卒業論文

順列エントロピーに基づくカオスのサロゲートデータに対する比較分析

Future University Hakodate
システム情報科学部　複雑系知能学科
複雑系コース　1021148

小林 未佳

ヴラジミール・リアボフ

提出日　2025 年 1 月 23 日

BA Thesis

Comparative Analysis of surrogates of chaos based on Permutation Entropy

by

Mika Kobayashi

Complex Systems Course, Department of Complex and Intelligent Systems
School of Systems Information Science, Future University Hakodate

Supervisor:　Volodymyr Riabov

Submitted on January 23th, 2025

Abstract–

Distinguishing between chaos and noise has long been a significant challenge in the field of nonlinear dynamics. In chaotic regimes, the underlying dynamics can often be described and predicted using mathematical models. One approach to differentiate chaos from noise is the Complexity-Entropy Plane [6], which leverages Permutation Entropy —a concept introduced by Bandt and Pompe in 2002.

In this study, we focus on analyzing the Complexity-Entropy Plane for surrogate time series and investigating its properties in the context of two well-known two-dimensional maps: Ikeda map and Standard map. Ikeda map represents dissipative system, characterized by energy loss and attractors, while Standard map exemplifies a Hamiltonian system, which conserves energy and exhibits intricate phase-space structures. By studying these maps, we aim to explore how the Complexity-Entropy Plane captures the difference between noise and these chaos using surrogate method.

Keywords: chaos, noise, permutation entropy

概 要： カオスとノイズを区別することは、非線形ダイナミクスの分野において長年にわたり重要な課題である。カオスであれば適切な数学モデルを用いることで、その基礎となるダイナミクスを記述し、予測することが可能である。カオスとノイズを区別する効果的な方法の１つに、複雑性エントロピー平面 [6] がある。これは、2002 年に Bandt と Pompe によって提案された順列エントロピーを利用したものである。

本研究では、池田写像と標準写像に注目する。それぞれ、池田写像はエネルギーの損失やアトラクタを特徴とする散逸系を、標準写像はエネルギーを保存し複雑な位相空間構造を示すハミルトン系を代表する。この研究では、複雑系エントロピー平面がこれらの散逸系とハミルトン系について、サロゲートデータ法を用いてどの程度ノイズとカオスの違いを捉えることができるかを探る。

キーワード： カオス, ノイズ, 順列エントロピー

Contents

Chapter 1

Background

Distinguishing chaos and noise is one of the important problems in the field of signal processing. In the chaotic case, we can forecast the evolution of the system and describe the system using some non-linear equations. The problem becomes difficult due to many reasons, such as data pollution, control parameters of the methods, and restricted data length, etc. Some properties of chaotic systems also make this probelem difficult. In particular, dynamical chaos has displaying three properties: sensitivity to initial conditions, topological transitiveness and the presence of dense periodic orbits. In addition, noise and chaos are two intermingled concepts as for instance the former can induse the latter. There also has been suggested approaches: Lyapnov exponent, Fractal dimension. However, in this study, we focus on Permutation Entropy suggested by Bandt and Pompes [1].

1.1 Permutation Entropy

Permutation Entropy had been proposed by Bandt and Pompes in 2002 [1]. This approch is simple, robust, and computationally effeicient, so it has become popular for more than two decades and been used by reserchers from various fields. Robustness means that the result of this method is not very sensitive to parameters, such as time seires length or the length of the symbolic words.

The quantifer defiend as a Shannon Entropy can be calculated from the probability distribution of ordinal patterns obtained from a time series. Bandt and Pompe's method includes the algorithm for symbolization. This method has two parameters, embedding dimension d_x and embedding delay τ_x. Let us consider data X as an example.

$$X = \{9, 4, 5, 6, 7\}$$

when embedding dimension $d_x = 3$ and embedding delay $\tau_x = 1$, we can get the first embedded word w_1 as

$$w_1 = \{9, 4, 5\}$$

and its ordinal pattern π_1 is

$$\pi_1 = \{2, 3, 1\}$$

because $4 \leq 5 \leq 9$. The elements in π_1 are the indexes of all the elements sorted by ascending order in data distribution of w_1. The probability for each pattern ρ_i can be calculated as

$$\rho_i(\pi_i) = \frac{\text{occurence of the word i}}{n_x}$$

where n_x is the number of time steps and π_i is a possible pattern. Finally, Shannon entropy is calculated as

$$S(P) = -\Sigma_{i=1}^{n_\pi} \rho(\pi_i) log \rho_i(\pi_i)$$

In this study, I have been using the ordpy python package suggested by Pessa et al. [4] to calculate the permutation entropy.

1.2 Complexity-Entropy Plane

Complexity-Entropy Plane was initially introduced for distingusishing between chaos and stochastic time series by Rosso et al. in 2007 [6]. The formula is as following,

$$C(P) = \frac{D(P,U)H(P)}{D^{max}}$$

where D(P, U) is the Jensen-Shanon divergence between ordinal distribution P annd uniform distributon U which elements are $\frac{1}{n_\pi}$.

$$D(P,U) = S[\frac{(P+U)}{2}] - \frac{1}{2}S(P) - \frac{1}{2}S(U)$$

D_{max} is the maximum possible value of D(P, U) occuring from P, where it probabilities are 1, 0, 0, ..., 0

$$D_{max} = -\frac{1}{2}(\frac{n_\pi! + 1}{n_\pi} log(n_\pi + 1) - 2log(2n_\pi!) + log n_\pi)$$

Using these measures, we can see the difference between the chaos and noise by the position of points in 2-dimensional space with the values of C and H.(See Figure 1.1). C quantifies statistical complexity, and H is normalized Shanon Entropy. Figure 1.1 render the points Henon map

In [6], chaotic time series have high statistical complexity and lower entropy values compared to noise. The two solid lines show accesible region of the complexity-entropy values which depends on how to select the embedding dimension parameter in the Bandt and Pompe's method [1].

1.3 Preliminary Calculation for Logistic Map

First, we tried to calculate the permutation entropy of Logistic map as an example. Figure 2 shows my result of permutation entropy calculation for logistic map $x_{n+1} = rx_n(1 - x_n)$. The number of data points in a time series is 10000.

From Figure 2, we observe that permutation entropy tends to increase. This property is similar to Lyapnov exponent. However, the values somtimes fall abruptly, for example when the parameter r is between 3.8 and 3.9. It's caused because Logistic map shows intermittency and the values fall instantly in this interval. To calculate this entropy, the parameter of length of time seires is very inportant since the results depend on it.

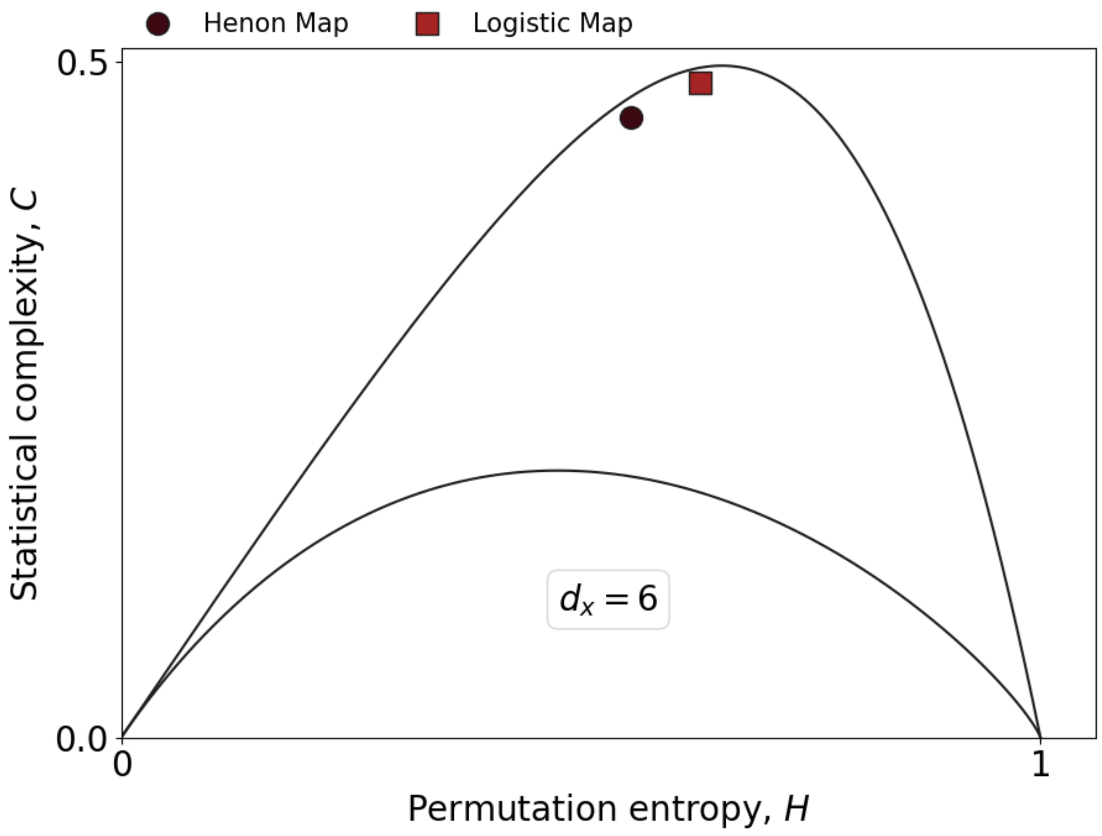

Figure 1.1: Complexity-Entropy Plane [6] for Henon map and Logistic map

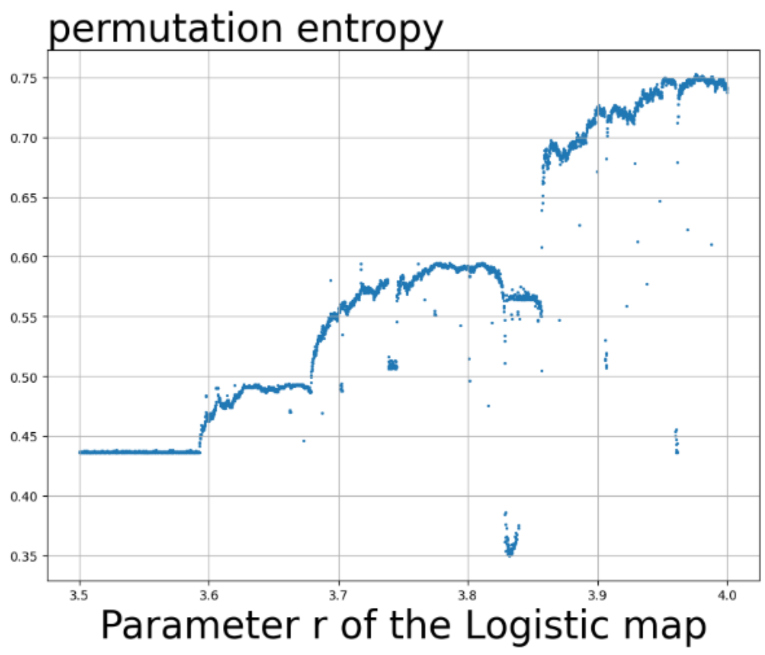

Figure 1.2: Permutation Entropy with embedding dimension d_x=4 for Logistic map

Chapter 2

Chaotic map

2.1 Ikeda map

In addition to maps in the seminal work[1], we're trying to study about Ikeda map and Standard map. Standard map has more noise-like timeseries, and Ikeda map has spiral shape. Ikeda map is calculated by following formula, and the Figure3 shows this map.

$$z_{n+1} = p + Bz_n(iexp[\kappa - \frac{\alpha}{1+|z_n|^2}])$$

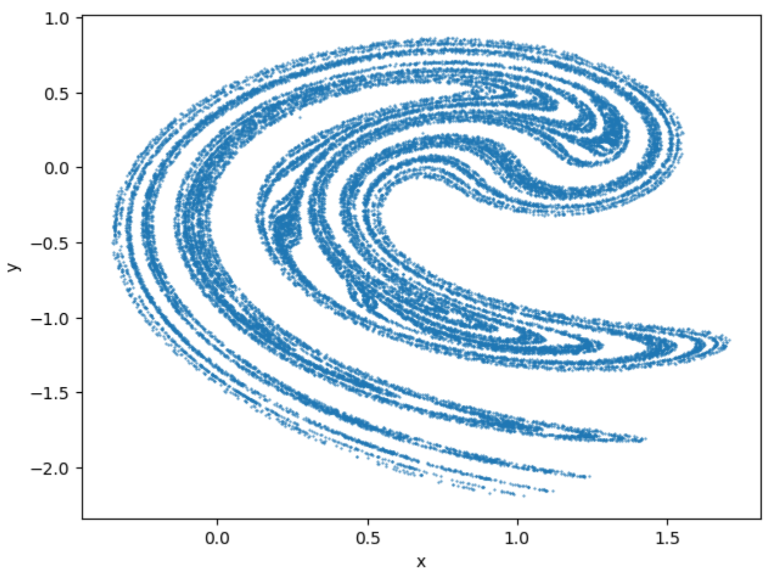

Figure 2.1: Ikeda map with parameter u=0.9, data length $N = 2^{15}$. The code is available on Appendix A.1

The 2D real example is also derivated using Euler's formula,

$$\begin{aligned} x_{n+1} &= 1 + u(x_n cos\theta_n - y_n sin\theta_n) \\ y_{n+1} &= u(x_n sin\theta_n + y_n cos\theta_n) \\ \text{where } \theta &= \kappa - \frac{\alpha}{1+|z_n|^2} \end{aligned}$$

This experiment calculated this map where parameters are $u = 0.9, \alpha = 6.0, \kappa = 0.4$

For a simple inference of Shannon Entrropy, We calculated ordinal distribution for Ikeda map and compared it to one of Henon map(see Figure 2.2 and Figure 2.3). The Henon map is calculated as following formula:

$$\begin{aligned} x_{n+1} &= 1 - ax_n^2 + y_n \\ y_{n+1} &= bx_n \end{aligned}$$

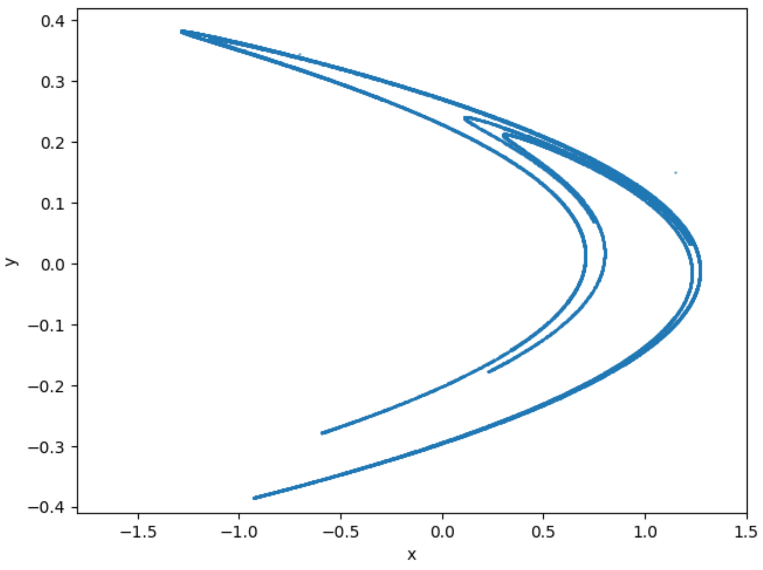

Figure 2.2: Henon map with parameter a=1.4, b=0.3, data length $N = 2^{15}$.

From Figure 2.3, Ikeda map has more ordinal patterns than those of Henon map, which is 2 dimensional chaotic map. Apparently, Both of these maps are dissipative, but Ikeda map seems to be more complicated.

2.2 Standard Map

Standard map [2] is calculated as

$$\begin{aligned} p_{n+1} &= p_n + \kappa sin x_n \\ x_{n+1} &= x_n + p_{n+1} \end{aligned}$$

and map is showed in Figure 2.4.

Standard map has chaotic tragectories for any κ, and in this experiments the map is calculated for $\kappa = 6.908745$. Domains of chaotic trajectories are bounded in most area, and unbounded in the p-direction for certain value κ_c, forming so-called stochastic sea.

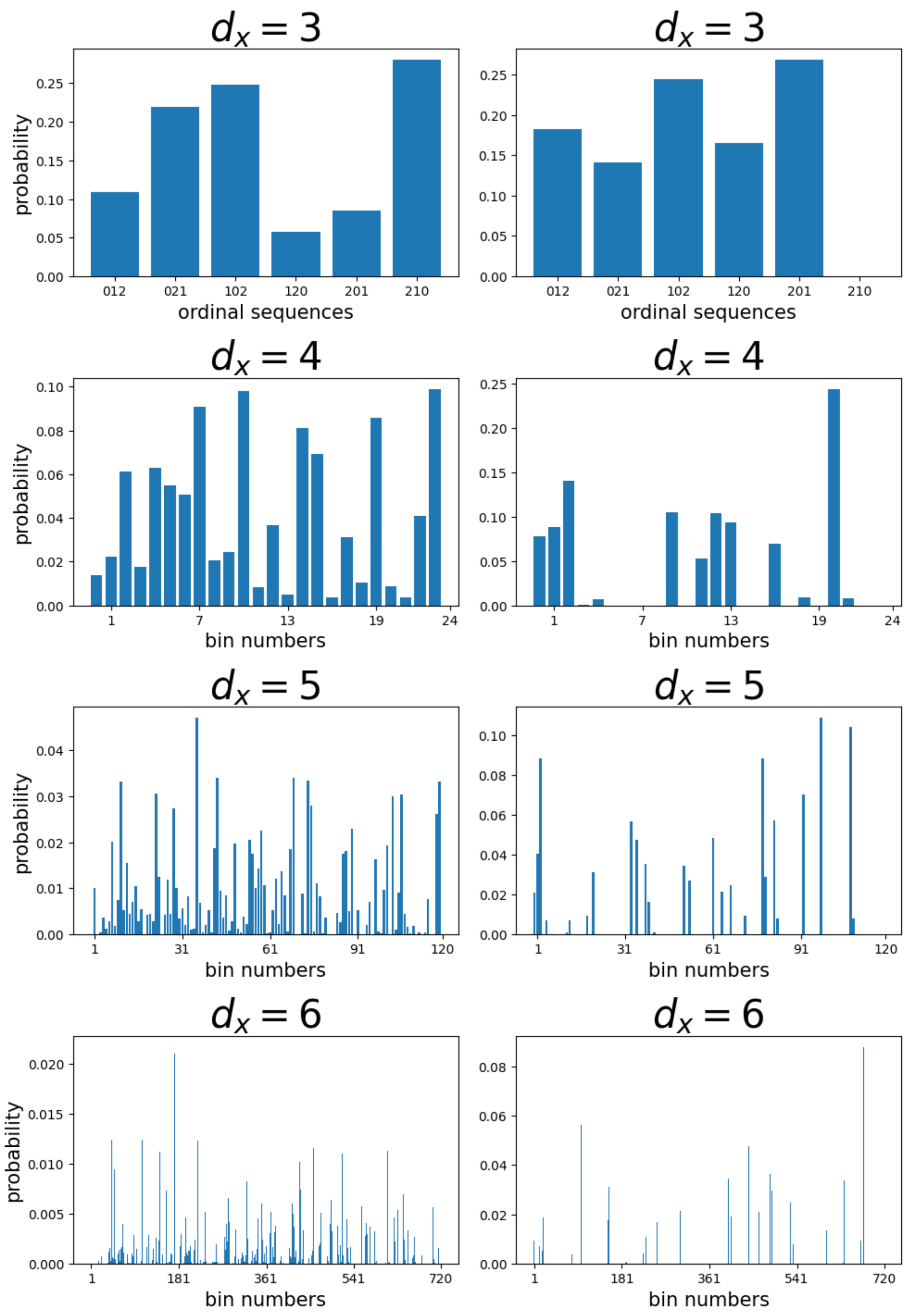

Figure 2.3: Ordinal distribution with different embedding dimension d_x of Ikeda map(left) and one of Henon map(right) of x-coordinate. The code is available on Appendix A.7

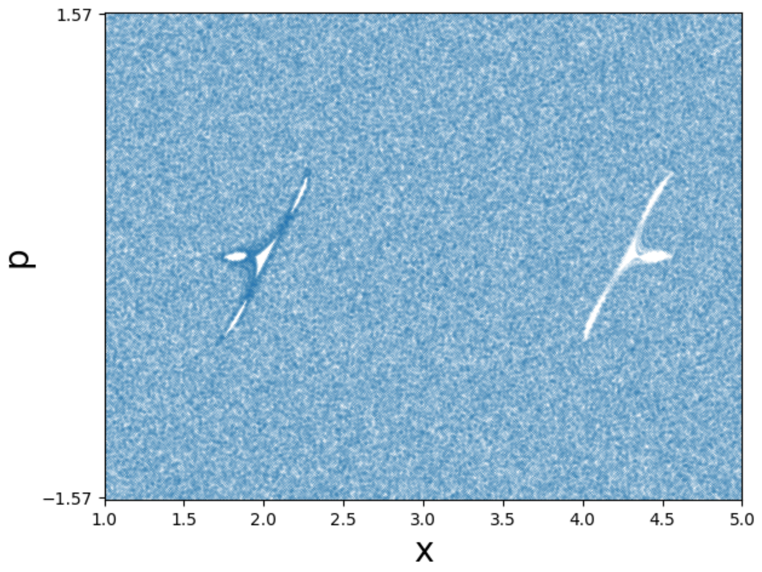

Figure 2.4: Standard map with parameter $\kappa = 6.908745$. The code is available on Appendix A.4

2.3 Complexity-Entropy Plane for Ikeda map and Standard map

Using ordpy package, Complexity-Entropy Plane for Ikeda map and standard map is plotted in Figure 2.5 The Ikeda map and Standard map are examples of 2D chaotic maps. Following the method from the original paper, the time series with x-coordinate was used for calculating Complexity-Entropy Plane. The algorithm is conducted with the embedding dimension $d_x = 6$.

According to the article [6], chaos has higher statistical complexity and lower entropies than noise.

However, in this numerical experiments (see Figure 2.5 and 2.6), the point of Ikeda map is in the chaos region, but Standard map seems like noise rather than chaos. This may be caused because Ikeda map is dissipative and Standard map is one of Hamiltonian system . The purpose of my work is distinguish chaos from noise in this situation.

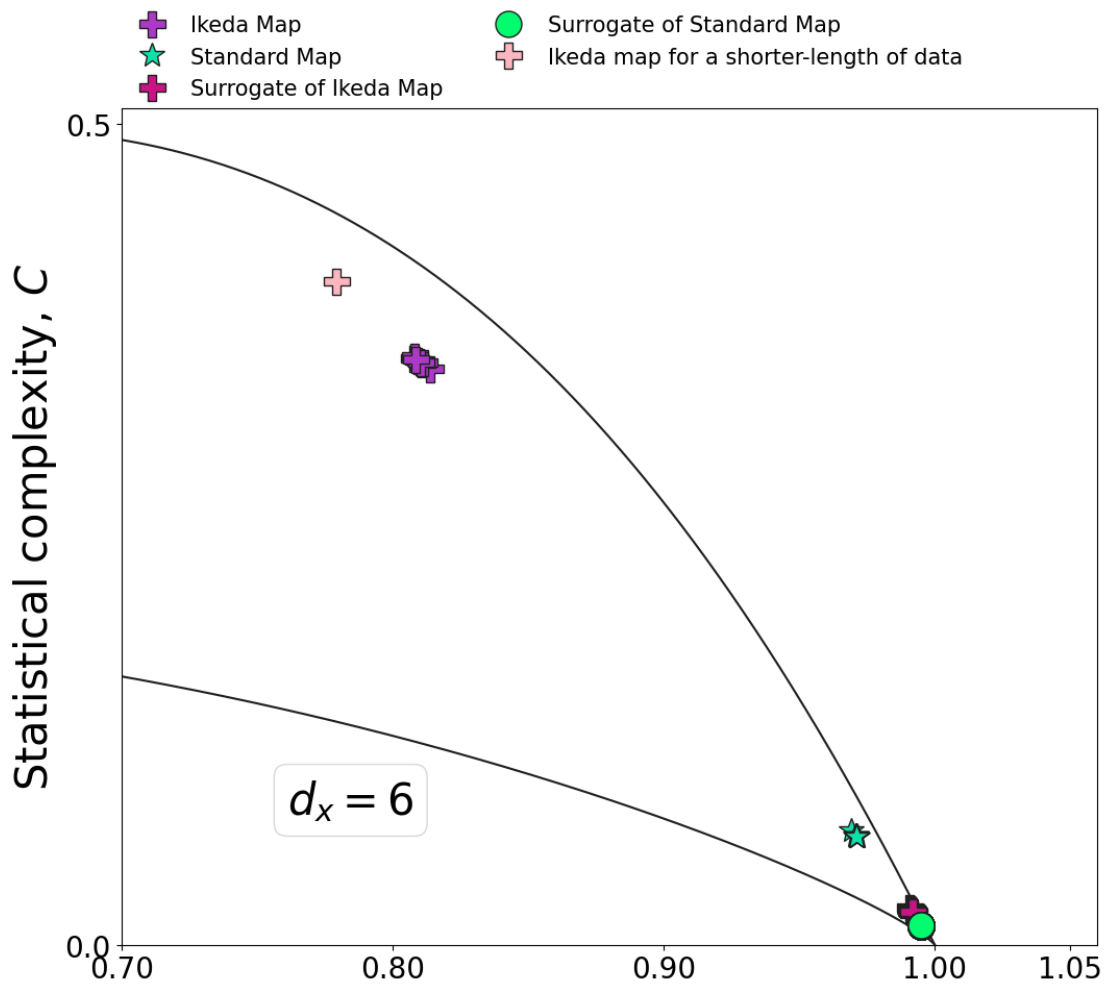

Figure 2.5: Complexity-Entropy Plane for Ikeda map and Standard map for embedding dimension $d_x = 6$. Data points of Ikeda map and Standard map are calculated 10 times with different initial condition.

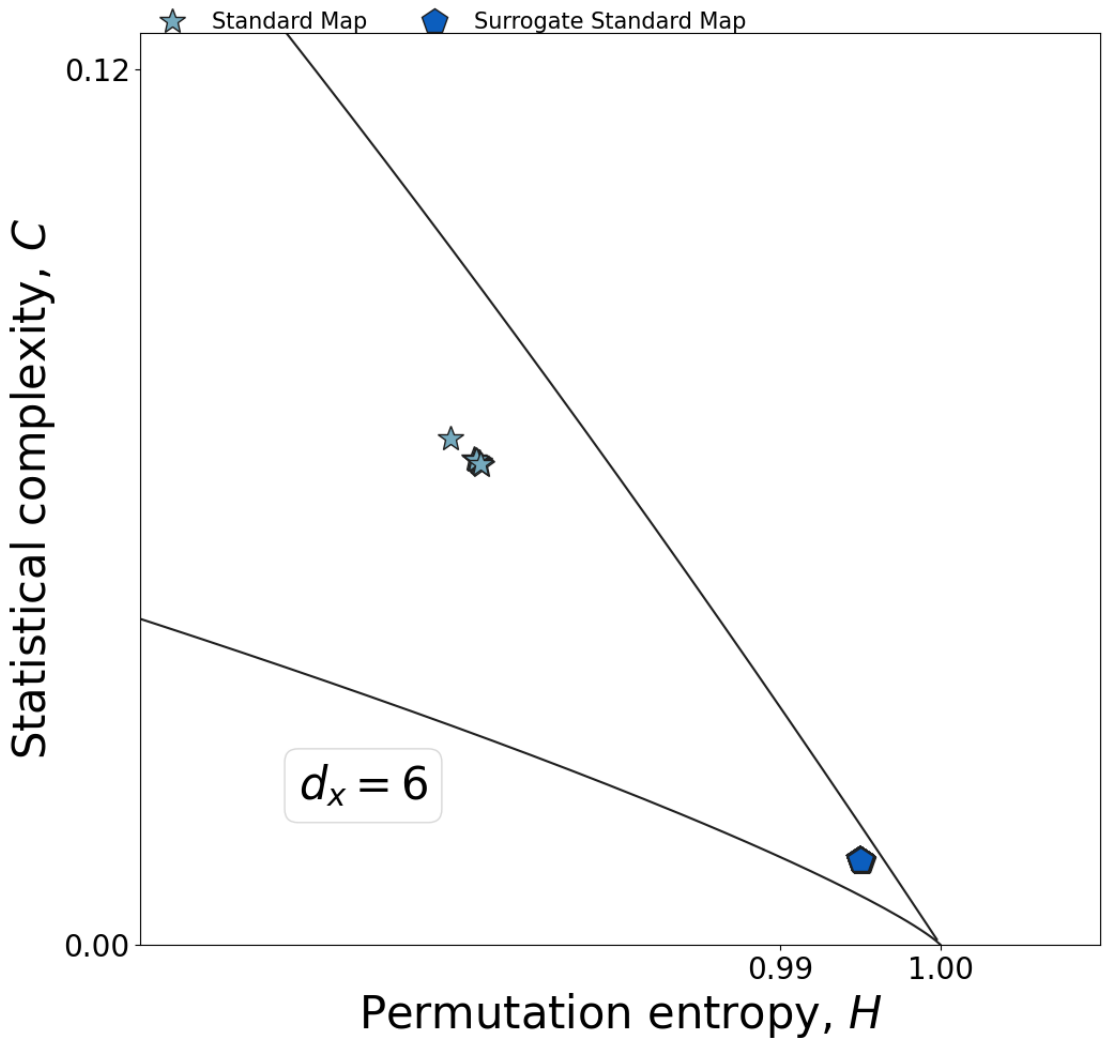

Figure 2.6: Zoomed Figure2.5 around the position of noise. 1000 surrogate data of Standard map are plotted, but they are almost at the same position.

Chapter 3

Method

3.1 Surrogate

Surrogate is a statistical hypothesis method for testing correlation properties of a dynamical system, suggested by Theiler [7]. This method has originally suggested for detecting nonliniarity from observed time series, and generating time series that preserve one or more statistical but destroy dynamical properties of a given data. There are several methods for creating surrogates, such as Shuffling, Phase randomization, etc. This experiment uses Amplitude adjusted Fourier transform(AAFT) algorithm.

3.2 Amplitude adjusted Fourier transform

Amplitude adjusted Fourier Transform algorithm have originally suggested for testing null hypothesis that the original data set is monotonic nonlinear transformation of a linear gaussian process. In original Fourier transform algorithm, Fourier transform is computed and only phase ϕ is randomized. Surrogate time series given by this algorithm have the same Fourier spectrum as the original time series, like simple Fourier transform surrogate algorithm. In AAFT, they are also adjusted to have the same amplitude distribution. The behaviour of surrogate data given by algorithm on the Complexity-Entropy Plane is the same as white noise, so the ordinal distribution is uniform distribution.

There is Julia package [3] for generating surrogate data sets available online. You can see how to add this julia package to your PC in appendix for the younger members.

3.3 Chi-Square statistic

After generating surrogate data sets, Chi-Squared statistic [5] was computed in this experiments. To compute chi-square statistic, follow this formula:

$$\chi^2 = \sum_i \frac{(R_i - S_i)^2}{R_i + S_i} \tag{3.1}$$

where R_i is the number of events in bin i for the first data set and S_i is the number of events in the same bin i for the second data set.

Using Chi-Square distance between surrogate data sets and original chaotic time series, the difference will get clearer. The program to compute this statistic is on Appendix.

Chapter 4

Results

4.1 Computing chi-square distance

The results of this experiment is Figure 4.1. The number of chi-square statistic data in this figure is each 100 at maximum.

Although the shape of distribution of chi-square statistic of surrogate data (Figure 4.2) is like a normal distribution, it is clear that it was far from one between surrogate data and each original chaotic data. From this results, we can see the diffirence between chaotic data and the randomized time series.

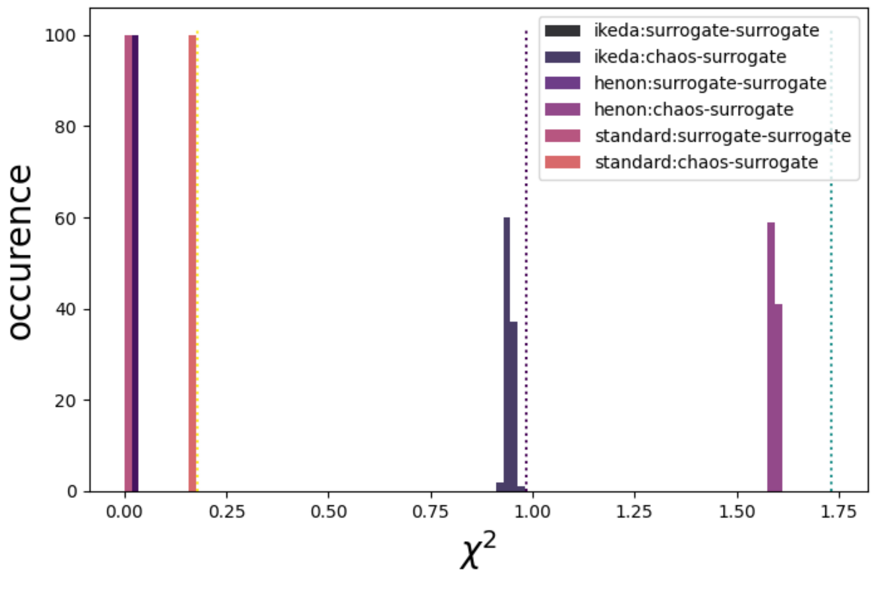

Figure 4.1: Histogram of chi-square statistic (Formula 3.1). The code to create this figure is on Appendix A.8

The dot line is chi-square distance between uniform distribution($U = \frac{1}{6!} = \frac{1}{720}$, 6 is the number of word) and each chaotic data. The chaotic map of each dot line is the same with the one of nearest bins of statistics. Considering the meaning of Statistical Complexity, the distance between the dot line and each chaotic data is related

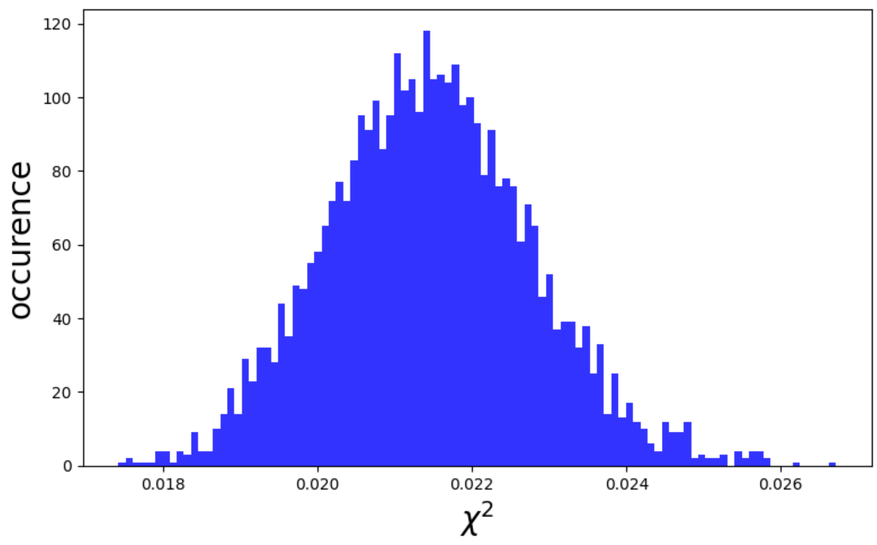

Figure 4.2: Histogram of chi-square statistic among surrogate data of Ikeda map with parameter $u = 0.9$. The code is on Appendix A.9

to Statistical Complexity. For example, as for Ikeda map, the distance betweeen original chaotic data and surrogate data is smaller than one of Henon map.

In order to compare the values of chi-square statistic to one of shanon entropy (x-cordinate of Complexity-Entropy Plane) and statistical complexity (y-cordinate) , please see Figure 4.3.

The values(D_h, D_c) of Histogram Figure 4.3 was calculated as following:

$$\begin{aligned} D_h &= H_{\text{surrogate}} - H_{\text{chaos}} \\ D_c &= C_{\text{surrogate}} - C_{\text{chaos}} \end{aligned}$$

where H_{chaos} is Shannon Entropy of chaotic data, which was caluculated as x-cordinate of Complexity-Entropy Plane, and C_{chaos} is Statistical Complexity of chaotic data, which was caluculated as y-cordinate of the plane. In addition, we calculated the difference among surrogate data sets.

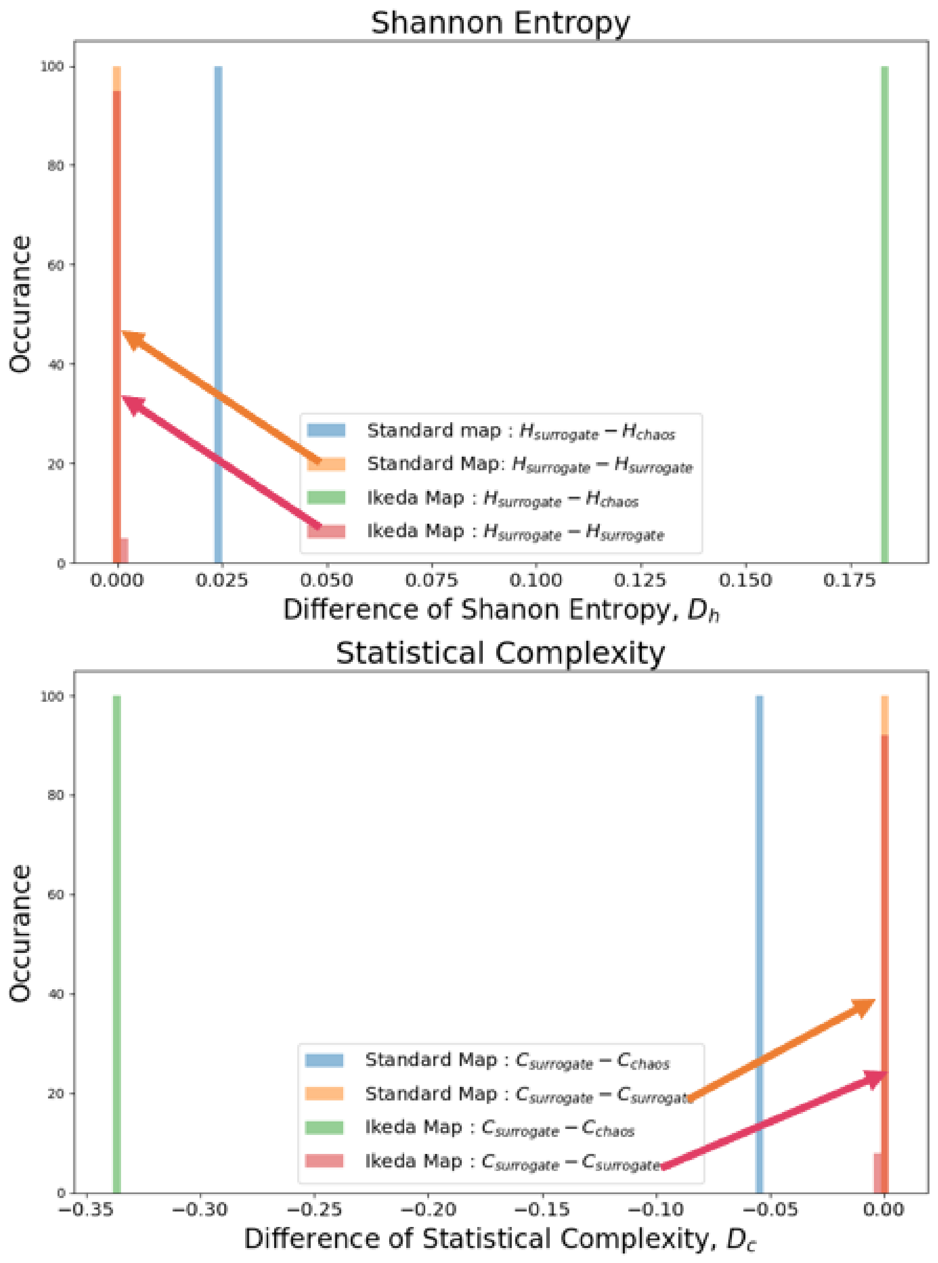

Figure 4.3: Histogram of difference of Shanon Entropy and one of Statistical Complexity. The code is available on Appendix A.10

Chapter 5

Discussion and Conclusion

5.1 Discussion

Using surrogate method, the distance between chaos and noise got clear. To investigate whether the given time series are chaos or not, making surrogate data from the original and computing the chi-square distance is useful.

To be honest, the chi-square distance between the surrogate data and the original data was significantly larger than initially expected. Prior to the experiment, we hypothesized that the chi-square distribution would resemble a normal distribution. While each dataset individually exhibited characteristics of a normal distribution as you can see Figure 4.2, the experimental results demonstrated a substantial difference in values, rendering the assumption of a normal distribution unnecessary.

To go further study, there are 2 curious questions. The first is about testing for Ikeda map with other parameter values. (Standard map has chaotic trajectory for $\kappa > 1$, and Ikeda map change its property depending on parameter u.) Second, we can get more information when using another embedding dimension and embedding delay. The software created in the prior work [4] has variables including y-direction, so we can easily compute entropies including y-direction data. Figure 5.1 shows Complexity-Entropy Plane using embedding dimension $d_x = 2, d_y = 3$. It's undisputable true that Ikeda map with parameter u=0.9 is chaos, judging from this map, but we can't see Ikeda map with parameter $u = 0.6$ and $u = 0.85$ are chaos or not. In addition, as you can see in Figure 5.2, it also seems interesting to inverstigate how the points move depending on embedding dimension. It seems to be interesting to use surrogate method to solve this problem.

5.2 Conclusion

This study showed that the surrogate method is helpful for distinguishing chaos from noise by using ordinal patterns distribution and chi-square statistic. In future studies, exploring different parameters and settings could provide further insights into chaotic systems. The surrogate method shows potential as a tool for addressing these questions and improving our understanding of chaos.

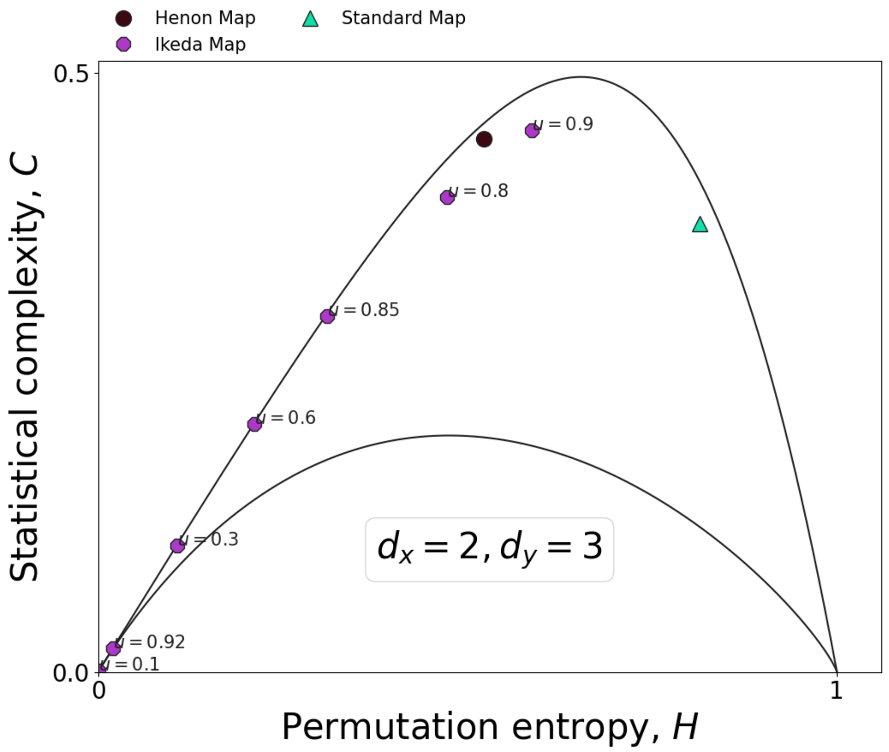

Figure 5.1: Complexity-Entropy Plane using embedding dimension $d_x = 2, d_y = 3$. In this figure, Ikeda map was computed with parameter $u = [0.1, 0.3, 0.6, 0.8, 0.85, 0.9, 0.92]$. The code is available on Appendix A.11

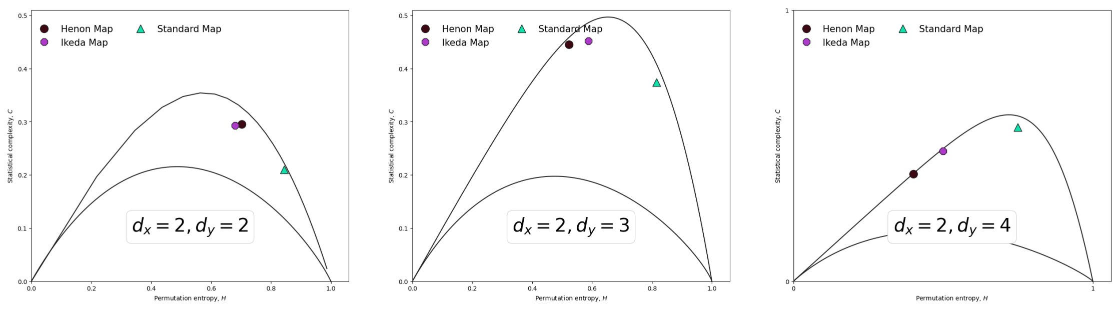

Figure 5.2: Complexity-Entropy Plane using combinations of embedding dimension. In this figure, Ikeda map was computed with parameter $u = 0.9$. The program is on Appendix A.12

Acknowledgement

I sincerely thank Professor Riabov. He patiently supported me through my graduation research, and his unwavering guidance made it possible for me to complete this work. Thank you very much for your support over the year.

Bibliography

[1] C. Bandt and B. Pompe. Permutation entropy: A natural complexity measure for time series. *Phys.Rev.Lett*, 88(17):174102, 2002.

[2] G.M.Zaslavsky. Chaos, fractional kenetics, and anomalous transport. *Physics Reports*, 371:461–580, 2002.

[3] Kristian Agaster Haaga and George Datseris. Timeseriessurrogates.jl: a julia package for generating surrogate data. *Journal of Open Source Software*, 7(77):4414, 2022.

[4] A. A. B. Pessa and H. V. Ribeiro. ordpy: A python package for data analysis with permutation entropy and ordinal network methods. *Chaos*, 31(6):063110, 2021.

[5] W. H. Press, S. A. Teukolsky, W. T. Vetterring, and B. P. Flannery. *Numerical Recipes.* The Art of Scientific Computing. Cambridge University Press, third edition edition, 2007.

[6] O. A. Rosso, H. A. Larrondo, M. T. Martin, A. Plastino, and M. A. Fuentes. Distinguishing noise from chaos. *Phys. Rev. Lett.*, 99:154102, Oct 2007.

[7] James Theiler, Stephen Eubank, André Longtin, Bryan Galdrikian, and J. Doyne Farmer. Testing for nonlinearity in time series: the method of surrogate data. *Physica D: Nonlinear Phenomena*, 58(1):77–94, 1992.

Appendix A

Appendix

In Appendix, we will show the programs used in this study.

A.1 Ikeda map

A.1.1 Calculate Ikeda map

This ikeda.py python program contains 3 functions (ikeda_map(), ikeda_map2(), ikeda_map3()). Please note that the first function "ikeda map()" returns data including the first 1000 transient. I used this method in this study because the chaotic data examined in ordpy package [4] conducted experiments including transient. However, if you want to use data without transient, please use the second function "ikeda_map2()". It returns no transient data. Finally, in the third function "ikeda_map3()", you can assign the parameter in ikeda map of u for each direction(x and y). The result of the first function is Figure 2.1.

Listing A.1: ikeda.py

```python


import numpy as np
import warnings
import os
import argparse


# calculate data including transient
def ikeda_map(n=2**15, u=0.9, kappa=0.4, alpha = 6.0, x0=0.6, y0=0.1):
    """
    n:time series length
    kappa : 0.4
    alpha : 0.6

    u:0.9
    x0 : initial condition
    y0: initial condition
    ==============================
    Returns the x and y variables of Ikeda map
    """

    with warnings.catch_warnings():
        warnings.simplefilter("error")

        bool_ = False
        while bool_ == False:
            try:
                x = np.zeros(n)
                y = np.zeros(n)

                x[0] = x0
                y[0] = y0

                for i in range(1, n):
                    theta = kappa - alpha/(1+x[i-1]**2 + y[i
                        -1]**2)
                    x[i] = 1+u*(x[i-1]*np.cos(theta) - y[i-1]*np.
                        sin(theta))
                    y[i] = u*(x[i-1] *np.sin(theta) + y[i-1]*np.
                        cos(theta))

                bool_ = True

            except RuntimeError:
                x0 = np.random.uniform()
                y0 = np.random.uniform()
    return x, y


# calculate without transient
def ikeda_map2(n=2**15, u=0.9, kappa=0.4, alpha = 6.0, x0=0.6, y0
    =0.1):
    """
    discard first 1000(transient)
    n:time series length
    kappa : 0.4
    alpha : 0.6

    u:0.9
    x0 : initial condition
    y0: initial condition
    ============================
    Returns the x and y variables of Ikeda map
    """

    with warnings.catch_warnings():
        warnings.simplefilter("error")

        bool_ = False
        while bool_ == False:
            try:
                x = np.zeros(n)
                y = np.zeros(n)

                x[0] = x0
                y[0] = y0

                for i in range(1, n):
                    theta = kappa - alpha/(1+x[i-1]**2 + y[i-1]**2)
                    x[i] = 1+u*(x[i-1]*np.cos(theta) - y[i-1]*np.sin(theta))
                    y[i] = u*(x[i-1] *np.sin(theta) + y[i-1]*np.cos(theta))

                bool_ = True

            except RuntimeError:
                x0 = np.random.uniform()
                y0 = np.random.uniform()
    return x[1000:], y[1000:]


# set u_x and u_y distinctively
def ikeda_map3(n=2**15, ux=0.9, uy=0.9, kappa=0.4, alpha = 6.0, x0=0.6, y0=0.1):
    """
    discard first 1000(transient)
    n:time series length
    kappa : 0.4
    alpha : 0.6

    u:0.9
    x0 : initial condition
    y0: initial condition
    ============================
    Returns the x and y variables of Ikeda map
    """

    with warnings.catch_warnings():
        warnings.simplefilter("error")

        bool_ = False
        while bool_ == False:
            try:
                x = np.zeros(n)
                y = np.zeros(n)

                x[0] = x0
                y[0] = y0

                for i in range(1, n):
                    theta = kappa - alpha/(1+x[i-1]**2 + y[i-1]**2)
                    x[i] = 1+ux*(x[i-1]*np.cos(theta) - y[i-1]*np.sin(theta))
                    y[i] = uy*(x[i-1] *np.sin(theta) + y[i-1]*np.cos(theta))

                bool_ = True

            except RuntimeError:
                x0 = np.random.uniform()
                y0 = np.random.uniform()
    return x[1000:], y[1000:]
```

A.1.2 Calculate probabilies sorted by ascending order of patterns

This function ”sort_probs()” returns ordinal distribution sorted by acesnding order. The function ”ordinal_distribution” in ordpy python package [4] returns ordinal distribution (the values are from 0 to 1.0) and ordinal patterns ($[0, 1, 2], [1, 2, 0], [2, 1, 0]$...).

Listing A.2: stats_func.py

```python
def sort_probs(pats, probs):
    def make_label(pis):
        label = []
        for l in pis:
            label.append(''.join(map(str, l)))
        return label

    probs = np.array(probs)
    probs = probs[np.argsort(make_label(pats))]
    return probs
```

A.1.3 Save ordinal distribution

The next program is used for saving data of distribution sorted by pattern’s ascending order. Please execute this function if you want to save distribution before computing chi-square distances, because it takes long time.

Listing A.3: dist.py

```python




import glob
import numpy as np
from ordpy import ordinal_distribution
from stats_func import *
# import os

# paths = glob.glob("aaft/**.npy")
```

```python
paths = glob.glob("original/**.npy")
print(paths)

for i, path in enumerate(paths):
    data = np.load(path)
    x = data[:, 0]
    pats, probs = ordinal_distribution(x, dx=6, return_missing=
        True)

    probs = sort_probs(pats, probs)
    # np.save(f"dist/aaft/dist_{i}", probs)
    np.save(f"dist/original/dist_{i}", probs)
```

A.2 Standard map

Listing A.4: standard.py

```python


import numpy as np
import warnings
import matplotlib.pyplot as plt
import glob

# standard map
def standard_map(n=2**20, k=6.908745, theta0=np.random.random(),
    p0=np.random.random()):
    """
    Parameters
    ----------
    n: time series length
    z: map parameter
    ----------
    Returns an orbit of an iterated standard map
    """

    with warnings.catch_warnings():
        warnings.simplefilter("error")

        bool_ = False
        while bool_ == False:
            try:
                theta = np.zeros(n)
                p = np.zeros(n)

                theta[0] = np.remainder(theta0, 2*np.pi)
                p[0] = np.remainder(p0, 2*np.pi)

                for i in range(1, n):
                    p[i] = np.remainder(p[i-1] + k*np.sin(theta[i
                        -1]), 2*np.pi)
                    theta[i] = np.remainder(theta[i-1] + p[i], 2*np.pi)


                bool_ = True

            except RuntimeError: # change the initial condition
                theta0 = np.random.uniform()
                p0 = np.random.uniform()

    return np.stack([theta, p], axis=1)


def show_standard(paths="original/**.npy"):
    '''
    Use this method to show standard show.
    Parameters
    ------------------
    paths : path for standard map's numpy data
    ------------------
    '''
    paths = glob.glob(paths)
    data = np.load(paths[0])

    x = data[:,0]
    y = data[:, 1] -np.pi

    y[y>0] = y[y>0]-2*np.pi
    y[y<0] = y[y<0]+np.pi

    plt.figure(figsize=(8, 8))
    plt.scatter(x,y, s=0.005)
    plt.xlabel("x", fontsize=20)
    plt.ylabel("p", fontsize=20)
    plt.yticks([-np.pi/2, np.pi/2], fontsize=10)
    plt.xlim([1, 5])
    plt.ylim([-np.pi/2-0.01, np.pi/2+0.01])
    plt.tight_layout(pad=0.6, h_pad=0.2, w_pad=0.2)
```

A.3 Surrogate

A.3.1 Computes surrogate data with julia package

Listing A.5: surrogate.jl

```julia


using Plots
using Printf
using TimeseriesSurrogates, CairoMakie
using PyCall
```

```

np = pyimport("numpy")
glob = pyimport("glob")
paths = glob.glob(the_path_of_ikeda_map)
N = 100

data = np.load(paths[1])
x = data[:, 1]
y = data[:, 2]

for i in 1:N
    sx = surrogate(x, AAFT())
    sy = surrogate(y, AAFT())

    ss = np.stack([sx, sy], axis=1)
    np.save(the_path_where_you_want_to_save_surrogate_data, ss)
end
```

A.3.2 Computes chi-square distance

Listing A.6: stats_func.py

```



def chstwo(bins1, bins2, knstrn = 1):
    # computes chi-sqare distance
    bins1 = np.array(bins1)
    bins2 = np.array(bins2)
    mask = np.where((bins1!=0) & (bins2!=0))
    chsq = np.sum((bins1[mask]-bins2[mask])**2 / (bins1[mask]+
       bins2[mask]))
    # print(chsq)
    return chsq
```

A.4 Chaotic map

A.4.1 Figure 2.3

Listing A.7: Ordinal distribution with each embedding dimension d_x of Ikeda map(left) to one of Henon map(right) of x-coordinate.

```

import glob
import numpy as np
import matplotlib.pyplot as plt
from ordpy import ordinal_distribution

paths0 = glob.glob("data/ikeda/original/u090/**.npy")
paths1 = glob.glob("data/henon/original/**.npy")
```

```python
ikeda = np.load(paths0[0])
henon = np.load(paths1[0])

def make_label(pis):
    label = []
    for l in pis:
        label.append(''.join(map(str, l)))
    return label

ikeda_PATTERNS = []
ikeda_PROBS = []
henon_PATTERNS = []
henon_PROBS = []
for i in range(3, 7):
    patterns, probs = ordinal_distribution(ikeda[:, 0], dx=i,
        return_missing=True)

    probs = np.array(probs)
    probs = probs[np.argsort(make_label(patterns))]
    patterns = np.sort(make_label(patterns))

    ikeda_PATTERNS.append(patterns)
    ikeda_PROBS.append(probs)

    patterns, probs = ordinal_distribution(henon[:, 0], dx=i,
        return_missing=True)

    probs = np.array(probs)
    probs = probs[np.argsort(make_label(patterns))]
    patterns = np.sort(make_label(patterns))

    henon_PATTERNS.append(patterns)
    henon_PROBS.append(probs)

# ip = [int(x) for x in patterns for patterns in ikeda_PATTERNS]
# hp = [int(x) for x in patterns for patterns in henon_PATTERNS]

# np.save("ikeda_ordinal_bars", np.stack([ip, ikeda_PROBS], axis
    =1))
# np.save("henon_ordinal_bars", np.stack([henon_PATTERNS,
    henon_PROBS], axis=1))
# print(len(henon_PROBS), len(ikeda_PROBS))

# Draw
plt.rcParams['xtick.labelsize'] = 10 # 軸だけ変更されます。
plt.rcParams['ytick.labelsize'] = 10 # 軸だけ変更されます

# 描画
fig, axes = plt.subplots(4, 2, figsize=(10, 14.5))
# fig.suptitle("Ordinal distribution of Ikeda map and Henon map",
    fontsize=30)
for i in range(0, 4):

    if i ==0:
        axes[i, 0].bar(ikeda_PATTERNS[i], ikeda_PROBS[i])
        axes[i, 0].set_title("$d_x={}$".format(i+3), fontsize=30)
        axes[i, 0].set_ylabel("probability", fontsize=15)
        axes[i, 0].set_xlabel("ordinal␣sequences", fontsize=20)

        axes[i, 1].bar(henon_PATTERNS[i], henon_PROBS[i])
        axes[i, 1].set_title("$d_x={}$".format(i+3), fontsize=30)
        axes[i, 1].set_xlabel("ordinal␣sequences", fontsize=15)
    else:
        labels = [int(x) for x in range(1, len(ikeda_PATTERNS[i])+1, int(len(ikeda_PATTERNS[i])/4))]
        labels.append(len(ikeda_PATTERNS[i]))
        print(labels)

        axes[i, 0].bar(np.arange(0, len(ikeda_PATTERNS[i]), 1), ikeda_PROBS[i])
        axes[i, 0].set_title("$d_x={}$".format(i+3), fontsize=30)
        axes[i, 0].set_xticks(labels)
        axes[i, 0].set_ylabel("probability", fontsize=15)
        axes[i, 0].set_xlabel("bin␣numbers", fontsize=15)

        axes[i, 1].bar(np.arange(0, len(henon_PATTERNS[i]), 1), henon_PROBS[i])
        axes[i, 1].set_title("$dx={}$".format(i+3), fontsize=30)
        axes[i, 1].set_xticks(labels)
        axes[i, 1].set_xlabel("bin␣numbers", fontsize=15)


plt.tight_layout(rect=[0,0,1,0.99], h_pad=1.2, w_pad=1.5)
```

A.5 Results

To create the figures in Chapter 4, Please see the programs below.

A.5.1 Figure 4.1

Listing A.8: example13.py

```


import numpy as np
import matplotlib.pyplot as plt
from ordpy import ordinal_distribution
import glob
from stats_func import *
import tqdm
import random

uniform_dist = [1/720]*720
```

```

# ikeda map u=0.9
ikeda = glob.glob("data/ikeda/dist/original/u090/**.npy")
surrogates = glob.glob("data/ikeda/dist/aaft/u090/**.npy")
surrogates2 = surrogates
ikedaSUD = []
ikedaORD = []

for ss in tqdm.tqdm(surrogates):
    if len(surrogates2)==0:
        break

    probs1 = np.load(ss)
    surrogates2.remove(ss)
    for st in surrogates2:
        probs2 = np.load(st)
        ikedaSUD.append(chstwo(probs1, probs2))


ikedaSUD = random.sample(ikedaSUD, 100)
ikedaP = np.load(ikeda[0])

# read path variable again
del surrogates
surrogates = glob.glob("data/ikeda/dist/aaft/u090/**.npy")
for ss in tqdm.tqdm(surrogates):
    probs1 = np.load(ss)
    ikedaORD.append(chstwo(probs1, ikedaP))


# henon map
henon = glob.glob("data/henon/dist/original/**.npy")
surrogates = glob.glob("data/henon/dist/aaft/00/**.npy")
surrogates2 = surrogates
henonSUD = []
henonORD = []

for ss in tqdm.tqdm(surrogates):
    probs1 = np.load(ss)
    surrogates2.remove(ss)

    for st in surrogates2:
        probs2 = np.load(st)
        henonSUD.append(chstwo(probs1, probs2))

henonSUD = random.sample(henonSUD, 100)
henonP = np.load(henon[0])

del surrogates
surrogates = glob.glob("data/henon/dist/aaft/00/**.npy")
for ss in tqdm.tqdm(surrogates):
    probs1 = np.load(ss)
    henonORD.append(chstwo(probs1, henonP))



# standard
standard = glob.glob("data/standard/dist/original/**.npy")
surrogates = glob.glob("data/standard/dist/aaft/**.npy")
surrogates2 = surrogates
stdSUD = []
stdORD = []

for ss in tqdm.tqdm(surrogates):
    probs1 = np.load(ss)
    surrogates2.remove(ss)

    for st in surrogates2:
        probs2 = np.load(st)
        stdSUD.append(chstwo(probs1, probs2))

stdSUD = random.sample(stdSUD, 100)
stdP = np.load(standard[0])

del surrogates
surrogates = glob.glob("data/standard/dist/aaft/**.npy")
for ss in tqdm.tqdm(surrogates):
    probs1 = np.load(ss)
    stdORD.append(chstwo(probs1, stdP))
print(len(stdORD), len(henonORD), len(ikedaORD))


########################
## ここから先、描画のみ。##
########################
data = np.concatenate([
    np.array(ikedaSUD), np.array(ikedaORD), [chstwo(ikedaP,
        uniform_dist)],
    # np.array(ikedaSUD2), np.array(ikedaORD2), [chstwo(ikedaP2,
        uniform_dist)],
    np.array(henonSUD), np.array(henonORD), [chstwo(henonP,
        uniform_dist)],
    np.array(stdSUD), np.array(stdORD), [chstwo(stdP, uniform_dist
        )]
])
n_bin = 100
x_max = np.max(data)
x_min = np.min(data)
bins = np.linspace(x_min, x_max, n_bin)


data = [
    ikedaSUD, ikedaORD,
    henonSUD, henonORD,
    stdSUD, stdORD
]

labels = [
    "ikeda:surrogate-surrogate","ikeda:chaos-surrogate",
    "henon:surrogate-surrogate","henon-chaos-surrogate",
    "standard:surrogate-surrogate","standard:chaos-surrogate"
]

colormap = plt.cm.inferno  # 使用するカラーマップ
colors = [colormap(i) for i in np.linspace(0, 1, 10)]  # からまでの範
    囲で色を個生成0110

plt.figure(figsize=(8, 5))
for d_, c_, l_ in zip(data, colors, labels):
    plt.hist(d_, bins=bins, color=c_, label=l_, alpha=0.8)

data = [
    chstwo(ikedaP, uniform_dist), chstwo(henonP, uniform_dist),
        chstwo(stdP, uniform_dist)
]

colormap = plt.cm.viridis
colors2 = [colormap(i) for i in np.linspace(0, 2, 5)]

for d_, c_ in zip(data, colors2):
    plt.vlines(d_, 0, 101, color=c_, linestyles='dotted')
plt.legend()
```

A.5.2 Figure 4.2

Listing A.9: example14.py

```


import numpy as np
import matplotlib.pyplot as plt
from ordpy import ordinal_distribution
import glob
from stats_func import *
import tqdm
import random

# ikeda map u=0.9
ikeda = glob.glob("data/ikeda/dist/original/u090/**.npy")
surrogates = glob.glob("data/ikeda/dist/aaft/u090/**.npy")
surrogates2 = surrogates
data = []

for ss in tqdm.tqdm(surrogates):
    if len(surrogates2)==0:
        break

    probs1 = np.load(ss)
    surrogates2.remove(ss)
    for st in surrogates2:
        probs2 = np.load(st)
        data.append(chstwo(probs1, probs2))

#########################
## ここから先、描画のみ。##
#########################
n_bin = 100
x_max = np.max(data)
x_min = np.min(data)
bins = np.linspace(x_min, x_max, n_bin)

plt.figure(figsize=(8, 5))
plt.hist(data, bins=bins, color="blue", label="surrogate", alpha
    =0.8)
```

A.5.3 Figure 4.3

Listing A.10: example20.py

```

import numpy as np
import glob
import matplotlib.pyplot as plt
from ordpy import ordinal_distribution, ordinal_sequence,
    ordinal_network, complexity_entropy
import tqdm
import random


###################
##  standard map ##
###################

paths_standard = glob.glob("data/standard/dist/aaft/**.npy")
ent1_standard = list()
ent2_standard = list()

for path in paths_standard:
    xx = complexity_entropy(np.load(path), dx=6, probs=True)
    ent1_standard.append(xx[0])
    ent2_standard.append(xx[1])


#############################################
# Diff between surrogate and surrogate #
#############################################

# entropy1
```

```
enn1_standard = ent1_standard
ENT1_standard = list() # list saving data
for x in tqdm.tqdm(ent1_standard):
    enn1_standard.remove(x)
    for y in enn1_standard:
        ENT1_standard.append(x-y)
ENT1_standard = random.sample(ENT1_standard, 100)

# entropy2
enn2_standard = ent2_standard
ENT2_standard = list() # list saving data
for x in tqdm.tqdm(ent2_standard):
    enn2_standard.remove(x)
    for y in enn2_standard:
        ENT2_standard.append(x-y)
ENT2_standard = random.sample(ENT2_standard, 100)


##################################################
# Diff between original chaos and surrogate #
##################################################
xx_standard = complexity_entropy(np.load("data/standard/dist/
    original/dist_0.npy"), dx=6, probs=True)

paths_standard = glob.glob("data/standard/dist/aaft/**.npy")
ent1_standard = list()
ent2_standard = list()

for path in paths_standard:
    xx = complexity_entropy(np.load(path), dx=6, probs=True)
    ent1_standard.append(xx[0])
    ent2_standard.append(xx[1])

diff1_standard = ent1_standard - xx_standard[0]
diff2_standard = ent2_standard - xx_standard[1]
print(len(ent1_standard))
print(len(diff1_standard))

###################
##  ikeda map  ##
###################

paths_ikeda = glob.glob("data/ikeda/dist/aaft/u090/**.npy")
ent1_ikeda = list()
ent2_ikeda = list()

for path in paths_ikeda:
    xx = complexity_entropy(np.load(path), dx=6, probs=True)
    ent1_ikeda.append(xx[0])
    ent2_ikeda.append(xx[1])


```

```
#############################################
# Diff between surrogate and surrogate #
#############################################

# entropy1
print(len(ent1_ikeda))
enn1_ikeda = ent1_ikeda
ENT1_ikeda = list() # list saving data
for x in tqdm.tqdm(ent1_ikeda):
    enn1_ikeda.remove(x)
    for y in enn1_ikeda:
        ENT1_ikeda.append(x-y)
ENT1_ikeda = random.sample(ENT1_ikeda, 100)

# entropy2
enn2_ikeda = ent2_ikeda
ENT2_ikeda = list() # list saving data
for x in tqdm.tqdm(ent2_ikeda):
    enn2_ikeda.remove(x)
    for y in enn2_ikeda:
        ENT2_ikeda.append(x-y)
ENT2_ikeda = random.sample(ENT2_ikeda, 100)


##################################################
# Diff between original chaos and surrogate #
##################################################
xx_ikeda = complexity_entropy(np.load("data/ikeda/dist/original/
   u090/dist_0.npy"), dx=6, probs=True)

paths_ikeda = glob.glob("data/ikeda/dist/aaft/u090/**.npy")
ent1_ikeda = list()
ent2_ikeda = list()

for path in paths_ikeda:
    xx = complexity_entropy(np.load(path), dx=6, probs=True)
    ent1_ikeda.append(xx[0])
    ent2_ikeda.append(xx[1])

diff1_ikeda = ent1_ikeda - xx_ikeda[0]
diff2_ikeda = ent2_ikeda - xx_ikeda[1]


########################
## ここから先描画の調整 ##
########################

# 軸の調整
bins0 = np.concatenate([diff1_standard, ENT1_standard, diff1_ikeda
   , ENT1_ikeda])
bins0 = np.nan_to_num(bins0, nan=np.nanmean(bins0))
n_bin = 100
```

```
x_max = np.max(bins0)
x_min = np.min(bins0)
bins0 = np.linspace(x_min, x_max, n_bin)

bins1 = np.concatenate([diff2_standard, ENT2_standard, diff2_ikeda
    , ENT2_ikeda])
bins1 = np.nan_to_num(bins1, nan=np.nanmean(bins1))
x_max = np.max(bins1)
x_min = np.min(bins1)
bins1 = np.linspace(x_min, x_max, n_bin)

# 描画
plt.rcParams['axes.titlesize'] = 20
plt.rcParams['legend.fontsize'] = "x-large"
plt.rcParams['xtick.labelsize'] = "x-large"  # font size of the
    tick labels
fig, axes = plt.subplots(2, 1, figsize=(10, 15))

# Shannon Entropy
axes[0].set_title("Shannon␣Entropy")
axes[0].hist(diff1_standard, bins=bins0, label="Standard␣Map␣chaos
    -surrogate", alpha=0.5)
axes[0].hist(ENT1_standard, bins=bins0, label="Standard␣Map␣
    surrogate-surrogate", alpha=0.5)
axes[0].hist(diff1_ikeda, bins=bins0, label="Ikeda␣Map␣chaos-
    surrogate", alpha=0.5)
axes[0].hist(ENT1_ikeda, bins=bins0, label="Ikeda␣Map␣surrogate-
    surrogate", alpha=0.5)

# Statistical Complexity
axes[1].set_title("Statistical␣Complexity")
axes[1].hist(diff2_standard, bins=bins1, label="Standard␣Map␣chaos
    -surrogate", alpha=0.5)
axes[1].hist(ENT2_standard, bins=bins1, label="Standard␣Map␣
    surrogate-surrogate", alpha=0.5)
axes[1].hist(diff2_ikeda, bins=bins1, label="Ikeda␣Map␣chaos-
    surrogate", alpha=0.5)
axes[1].hist(ENT2_ikeda, bins=bins1, label="Ikeda␣Map␣surrogate-
    surrogate", alpha=0.5)

for ax in axes:
    ax.legend()

plt.tight_layout(rect=[0, 0, 1, 0.96])
plt.show();
```

A.6 Discussion and Conclusion

A.6.1 Figure 5.1

Listing A.11: example9.py

```python


###########################################
## Calculate Shanon and Complexity Measur
###########################################

import numpy as np
import matplotlib.pyplot as plt
from ordpy import complexity_entropy, maximum_complexity_entropy,
    minimum_complexity_entropy, ordinal_distribution
import warnings
import matplotlib as mpl
import matplotlib.image as mpimg

import string
import glob
import warnings


def stdfigsize(scale=1, nrows=1, ncols=1, ratio=1.5):
    """
    Returns a tuple to be used as figure size.

    Parameters
    ----------
    returns (7*ratio*scale*nrows, 7.*scale*ncols)
    By default: ratio=1.3
    ----------
    Returns (7*ratio*scale*nrows, 7.*scale*ncols).
    """

    return((7*ratio*scale*ncols, 8.*scale*nrows))



# theoretical curves
hc_max_curve = maximum_complexity_entropy(dx=2, dy=3).T
hc_min_curve = minimum_complexity_entropy(dx=2,dy=3, size=719).T

# noise data
hc_knoise = np.load('data/paper/fig3/hc_knoise.npy')
hc_fbm = np.load('data/paper/fig3/hc_fbm.npy')
hc_fgn = np.load('data/paper/fig3/hc_fgn.npy')

# 2D
hc_henon = np.load("hc/2D_hc_henon.npy")
hc_ikeda = np.load("hc/2d_hc_ikeda_dx2dy3.npy")
hc_standard = np.load("hc/2D_hc_standard.npy")
hc_aaft_standard = np.load("hc/each_data/each_hc_standard_dx6.npy"
    )

```

```

# Draw


hc_data = [
    hc_henon[1], hc_ikeda, hc_standard[1], hc_aaft_standard,
    hc_knoise, hc_fbm, hc_fgn
]

# hc_data = np.array(hc_data)

labels = [
    'Henon Map', 'Ikeda Map','Standard Map', 'AAFT fot Standard Map',
    "knoise", "fbm", "fgn"
]

markers = [
    'o', '8', '^', '.',
    'v', '*', 'p'
]

colors = [
    '#3C0912', '#ad39c9', '#10e1ab',
    '#F1ECEB', '#75AABE', '#0C5EBE', '#181C43'
]


plt.figure(figsize=stdfigsize(nrows=1,ncols=1))
for data_, marker_, color_, label_, cnt in zip(hc_data, markers, colors,
                                               labels, range(len(hc_data))):
    #point plotting
    h_, c_ = data_.T
    plt.plot(h_,
             c_,
             marker_,
             markersize=13,
             markeredgecolor='#202020',
             color=color_,
             label=label_)

    if cnt == 1:  #ikeda
        labels = [0.1, 0.3, 0.6, 0.8, 0.85, 0.9, 0.92]
        labels = [str(label) for label in labels]
        indices = [0,1,2,3, 4, 5, 6]
        toriaezu = [0]*len(indices) #とりあえずの位置

        for tx_, x_, y_, adjx_, adjy_ in zip(
            [labels[i] for i in indices],
                h_[indices],
                    c_[indices],
                        toriaezu,
                        toriaezu):

            plt.annotate(r'$u␣=␣{}$'.format(tx_),
                         xy=(x_ + adjx_, y_ + adjy_),
                         fontsize=10,color='#202020')

    #dotted #202020 line connecting dots
    if cnt in [4, 5]:
        plt.plot(h_, c_, '--', linewidth=1, color='#202020',
           zorder=0)

        if cnt == 4:  #colored noise
            adjx_ = [0.015, 0.025, -.005, 0.0]
            adjy_ = [-0.00, -0.005, 0.015, .015]
            ncnt = 0
            for n_, x_, y_ in zip(
                    np.arange(0, 3.1, .25).round(decimals=2), h_,
                       c_):
                if n_ in [0, 1, 2, 3]:
                    plt.annotate('$k␣=␣{}$'.format(int(n_)),
                                   xy=(x_ + adjx_[ncnt], y_ +
                                      adjy_[ncnt]),
                                   fontsize=15,
                                   color='#202020')
                    ncnt += 1
        if cnt == 5:  #fBm
            for tx_, x_, y_, adjx_, adjy_ in zip(['0.1', '0.5', '
               0.9'],
                                                 h_[[0, 4, 8]], c_
                                                    [[0, 4, 8]],
                                                 [-.14, -.13,
                                                    -.04],
                                                 [-0.008, -0.010,
                                                    -0.03]):
                plt.annotate(r'$h␣=␣{}$'.format(tx_),
                               xy=(x_ + adjx_, y_ + adjy_),
                               fontsize=15,
                               color='#202020')




plt.legend(frameon=False, loc=(0, .85), ncol=2, fontsize=15)
plt.ylim(bottom=0, top=.51)
plt.xlim(left=0, right=1.06)
plt.xticks([0, 1.0])
plt.yticks([0, 0.5])


#theoretical curves
```

```
hmin, cmin = hc_min_curve  #(this variable is defined in the cell
    above)
hmax, cmax = hc_max_curve  #(this variable is defined in the cell
    above)
plt.plot(hmin, cmin, linewidth=1.5, color='#202020', zorder=0)
plt.plot(hmax, cmax, linewidth=1.5, color='#202020', zorder=0)

plt.ylabel('Statistical␣complexity,␣$C$')
plt.xlabel('Permutation␣entropy,␣$H$')
plt.annotate('$d_x␣=␣2,␣d_y=3$', (.5, .2),
                va='center',
                ha='center',
                xycoords='axes␣fraction',
                fontsize=30,
                bbox={
                    'boxstyle': 'round',
                    'fc': 'white',
                    'alpha': 1,
                    'ec': '#d9d9d9'
                })

plt.tight_layout()
# plt.savefig('fig5.', dpi=300, bbox_inches='tight')
```

A.6.2 Figure 5.2

Listing A.12: example8.py

```


##########################################
## Calculate Shanon and Complexity Measur
##########################################

import numpy as np
import matplotlib.pyplot as plt
from ordpy import complexity_entropy, maximum_complexity_entropy,
    minimum_complexity_entropy, ordinal_distribution
import warnings
import matplotlib as mpl
import matplotlib.image as mpimg

import string
import glob
import warnings


def stdfigsize(scale=1, nrows=1, ncols=1, ratio=1.5):
    """
    Returns a tuple to be used as figure size.

    Parameters
    ----------
    returns (7*ratio*scale*nrows, 7.*scale*ncols)
    By default: ratio=1.3
    ----------
    Returns (7*ratio*scale*nrows, 7.*scale*ncols).
    """

    return((7*ratio*scale*ncols, 8.*scale*nrows))


# 2D
hc_henon = np.load("hc/2D_hc_henon.npy")
hc_ikeda = np.load("hc/2D_hc_ikeda.npy")
hc_standard = np.load("hc/2D_hc_standard.npy")


hc_data = [
    hc_henon, hc_ikeda, hc_standard
]

hc_data = np.array(hc_data)

labels = [
    'Henon␣Map', 'Ikeda␣Map','Standard␣Map'
]

markers = [
    'o', '8', '^'
]

colors = [
    '#3C0912', '#ad39c9', '#10e1ab',
]

# Draw
plt.rcParams['xtick.labelsize'] = 20
plt.rcParams['ytick.labelsize'] = 20

fig, axes = plt.subplots(1, 3, figsize=stdfigsize(nrows=1,ncols=3))
fig.supylabel('Statistical␣complexity,␣$C$')
fig.supxlabel('Entropy,␣$H$')
for i in range(3):
    for data_, marker_, color_, label_, cnt in zip(hc_data[:, i], markers, colors,
                                                          labels, range(len(hc_data))):
        #point plotting
        h_, c_ = data_.T
        axes[i].plot(h_,
                c_,
                marker_,
                markersize=13,
                markeredgecolor='#202020',
                color=color_,
                label=label_)

    axes[i].legend(frameon=False, loc=(0, .85), ncol=2, fontsize=15)
    axes[i].set_ylim(bottom=0, top=.51)
    axes[i].set_xlim(left=0, right=1.06)


    ddy = [2, 3, 4]
    #theoretical curves
    hc_max_curve = maximum_complexity_entropy(dx=2, dy=ddy[i]).T
    hc_min_curve = minimum_complexity_entropy(dx=2, dy=ddy[i], size=719).T
    #theoretical curves
    hmin, cmin = hc_min_curve  #(this variable is defined in the cell above)
    hmax, cmax = hc_max_curve  #(this variable is defined in the cell above)
    axes[i].plot(hmin, cmin, linewidth=1.5, color='#202020', zorder=0)
    axes[i].plot(hmax, cmax, linewidth=1.5, color='#202020', zorder=0)
    axes[i].annotate(f'$d_x␣=␣2,␣d_y={i+2}$', (.5, .2),
                    va='center',
                    ha='center',
                    xycoords='axes␣fraction',
                    fontsize=30,
                    bbox={
                        'boxstyle': 'round',
                        'fc': 'white',
                        'alpha': 1,
                        'ec': '#d9d9d9'
                    })
# plt.tight_layout()

axes[2].set_xticks([0, 1.0])
axes[2].set_yticks([0, 0.6])
axes[2].set_xticks([0. 1.0])
axes[2].set_yticks([0. 1.0])
plt.savefig('fig_2d.pdf', dpi=300, bbox_inches='tight')
```

List of Figures