64)

```

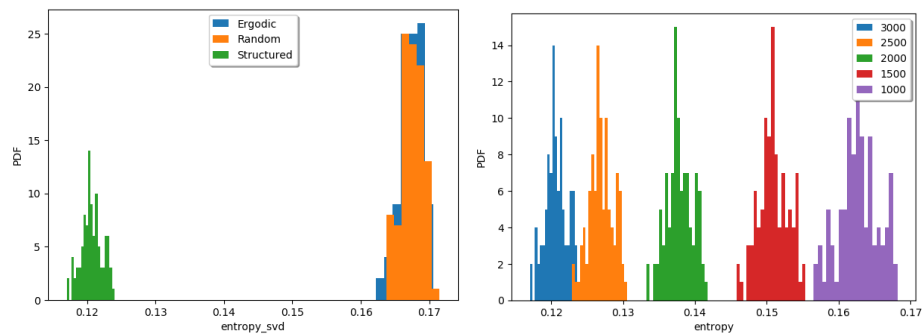
```

30                                     #
    compute phi(order + 1, r)
31 emb_data2 = _embed(x, order + 1, 1) # same matrix but Nx3
32 count2 = KDTree(emb_data2, metric=metric).query_radius(emb_data2, r
    ,
33                                     count_only=
    True
34                                     ).astype(np.
    float64) # count the number of neighbours according
    to Chebyshev norm for
35
    # each point and put this number in a
    vector.
36
37 if approximate: #true
    phi[0] = np.mean(np.log(count1 / emb_data1.shape[0]))
    phi[1] = np.mean(np.log(count2 / emb_data2.shape[0]))
38
39 else:
40     phi[0] = np.mean((count1 - 1) / (emb_data1.shape[0] - 1))
41     phi[1] = np.mean((count2 - 1) / (emb_data2.shape[0] - 1))
42
43 return phi
44
45
46
47
48
49
50 def _embed(x, order=3, delay=1):
51
52     N = len(x)
53     if order * delay > N:
54         raise ValueError("Error: order * delay should be lower than
    x.size")
55     if delay < 1:
56         raise ValueError("Delay has to be at least 1.")
57     if order < 2:
58         raise ValueError("Order has to be at least 2.")
59     Y = np.zeros((order, N - (order - 1) * delay))
60     for i in range(order):
61         Y[i] = x[(i * delay):(i * delay + Y.shape[1])]
62     return Y.T
63
64
65 @jit('UniTuple(float64, 2)(float64[:], float64[:])', nopython=True)

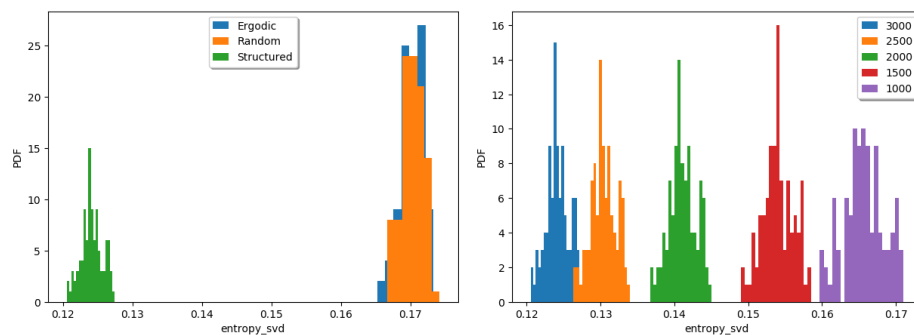
```

1.5 Plots Entropy

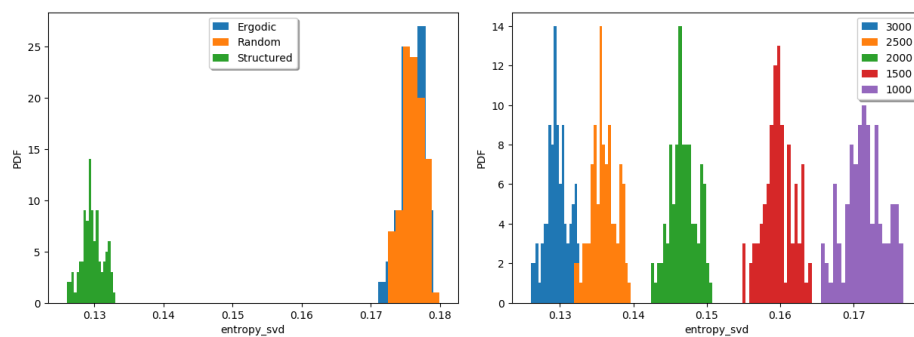
1.5.1 SVD with order 3



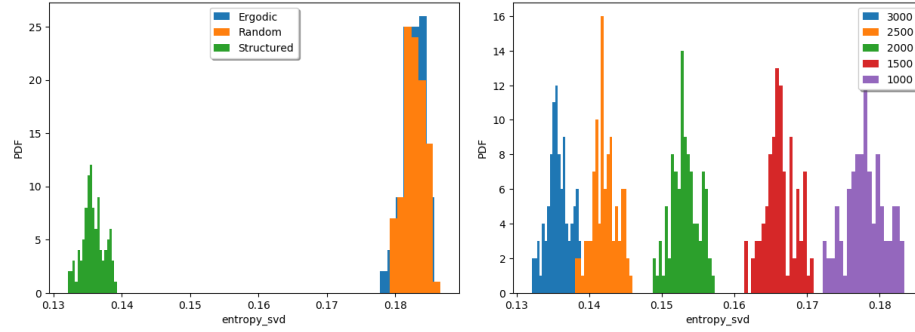
1.5.2 SVD with order 4



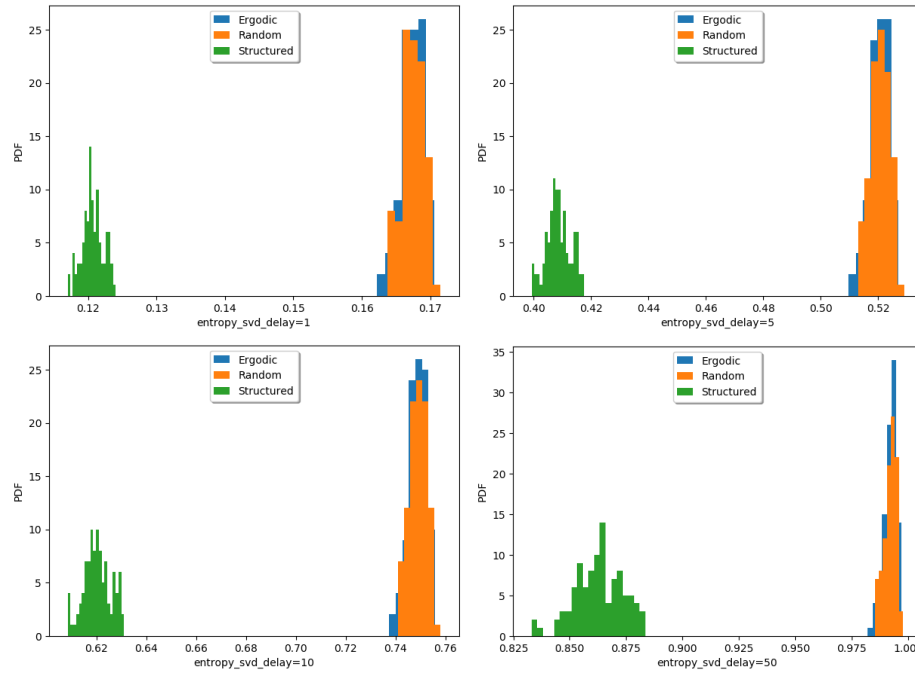
1.5.3 SVD with order 5

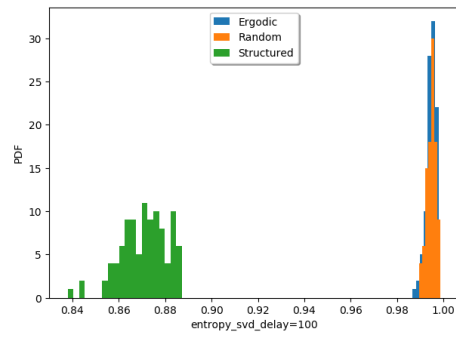


1.5.4 SVD with order 6

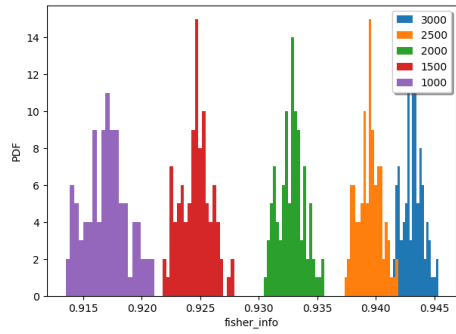
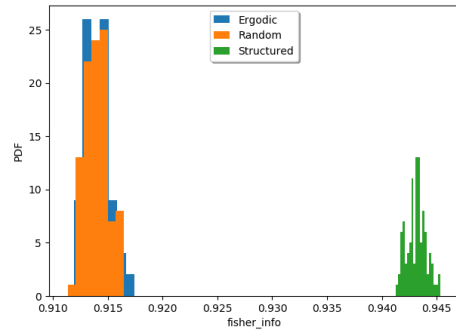


1.5.5 SVD time delay

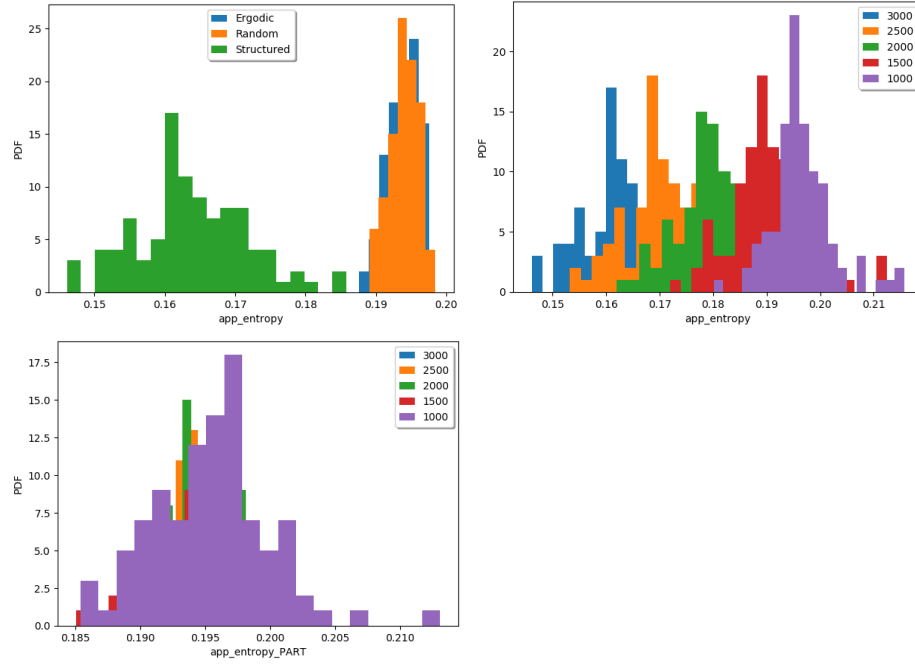




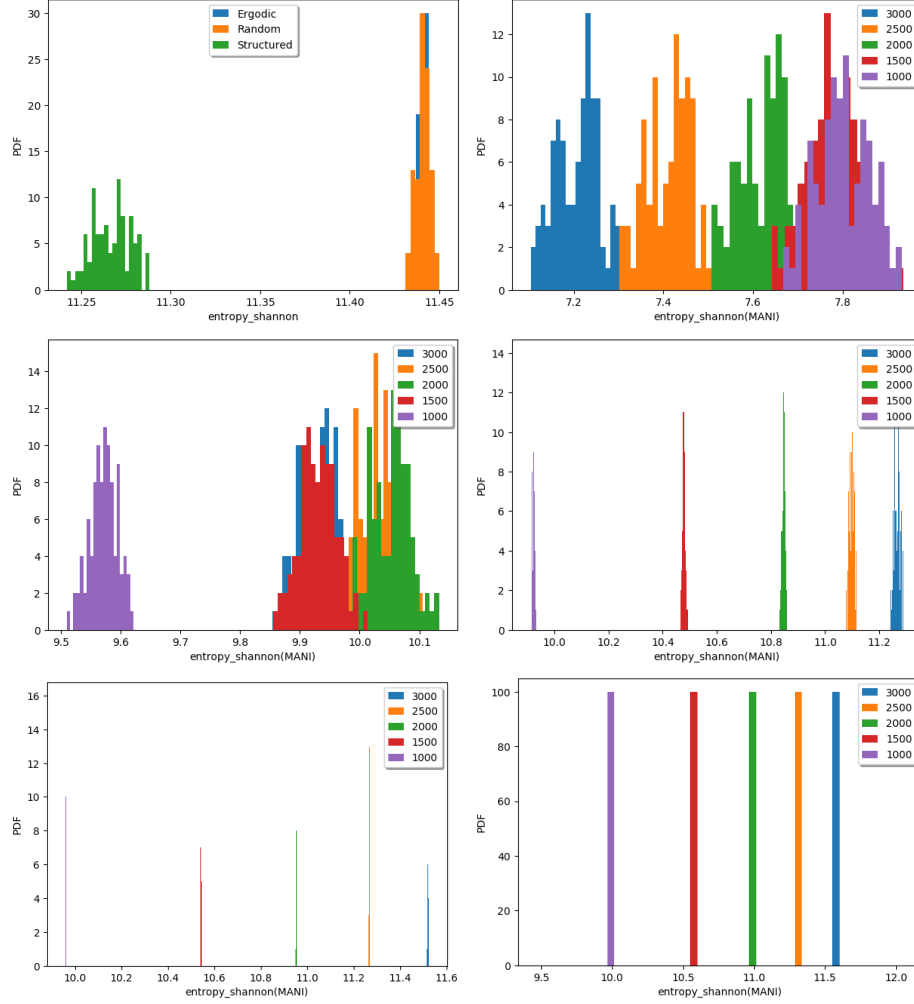
1.5.6 FISHER INFORMATION



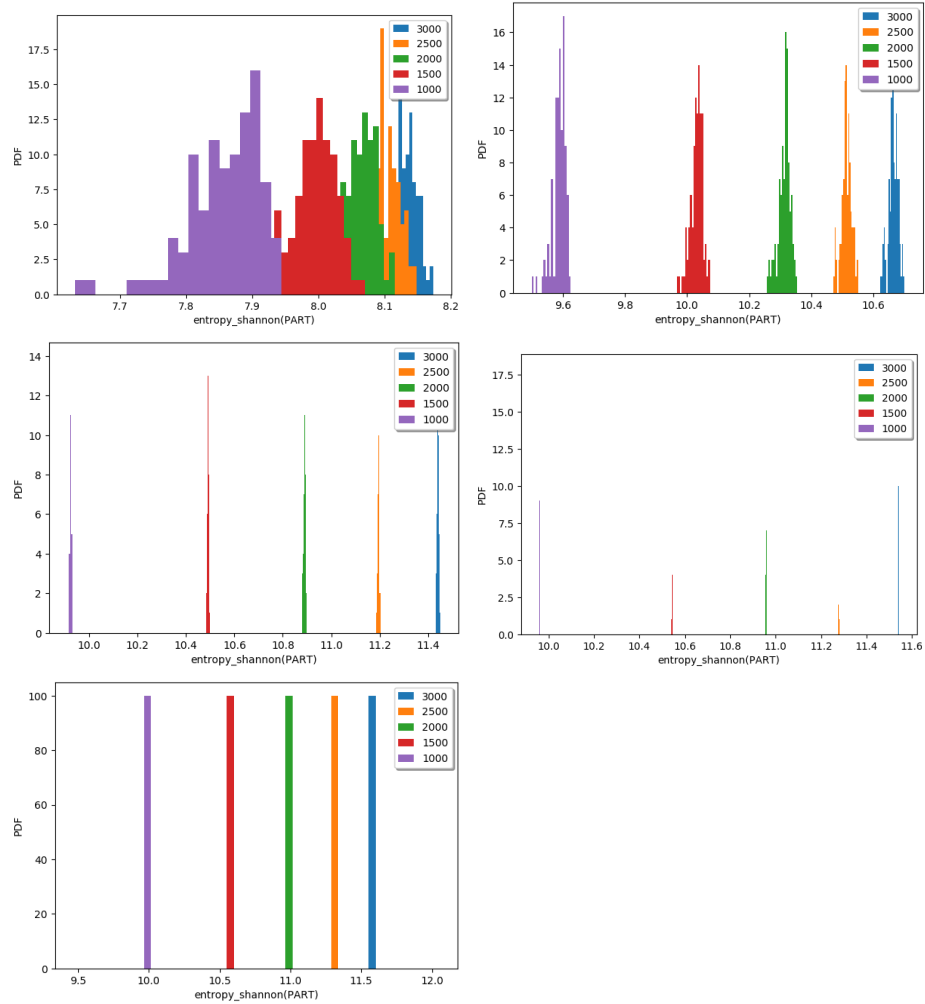
1.5.7 Approx Entropy



1.5.8 Shannon Entropy




1,2,3,4,exact digits precision. Instead of use trajectories closed to the fixed points we can evaluate entropy in random trajectories sampled from the Lorenz attractor.



2 Appendix

2.1 Chebyshev Norm

	a	b	c	d	e	f	g	h	
8	5	4	3	2	2	2	2	2	8
7	5	4	3	2	1	1	1	2	7
6	5	4	3	2	1		1	2	6
5	5	4	3	2	1	1	1	2	5
4	5	4	3	2	2	2	2	2	4
3	5	4	3	3	3	3	3	3	3
2	5	4	4	4	4	4	4	4	2
1	5	5	5	5	5	5	5	5	1
	a	b	c	d	e	f	g	h	