Nonlinear Time Series Analysis

Luke Fairbanks, Daniel Primosch

March 2023

Contents

1 Introduction

Time series analysis, especially that of data from systems governed by nonlinear dynamics, can be extremely complex and intractable. The purpose of this paper is to explore, on an admittedly superficial basis, the theoretical underpinnings of a handful of analysis methods and to demonstrate the application and results of such analysis on a handful of test data from various systems.

More than anything we hope to provide a rough roadmap for those wishing to get their toes wet in the subject from a standpoint of programming; however, for further exploration we point to our references & the field more broadly as this is both a well-explored science and an area of intense active research. Before continuing, minor note that many pieces of code provided have descriptive comments within, hence the body of paper might lack explicit explanations of each element within code blocks. Also note many figures are at the bottom of the document. Apologies, but we wanted to make sure everything was sufficiently readable.

2 Time Delay Embedding (TDE)

One of the more ubiquitous and powerful tools of time series analysis is time delay embedding. The method has been formalized through the framework of Taken's theorem discussed below, but the basic idea is pretty simple: useful information from complex systems, such as the form of the phase space attractors, is encoded within even low dimensional observables from the system in the form of changes from one state to the next and so on.

Many metrics exist for characterizing dynamical time series data; however, many methods utilize time delay embedding to reconstruct more workable data objects for analysis. We do not presume monolithic understanding of the nuanced underpinnings of the practice or the scope of implications within data science, for that we point to references such as the work of Floris Taken or those who use the theory as a tool in applied science such as Henry Abarbanel.

Suffice to say we expect relationships of the form

$$\mathbf{x}(n+1) = \mathbf{f}(\mathbf{x}(n), \boldsymbol{\theta})$$

within the time series state variable data where f is a mapping function from the state $\mathbf{x}(n)$ to the next state $\mathbf{x}(n+1)$, typically represented by continuous or discrete differential equations with a set of parameters $\boldsymbol{\theta}$.

A rudimentary understanding of Taken's theorem is that these differential relationships are preserved, locally, under certain phase space transformations, namely observing one of the state variables of the system, $s(t)$ and using it to build a higher dimensional object to represent the system's dynamics. This object in question is the time delay vector

$$[s(t_n), s(t_n - \tau), ..., s(t_n - (D_E - 1)\tau)]$$

where τ and D_E are the the time delay between each snapshot of the time series and the embedding dimension of the vector, respectively. These parameters are technically arbitrary; however, we will discuss in a later section how some optimization can be achieved with respect to these values.

This method is not all-encompassing yet it is a powerful tool in time series analysis which we explore in an application sense and point to our references for further theory.

We'll introduce the Lorenz '63 system soon, but just to demonstrate the time delay reconstruction of phase space dynamics, here are some plots. The first set of plots are the Lorenz system, displayed in the natural (x,y,z) phase space whereas the plots immediately following are displaying a time delay phase space which has been constructed from the x data alone. (figures 1, 2)(figures 3, 4)

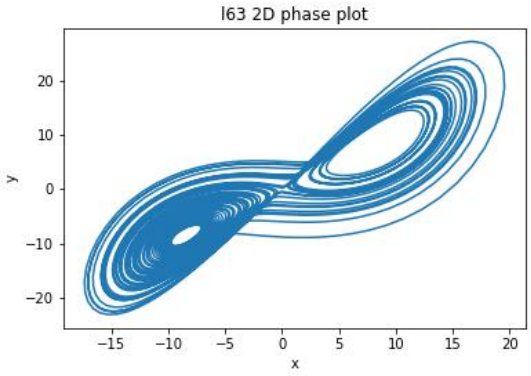


Figure 1: 2D rendering of Lorenz '63 system in the original phase space of (x,y)

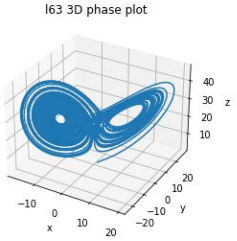


Figure 2: 3D rendering of Lorenz '63 system in the original phase space of (x,y,z)

For clarity purposes here is the code used to produce the data as well as the code used to build up the time delay plots; however, note that some functions within the time delay plots will be displayed later in the paper, namely the

section 'Average Mutual Information & False Nearest Neighbors' (figures 5, 6, 7)

3 Aside: Logistic Map & Lorenz63 as Test-beds for Chaos

Later on in the paper we'll be applying the set of produced algorithms across a handful of different systems to conduct a comparative analysis, in the 'Time Series Examples & Analysis Results Comparison' section.

For the sections between this one and that one, we'll be using a couple familiar dynamical systems to test and demonstrate the algorithm.

The first testing system we'll consider is the logistic map, which can be described by

$$x(n+1) = rx(n) - rx^2(n)$$

The logistic map serves as a convenient test bed because of its simplicity, governed by only one parameter, r, which bifurcates the system from a stable fixed point, to a limit cycle, to chaotic behavior as r is varied from around $r = 2$ to $r = 4$. After this value of r the time series blows up so this is the range we'll consider during testing.

Here is an example of what logistic set data looks like in the chaotic regime of $r = 4$. Note some demonstration plots are on small numbers of datapoints for visualization purposes, but rest assured analyses was conducted on larger samples. (figure 8)

The second testing system will be an oldie but a goodie, Lorenz 63. This dynamical system has been thoroughly explored for decades, and is useful for being relatively simple while displaying slightly more complex behavior than the logistic map. The set of equations is as follows:

$$\frac{dx}{dt} = \sigma(y - x)$$

$$\frac{dy}{dt} = x(\rho - z) - y$$

$$\frac{dz}{dt} = xy - \beta z$$

We again point to sources and broad community-based analysis of these systems for a comprehensive understanding of all their nuance & complexity, but for now we'll be using data simulated from them for testing and demonstration of our algorithms.

Here is a few examples of what the Lorenz data looks like one variable at a time. (figures 9, 10, 11)

4 Average Mutual Information & False Nearest Neighbors Algorithms

We'll be exploring some time series analysis functions from the python library 'Nolds', but as a preliminary we'll discuss a couple methods of optimizing time delay embedding since we found many of the 'Nolds' algorithms utilize time delay embedding within their method. The idea is we want to find the optimal time delay τ and embedding dimension D_E so these 'Nolds' functions give us the most accurate or well-aligned results possible.

To optimize the τ TDE parameter, the main method we utilize average mutual information. In a nutshell, we calculate the average mutual information

$$AMI(\tau) = \sum_{x(t)} \sum_{x(t-\tau)} P(x(t), x(t-\tau)) \log \frac{P(x(t), x(t-\tau))}{P(x(t))P(x(t-\tau))}$$

as a function of the parameter τ where the probability distribution functions are essentially histograms of each vector from the time series, and the joint probability distribution is similarly a 2D histogram with respect to both vectors. The sums are understood to act over the 'bins' of those distributions.

We assess the AMI for each choice of τ with hopes to find a choice of τ which provides a low level of average mutual information, such that the two time delay vectors 'span' the space better.

Following is the set of functions developed for the calculation of the average mutual information for time series data: (figures 12-17)

Now shown are the resultant output from the 'AMIanalysis' function wrt data from the logistic $r = 4$ map and the Lorenz '63 system. We show results from L63 x, y, and z to demonstrate how different variables don't always give the same output even when from the same system. This fact is worth further analysis, but for now: (figures 18-21)

Now considering the other relevant TDE parameter, embedding dimension D_E, we want to conduct a similar optimization process, but here we utilize the idea of false nearest neighbors.

The core idea is that the system of interest's attractor in phase space needs to have a high enough dimensional space to be fully 'unfolded' when conducting time delay embedding, while also not arbitrarily assigning an unnecessarily large dimension since that can lead to corruption by high dimensional noise in the signal. When the data is sampled from the original phase space down onto a low dimensional observable state space such as a single state variable, we find the complexity of the attractor is flattened which leads to many points appearing to be close neighbors to each other despite existing far away within the real phase space, false neighbors. We conduct an analysis where the embedding dimension is increased steadily and the percentage of false nearest neighbors is monitored to see what dimensionality sufficiently 'unfolds' the attractor in the new TD phase space such that the false nearest neighbor percentage is minimized.

Following are the code blocks to conduct this analysis, note some functions

are called upon yet not displayed since they have been displayed elsewhere. In other words avoiding redundancy. (figures 22-25)

Now for some results from the FNN vs D_E analysis function, again on the logistic $r = 4$ data and the Lorenz '63 system: (figures 26-29)

Now that we have some functions to conduct the AMI and FNN analysis, the only thing left is to use the determined τ and D_E to run the 'Nolds' algorithms for some insightful metrics. Before we do that; however, we'd like to automate this process further by creating algorithms which determine the optimal τ and D_E computationally rather than always needing human judgement, at least in simple cases.

5 Time Delay Optimization Algorithms

Here we'd like to take a minor tangent into a handful of algorithms designed to accommodate full automation of the time series analysis process. (figures 30-40)

6 Python 'Nolds' Time Series Analysis Functions

Now that we have algorithms to conduct the AMI and FNN analyses and subsequently find the TDE parameters τ and D_E, we can run some of these time series metric algorithms from the 'Nolds' python library with a bit more confidence that the results we output will be reasonable or at least informed somewhat by preliminary data analysis.

First we'll briefly explain the metrics contained in the python library, and their implications wrt dynamical systems and their time series. We don't presume thoroughly exhaustive elucidations of each metric, for sake of brevity, but would be open to a deeper dive into the implications and scope of all of these functions in a future study.

1. (Rosenstein algorithm) Maximal Lyapunov Exponent: The lyapnuov exponent is ubiquitous in dynamical system analysis since it corresponds to the exponential divergence rate of trajectories in phase space, according to a formulation such as

$$|\delta \boldsymbol{Z}(t)| = \exp \lambda t |\delta \boldsymbol{Z}(0)|$$

where the $|\delta \boldsymbol{Z}(0)|$ corresponds to the intial separation between two points in phase space, the $|\delta \boldsymbol{Z}(t)|$ is the separation at some late time t, and the lyapunov exponent is the exponential scaling parameter λ. One of the basic insights to glean from this number is that if the value is positive, it means the trajectories diverge exponentially hence the dynamics are chaotic in that initial condition perturbations produce wildly different

results. This algorithm is tuned to focus on zeroing in on the largest lyapunov exponent in the system since the largest one is typically dominant ie. controls the dynamics in large degree.

2. (Eckman algorithm) Lyapunov Exponent Spectrum: this algorithm attempts to find a set of lyapunov exponents rather than just the largest. A condensed way to describe this set of numbers is that they represent the eigenvalues of the Oseledets matrix which is concerned with the evolution of phase space trajectories in tangent space according to the Jacobian of the system. Note with these lyapunov algorithms that the results depend upon the parameters such as time delay embedding parameters, hence the preliminary work, as well as things like which attractor your choice of initial conditions leads to.

3. Sample Entropy: The sample entropy of a time series is the negative natural logarithm of the conditional probability that two sequences similar for D_E points remain similar at the next point, excluding self-matches. Therefore, A lower value for the sample entropy therefore corresponds to a higher probability indicating more self-similarity. Self-similarity is often of interest to both analysts and laypeople alike because it helps describe the 'fractal' or fraction-dimensional nature of the attractor which can be both a beautiful thing to visualize as well as insightful wrt how the dynamics are structured.

4. Hurst Exponent: a measure of the long-range statistical dependencies within the data which do not originate from simple cycles, in other words the "long-term memory". In a nutshell we examine the data of a series, and calculate the cumulative sum of the data. We look at the range R of this cumulative sum and find the hurst exponent K within the formula

$$\frac{R}{\sigma} = (\frac{N}{2})^K$$

where N is the data series length and $\sigma$is the standard deviation of the series.

5. (Grassberger-Procaccia algorithm) Correlation Dimension: describes the geometry of the phase space attractor, or the 'strangeness' of the attractor. Defined using the correlation sum $C(r)$ which is the faction of pairs of points in the space whose distance is smaller than r. Then the correlation dimension follows as the D within

$$C(r)\ r^D$$

6. Detrended Fluctuation Analysis: Another measure of the long term statistical dependencies within the data. 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 $STD(X, n)$ is the standard deviation of the process over window of length n. The parameter of interest, 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 $\log n$. An estimate of H, α is obtained via a process we won't elucidate fully here related to the 'detrended' nature of the analysis. Suffice to say 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 $H = \alpha - 1$.

Before moving on to the 'ensemble' algorithm which combines all the metrics listed above into one display for a given dataset, let's see the results of each algorithm on logistic set data as the parameter r is varied slowly from 2 to 4. Recall that this parameter rising tunes the output data to become more and more nonlinear and chaotic.

We admittedly need further testing and validation of this whole pipeline on various data from different systems with known benchmark values for these metrics to prove to ourselves that everything is working as intended and refine our interpretability of the metrics; however, for now due to time constraints our analysis and discussion therein is limited.

One last note before displaying the (metric vs logistic r value) plots we note the maximum laypunov from the eckman spectrum is chosen for display purposes rather than the entire set of spectrum exponents. (code figures 41-43)(results figures 44-49)

7 Ensemble Analysis Algorithm

Finally, let's see some algorithms which boil all of the analyses above together into a single function or two.

Following, in figures 50 and 51, are a couple functions to run all of the metrics from 'nolds' onto a time series, as well as a function to print the results out in a more readable format. (figures 50, 51)

For good measure here is a set of results from the algorithm, run, of course, on logistic $r = 4$ (figure 52)

Why go through all the trouble of building algorithms to automate the choice of τ and D_E and not build the subsequent function to automate the entire process? Also shown are the automated process run, again on the test example logistic $r = 4$ (figure 53, 54)

8 Time Series Examples & Analysis Results Comparison

In this section we display some results from the algorithm applied onto different systems. Rather than exhaustively posting every figure produced, we'll try to keep it to just a handful of examples for the sake of brevity. Also note some results for the Colpitt's oscillator will be displayed after the theory section below

The following parameters were used for most of the datasets. Further exploration is needed to see if any further optimization can be gleaned from tuning these, they were found mostly sufficient. The only parameter which displayed any issues so far was 'numSTDallowed' which controls how generous the FNN analysis algorithm is wrt how it classifies adjacent points as neighbors. We found a value of 15 was usually good, but certain systems required a smaller neighborhood threshold to converge. (parameters figures 55-57)

Lorenz '63 analysis results for x, y, and z. Note how the data is from the same system yet metric results have some variance, further study needed to nail down get consistent answers. (figures 58-60)

For now we'll continue since the remaining results are regarding the colpitt's oscillator, tbd.

9 Colpitts Oscillator

The Colpitts oscillator is an example of a circuit whose behavior is non-linear. This non-linearity comes from the bipolar transistor that is part of the circuit. The circuit is biased with a constant voltage and produces a sinusoidal oscillation as output. This is achieved via a feedback loop created by a combination of the transistor, capacitors, and an inductor (figure 62).

With appropriate choices of resistances, the oscillator produces a stable signal in the RF regime, making it useful for various applications in radio and telecommunication.

The Colpitts oscillator can be modeled via three coupled ordinary differential equations derived using Kirchhoff's Law. These equations take the following form:

$$\begin{aligned} C_1\frac{dV_{CE}(t)}{dt} &= I_L(t) - I_S\exp(\frac{-V_E(t)}{V_T}),\\ C_2\frac{dV_E(t)}{dt} &= I_L(t) - \frac{V_E(t)-V_{ee}}{R_e e} + \frac{I_S}{\beta_F}I_B\exp(\frac{-V_E(t)}{V_T}),\\ L\frac{dI_L(t)}{dt} &= V_{CC} - V_E(t) - V_{CE}(t) - RI_L(t), \end{aligned} \tag{1}$$

where β_F is the forward current gain of the transistor and I_S is the reverse saturation current. $V_T = kT/e$ is the thermal voltage.The exponential terms come from a simplified version of the Ebers-Moll equations, which can be used to model the transistor current.

9.1 Bifurcations and Chaos

From the perspective of dynamical systems, the Colpitts oscillator operates in three main regimes dictated by the forcing of the circuit. The bias voltage across the circuit constitutes this driving force of the oscillation. For a very small bias voltage (no forcing), the dynamics of the circuit unsurprisingly result in a fixed point in the state space. Increasing the bias voltage results in bifurcations to limit cycle periodic oscillations of the emitted signal. Increasing the driving force even further puts the system into the third regime: chaotic oscillations. For practical applications, the oscillator is typically kept in the second regime. We will simulate the oscillator in both the second and third regimes. Furthermore, we will use a physical circuit to generate data which will allow us to determine which regime the circuit operated in.

9.2 Simulating the Colpitts Oscillator

We can make the differential equations of the Colpitts oscillator circuit dimensionless and obtain the following:

$$\begin{aligned} \frac{dy_1(t)}{dt} &= \alpha_D y_2(t), \\ \frac{dy_2(t)}{dt} &= -\gamma_D(y_1(t) + y_3(t)) - q_D y_2(t), \\ \frac{y_3(t)}{dt} &= \eta_D(y_2(t) + 1 - \exp(-y_1(t))), \end{aligned} \tag{2}$$

where α_D, γ_D, and η_D are dimensionless constants that depend on the Colpitts circuit parameters. Specifically, α_D represents the driving force, which we can adjust to obtain the oscillator behavior in different regimes.

While these equations can't be solved analytically, we can solve them numerically using various integration schemes. Here, we choose the 4th order Runge-Kutta integration method, which makes a reasonably accurate estimate of the state after a time-step (figure 63).

For $\alpha_D = 1$, we observe what resembles approximately a limit cycle (up to deviations due to numerical noise) (figure 64). Plotting this in 3 dimensions further illustrates that the state of the system remains within a bounded area of state space (figure 65). Increasing this value to $\alpha_D = 5$, we observe a strange attractor. Here, the trajectory is not periodic and appears to diverge. This suggests chaotic behavior (figure 66). However, a visual inspection of the state space trajectory alone is not enough to confirm the presence of chaos.

10 Example Circuit

10.1 Description

To obtain data from a real circuit we used a Colpitts oscillator circuit built by Dan Creveling at Los Alamos National Laboratory (figure 67). The output of

this circuit is buffered and corresponds to

$$\begin{aligned} \frac{dz(t)}{dt} &= 200 I_L(t), \\ \frac{dy(t)}{dt} &= 5(V_C(t) - 12), \\ \frac{dx(t)}{dt} &= 5 V_E, \end{aligned} \tag{3}$$

where $I_L(t)$, $V_C(t)$, and $V_E(t)$ are defined by equations (1). The key values of this circuit are that of the inductor, $L = 0.93mH$, and the capacitors, $C_1 = 0.653$ and $C_2 = 0.654$. We were only able to provide the circuit with a bias of 10 V rather than the required 12 V, but still obtained sensible data.

The data was taken by connecting the circuit to a power source and connecting the buffered outputs to voltage probes designed to interact with the LoggerPro student laboratory software.

Plotting the data-points of the x,y-space of this circuit, we can see a state space trajectory that approximately resembles that of a limit-cycle (up to deviations introduced by noise), suggesting that the circuit is set to operate in a non-chaotic regime (figures 68,69). While this constitutes a good first analysis of the system, more rigorous approaches like the computation of the maximal Lyapunov exponent need to be performed to confirm the chaotic nature of the system. However, this proves computationally expensive for some systems, making a visual analysis as the one above more expedient.

11 Colpitts 'bigAnalysis' results

This section references our results of the simulated and real Colpitts oscillator data from the 'nolds' algorithms

Simulated Colpitts results (figures 70-72)

Physical circuit colpitt's 'bigAnalysis' results (figures 73-75)

12 Note: Tisean Software

Another software which has been utilized by many people & institutions do conduct similar analyses to ours as well as a handful of others, namely the Tisean program. Suffice to say it offers expansive functionality for nonlinear time series analysis. We had it working but not on a production level so we decided to simply reference it and note that we're still extending our work to include the methods provided within.

13 Disclaimer

Want to note that the algorithms shown above are still a work in progress and require further verification, validation, extension, and automation.

14 references & acknowledgments

1. Analysis of Observed Chaotic Data; Henry Abarbanel; ISBN: 978-1-4612-0763-4

2. Nonlinear Time Series Analysis; Holger Kantz, Max-Planck-Institut für Physik komplexer Systeme, Dresden, Thomas Schreiber, Max-Planck-Institut für Physik komplexer Systeme, Dresden; 9780511755798

3. Predicting the Future; Henry Abarbanel; 2013 Nolds documentation

4. Tisean documentation

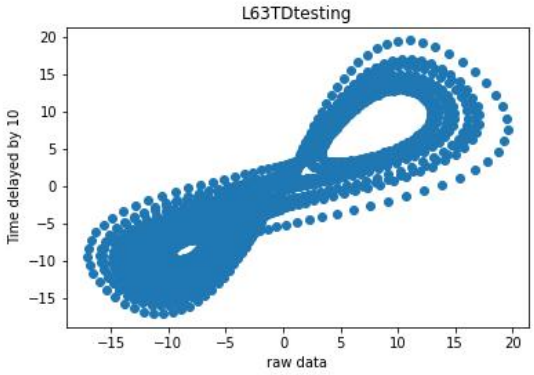


Figure 3: 2D rendering of the Lorenz 63' x variable which has been up-projected into a higher dimensional space via time delay embedding

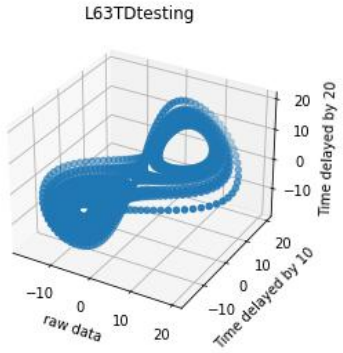
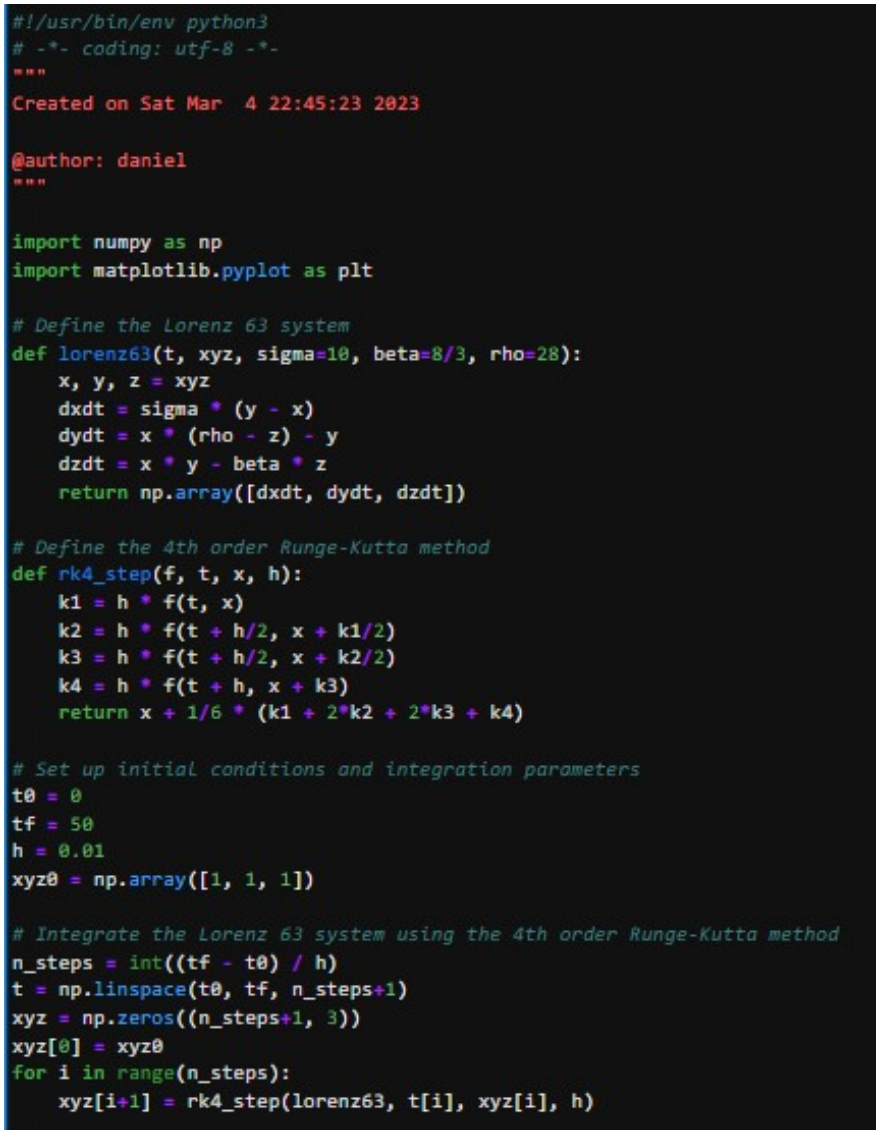


Figure 4: 3D rendering of the Lorenz 63' x variable which has been up-projected into a higher dimensional space via time delay embedding



```python
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Created on Sat Mar  4 22:45:23 2023

@author: daniel
"""

import numpy as np
import matplotlib.pyplot as plt

# Define the Lorenz 63 system
def lorenz63(t, xyz, sigma=10, beta=8/3, rho=28):
    x, y, z = xyz
    dxdt = sigma * (y - x)
    dydt = x * (rho - z) - y
    dzdt = x * y - beta * z
    return np.array([dxdt, dydt, dzdt])

# Define the 4th order Runge-Kutta method
def rk4_step(f, t, x, h):
    k1 = h * f(t, x)
    k2 = h * f(t + h/2, x + k1/2)
    k3 = h * f(t + h/2, x + k2/2)
    k4 = h * f(t + h, x + k3)
    return x + 1/6 * (k1 + 2*k2 + 2*k3 + k4)

# Set up initial conditions and integration parameters
t0 = 0
tf = 50
h = 0.01
xyz0 = np.array([1, 1, 1])

# Integrate the Lorenz 63 system using the 4th order Runge-Kutta method
n_steps = int((tf - t0) / h)
t = np.linspace(t0, tf, n_steps+1)
xyz = np.zeros((n_steps+1, 3))
xyz[0] = xyz0
for i in range(n_steps):
    xyz[i+1] = rk4_step(lorenz63, t[i], xyz[i], h)
```

Figure 5: The code used to simulate the Lorenz '63 system, credit Daniel Primosch

```
#builds a 2 dimensional time delay array and plots the points as a scatter plot in time delay space
#data = time series data, TDmult = time delay step scaling, windowLength = data cut length, title = string for plot title, path = .jpg save path
def TDplot2D(data, TDmult, windowLength, title, path):
    arr = TDset(data, windowLength, 2, TDmult)
    x = arr[:,0]
    y = arr[:,1]
    fig = plt.figure()
    ax = fig.add_subplot()
    ax.scatter(x, y)
    ax.set_xlabel('raw data')
    ax.set_ylabel('Time delayed by ' + str(TDmult))
    ax.set_title(title)
    plt.savefig(path + '/TD2Dplot' + titleNow + '.jpg')
    plt.show()
```

Figure 6: The function used to produce time delay plots in 2D wrt embedding dimension, credit Luke Fairbanks

```
#builds a 3 dimensional time delay array and plots the points as a scatter plot in time delay space
#data = time series data, TDmult = time delay step scaling, windowLength = data cut length, title = string for plot title, path = .jpg save path
def TDplot3D(data, TDmult, windowLength, title, path):
    arr =  TDset(data, windowLength, 3, TDmult)
    x = arr[:,0]
    y = arr[:,1]
    z = arr[:,2]
    fig = plt.figure()
    ax = fig.add_subplot(111, projection='3d')
    ax.scatter(x, y, z)
    ax.set_xlabel('raw data')
    ax.set_ylabel('Time delayed by ' + str(TDmult))
    ax.set_zlabel('Time delayed by ' + str(2*TDmult))
    ax.set_title(title)
    plt.savefig(path + '/TD3Dplot' + titleNow + '.jpg')
    plt.show()
```

Figure 7: The function used to produce time delay plots in 3D wrt embedding dimension, credit Luke Fairbanks

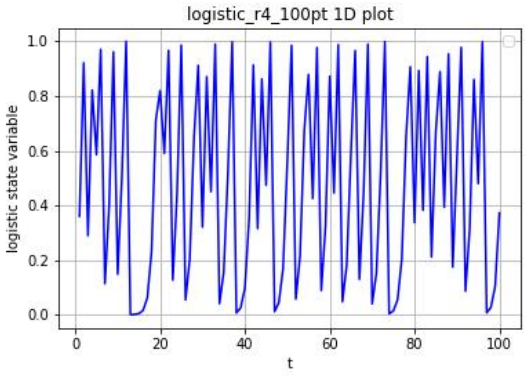


Figure 8: 100 points from Logistic r=4

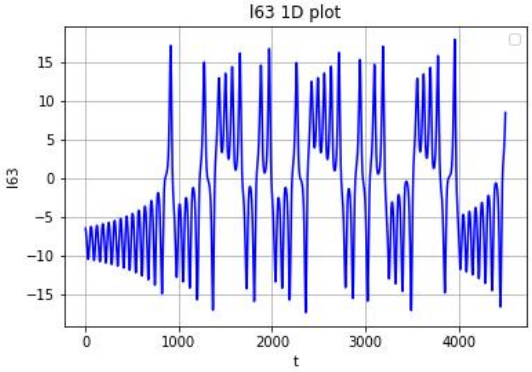


Figure 9: Lorenz '63 X variable

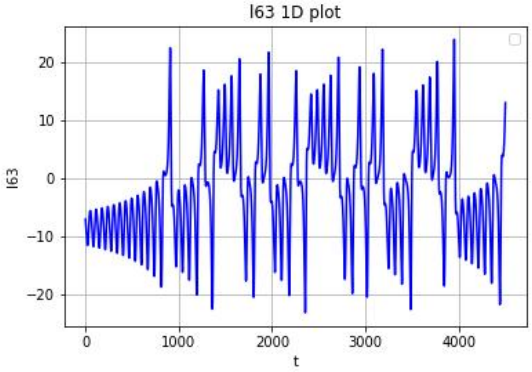


Figure 10: Lorenz '63 Y variable

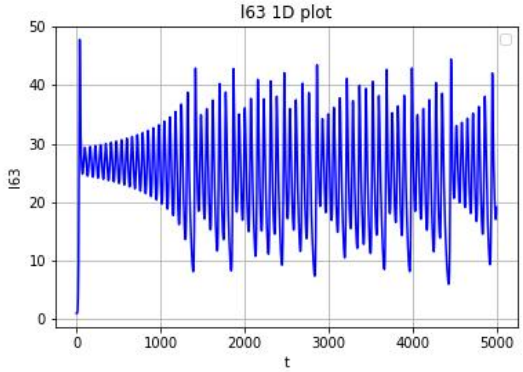


Figure 11: Lorenz '63 Z variable

```
#time delay vector maker (function)
#takes in a one dimensional vector, implements a time delay via a tau = step*multiplier = n*T where T is a multiple of sampling rate timing
#returns a time delayed vector for use in time delay embedding

def TDvec(vec, step, l, multiplier): #vec = time series data vector, step = 'shift' value, L = length of data window cut, multiplier = scaling for step shifts
    delay = step*multiplier #establish the tau = delay
    new = vec[delay:(l + delay)] #cut a 'window' out of data
    return new
```

Figure 12: create a single time delay vector

```
#returns an array where each column of the array is a sample from the time series at a different time delay tau
def TDset(X, l, num, mult): #X = time series data vector, L = length of data window cut, num = number of columns or TD shifts, mult = TD shift scaling
    shape = (l,num)
    arr = np.empty(shape)
    for i in range(num):
        arr[:, i] = TDvec(X, i, l, mult) #call time delay vector generator
    return arr
```

Figure 13: create a set of TD vectors

```
#returns a set of histograms where each histogram corresponds to a vector of the original array
def histAll(data, binNum): #data = input array, binNum = number of different bins the data should be sorted into/'resolution' of histogram
    dataShape = np.shape(data)
    histSetShape = (binNum, dataShape[1]) #num of rows is num of hist bins, num of columns for each diff hist
    fullHist = np.empty(histSetShape)
    for i in range(dataShape[1]): #iterate over each vector of data
        hist, bins = np.histogram(data[:,i], bins = binNum) #histogram operations, numpy function
        fullHist[:,i] = hist
    return fullHist
```

Figure 14: create a set of histograms for the time delay vector array

```
#return JOINT probability distributions of the (undelayed) first vector in an array with all of the following (delayed) vectors
def jointSet(data, binNum): #data = input array, binNum = number of different bins the data should be sorted into/'resolution' of histogram
    numCombos = (np.shape(data)[1]-1) #since just comparisons with 1st vector
    arrShape = (binNum, binNum, numCombos) #note assuming SYMMETRY in binNum for 2D histograms ie symmetrical resolution
    arr = np.empty(arrShape)
    x = data[:,0]
    for i in range(numCombos):
        y = data[:,i+1]
        H, xedges, yedges = np.histogram2d(x, y, bins=binNum) #call histogram2d from numpy
        arr[:,:,i] = H
    return arr
```

Figure 15: create set of 2D histograms between the undelayed vector and the delayed ones

```
#return the average mutual information SET where each element is the AMI between the original data vector and a delayed vector
def AMI(singles, joints): #singles = histograms from the histALL function, joints = 2d histograms from the jointSet function
    index1 = np.shape(joints)[0]
    index2 = np.shape(joints)[1]
    dist1 = singles[:,0] #define the (undelayed) reference histogram
    index3 = np.shape(joints)[2]
    AMIset = np.empty(index3)
    for k in range(index3): #for the wider sum in the AMI function
        count=0
        dist2 = singles[:,(k+1)]
        for i in range(index1):
            for j in range(index2):
                count += joints[i,j,k]*mutual_info_score(dist1, dist2)  #where the average mutual information is added to, double sum for 2D histogram indices
        AMIset[k] = count
    return AMIset
```

Figure 16: calculate the AMI given sets of histograms and joint histograms

```
#plots the results of average mutual information vs time delay for use in determining the 'optimal' tau
#data - from time series, dataWindowLength - self explanatory, numComparisons - 'how many time delays to do?'
#numBins - resolution of histograms, TDmultiplier - how large the steps do we want TD vectors to take
#NOTE takes a while to run, depending upon histogram resolution since many sort operations to conduct
def AMIanalysis(data, dataWindowLength, numComparisons, numBins, TDmultiplier):
    TDsetNow = TDset(data, dataWindowLength, numComparisons, TDmultiplier) #form set of time delay vectors
    histAllNow = histAll(TDsetNow, numBins) #determine histograms for each time delay
    jointSetNow = jointSet(TDsetNow, numBins) #determine joint probability histograms between undelayed vector and delayed vector set
    AMIsetNow = AMI(histAllNow, jointSetNow) #determines the AMIs
    plotTimeVec = np.linspace(1, (numComparisons-1), (numComparisons-1)) #plot the results
    plt.plot(plotTimeVec, AMIsetNow)
    plt.xlabel('Time delay steps, units = ' + str(TDmultiplier) + 'steps')
    plt.ylabel('Average Mutual Information')
    plt.show()
```

Figure 17: conduct the entire AMI analysis process and plot results

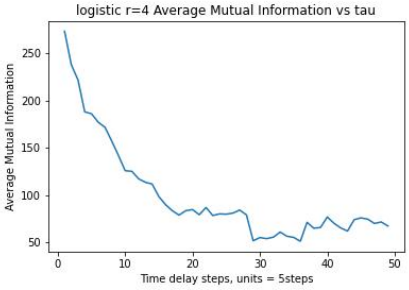


Figure 18: Average mutual information for the logistic map where $r = 4$

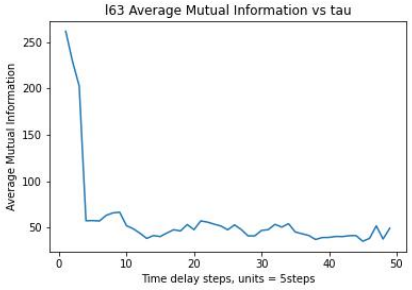


Figure 19: Average mutual information for Lorenz '63 x variable

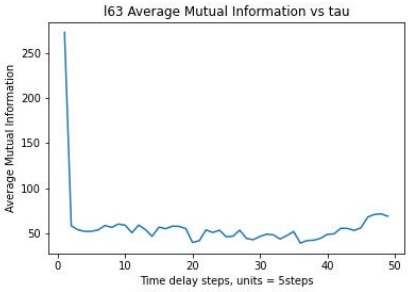


Figure 20: Average mutual information for Lorenz '63 y variable

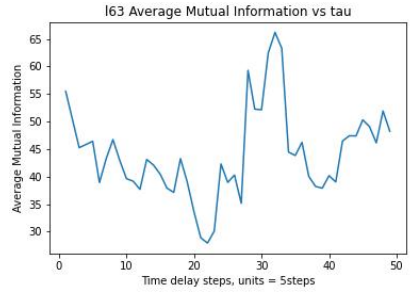


Figure 21: Average mutual information for Lorenz '63 z variable

```
#returns an array of time delayed vectors with the dimensionality of the array defined by DE
#eg. if DE = 2 only original data & 1 delayed vector, if DE = 5 original vector and 4 vectors with increasingly more delay
def TD_DE_object(data, step, L, multiplier, DE): #data = time series data, step = 'shift' value, L = window length, mult = step scaling, DE = dimensionality
    shape = (L, DE)
    arr = np.empty(shape)
    for i in range(DE):
        arr[:,i] = TDvec(data, i*step, L, multiplier) #call TD vec func
    return arr
```

Figure 22: creates array of time delay vectors with dimensionality D_E

```
#calls the nearest neighbor function of a dataset to determine the average distance and std distance between neighbors
#then returns a value, used for FNN analysis, which is a certain distance 'threshold' for neighbors to be classified with
def threshold(data, numSTDallowed): #data = time series data, numSTDallowed = 'how big do we want threshold to be?', typically choose between 3-15
    nbrs = NearestNeighbors(n_neighbors=2, algorithm='auto').fit(data) #note 'auto' means algo choosest 'optimal' method
    distances, indices = nbrs.kneighbors(data)
    avg = np.mean(distances[:])
    STD = np.std(distances[:])
    value = avg + STD*numSTDallowed #threshold value, NOTE how numSTDallowed can drastically change output of FNN analysis
    #eg very large numSTDallowed => all points have neighbors FNN constant 100%, very small numSTDallowed => 'neighborhood' tiny, FNN rapidly converges to 0
    return value
```

Figure 23: defines the threshold within which adjacent points are considered 'neighbors'

```
#returns the % of a set of data which has a nearest neighbor within 'threshold' distance
def percentNN(data, threshold): #data = time series vector, threshold = as per 'threshold' function
    nbrs = NearestNeighbors(n_neighbors=2, algorithm='auto').fit(data)
    distances, indices = nbrs.kneighbors(data)
    percent = np.sum(distances <= threshold, axis=0)[1] / data.shape[0] #where % NN is determined from the distances
    return percent
```

Figure 24: calculates the percentage of false nearest neighbors of the time delay array object in question

```
#plots the results of an analysis of false nearest neighbors vs embedding dimension
#data = time series vector, step - 'shift' for each TD vector, L = data window length, multiplier = scaling for TD shifts
#MDE_max = total number of different dimensionalities to try, numSTDallowed = size of 'threshold' for nearest neighbors
#NOTE numSTDallowed can affect results significantly, see 'threshold function for detail
#tldr: if FNN very high or near 1, lower numSTDallowed. if FNN near 0 for all values, raise numSTDallowed. suggest numSTDallowed between 3 and 15
def FNNanalysis(data, step, L, multiplier, DE_max, numSTDallowed):
    FNNvsDE = np.empty(DE_max)
    thresholdData = TD_DE_object(data, 0*step, L, multiplier, 1) #use undelayed data for threshold
    thresh = threshold(thresholdData, numSTDallowed) #define the FNN threshold
    FNNvsDE[0] = percentNN(thresholdData, thresh)  #first element of result, should be 100% or 1
    for i in range(DE_max-1):
        actualDimension = i+2 #used to avoid some issues with using 0 in certain functions
        arr = TD_DE_object(data, step, L, multiplier, actualDimension) #form a new array with dimensionality corresponding to loop iterator
        FNNvsDE[actualDimension - 1] = percentNN(arr, thresh) #FNN for the current TD system with dimensionality actualDimension
    plotDEVec = np.linspace(1, (DE_max), (DE_max))
    plt.plot(plotDEVec, FNNvsDE)
    plt.xlabel('Time delay embedding dimension')
    plt.ylabel('False Nearest Neighbor percentage')
    plt.show()
```

Figure 25: Conducts the entire FNN vs D_E analysis and plots results

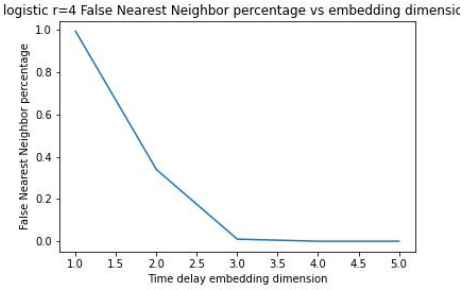


Figure 26: False Nearest Neighbors for the logistic map where $r = 4$

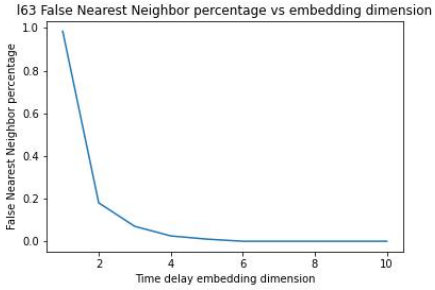


Figure 27: False Nearest Neighbors for Lorenz '63 x variable

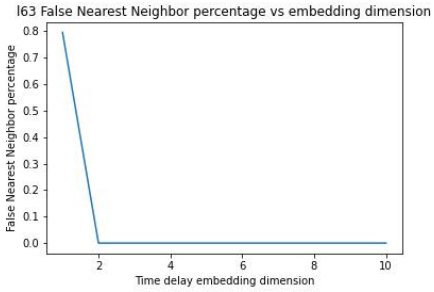


Figure 28: False Nearest Neighbors for Lorenz '63 y variable

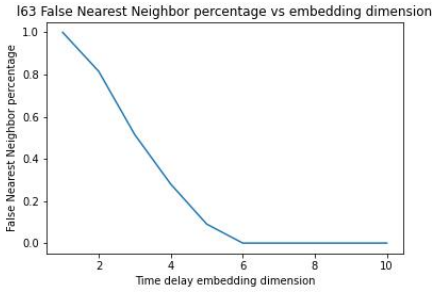
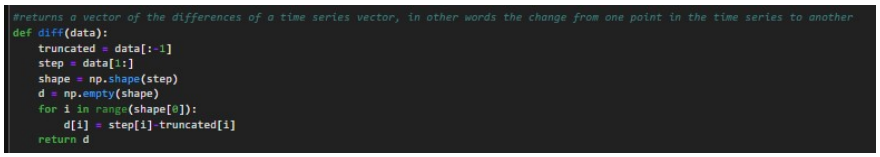
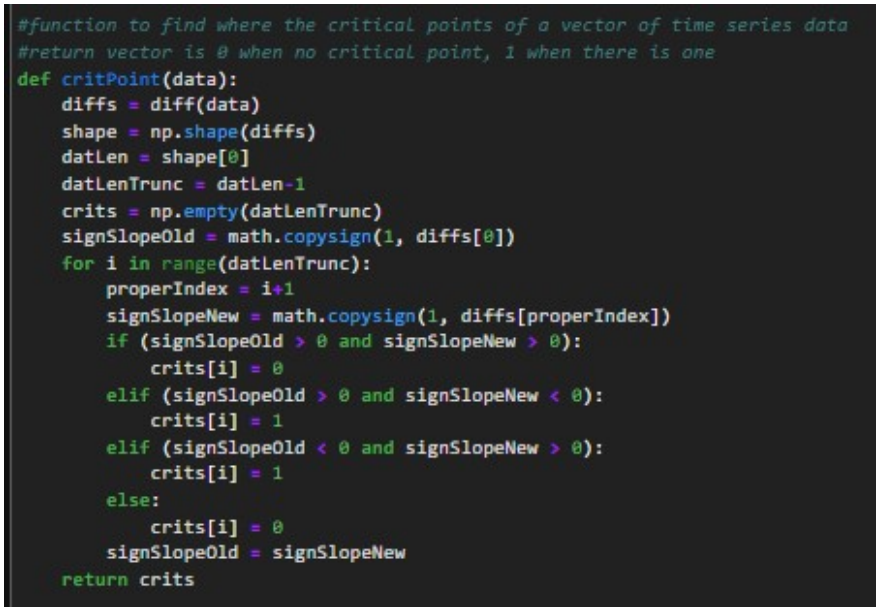


Figure 29: False Nearest Neighbors for Lorenz '63 z variable



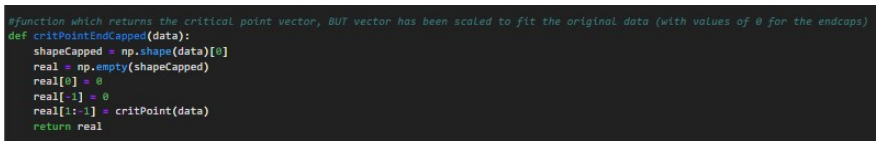
```
#returns a vector of the differences of a time series vector, in other words the change from one point in the time series to another
def diff(data):
    truncated = data[:-1]
    step = data[1:]
    shape = np.shape(step)
    d = np.empty(shape)
    for i in range(shape[0]):
        d[i] = step[i]-truncated[i]
    return d
```

Figure 30: returns the changes from one point in time series to another as a vector



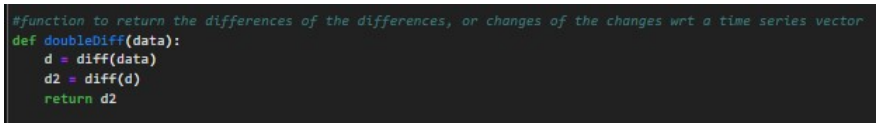
```
#function to find where the critical points of a vector of time series data
#return vector is 0 when no critical point, 1 when there is one
def critPoint(data):
    diffs = diff(data)
    shape = np.shape(diffs)
    datLen = shape[0]
    datLenTrunc = datLen-1
    crits = np.empty(datLenTrunc)
    signSlopeOld = math.copysign(1, diffs[0])
    for i in range(datLenTrunc):
        properIndex = i+1
        signSlopeNew = math.copysign(1, diffs[properIndex])
        if (signSlopeOld > 0 and signSlopeNew > 0):
            crits[i] = 0
        elif (signSlopeOld > 0 and signSlopeNew < 0):
            crits[i] = 1
        elif (signSlopeOld < 0 and signSlopeNew > 0):
            crits[i] = 1
        else:
            crits[i] = 0
        signSlopeOld = signSlopeNew
    return crits
```

Figure 31: returns vector to indicate where critical points occur in the time series data



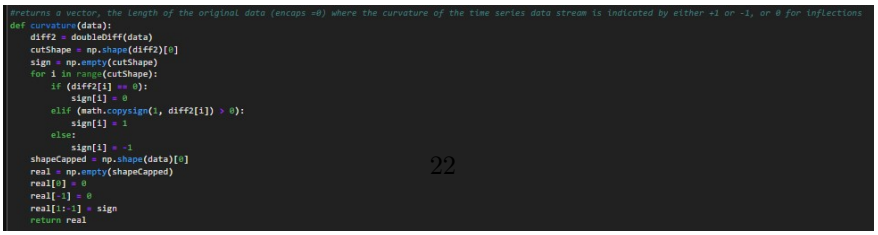
```
#function which returns the critical point vector, BUT vector has been scaled to fit the original data (with values of 0 for the endcaps)
def critPointEndCapped(data):
    shapeCapped = np.shape(data)[0]
    real = np.empty(shapeCapped)
    real[0] = 0
    real[-1] = 0
    real[1:-1] = critPoint(data)
    return real
```

Figure 32: produces a critical point vector, but properly scaled to match the original data vector's length



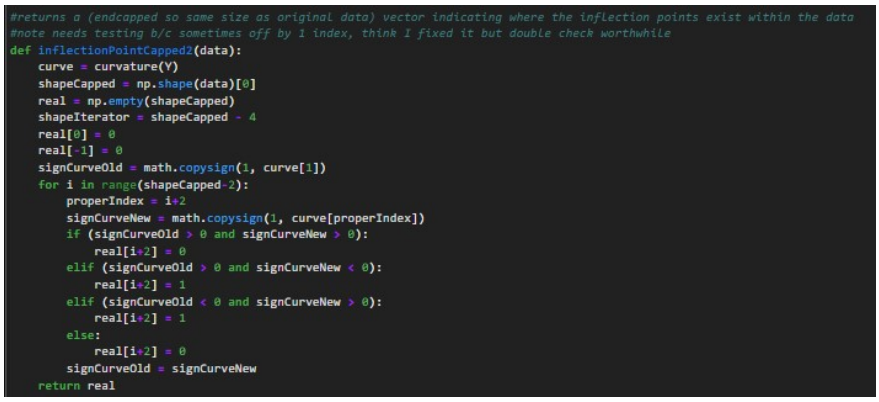
```
#function to return the differences of the differences, or changes of the changes wrt a time series vector
def doubleDiff(data):
    d = diff(data)
    d2 = diff(d)
    return d2
```

Figure 33: produces a vector of 2nd order changes, ie changes of the changes within time series



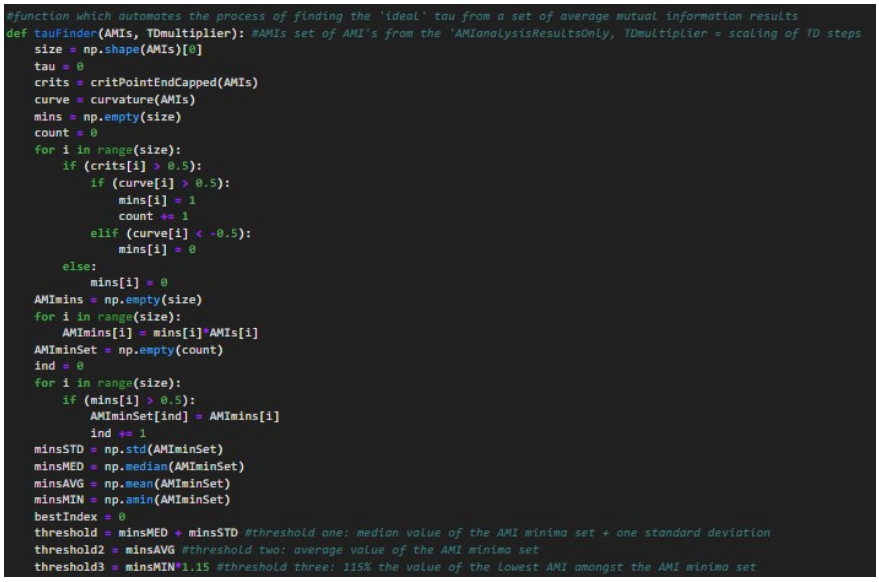
```
#returns a vector, the length of the original data (encaps +0) where the curvature of the time series data stream is indicated by either +1 or -1, or 0 for inflections
def curvature(data):
    diff2 = doubleDiff(data)
    cutShape = np.shape(diff2)[0]
    sign = np.empty(cutShape)
    for i in range(cutShape):
        if (diff2[i] == 0):
            sign[i] = 0
        elif (math.copysign(1, diff2[i]) > 0):
            sign[i] = 1
        else:
            sign[i] = -1
    shapeCapped = np.shape(data)[0]
    real = np.empty(shapeCapped)
    real[0] = 0
    real[-1] = 0
    real[1:-1] = sign
    return real
```

Figure 34: produces a vector indicating the curvature of the data at each point



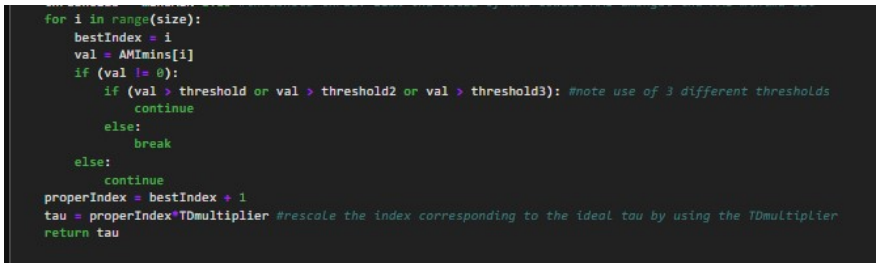
```
#returns a (endcapped so same size as original data) vector indicating where the inflection points exist within the data
#note needs testing b/c sometimes off by 1 index, think I fixed it but double check worthwhile
def inflectionPointCapped2(data):
    curve = curvature(Y)
    shapeCapped = np.shape(data)[0]
    real = np.empty(shapeCapped)
    shapeIterator = shapeCapped - 4
    real[0] = 0
    real[-1] = 0
    signCurveOld = math.copysign(1, curve[1])
    for i in range(shapeCapped-2):
        properIndex = i+2
        signCurveNew = math.copysign(1, curve[properIndex])
        if (signCurveOld > 0 and signCurveNew > 0):
            real[i+2] = 0
        elif (signCurveOld > 0 and signCurveNew < 0):
            real[i+2] = 1
        elif (signCurveOld < 0 and signCurveNew > 0):
            real[i+2] = 1
        else:
            real[i+2] = 0
        signCurveOld = signCurveNew
    return real
```

Figure 35: produces a vector indicating inflection points, scaled to match length of original data vector



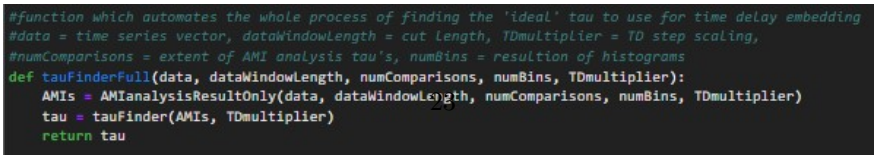
```
#function which automates the process of finding the 'ideal' tau from a set of average mutual information results
def tauFinder(AMIs, TDmultiplier): #AMIs set of AMI's from the 'AMIanalysisResultsOnly, TDmultiplier = scaling of TD steps
    size = np.shape(AMIs)[0]
    tau = 0
    crits = critPointEndCapped(AMIs)
    curve = curvature(AMIs)
    mins = np.empty(size)
    count = 0
    for i in range(size):
        if (crits[i] > 0.5):
            if (curve[i] > 0.5):
                mins[i] = 1
                count += 1
            elif (curve[i] < -0.5):
                mins[i] = 0
        else:
            mins[i] = 0
    AMImins = np.empty(size)
    for i in range(size):
        AMImins[i] = mins[i]*AMIs[i]
    AMIminSet = np.empty(count)
    ind = 0
    for i in range(size):
        if (mins[i] > 0.5):
            AMIminSet[ind] = AMImins[i]
            ind += 1
    minsSTD = np.std(AMIminSet)
    minsMED = np.median(AMIminSet)
    minsAVG = np.mean(AMIminSet)
    minsMIN = np.amin(AMIminSet)
    bestIndex = 0
    threshold = minsMED + minsSTD #threshold one: median value of the AMI minima set + one standard deviation
    threshold2 = minsAVG #threshold two: average value of the AMI minima set
    threshold3 = minsMIN*1.15 #threshold three: 115% the value of the lowest AMI amongst the AMI minima set
```

Figure 36: first part of the function to automate the determination of τ



```
    for i in range(size):
        bestIndex = i
        val = AMImins[i]
        if (val != 0):
            if (val > threshold or val > threshold2 or val > threshold3): #note use of 3 different thresholds
                continue
            else:
                break
        else:
            continue
    properIndex = bestIndex + 1
    tau = properIndex*TDmultiplier #rescale the index corresponding to the ideal tau by using the TDmultiplier
    return tau
```

Figure 37: second part of the function to automate the determination of τ



```
#function which automates the whole process of finding the 'ideal' tau to use for time delay embedding
#data = time series vector, dataWindowLength = cut length, TDmultiplier = TD step scaling,
#numComparisons = extent of AMI analysis tau's, numBins = resultion of histograms
def tauFinderFull(data, dataWindowLength, numComparisons, numBins, TDmultiplier):
    AMIs = AMIanalysisResultOnly(data, dataWindowLength, numComparisons, numBins, TDmultiplier)
    tau = tauFinder(AMIs, TDmultiplier)
    return tau
```

Figure 38: function which returns the optimal τ

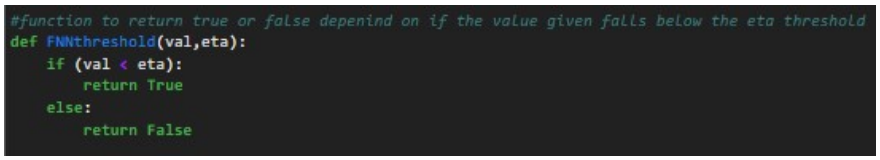


Figure 39: function which simply checks if FNN percentage has fallen below a small, non-zero threshold

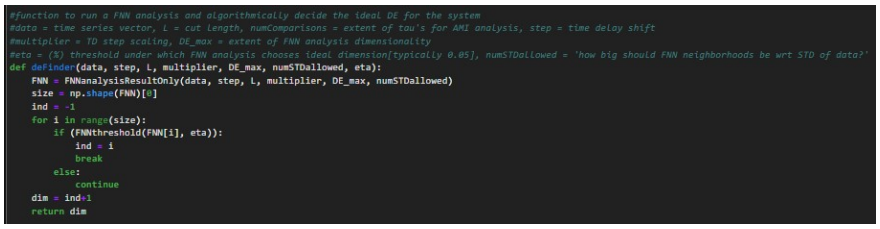


Figure 40: function which returns the optimal D_E

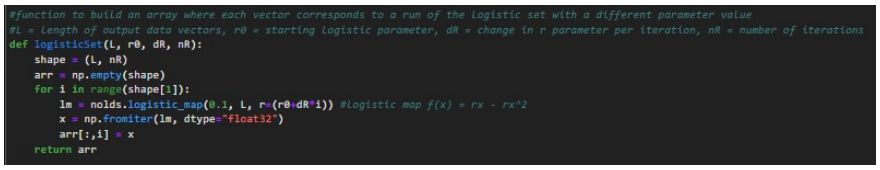


Figure 41: function to produce a set of data vectors from the logistic set with different values of r

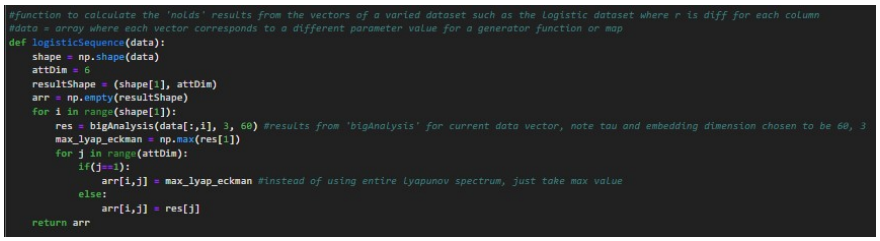


Figure 42: calculates the 'nolds' metrics on each of the vectors from the varied logistic array

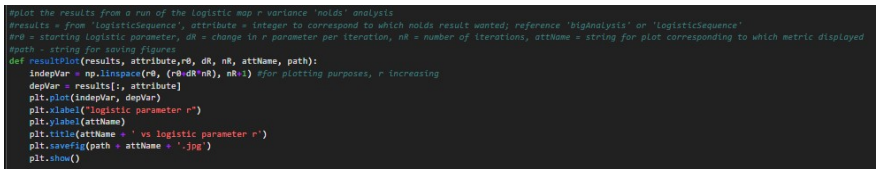


Figure 43: Plot the results from the logistic set analysis of the 'nolds' functions

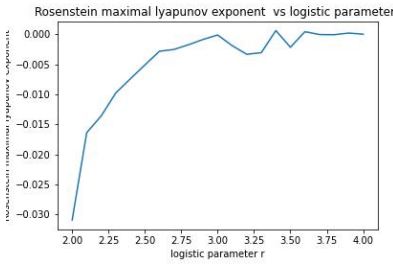


Figure 44: Rosenstein maximal lyapunov exponent as a function of logistic parameter r

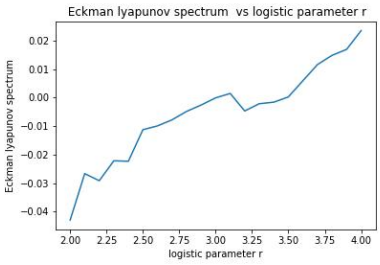


Figure 45: Maximum of the Eckman lyapunov exponent spectrum as a function of logistic parameter r

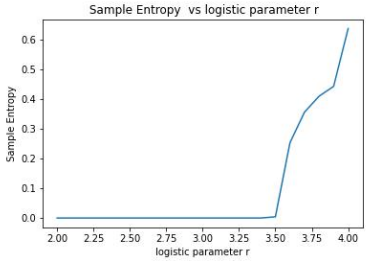


Figure 46: Sample entropy as a function of logistic parameter r

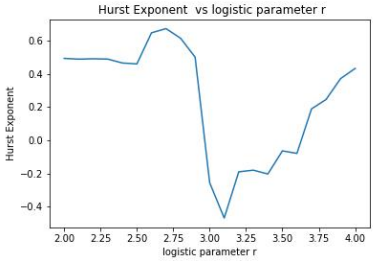


Figure 47: Hurst exponent as a function of logistic parameter r

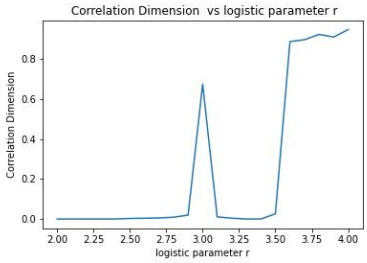


Figure 48: Correlation dimension as a function of logistic parameter r

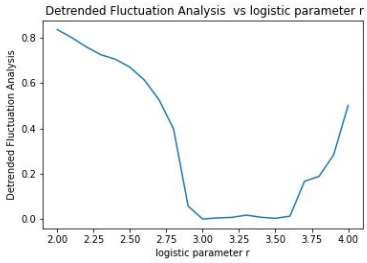


Figure 49: DFA parameter α as a function of logistic parameter r

```
#function to conduct various analysese pulled from the python library 'Nolds'
#returns Rosenstein maximal lyapunov exponent, Eckman lyapunov spectrum with embedding dimension Emb_dim and time delay Tau
#sample entropy, hurst exponent, correlation dimension, detrended fluctuation analysis
#NOTE workable, but in future want to 'optimize' since parameter choices of functions can affect outcomes
#for 'ideal' parameter choices, reference 'nolds' library documentation, will create subroutines and guidelines in the future but for now ok
def bigAnalysis(data, Emb_dim, Tau): #data = time series data, Emb_dim = DE for time delay, suggest to use FNN analysis to determine, Tau = time delay, use AMI analysis
    lyap_r = nolds.lyap_r(data, emb_dim = Emb_dim, tau = Tau) #Rosenstein maximal lyapunov exponent
    lyap_e = nolds.lyap_e(data, emb_dim = Emb_dim, matrix_dim = Emb_dim, tau = Tau) #Eckman lyapunov spectrum
    sampleEntropy = nolds.sampen(data, emb_dim=Emb_dim) #sample entropy
    hurstExponent = nolds.hurst_rs(data) #hurst exponent
    corrDim = nolds.corr_dim(data, emb_dim = Emb_dim) #correlation dimension
    detrFluctAnalysis = nolds.dfa(data) #detrended fluctuation analysis
    resultVec = [lyap_r, lyap_e, sampleEntropy, hurstExponent, corrDim, detrFluctAnalysis]
    return resultVec
```

Figure 50: function which runs the set of different 'nolds' analyses

```
#function to print, in python output, the result object from the 'bigAnalysis' function
def resultsPrint(results):
    shape = np.shape(results)
    attributes = [" Rosenstein maximal lyapunov exponent "," Eckman lyapunov spectrum "," Sample Entropy "," Hurst Exponent "," Correlation Dimension "," Detrended Fluctuatio
    for i in range(shape[0]):
        sentence = attributes[i] + ": " + str(results[i])
        print(sentence)
```

Figure 51: function to print 'nolds' analysis results from 'bigAnalysis' function

```
Rosenstein maximal lyapunov exponent : -2.7977195078119413e-06
Eckman lyapunov spectrum : [0.00951651 0.00511997 0.00275776]
Sample Entropy : 0.6444771385822813
Hurst Exponent : 0.4835154766553006
Correlation Dimension : 0.9595525204797065
Detrended Fluctuation Analysis : 0.4888508690032204
```

Figure 52: 'bigAnalysis' results from logistic $r = 4$

```
#function to automate the 'nolds' bigAnalysis pipeline by running AMI and FNN analyses and determining tau and DE automatically
#data = time series vector, dataWindowLength = cut Length, numComparisons = for AMI analysis
#numBins = resolution of histograms, TDmultiplier = TD step scaling, DE_max = extent of FNN analysis dimensionality
#eta = (%) threshold under which FNN analysis chooses ideal dimension[typically 0.05], numSTDallowed = 'how big should FNN neighborhoods be wrt STD of data?'
def YugeAnalysis(data, dataWindowLength, numComparisons, numBins, TDmultiplier, DE_max, eta, numSTDallowed):
    tau = tauFinderFull(data, dataWindowLength, numComparisons, numBins, TDmultiplier)
    step = int(tau/TDmultiplier)
    DE = deFinder(data, step, dataWindowLength, TDmultiplier, DE_max, numSTDallowed, eta)
    print("AMI derived tau: " + str(tau))
    print("FNN derived embedding dimension: " + str(DE))
    print("results from nolds functions:")
    results = bigAnalysis(data, DE, tau)
    resultsPrint(results)
```

Figure 53: function to automate the process of determining τ and D_E before running the 'bigAnalysis' function on the same data with the determined parameters

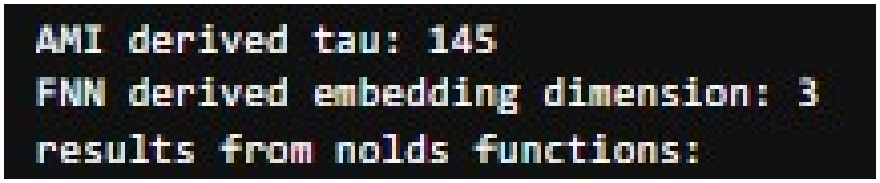


Figure 54: 'YugeAnalysis' results from logistic $r = 4$, part 1

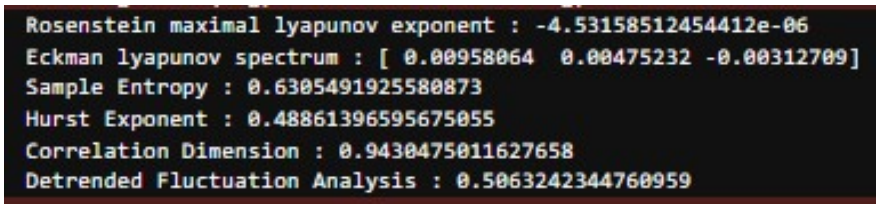


Figure 55: 'YugeAnalysis' results from logistic $r = 4$, part 2

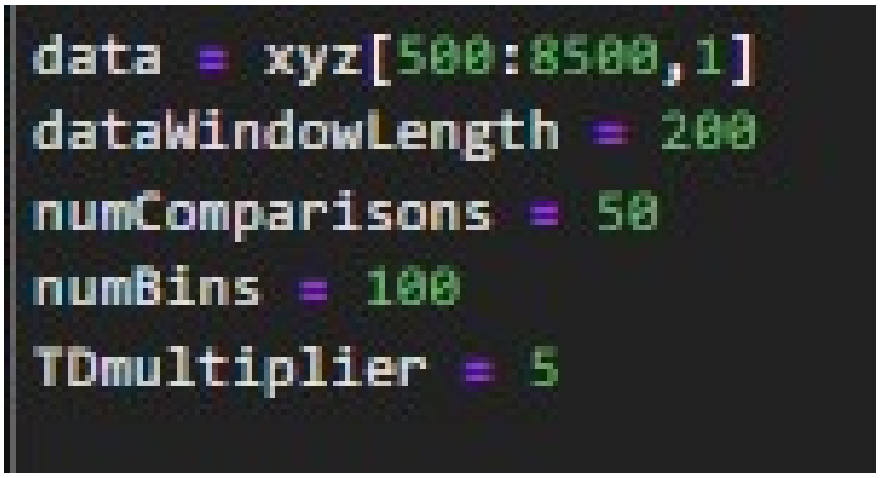


Figure 56: relevant parameters for the running of program, part 1

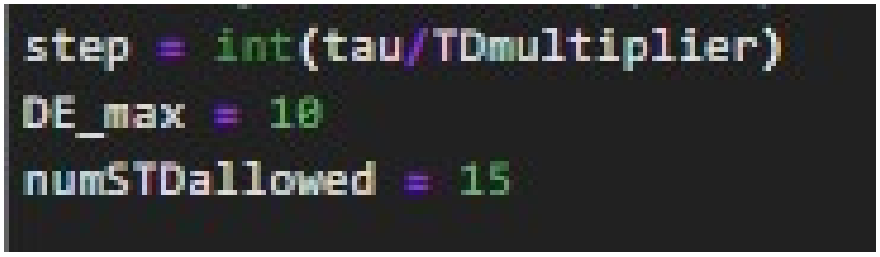


Figure 57: relevant parameters for the running of program, part 1

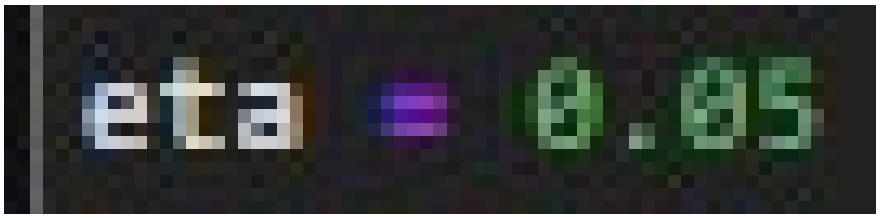


Figure 58: relevant parameters for the running of program, part 3

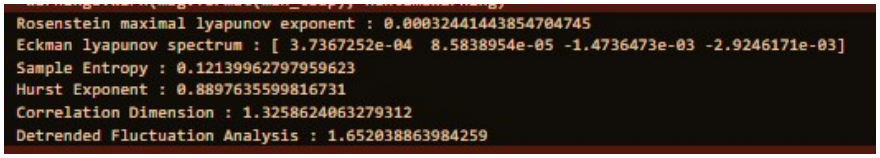


Figure 59: 'bigAnalysis' results from Lorenz '63 X

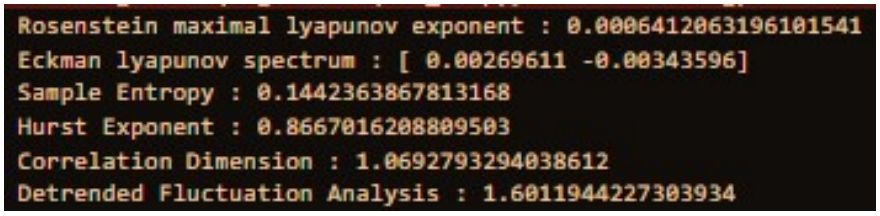


Figure 60: 'bigAnalysis' results from Lorenz '63 Y

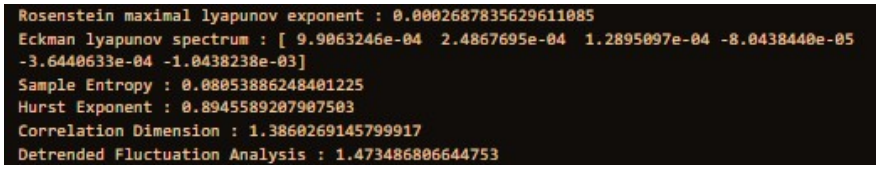


Figure 61: 'bigAnalysis' results from Lorenz '63 Z

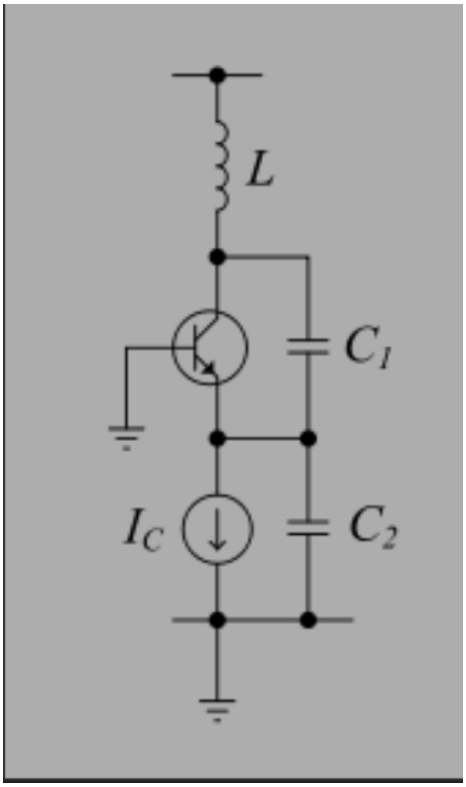


Figure 62: An example of a Colpitts oscillator circuit diagram. The oscillations are driven by a feedback loop in which energy passed back and forth between the inductor and the capacitors, driven by the transistor.

```
12  # Define the Colpitts system
13  def colpitts(t, xyz, alpha=1, gamma=0.0797, q=0.6898, eta=6.2723):
14      x, y, z = xyz
15      dxdt = alpha*y
16      dydt = -gamma*(x + z) - q*y
17      dzdt = eta*(y + 1 - np.exp(-x))
18      return np.array([dxdt, dydt, dzdt])
19
20  # Define the 4th order Runge-Kutta method
21  def rk4_step(f, t, x, h):
22      k1 = h * f(t, x)
23      k2 = h * f(t + h/2, x + k1/2)
24      k3 = h * f(t + h/2, x + k2/2)
25      k4 = h * f(t + h, x + k3)
26      return x + 1/6 * (k1 + 2*k2 + 2*k3 + k4)
27
28  # Set up initial conditions and integration parameters
29  t0 = 0
30  tf = 200
31  h = 0.01
32  xyz0 = np.array([1, 1, 1])
33
34  # Integrate the Colpitts system using the 4th order Runge-Kutta m
35  n_steps = int((tf - t0) / h)
36  t = np.linspace(t0, tf, n_steps+1)
37  xyz = np.zeros((n_steps+1, 3))
38  xyz[0] = xyz0
39  for i in range(n_steps):
40      xyz[i+1] = rk4_step(colpitts, t[i], xyz[i], h)
```

Figure 63: The simulated states of the Colpitts oscillator were obtained by integrating the differential equations of the dynamical system using the 4th-order Runge-Kutta numerical integratkion method.

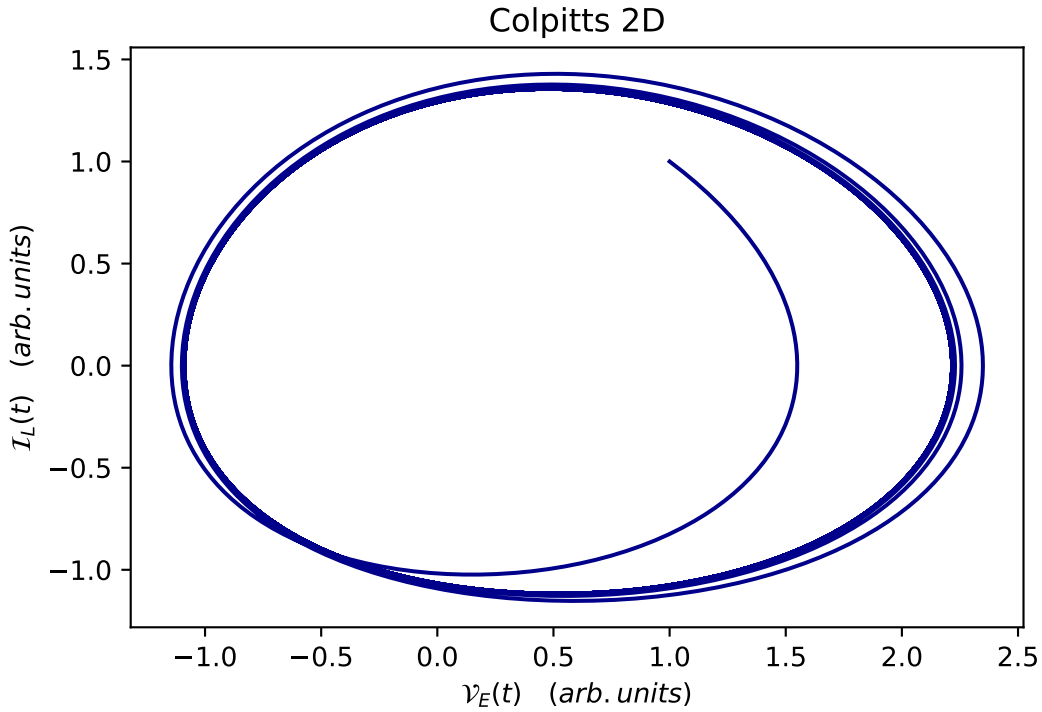


Figure 64: A 2 dimensional projection of the attractor of the simulated Colpitts oscillator with $\alpha_D = 1$. The periodic nature of the trajectory suggests a limit cycle, indicating stable oscillations and no chaos.

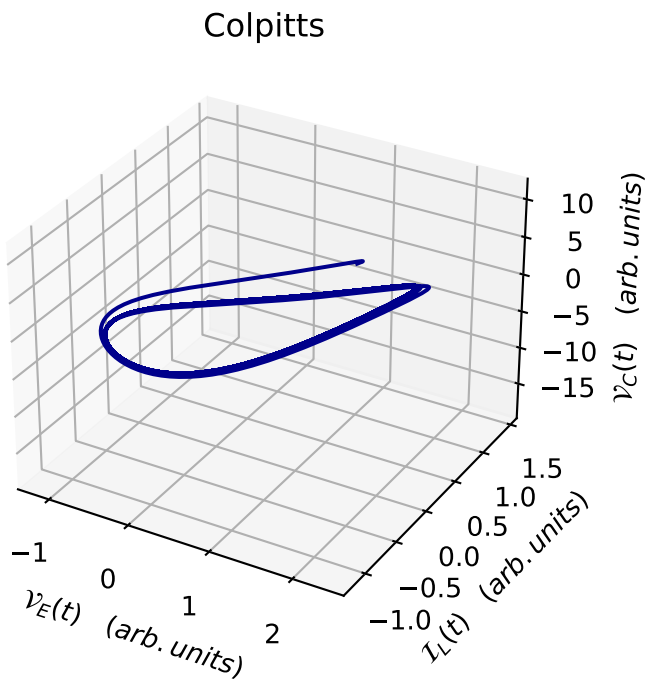


Figure 65: The 3 dimensional plot of the state space of the Colpitts oscillator with $\alpha_D = 1$. In addition to the periodic behavior, we can see that the trajectory remains bounded with no evidence of divergence.

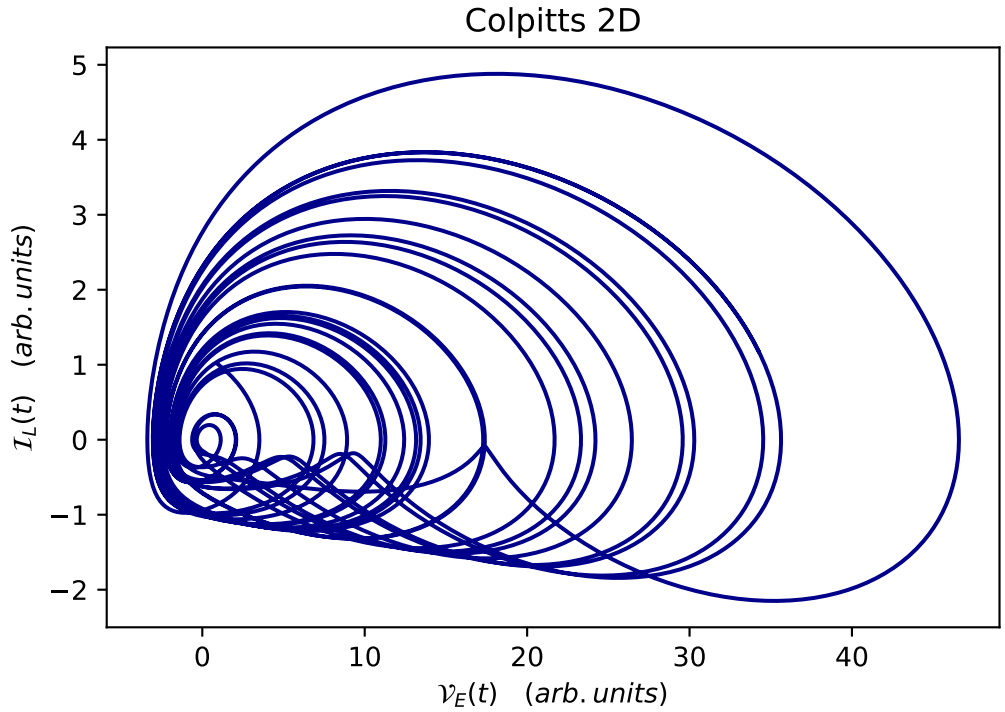
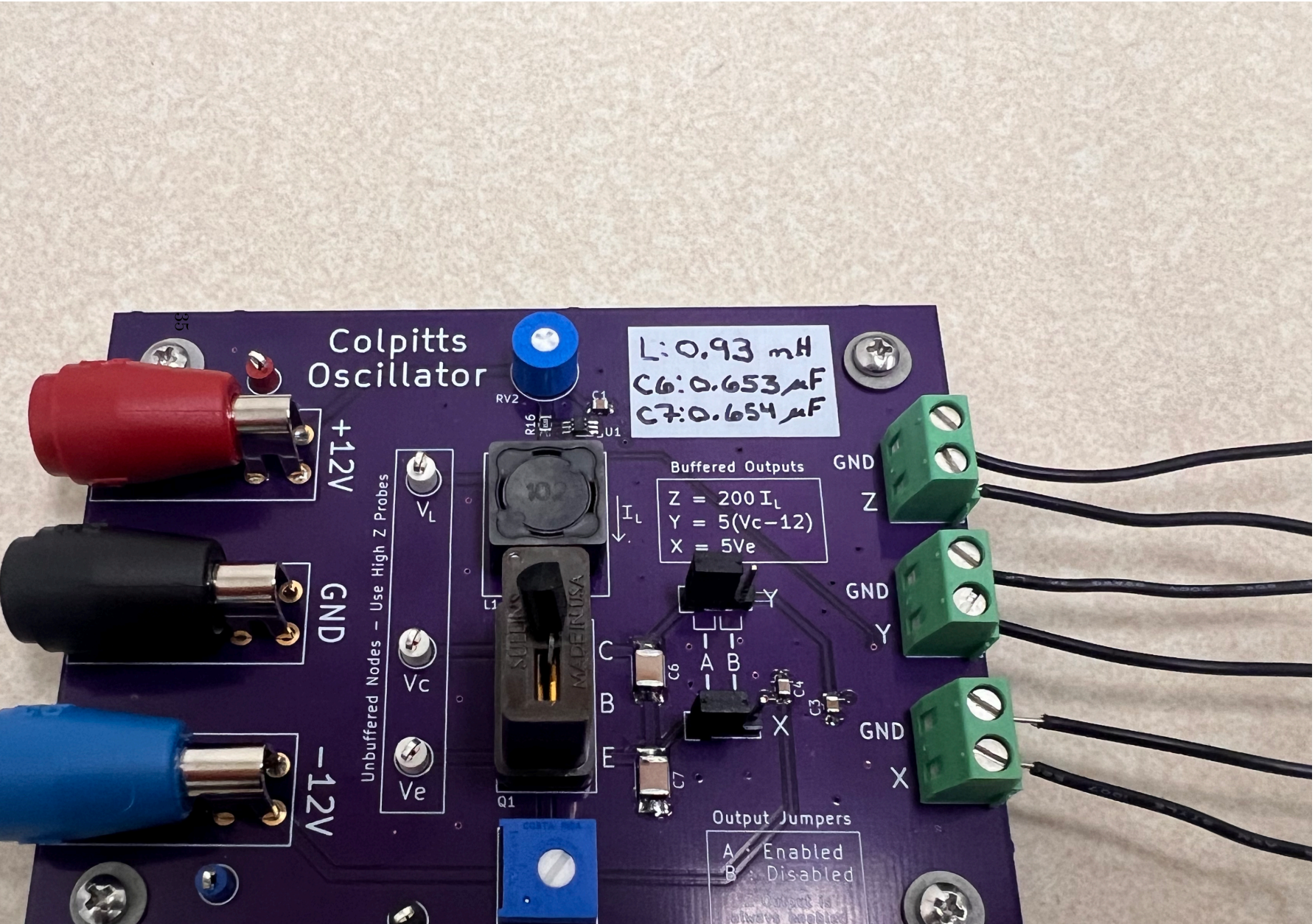


Figure 66: A 2 dimensional projection of the attractor of the simulated Colpitts oscillator with $\alpha_D = 5$. The trajectory appears to be a strange attractor, indicating a chaotic system.


Colpitts
Oscillator
L: 0.93 mH
C6: 0.653 μF
C7: 0.654 μF
+12V
GND
−12V
Unbuffered Nodes – Use High Z Probes
V_L
Vc
Ve
RV2
R16
C1
U1
L1
I_L
Q1
C
B
E
C6
C7
C4
C3
A
B
Buffered Outputs
Z = 200 I_L
Y = 5(Vc−12)
X = 5Ve
Y
X
GND
Z
GND
Y
GND
X
Output Jumpers
A : Enabled
B : Disabled

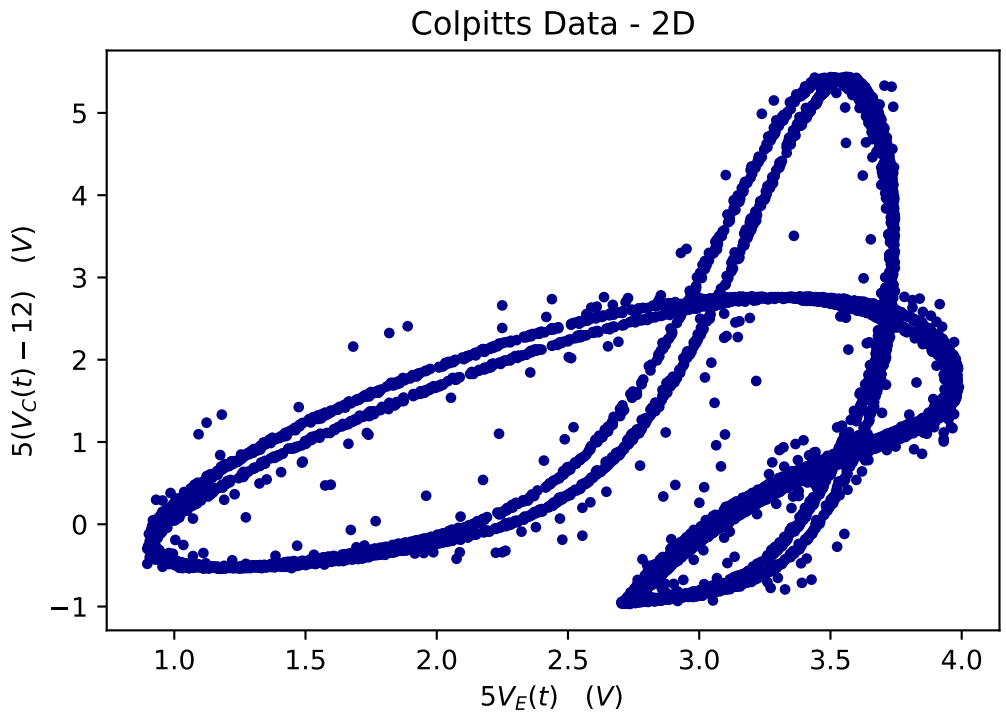


Figure 68: Data taken from an example of a Colpitts oscillator circuit. This 2 dimensional projection of the state space appears to show periodicity, indicating a limit cycle. Thus the oscillator seems to be in a non-chaotic regime where it emits stable oscillations.

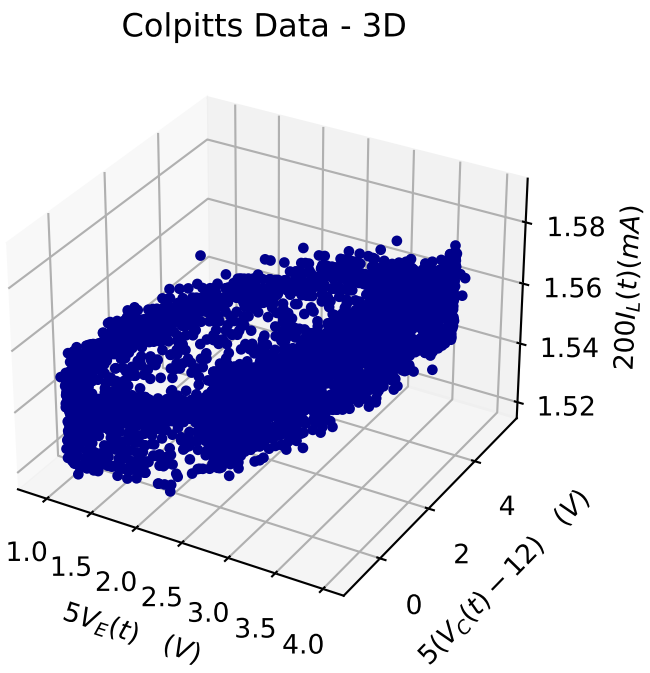


Figure 69: Data taken from an example of a Colpitts oscillator circuit. This plot of the state space is evidence of boundedness of the trajectory, further indicating a limit cycle due to the absence of any discernable divergence of the trajectory.

```
Rosenstein maximal lyapunov exponent : 3.4302766092895287e-06
Eckman lyapunov spectrum : [ 1.5028806e-03  6.1364897e-04  2.6467207e-04  8.3631327e-05
-3.2785963e-05 -3.2462631e-04 -1.1551298e-03 -1.3897303e-03
-2.2741142e-03 -2.7217541e-03 -3.4194286e-03 -3.6676528e-03
-4.2936672e-03 -4.8779869e-03 -5.2477825e-03 -6.1874506e-03
-7.3424210e-03 -9.7742081e-03 -1.4561843e-02 -3.4713712e-02]
Sample Entropy : 0.014711789994239284
Hurst Exponent : 0.9490835714915717
Correlation Dimension : 1.3629295030008275
Detrended Fluctuation Analysis : 1.9867479492407156
```

Figure 70: 'bigAnalysis' results from Simulated Colpitts X

```
Rosenstein maximal lyapunov exponent : -2.0532307270276936e-06
Eckman lyapunov spectrum : [ 7.4935037e-05  3.2064454e-05  8.3249852e-06 -3.6199843e-05
-9.7133539e-05 -2.7433815e-04 -5.1632884e-04 -8.3724095e-04
-1.8802964e-03]
Sample Entropy : 0.018947241598794407
Hurst Exponent : 0.946682555748229
Correlation Dimension : 1.0630269757083122
Detrended Fluctuation Analysis : 1.9760336138539878
```

Figure 71: 'bigAnalysis' results from Simulated Colpitts Y

```
Rosenstein maximal lyapunov exponent : 1.9881579909806124e-05
Eckman lyapunov spectrum : [ 1.3375196e-04  5.5807010e-05  4.7243457e-06 -4.2549389e-05
-1.4963294e-04 -3.4237781e-04 -6.7548745e-04 -1.1193720e-03
-1.7698192e-03 -4.0056817e-03]
Sample Entropy : 0.01568180510705215
Hurst Exponent : 0.9480637195978585
Correlation Dimension : 1.0789898394343327
Detrended Fluctuation Analysis : 1.9533407121606459
```

Figure 72: 'bigAnalysis' results from Simulated Colpitts Z

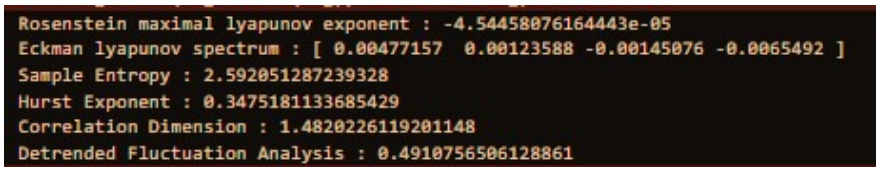


Figure 73: 'bigAnalysis' results from hardware Colpitts X

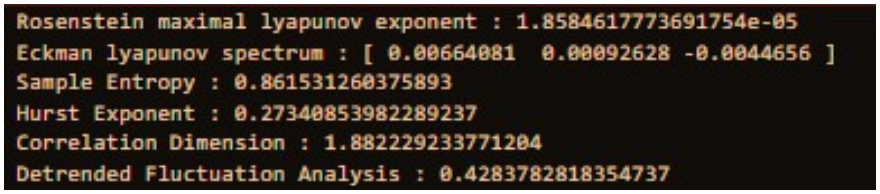


Figure 74: 'bigAnalysis' results from hardware Colpitts Y

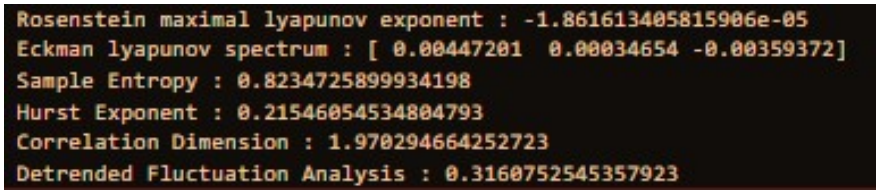


Figure 75: 'bigAnalysis' results from hardware Colpitts Z