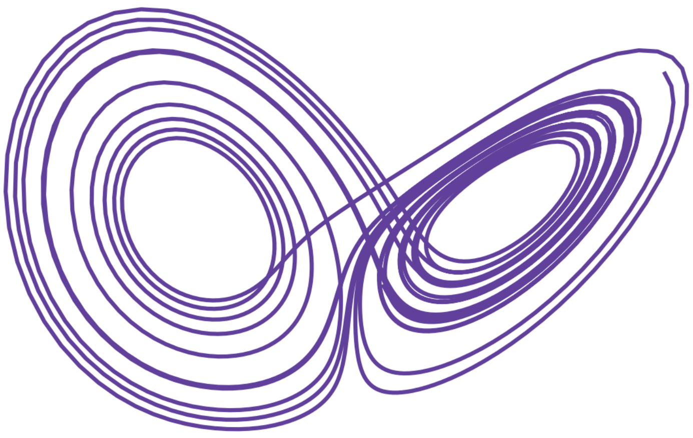
O25 Practical Report:
Exploring the Lorenz System

Anonymous Number: 2537



Due: 18/04/2025
Wordcount: 1992

Overall Aims

- Explore the dynamics of the Lorenz system by numerical simulation.
- Calculate the Metric Entropy of the Lorenz system using the forecasting error of the k-nearest neighbour method.
- Compute the Lyapunov spectrum of the system.
- Find the best embedding parameters for the Lorenz system using Cao's method and the self-mutual information.
- Perform the same calculations with added noise.

1 Introduction

In modelling the climate, chaotic systems are commonplace. The most famous chaotic system (Ghys 2013) is the Lorenz system, (Lorenz 1963) which is a simplified convective model. It holds particularly relevance to atmospheric convection and cliamte science (Slingo and Palmer 2011), but also has been applied to ocean dynamics (Shevchenko and Berloff 2023), circuit oscillations (Cuomo and Oppenheim 1993) and biological reactions (Zou, Zhang, and Wei 2021) among many disciplines where using a chaotic system can be useful.

In this report, we will explore the Lorenz system, create a forecasting model, compute information theoretic measures of the system, re-embed lower dimensional series, and discuss the effect of noise. This serves as a schematic for understanding the power of these analyses on chaotic physical time series.

2 Solving the Lorenz System

The Lorenz System is expressed as three ordinary differential equations (ODEs). x is proportional to the intensity of convection, y the temperature difference across the cell, and z a measure of the mismatch between the temperature profile and linearity (with positive describing the strong gradients near the boundaries).

$$\begin{aligned} \frac{dx}{dt} &= \sigma(y-x) \\ \frac{dy}{dt} &= x(\rho - z) - y \\ \frac{dz}{dt} &= xy - \beta z \end{aligned} \tag{1}$$

Using the below parameters yields the Lorenz attractor, which displays two "wings" in phase space, interpreted as being analagous to two directions of convection. The choice of values for these parameters follows Saltzman (1962), where σ was chosen for numerical convenience, about twice the value of the Prandtl Number for water, 4.8; ρ was chosen to be slightly above the critical value for instability of steady convection of 24.74, and β was chosen to minimise the critical Rayleigh number. This allows us to ignore the steady solutions at low Rayleigh numbers and only explore the chaotic behaviour that may be useful in modelling complex dynamical systems (Lorenz 1963).

- Prandtl Number, $\sigma = 10$, represents the ratio of momentum diffusivity to thermal diffusivity.
- Rayleigh Fraction, $\rho = 28$, the ratio of the critical Rayleigh number to the Rayleigh number of the system.
- Layer Parameter, $\beta = \frac{8}{3}$, is a geometric parameter; $\beta = 4/(1 + a^2)$ where a is the relative thickness of the layer.

The Lorenz system is solved using the Runge-Kutta method in fifth order with fourth order error control (RK5(4)) (Dormand and Prince 1980), with the behaviour plotted in Figure 1.

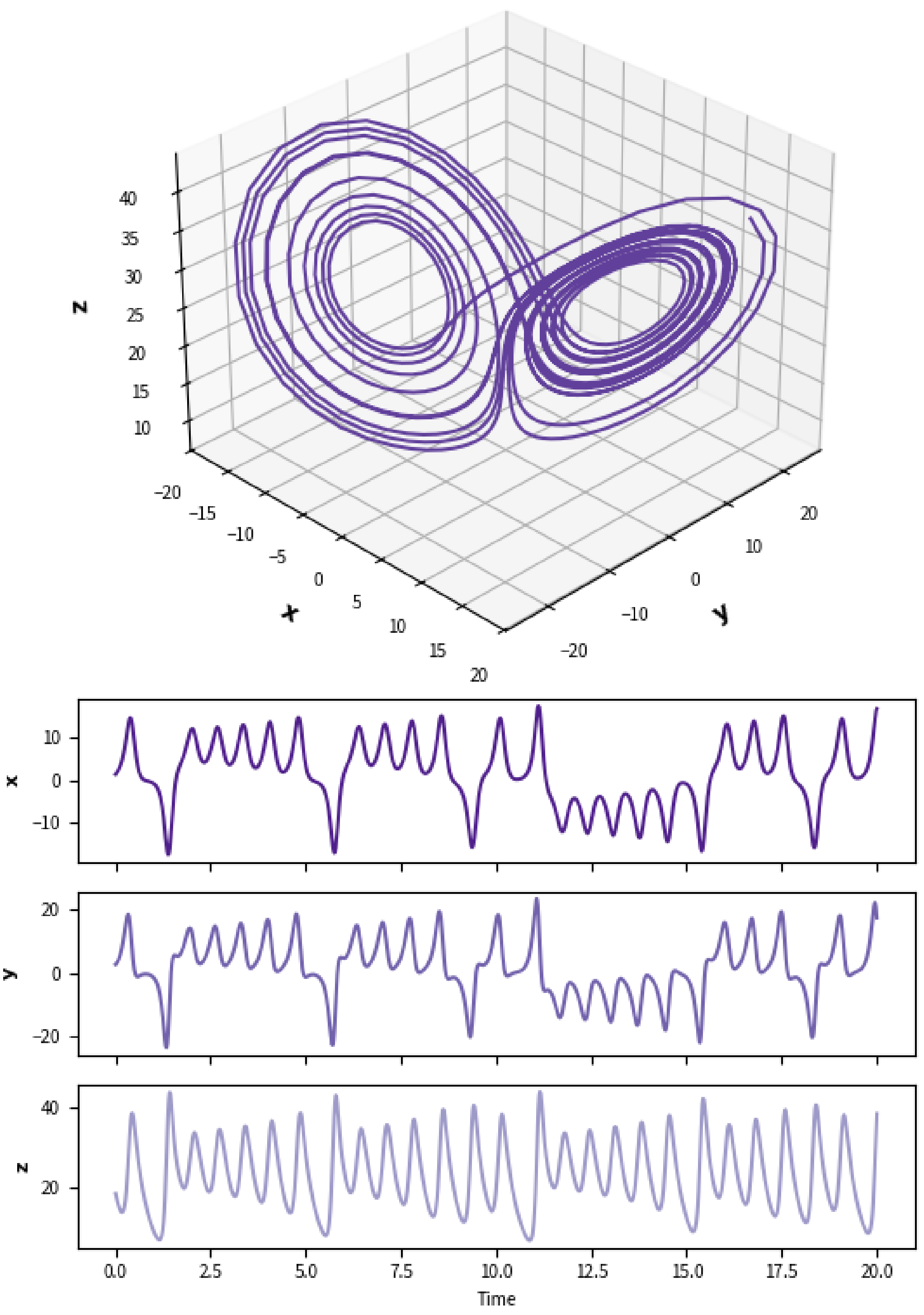


Figure 1: Lorenz Attractor over $t_{\text{final}} = 20.0$ time units simulated with n=1000 steps. Initial conditions were $\mathbf{x_0} = (1.4324, 2.6837, 18.5043)^T$.

3 Nearest Neighbour Forecasting

The k-nearest neighbour (kNN) technique was used for forecasting our Lorenz model, to generate an estimate for forecasting error and ultimately metric entropy. kNN is an unsupervised learning techhnique which simply uses the evolution of past similar points in phase space as an analogue for future changes (Martínez et al. 2019). A schematic of the method is shown in 2.

This method assumes the system is deterministic (i.e every point in phase space is unique and evolves uniquely), and the system is ergodic (i.e. the behaviour of a typical point is similar to the average behaviour).

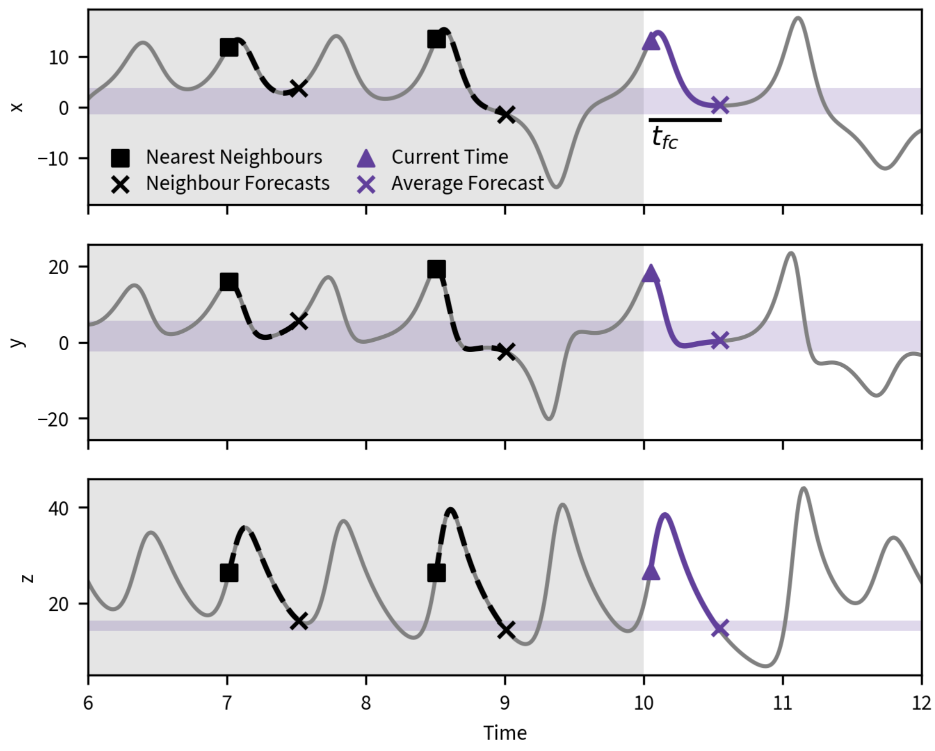


Figure 2: Schematic of nearest neighbour method. Grey shading indicates the training dataset. Purple shading indicates the range of the neighbour forecasts which are averaged. t_{fc}, the forecasting time, is indicated (0.5 units). Grey line is plotted from the simulated training + test dataset

To apply this to our system, we apply the following protocol. First, the total time series is split into training and test sets; in this report we have generally used a 50% : 50% split of the dataset. The k nearest neighbours in the training set are found by partitioning the points by their distance (Frobenius norm) from a given sample point. By sampling the point in the training set t_{fc} after each of these neighbours, we have a set of forecasts, which we can average to obtain a reasonable forecast of the future of the sample point chosen.

Applying this method to every point in the test set, we obtain a time series whose correlation with the actual test set can be computed, as shown in Figure 3.

The choice of k for our forecasting is important for our results; low values give a noisy and imprecise forecast, while high values are computationally expensive, and threaten to include too much of the training dataset. In this case, a weighted average modification should be considered.

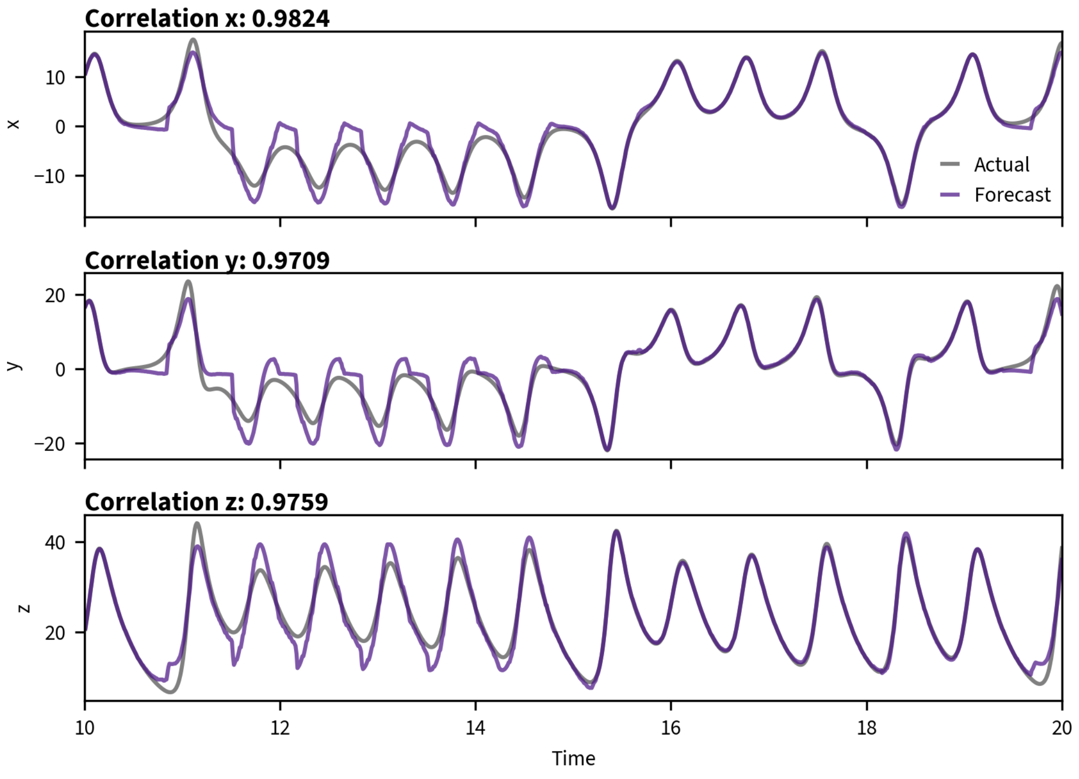


Figure 3: Forecasted and actual values from the test set ($t > 10$). $t_{fc} = dt = 0.01$ units or 1 timestep. Pearson's r for each dimension is shown.

4 Metric Entropy

The metric entropy (Kolmogorov entropy, K-entropy, Kolmogorov-Sinai entropy) (Sinai 1959), is defined as the rate of creation of information ($h = \langle \dot{I} \rangle$). We can calculate it by using the forecasting error of the kNN method, using the expression derived by Wales (1991), who proposes its validity regardless of forecasting method. For consistency with Wales and to maintain the same units as those commonly cited for the Lyapunov exponents, the metric entropy is given as $\frac{1}{dt}$ (per time step). Detailed discussion of this choice can be found in Appendix A.

By rearranging the expression for the Pearson's r and the variance of the forecasting error, as well as incorporating the effect of the forecasting time on the standard deviation, we obtain:

$$\ln(1 - r(t_{fc})) = \ln\left(\frac{s^2(0)}{2\sigma_x^2}\right) + 2ht_{fc} \tag{2}$$

By plotting $1 - r$ against t_{fc} we can fit the slope to the linear section of this graph (using the Random Sample Consense algorithm) (Fischler and Bolles 1981), as shown in Figure 4. Wales (1991) suggests taking the first two or four points of this relation for the slope is all that is needed to obtain reasonable results, but goes on to suggest the full initial slope up to the point of gradient decrease would provide a better estimate. Barraclough and Santis (1997) uses six points and obtains better results. The Random Sample Consensus (RANSAC) algorithm allows us to discard non-linear section and automatically identify the linear section predicted to have the slope of $h/2$ by equation 2.

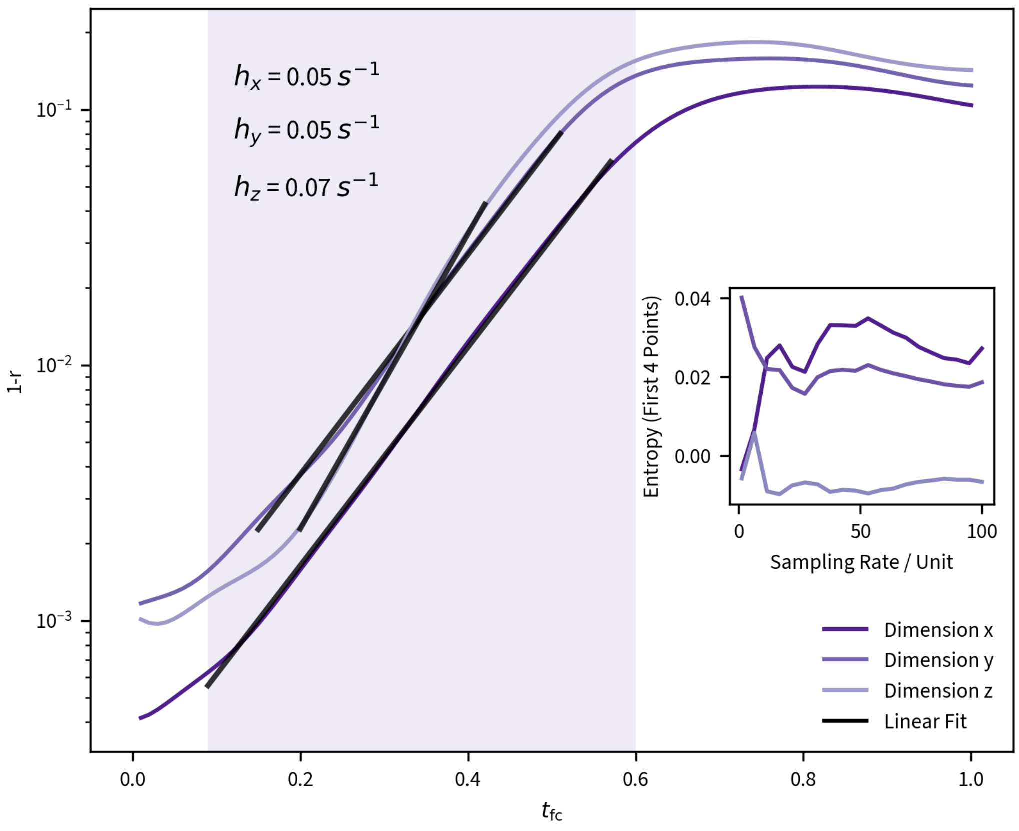


Figure 4: Forecasting Error vs Forecasting Time, used to calculate the metric entropy h for each dimension. Purple shading indicates maximum extent of linear region. Inset shows ensemble of sampling rates against entropies calculated from first 4 forecast values alone.

Values for h should range from 0 to 1. Our values are shown on Figure 4, and can be combined by summation to give $h_{\text{tot}} = 0.17$, which is well within the range of values from the literature (Wales 1991; Ruelle 1978). The inset figure takes a worse estimate of the entropy by only using the first four points only, but proves that the shape of this curve has stabilised and further increases in step size are unlikely to change the entropy values. The negative value for the z dimension's entropy on the inset represents the slight negative gradient for the first few points which is likely caused by correlation with small timescale features; not representing the full entropy.

4.1 Impact of Noise

From the definition of metric entropy, the presence of noise should increase our values of h, (Anishchenko and Astakhov 2008) as on shorter forecast times the system has less predictability; the information loss with time is higher. To appropriate scale noise across our three dimensions, the standard deviation of the noise is set according to the noise fraction a in equation 3.

$$\begin{gathered}\textbf{Where } \mathbf{x}(t) = [x, y, z] \text{ is the state of the system at time } t \in \mathbf{t}, \\ \text{and } a \in [0,1] \text{ is a defined constant, the noise fraction :} \\ \sigma_{\mathbf{N}} = [\sigma_{N,x}, \sigma_{N,y}, \sigma_{N,z}]. \\ \sigma_{\mathbf{N}} = a\left(\max_{i\in\mathbf{t}} \mathbf{x}(i) - \min_{i\in\mathbf{t}} \mathbf{x}(i)\right)\end{gathered} \tag{3}$$

The impact of varying noise fraction on the metric entropy is shown in Figure 5.

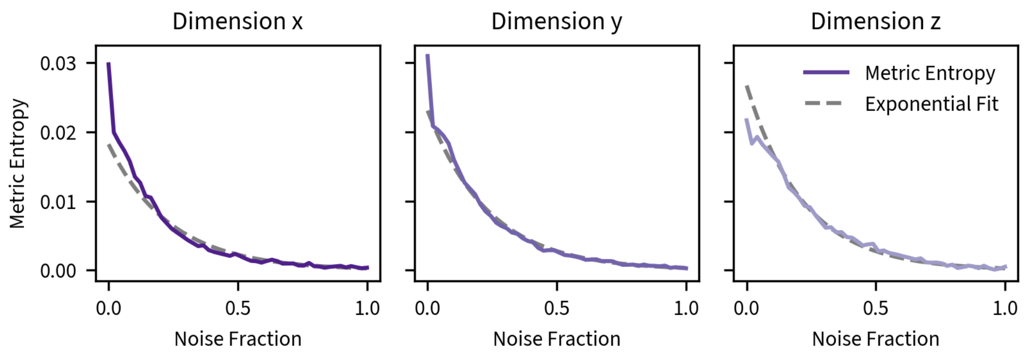


Figure 5: Metric Entropy vs Noise Fraction. 500 values for a were selected in a uniform range from 0 to 1.

The decay in metric entropy is roughly exponentional. Ordinarily, we would expect the values of h to increase with more noise, as the theoretical entropy of a purely stochastic system is infinite. Using the method of Wales (1991), however, yields values close to 0 for noisy systems (Barraclough and Santis 1997) due to the breakdown of equation 2, which seems to agree with our above exponential decrease.

5 Sensitivity to Initial Conditions

Chaotic systems are known to be sensitive to initial conditions (Lorenz 1963; Ghys 2013). Small early perturbations can cause divergence from the original trajectory only after large timescales (the "butterfly effect"). Figure 6 shows one example of a perturbation which does not strongly diverge until 10 time units, around 1000 time steps later.

5.1 Lyapunov Exponents

The Lyapunov spectrum is a measure of how sensitive the system is to perturbations. (Greene and Kim 1987)

For each dimension, we calculate the Lyapunov exponent by first defining a unit perturbation vector, and then evolving the perturbed state by one time step.
After a small number of timesteps ($n = 1$ in our case) the evolved perturbations are expressed in a a new orthonormal basis by QR decomposition (Schmidt 1907; Francis 1961). The resulting matrix R contains our renormalised vectors where taking the log of the lengths divided by the current time gives Lyapunov exponents (Equation 4). A new pertubation vector can be constructed by the transform Q on the previous orthonormal basis, and this process repeated iteratively to improve our estimate of the spectrum.

$$\lambda_i = \frac{1}{t}\ln\left(|r_{i,i}|\right) \quad \text{where } r_{i,i} = \text{the } i^{th} \text{ diagonal value of } \mathbf{R} \tag{4}$$

Instead of using the RK5 method to evolve the system, we can assume the perturbations are sufficiently small and compute the Jacobian which can be applied as a matrix multiplication rather than numerically integrating the perturbations. The Jacobian of the Lorenz system is given by equation 5.

$$J = \begin{pmatrix} -\sigma & \sigma & 0 \\ \rho - z & -1 & -x \\ y & x & -\beta \end{pmatrix} \tag{5}$$

The total effect of this process is shown over time in Figure 6.

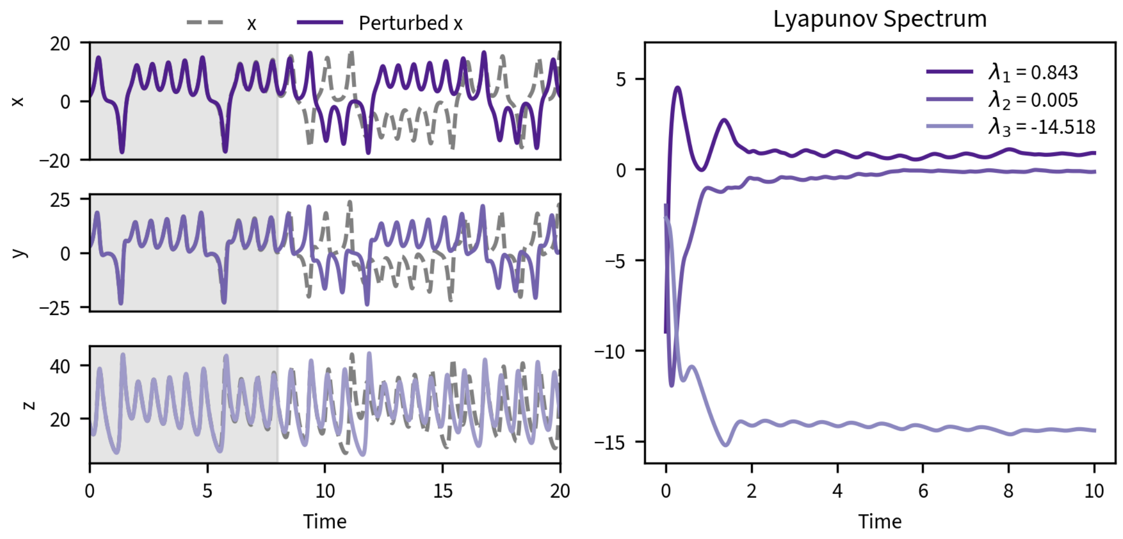


Figure 6: (Right) Example sensitivity to initial conditions. Grey shading indicates time before the perturbation added at time t_0 becomes significant. Perturbations were generated randomly from a normal distribution $\mathbf{x_0}_{\text{perturbed}} = \mathbf{x_0} + N(0, 1 \times 10^{-3})$; (Left) Lyapunov Spectrum

By summing the positive Lyapunov exponents, we obtain an upper bound (Ruelle 1978) on the metric entropy of 0.82. This agrees with our previous value of $h_{tot} = 0.17$. We can further calculate the Lyapunov dimension taking the sum of the first n elements such that $\sum_{i=1}^{n} \lambda_i > 0$ and dividing by λ_{n+1}, by which we obtain an estimate of 2.05, agreeing with the known value of the Lyapunov dimension. These values are robust with regard to sampling rate, so the inequality from Ruelle holds across most time. When sampling rate is very low, the metric entropy is low, as shown in the inset of 4, and similarly when high noise is applied, the metric entropy drops, but these will still maintain a lower bound on the Lyapunov exponent.

6 Efficient Embedding

In real world scenarios, we do not obtain the full phase space of a system, but often only time series of a single variable. Due to the correlation between the variables, by Taken's Theorem we can create a delay embedding (Takens 1981) by constructing the matrix shown in equation 6. In our case, I took only the x parameter.

$$\zeta = \begin{pmatrix} \mathbf{x}(t_0) & \mathbf{x}(t_1) & \cdots & \mathbf{x}(t_{n-m}) \\ \mathbf{x}(t_0+\tau) & \mathbf{x}(t_1+\tau) & \cdots & \mathbf{x}(t_{n-m}+\tau) \\ \vdots & \vdots & \ddots & \vdots \\ \mathbf{x}(t_0+(m-1)\tau) & \mathbf{x}(t_1+(m-1)\tau) & \cdots & \mathbf{x}(t_{n-m}+(m-1)\tau) \end{pmatrix} \tag{6}$$

Taking the first minimum of the self mutual information (SMI) of the time series gives the optimal time delay τ for the embedding, and the optimal dimension m can be determined by computing the $E1$ and $E2$ ratios. When the $E1$ ratio reaches 1 and plateaus, this indicated the upper bound on the optimal embedding dimension m. $E2$ represents the randomness of the system. The system is stochastic if $E2 \approx 1$ for all m. (Cao 1997)

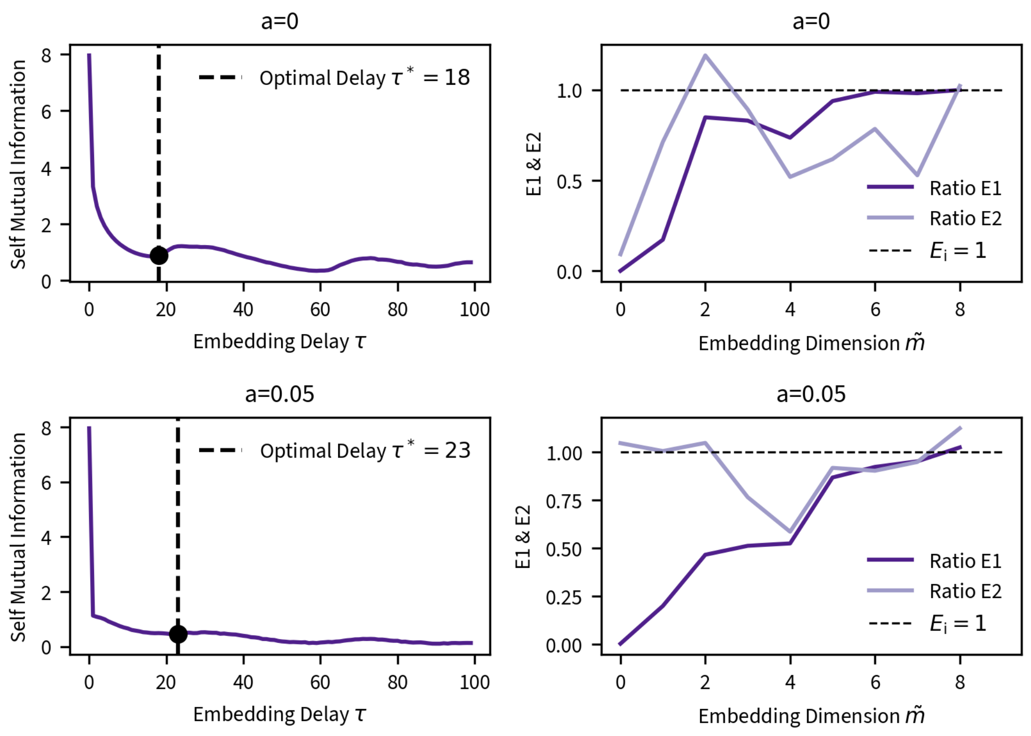


Figure 7: Computing the optimal embedding parameters. Noise fractions $a = 0, 0.05$ are shown.

7 Metric Entropy of Embedded Data

The same calculation is performed on an embedding with the optimal parameters determined. This allows us to capture the dynamics and information content of the system with a univariate time series of observations. The metric entropy here $h_{tot} = 0.12$ is lower but still bounded by λ_1.

8 Noise in Embedded Data

The same addition of noise fraction as performed in section 4.1 is performed on the embedded data. The information was lost with addition of any noise, therefore we applied a noise fraction $a = 0.05$. In figure 7 we can see the E1/E2 plot looks similar to a stochastic system with E1 increasing with no plateau and E2 being roughly equal to 1 for all m. This indicates that there is no strong embedding after addition of noise; the data is very similar to a stochastic system. The optimal time delay was increased, but from the shape of the graph this seems more to do with the lack of minima than an actual increase in SMI. Most of the SMI is lost in the first couple steps once noise is added.

From figure 8, the metric entropy of the noisy embedding is much lower, and the calculation leaves some strange artifacts, with a periodicity of τ in the correlation. This represents the noise being duplicated by the embedding, and so the metric entropy appears less defined. Nonetheless, we can obtain a h_{tot} of 0.05, which is much lower. This seems, as before, to be a breakdown in the Wales (1991) method for noisy data.

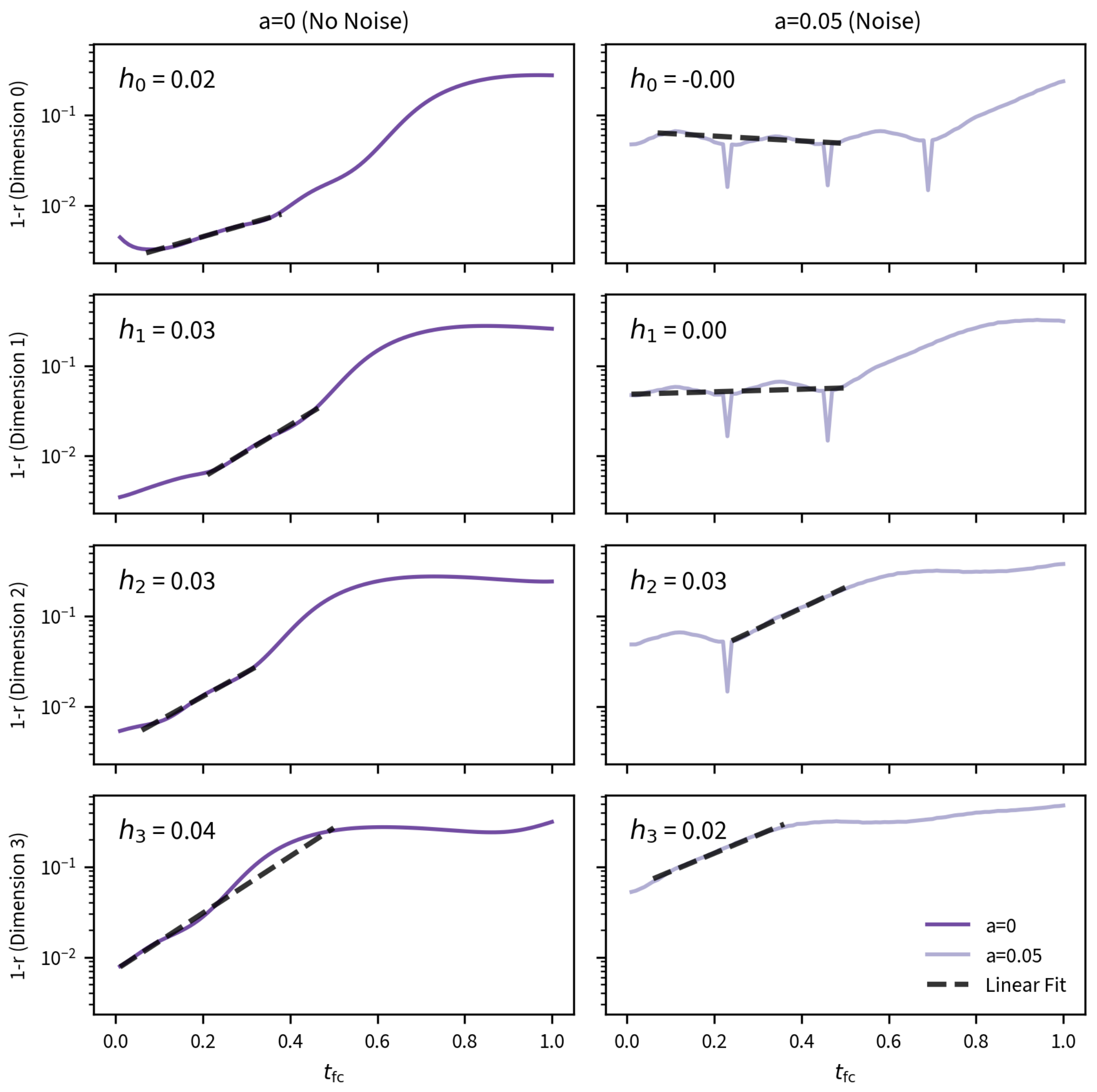


Figure 8: Metric entropy of the embedding with optimal parameters of $\tau = 18$ (or 23 for noise) and $\tilde{m} = 4$, with noise fraction $a = 0$ and $a = 0.05$

9 Conclusions & Improvements

The Lorenz system is a well studied chaotic system, and we obtained results for the Lyapunov spectrum and metric entropy consistent with the literature and their own bounds. We proposed using the RANSAC method to get more accurate values of the metric entropy, and ran an ensemble of timesteps to show that the entropy is invariant after a certain threshold. Noise causes a quick breakdown in the metric entropy expression used, and finding embedding parameters also becomes significantly harder with the addition of noise.

Further work could include collating the Lyapunov spectrum under the effect of noise, running larger ensembles for noise, embeddings, and number of nearest neighbours as well as varying the timescale of the system. This work could also be applied to other chaotic systems.

References

Anishchenko, Vadim and Sergey Astakhov (Sept. 2008). "RELATIVE KOLMOGOROV ENTROPY OF A CHAOTIC SYSTEM IN THE PRESENCE OF NOISE". In: *International Journal of Bifurcation and Chaos* 18, pp. 2851–2855. DOI: 10.1142/S021812740802210X.

Barraclough, David R. and Angelo De Santis (1997). "Some possible evidence for a chaotic geomagnetic field from observational data". In: *Physics of the Earth and Planetary Interiors* 99.3, pp. 207–220. ISSN: 0031-9201. DOI: https://doi.org/10.1016/S0031-9201(96)03215-3. URL: https://www.sciencedirect.com/science/article/pii/S0031920196032153.

Cao, Liangyue (1997). "Practical method for determining the minimum embedding dimension of a scalar time series". In: *Physica D: Nonlinear Phenomena* 110.1, pp. 43–50. ISSN: 0167-2789. DOI: https://doi.org/10.1016/S0167-2789(97)00118-8. URL: https://www.sciencedirect.com/science/article/pii/S0167278997001188.

Cuomo, Kevin M. and Alan V. Oppenheim (July 1993). "Circuit implementation of synchronized chaos with applications to communications". In: *Phys. Rev. Lett.* 71 (1), pp. 65–68. DOI: 10.1103/PhysRevLett.71.65. URL: https://link.aps.org/doi/10.1103/PhysRevLett.71.65.

Dormand, J.R. and P.J. Prince (1980). "A family of embedded Runge-Kutta formulae". In: *Journal of Computational and Applied Mathematics* 6.1, pp. 19–26. ISSN: 0377-0427. DOI: https://doi.org/10.1016/0771-050X(80)90013-3. URL: https://www.sciencedirect.com/science/article/pii/0771050X80900133.

Fischler, Martin A. and Robert C. Bolles (June 1981). "Random sample consensus: a paradigm for model fitting with applications to image analysis and automated cartography". In: *Commun. ACM* 24.6, pp. 381–395. ISSN: 0001-0782. DOI: 10.1145/358669.358692. URL: https://doi.org/10.1145/358669.358692.

Francis, J. G. F. (Jan. 1961). "The QR Transformation A Unitary Analogue to the LR Transformation—Part 1". In: *The Computer Journal* 4.3, pp. 265–271. ISSN: 0010-4620. DOI: 10.1093/comjnl/4.3.265. eprint: https://academic.oup.com/comjnl/article-pdf/4/3/265/1080833/040265.pdf. URL: https://doi.org/10.1093/comjnl/4.3.265.

Ghys, Étienne (2013). "The Lorenz attractor, a paradigm for chaos". In: *Chaos: Poincaré Seminar 2010*. Springer, pp. 1–54.

Greene, John M. and Jin-Soo Kim (1987). "The calculation of lyapunov spectra". In: *Physica D: Nonlinear Phenomena* 24.1, pp. 213–225. ISSN: 0167-2789. DOI: https://doi.org/10.1016/0167-2789(87)90076-5. URL: https://www.sciencedirect.com/science/article/pii/0167278987900765.

Lorenz, Edward N. (1963). "Deterministic Nonperiodic Flow". In: *Journal of Atmospheric Sciences* 20.2, pp. 130–141. DOI: 10.1175/1520-0469(1963)020<0130:DNF>2.0.CO;2. URL: https://journals.ametsoc.org/view/journals/atsc/20/2/1520-0469_1963_020_0130_dnf_2_0_co_2.xml.

Martínez, Francisco et al. (Oct. 2019). "A methodology for applying k-nearest neighbor to time series forecasting". In: *Artificial Intelligence Review* 52.3, pp. 2019–2037. ISSN: 1573-7462. DOI: 10.1007/s10462-017-9593-z. URL: https://doi.org/10.1007/s10462-017-9593-z.

Ruelle, David (Mar. 1978). "An inequality for the entropy of differentiable maps". In: *Boletim da Sociedade Brasileira de Matemática - Bulletin/Brazilian Mathematical Society* 9.1, pp. 83–87. ISSN: 1678-7714. DOI: 10.1007/BF02584795. URL: https://doi.org/10.1007/BF02584795.

Saltzman, Barry (1962). "Finite Amplitude Free Convection as an Initial Value Problem—I". In: *Journal of Atmospheric Sciences* 19.4, pp. 329–341. DOI: 10.1175/1520-0469(1962)019<0329:FAFCAA>2.0.CO;2. URL: https://journals.ametsoc.org/view/journals/atsc/19/4/1520-0469_1962_019_0329_fafcaa_2_0_co_2.xml.

Schmidt, Erhard (Dec. 1907). "Zur Theorie der linearen und nichtlinearen Integralgleichungen". In: *Mathematische Annalen* 63.4, pp. 433–476. ISSN: 1432-1807. DOI: 10.1007/BF01449770. URL: https://doi.org/10.1007/BF01449770.

Shevchenko, Igor and Pavel Berloff (2023). "On a probabilistic evolutionary approach to ocean modelling: From Lorenz-63 to idealized ocean models". In: *Ocean Modelling* 186, p. 102278. ISSN: 1463-5003. DOI: https://doi.org/10.1016/j.ocemod.2023.102278. URL: https://www.sciencedirect.com/science/article/pii/S146350032300118X.

Sinai, Yakov G. (1959). “On the notion of entropy of a dynamical system”. en. In: *Doklady of Russian Academy of Sciences* 124.3.

Slingo, Julia and Tim Palmer (2011). “Uncertainty in weather and climate prediction”. In: *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences* 369.1956, pp. 4751–4767. DOI: 10.1098/rsta.2011.0161. eprint: https://royalsocietypublishing.org/doi/pdf/10.1098/rsta.2011.0161. URL: https://royalsocietypublishing.org/doi/abs/10.1098/rsta.2011.0161.

Takens, Floris (1981). “Detecting strange attractors in turbulence”. In: *Dynamical Systems and Turbulence, Warwick 1980*. Ed. by David Rand and Lai-Sang Young. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 366–381. ISBN: 978-3-540-38945-3.

Wales, David J. (Apr. 1991). “Calculating the rate of loss of information from chaotic time series by forecasting”. In: *Nature* 350.6318, pp. 485–488. ISSN: 1476-4687. DOI: 10.1038/350485a0. URL: https://doi.org/10.1038/350485a0.

Zou, Chengye, Qiang Zhang, and Xiaopeng Wei (2021). “Compilation of a Coupled Hyper-Chaotic Lorenz System Based on DNA Strand Displacement Reaction Network”. In: *IEEE Transactions on NanoBioscience* 20.1, pp. 92–104. DOI: 10.1109/TNB.2020.3031360.

Appendices

A The Units of Forecasting Time for Metric Entropy Values

The literature is inconsistent with how to use forecasting time for the calculation of metric entropy by the Wales (1991) method. Some authors use the forecasting time in number of iterations to calculate the slope, multiplying by dt to obtain their values, whereas others use t_{fc} in time units, producing larger K.

Wales’s figures indicate he used the forecasting times measured in number of iterations, and only considered the first few points, and his equations suggest that the prior probability distribution is normally distributed according to the actual time. For his case he sets them equal, but there is a difference in magnitude of values when the timestep is less than the time unit, as in our case.

Barraclough and Santis (1997) used this method on a dataset of annual means, and reported the K-entropy in units of year^{-1}, and then verified their values against known maps to a reasonable agreement, which is essentially equivalent to using timesteps, as their timestep and forecasting time were equal.

As the Lyapunov exponent is a real property of the system rather than just the time series, we can assume that a lower bound on the exponent should not always grow. When the K-entropy is measured in time units, this is precisely what happens, up to sampling rates of higher than 0.001 as demonstrated in figure 9:

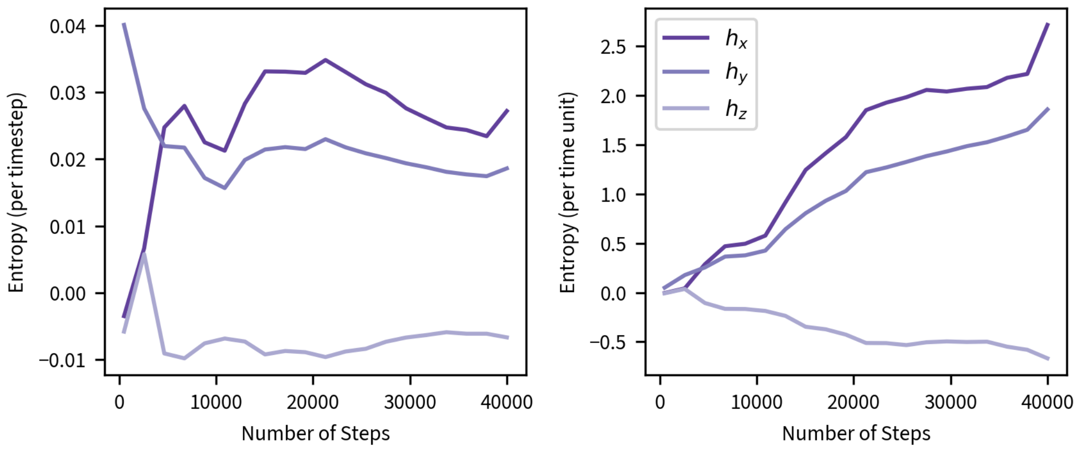


Figure 9: Diverging entropy when measured in time units. k for nearest neigbours was set to 10% of the training set, though the same trends were present with constant $k = 100$. End time was kept constant so only the sampling rate is varied.

B Notation

Notation	Description
$\mathbf{x}$	State of the system at time t
$\mathbf{t}$	Time series of the system
$\mathbf{x_0}$	Initial conditions of the system $= (1.4324, 2.6837, 18.5043)^T$
$\sigma_\mathbf{N}$	Noise vector
t_{fc}	Forecasting time
t_0	Simulation start time
t_{final}	Simulation end time
x	Convective strength
y	Temperature difference
z	Temperature profile
a	Noise fraction
δt	Time step
τ	Embedding shift
σ	Prandtl number
ρ	Rayleigh fraction
β	Layer parameter
λ	Lyapunov exponent
h_{tot}	Total metric entropy
h_i	Metric entropy per dimension
$\mathbf{Q}$	Matrix of orthonormal vectors
$\mathbf{R}$	Matrix of renormalised vectors

Table 1: Notation used in this report.

C Parameters Used for Figures

Figure	Time Step (dt)	Start Time	End Time	Forecasting Time (t_{fc})	Neighbours (k)
1	0.02	0	20	/	/
2	0.02	0	12	0.5	5
3	0.01	0	20	0.01	10
4	0.01 (0.01 → 0.4)	0	400	0.01→0.1	2000 (10% training)
5	0.01	0	20	/	/
6	0.01	0	100	/	/
7	0.01	0	10	/	/
8	0.01	0	20	/	/
9	0.01→0.4	0	400	/	(50 → 2000)

Table 2: Simulation parameters used for figure generation.

D Code

Imports and Setup

```
#Imports
import numpy as np
import scipy as sp
import matplotlib.pyplot as plt
from scipy.spatial import cKDTree
from scipy.spatial import distance_matrix
from sklearn.linear_model import RANSACRegressor
from sklearn.feature_selection import mutual_info_regression
import gtda.time_series
import EntropyHub
from tqdm import tqdm

#Style setup
plt.style.use('default')
plt.rc('font',**{'family':'Source Sans 3','weight':'normal','size':'8'})
plt.rc('axes', prop_cycle=plt.cycler('color', plt.cm.Purples(np.linspace(0.9,
 ↪  0.1, 6))))
```

D.1 Lorenz System Code

```python
SIGMA = 10
RHO = 28
BETA = 8/3

START_TIME = 0
END_TIME = 100
NUM_STEPS = 10000

dt = (END_TIME-START_TIME)/NUM_STEPS
print(f"dt {dt}")

labels = ['x','y','z']
# x = [x,y,z]
def lorenz63(t:float,x:np.ndarray) -> np.ndarray:
    dxdt = np.zeros(3)
    dxdt[0] = SIGMA*(x[1]-x[0])
    dxdt[1] = x[0]*(RHO-x[2])-x[1]
    dxdt[2] = x[0]*x[1]-BETA*x[2]
    return dxdt

# Integrate the system
def integrate_system(x0:np.ndarray, start_time:float, end_time:float,
 ↪  num_steps:int) -> sp.integrate._ivp.ivp.OdeResult:
    results = sp.integrate.solve_ivp(lorenz63,[START_TIME,END_TIME],x0,
                method='RK45', t_eval =
                 ↪  np.linspace(START_TIME,END_TIME,NUM_STEPS+1))
    return results

X0 = np.array([1.4324, 2.6837, 18.5043])
results = integrate_system(X0, START_TIME, END_TIME, NUM_STEPS)
```

```
# Making Figure 1 Plots
fig, axes = plt.subplots(4, 1, figsize=(5.5, 8), gridspec_kw={'height_ratios':
↪  [1.3, 0.3, 0.3, 0.3]},sharex=True)

# Make the top subplot square
axes[0].remove()
axes[0] = fig.add_subplot(4, 1, 1, projection='3d')


# Plot 1: 3D Lorenz attractor
axes[0].view_init(elev=30, azim=315)
axes[0].plot(results.y[0, :], results.y[1, :], results.y[2, :])
axes[0].set_xlabel('$\\mathbf{x}$', fontsize=10)
axes[0].set_ylabel('$\\mathbf{y}$', fontsize=10)
axes[0].set_zlabel('$\\mathbf{z}$', fontsize=10)
axes[0].zaxis.set_ticks_position('lower')
axes[0].zaxis.set_label_position('lower')

# Plot 2: x, y, z vs time
axes[1].plot(results.t, results.y[0, :], label='x',color=plt.cm.Purples(0.9))
axes[1].set_ylabel('$\\mathbf{x}$',rotation=90)
axes[2].plot(results.t, results.y[1, :], label='y',color=plt.cm.Purples(0.7))
axes[2].set_ylabel('$\\mathbf{y}$')
axes[3].plot(results.t, results.y[2, :], label='z',color=plt.cm.Purples(0.5))
axes[3].set_ylabel('$\\mathbf{z}$')
axes[3].set_xlabel('Time')


plt.tight_layout()
plt.savefig("p1.png")
plt.show()
```

D.2 Nearest Neighbour Code

```
#Data Partition
TRAIN_TEST_SPLIT = 0.5
def partition_data(results:np.ndarray, train_test_split:float, num_steps:int)
↪  -> tuple[np.ndarray, np.ndarray]:
    split_index = int(num_steps * train_test_split)
    print(split_index)
    training = results[:,:split_index] # shape (3, split_index)
    test = results[:,split_index:] # shape (3, NUM_STEPS - split_index)
    return training, test

training, test = partition_data(results.y, TRAIN_TEST_SPLIT, NUM_STEPS)
```

```
#Point alogorithm definition
def find_nearest_neighbours_point(training:np.ndarray, test:np.ndarray,
↪   k:int)->np.ndarray:
    tree = cKDTree(training.T)
    # Query the tree for the k nearest neighbours
    _, neighbour_indices = tree.query(test.T, k=k)
    return neighbour_indices.T

def forecast_from_neighbours_point(neighbour_indices:np.ndarray,
↪   training:np.ndarray, forecast_timesteps:int)->np.ndarray:
    future_indices = neighbour_indices+forecast_timesteps
    forecasts = training[:, future_indices]
    return forecasts.mean(axis=1)

def k_nearest_neighbour_forecast(training:np.ndarray, test:np.ndarray,
↪   k:int=10, forecast_timesteps:int=1)->np.ndarray:
    neighbour_indices =
    ↪   find_nearest_neighbours_point(training[:,:-forecast_timesteps], test,
    ↪   k)
    forecast = forecast_from_neighbours_point(neighbour_indices,training,
    ↪   forecast_timesteps)
    return forecast
```

```
# Figure 2 Plot: KNN Schematic
XMIN = 6
XMAX = 12
T_TEST = 10.05 # Point to forecast
K = 10 # Number of neighbours
NEIGHBOUR_DIFF_THRESH = 40 # Minimum distance between neighbours
T_FORECAST = 0.5 # Forecast time

# Derived Values
forecast_timesteps = int(T_FORECAST/dt)
test_index = int((T_TEST-START_TIME)/dt)
end = int(XMAX/dt)
split_index = int(END_TIME*dt*TRAIN_TEST_SPLIT)

# Find the Neighbours
neighbours = find_nearest_neighbours_point(training, results.y[:,test_index],
 ↪  k=K)
forecast = forecast_from_neighbours_point(neighbours, training,
 ↪  forecast_timesteps)

# filter neighbours to avoid points too close to each other (they look ugly)
filtered_neighbours = []
for n in neighbours:
    if all(abs(n - other_n) > NEIGHBOUR_DIFF_THRESH for other_n in
     ↪  filtered_neighbours):
        filtered_neighbours.append(n)
neighbours = np.array(filtered_neighbours)

# Step the neighbours forward in time
nn_forecasts = []
for n in neighbours:
    nn_forecasts.append(n + forecast_timesteps)
```

```python
# Setup plot
fig, axs = plt.subplots(3, 1, figsize=(5.6, 4.5), sharex=True)
for i, ax in enumerate(axs): # For each dimension
    ax.plot(results.t[:end], results.y[i, :end], color='grey') # Plot data
    ↪ behind the points

    ax.plot(results.t[neighbours], results.y[i, neighbours], 's',
    ↪ color='black', label='Nearest Neighbours') # Plot the nearest
    ↪ neighbours
    ax.plot(results.t[nn_forecasts], results.y[i, nn_forecasts], 'x',
    ↪ color='black', label='Neighbour Forecasts') # Plot the forecasted
    ↪ neighbours

    ax.plot(results.t[test_index], results.y[i, test_index], '^',
    ↪ color=plt.cm.Purples(0.8), label='Current Time') # Plot the test point
    ax.plot(results.t[test_index+forecast_timesteps], forecast[i], 'x',
    ↪ color=plt.cm.Purples(0.8), label='Average Forecast') # Plot the
    ↪ forecast

    # Plot in a different colour the forecasting time
    for nn, nn_shifted in zip(neighbours, nn_forecasts):
        ax.plot(results.t[nn:nn_shifted], training[i,nn:nn_shifted],
        ↪ color='black', linewidth=2,linestyle='--')
    ax.plot(results.t[test_index:test_index+forecast_timesteps], results.y[i,
    ↪ test_index:test_index+forecast_timesteps], color=plt.cm.Purples(0.8),
    ↪ linewidth=2,linestyle='-')

    # Add the shaded panels
    ax.axvspan(0,results.t[split_index], color='grey', alpha=0.15,lw=0)
    ax.axhspan(min(results.y[i,nn_forecasts]), max(results.y[i,nn_forecasts]),
    ↪ color=plt.cm.Purples(0.8), alpha=0.2,lw=0)
    ax.set_xlim(XMIN, XMAX)

# Labels
axs[0].set_ylabel('x')
axs[1].set_ylabel('y')
axs[2].set_ylabel('z')
axs[2].set_xlabel('Time')

# Tfc label
axs[0].plot([results.t[test_index], results.t[test_index+forecast_timesteps]],
↪ [forecast[0]-3, forecast[0]-3], color='black', linestyle='-', )
axs[0].text(results.t[test_index], forecast[0]-8, '$t_{fc}$', fontsize=10,
↪ color='black')

# Legend
axs[0].legend(ncol=2,loc='lower left', fontsize=8,frameon=False,
↪ labelspacing=0.3,borderpad=0,handletextpad=0.2,columnspacing=0.3)

# Saveout
plt.tight_layout
plt.savefig("p2.png", dpi=300, bbox_inches='tight')
plt.show()
```

```
# KNN Forecasting and Correlation
K = 10
FORECAST_TIMESTEPS = 1

# Forecast
forecast = k_nearest_neighbour_forecast(training, test, K, FORECAST_TIMESTEPS)

# Calculate the correlation coefficient
def calculate_correlation(forecast:np.ndarray, test:np.ndarray,
↪   forecast_timesteps:int)->np.ndarray:
    correlation = []
    for i in range(len(forecast)):
        correlation.append(np.corrcoef(forecast[i,:-forecast_timesteps],
        ↪   test[i,forecast_timesteps: ])[0,1])
    return np.array(correlation)

correlation = calculate_correlation(forecast, test, FORECAST_TIMESTEPS)
```

```
# Figure 3 Plot: Correlation with Forecast
split_index = int(NUM_STEPS * TRAIN_TEST_SPLIT)
t = np.linspace(END_TIME-split_index*dt,END_TIME, 1+ NUM_STEPS - split_index)

fig, axs = plt.subplots(3, 1, figsize=(5.8, 4.2), sharex=True)

for i,h in zip(range(len(forecast)),[20,25,45]):

    axs[i].plot(t[FORECAST_TIMESTEPS:], test[i,FORECAST_TIMESTEPS:],
    ↪   label='Actual', color='grey')
    axs[i].plot(t[FORECAST_TIMESTEPS:], forecast[i,:-FORECAST_TIMESTEPS],
    ↪   label='Forecast', color=plt.cm.Purples(0.95),alpha=0.7)
    axs[i].set_ylabel(labels[i])
    axs[i].set_xlim(END_TIME-split_index*dt, END_TIME)
    axs[i].text(0, 1.06, f'Correlation {labels[i]}: {correlation[i]:.4f}',
    ↪   transform=axs[i].transAxes, ha='left',
    ↪   va='center',fontsize=10,weight='bold')



axs[0].legend(loc='lower right', frameon=False, handlelength=0.7,borderpad=0)
axs[2].set_xlabel('Time')

# Saveout
plt.tight_layout()
plt.savefig("p3.png", dpi=300, bbox_inches='tight')
plt.show()
```

D.3 Metric Entropy Code

```python
# Metric Entropy Calculation
DEPTH_PLOT = 100 # Depth to obtain x,y values for
K = 200
forecast_timesteps_array = np.linspace(1, DEPTH_PLOT, DEPTH_PLOT).astype(int)

# Calculate the graph of ln(1-r) to time step as the slope is 2 times the
↪  metric entropy
def calculate_entropy(training: np.ndarray, test:np.ndarray,
↪  forecast_timesteps_array:list)->np.ndarray:
    y = []
    x = forecast_timesteps_array * dt
    for forecast_timesteps in tqdm(forecast_timesteps_array):
        forecast = k_nearest_neighbour_forecast(training, test, K,
        ↪  forecast_timesteps)
        correlation = calculate_correlation(forecast, test,
        ↪  forecast_timesteps)
        y.append(np.log(1-correlation))
    slope, _ = np.polyfit(x, y, 1)
    y = np.array(y)
    return slope/2, x, y

# This entropy calculation only holds if the graph is approx linear on the
↪  depth_plot range
h, x, y = calculate_entropy(training,test,forecast_timesteps_array)
print(f'Entropy = {sum(h)}, with components: {h[0]}, {h[1]}, {h[2]}')
```

```
# Identify the linear section
PLATEAU_TIME = 60
fig = plt.figure(figsize=(5.5, 4.5))

# Plotting t_fc vs log(1-r)
def calculate_ransac_slope(x: np.ndarray, y: list, plateau_time: int,
 ↪ threshold: int=0.10) -> tuple:
    slopes = []
    intercepts = []
    inlier_masks = []
    for dim in range(len(y[0])):
        ransac = RANSACRegressor(residual_threshold=threshold, random_state=3)
         ↪ # Set random state for consistency
        ransac.fit(x.reshape(-1, 1)[:plateau_time], y[:plateau_time,dim])  #
         ↪ Fit RANSAC for the current dimension
        slopes.append(ransac.estimator_.coef_[0])  # Extract the slope
        inlier_masks.append(ransac.inlier_mask_) # Extract the inlier mask
         ↪ (the points in the linear set)
        intercepts.append(ransac.estimator_.intercept_)  # Extract the
         ↪ intercept

    return np.array(slopes),np.array(intercepts), np.array(inlier_masks)

ransac_slopes, intercepts, inlier_masks = calculate_ransac_slope(x, y,
 ↪ PLATEAU_TIME)
print(f"RANSAC slopes: {ransac_slopes}")
```

```
# Run the ensemble for varying sampling rates
DEPTH_PLOT = 4
num_steps_array = np.linspace(500,40000,20).astype(int)
entropies = []

for num_steps in tqdm(num_steps_array):
    K = int(num_steps/20)
    dt = (END_TIME - START_TIME) / num_steps
    results = integrate_system(X0, START_TIME, END_TIME, num_steps)
    training, test = partition_data(results.y, TRAIN_TEST_SPLIT, num_steps)
    forecast_timesteps_array = np.linspace(1, DEPTH_PLOT,
     ↪ DEPTH_PLOT).astype(int)
    h, _, _ = calculate_entropy(training, test, forecast_timesteps_array)
    entropies.append(h)

#correct for step size
corrected_entropies = []
for steps, h in zip(num_steps_array, entropies):
    dt = (END_TIME - START_TIME) / steps
    corrected_entropies.append(h * dt)
    print(f"NUM_STEPS: {steps}, Entropy: {sum(h)}, Components: {h[0]}, {h[1]},
     ↪ {h[2]}")
```

```
# Figure 4 Plot: Metric Entropy
# values for the shading region
maxx = 0
minx=100
fig = plt.figure(figsize=(5.5, 4.5))

colors = [plt.cm.Purples(0.9), plt.cm.Purples(0.7), plt.cm.Purples(0.5)] # set
 ↪  a colour map

for dim in range(len(ransac_slopes)):
    plt.plot(x,np.exp(y[:,dim]),color = colors[dim], label=f'Dimension
     ↪  {labels[dim]}', alpha=1) # Plot the actual data

    # Plot the slopes
    plt.plot(x[:PLATEAU_TIME][inlier_masks[dim]][:-3],
             np.exp(ransac_slopes[dim]*x[:PLATEAU_TIME][inlier_masks[dim]] +
              ↪  intercepts[dim])[:-3],
             ls='-', color = 'black', alpha=0.8, lw=2)

    # Work out the min and max extent to shade
    maxx = max(x[:PLATEAU_TIME][inlier_masks[dim]][-1],maxx)
    minx = min(x[:PLATEAU_TIME][inlier_masks[dim]][0],minx)

plt.plot(0,0, color='black', label='Linear Fit') # Dummy plot for legend
plt.axvspan(minx, maxx, color=plt.cm.Purples(0.8), alpha=0.1,lw=0) # Shading

# Add the entropy labels
plt.text(minx+0.03, np.exp(-2), f'$h_x$ = {ransac_slopes[0]/2 * dt:.2f}
 ↪  $s^{{-1}}$', fontsize=10, color='black', ha='left', va='center')
plt.text(minx+0.03, np.exp(-2.5), f'$h_y$ = {ransac_slopes[1]/2 * dt:.2f}
 ↪  $s^{{-1}}$', fontsize=10, color='black', ha='left', va='center')
plt.text(minx+0.03, np.exp(-3), f'$h_z$ = {ransac_slopes[2]/2 * dt:.2f}
 ↪  $s^{{-1}}$', fontsize=10, color='black', ha='left', va='center')


# Labels
plt.legend(loc='lower right', frameon=False)
plt.xlabel('$t_\\text{fc}$')
plt.ylabel('1-r')
plt.yscale('log')

# Inset the ensemble plot
axins = fig.add_axes([0.71, 0.4, 0.25, 0.25])

axins.plot(num_steps_array/END_TIME,corrected_entropies, label='Ensemble
 ↪  Entropy')
axins.set_xlabel('Sampling Rate / Unit')
axins.set_ylabel('Entropy (First 4 Points)')

# Saveout
plt.tight_layout()
plt.savefig("p4.png", dpi=300, bbox_inches='tight')
plt.show()
```

D.4 Noise Code

```
# Add Noise
NOISE_FRACTION = 0.1

noise_levels_dim = np.array([(np.max(results.y[0])-np.min(results.y[0])) *
↪  NOISE_FRACTION,(np.max(results.y[1])-np.min(results.y[1])) *
↪  NOISE_FRACTION, (np.max(results.y[2])-np.min(results.y[2])) *
↪  NOISE_FRACTION])

print(noise_levels_dim)

def add_noise(results:np.ndarray, noise_levels:np.ndarray)->np.ndarray:
    noisy_results = np.zeros_like(results)
    for i, sd in enumerate(noise_levels):
        noisy_results[i,:] = results[i, :] + np.random.normal(0, sd,
        ↪  size=results.shape[1])
    return noisy_results

noisy_results = add_noise(results.y, noise_levels_dim)
noisy_train, noisy_test = partition_data(noisy_results, TRAIN_TEST_SPLIT,
↪  NUM_STEPS)
```

```
# Calculate Metric Entropy for Each Noise Level
PLATEAU_TIME = 20
NOISE_FRACTIONS = np.linspace(0, 1, 50) # Noise fractions to test
DEPTH_PLOT = 30 # Depth to obtain x,y values for
K = 100
forecast_timesteps_array = np.linspace(1, DEPTH_PLOT, DEPTH_PLOT).astype(int)

def calculate_entropy_for_noise(training: np.ndarray, test:np.ndarray,
↪  noise_fractions:np.ndarray, forecast_timesteps_array:list)->np.ndarray:
    h_values = []
    for noise_fraction in noise_fractions:
        noise_levels_dim = np.zeros(3)
        for i in range(len(results.y)):
            noise_levels_dim[i] = (np.max(results.y[i])-np.min(results.y[i]))
            ↪  * noise_fraction
        noisy_training = add_noise(training, noise_levels_dim)
        noisy_test = add_noise(test, noise_levels_dim)
        _,x, y = calculate_entropy(noisy_training, noisy_test,
        ↪  forecast_timesteps_array)
        slopes,_,_= calculate_ransac_slope(x, y, PLATEAU_TIME)
        h = slopes/2
        h_values.append(h)
    return np.array(h_values)


h = calculate_entropy_for_noise(training, test, NOISE_FRACTIONS,
↪  forecast_timesteps_array)
```

```
        #  Figure 5 Plot: Metric Entropy with Noise

fig, axs = plt.subplots(1, 3, figsize=(5.5, 2), sharey=True)


axs[0].set_ylabel('Metric Entropy')
slopes, intercepts = np.polyfit(NOISE_FRACTIONS,
↪   np.log(np.clip(h,0.00001,None)), 1)
axs[2].plot(0,0, color=plt.cm.Purples(0.8), label='Metric Entropy') # Dummy
↪   plot for legend
for i, (slope, intercept) in enumerate(zip(slopes, intercepts)):
    axs[i].plot(NOISE_FRACTIONS, dt* np.exp(slope*NOISE_FRACTIONS +
    ↪   intercept),color='black',alpha = 0.5,ls='--', label='Exponential Fit')

for i, ax in enumerate(axs):
    ax.plot(NOISE_FRACTIONS,dt* h[:, i], color=colors[i])
    ax.set_xlabel('Noise Fraction')
    ax.set_title(f'Dimension {labels[i]}')

axs[2].legend(frameon=False)


plt.tight_layout()
plt.savefig("p5.png", dpi=300, bbox_inches='tight')
plt.show()
```

D.5 Perturbation Code

```

# Figure 6 Plot: Lorenz Attractor with Noise
# Integrate the system
x0 = np.array([1.4324, 2.6837, 18.5043])
results =
↪   sp.integrate.solve_ivp(lorenz63,[START_TIME,END_TIME],x0,method='RK45',
t_eval=np.linspace(START_TIME,END_TIME,NUM_STEPS))

# Integrate with a small random perturbation
np.random.seed(0)
perturbation = np.random.normal(0, 1e-3, size=3)
x0_perturbed = x0 + perturbation
results_perturbed =
↪   sp.integrate.solve_ivp(lorenz63,[START_TIME,END_TIME],x0_perturbed,
method='RK45', t_eval=np.linspace(START_TIME,END_TIME,NUM_STEPS))
```

```python
def calculate_lyapunov_exponents(lorenz63, x0, num_steps, dt, num_vectors=3):
    # Initialize perturbation vectors
    perturbations = np.eye(num_vectors)
    lyapunov_exponents = np.zeros(num_vectors)
    lyapunov_exponents_over_time = np.zeros((num_vectors, num_steps))

    # Initialize the state
    x = x0.copy()

    for step in tqdm(range(num_steps)):
        # Evolve the system
        t_span = [step * dt, (step + 1) * dt]
        result = sp.integrate.solve_ivp(lorenz63, t_span, x, method='RK45',
         ↪  t_eval=[(step + 1) * dt])
        x_start = x
        x = result.y[:, -1]

        # Evolve the perturbations
        evolved_perturbations = []
        for perturbation in perturbations:
            perturbed_state = x_start + perturbation
            result = sp.integrate.solve_ivp(lorenz63, t_span, perturbed_state,
             ↪  method='RK45',
                                             t_eval=[(step + 1) * dt])
            evolved_perturbations.append(result.y[:, -1] - x)

        # Reorthonormalize the perturbations using QR decomposition
        evolved_perturbations = np.array(evolved_perturbations).T
        q, r = np.linalg.qr(evolved_perturbations)
        perturbations = q.T
        # Accumulate the logarithm of the diagonal elements of R
        lyapunov_exponents += np.log(np.abs(np.diag(r)))
        lyapunov_exponents_over_time[:, step] = lyapunov_exponents / ((step +
         ↪  1) * dt)

    # Normalize by the total time
    lyapunov_exponents /= (num_steps * dt)
    return lyapunov_exponents, lyapunov_exponents_over_time

num_vectors=3
lyapunov_exponents, lyapunov_exponents_over_time =
 ↪  calculate_lyapunov_exponents(lorenz63, X0, 2000, dt, num_vectors)
```

```
# Fig 6 Plot: Lyapunov Exponents & Noisy attractor
fig = plt.figure(figsize=(6, 3))

gs = fig.add_gridspec(3, 2, width_ratios=[1, 1])

# Left column: 3x1 grid
ax1 = fig.add_subplot(gs[0, 0])
ax2 = fig.add_subplot(gs[1, 0])
ax3 = fig.add_subplot(gs[2, 0])
# Make the left column subplots share the x-axis
ax2.sharex(ax1)
ax3.sharex(ax1)
# Plot the noisy Lorenz attractor
ax1.plot(sol.t, sol.y[0], color='grey', label='x',ls='--')
ax2.plot(sol.t, sol.y[1], color='grey', label='y',ls='--')
ax3.plot(sol.t, sol.y[2], color='grey', label='z',ls='--')

# Plot the perturbed Lorenz attractor
ax1.plot(results_perturbed.t, results_perturbed.y[0],
 ↪  color=plt.cm.Purples(0.9), label='Perturbed x', linestyle='-')
ax2.plot(results_perturbed.t, results_perturbed.y[1],
 ↪  color=plt.cm.Purples(0.7), label='Perturbed y', linestyle='-')
ax3.plot(results_perturbed.t, results_perturbed.y[2],
 ↪  color=plt.cm.Purples(0.5), label='Perturbed z', linestyle='-')

for ax, label in zip([ax1, ax2, ax3], ['x', 'y', 'z']):
    ax.axvspan(0, 8, color="grey", alpha=0.2)
    ax.set_xlim(0, 20)
    ax.set_ylabel(label)
ax1.legend(frameon=False, loc='upper center', fontsize=8,
 ↪  bbox_to_anchor=(0.5,1.4),ncols=2)

ax4 = fig.add_subplot(gs[:, 1])
ax4.set_title('Lyapunov Spectrum')

# Example usage: Add titles to the subplots
# Plot the Lyapunov exponents over time
time = np.linspace(START_TIME, 10, 1000)

for i in range(num_vectors):
    ax4.plot(time[:split_index][:1000],
     ↪  lyapunov_exponents_over_time[i][:split_index][:1000],
     ↪  label=f'$\\lambda_{i+1}$ = {lyapunov_exponents[i]:.3f}')

ax3.set_xlabel('Time')
ax4.set_xlabel('Time')
ax4.legend(frameon=False, loc='upper right', fontsize=8, labelspacing=0.2)
ax4.set_ylim(None, 7)
ax1.tick_params(axis='x', which='both', bottom=False, labelbottom=False)
ax2.tick_params(axis='x', which='both', bottom=False, labelbottom=False)
plt.tight_layout()
plt.savefig("p6.png", dpi=300, bbox_inches='tight')
plt.show()
```

D.6 Embedding Code

```python
def calc_smi_min(x, n_tau):
    smi = np.zeros(n_tau)
    smi[0] = mutual_info_regression(x[:,np.newaxis],x[:])[0]
    for i in tqdm(range(1,n_tau)):
        smi[i] = mutual_info_regression(x[i:,np.newaxis],x[:-i])[0]
    print(smi)
    plt.plot(smi)
    ind_smi_min = np.where(np.diff(smi)>=0)[0][2]
    return [smi, ind_smi_min]

n_tau = 100
min_smi = calc_smi_min(results.y[0], n_tau)
print(min_smi[1])
plt.plot(min_smi[0])
plt.axvline(min_smi[1], color='black', linestyle='--')
plt.plot(min_smi[1], min_smi[0][min_smi[1]], 'ko')
plt.xlabel('Embedding Delay $\\tau$')
plt.ylabel('Self Mutual Information')
```

```python
def embed(x, tau, m):
    n = len(x) - (m - 1) * tau
    Zeta = np.zeros((m, n))
    for i in range(m):
        Zeta[i] = x[i * tau:i * tau + n]
    return Zeta
sol = results

def calc_min_m(x, tau, mtilde_max, w):

    E_mtilde = np.zeros([mtilde_max,])
    Estar_mtilde = np.zeros([mtilde_max,])
    for mtilde in tqdm(range(1,mtilde_max+1)):
        Zeta_mtilde = embed(x, tau, mtilde)
        Zeta_mtilde1 = embed(x, tau, mtilde+1)

        n_samples_embed_mtilde = np.shape(Zeta_mtilde)[1]
        n_samples_embed_mtilde1 = np.shape(Zeta_mtilde1)[1]
        d = distance_matrix(Zeta_mtilde.T,Zeta_mtilde.T)
        d1 = distance_matrix(Zeta_mtilde1.T,Zeta_mtilde1.T)
        ind_d_sort = np.argsort(d)
        ind_nn = np.zeros([n_samples_embed_mtilde,], dtype=int)
        deltax_mtilde = []
        delta_ratio = []
        for i in tqdm(range(n_samples_embed_mtilde)):
            ind_nn[i] = ind_d_sort[i,np.where(ind_d_sort[i,:]>w)[0][0]]
            if ind_nn[i] < n_samples_embed_mtilde1:
                zeta_mtilde_nn = Zeta_mtilde[:,ind_nn[i]]
                zeta_mtilde1_nn = Zeta_mtilde1[:,ind_nn[i]]
                delta_mtilde_nn = d[i,ind_nn[i]]
                delta_mtilde1_nn = d1[i,ind_nn[i]]
                deltax_mtilde_nn = \
                    np.abs(x[i+mtilde*tau] - x[ind_nn[i]+mtilde*tau])
                deltax_mtilde.append(deltax_mtilde_nn)
                if delta_mtilde_nn==0 and delta_mtilde1_nn == 0:
                    delta_ratio.append(np.nan)
                else:
                    delta_ratio.append(delta_mtilde1_nn / delta_mtilde_nn)

        E_mtilde[mtilde-1] = np.nanmean(delta_ratio)
        Estar_mtilde[mtilde-1] = np.nanmean(deltax_mtilde)
    E1 = E_mtilde[1:]/E_mtilde[:-1]
    E2 = Estar_mtilde[1:]/Estar_mtilde[:-1]
    return [E1, E2]

min_m = calc_min_m(sol.y[0], 18, 10, 100)
print(min_m[1])
plt.plot(min_m[0], label='E1')
plt.plot(min_m[1], label='E2')
```

```python
# Set parameters for embedding
embedding_dimension = 4
embedding_delay = 12
PLATEAU_TIME = 50
# Embed the time series
embedded_time_series = embed(results.y[0], embedding_delay,
 ↪  embedding_dimension)

# Calculate metric entropy using RANSAC
forecast_timesteps_array = np.linspace(1, 100, 100).astype(int)
forecast_timesteps_array_linear_only = np.linspace(1,5,5).astype(int)
# Partition the embedded time series
embedded_training, embedded_test = partition_data(embedded_time_series,
 ↪  TRAIN_TEST_SPLIT, NUM_STEPS)

_, x, y = calculate_entropy(embedded_training, embedded_test,
 ↪  forecast_timesteps_array)
h, _, _ = calculate_entropy(embedded_training, embedded_test,
 ↪  forecast_timesteps_array_linear_only)
slopes, intercepts, inlier_masks = calculate_ransac_slope(x, y, PLATEAU_TIME)
```

```
# Add noise to the results

NOISE_FRACTION = 0.05

noise_levels_dim = np.array([(np.max(results.y[0])-np.min(results.y[0])) *
 ↪  NOISE_FRACTION,(np.max(results.y[1])-np.min(results.y[1])) *
 ↪  NOISE_FRACTION, (np.max(results.y[2])-np.min(results.y[2])) *
 ↪  NOISE_FRACTION])

noisy_results = add_noise(results.y, noise_levels_dim)

# Partition the noisy data
noisy_training, noisy_test = partition_data(noisy_results, TRAIN_TEST_SPLIT,
 ↪  NUM_STEPS)

# Calculate self mutual information with noisy data
min_smi_noisy = calc_smi_min(noisy_results[0], n_tau)

# Calculate embedding dimension with noisy data
min_m_noisy = calc_min_m(noisy_results[0], min_smi_noisy[1], 10, 100)

# Plot p8 with noisy data
fig, axes = plt.subplots(1, 2, figsize=(5.8, 2))

# Plot the first subplot
axes[0].plot(min_smi_noisy[0])
axes[0].axvline(min_smi_noisy[1], color='black', linestyle='--',
 ↪  label=f'Optimal Delay $\\tau^* = {min_smi_noisy[1]} $')
axes[0].legend(frameon=False)
axes[0].plot(min_smi_noisy[1], min_smi_noisy[0][min_smi_noisy[1]], 'ko')
axes[0].set_xlabel('Embedding Delay $\\tau$')
axes[0].set_ylabel('Self Mutual Information')

# Plot the second subplot
axes[1].plot(min_m_noisy[0], label='Ratio E1')
axes[1].plot(min_m_noisy[1], label='Ratio E2', color=plt.cm.Purples(0.5))
axes[1].hlines(1, 0, 8, color='black', linestyle='--', lw=0.8,
 ↪  label='$E_\\text{i} = 1$')
axes[1].set_xlabel('Embedding Dimension $\\tilde{m}$')
axes[1].set_ylabel('E1 & E2')
axes[1].legend(frameon=False)

# Adjust layout and display
plt.tight_layout()
plt.savefig("p8_noisy.png", dpi=300, bbox_inches='tight')
plt.show()
```

```
# Create subplots for embedding delay and E1 & E2 for noisy and non-noisy data
fig, axes = plt.subplots(2, 2, figsize=(5.5, 4), sharex=False, sharey=False)

# Plot embedding delay for non-noisy data
axes[0, 0].plot(min_smi[0])
axes[0, 0].axvline(min_smi[1], color='black', linestyle='--', label=f'Optimal
↪  Delay $\\tau^* = {min_smi[1]}$')
axes[0, 0].plot(min_smi[1], min_smi[0][min_smi[1]], 'ko')
axes[0, 0].set_title('a=0')
axes[0, 0].set_xlabel('Embedding Delay $\\tau$')
axes[0, 0].set_ylabel('Self Mutual Information')
axes[0, 0].legend(frameon=False)

# Plot E1 & E2 for non-noisy data
axes[0, 1].plot(min_m[0], label='Ratio E1')
axes[0, 1].plot(min_m[1], label='Ratio E2', color=plt.cm.Purples(0.5))
axes[0, 1].hlines(1, 0, len(min_m[0]), color='black', linestyle='--', lw=0.8,
↪  label='$E_\\text{i} = 1$')
axes[0, 1].set_title('a=0')
axes[0, 1].set_xlabel('Embedding Dimension $\\tilde{m}$')
axes[0, 1].set_ylabel('E1 & E2')
axes[0, 1].legend(frameon=False)

# Plot embedding delay for noisy data
axes[1, 0].plot(min_smi_noisy[0])
axes[1, 0].axvline(min_smi_noisy[1], color='black', linestyle='--',
↪  label=f'Optimal Delay $\\tau^* = {min_smi_noisy[1]}$')
axes[1, 0].plot(min_smi_noisy[1], min_smi_noisy[0][min_smi_noisy[1]], 'ko')
axes[1, 0].set_title('a=0.05')
axes[1, 0].set_xlabel('Embedding Delay $\\tau$')
axes[1, 0].set_ylabel('Self Mutual Information')
axes[1, 0].legend(frameon=False)

# Plot E1 & E2 for noisy data
axes[1, 1].plot(min_m_noisy[0], label='Ratio E1')
axes[1, 1].plot(min_m_noisy[1], label='Ratio E2', color=plt.cm.Purples(0.5))
axes[1, 1].hlines(1, 0, len(min_m_noisy[0]), color='black', linestyle='--',
↪  lw=0.8, label='$E_\\text{i} = 1$')
axes[1, 1].set_title('a=0.05')
axes[1, 1].set_xlabel('Embedding Dimension $\\tilde{m}$')
axes[1, 1].set_ylabel('E1 & E2')
axes[1, 1].legend(frameon=False)

# Adjust layout and display
plt.tight_layout()
plt.savefig("p7.png", dpi=300, bbox_inches='tight')
plt.show()
```

```
# Adjust the layout to split noisy and non-noisy into two columns
fig, axs = plt.subplots(nrows=4, ncols=2, figsize=(6, 6),
↪  sharex=True,sharey=True)

# Plot non-noisy data in the first column
for dim in range(len(y[0, :])):
    axs[dim, 0].plot(x, np.exp(y[:, dim]), label='a=0', alpha=0.8,
    ↪  color=plt.cm.Purples(0.9))
    axs[dim, 0].plot(x[:PLATEAU_TIME][inlier_masks[dim]],
                     np.exp(slopes[dim] * x[:PLATEAU_TIME][inlier_masks[dim]]
                     ↪  + intercepts[dim]),
              color='black', alpha=0.8, lw=2, ls='--', label='Linear Fit')
    axs[dim, 0].text(0.05, 0.9, f'$h_{{{dim}}}$ = {dt * slopes[dim] / 2:.2f}',
                     fontsize=10, color='black', ha='left', va='top',
                     ↪  transform=axs[dim, 0].transAxes)
    axs[dim, 0].set_ylabel(f'1-r (Dimension {dim})')
    axs[dim, 0].set_yscale('log')
axs[0, 0].set_title(f'a=0 (No Noise)')

# Plot noisy data in the second column
for dim in range(len(y_noisy[0, :])):

    axs[dim, 1].plot(0, 0, label='a=0', alpha=0.8, color=plt.cm.Purples(0.9))
    axs[dim, 1].plot(x_noisy, np.exp(y_noisy[:, dim]), label='a=0.05',
    ↪  alpha=0.8, color=plt.cm.Purples(0.5))
    axs[dim, 1].plot(x_noisy[:PLATEAU_TIME][inlier_masks_noisy[dim]],
                     np.exp(slopes_noisy[dim] *
                     ↪  x_noisy[:PLATEAU_TIME][inlier_masks_noisy[dim]] +
                     ↪  intercepts_noisy[dim]),
                     ls='--', color='black', alpha=0.8, lw=2, label='Linear
                     ↪  Fit')
    axs[dim, 1].text(0.05, 0.9, f'$h_{{{dim}}}$ = {dt * slopes_noisy[dim] /
    ↪  2:.2f}',
                     fontsize=10, color='black', ha='left', va='top',
                     ↪  transform=axs[dim, 1].transAxes)
    axs[dim, 1].set_yscale('log')
axs[0, 1].set_title(f' a=0.05 (Noise)')

# Set x-axis label for the bottom row
for ax in axs[-1, :]:
    ax.set_xlabel('$t_\\text{fc}$')

# Add legends
axs[3, 1].legend(loc='lower right', frameon=False)

# Save and show the plot
plt.tight_layout()
plt.savefig("p8.png", dpi=300, bbox_inches='tight')
plt.show()
```

D.7 Appendix Code

```
# Appendix Plot
# Metric entropy calculation for a variety of number of steps
DEPTH_PLOT = 4
num_steps_array = np.linspace(500,40000,20).astype(int)
entropies = []

for num_steps in tqdm.tqdm(num_steps_array):
    K = int(num_steps/20)
    dt = (END_TIME - START_TIME) / num_steps
    results = integrate_system(X0, START_TIME, END_TIME, num_steps)
    training, test = partition_data(results, TRAIN_TEST_SPLIT, num_steps)
    forecast_timesteps_array = np.linspace(1, DEPTH_PLOT,
     ↪  DEPTH_PLOT).astype(int)
    h, x, y = calculate_entropy(training, test, forecast_timesteps_array)
    entropies.append(h)

# Print the results
for steps, h in zip(num_steps_array, entropies):
    print(f"NUM_STEPS: {steps}, Entropy: {sum(h)}, Components: {h[0]}, {h[1]},
     ↪  {h[2]}")

corrected_entropies = []

for steps, h in zip(num_steps_array, entropies):
    dt = 400/steps
    print(f"NUM_STEPS: {steps}, Entropy: {sum(h)*dt}, Components: {h[0]*dt},
     ↪  {h[1]*dt}, {h[2]*dt}")
    corrected_entropies.append((h)*dt)
```

```
fig, axs = plt.subplots(1, 2, figsize=(5.8, 2.5), sharex=True)
axs[0].plot(num_steps_array, corrected_entropies,label=['$h_x$', '$h_y$',
 ↪  '$h_z$'])
axs[1].plot(num_steps_array, entropies,label=['$h_x$', '$h_y$', '$h_z$'])
plt.legend()
axs[0].set_ylabel('Entropy (per timestep)')
axs[1].set_ylabel('Entropy (per time unit)')
axs[0].set_xlabel('Number of Steps')
axs[1].set_xlabel('Number of Steps')

plt.tight_layout()
plt.savefig('a1', dpi=300, bbox_inches='tight')
plt.show()
```