# Time series analysis: Nonlinear time series analysis

ΕΠΛ 428: IOT PROGRAMMING

Dr. Panayiotis Kolios
Assistant Professor, Dept. Computer Science,
KIOS CoE for Intelligent Systems and Networks
Office: FST 01, 116
Telephone: +357 22893450 / 22892695
Web: https://www.kios.ucy.ac.cy/pkolios/

Πανεπιστήμιο Κύπρου

- Dynamical systems and nonlinear dynamics
- Chaotic systems
- Higher-dimensional continuous-time systems
- Phase (state) space of a system
- Fractal dimensions
- Phase space reconstruction and Taken's embedding theorem
- Forecasting nonlinear time series

```
import numpy as np
import matplotlib.pyplot as plt
import plotly.graph_objects as go
from scipy.integrate import solve_ivp
from sklearn.neighbors import NearestNeighbors
from sklearn.ensemble import RandomForestRegressor
from sklearn.metrics import mean_squared_error
from sklearn.decomposition import PCA
from reservoir_computing.reservoir import Reservoir
from reservoir_computing.utils import make_forecasting_dataset
from tsa_course.lecture11 import computeLE, plot_bifurcation_diagram
np.random.seed(0)
```

- A dynamic system is a set of functions (rules, equations) specifying how variables change over time
- For example:

$$x(t+1) = a \cdot x(t) + b \cdot y(t)$$

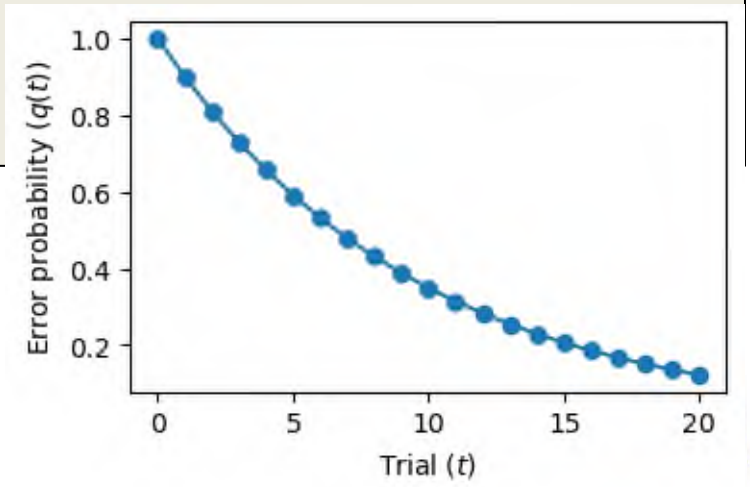
- Here, $x$ and $y$ represent the variables of a 2-dimensional system, while $a, b, c$ are the parameters
- Variables change over time, parameters do not
  - In a discrete system the variables are restricted to integer values
  - In a continuous system, variables can assume real values
- The system can be stochastic (one set of rules, many realizations) or deterministic (one set of rules, one realization)
- The state of a dynamic system at time is specified by the current value of its variables $x(t), y(t), \ldots$
- The process of calculating the new state of a discrete system is called iteration
- To evaluate how a system behaves, we need the functions, the parameter values, and the initial conditions, e.g., $x(0), y(0), \ldots$

- Let's consider a classic example: *alpha model*
- It specifies how $q(t)$, the probability of making an error on trial, changes from one trial to the next:

$$q(t+1) = \beta q(t)$$

- The new error probability is diminished by $\beta \in (0,1)$

```
def alpha(beta, q):
    return beta * q # Sytem equation

beta = 0.9          # System parameter
q = [1]             # Initial condition

for _ in range(20):
    q.append(alpha(beta, q[-1])) # Iteration

fig = plt.figure(figsize=(4, 2.5))
plt.plot(q, 'o-')
plt.xlabel('Trial ($t$)')
plt.ylabel('Error probability ($q(t)$)')
plt.show()
```

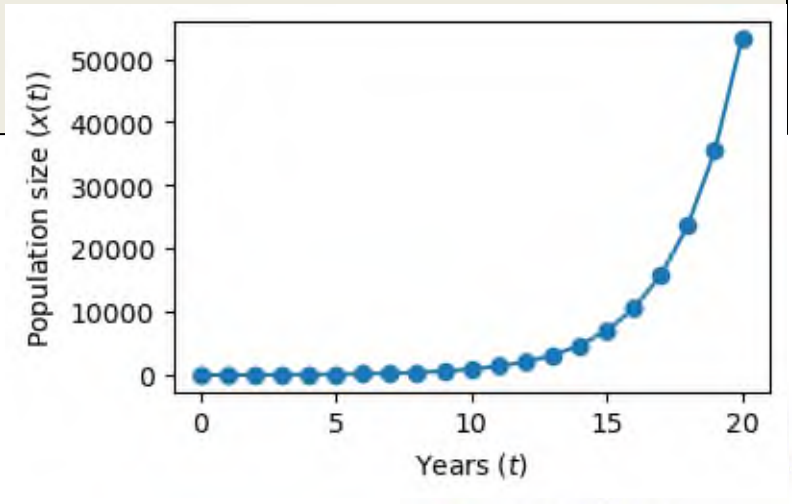


- Linear systems are those with linear state updates equations, i.e., something of the form $y(t) = ax(t) + b$
- The alpha model is linear
- Do not be confused by the non-linear curve we plot above: that is the behavior of the system, not the functions that specify its changes

- Logistic map
  - Is a very famous equation to describe the growth of a population
  - Is a non-linear model
  - Is often used to introduce the notion of chaos

- Let's first use a simple linear model to describe the population growth $x(t+1) = rx(t)$, with $r$ the growth rate
- If $r > 1$, the population grows exponentially without limit

```
def simple_growth(r, x):
    return r * x

r = 1.5                 # Each year x is increased by 50%
x = [16]                # Initial number of individuals
for _ in range(20):
    x.append(simple_growth(r, x[-1]))

fig = plt.figure(figsize=(4, 2.5))
plt.plot(x, 'o-')
plt.xlabel('Years ($t$)')
plt.ylabel('Population size ($x(t)$)')
plt.show()
```



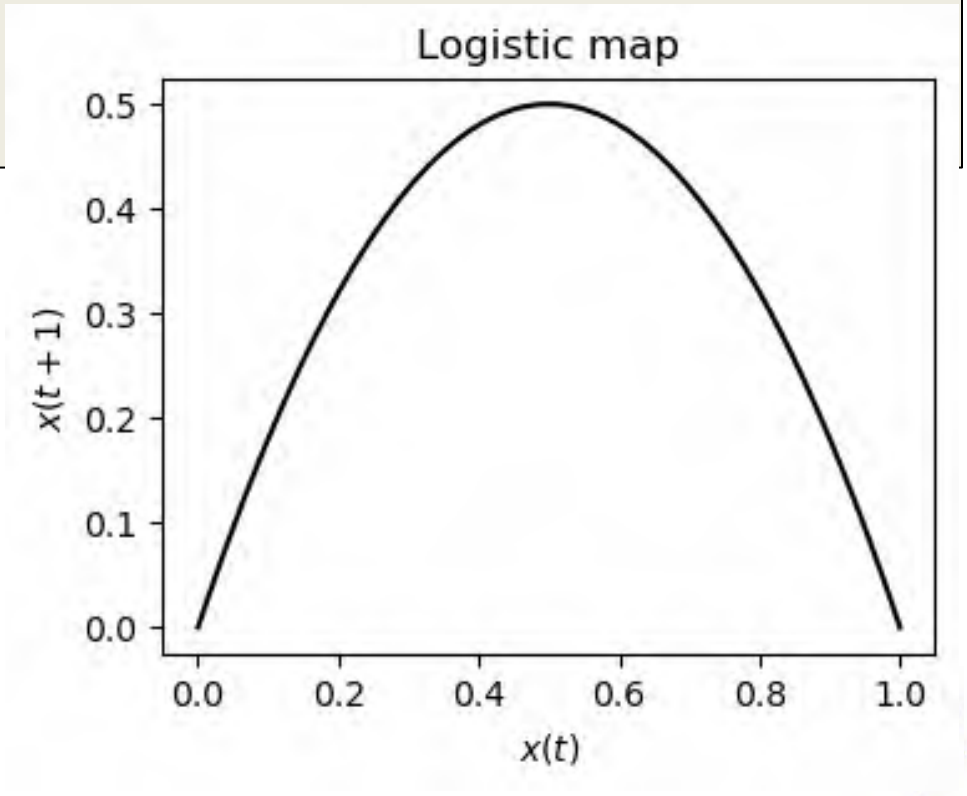
- Logistic Map prevents unlimited growth by inhibiting growth whenever it achieves a high level
    - This is achieved by introducing an additional term $(1 - x(t))$
- The growth measure $x$ is also rescaled so that the maximum value that $x$ can achieve is 1
- So if the maximum size is 8 billions, $x$ is the proportion of that maximum
- The new growth model is

$$x(t+1) = rx(t)[1 - x(t)], r \in [0,4]$$

- and $1 - x(t)$ inhibits the growth because
    - as $x(t)$ approaches 1, $[1 - x(t)]$ approaches 0

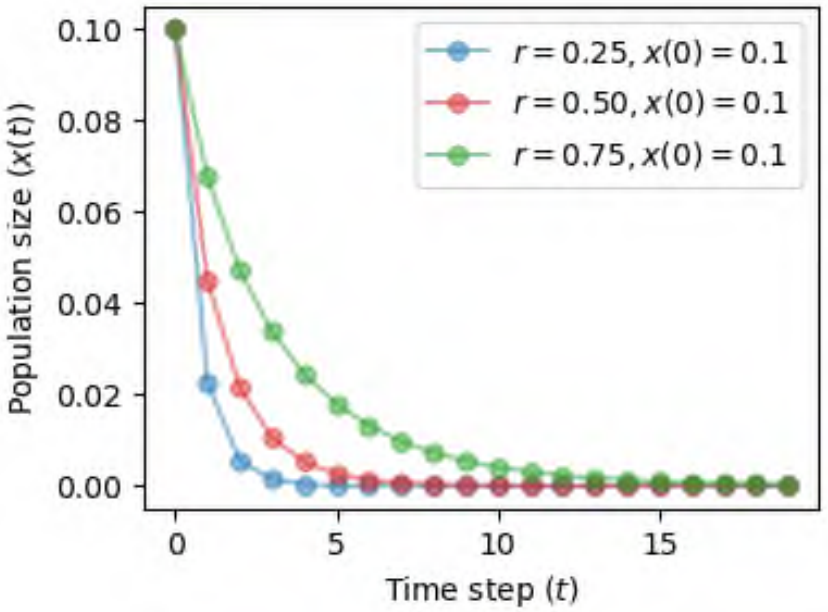
- Let's first use a simple linear model to describe the population growth $x(t+1) = rx(t)$, with $r$ the growth rate
- If $r > 1$, the population grows exponentially without limit

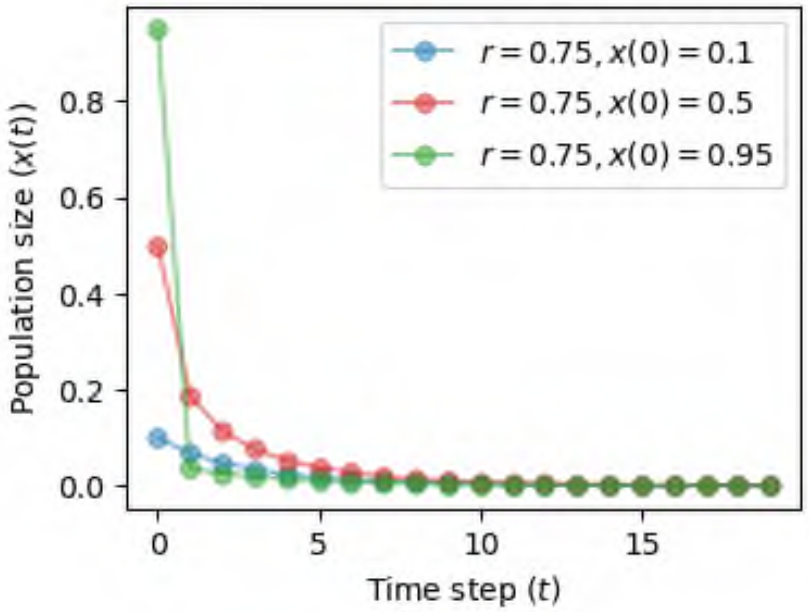
```
def logistic(r, x):
    return r * x * (1 - x)

x = np.linspace(0, 1)
fig, ax = plt.subplots(1, 1, figsize=(4, 3))
ax.plot(x, logistic(2, x), 'k')
ax.set_title('Logistic map')
ax.set_xlabel('$x(t)$')
ax.set_ylabel('$x(t+1)$')
plt.show()
```



- logistic map changes drastically its behavior depending the the value of the parameter $r$
- Regime $r < 1$
  - the system x will go toward 0 (one-point attractor)

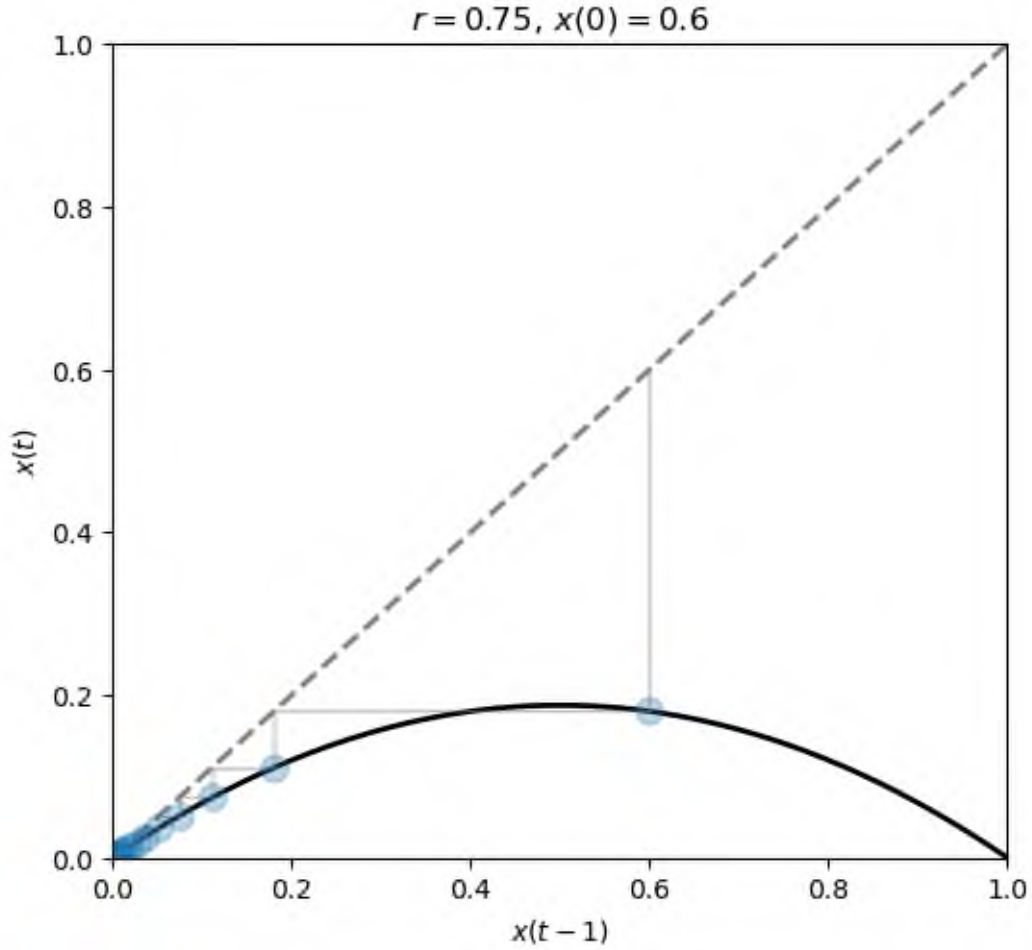
```
fig, ax = plt.subplots(1, 1, figsize=(4, 3))
plot_time_series(r=0.25, x0=0.1, n=20, color='tab:blue', ax=ax)
plot_time_series(r=0.5, x0=0.1, n=20, color='tab:red', ax=ax)
plot_time_series(r=0.75, x0=0.1, n=20, color='tab:green', ax=ax)
```



- See a current state $x(t)$ relates to previous state $x(t-1)$

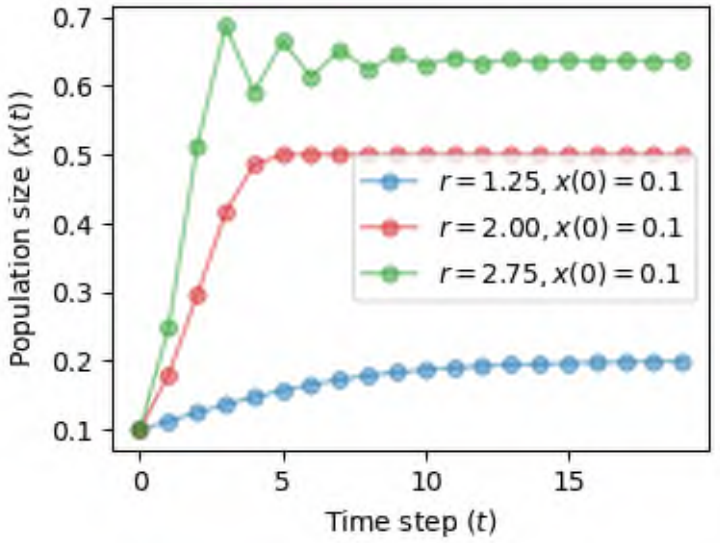
```
fig, ax = plt.subplots(1, 1, figsize=(6, 5.5))
plot_system(r=0.75, x0=.6, n=20, ax=ax)
```

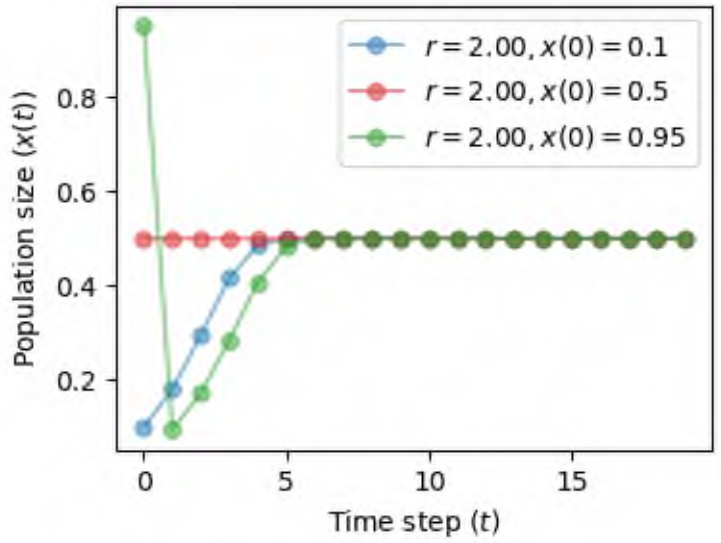


- Regime $1 < r < 3$
  - still a one-point attractor, but now the shape changes with the value of $r$

```
fig, ax = plt.subplots(1, 1, figsize=(4, 3))
plot_time_series(r=1.25, x0=0.1, n=20, color='tab:blue', ax=ax)
plot_time_series(r=2.0, x0=0.1, n=20, color='tab:red', ax=ax)
plot_time_series(r=2.75, x0=0.1, n=20, color='tab:green', ax=ax)
```

  - As before, the initial state is inconsequential and its effect is washed-out eventually

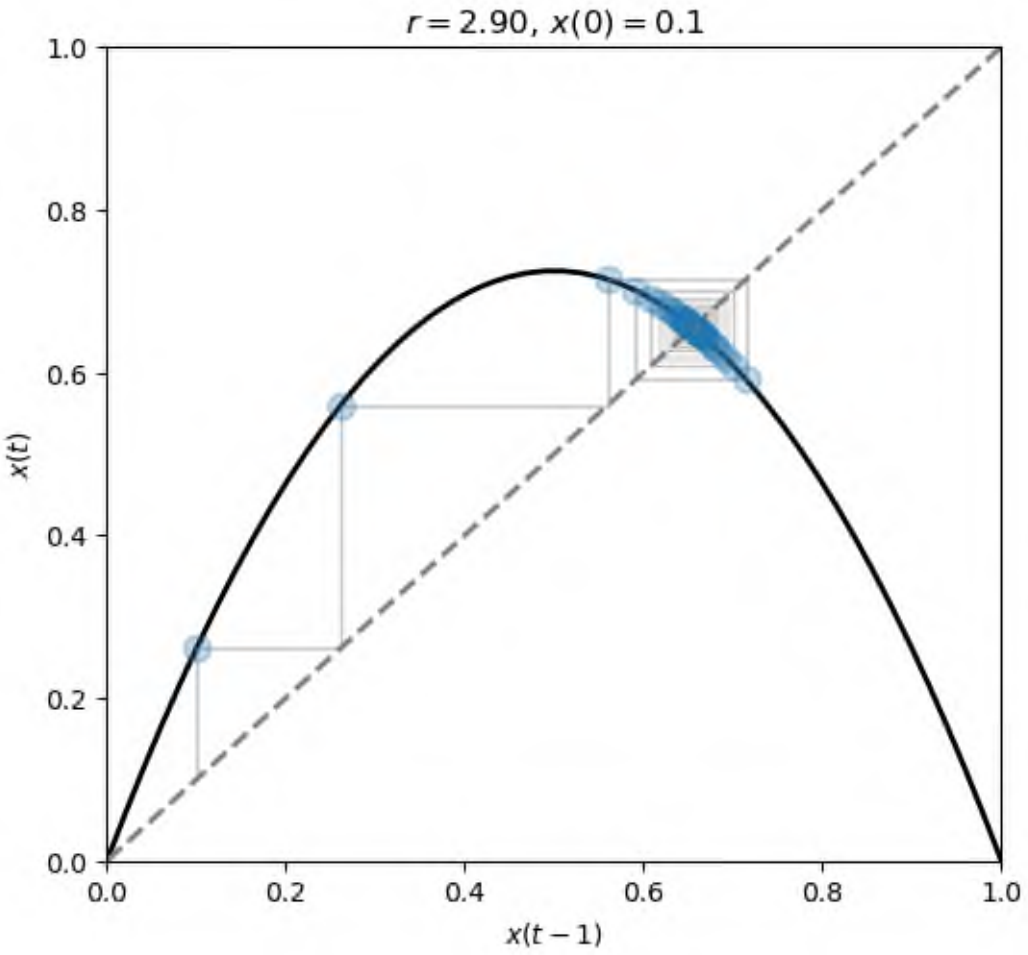



Πανεπιστήμιο Κύπρου

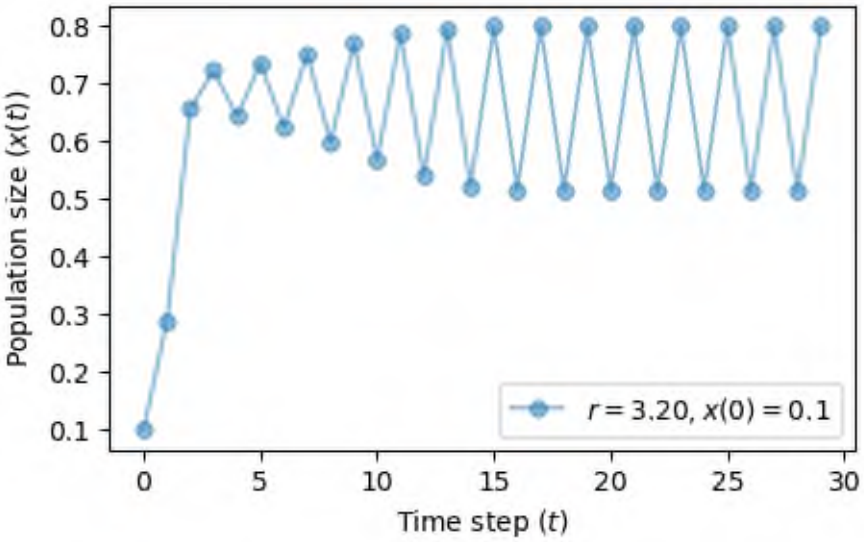
- As we get closer to $r = 3$ the state oscillates more and more before settling down to its attractor

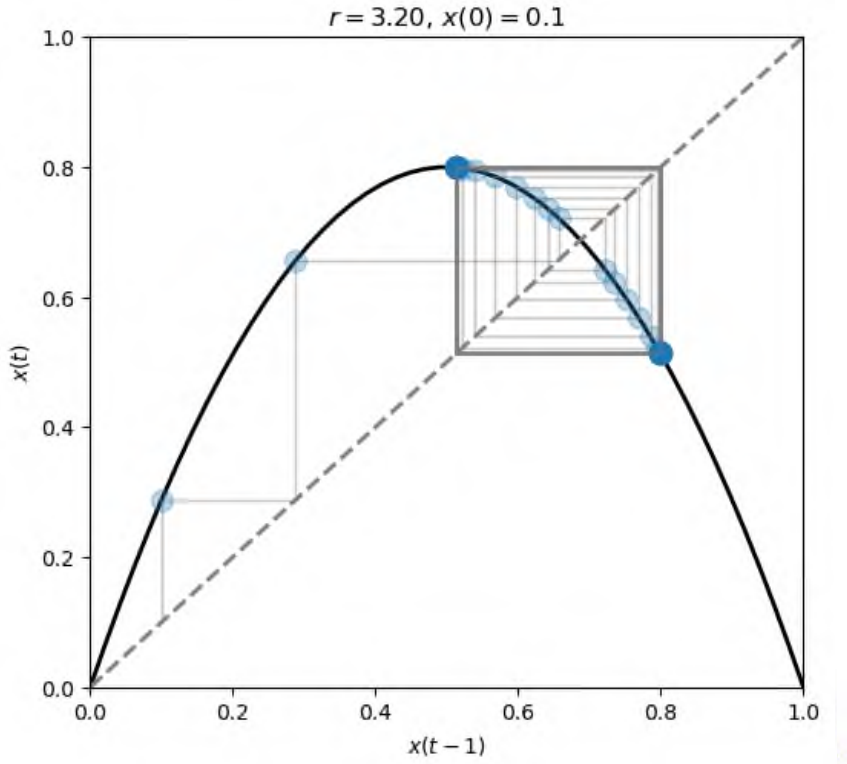
```
fig, ax = plt.subplots(1, 1, figsize=(6, 5.5))
plot_system(r=2.9, x0=.1, n=30, ax=ax)
```



- Regime $r > 3$
  - system starts oscillating between two points
  - We have a two-point attractor
- The phenomenon called bifurcation, or period-doubling

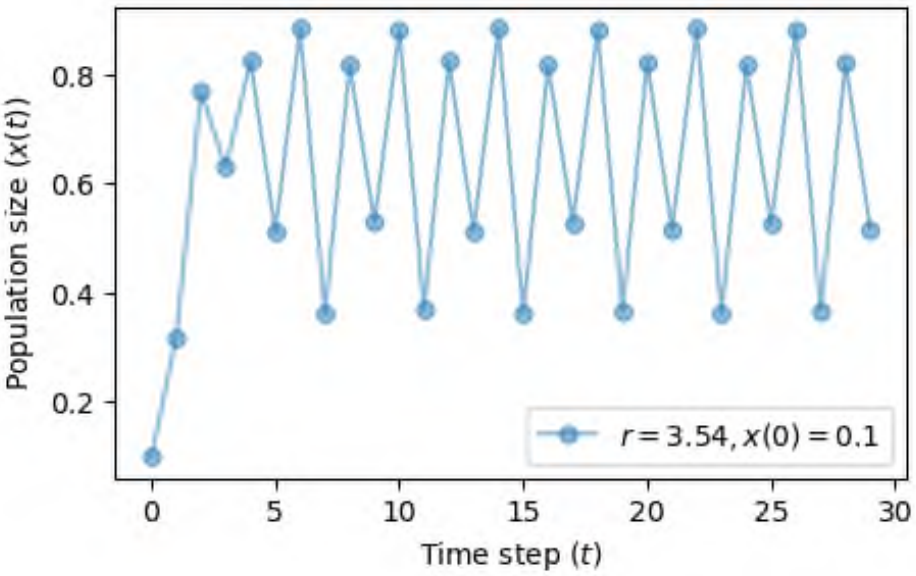
```
fig, ax = plt.subplots(1, 1, figsize=(5, 3))
plot_time_series(r=3.2, x0=0.1, n=30, color='tab:blue', ax=ax)
```

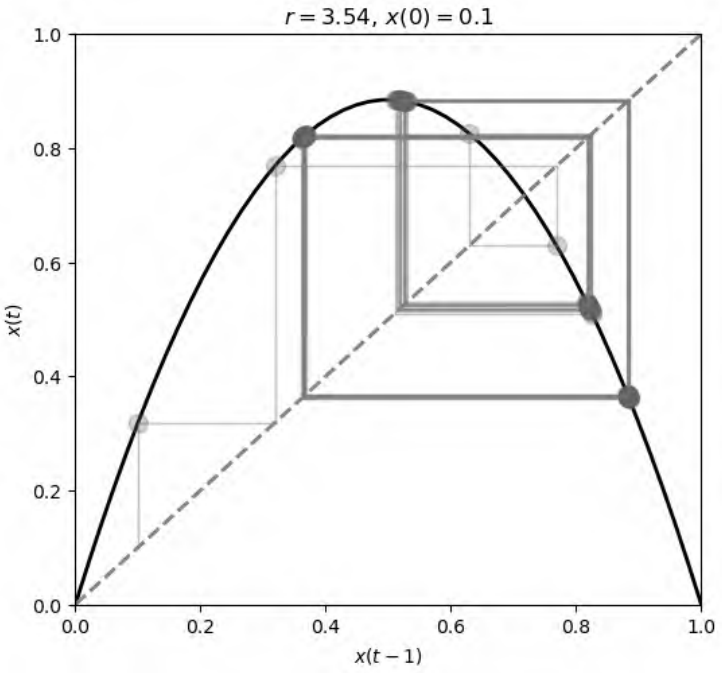


- At $r = 3.54$
  - We have another bifurcation, 4-point attractor

```
fig, ax = plt.subplots(1, 1, figsize=(5, 3))
plot_time_series(r=3.54, x0=0.1, n=30, color='tab:blue', ax=ax)
```



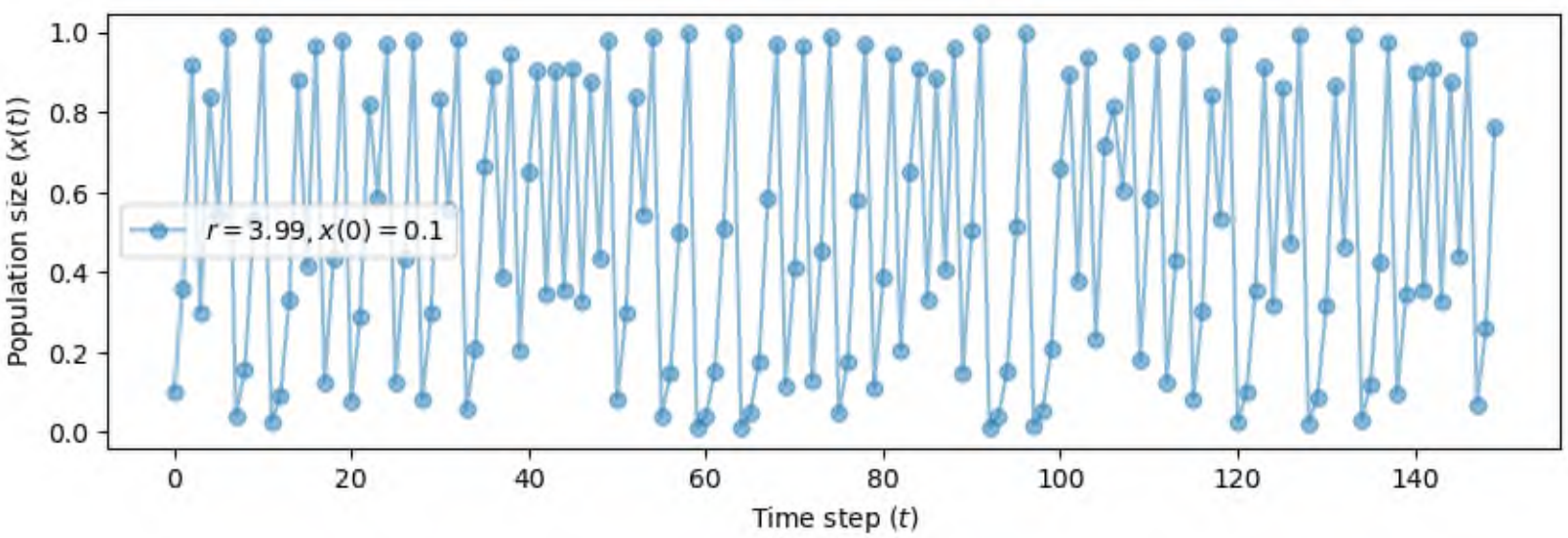



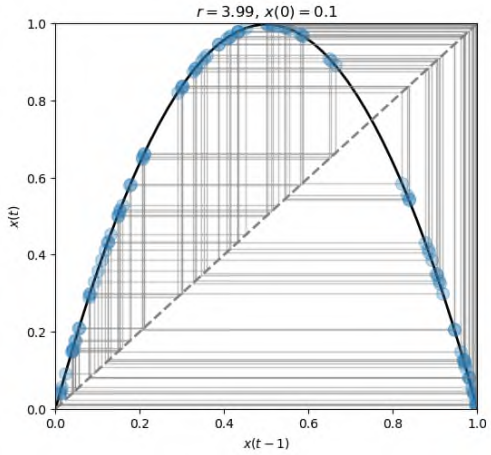

- Number of bifurcations keeps growing as $r$ increases
- Results to N-point attractor that looks “unstable”

Πανεπιστήμιο Κύπρου

- This a characterizing property of chaotic systems.

```
fig, ax = plt.subplots(1, 1, figsize=(5, 3))
plot_time_series(r=3.99, x0=0.1, n=30, color='tab:blue', ax=ax)
```



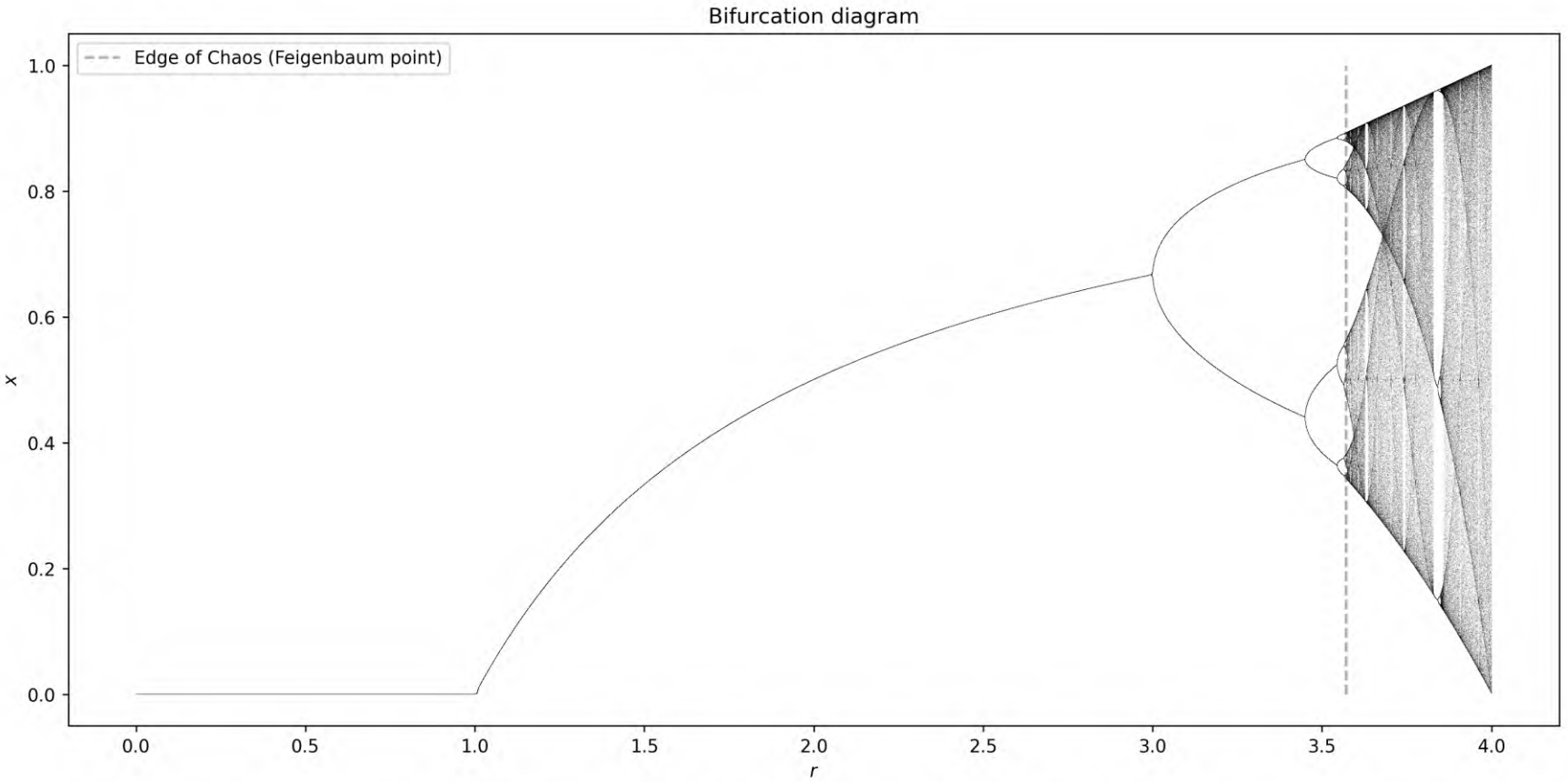

Πανεπιστήμιο Κύπρου

- A bifucation is a period-doubling, i.e., a change from an $N$-point attractor to a $2N$-point attractor
- In the Logistic map, it occurs when the control parameter r changes
- A bifurcation diagram is a visual summary of the succession of period-doubling produced as the control parameter changes

- To compute the bifurcation diagram of the logistic map.
  - On the x-axis, we have the value of $r$
  - On the y-axis, we have the number of distinct points the system settles down to

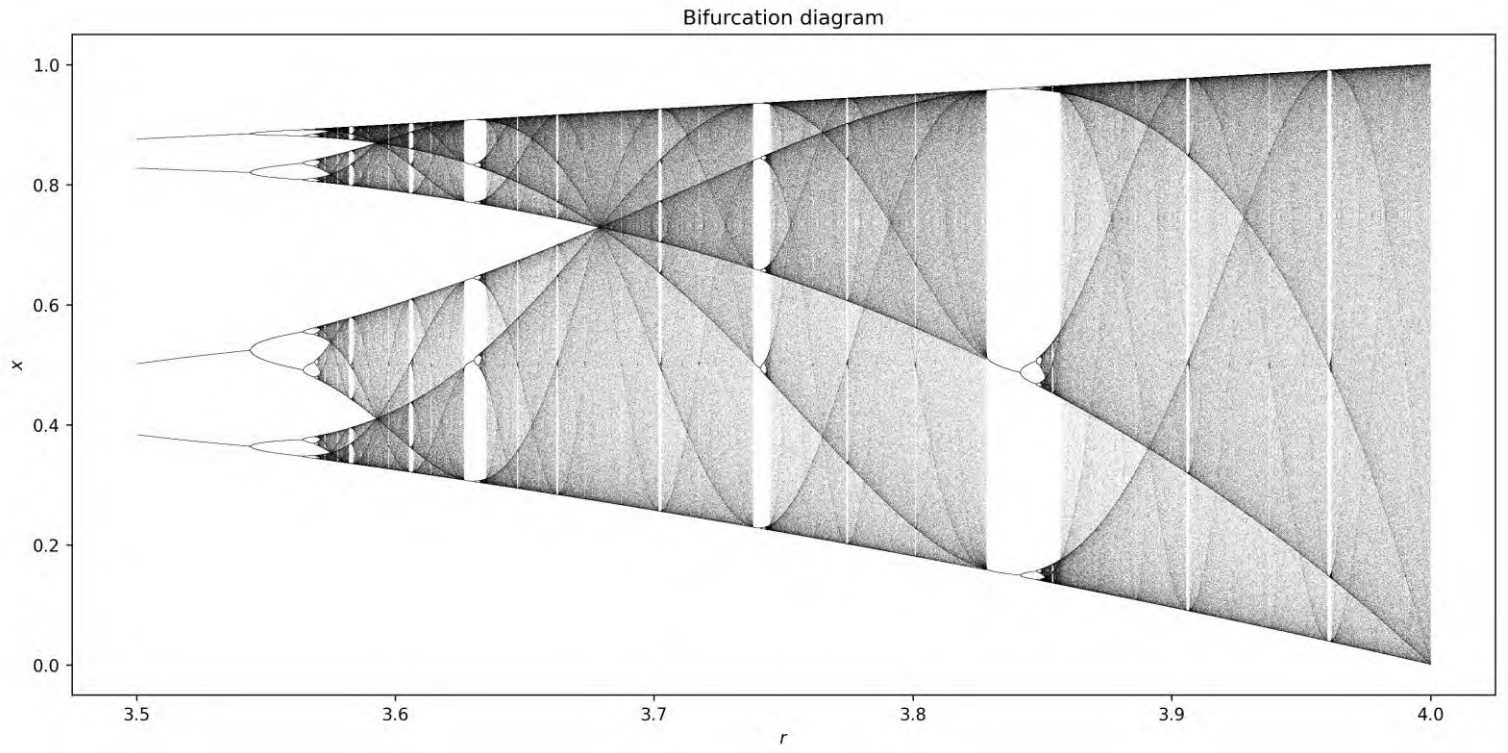
```
def bifuracion_diagram(r, ax, iterations=1000, last=100):
    x = 1e-5 * np.ones_like(r)
    for i in range(iterations):
        x = logistic(r, x)
        if i >= (iterations - last): # plot only the 'last' last iterations
            ax.plot(r, x, ',k', alpha=.25)
    ax.set_title("Bifurcation diagram")
    ax.set_xlabel("$r$")
    ax.set_ylabel("$x$")
```

```
r = np.linspace(0, 4.0, 30000)

fig, ax = plt.subplots(1, 1, figsize=(15, 7), dpi=250)
bifuracion_diagram(r, ax)
ax.vlines(3.569945, 0, 1, 'tab:red', linestyles='--', alpha=0.5, label='Edge of
Chaos (Feigenbaum point)')
plt.legend()
plt.show()
```




- We see that for $r < 1$, zero is the one point attractor.
- For $1 < r < 3$ we still have one-point attractors, but the 'attracted' value of $x$ increases as r increases
- Bifurcations occur at $r = 3, 3.45, 3.54, 3.564, 3.569$, etc
- Approximately at $r = 3.57$ our system becomes chaotic
- However, the system is not chaotic for all values of $r$ greater than 3.57

```
n = 30000 # number of values of r
r = np.linspace(3.5, 4.0, n)

fig, ax = plt.subplots(1, 1, figsize=(15, 7), dpi=250)
bifuracion_diagram(r, ax)
plt.show()
```



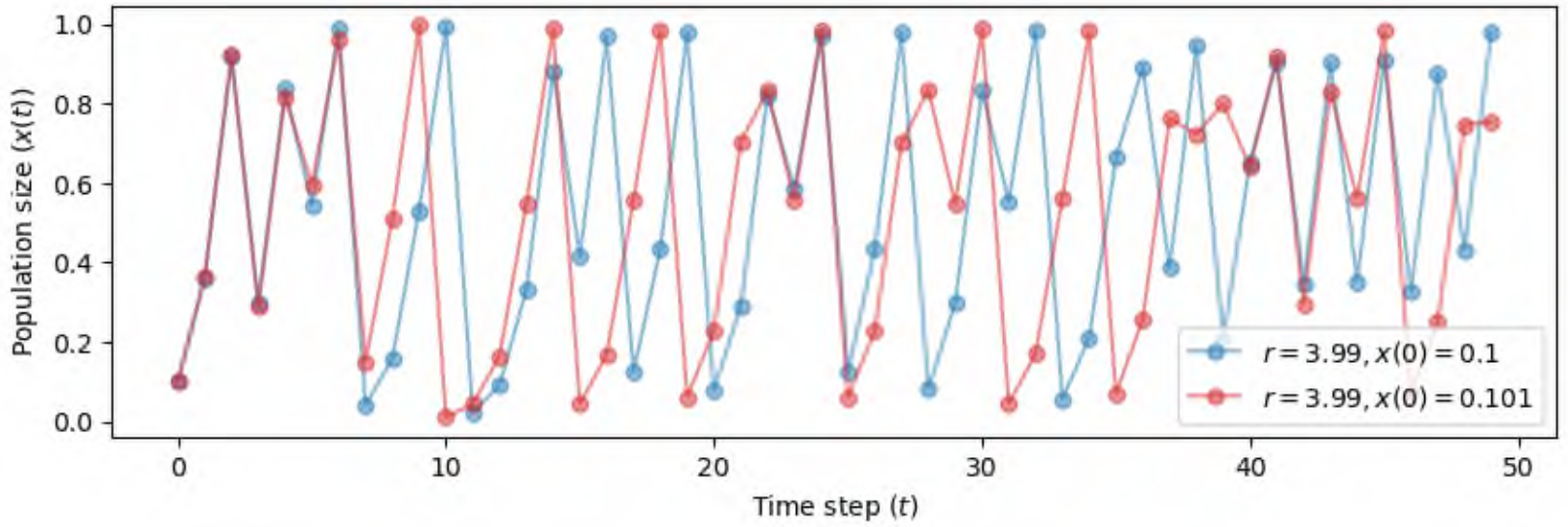

- At several values of $r > 3.57$ there are regions where a small number of $x$-values are visited
- These regions produce the white spaces in the diagram
- For example, at $r = 3.83$ there is a three-point attractor
- Between 3.57 and 4 there is a rich interleaving of chaos and order
- A small change in $r$ can make a stable system chaotic, and vice-versa

Πανεπιστήμιο Κύπρου

- Many real-world phenomena are **chaotic**, particularly those that involve nonlinear interactions among many agents (complex systems)
- Examples can be found in meteorology, economics, biology, and other disciplines

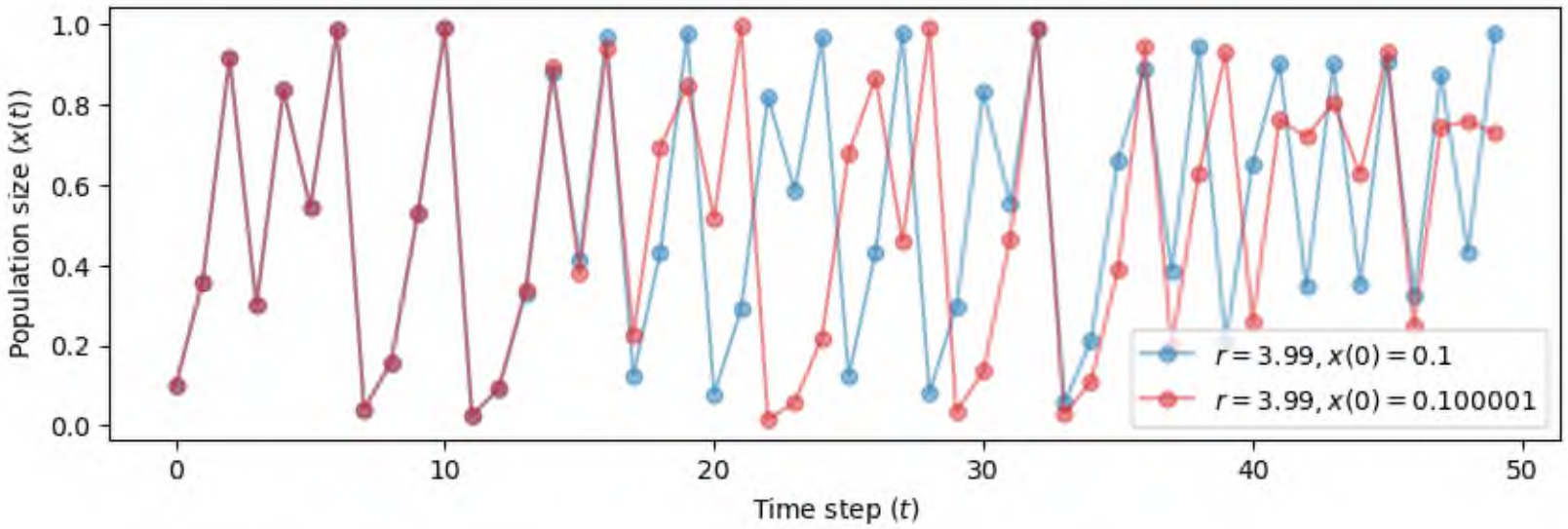
- Sensitivity to initial conditions
  - A characterizing feature of chaotic systems is their sensitivity to initial conditions
  - Logistic map for $r = 3.99$
  - We will start from two very close initial conditions:
    - $x(0) = 0.1$ and $x(0) = 0.101$

```
fig, ax = plt.subplots(1, 1, figsize=(10, 3))
plot_time_series(r=3.99, x0=0.1, n=50, color='tab:blue', ax=ax)
plot_time_series(r=3.99, x0=0.101, n=50, color='tab:red', ax=ax)
```



- Sensitivity to initial conditions
  - Even for two really close points $x(0) = 0.1$ and $x(0) = 0.100001$

```
fig, ax = plt.subplots(1, 1, figsize=(10, 3))
plot_time_series(r=3.99, x0=0.1, n=50, color='tab:blue', ax=ax)
plot_time_series(r=3.99, x0=0.100001, n=50, color='tab:red', ax=ax)
```




- Hence in a chaotic system, no matter how close the initial conditions are, if they are different, the trajectories will eventually diverge

Πανεπιστήμιο Κύπρου

## Detecting chaos

- How do we determine if a system has a contractive or chaotic dynamics?

- There are several tools
    1. Measure the sensitivity to initial conditions
        - The Lyapunov exponent
    2. Return maps
    3. Power Spectrum

- The Lyapunov exponent quantifies the rate at which nearby trajectories in the system diverge or converge over time
- It tells us how sensitive a system is to its initial conditions, a property often associated with chaotic behavior
- There are actually several Lyapunov exponents for a given system, corresponding to different directions in the system's phase space
- The **largest** Lyapunov exponent is most commonly used to detect chaos
- A system is **considered chaotic** if it has **at least one positive** Lyapunov exponent

- Let a dynamical system be defined as:

$$\dot{x} = f(x, t)$$

- where
  - $\dot{x}$ is the time derivative of $x$
  - $f(x, t)$ is the function defining the system's evolution over time
- Consider two nearby points in the system's phase space $x_0$ and $x_0 + \delta x_0$, where $\delta x_0$ is a small perturbation
- Their trajectories diverge over time according to

$$\delta x(t) \approx \delta x_0 e^{\lambda t}$$

- where
  - $\delta x(t)$ is the separation between the two trajectories at time $t$
  - $\lambda$ is the Lyapunov exponent

- The exponent is calculated as follows:

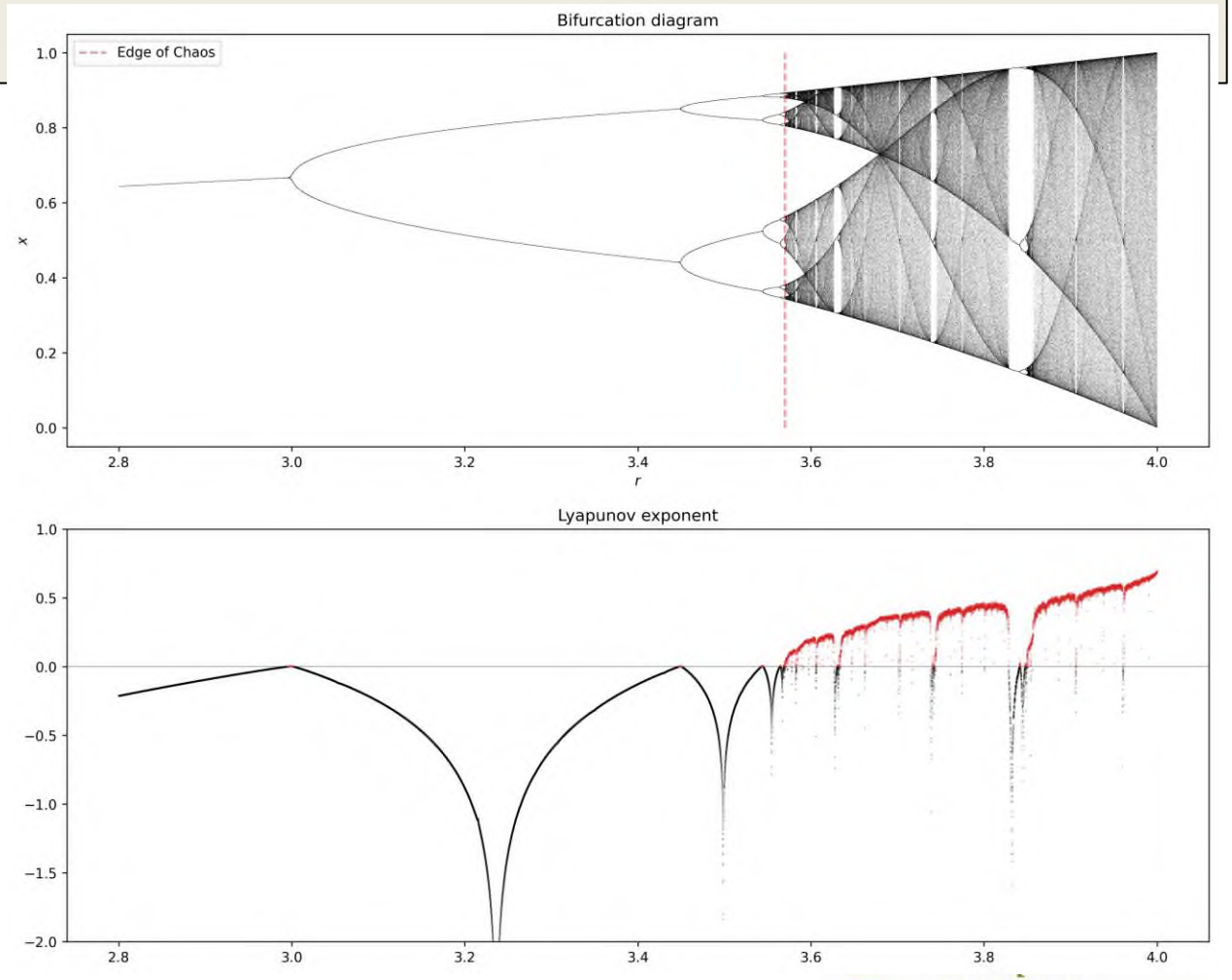
$$\lambda = \lim_{t\to\infty} \lim_{|\delta x_0|\to 0} \frac{1}{t} \ln\left(\frac{|\delta x(t)|}{|\delta x_0|}\right)$$

- This limit, if it exists, gives the average rate of exponential divergence (if $\lambda > 0$) or convergence (if $\lambda < 0$) of trajectories starting from infinitesimally close initial conditions
- The exponent can often be computed analytically
- Otherwise, computed numerically by observing how small perturbations evolve over time
    - A positive Lyapunov exponent implies that small differences in initial conditions lead to exponential divergence of trajectories
    - This makes long-term predictions very difficult despite the system being deterministic
- This sensitivity to initial conditions is often referred to as the **butterfly effect** in the context of chaos theory

- Function to plot the Lyapunov exponent in the logistic map as we increase $r$
- We will color in red values of $r$ associated with $\lambda > 0$ , which indicates that the system has a chaotic behavior
- To check if the Lyapunov exponent works as a chaos detector, we will compare it with the bifurcation map

```
def lyapunov(r, ax, iterations=1000):
    x = 1e-5 * np.ones_like(r)
    lyapunov = np.zeros_like(r)
    for i in range(iterations):
        x = logistic(r, x)
        lyapunov += np.log(abs(r - 2 * r * x)) # Partial sum of the Lyapunov exponent
    ax.axhline(0, color='k', lw=.5, alpha=.5)

    # Negative Lyapunov exponent
    ax.plot(r[lyapunov < 0],
            lyapunov[lyapunov < 0] / iterations,
            '.k', alpha=.5, ms=.5)

    # Positive Lyapunov exponent
    ax.plot(r[lyapunov >= 0],
            lyapunov[lyapunov >= 0] / iterations,
            '.', color='tab:red', alpha=.5, ms=.5)
    ax.set_ylim(-2, 1)
    ax.set_title("Lyapunov exponent")
```

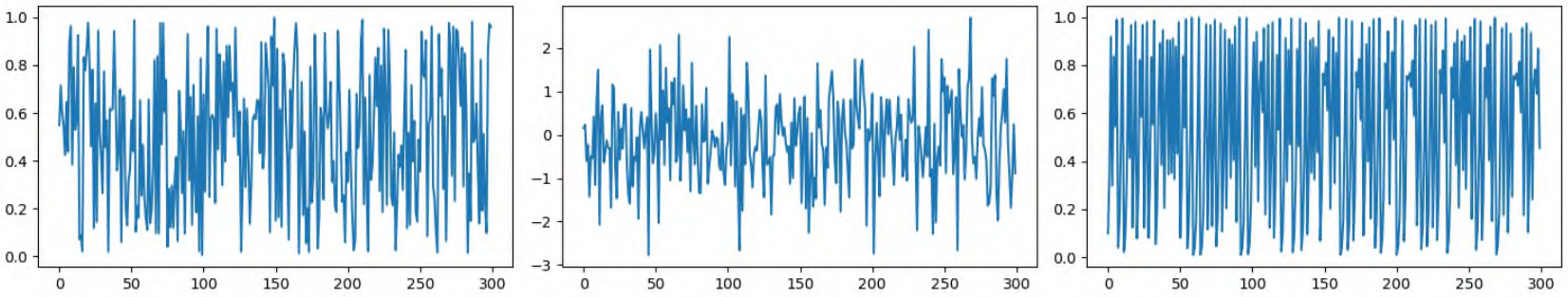
```
r = np.linspace(2.8, 4.0, 30000)

fig, axes = plt.subplots(2, 1, figsize=(15, 12), dpi=250)
bifuracion_diagram(r, axes[0])
axes[0].vlines(3.569945, 0, 1, 'tab:red', linestyles='--', alpha=0.5, label='Edge of
Chaos')
axes[0].legend()
lyapunov(r, axes[1])
plt.show()
```



- Another important distinction is between chaos and randomness
    - At first glance, a chaotic and a stochastic processes look alike
- Consider the time series of samples generated by:
    - a uniform distribution,
    - a normal distribution,
    - the Logistic map for $r = 3.99$

```
uniform_data = np.random.uniform(low=0, high=1, size=300) # Random uniform

normal_data = np.random.randn(300) # Random normal

lgt_data = np.empty(300) # Logistic map
lgt_data[0] = 0.1
for i in range(1, len(lgt_data)):
    lgt_data[i] = logistic(3.99, lgt_data[i-1])
```

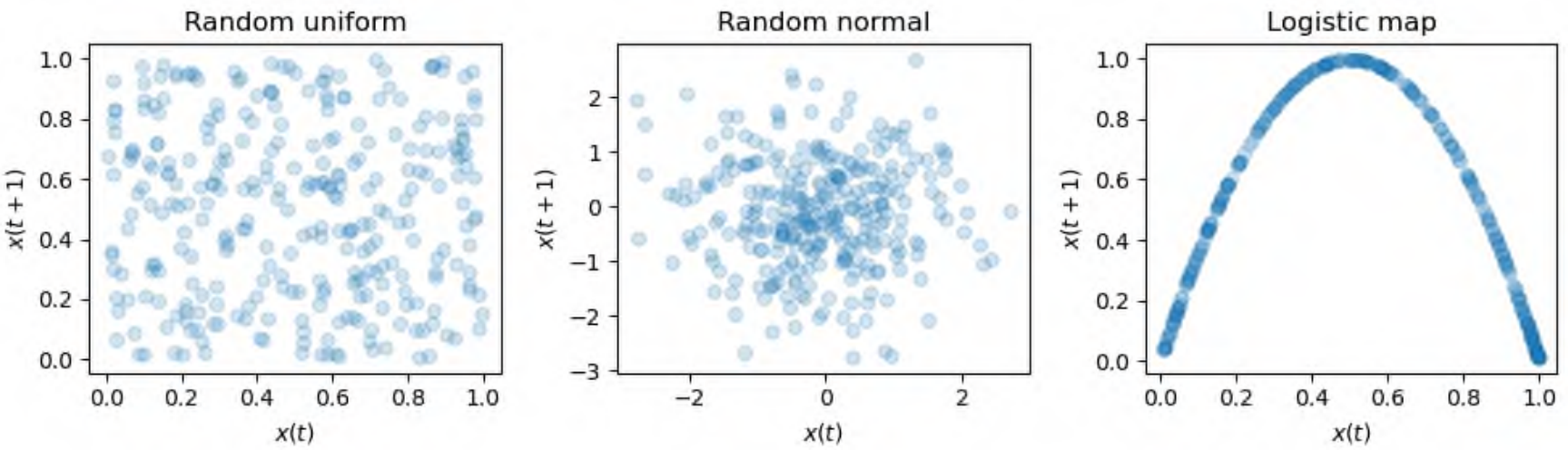
```
fig, axes = plt.subplots(1,3,figsize=(15, 3))
axes[0].plot(uniform_data)
axes[1].plot(normal_data)
axes[2].plot(lgt_data)
plt.tight_layout()
plt.show()
```



- Tools to determine if a system is chaotic or random
  - return map is one of them
- Simply plotting the current value $x(t)$ against the next one $x(t+1)$
  - if random, no or little structure seen as the next value is uncorrelated with the current one
  - if chaotic, a well-defined structure seen

```
def return_map(series, ax, title):
    ax.plot(series[:-1], series[1:], 'o', alpha=0.2)
    ax.set_xlabel('$x(t)$')
    ax.set_ylabel('$x(t+1)$')
    ax.set_title(title)
```

```
fig, axes = plt.subplots(1,3,figsize=(10, 3))
return_map(uniform_data, axes[0], "Random uniform")
return_map(normal_data, axes[1], "Random normal")
return_map(lgt_data, axes[2], "Logistic map")
plt.tight_layout()
plt.show()
```



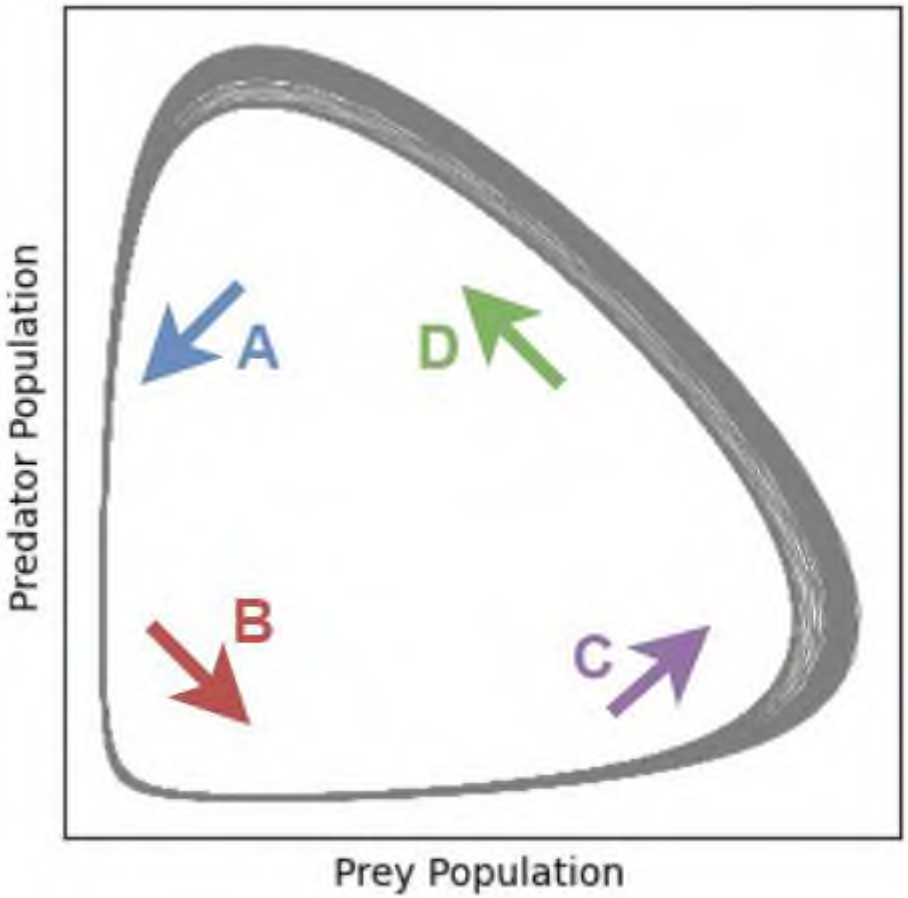
- Logistic map had only one variable: $x(t)$
- In general, we can have systems with two or more variables
- The number of variables defines the dimensionality of the system
- The state of a system is the current values of its variables
    - Most of the time series we saw originated from discrete-time systems that evolve at specific intervals in time
- The time variable $t$ takes values from a discrete set, often *integers*, indicating distinct time steps or periods
- The dynamics of discrete-time systems are described using difference equations, e.g., $x(t+1) = f(x(t))$
- Discrete-time systems are common in DSP, computer algorithms, and any context where observations or changes occur at well-defined intervals
- In continuous-time systems, the time variable $t$ can take any value in a range of real numbers
- The dynamics of continuous-time systems are described using *differential equations* that model how the state variables change with respect to continuous time, e.g., $\frac{dx}{dt} = f(x)$
    - Many physical and natural processes are modeled as continuous-time systems, such as the motion of planets, electrical circuits, and fluid dynamics

- The Lotka-Volterra equations are used to model the predator-prey population system
- It extends the logistic map by modeling the interactions between two species: a predator and its prey
- The growth of the two species is affected by the presence of the other speciemen
- The Logistic map is a discrete-time, univariate model represented by a single equation
- The predator-prey model is a continuous-time, bivariate system
- The Lotka-Volterra equations for predator-prey dynamics are given by:

$$\frac{dx}{dt} = \alpha x - \beta xy$$
$$\frac{dy}{dt} = \delta xy - \gamma y$$

- where x, y is the prey/predator populations, $\frac{dx}{dt}$ and $\frac{dy}{dt}$ are rate of change over $t$
- $\alpha$ is the natural growth rate of prey in the absence of predators
- $\beta$ is the death rate of prey due to predation
- $\delta$ is the efficiency of converting consumed prey into predator offspring
- $\gamma$ is the natural death rate of predators in the absence of food (prey)

- This system can exhibit a variety of dynamics, including stable limit cycles, where the populations of predators and prey oscillate in a regular, periodic fashion
- The attractor in the Lotka-Volterra system, in this case, is typically a closed loop, reflecting the cyclic nature of the predator-prey interactions




**A**: Too few preys.

**B**: Few predators and preys, preys can grow.

**C**: Few predators, lot of preys.

**D**: Too many predators.

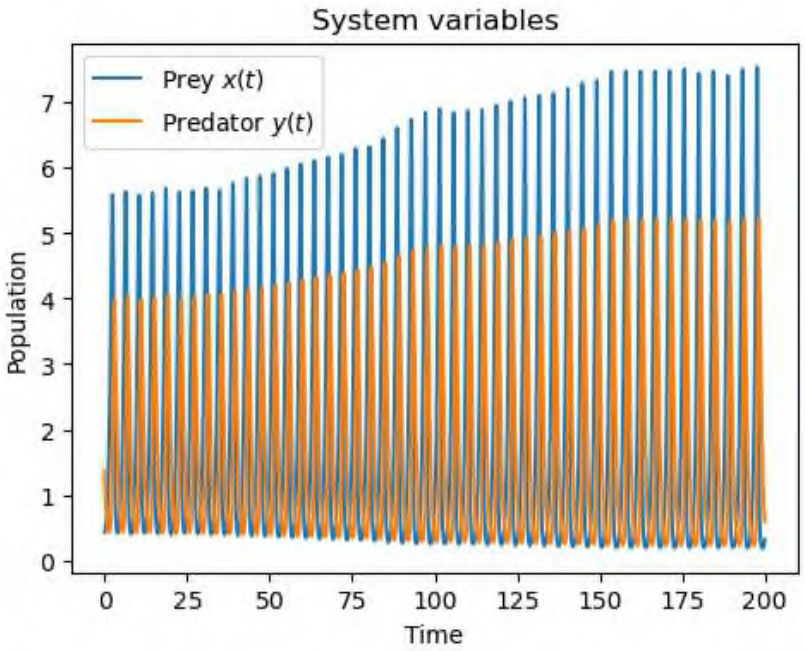
- Define the system of differential equations
- Compute the system values, using numerical solver scipy.integrate.solve_ivp
  - Solver takes a single state variable, so define z = [x, y]
  - Note that the variable *t* is not used in the function we define, but is necessary to the solver

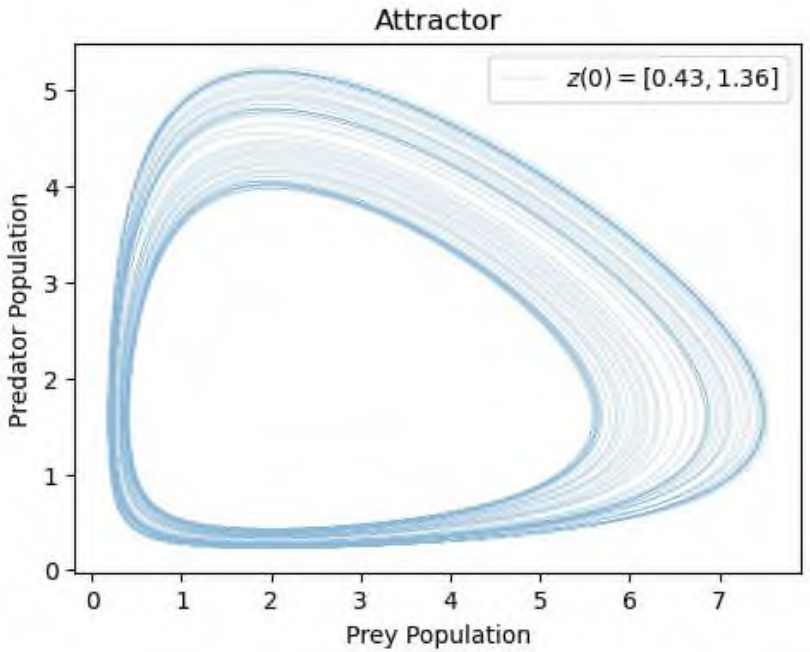
```
# Lotka-Volterra equations
def lotka_volterra(t, z, alpha, beta, delta, gamma):
    x, y = z
    dxdt = alpha * x - beta * x * y
    dydt = delta * x * y - gamma * y
    return [dxdt, dydt]
```

- Use a function `lokta_volterra_attractor` that:
  - Runs the solver computing the sequence of states over time
  - Plots the time series of the state variables and attractor

```
def lokta_volterra_attractor(z0, t, alpha=2, beta=1.25, gamma=1.5, delta=0.75,
                             ax1=None, ax2=None):

    # Solve the differential equations
    sol = solve_ivp(lotka_volterra, [t.min(), t.max()], z0,
        args=(alpha, beta, delta, gamma), t_eval=t)
```

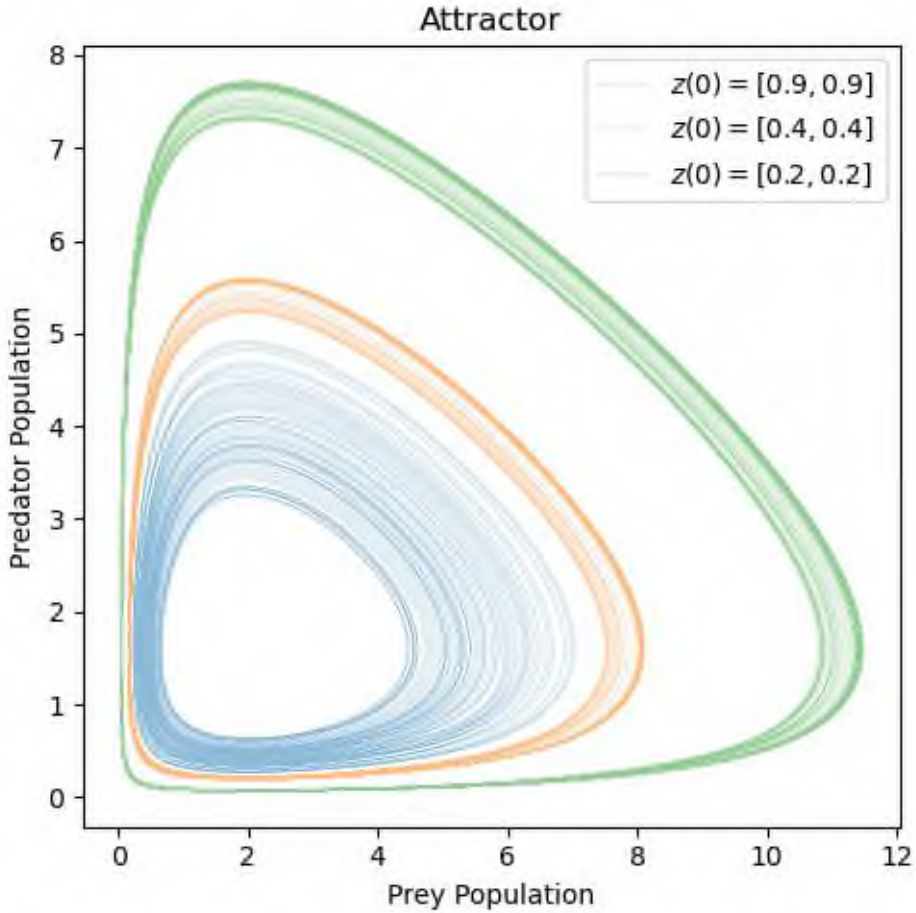
```
t = np.linspace(0, 200, 10000)
_, axes = plt.subplots(1, 2, figsize=(12,4))
z0 = [0.43, 1.36]
lokta_volterra_attractor(z0, t, ax1=axes[0], ax2=axes[1])
```




Πανεπιστήμιο Κύπρου

```
_, ax = plt.subplots(1, 1, figsize=(5, 5))
z0 = [0.9, 0.9]
lokta_volterra_attractor(z0, t, ax2=ax)
z0 = [0.4, 0.4]
lokta_volterra_attractor(z0, t, ax2=ax)
z0 = [0.2, 0.2]
lokta_volterra_attractor(z0, t, ax2=ax)
plt.tight_layout()
plt.show()
```




Πανεπιστήμιο Κύπρου

- The Rössler system is notable for its chaotic behavior, which emerges from a simple set of non-linear ordinary differential equations (ODEs)

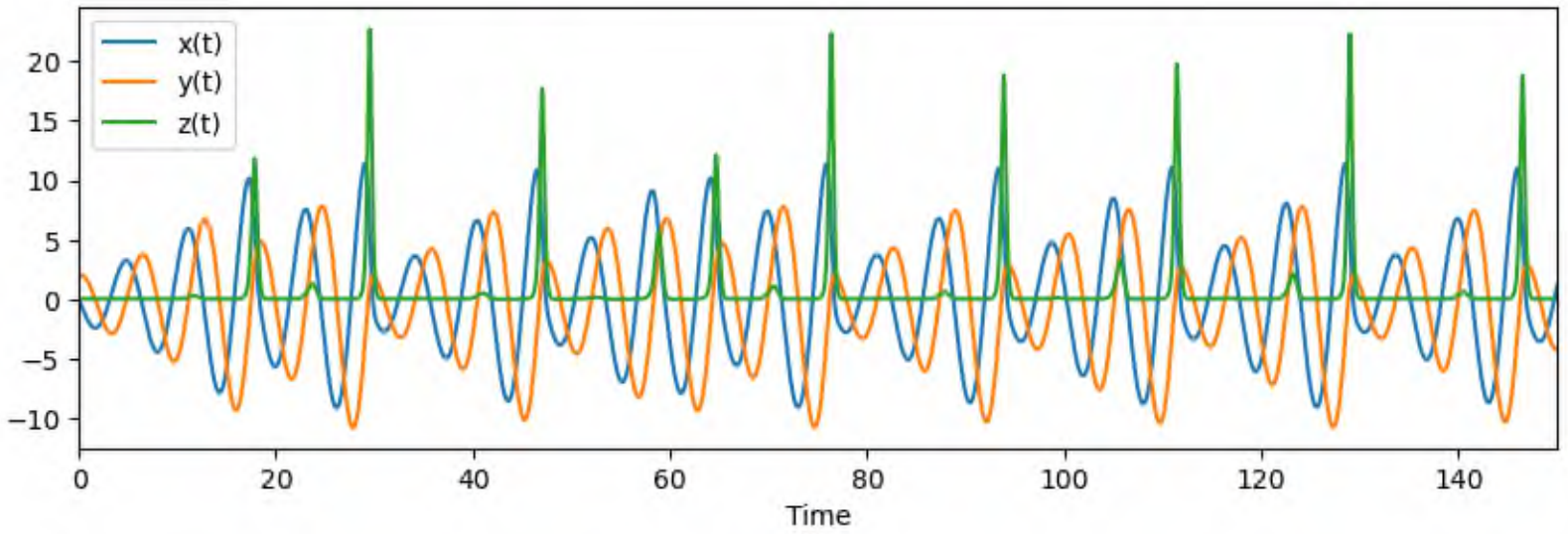
$$\frac{dx}{dt} = -y - z$$
$$\frac{dy}{dt} = x + ay$$
$$\frac{dz}{dt} = b + z(x - c)$$

- x, y and z are the system states over time t, and a, b and c, and are parameters that determine the system's behavior
- Typical values that lead to chaotic behavior are $a = 0.2$, $b = 0.2$, and $c = 5.7$, though chaos can be observed for other values as well

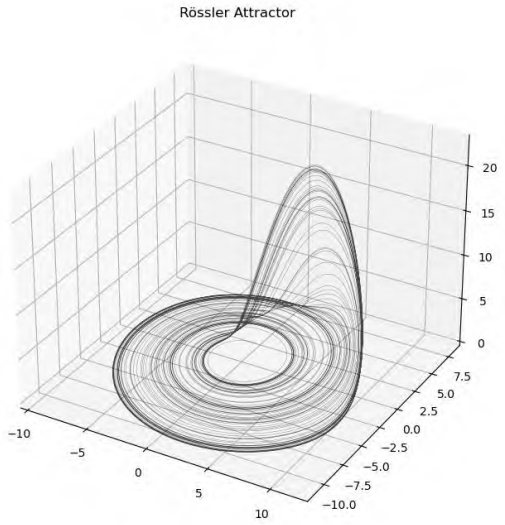
```
# Define the Rössler attractor system of equations
def rossler_system(t, y, a, b, c):
    x, y, z = y
    dxdt = -y - z
    dydt = x + a*y
    dzdt = b + z*(x - c)
    return [dxdt, dydt, dzdt]
```

```
# Parameters
a, b, c = 0.2, 0.2, 5.7
y0 = [0.0, 2.0, 0.0]  # Initial conditions
T = 500 # Final time
t_span = [0, T]  # Time span for the integration

# Solve the differential equations
solution = solve_ivp(rossler_system, t_span, y0, args=(a, b, c),
dense_output=True)
t = np.linspace(0, T, int(5e4))
ross_sol = solution.sol(t)
```

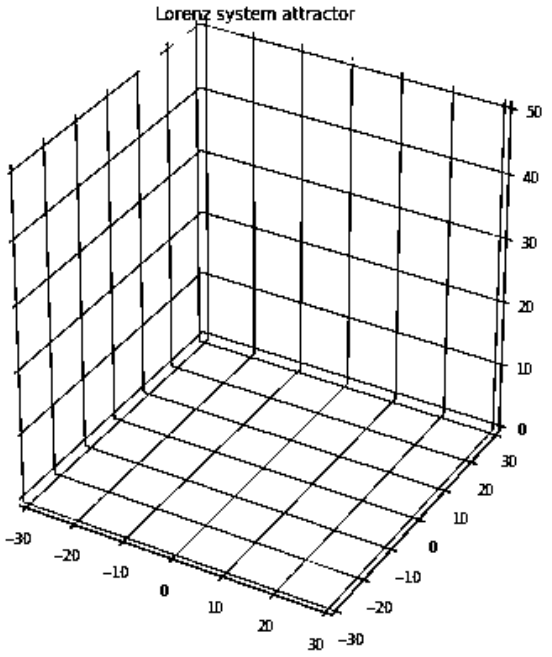
```
# Plot time series
fig, ax = plt.subplots(1,1,figsize=(10, 3)  )
ax.plot(t, ross_sol[0], label="x(t)")
ax.plot(t, ross_sol[1], label="y(t)")
ax.plot(t, ross_sol[2], label="z(t)")
ax.set_xlabel("Time")
ax.set_xlim(0, 150)
plt.legend()
plt.show()
```

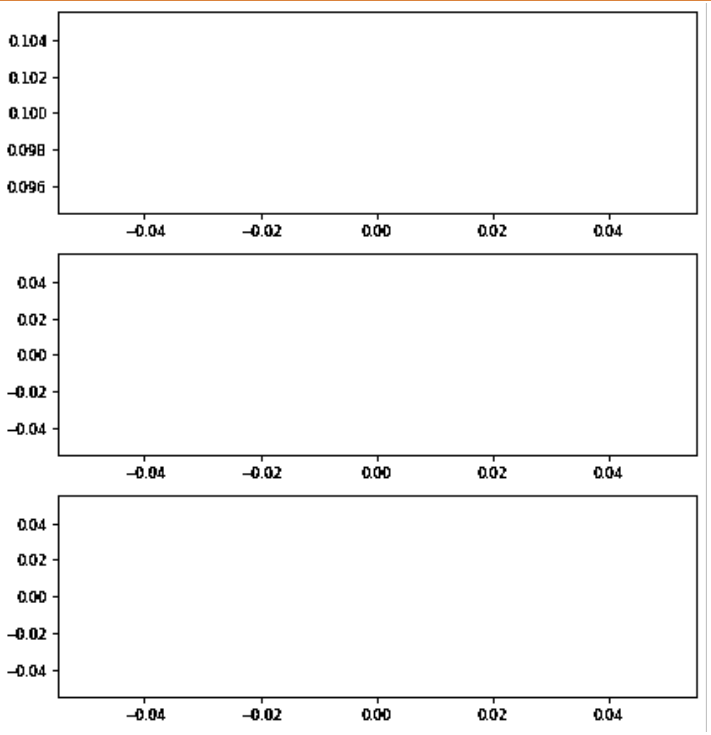


```
def plot_attractor(data, title="", interactive=False):
    xt, yt, zt = data[0], data[1], data[2]
    if interactive:
        fig = go.Figure(data=[go.Scatter3d(x=xt, y=yt, z=zt, mode='lines',
line=dict(color='black', width=1))])
        fig.update_layout(title=title, scene=dict(xaxis_title='x(t)',
yaxis_title='y(t)', zaxis_title='z(t)'),
                          autosize=False, width=800, height=600,
margin=dict(l=0, r=0, b=0, t=0))
        fig.show()
    else:
        fig = plt.figure(figsize=(8, 8))
        ax = fig.add_subplot(111, projection='3d')
        ax.plot(xt, yt, zt, linewidth=0.2, alpha=0.7, color='k')
        ax.set_title(title)
        plt.show()
# Plot the Rössler attractor
plot_attractor(ross_sol, title="Rössler Attractor", interactive=False)
```




Πανεπιστήμιο Κύπρου




- The Lorenz system was one of the first examples to demonstrate the phenomenon of deterministic chaos
- Like the other chaotic systems, this one exhibits sensitive dependence on initial conditions
- Also, it generates apparently random and unpredictable patterns, despite being governed by deterministic laws
- The study of the Lorenz system has had profound implications across numerous fields, from meteorology and climate science to engineering.

Πανεπιστήμιο Κύπρου

- It challenged previous notions of predictability in physical systems and contributed to the development of chaos theory:
  - deepening the understanding of complex systems,
  - highlighting the limitations of current prediction models in systems with chaotic dynamics,
  - inspiring new approaches in the analysis and control of such systems.
- The Lorenz system is defined by the following set of non-linear ODEs:

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

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

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

- Typical values at which the Lorenz system shows a chaotic behavior are $\sigma = 10$, $\rho = 28$, and $\beta = 8/3$

- Previously, we introduced the largest Lyapunov exponent $\lambda$ as a tool for detecting chaos
    - For the Logistic map, we showed that when $\lambda > 0$ the dynamic of the system is chaotic

- In higher dimensional continuous systems, the analysis of Lyapunov exponents remains a crucial method for understanding the system's dynamics

- In particular, they can still be used to determine stability and the onset of chaos

- However, in this setting things become more complicated

- In one-dimensional discrete systems like the Logistic map, there is typically one Lyapunov exponent
- In a continuous system of dimension, there are usually Lyapunov exponents

- Calculating these exponents involves complex numerical methods to handle the evolving tangent space dynamics

- Calculating them is also computationally expensive and sensitive to numerical precision

- Additionally, the initial conditions and parameter values have a stronger influence on the Lyapunov spectrum
  - Positive exponents indicate directions in which the system exhibits sensitive dependence on initial conditions, a characteristic of chaotic dynamics
  - Zero exponents suggest neutral stability along certain directions, often associated with conserved quantities or symmetries in the system
  - Negative exponents reflect directions of convergence, indicating stability in those dimensions
- The largest Lyapunov exponent is still the most important in predicting the overall system behavior, particularly chaos
- However, the entire spectrum can provide insights into more complex dynamics like:
  - mixed modes (simultaneous stable and chaotic behaviors),
  - hyperchaotic systems (systems with more than one positive Lyapunov exponent)

- To compute the Lyapunov spectrum we use the function `computeLE(func, func_jac, x0, t, p)`
- The arguments of the function are:
  - `func`, which specifies the differential equations of the dynamical system,
  - `func_jac`, which is the Jacobian of the system, i.e., the partial derivatives,
  - `x0`, which represents the initial conditions,
  - `t`, the time vector specifying the time steps along which the trajectory of the system is computed,
  - `p`, the parameters of the system.
- For the Lorenz system, `func` is defined as follows

```
def lorenz(t, x, sigma, rho, beta):
    res = np.zeros_like(x)
    res[0] = sigma*(x[1] - x[0])
    res[1] = x[0]*(rho - x[2]) - x[1]
    res[2] = x[0]*x[1] - beta*x[2]
    return res
```

- The Jacobian of the system is the matrix of the partial derivatives
- Consider a general dynamical system with three variables $x, y$ and $z$

$$\begin{cases} \dot{x} = f(x, y, z) \\ \dot{y} = g(x, y, z) \\ \dot{z} = h(x, y, z) \end{cases}$$

- Jacobian is:

$$J = \begin{bmatrix} \frac{\partial f}{\partial x} & \frac{\partial f}{\partial y} & \frac{\partial f}{\partial z} \\ \frac{\partial g}{\partial x} & \frac{\partial g}{\partial y} & \frac{\partial g}{\partial z} \\ \frac{\partial h}{\partial x} & \frac{\partial h}{\partial y} & \frac{\partial h}{\partial z} \end{bmatrix}$$

- For the Lorenz system:

$$J_{Lorens} = \begin{bmatrix} -\sigma & \sigma & 0 \\ \rho - \zeta & -1 & -x \\ y & x & -\beta \end{bmatrix}$$

- Jacobian of the Lorenz system:

```
def lorenz_jac(t, x,  sigma, rho, beta):
    res = np.zeros((x.shape[0], x.shape[0]))
    res[0,0], res[0,1] = -sigma, sigma
    res[1,0], res[1,1], res[1,2] = rho - x[2], -1., -x[0]
    res[2,0], res[2,1], res[2,2] = x[1], x[0], -beta
    return res
```

- Specify the initial conditions $x0$, the time vector $t$, and parameters $p$ $(\sigma, \rho, \beta)$ for the Lorenz system

```
x0 = np.array([1.5, -1.5, 20.])
t = np.arange(0, 1000, 1e-2)
sigma, rho, beta = 10., 28., 8/3
p = (sigma, rho, beta)

LEs, hist = computeLE(lorenz, lorenz_jac, x0, t, p=p)
print(f"Lyapunov Exponents: {LEs}")
```

Lyapunov Exponents: [ 9.06320944e-01 -2.38039941e-03 -1.45705049e+01]

- One of the Lyapunov exponents is positive
- This suggests that the Lorenz system configured with the specific values of exhibits a chaotic behavior along one direction

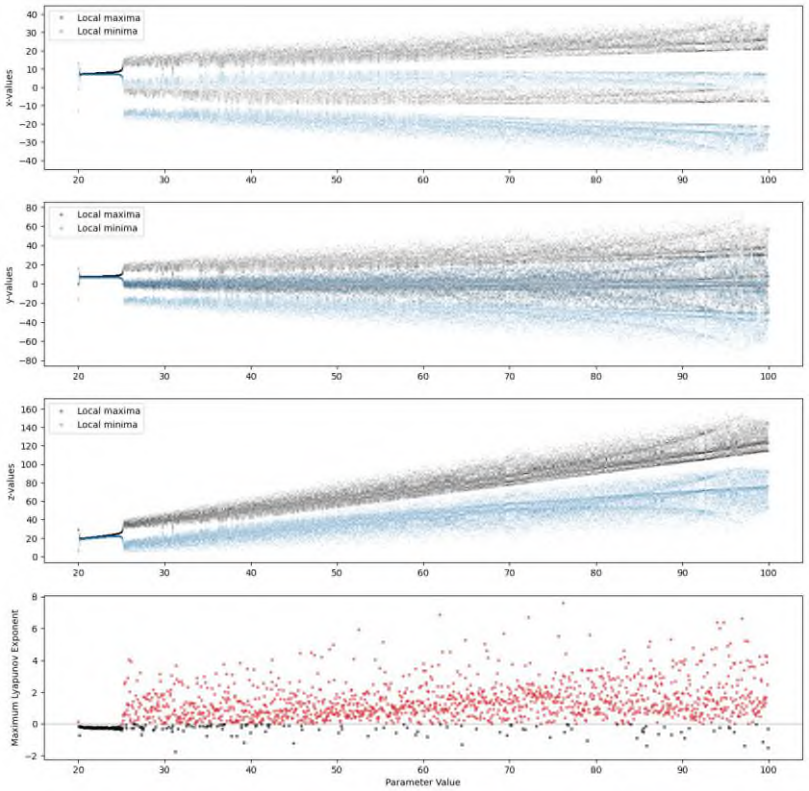
Πανεπιστήμιο Κύπρου

- Bifurcation diagrams also for higher dimensional continuous systems
- With 2 differences compared to unidimensional discrete systems
    1. Since we have more system's parameters, we should change one at the time
    2. System variables assume values in a continuous interval
        - In the continuum we cannot plot all the values assumed by the time series
        - To detect period-doubling we can plot the local maxima and minima of the time series
        - In a contractive regime, the time series will oscillate between a limited number of local minima/maxima
        - At the onset of chaos, the number of local minima/maxima will start to grow

- As for the Lyapunov exponent, we will use a function from the companion code of this course, plot_bifurcation_diagram
- The function takes the following arguments:
    - func, func_jac, x0, time_vector are the same as before
    - parameters is the set of parameters we want to try. Only one parameter should vary, while the others remain constant
    - p_idx is the index of the parameter that changes in parameters.
    - max_time is the number of time steps from time_vector used to estimate the Lyapunov exponents. If left to None all time steps will be used. Since this function takes a lot of time, it's a good idea to set a limit.
- Example to compute the bifurcation diagram for Lorenz system as we vary the parameter $\rho \in [1, 100]$
- The other parameters are kept fixed: $\sigma = 10$ and $\beta = 8/3$

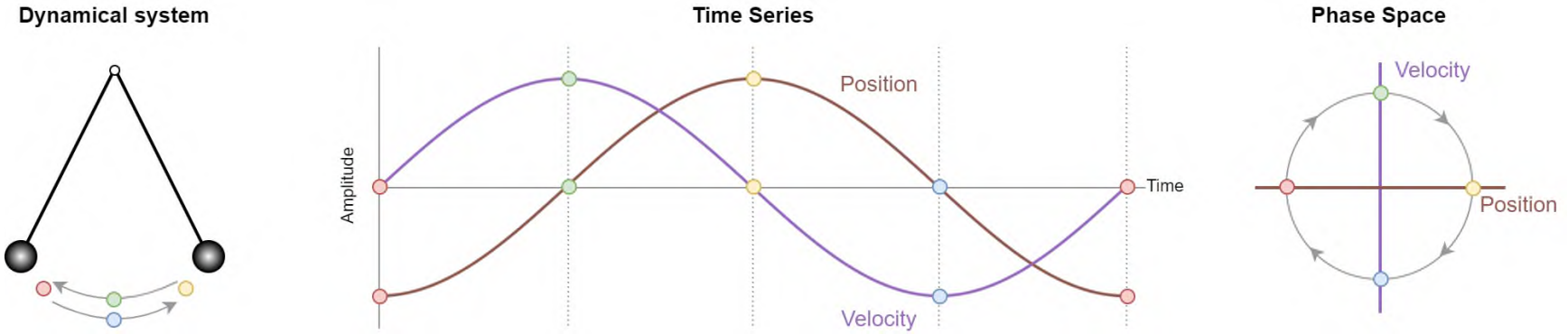
```
# Specify the parameters to try
sigma, beta = 10, 8/3
r_values = np.arange(20, 100, 0.05)
params = np.array([np.tile(sigma, len(r_values)), r_values, np.tile(beta,
len(r_values))]).T

x0 = [1.5, -1.5, 20.]  # Initial conditions
t = np.arange(0, 10, 0.002) # Time vector
```

```
plot_bifurcation_diagram(func=lorenz, func_jac=lorenz_jac,
                         x0=x0, time_vector=t,
                         parameters=params, p_idx=1, max_time=500)
```




Πανεπιστήμιο Κύπρου

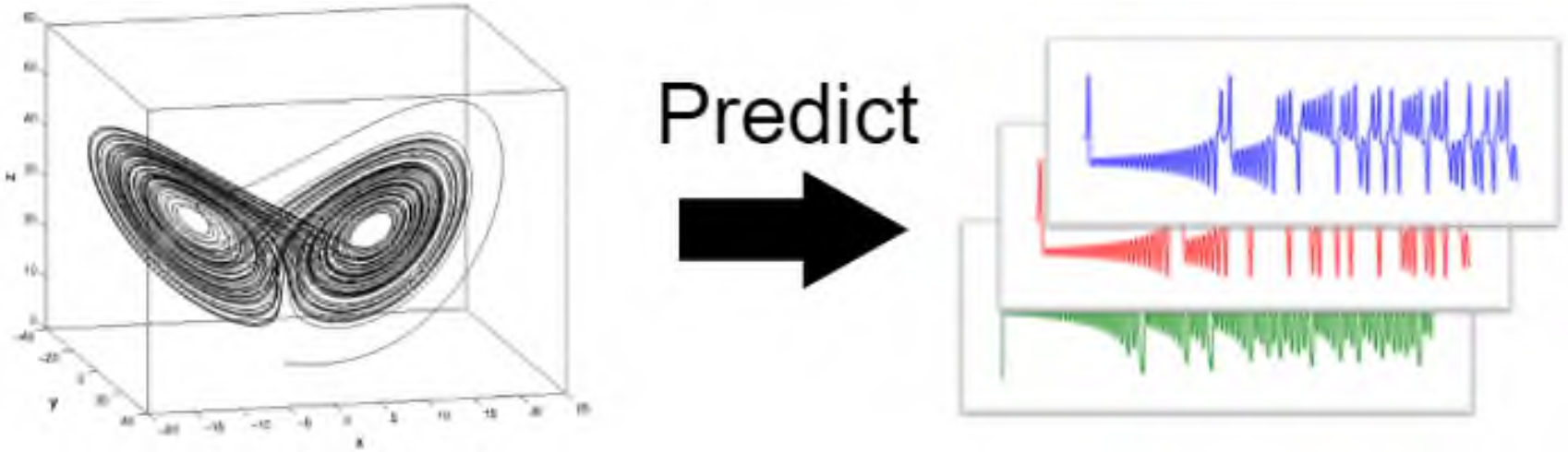
- Current state of a system is treated as a point in the *phase space* or state space
- The phase space represents all the possible states of a system
- It defines how system variables $x, y, z$ interact and evolve




- Phase space gives us complete knowledge about the current state of system
- We are particularly interested its **trajectory** (or orbit) over time
- Knowing it, allows us to make predictions about future states of the system
- Clearly, this of key importance in time series forecasting

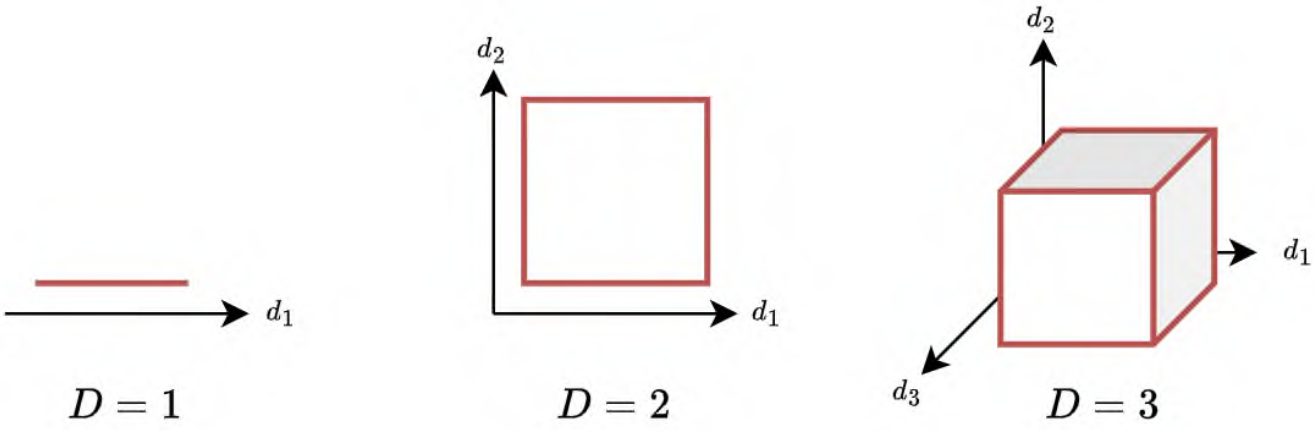
Πανεπιστήμιο Κύπρου

- To identify the trajectory of the system in the phase space, we have to identify properties of its attractor



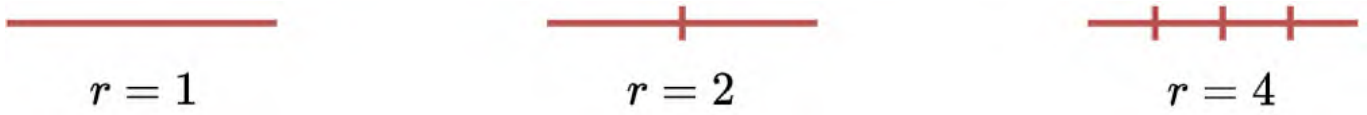

- One of these properties, is its **dimentionality**
- To determine the dimensionality of an attractor, especially strange attractors of chaotic systems, we have to revise our concept of dimensionality

- We commonly understand dimensions from a geometric perspective: length, width, depth
- Common geometrical objects such as a line, a square, or a cube require an increasing number of dimensions to contain them
- The concept of dimensions is also tied to common measures on these objects such as their perimeter, area, and volume, which require an increasing number of dimensions to express them



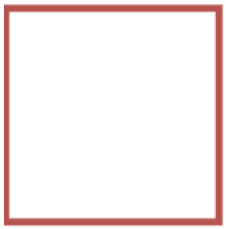

- In this perspective, dimensions are integers and the idea of a fractional dimension, e.g., $d = 1.26$ makes little sense

- Scaling perspective
- Think at dimensions as how much they scale up some quantity
    - For example, let's double the edge of a square
    - The perimeter will grow by a factor of 2, while the area by a factor of 4
    - If we double the edge of a cube its volume gets 8 times bigger
    - Similarly, we can divide an edge into $r$ pieces and counting how many parts we obtain
- Let $r$ be the scaling factor and $\mathrm{N}$ the number of pieces we obtain
- For the line is straightforward:

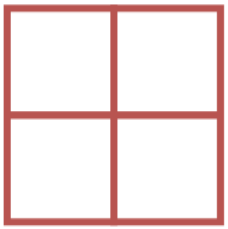



- Easily derive the following rule: $\mathrm{N} = r^D$ with $D = 1$ for the line

- $\mathrm{N} = r^D$ with $D = 2$ for the square



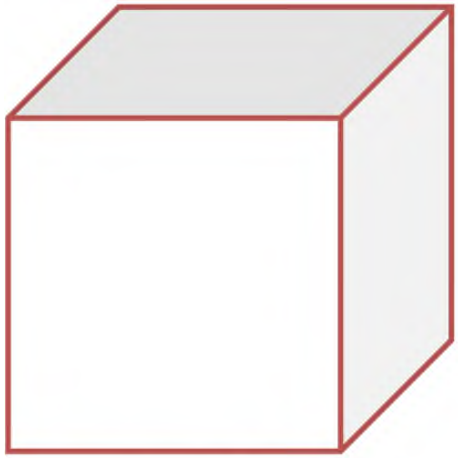
$r = 1$



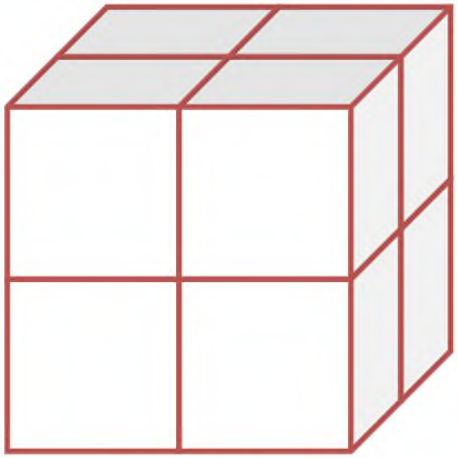
$r = 2$

$r = 4$

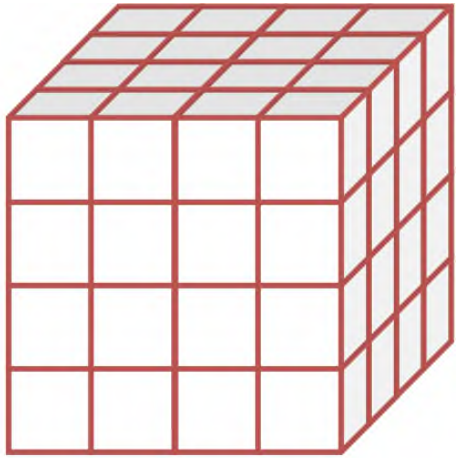
- $\mathrm{N} = r^D$ with $D = 3$ for the cube



$r = 1$

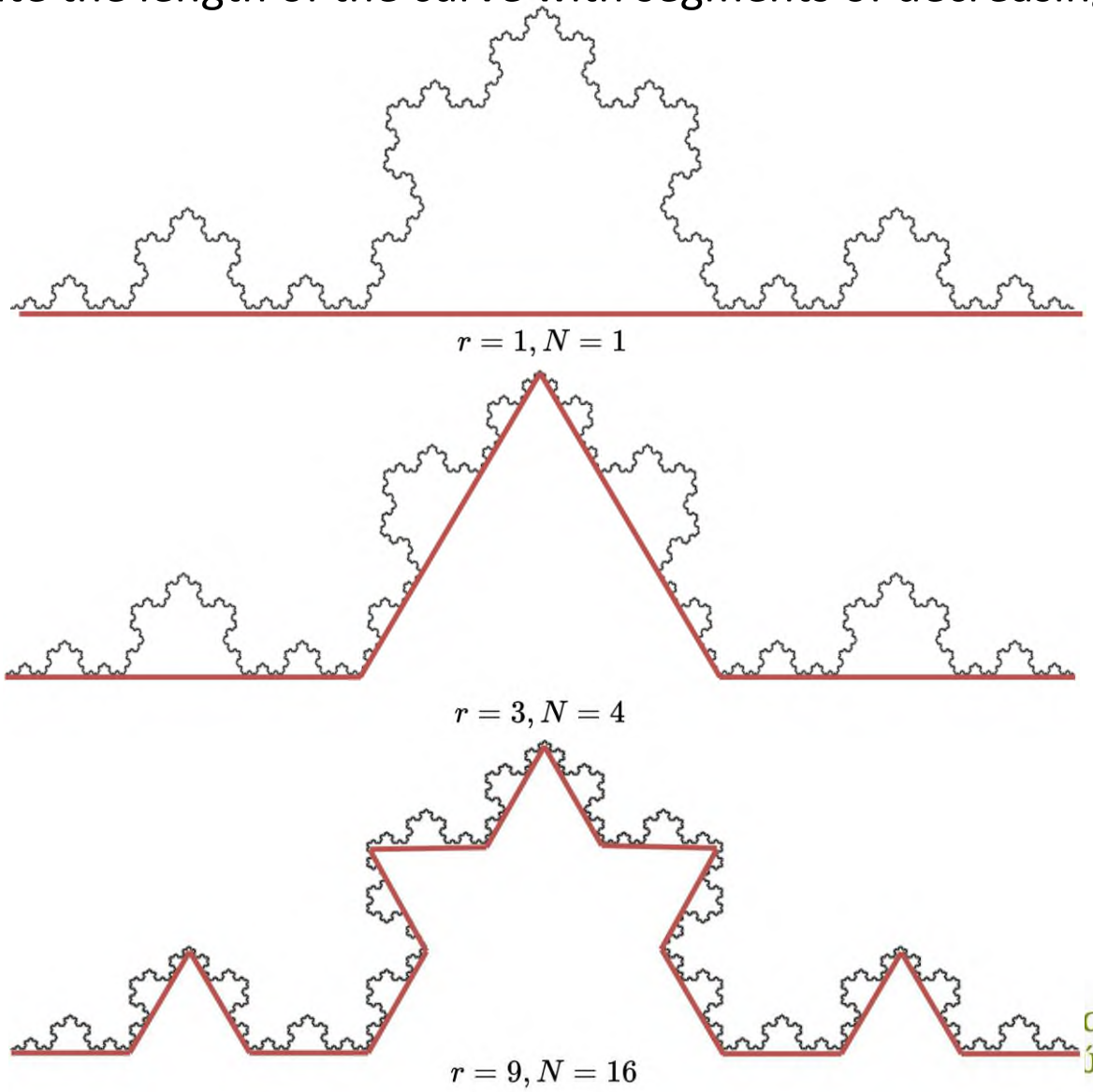


$r = 2$



$r = 4$

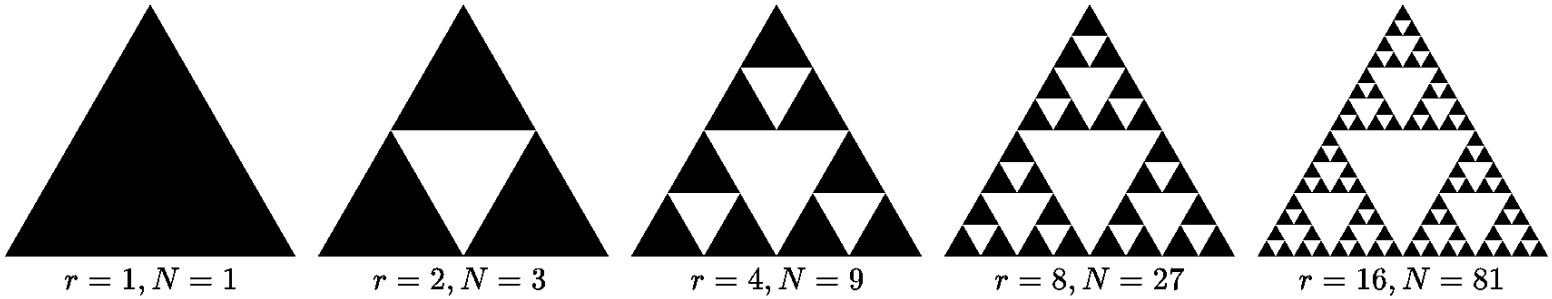
- Fractal object (or Koch curve or snowflake)
- Approximate the length of the curve with segments of decreasing length



- $\mathrm{N} = r^D$ is key to find the dimensionality of the fractal object

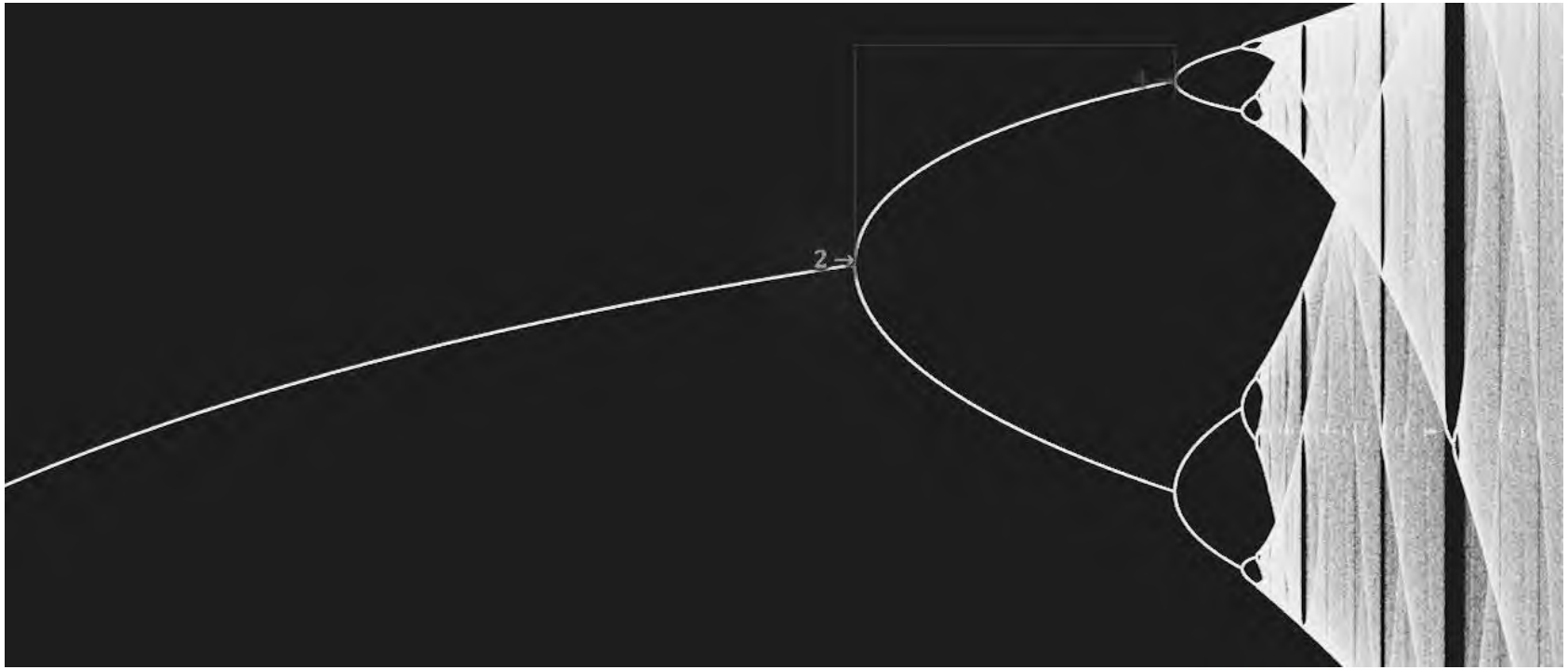
$$4 = 3^D \rightarrow \log(4) = \log(3) \cdot D \rightarrow D = \frac{\log(4)}{\log(3)} \approx 1.26$$

- **Intuition**
  - need a bit more "dimensionality" to contain this object than what we need to contain a line, but less than what we need to contain an area
- Sierpinski triangle example $D = \frac{\log(3)}{\log(2)} \approx 1.58$




- Need more "dimensionality" to contain this object than the Koch curve, but, due to all the "holes", still less than an area

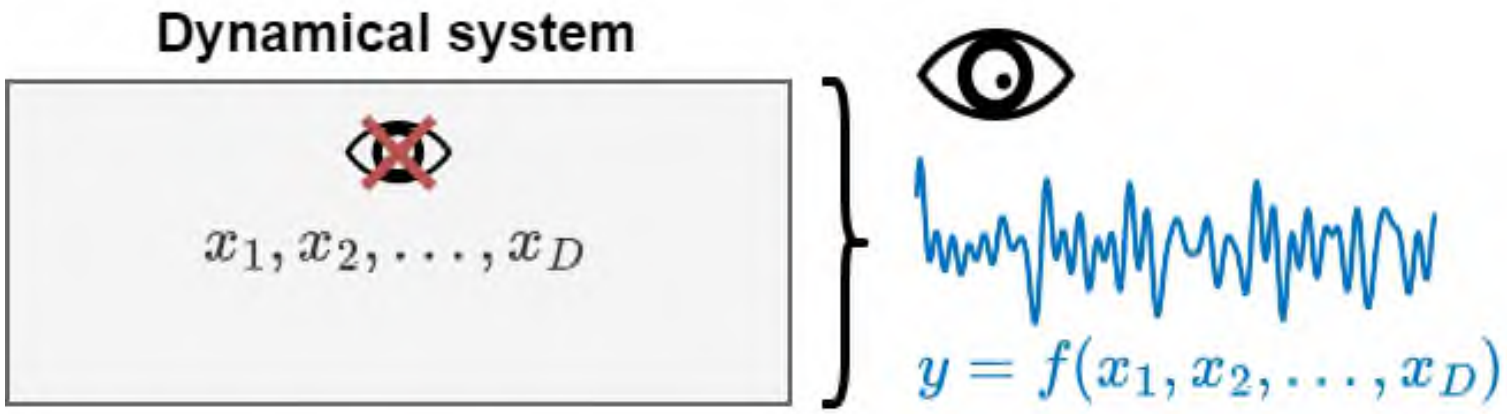
- There is a deep and ubiquitous connection between chaos and fractals
- For example, some bifurcation diagrams are self-similar
- If we zoom-in on the value $r \approx 3.83$ of the bifurcation diagram of the Logistic map, the situation nearby looks like a shrunk and slightly distorted version of the whole diagram
- The same is true for all other non-chaotic points



- Mandelbrot Set and the bifurcation map of the Logistic map

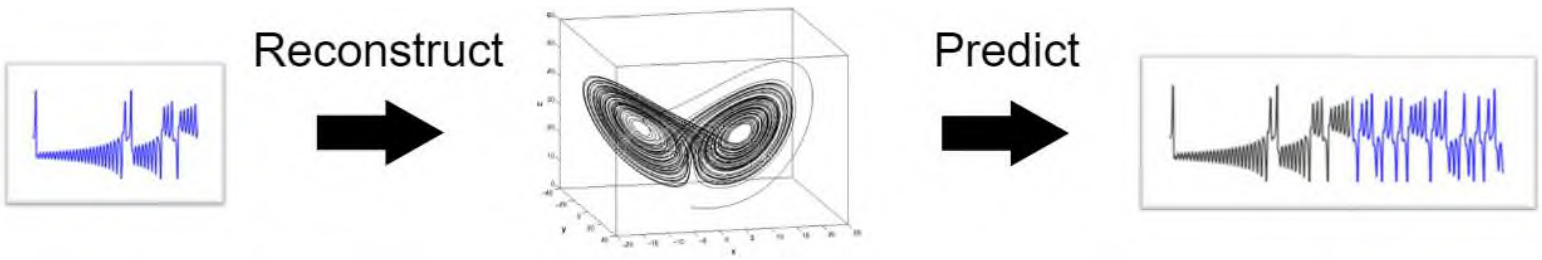
- the trajectory of a strange attractor fills in a fraction of the phase space
- i.e., the dimensionality of the attractor of a chaotic system is fractal
- Determining such dimensionality is key to reconstruct the phase space

- Dynamical systems are often governed by the evolution of several interacting variables $x_1, x_2, \dots, x_D$
- In reality we can only observe (measure) a subset of the system's variables or a function of them $y = f(x_1, x_2, \dots, x_D)$




- Examples
  - In the Lorenz system, we might observe only the variable $y(t)$
  - In weather forecasting, meteorologists often have access only to partial observations of a subset of variables such as temperature, pressure, humidity, wind patterns, and geographic features. However, there are many other variables that are inaccessible such as detailed atmospheric conditions, microclimates, oceans' temperatures, currents, the presence of pollutants, etc.

- Can we reconstruct the trajectory of the system in the state space from partial observations?
- If we are able to do so, we can predict the future states of the systems, including the observed time series




- Takens' Embedding Theorem
    - Assume a dynamical system with an unknown or partially known state space
    - Takens' Embedding Theorem says that it is possible to reconstruct the dynamics of the whole system using a series of observations from a single variable
- This means that even if we cannot observe the entire state of a system directly, we can still understand its dynamics through proper analysis of a single observable variable

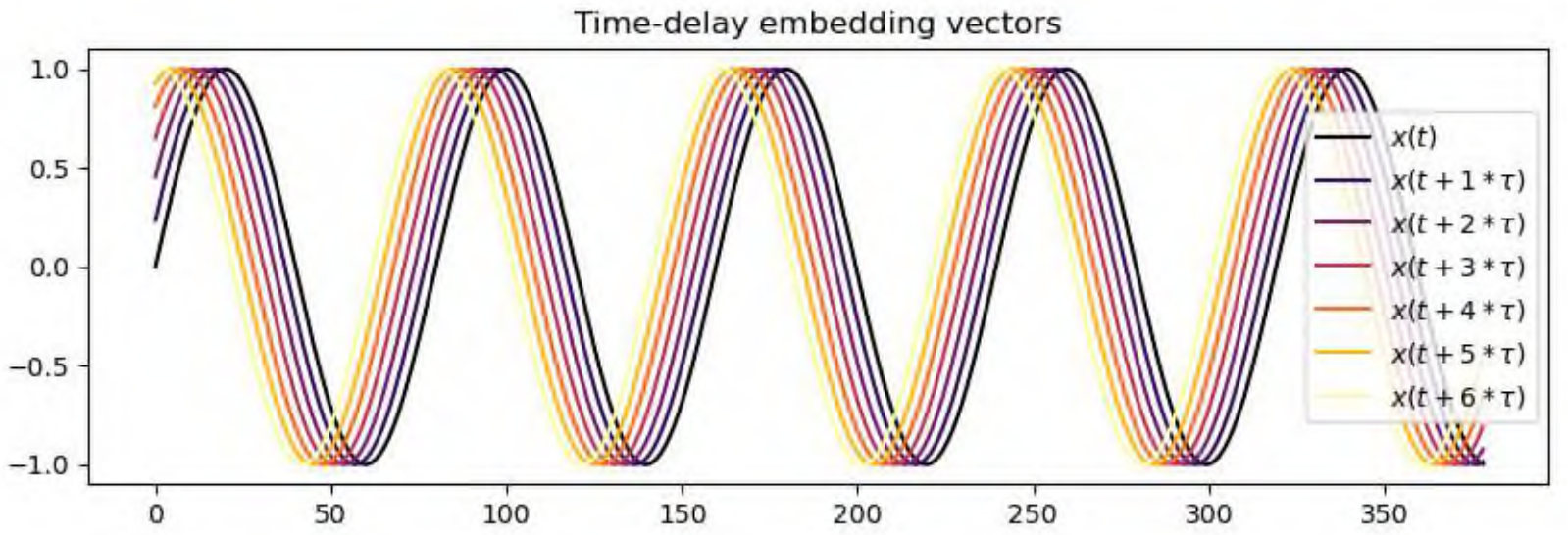
- Time-delay embedding vector
  - Let $x(t)$ be the observed time series
  - Let embedding vector $e(t) = [e_1, e_2, \dots, e_N]$ defined as:

$$e_1(t) = x(t)$$
$$e_2(t) = x(t + \tau)$$
$$e_3(t) = x(t + 2\tau)$$
$$\dots$$
$$e_m(t) = x(t + (m - 1)\tau)$$

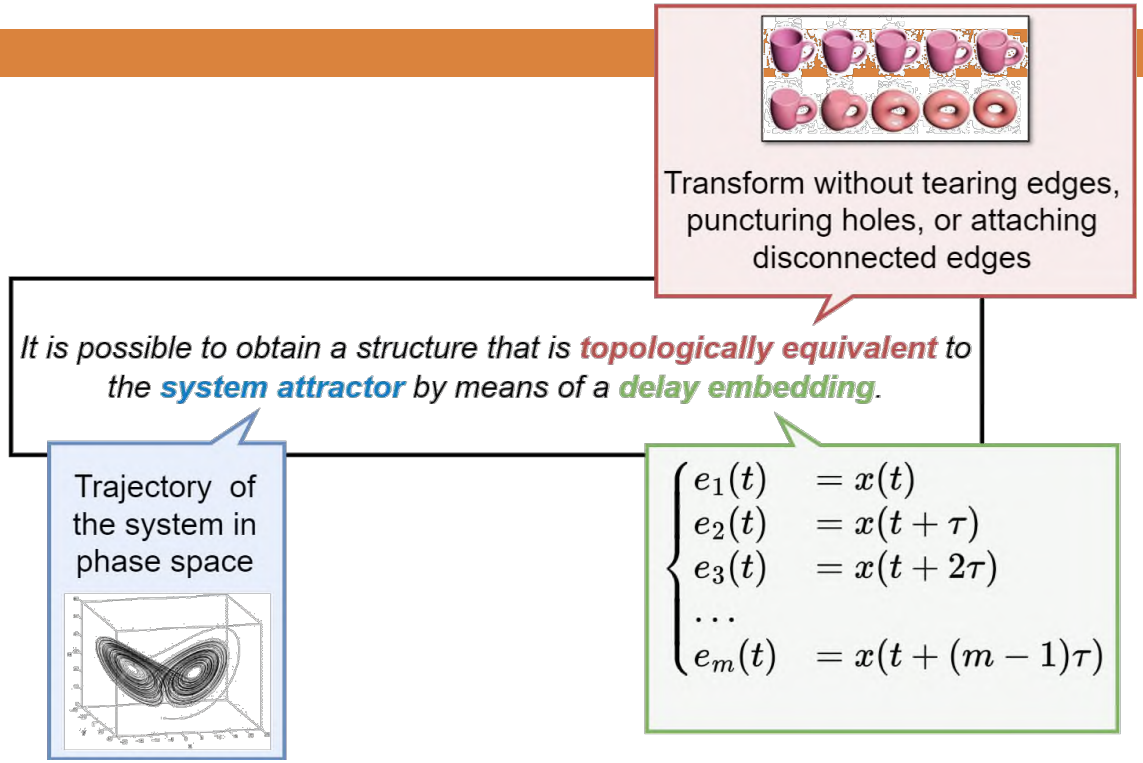
  - Here, $\tau$ is a chosen time delay and $m$ is the embedding dimension

```
def takens_embedding(data, delay, dimension):
    embedding = np.array([data[0:len(data)-delay*dimension]])
    for i in range(1, dimension):
        embedding = np.append(embedding, [data[i*delay:len(data) -
delay*(dimension - i)]], axis=0)
    return embedding.transpose()
```

```
t = np.linspace(0, 10*np.pi, 400)
x = np.sin(t)                          # time series
tau = 3                                # time delay
N = 7                                  # embedding dimension
emb = takens_embedding(x, tau, N)      # compute embeddings

cmap = plt.get_cmap('inferno')
fig, ax = plt.subplots(1,1, figsize=(10,3))
for i in range(N):
    _label = f"$x(t + {i}*\\tau )$" if i > 0 else f'$x(t)$'
    ax.plot(emb[:,i], label=_label, color=cmap(i / (N - 1)))
plt.legend()
plt.title('Time-delay embedding vectors')
plt.show()
```




Πανεπιστήμιο Κύπρου




- Takens' theorem does not specify how to choose $m$ and $\tau$
- It only asserts that for a system with an attractor of (fractional) dimension
- $D$, an embedding dimension $m > 2D$ is sufficient to ensure a diffeomorphic (topologically equivalent) embedding under generic conditions
- Intuitively, the embedding dimension must be large enough to unfold the attractor fully in the reconstructed phase space, capturing its dynamics without self-intersections

Estimating time delay, $\tau$

- We can rely on **mutual information** (MI) to compute $\tau$

- MI quantifies the amount of information between two random variables
  - Here computing MI between time series and its lagged version
  - MI like a powerful autocorrelation that captures also non-linear relationships

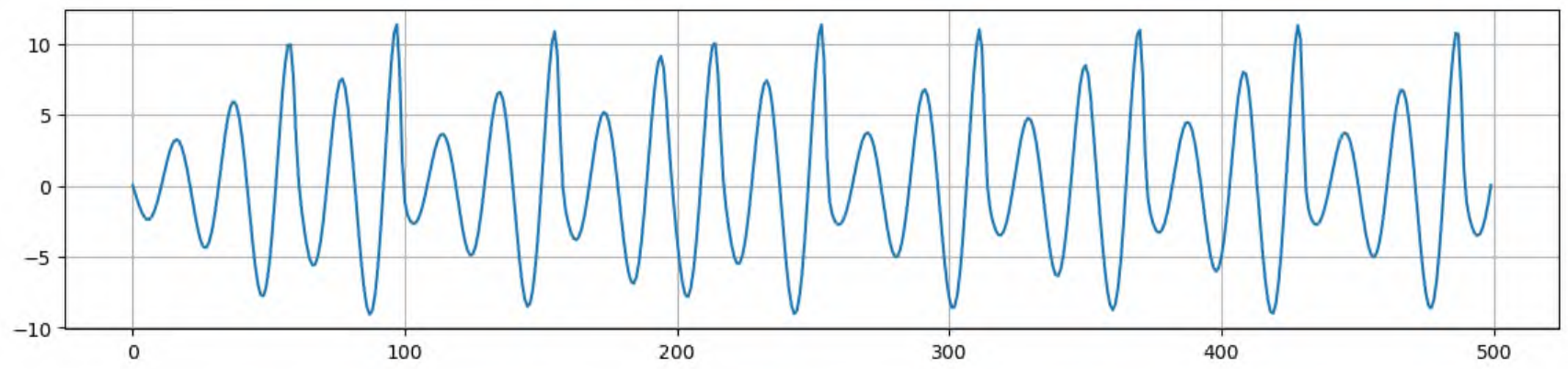
- When MI reaches its minimum for a certain lag $\tau$, the observations are sufficiently independent while still retaining meaningful information about the dynamics of the system

1. Compute the minimum $x_{min}$ and maximum $x_{max}$ of the time series
2. Split the interval $[x_{min}\ x_{max}]$ into $n_{bins}$ bins
3. Denote $p_t(h)$ the probability that an element of $x(t)$ is in the $h$-bin
4. Denote $p_{t+\tau}(k)$ the probability that an element of $x(t+\tau)$ is in the $k$-th bin
5. Denote the probability $p_{t,t+\tau}(h,k)$ that $x(t)$ is in the $h$-th bin, while $x(t+\tau)$ is in the $k$-th bin
6. Define the MI as:

$$I(\tau) = -\sum_{h=1}^{n_{bins}} \sum_{k=1}^{n_{bins}} p_{t,t+\tau}(h,k)\ \log \frac{p_{t,t+\tau}(h,k)}{p_t(h) \cdot p_{t+\tau}(k)}$$

7. Compute $\tau^* = argmin_\tau I(\tau)$

```
# Parameters
a, b, c = 0.2, 0.2, 5.7
y0 = [0.0, 2.0, 0.0]  # Initial conditions
T = 1500 # Final time
t_span = [0, T]  # Time span for the integration
# Solve the differential equations
solution = solve_ivp(rossler_system, t_span, y0, args=(a, b, c),
dense_output=True)
t = np.linspace(0, T, int(5e3))
ross_sol = solution.sol(t)

ross_ts = ross_sol[0]
plt.figure(figsize=(14,3))
plt.plot(ross_ts[:500])
plt.grid()
plt.show()
```

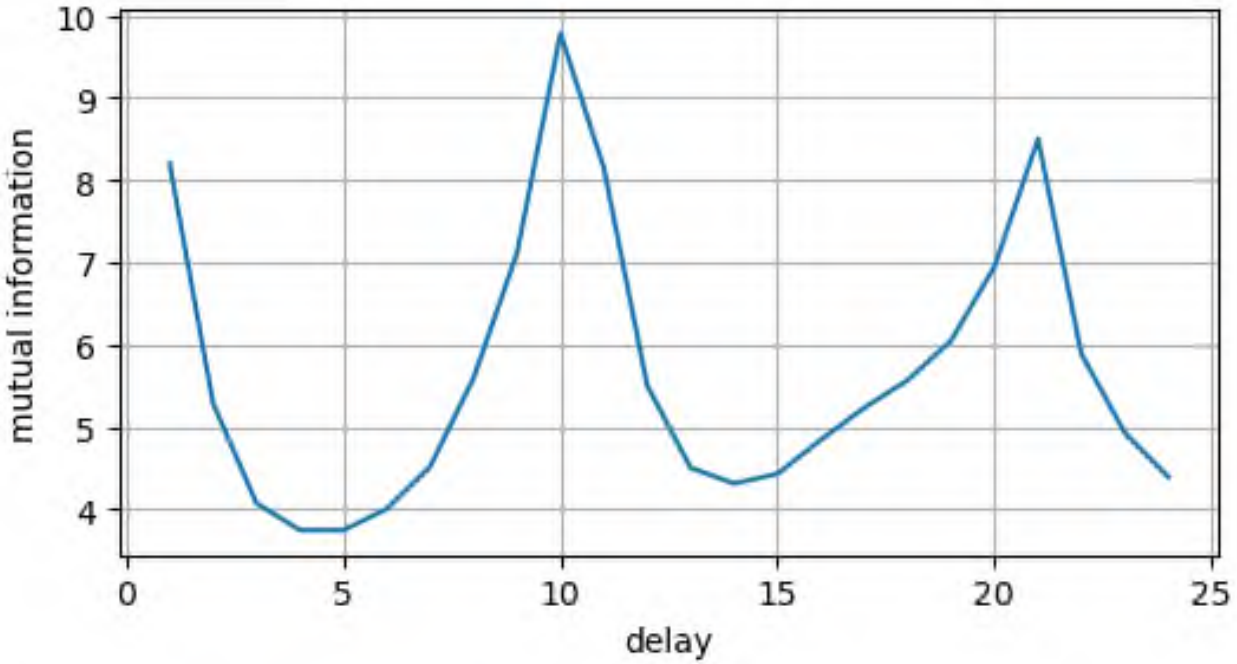



Πανεπιστήμιο Κύπρου

```
def mutual_information(data, delay, n_bins):
    """
    Calculate the mutual information for a given delay using histograms.
    """
    # Prepare delayed data
    delayed_data = data[delay:]
    original_data = data[:-delay]
    # Compute histograms
    p_x, bin_edges = np.histogram(original_data, bins=n_bins, density=True)
    p_y, _ = np.histogram(delayed_data, bins=bin_edges, density=True)
    p_xy, _, _ = np.histogram2d(original_data, delayed_data, bins=bin_edges,
density=True)
    # Calculate mutual information
    mutual_info = 0
    for i in range(n_bins):
        for j in range(n_bins):
            if p_xy[i, j] > 0 and p_x[i] > 0 and p_y[j] > 0:
                mutual_info += p_xy[i, j] * np.log(p_xy[i, j] / (p_x[i] *
p_y[j]))
    return mutual_info
```

- Calculate MI for different values of τ, display it, and find its first minimum (optimal embedding delay)

```
MI = []
for i in range(1,25):
    MI = np.append(MI,[mutual_information(ross_ts,i,50)])
plt.figure(figsize=(6,3))
plt.plot(range(1,25), MI)
plt.xlabel('delay') plt.ylabel('mutual information')
plt.grid() plt.show()
```




- Optimal value τ=4 or 5
- Things change quite a lot if data are sampled with different frequencies
- In this example we used t = np.linspace(0, 1500, int(5e3)).
- Try to see how things change for t = np.linspace(0, 500, int(5e4)) the time vector we used before to draw the attractor in the Rössler system
- The result is also influenced by the value of n_bins
- Choosing these kind of hyperparameters is often a sensitive choice in estimators for information theoretical quantities such as the MI

Estimating the embedding dimension $m$

- We can estimate $m$ by using a measure called false nearest neighbours (FNN)

- The false neighbors are points that appear close in lower dimensions due to projection, but are not actually close in the higher-dimensional space

- The main idea behind FNN is to increase the embedding dimension until the fraction of false neighbors falls below a certain threshold

1. Start with a low embedding dimension $m = 1$
2. For each point in the reconstructed phase space, identify its nearest neighbor
3. Calculate the distance between each point and its nearest neighbor in the current embedding dimension $m$ and then in the next higher dimension $m + 1$
4. Determine if the neighbor is 'false' by checking if the distance between the point and its nearest neighbor increases significantly when moving from dimension $m$ to $m + 1$. A neighbor is considered false if:

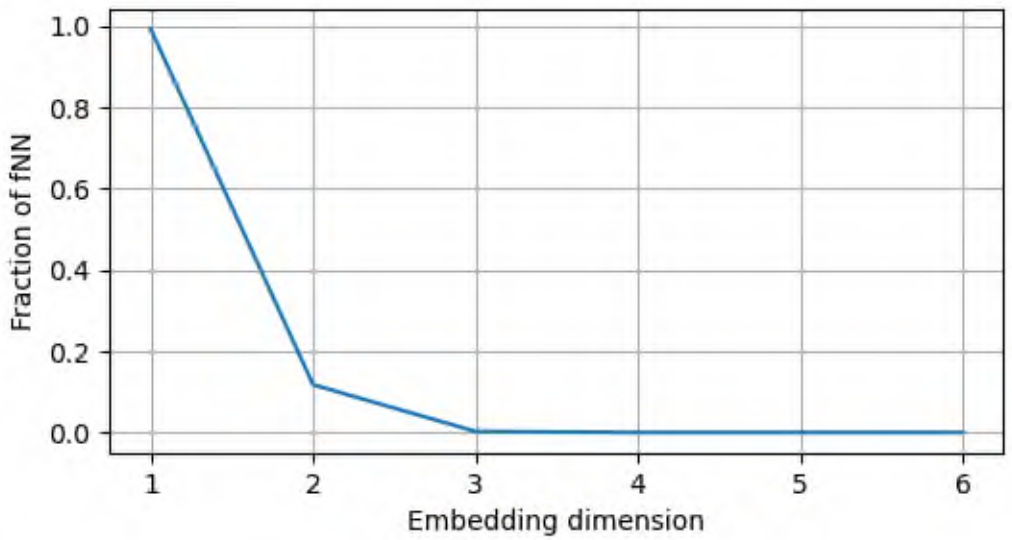
$$\frac{|R_{m+1} - R_m|}{R_m} > R_{tol}$$

- where $R_m$ is the distance between the point and its nearest neighbor in dimension $m$
- $R_{total}$ threshold for deciding if the increase is significant

5. Compute the fraction of false nearest neighbors for the current dimension
6. Repeat the process by increasing $m$ until the fraction of false nearest neighbors a sufficiently low value indicating that the attractor is completely unfolded

```
def calculate_fnn(data, delay, max_emb_dim, R_tol=10):
    fnn_proportions = []
    for m in range(1, max_emb_dim + 1):
        # Compute embeddings in m and m+1
        emb_m = takens_embedding(data, delay, m)
        emb_m_plus_one = takens_embedding(data, delay, m + 1)
        # Compute the nearest neighbors in m
        nbrs = NearestNeighbors(n_neighbors=2).fit(emb_m[:-delay])
        distances, indices = nbrs.kneighbors(emb_m[:-delay])
        n_false_nn = 0
        for i in range(0, len(distances)):
            # Nearest neighbor of i in m and distance from it
            neighbor_index, R_m = indices[i, 1], distances[i, 1]
            # Dinstance in m+1 from the nearest neighbor in m
            R_m_plus_one = np.linalg.norm(emb_m_plus_one[i] -
emb_m_plus_one[neighbor_index])
            # fNN formula
            if abs(R_m_plus_one - R_m) / R_m > R_tol:
                n_false_nn += 1
        fnn_proportion = n_false_nn / len(indices)
        fnn_proportions.append(fnn_proportion)
    return fnn_proportions
```

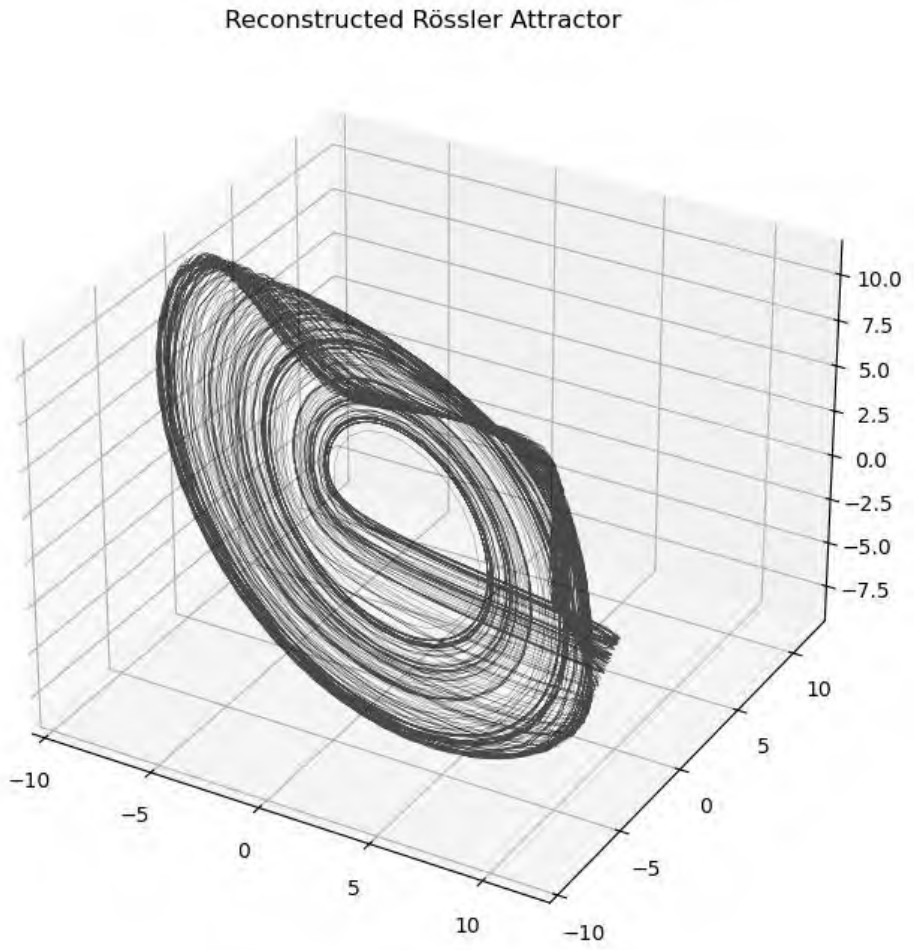
```
nFNN = calculate_fnn(ross_ts, delay=5, max_emb_dim=6)

plt.figure(figsize=(6,3))
plt.plot(range(1,len(nFNN)+1),nFNN);
plt.xlabel('Embedding dimension')
plt.ylabel('Fraction of fNN')
plt.grid()
plt.show()
```




- The fraction of fNN drops to zero for $m = 3$
- This makes sense, since we know that the Rössler system has 3 state variables and its attractor is contained in more than 2 dimensions

- Reconstructing the attractor

```
emb = takens_embedding(ross_ts, delay=5, dimension=3)
plot_attractor(emb.T, title="Reconstructed Rössler Attractor",
interactive=False)
```




- The reconstruction closely resemble the original attractor
- Remember that the reconstructed attractor needs not to be equal to the actual attractor but only diffeomorphic (topologically equivalent)

- Regression on the Taken's embeddings
  - Taken's embeddings are related to the windowed approach for forecasting
- Example of the relationship
  - Given a time series $x(t)$, a time delay $\tau$, and an embedding dimension $m$, the matrix $\mathrm{E}$ of Takens' embeddings can be represented as:

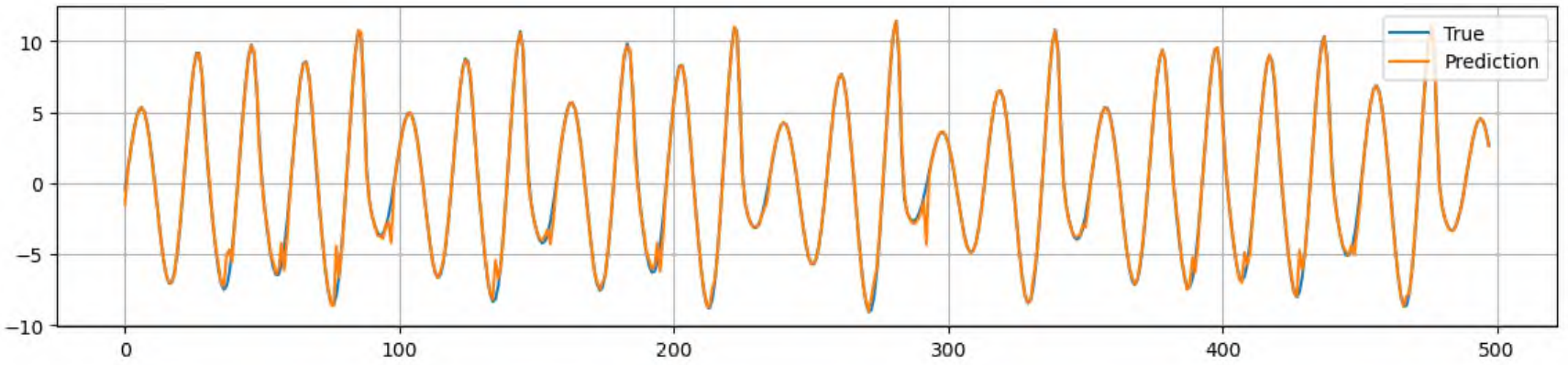
$$\mathrm{E} = \begin{pmatrix} x(t) & x(t+\tau) & x(t+2\tau) & \dots & x(t-(m-1)\tau) \\ x(t+1) & x(t+\tau+1) & x(t+2\tau+1) & \dots & x(t-(m-1)\tau+1) \\ x(t+2) & x(t+\tau+2) & x(t+2\tau+2) & \dots & x(t-(m-1)\tau+2) \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ x(T-(m-1)\tau) & x(T-(m-2)\tau) & x(T-(m-3)\tau) & \dots & x(T) \end{pmatrix}$$

- Each row should contain enough information to describe the dynamics of the system at a certain time step
- Use this information to make future predictions
- For example, we can use the first row to predict the next value $x(t+m\tau)$

1. Compute the embeddings using as dimension $m + 1$
2. Then, use as input all the columns of $\mathrm{E}$ except the last one, emb[:,:-1], which will be the target

- This corresponds to taking a window of size $m$ samples taken every $\tau$ time steps and making a prediction $\tau$ steps ahead

```
def forecast_on_phase_space(y, delay, dimension, test_prop):
    # Compute embeddings
    emb = takens_embedding(y, delay=delay, dimension=dimension)
    # Create input and target
    X = emb[:, :-1]
    y = emb[:, -1]
    # Divide into train and test
    test_size = int(len(y)*test_prop)
    X_train = X[:-test_size, :]
    y_train = y[:-test_size]
    X_test = X[-test_size:, :]
    y_test = y[-test_size:]
    # Fit the regressor on the training data
    rf = RandomForestRegressor().fit(X_train, y_train)
    # Predict the test data
    preds = rf.predict(X_test)
    print(f'MSE: {mean_squared_error(y_test, preds):.3f}')
    plt.figure(figsize=(14,3))
    plt.plot(y_test, label='True') plt.plot(preds, label='Prediction')
    plt.grid() plt.legend() plt.show()
```




MSE: 0.215

Πανεπιστήμιο Κύπρου

- Instead of time embeddings, use states of the Reservoir of an Echo State Network as input for our regression model
- Perform the same prediction $\tau$-steps ahead

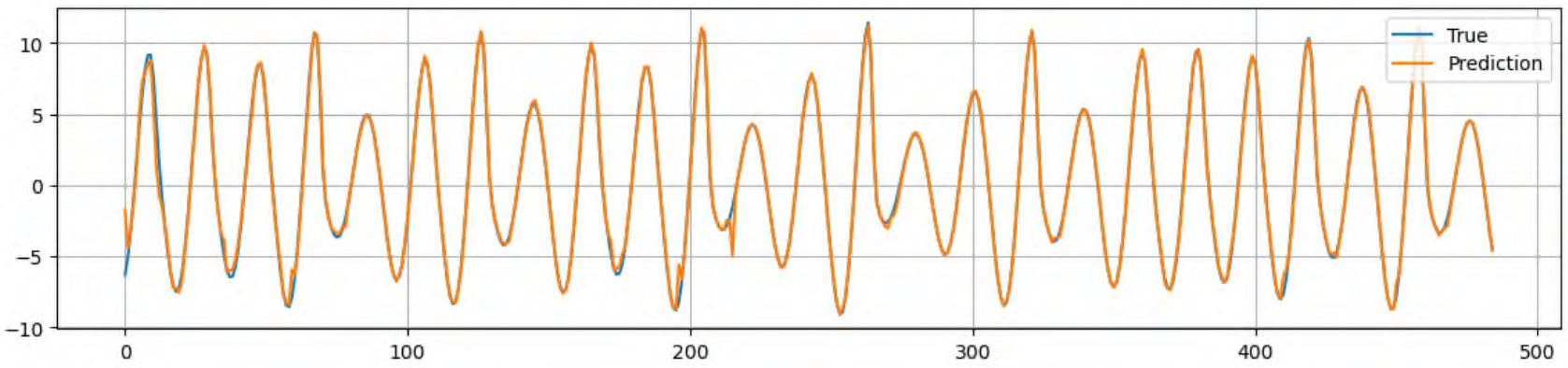
```
Xtr, Ytr, Xte, Yte, scaler = make_forecasting_dataset(ross_ts[:,None],
                                                      horizon=5,
                                                      test_percent = 0.1)
print(f"Xtr shape: {Xtr.shape}\nYtr shape: {Ytr.shape}\nXte shape:
{Xte.shape}\nYte shape: {Yte.shape}")
```

```
# initialize the Reservoir and compute the states
res= Reservoir(n_internal_units=900,
               spectral_radius=0.99,
               input_scaling=0.1,
               connectivity=0.25)

n_drop=10
states_tr = res.get_states(Xtr[None,:,:], n_drop=n_drop, bidir=False)
states_te = res.get_states(Xte[None,:,:], n_drop=n_drop, bidir=False)
print(f"states_tr shape: {states_tr.shape}\nstates_te shape: {states_te.shape}")
```

```
# reduce the dimensionality of the Reservoir states with PCA
pca = PCA(n_components=3)
states_tr_pca = pca.fit_transform(states_tr[0])
states_te_pca = pca.transform(states_te[0])
print(f"states_tr shape: {states_tr_pca.shape}\nstates_te shape:
{states_te_pca.shape}")
```

```
# Fit the regression model
rf = RandomForestRegressor().fit(states_tr_pca, Ytr[n_drop:,:].ravel())

# Compute the predictions
Yhat_pca = rf.predict(states_te_pca)[...,None]

# Compute the mean squared error
mse = mean_squared_error(scaler.inverse_transform(Yhat_pca), Yte[n_drop:,:])
print(f"MSE: {mse:.2f}")
```

MSE: 0.18




- ESN is particular good in predicting non-linear time series and chaotic systems in general
- Reservoir manages very well to reproduce the dynamics in the underlying system

Πανεπιστήμιο Κύπρου