Chaos, Solitons and Fractals 201 (2025) 117269

Contents lists available at ScienceDirect

Chaos, Solitons and Fractals

journal homepage: www.elsevier.com/locate/chaos

# pynamicalsys: A Python toolkit for the analysis of dynamical systems

Matheus Rolim Sales [a,b], Leonardo Costa de Souza [c,d], Daniel Borin [b,e], Michele Mugnaine [c,f], José Danilo Szezech [c,g,h], Ricardo Luiz Viana [i], Iberê Luiz Caldas [c], Edson Denis Leonel [b], Chris G. Antonopoulos [a,*]

[a] University of Essex, School of Mathematics, Statistics and Actuarial Science, Wivenhoe Park, Colchester, CO4 3SQ, United Kingdom
[b] São Paulo State University (UNESP), Institute of Geosciences and Exact Sciences, 13506-900, Rio Claro, SP, Brazil
[c] Institute of Physics, University of São Paulo, 05315-970, São Paulo, SP, Brazil
[d] Department of Physics, Institute for Complex Systems and Mathematical Biology, SUPA, University of Aberdeen, AB24 3UX, Aberdeen, United Kingdom
[e] University of North Dakota, School of Electrical Engineering and Computer Science, 58202, Grand Forks, ND, United States of America
[f] Lorena School of Engineering (EEL-USP), University of São Paulo, 12602-810, Lorena, SP, Brazil
[g] Graduate Program in Science, State University of Ponta Grossa, 84030-900, Ponta Grossa, PR, Brazil
[h] Department of Mathematics and Statistics, State University of Ponta Grossa, 84030-900, Ponta Grossa, PR, Brazil
[i] Federal University of Paraná, Department of Physics, Interdisciplinary Center for Science, Technology and Innovation, Center for Modeling and Scientific Computing, 81531-980, Curitiba, PR, Brazil

ARTICLE INFO

*Keywords:*
Nonlinear dynamical systems
Chaos theory
Python package

ABSTRACT

Since Lorenz's seminal work on a simplified weather model, the numerical analysis of nonlinear dynamical systems has become one of the main subjects of research in physics. Despite of that, there remains a need for accessible, efficient, and easy-to-use computational tools to study such systems. In this paper, we introduce `pynamicalsys`, a simple yet powerful open-source Python module for the analysis of nonlinear dynamical systems. In particular, `pynamicalsys` implements tools for trajectory simulation, bifurcation diagrams, Lyapunov exponents and several others chaotic indicators, period orbit detection and their manifolds, as well as escape and basins analysis. We demonstrate the capabilities of `pynamicalsys` through a series of examples that reproduces well-known results in the literature while developing the mathematical analysis at the same time. We also provide the Jupyter notebook containing all the code used in this paper, including performance benchmarks. `pynamicalsys` is freely available via the Python Package Index (PyPI) and is intended to support both research and teaching in nonlinear dynamics.

## 1. Introduction

The success of Newton's theory on mechanics led to the idea of a deterministic, and fully predictable Universe. Laplace once famously stated that if an intellect at a certain moment in time would know all the forces that set nature in motion, and all the positions of all objects, then this intellect would be able to predict the past and the future of the entire Universe [1]. This idea was later challenged by Poincaré in his seminal work on the stability of the solar system [2]. Poincaré, to simplify the problem,

* Corresponding author.
*E-mail addresses:* rolim.sales.m@gmail.com (M.R. Sales), canton@essex.ac.uk (C.G. Antonopoulos).

https://doi.org/10.1016/j.chaos.2025.117269
Received 7 July 2025; Received in revised form 16 September 2025; Accepted 17 September 2025

considered the gravitation interaction of only three objects, and demonstrated that the system is generally non-integrable. In other words, for an arbitrary initial condition, its motion cannot be described by a finite set of integrals of motion. Only a particular set of initial conditions results in exact solutions. He also discovered homoclinic points and homoclinic tangles, which make the motion sensitive to initial conditions and exhibit unpredictable behavior.

This was the first evidence of the deterministic chaos, or simply chaos, that we know today. Poincaré, however, could not visualize the behavior he was describing. There were no computers at the time and his work remained underappreciated for around 70 years. In 1963, Lorenz when working on a simplified weather model, accidentally discovered that small changes in the initial conditions can lead to considerably different future states [3]. He considered a system of three differential equations and showed for the very first time a strange attractor: a geometric structure in phase space that is deterministic, i.e., follows a set of rules (the differential equations) yet is aperiodic and highly sensitive to small changes in the initial conditions.

After Lorenz's discovery, the mathematical foundations of dynamical systems theory were revitalized. Smale [4,5] introduced the concept of the horseshoe map, illustrating how repeated stretching and folding of phase space can lead to unpredictable behavior. Around the same time, the foundation of ergodic theory was also developed [6,7] and the term "chaos" started to become popular in the scientific community. The rapid development of computers and different programming languages, such as Pascal, Assembly, C, and Fortran, popular among the scientific community at the time, made large-scale numerical simulations feasible. In this context, the seminal work of Li and Yorke [8] formalized the modern notion of chaos by showing that a system with a period-3 orbit must exhibit chaotic dynamics. Feigenbaum also contributed to this when he discovered a universal law in period-doubling bifurcation, today known as Feigenbaum constant, that shows that different systems can exhibit the same route to chaotic dynamics [9,10].

Since then, and continuing to the present day, chaos theory has become a cornerstone of nonlinear science, with profound interdisciplinary influence. In physics, areas such as plasma physics [11–14], fluid turbulence [15,16], and astrophysical systems [17] rely heavily on chaotic models to describe their complex behavior. However, the reach of chaos theory extends far beyond physics. In biology, it plays a critical role in understanding heart rhythms [18–20], ecological models [21,22], and neuronal activity [23–26]. The latter has become a major focus of current research. Chaotic models have also influenced the economics field by modeling complex market behavior and financial instabilities [27–31], while in computer science, it has influenced fields like cryptography [32–34] and random number generation [35,36]. More recently, chaos theory has also found applications in the study of memristive systems, where nonlinear dynamics and memory effects give rise to complex and potentially chaotic behavior [37,38]. Additionally, machine learning techniques have become increasingly relevant in the study of dynamical systems [39], enabling prediction [40], classification [41,42], and parameter inference of nonlinear dynamical systems [43,44].

Therefore, in this paper, we present the `pynamicalsys` module, a simple yet powerful, open-source Python package implementing several tools for the analysis of nonlinear dynamical systems. Despite being written purely in Python, `pynamicalsys` offers high-performance thanks to Numba[1] [45] accelerated computation, offering speedups up to 130x compared to the pure Python version of the corresponding functions. We choose Python for its simplicity and extensive use within the scientific and programming communities, as it is currently one of the most, if not the most, widely used programming languages. All computations in `pynamicalsys` are performed using IEEE-754 double precision (float64), which provides approximately 15–16 decimal digits of precision. This corresponds to defining variables as `real(kind=8)` in Fortran or `double` in C. While floating-point arithmetic is inherently approximate, key results were verified to be robust against changes in time step and precision, ensuring that they are not dominated by numerical noise. You can install `pynamicalsys` using the Python Package Index (PyPI) via

```
$ pip install pynamicalsys
```

The package requires Python 3.8 or later and should run on any modern operating system with a standard Python environment and all required dependencies are installed automatically with the package.

We present the `pynamicalsys` classes and methods and illustrate their usage together with the theoretical discussion of the methods. We use `pynamicalsys` to reproduce several known results in the literature. This paper is accompanied by two Jupyter notebooks (see the Supplementary Material), which contains all the code needed to reproduce the results presented in this paper. The first notebook, *paper.ipynb* shows the CPU time for each calculation shown in this paper and the second notebook *benchmarks.ipynb*, contains all the benchmarks shown in the Appendix A confirming the high efficiency of `pynamicalsys`. These benchmarks were obtained on a MacBook Air equipped with an Apple M4 chip, featuring a 10-core CPU. In this paper, we only provide examples on how to obtain the data. For the plotting settings, we refer the reader to the Supplementary Material and the documentation page (https://pynamicalsys.readthedocs.io/en/latest/).

This paper is organized as follows. In Section 2, we demonstrate the basic use of the `DiscreteDynamicalSystem` class and perform some basic simulations such as trajectory and bifurcation diagram computation. In Section 3, we review some of the most used and efficient chaotic indicators for discrete dynamical systems and demonstrate their use by reproducing some known results in the literature. In Section 4, we focus on finding and classifying periodic orbits of two-dimensional maps and determining the stable and unstable manifolds of the saddles. Section 5 is devoted to the escape analysis, such as the computation and quantification of escape basins (or attraction basins). In Section 6, we demonstrate the use of the `ContinuousDynamicalSystem` and `HamiltonianSystem` classes and Section 7 contains our final remarks.

[1] https://numba.pydata.org.

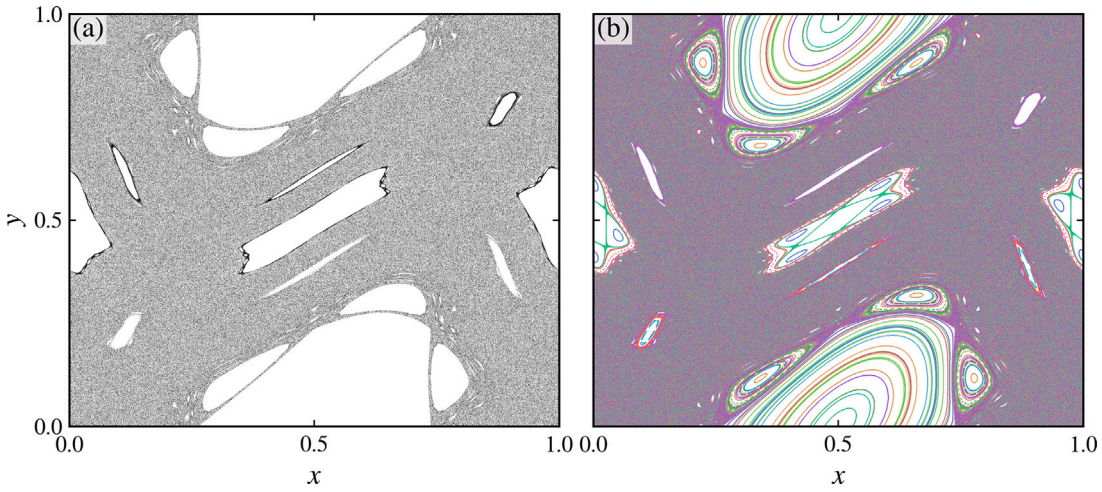


**Fig. 1.** Demonstration of the use of the `trajectory` method from the `DiscreteDynamicalSystem` class of `pynamicalsys` for the `model="standard map"`, with parameters $k = 1.5$ and total_time = 100000, for (a) a single initial condition and (b) 200 initial conditions. Execution times: (a) 48.9 ms and (b) 429 ms.

## 2. Basic system definition and simulation

We begin by presenting a few basic simulations on how to generate trajectories and phase space. We use the Chirikov–Taylor standard map [11], defined as

$$\begin{aligned} y_{n+1} &= y_n + \frac{k}{2\pi}\sin\left(2\pi x_n\right) \bmod 1, \\ x_{n+1} &= x_n + y_{n+1} \bmod 1. \end{aligned} \tag{1}$$

The Chirikov–Taylor standard map is a two-dimensional, area-preserving map where $x_n$ and $y_n$ are the conjugated canonical variables, $n = 0, 1, 2\ldots$, is the discrete time, and $k \geq 0$ is the nonlinearity parameter. For $k = 0$, the system is integrable and all orbits lie on period and quasiperiodic invariant tori. For $k > 0$, the sufficient irrational tori survive the perturbation, as predicted by the Kolmogorov–Arnold–Moser (KAM) theorem [46] and the rational ones are destroyed, leaving behind a set of elliptic (center) and hyperbolic (saddle) periodic orbits (Poincaré–Birkhoff theorem [46]). The stability islands are formed around the elliptic orbits and the stable and unstable manifolds of the hyperbolic orbits intersect each other in infinitely many points, generating chaotic dynamics. We refer the reader to Refs. [11,46–50] for further details on the dynamics of the standard map.

Let us then create our dynamical system object using the `DiscreteDynamicalSystem` class. To import the class, we proceed as follows:

```
>>> from pynamicalsys import DiscreteDynamicalSystem as dds
```

The `pynamicalsys` class takes on six arguments: `model`, `mapping`, `jacobian`, `backwards_mapping`, `system_dimension`, and `number_of_parameters`. You should either inform `model` *or* the remaining five arguments. The `DiscreteDynamicalSystem` class comes with a few built-in systems and the standard map is one of them. To check all the built-in systems, run

```
>>> dds.available_models()
['standard map', 'unbounded standard map', 'henon map', 'lozi map', 'rulkov map', 'logistic map', 'standard
    nontwist map', 'extended standard nontwist map', 'leonel map', '4d symplectic map']
```

Thus, to create an object of the standard map system, you proceed as

```
>>> ds = dds(model="standard map")
```

Now all methods of the `DiscreteDynamicalSystem` class are accessible via the `ds` object. To generate the trajectories, we use the `trajectory` method:

```
obj.trajectory(u, total_time, parameters=None, transient_time=None)
```

It takes on four arguments: `u`, the initial condition, `total_time` defines the total iteration time, `parameters` is a list of the system parameters, which can be left empty if the system has no parameters, and `transient_time` corresponds to the discarded initial iterations (default is `None`). The initial condition can be an one-dimensional array, corresponding to a single initial condition [Fig. 1(a)]:

```
>>> import numpy as np
>>> u = np.array([0.05, 0.05])  # Initial condition
>>> k = 1.5  # Parameter of the map
>>> total_time = 1000000  # Total iteration time
>>> trajectory = ds.trajectory(u, total_time, parameters=k)
>>> trajectory.shape
(1000000, 2)
```

In this case, the `trajectory()` method returns a two-dimensional array with `total_time` rows (the state at each time step) and 2 columns (the $x$ and $y$ coordinates). The initial condition `u` can also be a two-dimensional array of shape $(M, 2)$, where $M$ is the number of initial conditions and 2 corresponds to the dimension of the system [Fig. 1(b)]:

```
>>> num_ic = 200  # Number of initial conditions
>>> np.random.seed(13)  # For reproducibility
>>> u = np.random.rand(num_ic, 2)  # Random initial conditions
>>> k = 1.5  # Parameter of the map
>>> total_time = 100000  # Total iteration time for each initial condition
>>> trajectories = ds.trajectory(u, total_time, parameters=k)
>>> trajectories.shape
(20000000, 2)
```

Now, the `trajectory` method returns a two-dimensional array with `total_time`$\times$`num_ic` rows, i.e., it concatenates each trajectory and returns the trajectory of all initial conditions. Fig. 1(a) and (b) present complementary and key features of the standard map. In case (a), a single chaotic initial condition was used. This illustrates the chaotic sea, the boundaries of multiple islands that act as barriers in phase space (represented by the white areas), and, finally, the stickiness around the central islands, evidenced by the concentration of points indicating that the trajectory becomes temporarily trapped for long but finite times [47,51,52]. On the other hand, case (b) demonstrates how easily `pynamicalsys` can be adjusted to run multiple initial conditions. Each initial condition is marked with a distinct color, clearly distinguishing the regular regions, represented by islands, from the chaotic sea that fills the phase space.

Let us now consider a two-dimensional, dissipative map: the Hénon map [53]. The map is defined as

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

where $a$ and $b$ are the parameters of the system. This system is built-in within the `DiscreteDynamicalSystem` class as well. To initialize it we simply define the dynamical system object with `model="henon␣map"`. In this case, we are dealing with two parameters instead of one, and the order in which they are passed to the methods matters. To obtain this information from the built-in systems, you proceed as follows:

```
>>> from pynamicalsys import DiscreteDynamicalSystem as dds
>>> ds = dds(model="henon map")
>>> info = ds.info
>>> info["parameters"]
['a', 'b']
```

The `info` property returns a dictionary with several information regarding the built-in system. For all the available information, check the documentation. Thus, from the previous example, we see that for the built-in Hénon map, the first parameter is $a$ and the second is $b$. Now, let us generate the chaotic Hénon attractor for the parameters $a = 1.4$ and $b = 0.3$ [Fig. 2(a)]:

```
>>> from pynamicalsys import DiscreteDynamicalSystem as dds
>>> ds = dds(model="henon map")
>>> u = [0.1, 0.1]  # Initial condition
>>> a, b = 1.4, 0.3  # Parameters for the Henon map
```

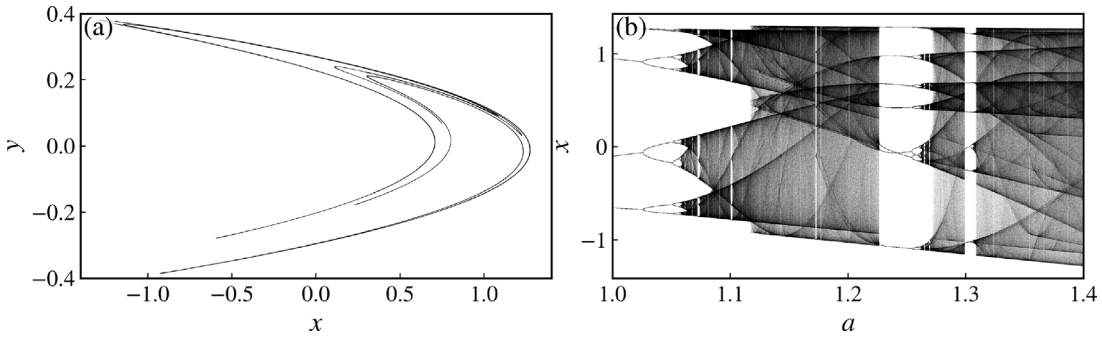


**Fig. 2.** Demonstration of the use of the `trajectory` method from the `DiscreteDynamicalSystem` class in `pynamicalsys` for the `model="henon map"` with parameters $a = 1.4$ and $b = 0.3$ in (a), and of the `bifurcation_diagram` method over the interval $a \in [1.0, 1.4]$ with $b = 0.3$ in (b). Execution times; (a) 283 ms and (b) 3.14 s.

```
>>> parameters = [a, b]
>>> total_time = 1000000  # total iteration time
>>> transient_time = 500000  # Transient time for the Henon map
>>> trajectory = ds.trajectory(u, total_time, parameters=parameters, transient_time=transient_time)
>>> trajectory.shape
(500000, 2)
```

The `transient_time` argument tells the `trajectory()` method the number of iterations to discard, thus it returns a two-dimensional array of `total_time` − `transient_time` rows.

When studying dissipative systems, one useful tool is the bifurcation diagram. The dynamical systems, in general, depend on different parameters and as these parameters change, the system can undergo transitions known as bifurcations. We use the `bifurcation_diagram` method of the `DiscreteDynamicalSystem` class of `pynamicalsys`:

```
obj.bifurcation_diagram(u, param_index, param_range, total_time, parameters=None, transient_time=None,
    ↪ continuation=False, return_last_state=False, observable_index=0)
```

Here, `u` is the initial condition, `param_index` is an integer that corresponds to the position of the bifurcation parameter in the parameter list, i.e., 0 is the first parameter, 1 is the second, and so on. `param_range` determines the bifurcation parameter values. It can either be a predefined sample of parameters, i.e., `param_range = [0.1, 0.2, 0.3]` or it can be a tuple indicating the starting and ending values and the number of values for a linear spacing: `param_range = (start, end, num_params)`. `total_time` is the total iteration time, including the transient time, and `parameters` is a list with the remaining parameter values, i.e., those that remain fixed. If the system only has one parameter, when computing the bifurcation diagram this should be set to `None`. `transient_time` is the initial iteration time to discard (default is `None`) and `continuation` determines whether to reset or not the initial condition for every new parameter value. If `continuation=False`, for every new parameter value, the bifurcation diagram is computed using the provided initial condition `u`. However, if `continuation=True`, then it is performed a numerical continuation sweep, i.e., the initial condition for the next parameter value is the last state of the previous parameter (default is `False`). The argument `return_last_state` determines whether to return the last state as well (default is `False`) and `observable_index` corresponds to the coordinate used in the bifurcation diagram. By default, it uses the first coordinate. To use a different one, set this argument to `observable_index=1`, to use the second coordinate, for example.

For the Hénon map, we choose to fix $b = 0.3$ and we want to change $a$ in the interval $a \in [1.0, 1.4]$. Thus, we proceed as follows [Fig. 2(b)]:

```
>>> from pynamicalsys import DiscreteDynamicalSystem as dds
>>> ds = dds(model="henon map")
>>> u = [0.1, 0.1]  # Initial condition
>>> b = 0.3  # Keep b fixed
>>> parameters = b
>>> param_range = (1, 1.4, 2500)  # Parameter range (a in this case)
>>> param_index = 0  # a is going to be changed (parameters = [a, b])
>>> total_time = 8000  # Total iteration time
>>> transient_time = 2000  # Transient time
>>> param_values, bifurcation_diagram = ds.bifurcation_diagram(u, param_index, param_range, total_time,
    parameters=parameters, transient_time=transient_time)
>>> bifurcation_diagram.shape
(2500, 6000)
```

The `bifurcation_diagram()` method returns two arrays: the first contains all parameter values, and the second is a 2D array representing the corresponding coordinates for each parameter value. If `return_last_state` is set to `True`, then it also returns the last state of the system at the final parameter value. This can be useful when studying hysteresis, intriguing phenomena that can be present in diverse dynamical systems [54].

## 3. Chaotic indicators

Chaos is notably present in both natural phenomena and mathematical models. Developing reliable methods and tools to identify and distinguish between chaotic and periodic behaviors is often a crucial objective. Numerous numerical techniques, each with distinct approaches, have been proposed to detect the presence of chaotic dynamics. To facilitate this, the `DiscreteDynamicalSystem` class of `pynamicalsys` offers a variety of easy-to-use methods.

This section is organized as follows: (i) Lyapunov exponents (LEs), (ii) Linear dependence indexes, (iii) Weighted Birkhoff averages, (iv) Recurrence time entropy, and (v) Hurst exponent.

### 3.1. Lyapunov exponents

#### 3.1.1. One-dimensional maps

The LEs are a measure of the average exponential rates at which infinitesimal perturbations to a trajectory in a dynamical system grow or decay over time. Given an one-dimensional discrete dynamical system $x_{n+1} = f(x_n)$, where $f : \mathbb{R} \to \mathbb{R}$ is a smooth map. Let $x_0$ and $y_0 = x_0 + \delta_0$ be two initial conditions infinitesimally close to each other ($|\delta_0| \ll 1$). After one iteration, the two initial conditions become

$$\begin{aligned} x_1 &= f(x_0), \\ y_1 &= f(x_0 + \delta_0). \end{aligned} \tag{3}$$

The difference between them is

$$\delta_1 = y_1 - x_1 = f(x_0 + \delta_0) - f(x_0). \tag{4}$$

For $\delta_0$ small enough, we can linearize $f(x_0 + \delta_0)$ using a first-order Taylor expansion:

$$f(x_0 + \delta_0) = f(x_0) + f'(x_0)\delta_0 + \mathcal{O}(\delta_0^2), \tag{5}$$

leading us to $\delta_1 = f'(x_0)\delta_0$. For the next iteration, we obtain $\delta_2 = f'(x_1)f'(x_0)\delta_0$. Thus, repeating this for $n$ iterations, we obtain

$$\delta_n = \left( \prod_{i=0}^{n-1} f'(x_i) \right) \delta_0. \tag{6}$$

We want to quantify how this perturbation grows exponentially. Thus, we derive the exponential rate of divergence:

$$\frac{1}{n} \log \left| \frac{\delta_n}{\delta_0} \right| = \frac{1}{n} \log \left( \prod_{i=0}^{n-1} |f'(x_i)| \right) = \frac{1}{n} \sum_{i=0}^{n-1} \log |f'(x_i)|. \tag{7}$$

The Lyapunov exponent $\lambda$ is defined as the long-term average exponential rate of separation, i.e.,

$$\lambda = \lim_{n \to \infty} \frac{1}{n} \sum_{i=0}^{n-1} \log |f'(x_i)|. \tag{8}$$

#### 3.1.2. Higher-dimensional maps

For higher-dimensional dynamical systems, the derivation is somewhat different. Given a $d$-dimensional discrete-time dynamical system $\mathbf{x}_{n+1} = \mathbf{f}(\mathbf{x}_n)$, where $\mathbf{f} : \mathbb{R}^d \to \mathbb{R}^d$ is a smooth map, the Lyapunov spectrum $\lambda_1 \geq \lambda_2 \geq \cdots \geq \lambda_d$ is defined under Oseledec's multiplicative ergodic theorem [55]. Let $J(\mathbf{x}) = \mathbf{Df}(\mathbf{x})$ be the Jacobian matrix of the map $\mathbf{f}$ at point $\mathbf{x}$. The matrix

$$J_n(\mathbf{x}_0) = J(\mathbf{x}_{n-1})J(\mathbf{x}_{n-2}) \ldots J(\mathbf{x}_1)J(\mathbf{x}_0) \tag{9}$$

describes the evolution of the tangent vectors under the linearized dynamics. Oseledec's theorem states that for almost every initial condition $\mathbf{x}_0$, the following limit exists:

$$\Lambda(\mathbf{x}_0) = \lim_{n \to \infty} \left[ J_n^T(\mathbf{x}_0) J_n(\mathbf{x}_0) \right]^{1/2n}. \tag{10}$$

The LEs are related to the eigenvalues of the matrix $\Lambda$ and are given by

$$\lambda_i = \lim_{n \to \infty} \frac{1}{n} \log \left\| J_n(\mathbf{x}_0)\mathbf{v}_i \right\|, \tag{11}$$

where $\mathbf{v}_i$ are the corresponding eigenvector of $\Lambda(\mathbf{x}_0)$.

While Eqs. (10) and (11) are extremely elegant in a theoretical sense, directly computing the matrix $J_n(\mathbf{x}_0)$ leads to numerical instability, and the product becomes dominated by the direction of maximal growth, making it impossible to calculate the smaller

LEs. To overcome this issue, several methods have been proposed to numerically estimate the LEs [56–59]. We describe in the following the QR-based approach that has become the standard procedure in numerical studies of the LEs. The idea of this method is to evolve an orthonormal basis of tangent vectors and reorthonormalize them at each step using a QR decomposition. Given an initial orthonormal matrix $Q_0 \in \mathbb{R}^{d\times d}$, where each column of $Q_0$ is a tangent vector, the evolution of $Q_0$ according to the linearized dynamics is

$$A_1 = J_1 Q_0. \tag{12}$$

We now perform a QR decomposition on $A_1$: $A_1 = Q_1 R_1$, where $Q_1 \in \mathbb{R}^{d\times d}$ is an orthonormal matrix and $R_1 \in \mathbb{R}^{d\times d}$ is an upper triangular matrix. We can then write

$$J_1 = Q_1 R_1 Q_0^{-1}. \tag{13}$$

For the second iteration, the procedure is analogous and we obtain $J_2 = Q_2 R_2 Q_1^{-1}$. By repeating this procedure recursively, we can express the Jacobian matrix at each time $n$ as

$$J_n = Q_n R_n Q_{n-1}^{-1}. \tag{14}$$

The physical meaning behind this method is as follows: the Jacobian matrix $J_n$ evolves the orthonormal basis $Q_{n-1}$ under the linearized dynamics, resulting in a new set of vectors $A_n = J_n Q_{n-1}$, which are generally not orthonormal. If we continually evolve this set of vectors, eventually all of them align with the direction of maximal growth and we lose all information about the other LEs. When we perform a QR decomposition on $A_n$, yielding $A_n = Q_n R_n$, we are reorthonormalizing the tangent basis to prevent the collapse onto the direction of maximum growth, while simultaneously extracting the local stretching and shrinking information captured by the matrix $R_n$. The matrix $Q_n$ then becomes the updated orthonormal basis, aligned with the principal directions of local deformation. The absolute values of the diagonal elements of $R_n$, $|r_{ii}^{(n)}|$, represent the instantaneous rates of expansion or contraction along each orthonormal direction. Therefore, by computing the time averages of the logarithms of these values, we can estimate the LEs:

$$\lambda_i = \lim_{n\to\infty} \frac{1}{n} \sum_{j=1}^{n} \log \left| r_{ii}^{(j)} \right|. \tag{15}$$

This QR method is not just a computational technique. It is connected with the theoretical framework established by Oseledec's theorem [Eq. (10)]. To see that, let us substitute each Jacobian matrix in terms of its QR decomposition in the matrix product in Eq. (9). We get

$$J_n = \left(Q_n R_n Q_{n-1}^{-1}\right)\left(Q_{n-1} R_{n-1} Q_{n-2}^{-1}\right) \dots \left(Q_1 R_1 Q_0^{-1}\right). \tag{16}$$

All of the $Q_k^{-1}$ and $Q_k$ cancel due to the orthogonality of the matrices and the expression simplifies to

$$J_n = Q_n \left(R_n R_{n-1} \dots R_1\right) Q_0^{-1} = Q_n \mathcal{R}_n Q_0^{-1}, \tag{17}$$

where $\mathcal{R}_n = R_n R_{n-1} \dots R_1$. Now, we compute the product $J_n^T J_n$:

$$\begin{aligned} J_n^T J_n &= (Q_0^{-1})^T \mathcal{R}_n^T Q_n^T Q_n \mathcal{R}_n Q_0^{-1}, \\ &= Q_0^{-T} \mathcal{R}_n^T \mathcal{R}_n Q_0^{-1}. \end{aligned} \tag{18}$$

This construction leads to a similarity transformation: the matrix $\mathcal{R}_n^T \mathcal{R}_n$ is similar to $J_n^T J_n$, and thus both share the same eigenvalues. As $n \to \infty$, the eigenvalues of the matrix $J_n^T J_n$ converge to those of the limiting matrix $\Lambda(\mathbf{x}_0)$ [Eq. (10)], whose eigenvalues define the LEs. Therefore, the eigenvalues of $\mathcal{R}_n^T \mathcal{R}_n$ are a numerical approximation to those of $J_n^T J_n$ and consequently yield the same LEs in the long-time limit [Eq. (15)].

For two-dimensional systems, it is possible to derive an analytical recursive expression for the diagonal elements of $R_n$. We can rewrite Eq. (14) as

$$R_n = Q_n^{-1} J_n Q_{n-1}. \tag{19}$$

Let us choose as our orthogonal matrix the two-dimensional rotation matrix:

$$Q_n = \begin{pmatrix} \cos\beta_n & -\sin\beta_n \\ \sin\beta_n & \cos\beta_n \end{pmatrix}. \tag{20}$$

This matrix rotates a two-dimensional vector counterclockwise by an angle $\beta$. Thus, Eq. (19) becomes

$$R_n = \begin{pmatrix} r_n^{(11)} & r_n^{(12)} \\ 0 & r_n^{(22)} \end{pmatrix} = \begin{pmatrix} \cos\beta_n & \sin\beta_n \\ -\sin\beta_n & \cos\beta_n \end{pmatrix} \begin{pmatrix} J_n^{(11)} & J_n^{(12)} \\ J_n^{(21)} & J_n^{(22)} \end{pmatrix} \begin{pmatrix} \cos\beta_{n-1} & -\sin\beta_{n-1} \\ \sin\beta_{n-1} & \cos\beta_{n-1} \end{pmatrix}. \tag{21}$$

The diagonal elements are then given by

$$\begin{aligned} r_n^{(11)} &= \cos\beta_n \left(J_n^{(11)} \cos\beta_{n-1} + J_n^{(12)} \sin\beta_{n-1}\right) + \sin\beta_n \left(J_n^{(21)} \cos\beta_{n-1} + J_n^{(22)} \sin\beta_{n-1}\right), \\ r_n^{(22)} &= -\sin\beta_n \left(-J_n^{(11)} \sin\beta_{n-1} + J_n^{(12)} \cos\beta_{n-1}\right) + \cos\beta_n \left(-J_n^{(21)} \sin\beta_{n-1} + J_n^{(22)} \cos\beta_{n-1}\right), \end{aligned} \tag{22}$$

and the relation between $\beta_n$ and $\beta_{n-1}$ is

$$\tan\beta_n = \frac{J_n^{(21)}\cos\beta_{n-1} + J_n^{(22)}\sin\beta_{n-1}}{J_n^{(11)}\cos\beta_{n-1} + J_n^{(12)}\sin\beta_{n-1}}. \tag{23}$$

Therefore, given an initial angle $\beta_0$ (typically $\beta_0 = 0$), we first calculate $\beta_1$ using Eq. (23). Then, we calculate the diagonal elements of the matrix $R_n$ via Eq. (22). By repeating this process iteratively, we can calculate the LEs using Eq. (15).

To calculate the LEs, we use the `lyapunov` method of the `DiscreteDynamicalSystem` class of our package `pynamicalsys`:

```
obj.lyapunov(u, total_time, parameters=None, method="QR", return_history=False, sample_times=None,
    ↪ transient_time=None, log_base=np.e)
```

The arguments `u`, `total_time`, and `parameters` are the initial condition, the list of parameters of the system, and the total iteration time, respectively. For an one-dimensional system, the `lyapunov` method computes the Lyapunov exponent via Eq. (8). For more than one dimension, the optional argument `method` determines which QR decomposition to use. For two-dimensional systems, if `method="QR"`, it automatically uses the analytical recursive expression we have derived [Eqs. (22) and (23)]. For higher-dimensional systems, the default method employs the modified Gram–Schmidt algorithm to perform QR decomposition. If the problem requires improved numerical stability (e.g. very large-scale problem), it is possible to set `method="QR_HH"` to use Householder reflections instead. The optional argument `return_history`, when set to `True` (default is `False`) tells the 'lyapunov method to return the value of each Lyapunov exponent at every time step. However, for long simulations, this can result in extremely large arrays that may exceed the system's memory capacity. To avoid this, you can pass to the optional argument `sample_times` a list of time points so that the `lyapunov` method returns the LEs only at those specified times. The optional argument `transient_time` determines the number of iterations to discard before starting the LEs calculation and `log_base` is the base of the logarithm used to calculate the LEs (default is set to `np.e`, i.e., the natural logarithm).

### 3.1.3. User-defined systems

Let us then consider a system that is not built-in. We consider the dissipative asymmetric kicked rotor map (DAKRM), defined as [60–63]

$$\begin{aligned} y_{n+1} &= (1-\gamma)y_n + k\left[\sin(x_n) + a\sin\left(2x_n + \frac{\pi}{2}\right)\right], \\ x_{n+1} &= x_n + y_{n+1} \bmod 2\pi, \end{aligned} \tag{24}$$

where $k \geq 0$ corresponds to the strength of the perturbation, similar to $k$ in the standard map [cf. Eq. (1)], $a$ is the asymmetry parameter which breaks the spatial symmetry for $a \neq 0$ and $\gamma \in [0, 1]$ is the dissipative parameter. For $a = 0$ and $\gamma = 0$, we recover the standard map. To calculate the LEs, we need the Jacobian matrix of the system, which is given by

$$J = \begin{pmatrix} 1 + \frac{\partial y_{n+1}}{\partial x_n} & \frac{\partial y_{n+1}}{\partial y_n} \\ \frac{\partial y_{n+1}}{\partial x_n} & \frac{\partial y_{n+1}}{\partial y_n} \end{pmatrix}, \tag{25}$$

where

$$\begin{aligned} \frac{\partial y_{n+1}}{\partial x_n} &= k\left[\cos(x_n) + 2a\cos\left(2x_n + \frac{\pi}{2}\right)\right], \\ \frac{\partial y_{n+1}}{\partial y_n} &= 1 - \gamma. \end{aligned} \tag{26}$$

Since the DAKRM is not built-in within the `pynamicalsys` package, we need to define it ourselves. The mapping function signature should be `u1 = f(u0, parameters)`, i.e., given the initial condition and the parameters, the function returns the next state:

```
>>> import numpy as np
>>> from numba import njit
>>> @njit
>>> def dakrm(u, parameters):
...     k, a, gamma = parameters
...     x, y = u
...     y_new = (1 - gamma) * y + k * (np.sin(x) + a * np.sin(2 * x + np.pi / 2))
...     x_new = (x + y_new) % (2 * np.pi)
...     return np.array([x_new, y_new])
```

Note the use of the `@njit` decorator before the function definition. It is absolutely crucial that both the model and Jacobian functions are decorated with `@njit`. This decorator enables Numba to compile the function to optimized machine code, resulting in a significant performance boost for numerical computations. Furthermore, since all methods within `pynamicalsys` are also

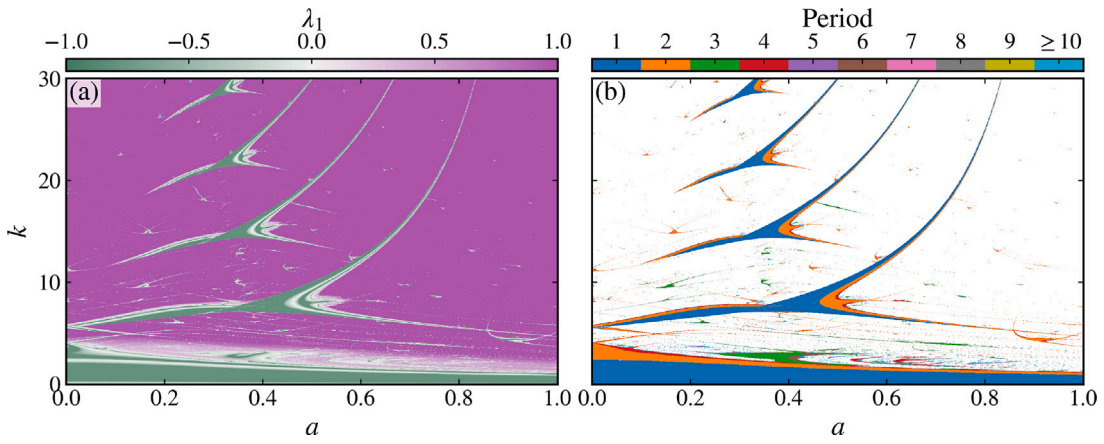


**Fig. 3.** Demonstration of the use of the (a) `lyapunov` method and (b) `period` method from the `DiscreteDynamicalSystem` class of `pynamicalsys` for the dissipative kicked rotor map [Eq. (24)] with parameter $\gamma = 0.8$. Execution times: (a) 25 min 31 s and (b) 7 min 56 s. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

decorated with `@njit`, the model functions must be decorated similarly to ensure compatibility. Without this, Numba would raise errors due to mixing compiled and uncompiled functions. Having defined the model function, the next step is to define the Jacobian matrix function. The Jacobian function signature is `J = jac(u, parameters, *args)`, i.e., given the current state of the system and its parameters, the function returns a two-dimensional array of shape `(d, d)`, where `d` is the dimension of the system:

```
>>> @njit
>>> def dakrm_jacobian(u, parameters, *args):
...     k, a, gamma = parameters
...     x, y = u
...     dFdx = k * (np.cos(x) + 2 * a * np.cos(2 * x + np.pi / 2))
...     dFdy = 1 - gamma
...     return np.array([
...         [1 + dFdx, dFdy],
...         [dFdx,     dFdy]])
```

Finally, with both the map equation and Jacobian matrix defined, we can create the dynamical system object as

```
>>> ds = dds(mapping=dakrm, jacobian=dakrm_jacobian, system_dimension=2, number_of_parameters=3)
```

Even though `dds` also takes on the `backwards_mapping` argument, since we are not using it to calculate the LEs, it is not necessary to provide it. If, however, one wishes to calculate the manifolds of a dynamical system, for instance, then it becomes necessary. The point is, when defining a dynamical system that is not built-in, you only need to provide the functions that are going to be used. If you are only interested in drawing the trajectories, then you do not have to provide the Jacobian matrix, for instance.

To illustrate the calculation of the LEs, let us fix $k = 8.0$ and $\gamma = 0.8$ and calculate them for $a = 0.47$:

```
>>> u = [1.78, 0.0]  # Initial condition
>>> total_time = 10000 # Total iteration time
>>> transient_time = 5000  # Transient time
>>> k, a, gamma = 8, 0.47, 0.8  # Parameters of the system
>>> parameters = [k, a, gamma]
>>> ds.lyapunov(u, total_time, parameters=parameters, transient_time=transient_time)
array([-0.35202562, -1.25741229])
```

and for $a = 0.6$:

```
>>> u = [1.78, 0.0]  # Initial condition
>>> total_time = 10000  # Total iteration time
>>> transient_time = 5000  # Transient time
>>> k, a, gamma = 8, 0.6, 0.8  # Parameters of the system
>>> parameters = [k, a, gamma]
>>> ds.lyapunov(u, total_time, parameters=parameters, transient_time=transient_time)
array([ 1.57224186, -3.18167977])
```

Thus, the `lyapunov()` method returns an one-dimensional array with all the LEs in decreasing order. For $a = 0.47$ we obtain two negative LEs, indicating periodic dynamics, and for $a = 0.6$ the largest Lyapunov exponent is positive while the second one is negative, i.e., the dynamics is chaotic.

Let us suppose now that you do not know the Jacobian matrix of your system. You can instantiate the `dds` class without providing a Jacobian matrix function:

```
>>> ds = dds(mapping=dakrm, system_dimension=2, number_of_parameters=3)
```

In this case, the Jacobian matrix is numerically determined as

$$J_{ij} = \frac{f_i(x_1, x_2, \dots, x_j + \epsilon, \dots, x_n) - f_i(x_1, x_2, \dots, x_j - \epsilon, \dots, x_n)}{2\epsilon}, \tag{27}$$

where $f_i$ is the $i$th component of the mapping and $\epsilon$ is chosen as

$$\epsilon = \left(\epsilon_{\text{mach}}\right)^{1/3} \cdot \max\left(1, \|\mathbf{u}\|_2\right), \tag{28}$$

where $\epsilon_{\text{mach}}$ is the machine epsilon (the smallest representable difference between floating point numbers), $\mathbf{u} = (x_1, \dots, x_d)^T$ is the state vector, and $\|\mathbf{u}\|_2$ is the Euclidean norm of the state vector. This choice is a compromise between truncation and round-off errors [64]. The LEs is computed in the same way as before:

```
>>> u = [1.78, 0.0]  # Initial condition
>>> total_time = 10000  # Total iteration time
>>> transient_time = 5000  # Transient time
>>> k, a, gamma = 8, 0.6, 0.8  # Parameters of the system
>>> parameters = [k, a, gamma]
>>> ds.lyapunov(u, total_time, parameters=parameters, transient_time=transient_time)
array([ 1.5740678 , -3.18114158])
```

The final value, of course, differs from the one obtained when we instantiated the `dds` class with the Jacobian matrix.

While the LEs tell us that for $a = 0.4$ the dynamics is periodic, they tell us nothing about the period itself. This information can be obtained using the `period` of the `DiscreteDynamicalSystem` class of `pynamicalsys`:

```
obj.period(u, max_time, parameters=None, transient_time=None, tolerance=1e-10, min_period=1, max_period
    ↪ =1000, stability_checks=3)
```

The arguments `u`, `parameters`, and `transient_time` have been covered already. The argument `max_time` is the maximum iteration time. The optional argument `tolerance` specifies the numerical tolerance for period detection; it defines the radius of the neighborhood around the initial condition used to determine if the trajectory has returned. The optional arguments `min_period` and `max_period` set the minimum and maximum periods to consider, respectively. The optional argument `stability_checks` determines how many consecutive returns to the neighborhood are required to confirm the period and ensure numerical stability.

For the previous period example, you can proceed as follows:

```
>>> u = [1.78, 0.0]  # Initial condition
>>> total_time = 10000  # Total iteration time
>>> transient_time = 5000  # Transient time
>>> k, a, gamma = 8, 0.47, 0.8  # Parameters of the system
>>> parameters = [k, a, gamma]
>>> ds.period(u, total_time, parameters=parameters, transient_time=transient_time)
2
```

The `period` method determines whether a given initial condition leads to a periodic orbit. If so, it returns the period, otherwise, it returns `-1` to indicate a quasiperiodic or chaotic orbit. In the last example, the period of the orbit is 2. By changing continuously the parameters, it is possible to calculate the LEs and the period in the parameter space as well, as shown in Fig. 3 [63]. The chaotic regions in Fig. 3(a) are depicted in pink, while the regular regions appear in green. White points indicate locations where $\lambda_1 \to 0$, corresponds to period-doubling bifurcations. These bifurcations can be more clearly observed in Fig. 3(b), where colors represent the periodicity of each point. Within the shrimp-shaped domains [65,66], the period progresses from 1 to 2, 4, and so on, eventually leading to chaotic dynamics (marked in white).

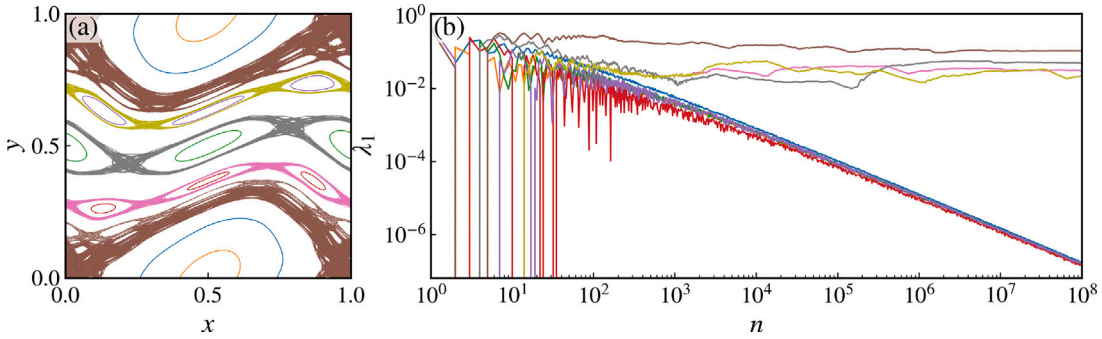


**Fig. 4.** Demonstration of the use of the `lyapunov` method of the `DiscreteDynamicalSystem` class of `pynamicalsys` to compute the time series of the LEs for the standard map [Eq. (1)] using specific sample times for the parameter $k = 0.9$. Execution times: (a) 682 ms and (b) 2 min 29 s. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

### 3.1.4. Lyapunov exponents time evolution

In many situations, the history of the LEs is as important as the final value. It tells us how the LEs approach their asymptotic value. To obtain the time evolution of the LEs, set `return_history=True` (the default is `False`) in the `lyapunov` method. To illustrate this feature, let us return to the standard map [Eq. (1)]. We choose $k = 0.9$ and select five regular and four chaotic initial conditions [Fig. 4(a)]. Then, we proceed as follows:

```
>>> import numpy as np
>>> from pynamicalsys import DiscreteDynamicalSystem as dds
>>> ds = dds(model="standard map")
>>> k = 0.9  # Parameter of the standard map
>>> total_time = 100000000  # Total iteration time
>>> sample_times = np.unique(np.logspace(np.log10(1), np.log10(total_time), 1000).astype(int))  # Sample times to return the LEs
>>> u = np.array([[0.26, 0], [0.4, 0], [0, 0.45], [0.1, 0.25], [0.1, 0.68], [0.06, 0.05], [0, 0.3], [0, 0.6], [0, 0.7]])  # Initial conditions (the first 5 are regular and the last 4 are chaotic)
>>> history_LEs = np.array([ds.lyapunov(u[i], total_time, parameters=k, return_history=True, sample_times=sample_times) for i in range(u.shape[0])])
>>> history_LEs.shape
(9, 836, 2)
```

This example calculates the history of the LEs for each initial condition at the specified sample times [Fig. 4(b)]. The largest Lyapunov exponent for the blue, orange, red, green, and purple initial conditions converges toward zero as a power law, indicating regular dynamics. Additionally, they all go to zero at the same rate. The fastest rate of $\lambda_1$ toward zero is obtained exactly on the elliptic points, as Manchein and Beims have demonstrated [67].

The remaining initial conditions, brown, pink, yellow, and gray, exhibit a positive largest Lyapunov exponent, indicating a chaotic behavior. They seem to converge to a positive value, however, on several occasions, when $\lambda_1$ seems to have converged, its value suddenly decreases and after some time it starts to increase again, and once again, its value decreases. This behavior happens for arbitrarily long times due to the phenomenon of stickiness [48,68–77]. The stickiness effect, first identified by Contopoulos [68], influences chaotic orbits as they approach stability islands. When near a regular region, these orbits can become trapped in the vicinity of the islands for arbitrarily long times, exhibiting quasiperiodic-like behavior during this transient phase. As a result, the largest Lyapunov exponent decreases. This phenomenon arises from the complex hierarchical structure of islands-around-islands embedded within the chaotic sea of two-dimensional, area-preserving maps. After escaping one sticky region, a chaotic orbit may later become trapped again in another, repeating the process. Such successive trappings lead to intermittent dynamics, affecting statistical properties, such as diffusion, decay of correlations, and transport.

### 3.1.5. Finite-time Lyapunov exponents

Due to the intermittent behavior of chaotic orbits, computing the LEs for longer times might not be the best approach to detect and characterize the stickiness effect. Instead, Szezech et al. [78] proposed the calculation of the finite-time LEs (FTLEs). Before we proceed, we should clarify the term *finite-time*. Strictly speaking, *all* numerical simulations are finite-time. However, their key idea was to compute the LEs not for the whole single trajectory with long times, $N \gg 1$, but rather for shorter time windows, $n \sim 1$, along the same trajectory. Since a chaotic orbit will eventually fill the whole available chaotic component, by considering a total time $N \gg 1$ and calculating the Lyapunov exponent in windows of size $n \ll N$, we obtain a collection of values for the FTLEs, $\{\lambda_1^{(i)}\}_{i=1,2,\ldots,M}$, where $M = N/n$, and these values characterize both the intervals where the chaotic orbits are trapped and the intervals where they are in the bulk of the chaotic sea.

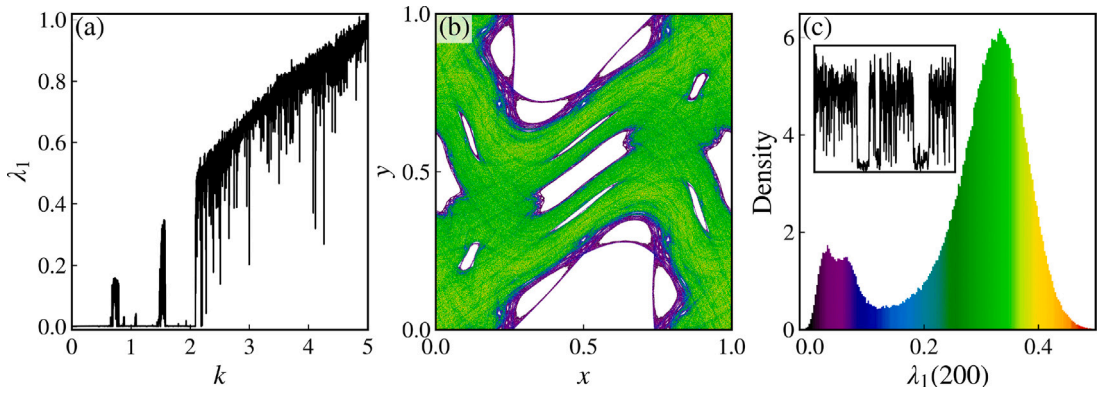


**Fig. 5.** (a) Demonstration of the use of the `lyapunov` exponent method of the `DiscreteDynamicalSystem` class of `pynamicalsys` to calculate the LEs as a function of the parameter $k$ of the standard map [Eq. (1)] and (b) and (c) demonstration of the use of the `finite_time_lyapunov` exponent method of the `DiscreteDynamicalSystem` class of `pynamicalsys` to calculate the distribution of FTLEs for the standard map with $k = 1.5$. Execution times: (a) 9.56 s, (b) 846 ms, and (c) 39 s.

To calculate the FTLEs, we use the `finite_time_lyapunov` method of the `DiscreteDynamicalSystem` class of our package `pynamicalsys`:

```
obj.finite_time_lyapunov(u, total_time, finite_time, parameters=None, method="QR", transient_time=None,
    ↪ log_base=np.e, return_points=False)
```

Here, `total_time` is the total iteration time, $N$, whereas `finite_time` is the size of the time windows, $n$, that the LEs are calculated. The optional argument `return_points`, if set to `True`, tells the `finite_time_lyapunov` method to also return the initial condition used to generate the corresponding FTLEs. The method thus returns two arrays. Both of them are arrays with $M = N/n$ rows and $d$ columns. Each row contains the FTLEs for the first array and for the second array, each row contains the initial condition that generated the respective finite-time Lyapunov exponent.

The following code snippet illustrates the calculation of the LEs as a function of the parameter $k$ as well as the calculation of the FTLEs:

```
>>> import numpy as np
>>> from pynamicalsys import DiscreteDynamicalSystem as dds
>>> ds = dds(model="standard map")
>>> u = [0.5, 0.25]  # Initial condition for λ vs k
>>> k_range = (0, 5, 5000)  # Interval in k
>>> k = np.linspace(*k_range)  # Create the k values
>>> total_time = 5000  # Total iteration time
>>> lyapunov_vs_k = [ds.lyapunov(u, total_time, k[i]) for i in range(k_range[2])]
>>> u = [0.05, 0.05]  # Initial condition for the finite-time Lyapunov exponents
>>> k = 1.5  # Parameter k
>>> total_time = 100000000  # Total iteration time
>>> finite_time = 200  # Finite-time (window size)
>>> ftle, points = ds.finite_time_lyapunov(u, total_time, finite_time, parameters=k, return_points=True)
```

In Fig. 5(a) we show the behavior of $\lambda_1$ as a function of $k$ for the initial condition $(x_0, y_0) = (0.5, 0.25)$. We use the `points` output of the `finite-time_lyapunov` method as initial conditions and we generate their trajectory for $n = 200$. We color the points of the trajectory according to their respective finite-time Lyapunov exponent value, and this yields Fig. 5(b). We also calculate the distribution of the finite-time values [Fig. 5(c)]. We color the histogram according to the finite-time value using the same color code as in Fig. 5(b). The distribution is a bi-modal distribution, with the largest peak corresponding to the times when the trajectory was on the bulk of the chaotic sea. The smaller peak corresponds to the times when the trajectory was trapped in the vicinity of a stability island. By comparing the colors in Fig. 5(b) and (c), we can identify the regions in phase space that generated each finite-time value. The smaller values, mainly purple color, correspond to the neighborhood of the islands. The inset in Fig. 5(c) is the "times series" of the finite-time values, i.e., for each time window, we plot the corresponding value of $\lambda_1^{(i)}$. In the inset, we notice mainly two sharp drops in the value of $\lambda_1^{(i)}$. These drops are the times when the trajectory became trapped and they are the reason for the bi-modal distribution.

### 3.2. Linear dependence indexes

As we have discussed in Section 3.1, given an orthonormal basis $Q \in \mathbb{R}^{d\times d}$, the Jacobian matrix $J$ evolves this basis under the linearized dynamics according to $A_n = J_n Q_{n-1}$. As we continue to evolve this basis, without performing the QR decomposition, all the basis vectors eventually align with the direction of maximal growth. In light of that, two efficient chaotic indicators have been proposed [79–81], the second being the generalization of the first. Given a $d$-dimensional discrete-dynamical system $\mathbf{x}_{n+1} = \mathbf{f}(\mathbf{x}_n)$, where $\mathbf{f} : \mathbb{R}^d \to \mathbb{R}^d$, let $J(\mathbf{x}) = \mathbf{Df}(\mathbf{x})$ be the Jacobian matrix of the map $\mathbf{f}$ at point $\mathbf{x}$ and $Q \in \mathbb{R}^{d\times 2}$ be a matrix whose columns are two deviation vectors, $\mathbf{v}_1 \in \mathbb{R}^d$ and $\mathbf{v}_2 \in \mathbb{R}^d$. As the deviation vectors evolve under the linearized dynamics, $\mathbf{v}_1$ and $\mathbf{v}_2$ gradually align with the direction of the maximum Lyapunov exponent. The deviation vectors can either align parallel or anti-parallel to the most unstable direction. Thus, we define the parallel alignment index

$$\mathrm{PAI}(n) = \left\| \frac{\mathbf{v}_1}{\|\mathbf{v}_1\|} - \frac{\mathbf{v}_2}{\|\mathbf{v}_2\|} \right\| \tag{29}$$

and the antiparallel alignment index

$$\mathrm{AAI}(n) = \left\| \frac{\mathbf{v}_1}{\|\mathbf{v}_1\|} + \frac{\mathbf{v}_2}{\|\mathbf{v}_2\|} \right\|. \tag{30}$$

The smaller alignment index (SALI) is then given by

$$\mathrm{SALI}(n) = \min\left[\mathrm{PAI}(n), \mathrm{AAI}(n)\right]. \tag{31}$$

When the two deviation vectors become parallel, $\mathrm{PAI} \to 0$ and $\mathrm{AAI} \to \sqrt{2}$. On the other hand, when the two deviation vectors become antiparallel, $\mathrm{AAI} \to 0$ and $\mathrm{PAI} \to \sqrt{2}$. Thus, the SALI captures the information on whether the two deviation vectors tend to have the same direction, either parallel or antiparallel. For $d$-dimensional dynamical systems with $d > 2$, the two vectors tend to become parallel or antiparallel for chaotic orbits [82], i.e., the SALI tends to zero. On the other hand, for regular orbits, the SALI fluctuates around a positive value. For 2-dimensional maps, the SALI goes to zero for both chaotic and regular orbits. However, the SALI displays an exponential decay for chaotic orbits whereas for regular orbits, the decay follows a power-law. Thus, by measuring how fast the SALI goes to zero, it is possible to distinguish between regularity and chaos in 2-dimensional maps as well.

The generalization of the SALI, the generalized alignment index (GALI), considers the evolution of more than two deviation vectors, i.e., considering the matrix $Q \in \mathbb{R}^{d\times k}$, with $k \leq d$, whose columns are the $k$ deviation vectors $\{\mathbf{v}_i\}_{i=1,2,\ldots,k}$, the deviation vectors evolve under the linearized dynamics as $Q_n = J_n Q_{n+1}$. The $\mathrm{GALI}_k$ is proportional to the volume elements formed by the $k$ deviation vectors and it is defined as the norm of the wedge product of the $k$ deviation vectors [81]:

$$\mathrm{GALI}_k = \left\|\hat{\mathbf{v}}_1 \wedge \hat{\mathbf{v}}_2 \wedge \cdots \wedge \hat{\mathbf{v}}_k\right\|, \tag{32}$$

where $\hat{\mathbf{v}} = \mathbf{v}/\|\mathbf{v}\|$ denotes a unit vector. The GALI has been shown to accurately distinguish between regular and chaotic orbits, identify the dimensionality of the space of regular motion, and predict the diffusion of chaotic orbits. Additionally, $\mathrm{GALI}_2 \sim \mathrm{SALI}$. However, $\mathrm{GALI}_k$ involves the computation of $\binom{d}{k}$ determinants each time step. For systems with high dimensionality, the computation of GALI thus becomes impractical. For this reason, Antonopoulos and Bountis [83] introduced a new methodology that takes advantage of the linear dependence of the deviation vectors. They realized that both SALI and GALI are, in fact, measures of the linear independence of the deviation vectors. During the time evolution of chaotic orbits, the deviation vectors, which evolve under the linearized dynamics, tend to become linearly dependent over time, i.e., to become asymptotically aligned with each other. Therefore, Antonopoulos and Bountis defined the linear dependence index (LDI) as

$$\mathrm{LDI}_k = \prod_{i=1}^{k} \sigma_i, \tag{33}$$

where $\{\sigma_i\}_{i=1,2,\ldots,k}$ are the singular values of the matrix $Q \in \mathbb{R}^{d\times k}$ whose columns are the $k$ deviation vectors. The singular values are, in fact, a measure of the linear dependence of the deviation vectors. As long as all deviation vectors are linearly independent, $\sigma_i > 0$ for all $i$. As the system evolves in time and the deviation vector aligns with the most unstable direction, $\mathrm{LDI}_k \to 0$. Thus, at each time step, we compute the singular value decomposition (SVD) of $Q$:

$$Q = U\,\Sigma V^T, \tag{34}$$

where $U \in \mathbb{R}^{d\times d}$ is an orthogonal matrix whose columns are the left singular vectors, $\Sigma \in \mathbb{R}^{d\times k}$ is a diagonal matrix containing the non-negative singular values, i.e., $\sigma_i = \Sigma_{ii}$, and $V \in \mathbb{R}^{k\times k}$ is an orthogonal matrix whose columns are the right singular vectors.

To illustrate the computation of the LDI's, we consider a four-dimensional symplectic map, given by

$$\begin{aligned}
x_{n+1}^{(1)} &= x_n^{(1)} + x_n^{(2)} \bmod 2\pi,\\
x_{n+1}^{(2)} &= x_n^{(2)} - \epsilon_1 \sin\left(x_n^{(1)} + x_n^{(2)}\right) - \xi\left[1 - \cos\left(x_n^{(1)} + x_n^{(2)} + x_n^{(3)} + x_n^{(4)}\right)\right] \bmod 2\pi,\\
x_{n+1}^{(3)} &= x_n^{(3)} + x_n^{(4)} \bmod 2\pi,\\
x_{n+1}^{(4)} &= x_n^{(4)} - \epsilon_2 \sin\left(x_n^{(3)} + x_n^{(4)}\right) - \xi\left[1 - \cos\left(x_n^{(1)} + x_n^{(2)} + x_n^{(3)} + x_n^{(4)}\right)\right] \bmod 2\pi.
\end{aligned} \tag{35}$$

This map is composed of two coupled standard maps with parameters $\epsilon_1$ and $\epsilon_2$. The parameter $\xi$ is the coupling strength. For $\xi = 0$, the two maps behave independently. We set the parameters to $(\epsilon_1, \epsilon_2, \xi) = (0.5, 0.1, 0.001)$ and we consider two initial conditions. One

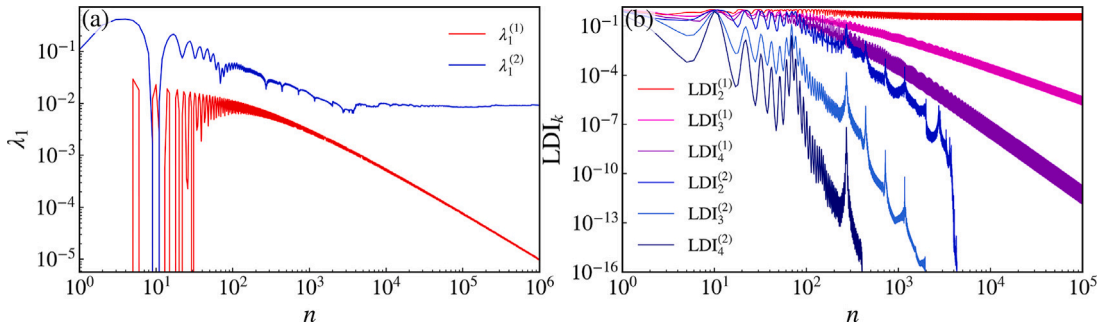


**Fig. 6.** Demonstration of the use of the `LDI` method of the `DiscreteDynamicalSystem` class of `pynamicalsys` for two different initial conditions of a four-dimensional symplectic map [Eq. (35)]. Execution times: (a) 9.58 s and (b) 9.91 s. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

regular: $(x_0^{(1)}, x_0^{(2)}, x_0^{(3)}, x_0^{(4)}) = (0.5, 0, 0.5, 0)$ [red curve in Fig. 5(a)] and one chaotic: $(x_0^{(1)}, x_0^{(2)}, x_0^{(3)}, x_0^{(4)}) = (3.0, 0, 0.5, 0)$ [blue curve in Fig. 5(a)].

To calculate the LDI's, we use the `LDI` method of the `DiscreteDynamicalSystem` class from `pynamicalsys`:

```
obj.LDI(u, total_time, k, parameters=None, return_history=False, sample_times=None, tol=1e-16,
    ↪ transient_time=None, seed=13)
```

Here, the integer `k` specifies the number of deviation vectors. Optional arguments include `return_history`, which determines whether to return the full-time evolution of the LDI's (default is `False`), `sample_times`, a list of times at which to sample the LDI (default is `None`), `tol`, the numerical tolerance for stopping the calculation (default is $10^{-16}$), `transient_time`, which allows skipping an initial transient phase (default is `None`), and `seed`, which sets the random seed to ensure reproducibility when generating the initial deviation vectors (default is `13`). The following code illustrates how to calculate the LDI$_k$ for different initial conditions [Fig. 6(b)]:

```
>>> import numpy as np
>>> from pynamicalsys import DiscreteDynamicalSystem as dds
>>> ds = dds(model="4D symplectic map")
>>> info = ds.info
>>> info["parameters"]  # Get the info about the order of the parameters
['eps1', 'eps2', 'xi']
>>> u = np.array([[0.5, 0.0, 0.5, 0.0], [3.0, 0.0, 0.5, 0.0]])  # Initial conditions
>>> parameters = [0.5, 0.1, 0.001]  # Define the parameters
>>> k = [2, 3, 4]  # The numbers of deviation vectors
>>> total_time = int(1e5)  # Total iteration time
>>> ldi = np.zeros((u.shape[0], total_time, len(k)))
>>> for i in range(len(k)):
...     for j in range(u.shape[0]):
...         ldi[j, :, i] = ds.LDI(u[j], total_time, k[i], parameters=parameters, return_history=True)
```

Fig. 6 shows the largest Lyapunov exponent and the LDI's for $k = 2$, $k = 3$, and $k = 4$, for two initial conditions, one regular and one chaotic, of the four-dimensional symplectic map [Eq. (35)]. The behavior of the LDI's for the regular orbit (red, pink, and purple) and the chaotic orbit (blue, light blue, and dark blue) differs fundamentally. For instance, LDI$_2$ for the regular orbit (red curve) oscillates around a positive value and does not converge to zero. On the other hand, the LDI$_2$ for the chaotic orbit (blue curve) decreases toward zero exponentially fast, reaching the machine precision of $10^{-16}$ for less than $4 \times 10^3$ iterations. The LDI$_3$ and LDI$_4$ for the chaotic orbits (light blue and dark blue curves, respectively) decreases to zero even faster. And even though the LDI$_3$ and LDI$_4$ for the regular orbit (pink and purple curves, respectively) also decrease to zero, they do so in a power law, which is much slower than the exponential decay of the chaotic orbit. Therefore, the LDI is an extremely fast and accurate chaotic indicator.

Antonopoulos and Bountis [83] demonstrated that LDI$_k \sim$ GALI$_k$ and thus LDI$_2 \sim$ SALI. The red, fuchsia, and purple curves in Fig. 6(b) are the LDI curves for $k = 2$, $k = 3$, and $k = 4$, respectively. These curves correspond to the regular initial condition. We observe that LDI$_2$ does not tend to zero, but it rather oscillates around a positive value, corroborating the statement of Antonopoulos and Bountis. The LDI$_3$ and LDI$_4$, on the other hand, decay to zero but following a power-law [81]. The LDI for the chaotic initial condition [shades of blue in Fig. 6(b)] all decay to zero exponentially, analogously to GALI [81].

Therefore, computing either GALI$_k$ or LDI$_k$ yields the same result with the difference of the LDI demanding less CPU time. Additionally, there is no real difference between computing SALI or LDI$_2$. However, in this case, SALI is computationally faster than

the $LDI_2$. Thus, if one is studying a two-dimensional map or for some reason is only interested in two deviation vectors, a faster alternative is to compute the SALI directly instead of the $LDI_2$:

```
obj.SALI(u, total_time, parameters=None, return_history=False, sample_times=None, tol=1e-16, transient_time=None, seed=13)
```

It is about 9 times computationally faster than the `LDI` method, which makes sense since to calculate the SALI, we only need to evaluate the norms of the deviation vectors while the LDI's require the singular value decomposition at each time step.

### 3.3. Weighted Birkhoff averages

The weighted Birkhoff average [84–89] is a powerful tool to classify the dynamics as regular or chaotic in a Hamiltonian system, without the problems introduced by the self-similar hierarchical structure, in which islands and chaotic orbits are mixed together in arbitrarily fine scales [87,88]. For a mapping of the form $\mathbf{v}_{n+1} = \mathbf{M}^n(\mathbf{v}_0)$. The Birkhoff average of some function $f(\mathbf{v})$ along this trajectory in phase space is defined as

$$B_N(f)(\mathbf{v}_0) = \frac{1}{N}\sum_{n=0}^{N-1} f\circ\mathbf{M}^n(\mathbf{v}_0). \tag{36}$$

The Birkhoff ergodic theorem [90] states that time averages of the function $f$ along the trajectory converge to the phase space averages as $N \to \infty$

$$\frac{1}{N}\sum_{n=0}^{N-1} f\circ\mathbf{M}^n(\mathbf{v}_0) \to \int f\,d\mu, \tag{37}$$

where $\mu$ is an invariant probability measure. However, convergence can be slow—scaling as $N^{-1}$ for quasiperiodic orbits and $N^{-1/2}$ for chaotic ones—due to edge effects from finite-time segments.

To mitigate this, a weighted Birkhoff average is introduced:

$$WB_N(f)(\mathbf{v}_0) = \sum_{n=0}^{N-1} w_{n,N} f\circ\mathbf{M}^n(\mathbf{v}_0), \tag{38}$$

where

$$w_{n,N} = \frac{g(n/N)}{\sum_{n=0}^{N-1} g(n/N)}, \tag{39}$$

with an exponential bump function

$$g(z) = \begin{cases} \exp\{-[z(1-z)]^{-1}\}, & \text{if } 0 < z < 1, \\ 0, & \text{otherwise.} \end{cases} \tag{40}$$

This choice ensures smooth vanishing at the endpoints and preserves regularity. When $f$, $g$, and $\mathbf{M}$ are $C^\infty$, the convergence becomes super-polynomial [86]:

$$\left| WB_N(f)(\mathbf{v}_0) - \int f\,d\mu \right| \leq C_m N^{-m}. \tag{41}$$

Notably, this acceleration applies only to regular orbits. Moreover, the convergence rate is largely independent of the choice of $f$, allowing the use of simple observables, such as $f(x,y) = \sin(x+y)$ [84] or $f(x) = \cos x$ [87].

To distinguish between regular and chaotic dynamics, one computes $2N$ iterations of $\mathbf{M}$ and compares:

$$\mathrm{dig} = -\log_{10}\left| WB_N(f)(\mathbf{v}_0) - WB_N(f)(\mathbf{v}_{N+1}) \right|. \tag{42}$$

A high value of dig indicates fast convergence and thus regular motion. Low values suggest chaos, though comparisons between chaotic orbits are not meaningful in the Lyapunov sense.

To calculate dig, we use the `dig` method of the `DiscreteDynamicalSystem` class from `pynamicalsys`:

```
obj.dig(u, total_time, parameters=None, func=lambda x: np.cos(2 * np.pi * x[:, 0], transient_time=None)
```

Here, `func` corresponds to the function $f$ discussed above. By default, it uses $f(x) = \cos(2\pi x)$ [Fig. 7(a)]. To use a different function, let us say $f(x) = \sin(2\pi x)$ [Fig. 7(b)] or $f(x,y) = \sin(2\pi(x+y))$ [Fig. 7(c)], we can pass a different lambda function to the `dig` method: `func`=lambda x: sin(2 * np.pi * x[:, 0]) or `func`=lambda x: sin(2 * np.pi * (x[:, 0] + y[:, 0])), for instance. The following code snippet illustrates the calculation of dig using these three functions $f$:

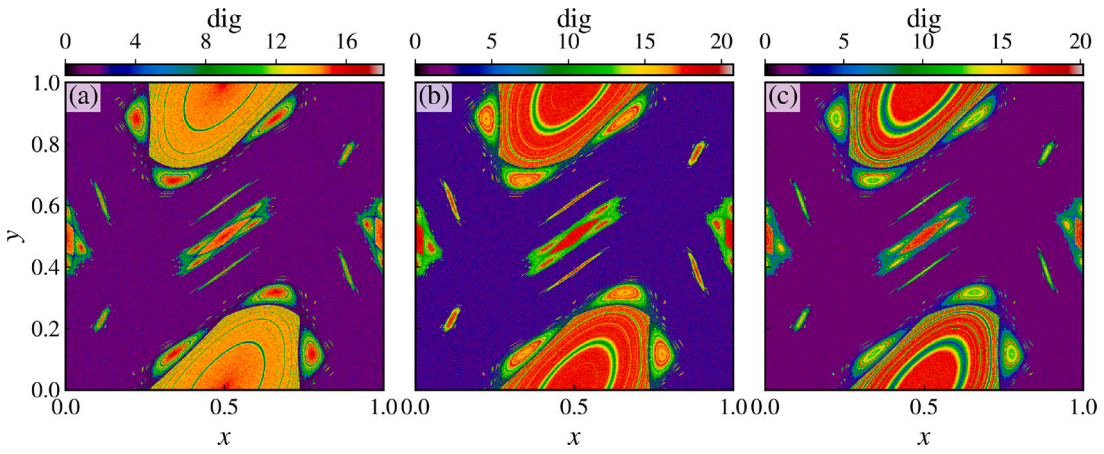


**Fig. 7.** Demonstration of the use of the `dig` method of the `DiscreteDynamicalSystem` class of `pynamicalsys` to calculate dig for a grid of uniformly distributed initial conditions for the standard map [Eq. (1)] with $k = 1.5$ and for the functions (a) $f(x) = \cos(2\pi x)$, (b) $f(x) = \sin(2\pi x)$, and (c) $f(x, y) = \sin[2\pi(x + y)]$. Execution times: (a) 8.66 s, (b) 8.76 s, and (c) 8.79 s.

```
>>> import numpy as np
>>> from pynamicalsys import DiscreteDynamicalSystem as dds
>>> ds = dds(model="standard map")
>>> grid_size = 1000  # Size of the grid in phase space
>>> x = np.linspace(0, 1, grid_size)  # Create the grid
>>> y = np.linspace(0, 1, grid_size)
>>> X, Y = np.meshgrid(x, y)
>>> u = np.array([X.flatten(), Y.flatten()]).T
>>> k = 1.5  # Parameter of the map
>>> total_time = 10000  # Total iteration time
>>> dig = [ds.dig(u[i], total_time, parameters=k) for i in range(u.shape[0])]
>>> function_b = lambda x: np.sin(2 * np.pi * x[:, 0])
>>> dig2 = [ds.dig(u[i], total_time, parameters=k, func=function_b) for i in range(u.shape[0])]
>>> function_c = lambda x: np.sin(2 * np.pi * (x[:, 0] + x[:, 1]))
>>> dig3 = [ds.dig(u[i], total_time, parameters=k, func=function_c) for i in range(u.shape[0])]
```

All functions $f$ are able to distinguish the regular and chaotic regions, the choice of this function is rather arbitrary, as long as it is sufficiently smooth, $C^\infty$, and maps into a finite-dimensional real vector space [86]. As expected, the quasiperiodic orbits exhibit a high value of dig, while chaotic orbits have a small value of dig. Since the weighted Birkhoff average does not improve the convergency of a chaotic orbit, we cannot compare two values of dig for two different chaotic orbits. For instance, we cannot say that an orbit with $\mathrm{dig} = 1$ is "more chaotic" than one with $\mathrm{dig} = 2$. Nevertheless, the weighted Birkhoff average can efficiently distinguish regular and chaotic regions, being simple and faster to compute. It helps the calculation of the dimension of the boundary of the chaotic sea and the islands [91], the classification of regions in a drift $\mathbf{E} \times \mathbf{B}$ model [92], and can also be extended to flows [89].

### 3.4. Recurrence time entropy

The recurrence plot (RP) was introduced in 1987 by Eckmann et al. [93] as a graphical representation of the recurrences of time series of dynamical systems in its $d$-dimensional phase space. For a given trajectory $\mathbf{x}_i \in \mathbb{R}^d$ $(i = 1, 2, \ldots, N)$ of length $N$, the $N \times N$ recurrence matrix is defined as

$$R_{ij} = H\left(\epsilon - \|\mathbf{x}_i - \mathbf{x}_j\|\right), \tag{43}$$

where $H(\cdot)$ is the Heaviside unit step function, $\epsilon$ is a small threshold, and $\|\mathbf{x}_i - \mathbf{x}_j\|$ is the distance between states $\mathbf{x}_i$ and $\mathbf{x}_j$ in phase space measured in terms of a suitable norm. The most commonly used norms are the Euclidean norm and the maximum (or supremum) norm, defined as

$$\begin{aligned} \|\mathbf{x}\|_2 &= \left(\sum_{i=1}^{d} |x_i|^2\right)^{1/2}, \\ \|\mathbf{x}\|_\infty &= \max_i \left(|x_i|\right), \end{aligned} \tag{44}$$

respectively. Both of these norms yield similar results. However, the maximum norm is computationally faster and it results in more recurrent points for a fixed threshold $\epsilon$ [94]. Therefore, the maximum norm is more commonly used.

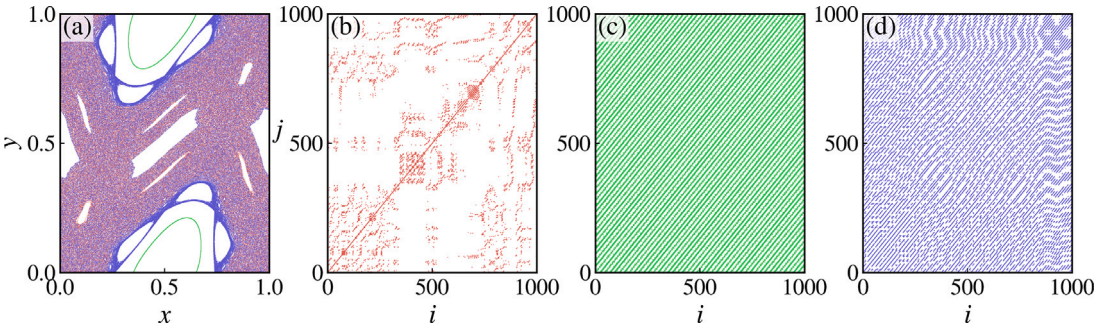


**Fig. 8.** Demonstration of the use of the `recurrence_matrix` method of the `DiscreteDynamicalSystem` class of `pynamicalsys` for three different initial conditions for the standard map [Eq. (1)] with $k = 1.5$. Execution times: (a) 22.9 ms and (b)–(d) 5.12 ms. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

The recurrent states are represented by the value 1 in the symmetric, binary recurrence matrix $\mathbf{R}$, whereas the nonrecurrent ones are represented by the value 0. Since it is numerically impossible to find *exactly* recurrent states, i.e., $\mathbf{x}_i = \mathbf{x}_j$, two states are said to be recurrent if they are sufficiently close to each other up to a distance $\epsilon$. The distance $\epsilon$ has to be carefully chosen. If $\epsilon$ is set too large, nearly every state is recurrent with every other state. On the other hand, if $\epsilon$ is chosen too small, there will be few recurrent states. Hence, a compromise has to be made between choosing $\epsilon$ as small as possible while resulting in a sufficient number of recurrent states. There is no general rule on choosing $\epsilon$. However, a few options have been proposed in the literature and each one of them has its own advantages and disadvantages depending on the purpose of the study. For instance, an alternative is to choose $\epsilon$ such that the recurrence point density, i.e., the recurrence rate, is fixed [95,96]. While this eliminates the issue of obtaining few recurrent states, it only shifts the problem to finding the optimal value of the recurrence rate. Another possibility is to define $\epsilon$ as a fraction of the time series standard deviation [94,97,98]. This has been proved efficient when distinguishing between dynamical regimes and analyzing dynamical transition [51,52,99,100].

Graphically, the recurrent states are represented by a colored dot and the recurrence matrix exhibits different patterns depending on the dynamics of the underlying system. These patterns are composed of mainly four distinct structures. They are (i) isolated recurrence points, (ii) diagonal lines, (iii) vertical lines, and (iv) white (non-recurrent) vertical lines. The recurrence matrix can be calculated using the `recurrence_matrix` method of the `DiscreteDynamicalSystem` class from `pynamicalsys`:

```
obj.recurrence_matrix(u, total_time, parameters=None, transient_time=None, **kwargs)
```

This method, given an initial condition $\mathbf{u} \in \mathbb{R}^d$, stored in the array `u`, returns the recurrence matrix $\mathbf{R} \in \mathbb{R}^{d\times d}$. The optional arguments include `metric`, which defines which norm to use. By default, the `recurrence_matrix` method uses the supremum norm (`metric="supremum"`). The method also supports the Euclidean norm `metric="euclidean"` and the Manhattan norm ($L_1$ norm) `metric="manhattan"`. The threshold $\epsilon$ setting is done via the `threshold` optional argument. By default, it is set to `threshold=0.1`. Additionally, the optional argument `threshold_std` determines whether to use the threshold in units of the standard deviation of the trajectory generated by the initial condition `u`. By default, it is set to `threshold_std=True`. In this case, `threshold=0.1` means 10% of the trajectory's standard deviation. Regarding the standard deviation, for a one-dimensional system, it is simply the standard deviation of the whole trajectory. However, for higher-dimensional systems, we define a standard deviation vector where each component contains the standard deviation of the corresponding component of the trajectory:

$$\boldsymbol{\sigma} = (\sigma_1, \sigma_2, \ldots, \sigma_d)^T, \tag{45}$$

where $\sigma_i = \text{std}(\mathbf{x}_i)$. Then, we define the standard deviation of the trajectory as the norm of the standard deviation vector $\boldsymbol{\sigma}$ (supremum, Euclidean, or Manhattan) using the optional argument `std_metric` (default is `std_metric="supremum"`).

The following code calculates the recurrence matrix for three different initial conditions for the standard map [Eq. (1)] using 10% of the trajectories' standard deviation as the threshold [Fig. 8(b)–(d)]:

```
>>> from pynamicalsys import DiscreteDynamicalSystem as dds
>>> ds = dds(model="standard map")
>>> u = [[0.05, 0.05], [0.35, 0.0], [0.42, 0.2]]  # Initial conditions
>>> k = 1.5  # Parameter of the map
>>> total_time = 1000  # Total iteration time
>>> recmats = [ds.recurrence_matrix(u[i], total_time, parameters=k) for i in range(len(u))]
```

Fig. 8(a) shows the trajectories for three distinct initial conditions, namely, (red) $(x, y) = (0.05, 0.05)$, (green) $(x, y) = (0.35, 0.0)$, and (blue) $(x, y) = (0.42, 0.2)$. The first one is an initial condition in the bulk of the chaotic sea and the second one is inside of an island. The third one, however, is an initial condition located at the sticky region around the period-6 islands. For the first iterations, this initial condition remains trapped inside the sticky region, hence a high density of blue points around the period-6 islands. Fig. 8(b)–(d) show the recurrence matrices for the initial conditions in Fig. 8(a). The recurrence matrix for the chaotic orbit [Fig. 8(b)] exhibits few diagonal lines while the recurrence matrix for the quasiperiodic orbit [Fig. 8(c)] is composed mainly of diagonal lines. The recurrence matrix for the sticky orbit [Fig. 8(d)], on the other hand, is not as regular as the quasiperiodic one but is more regular than the chaotic one. The recurrence matrix can also be calculated using only a given time series with the `recurrence_matrix` method of the `TimeSeriesMetrics` of `pynamicalsys` (see the documentation).

Therefore, the RPs of different dynamical processes are qualitatively different. In order to quantify the structures in RPs, several measures based on the diagonal and vertical lines have been proposed, such as the determinism and the laminarity, for instance. For a complete discussion on these and other measures, we refer the reader to Refs. [94,101–106] and references therein. Measures based on the white vertical lines, i.e., the vertical distance between two diagonal lines, have also been proposed [51,96,99,100]. The white vertical lines of an RP are an estimate of the return times of the corresponding trajectory [107–109], which is the time an orbit takes to return to the neighborhood of a previous point on the orbit. The RP for the chaotic orbit [Fig. 8(b)] shows no regularity in the vertical distances between the diagonal lines while the RP of the quasiperiodic process [Fig. 8(c)] consists diagonal lines with different distances between them. The vertical distances in the RP of the sticky orbit [Fig. 8(d)], on the other hand, are more regular than the chaotic case but it is not as regular as the quasiperiodic case. Thus, the RP of the sticky orbit has an intermediate complexity when compared to the RP of the quasiperiodic and chaotic ones.

Now, quasiperiodic processes yield three return times. This fact is stated by Slater's theorem [110–112]: for any irrational rotation, with rotation number $\omega$, over a unit circle, there are at most three different return times to a connected interval of size $\delta < 1$. Additionally, the third return time is the sum of the other two, and two of these three return times are consecutive denominators in the continued fraction expansion of $\omega$. Slater's theorem has been applied to study two-dimensional, area-preserving mappings [113,114] as well as to study the parameter space of a one-dimensional mapping [54]. It has also been employed to locate the position in phase space of invariant curves in Hamiltonian systems as well as to find the critical parameter values at which these curves break up [115–117].

Therefore, due to the intrinsic property of dynamical systems that quasiperiodic dynamics result in three recurrence times, measures based on the distribution of recurrence times are excellent alternatives to the characterization of the dynamics. We introduce, then, the recurrence time entropy (RTE), i.e., the Shannon entropy of the distribution of white vertical lines (recurrence times), estimated from the RP. Formally, the total number of white vertical lines of length $\ell$ is given by the histogram

$$P(\ell) = \sum_{i,j=1}^{N} R_{i,j-1} R_{i,j+\ell} \prod_{k=0}^{\ell-1} (1 - R_{i,j+k}), \tag{46}$$

such that the RTE is defined as [96]

$$\mathrm{RTE} = -\sum_{\ell=\ell_{\min}}^{\ell_{\max}} p(\ell) \log p(\ell), \tag{47}$$

where $\ell_{\min}$ ($\ell_{\max}$) is the length of the shortest (longest) white vertical line, $p(\ell) = P(\ell)/\mathcal{N}$ is the relative distribution of white vertical lines of length $\ell$ and $\mathcal{N}$ is the total number of white vertical line segments. The evaluation of the histogram given by Eq. (46) should be done carefully. Due to the finite size of an RP, the distribution of white vertical lines might be biased by the border lines, i.e., the lines that begin and end at the borders of an RP. These lines are cut short by the borders of the RP and their length is measured incorrectly. This influences measures such as the RTE [118]. To avoid such border effects, we exclude from the distribution the white vertical lines that begin and end at the border of the RP.

Originally, the RTE was introduced with no connections to RPs [119], and it has been shown that it provides a good estimate for the Kolmogorov–Sinai entropy [120]. The RTE has also been successfully applied to detect sticky regions in two-dimensional, area-preserving mappings [51,52] and to detect dynamical transitions in a fractional cancer model [121]. A periodic orbit, which has only one return time, yields RTE = 0. A quasiperiodic orbit, which has three return times, is characterized by a low value of RTE, whereas a chaotic orbit leads to a high value of RTE. Since the RP of a sticky orbit has an intermediate complexity, the RTE for such an orbit is smaller than the chaotic case but larger than the quasiperiodic case.

The RTE is the only recurrence-based measure implemented in `pynamicalsys`. That is not the aim of the package. We have chosen to implement the RTE due to its ability to detect different hierarchical levels in the islands-around-islands structure in two-dimensional, area-preserving maps and for being able to detect dynamical transitions [51,52,121]. For a complete package on RQAs, we refer the reader to the `pyunicorn` package [122].

To calculate RTE, we use the `recurrence_time_entropy` method of the `DiscreteDynamicalSystem` class of our package `pynamicalsys`:

```
obj.recurrence_time_entropy(u, total_time, parameters=None, transient_time=None, **kwargs)
```

It returns the RTE for the given initial conditions. The optional arguments include the `metric`, `threshold`, `threshold_std`, and `std_metric` we have discussed already. It also includes the `lmin`, which corresponds to the $\ell_{\min}$ value in Eq. (47) (default is `lmin=1`). The `recurrence_time_entropy` method can also return the final state (set `return_final_state=True`), the

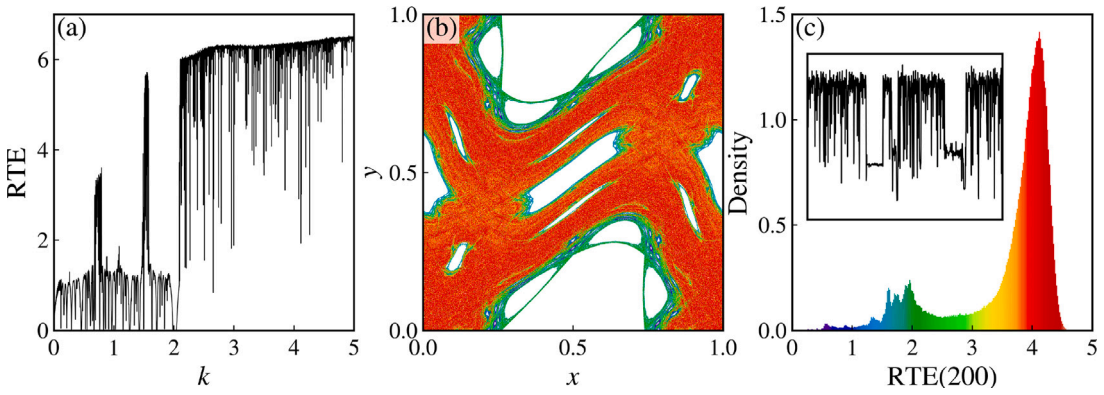


**Fig. 9.** (a) Demonstration of the use of the `recurrence_time_entropy` method of the `DiscreteDynamicalSystem` class of `pynamicalsys` to calculate the RTE as a function of the parameter $k$ of the standard map [Eq. (1)] and (b) and (c) demonstration of the use the `finite_time_recurrence_time_entropy` method of the `DiscreteDynamicalSystem` class of `pynamicalsys` to calculate the distribution of finite-time RTE for the standard map with $k = 1.5$. Execution times: (a) 19.2 s, (b) 804 ms, and (c) 35.2 s.

recurrence matrix used in the RTE calculation (set `return_recmat=True`), and the distribution of white vertical lines $p(\ell)$ (set `return_p=True`).

The following code demonstrates how to calculate the RTE for the standard map [Eq. (1)] as a function of the nonlinearity parameter $k$ [Fig. 9(a)]:

```
>>> import numpy as np
>>> from pynamicalsys import DiscreteDynamicalSystem as dds
>>> ds = dds(model="standard map")
>>> u = [0.5, 0.25]  # Initial condition
>>> k_range = (0, 5, 5000)  # Interval in k
>>> k = np.linspace(*k_range)  # Create the k values
>>> total_time = 5000  # Total iteration time
>>> rte = [ds.recurrence_time_entropy(u, total_time, parameters=k[i]) for i in range(k_range[2])]
```

Fig. 9(a) shows that the RTE distinguishes chaos and regularity successfully [cf. Fig. 5(a)]. And similarly to the LEs, the RTE can also be used to detect sticky orbits [51,52] by calculating the finite-time RTE, i.e., for a long trajectory of length $N$ (`total_time`), we calculate the RTE in windows of size $n \ll N$ (`finite_time`). We use the `finite_time_recurrence_time_entropy` method of the `DiscreteDynamicalSystem` class of `pynamicalsys`:

```
>>> obj.finite_time_recurrence_time_entropy(u, total_time, finite_time, parameter=None, return_points=False,
    ↪ **kwargs)
```

It returns an array of size $M = N/n$ with the value of the finite-time RTE for each time window. The optional arguments here include the `metric`, `threshold`, `threshold_std`, and `std_metric` and also the `return_points`. When set to `return_points=True`, the method also returns the initial conditions that generate the corresponding finite-time RTE value. The following code snippet illustrates the use of the `finite_time_recurrence_entropy` method to calculate the finite-time RTE distribution [Fig. 9(b) and (c)]:

```
>>> from pynamicalsys import DiscreteDynamicalSystem as dds
>>> ds = dds(model="standard map")
>>> u = [0.05, 0.05]  # Initial condition
>>> k = 1.5  # Parameter of the map
>>> total_time = 100000000  # Total iteration time
>>> finite_time = 200  # Finite time
>>> ftrte, points = ds.finite_time_recurrence_time_entropy(u, total_time, finite_time, parameters=k,
    return_points=True)
```

The distribution of the finite-time RTE is similar to the distribution of the finite-time Lyapunov exponent [Fig. 5(c)]. With the RTE, however, it is possible to detect more than one mode, i.e., the smaller mode in Fig. 5(c) is, in fact, composed of several smaller modes, as suggested by Harle and Feudel [123]. By inspecting the phase space positions that generate the smaller peaks in the distribution, we notice that smaller values of RTE are associated with inner levels in the hierarchical structure of islands-around-islands [51,52]. The inset in Fig. 9(c) corresponds to the same windows of the inset in Fig. 5(c). We notice the sharp drops in the value of the finite-time RTE, which leads to the multi-modal distribution we see in Fig. 9(c).

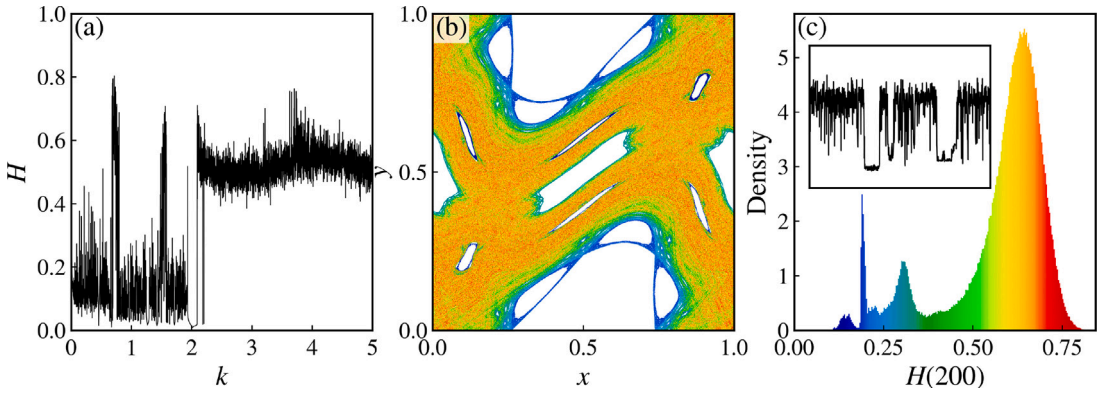


**Fig. 10.** (a) Demonstration of the use of the `hurst_exponent` method of the `DiscreteDynamicalSystem` class of `pynamicalsys` to calculate the Hurst exponent as a function of the parameter $k$ of the standard map [Eq. (1)] and (b) and (c) demonstration of the use the `finite_time_hurst_exponent` method of the `DiscreteDynamicalSystem` class of `pynamicalsys` to calculate the distribution of finite-time $H$ for the standard map with $k = 1.5$. Execution times: (a) 1 min 9 s, (b) 2.78 s, and (c) 2 min 11 s.

### 3.5. Hurst exponent

The Hurst exponent was introduced by H. E. Hurst in 1951 to model the cyclical patterns of the Nile floods [124] and is a key metric for assessing long-term memory in time series, revealing the extent to which data points are persistently or anti-persistently correlated over time. Several algorithms have been proposed to numerically estimate the Hurst exponent [125–128], however, the rescaled range analysis (R/S analysis) [124,129,130] has become the standard approach to its estimation. The approach is as follows: Given an 1-dimensional time series, $\mathbf{x} = (x_1, x_2, \dots, N)$ of length $N$, we divide the time series into $\kappa$ non-overlapping subseries of length $\ell$, $\{\mathbf{P}_k(\ell)\}_{k=1,2,\dots,\kappa}$, such that $\kappa = N/\ell$. For each subseries $\mathbf{P}_k(\ell)$, we calculate the mean $\mu_k(\ell)$ and the deviation from the mean:

$$\mathbf{D}_k(\ell) = \mathbf{P}_k(\ell) - \mu_k(\ell). \tag{48}$$

Next, we calculate the cumulative sum of the deviations as

$$Z_{i,k}(\ell) = \sum_{j=1}^{i} D_{i,k}(\ell), \tag{49}$$

for $i = 1, 2, \dots, \ell$, and the range of each cumulative sum subseries as

$$R_k(\ell) = \max \mathbf{Z}_k(\ell) - \min \mathbf{Z}_k(\ell). \tag{50}$$

Then, the mean of the rescaled ranges is calculated as

$$(R/S)_\ell = \left\langle \frac{R_k(\ell)}{S_k(\ell)} \right\rangle = \frac{1}{\kappa} \sum_{k=1}^{\kappa} \frac{R_k(\ell)}{S_k(\ell)}, \tag{51}$$

where $S_k(\ell)$ is the standard deviation of the subseries $\mathbf{P}_k(\ell)$. We then repeat the process considering a different value of $\ell$ and the Hurst exponent is estimated assuming a power-law relation between the rescaled ranges and the length of the subseries $\ell$:

$$(R/S)_\ell \sim \ell^H, \tag{52}$$

where $H$ is the Hurst exponent. The exponent $H$ is then estimated as the slope of the linear fit in the log–log plot of $(R/S)_\ell$ versus $\ell$, using the least squares method. Thus, $\ell \in [2, N/2]$.

This procedure considers only a 1-dimensional time series. For a $d$-dimensional data, we simply apply it to each component of the time series, yielding a Hurst exponent vector $\mathbf{H} \in \mathbb{R}^d$. Depending on the problem we can analyze only one component of the Hurst exponent vector or consider its mean, for instance. The Hurst exponent can be calculated for a given discrete dynamical system using the `hurst_exponent` method of the `DiscreteDynamicalSystem` class from `pynamicalsys`:

```
obj.hurst_exponent(u, total_time, parameters=None, wmin=2, transient_time=None)
```

Here, `wmin` is the minimum length of each subseries and it is by default set to `wmin=2`. This method returns the Hurst exponent vector **H** for higher-dimensional systems. To illustrate the calculation of the Hurst exponent, we consider the standard map [Eq. (1)] and calculate the Hurst exponent as a function of the parameter $k$ with initial condition $(x, y) = 0.5, 0.25$ [Fig. 10(a)]:

```
>>> import numpy as np
>>> from pynamicalsys import DiscreteDynamicalSystem as dds
>>> ds = dds(model="standard map")
>>> u = [0.5, 0.25]  # Initial condition
>>> k_range = (0, 5, 5000)  # Interval in k
>>> k = np.linspace(*k_range)  # Create the k values
>>> total_time = 5000  # Total iteration time
>>> H = [ds.hurst_exponent(u, total_time, parameters=k[i]) for i in range(k_range[2])]
```

The Hurst exponent also distinguishes regularity from chaos accurately [cf. Figs. 5(a) and 9(a)]. The regular regions yield a low value of $H$, while the chaotic regions exhibit a high value of $H$, just below 0.5, which makes sense as $H = 0.5$ indicates a random walk and a chaotic trajectory does not fall within this category.

Recently, Borin [131] has shown that the Hurst exponent is also an excellent tool for identifying and quantifying sticky orbits. By computing the Hurst exponent for a long chaotic trajectory of length $N$ (`total_time`) in windows of size $n \ll N$ (`finite_time`), it is possible to detect the different trappings around the hierarchical levels of the islands-around-islands structure. We use the `finite_time_hurst_exponent` method of the `DiscreteDynamicalSystem` class from `pynamicalsys`:

```
obj.finite_time_hurst_exponent(u, total_time, finite_time, parameters=None, wmin=2, return_points=False)
```

Similarly to the other finite-time methods (Lyapunov and RTE), this method returns an array of $M = N/n$ rows with $d$ columns, where $d$ is the dimension of the system. The optional argument `wmin` is the minimum length of each subseries and `return_points` when set to `True` tells the method to also return the initial conditions that generated the corresponding finite-time Hurst exponent value. The following code snippet illustrates the use of the `finite_time_hurst_exponent` method for the calculation of the finite-time Hurst exponent distribution:

```
>>> from pynamicalsys import DiscreteDynamicalSystem as dds
>>> ds = dds(model="standard map")
>>> u = [0.05, 0.05]  # Initial condition
>>> parameter = 1.5  # Parameter of the map
>>> total_time = 100000000  # Total iteration time
>>> finite_time = 200  # Finite time
>>> ftHE = ds.finite_time_hurst_exponent(u, total_time, finite_time, parameters=parameter)
>>> ftHE_avg = (ftHE[:, 0] + ftHE[:, 1]) / 2  # We use the average to calculate the distribution
```

The distribution of the finite-time Hurst exponent [Fig. 10(c)] is similar to the distribution of the finite-time RTE [Fig. 9(c)]. Both measures detect more than two modes and the Hurst exponent also distinguishes different trapping regions, as can be seen in Fig. 10(b). The inset shows the same sharp drops in the values of $H(200)$, which yields the multi-modal distribution.

## 4. Manifolds: the skeleton of the dynamics

The stable and unstable manifolds are invariant geometric structures associated with saddle points of periodic orbits in dynamical systems. Given a discrete dynamical system $\mathbf{x}_{n+1} = \mathbf{f}(\mathbf{x}_n)$, where $\mathbf{f} : \mathbb{R}^d \to \mathbb{R}^d$ is a smooth map, let $\mathbf{H} \in \mathbb{R}^d$ denote a hyperbolic fixed point (or a point on a hyperbolic periodic orbit) of $\mathbf{f}$. The stable manifold $W^s(\mathbf{H})$ is defined as the set of points $\mathbf{x}$ such that forward iterations under the map $\mathbf{f}$ asymptotically approach $\mathbf{H}$, i.e., $W^s(\mathbf{H}) = \left\{ \mathbf{x} \in \mathbb{R}^d \;\middle|\; \lim_{n\to\infty} \mathbf{f}^n(\mathbf{x}) = \mathbf{H} \right\}$. Analogously, the unstable manifold $W^u(\mathbf{H})$ consists of all points whose backward iterates converge to $\mathbf{H}$, that is, $W^u(\mathbf{H}) = \left\{ \mathbf{x} \in \mathbb{R}^d \;\middle|\; \lim_{n\to-\infty} \mathbf{f}^n(\mathbf{x}) = \mathbf{H} \right\}$ [132].

Both manifolds are invariant under the dynamics of $\mathbf{f}$. The manifolds cross each other transversely an infinite number of times, which generates an infinite but countable set of saddle points immersed in the chaotic region, this set is called chaotic saddle [133]. There are two types of intersections: homoclinic and heteroclinic. In the first case, the crossing is between the manifolds of the same hyperbolic point, while in the heteroclinic intersection, the manifolds of two distinct points cross each other [46]. Due to the invariance of these manifolds under the dynamics, all forward and backward iterates of a homoclinic point also belong to both manifolds. Consequently, the existence of a single homoclinic point implies the existence of an infinite number of such points. In Hamiltonian systems, where phase space volume is conserved (Liouville's theorem), the stable and unstable manifolds cannot intersect transversely just once. Instead, their intersections typically form intricate structures known as homoclinic tangles.

The relationship between the homoclinic tangle and the chaotic motion was given by Smale [5], using the nonattractive set call Smale horseshoe $\Lambda$, a complete description of this can be found in [55]. The set $\Lambda$, associated with a Smale horseshoe, has the following properties: (i) includes a countable set of periodic orbits with arbitrarily large periods; (ii) an uncountable set of bounded aperiodic orbits; and (iii) at least one dense orbit is present. Birkhof [134] and later Smale [4] demonstrated that every homoclinic point is an accumulation point of a family of infinitely many periodic orbits. Since the number of homoclinic points is infinite, it follows that in the neighborhood of each homoclinic point, there exist infinitely many periodic points. This implies the existence of

an integer $n$ such that the $n$th iterate of the map exhibits a horseshoe structure $\Lambda$. Consequently, any orbit with an initial condition sufficiently close to a homoclinic tangle will have chaotic behavior.

To numerically compute the stable and unstable manifolds of a given hyperbolic fixed point or hyperbolic periodic orbit [135], we first need to compute the eigenvectors of the Jacobian matrix. For two-dimensional maps, the Jacobian matrix evaluated at the hyperbolic fixed point has two eigenvalues, one larger than one and one less than one: $|\lambda_1| > 1 > |\lambda_2|$. The eigenvector with the largest eigenvalue, denoted by $\mathbf{v}_u$, represents the unstable direction, while the eigenvector with the smallest eigenvalue, denoted by $\mathbf{v}_s$, represents the stable direction. We select a large number of initial conditions uniformly distributed along the unstable eigenvector $\mathbf{v}_u$ and its negative counterpart $-\mathbf{v}_u$, within a distance $\delta \ll 1$ from the hyperbolic point. We then iterate these points forward in time, resulting in the unstable manifold of the hyperbolic fixed point (or hyperbolic periodic orbit). The stable manifold is obtained similarly. We distribute the initial conditions along the stable eigenvector $\mathbf{v}_s$ and its negative counterpart $-\mathbf{v}_s$ and iterate the initial conditions backward in time. This results in the stable manifold.

Assuming we know the coordinates of the hyperbolic fixed points or at least on point on a hyperbolic periodic orbit, we can compute the stable and unstable manifolds using the `manifold` method of the `DiscreteDynamicalSystem` class from `pynamicalsys`:

```
obj.manifold(u, period, parameters=None, delta=1e-4, n_points=100, iter_time=100, stability="unstable")
```

In this case, `u` represents the coordinates of the fixed point or periodic orbit, while `period` is the orbit's period. The argument `delta` defines the distance from the fixed point where the initial conditions will be distributed. The arguments `n_points` and `iter_time` specify the number of points along the eigenvector and the number of iterations for each point, respectively. Finally, `stability` indicates whether to calculate the stable or unstable manifold.

However, before calculating the manifolds, we need to find and classify the fixed points and periodic orbits. The standard map [Eq. (1)] has two fixed points: $(0, 0)$ and $(0.5, 0.0)$. We can analyze their stability analytically. The Jacobian matrix for the standard map is

$$J(x, y) = \begin{pmatrix} 1 + k\cos(2\pi x) & 1 \\ k\cos(2\pi x) & 1 \end{pmatrix}. \tag{53}$$

For area-preserving maps, such as the standard map, the stability of the fixed points and periodic orbits can be estimated using their residue [136,137]:

$$R(x, y) = \frac{1}{4}[2 - \mathrm{Tr}(J^p(x, y))], \tag{54}$$

where $J(x, y)$ is the Jacobian matrix, $\mathrm{Tr}(\cdot)$ is the trace, and $p$ is the period of the orbit. An elliptic orbit has $R \in (0, 1)$, and a parabolic orbit has $R = 0$ or $R = 1$. The orbit is hyperbolic otherwise. Therefore, for the fixed point $(0, 0)$, we have $R(0, 0) = -k/4$, which means the fixed point is hyperbolic regardless of the value of $k$. For the other fixed point, we have $R(0.5, 0) = k/4$. Therefore, for $k \in (0, 4)$, the fixed point is elliptic. We can verify that using the `classify_stability` method of the `DiscreteDynamicalSystem` class from `pynamicalsys`:

```
obj.classify_stability(u, period, parameters=None)
```

Here, `u` is a list with the coordinates of the periodic orbit, `parameters` is a list with the parameters of the system, and `period` is the period of the periodic orbit. The below code snippet demonstrates the use of the `classify_stability` method for the two fixed points we have discussed:

```
>>> from pynamicalsys import DiscreteDynamicalSystem as dds
>>> ds = dds(model="standard map")
>>> period = 1 # Period of the orbit
>>> u = [0, 0] # Fixed point
>>> stability = ds.classify_stability(u, period, parameters=1.5)
>>> stability["classification"], stability["eigenvalues"]
('saddle', array([3.18614066+0.j, 0.31385934+0.j]))
>>> stability = ds.classify_stability(u, period, parameters=5.0)
>>> stability["classification"], stability["eigenvalues"]
('saddle', array([6.85410197+0.j, 0.14589803+0.j]))
>>> u = [0.5, 0] # Fixed point
>>> stability = ds.classify_stability(u, period, parameters=1.5)
>>> stability["classification"], stability["eigenvalues"]
('elliptic (quasi-periodic)', array([0.25-0.96824584j, 0.25+0.96824584j]))
>>> u = [0.5, 0]
>>> stability = ds.classify_stability(u, period, parameters=5.0)
>>> stability["classification"], stability["eigenvalues"]
('saddle', array([-2.61803399+0.j, -0.38196601+0.j]))
```

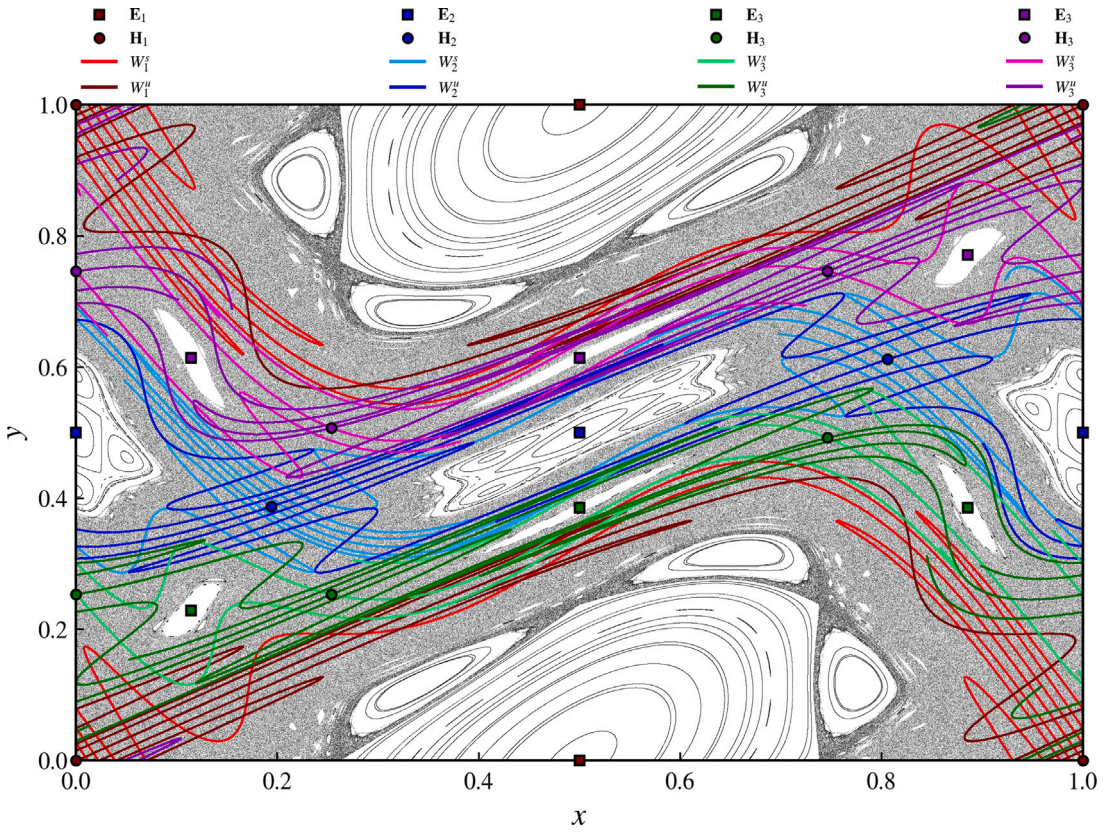


**Fig. 11.** Demonstration of the use of the `find_periodic_orbit` and `manifold` methods of the `DiscreteDynamicalSystem` class of `pynamicalsys` using the standard map [Eq. (1)] with $k = 1.5$ as an example. The execution time for finding the period 2 saddle point using the 2D search was 1.78 s. For the lower period 3 orbits (green points), the execution times were 228 ms for the center and 225 ms for the saddle using the 1D seach along a symmetry line. For the upper period 3 orbits (purple points), the execution times were 156 ms for the center and 151 ms for the saddle using the 1D seach along a symmetry line. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

Once we have determined the stability of the fixed points, we can calculate the manifolds as follows:

```
>>> saddle = [0, 0]
>>> period = 1
>>> k = 1.5
>>> n_points = 50000
>>> iter_time = 12
>>> wu = ds.manifold(saddle, period, parameters=k, n_points=n_points, iter_time=iter_time, stability="
    unstable")
>>> ws = ds.manifold(saddle, period, parameters=k, n_points=n_points, iter_time=iter_time, stability="stable
    ")
```

The manifolds of the hyperbolic fixed point are displayed in maroon (unstable) and red (stable) in Fig. 11.

Sometimes it is very difficult or even impossible to find the periodic orbits analytically, especially for higher periods. In this case, we can use the `find_periodic_orbit` method of the `DiscreteDynamicalSystem` class from `pynamicalsys` to perform a two-dimensional scan of a region in phase space where a periodic orbit might be present:

```
obj.find_periodic_orbit(grid_points, period, parameters=None, tolerance=1e-5, max_iter=1000,
    ↪ convergence_threshold=1e-15, tolerance_decay_factor=0.5, verbose=False, symmetry_line=None, axis=None
    ↪ )
```

This method locates a periodic orbit within a specified region of phase space using an iterative grid refinement strategy. The input `grid_points` is a three-dimensional array of shape `(grid_size_x, grid_size_y, 2)` representing a mesh of initial conditions in phase space. The method identifies periodic points by evolving each initial condition for a fixed number of steps defined by the `period` argument and checking whether the trajectory returns within a specified `tolerance` of its starting point. This tolerance acts as a numerical threshold, effectively defining the neighborhood within which a return is considered periodic. If periodic points are detected, their minimum and maximum positions are used to define a new, smaller region of phase space. The grid is then refined within this updated region, and the process repeats. The refinement continues until either the position of the orbit converges to within a specified `convergence_threshold` or the maximum number of iterations given by `max_iter` is reached. At each iteration, the `tolerance` is decreased according to the `tolerance_decay_factor` argument and if convergence is detected earlier, the process halts before reaching the maximum iteration count. By default, the method does not print any information to the user. However, by setting `verbose=True`, the method will print the convergence information at each refinement iteration.

Even though this is an efficient method, the search can be improved by using the symmetries of the map to reduce the 2D search to finding the root of a function of one variable [137]. If a map, denoted by $M$, is reversible, then it can be written as a product of two involutions:

$$M = I_2 \circ I_1, \tag{55}$$

where an involution is a map such that $I(I(\mathbf{x})) = \mathbf{x}$, i.e., $I^2 = I \circ I = 1$. The symmetry lines correspond to the fixed point sets of involutions, that is, the set of points where $I(\mathbf{x}) = \mathbf{x}$. For example, the standard can be written in the form of Eq. (55) using

$$\begin{aligned} I_1 &= \left(-x, y + \frac{k}{2\pi}\sin(2\pi x)\right), \\ I_2 &= (x - p, -p). \end{aligned} \tag{56}$$

A point $(x^*, y^*)$ is on the symmetry line of $I_1$ if it satisfies $I_1(x^*, y^*) = (x^*, y^*)$. We get

$$\begin{aligned} -x^* &= x^* \bmod 1, \\ \sin\left(2\pi x^*\right) &= 0. \end{aligned} \tag{57}$$

Thus, the fixed set of $I_1$ is $\{(x, y) \mid x = 0 \text{ or } 1/2 \forall y\}$. These are the vertical lines at $x = 0$ and $x = 0.5$. Using $I_2$ we find the horizontal line at $y = 0$. Note, however, that these are not the only symmetry lines. By using different involutions that satisfy Eq. (55), it is possible to find more symmetry lines [138].

Let us assume, for instance, that a symmetry line has the form $y(x) = f(x, \mathbf{p})$, where $\mathbf{p} = (p_1, p_2, \ldots)$ denotes the parameters of the system. Then, it is possible to define a function that takes on the coordinate $x$ and the system's parameters: `y = symm_line(x, parameters)` and pass it to the `find_periodic_orbit` method via `symmetry_line=symm_line`. It is also necessary to define the axis of the symmetry line. In this case, `axis=0`. If, however, the symmetry line has the form $x(y) = g(y, \mathbf{p})$, the procedure is analogous: the symmetry line function should be defined as `x = symm_line(y, parameters)` and `symmetry_line=symm_line` and `axis=1`.

For instance, to find the elliptic periodic orbit of period 2 of the standard map for $k = 1.5$ (see Fig. 1 for reference), we can perform the search along the vertical line $x = 0.0$:

```
>>> import numpy as np
>>> symmetry_line = lambda y, parameters: 0.0 * np.ones_like(y)
>>> k = 1.5
>>> period = 2
>>> y_range = (0.4, 0.6, 1000)
>>> y = np.linspace(*y_range)
>>> tolerance = 2 / len(y) # Initial tolerance for period detection
>>> periodic_orbit = ds.find_periodic_orbit(y, period, parameters=k, tolerance=tolerance, symmetry_line=
    symmetry_line, axis=1, verbose=False)
>>> periodic_orbit
array([0. , 0.5])
>>> stability = ds.classify_stability(periodic_orbit, period, parameters=k)
>>> stability["classification"], stability["eigenvalues"]
('elliptic (quasi-periodic)', array([-0.125-0.99215674j, -0.125+0.99215674j]))
```

Now, for the hyperbolic periodic orbit, we know that it is somewhere in between the two period 2 islands (Poincaré–Birkhoff theorem). So instead of trying to find another symmetry line to find these points, we perform a two-dimensional search within the region $(x, y) \in [0.1, 0.3] \times [0.3, 0.55]$:

```
>>> k = 1.5
>>> period = 2
>>> grid_size = 1000
>>> tolerance = 2 / grid_size # Initial tolerance for period detection
>>> x_range = (0.1, 0.3, grid_size) # Limits of the rectangular region
>>> y_range = (0.3, 0.5, grid_size)
```

```
>>> x = np.linspace(*xrange) # Generate a grid of points in the rectangular region
>>> y = np.linspace(*yrange)
>>> X, Y = np.meshgrid(x, y) # Create a meshgrid of points in the rectangular region
>>> grid_points = np.empty((grid_size, grid_size, 2)) # 3D array of points in the rectangular region
>>> grid_points[:, :, 0] = X
>>> grid_points[:, :, 1] = Y
>>> periodic_orbit = ds.find_periodic_orbit(grid_points, period, parameters=k, tolerance=tolerance)
>>> periodic_orbit
array([0.19397649, 0.38795298])
>>> stability = ds.classify_stability(periodic_orbit, period, parameters=k)
>>> stability["classification"], stability["eigenvalues"]
('saddle', array([4.09176343+0.j, 0.24439341+0.j]))
```

The calculation of the manifolds is similar to the period 1 case and Fig. 11 shows the fixed points and periodic orbits up to period three and also the stable and unstable manifolds of the saddles. The data of the figure has been generated using the methods we have discussed in this section. We have chosen, however, not to display all the code in this paper due to its size, and we refer the reader to the Supplementary Material for further details.

## 5. Escape analysis

In this section, we discuss the escape dynamics in discrete dynamical systems using `pynamicalsys`. In general, escape dynamics describes the statistical behavior of a collection of trajectories that leave a bounded region in phase space or escape through exits or holes present in the system. The analysis of the escape dynamics is essential to understanding the dynamical behavior of the system as it exhibits different statistical properties depending on the underlying dynamics. When two or more exits are present in the system, one can construct escape basins to analyze the uncertainty associated with the dynamical behavior. This approach is analogous to the basins of attraction in dissipative systems that have more than one attractor. Therefore, in general, we talk about escape basins in Hamiltonian systems, which preserve volume in phase space and thus cannot have attractors and basins of attraction in dissipative systems. This, however, does not prevent us from studying escape in dissipative systems.

For open systems, such as the Hénon map [53] or the Hénon–Heiles system [139], the definition of escape is natural: the trajectory has escaped when it leaves toward infinity [140–144]. In closed systems, however, it is necessary to introduce exits in the system. This is a classical approach, especially for Hamiltonian systems [52,145–149]. When the dissipative system has more than one attractor or the open system has more than one exit, the corresponding basins are divided by a basin boundary that can be either a smooth curve or a fractal curve. Smooth boundaries are related to regular dynamics, whereas fractal boundaries are a classical characteristic of chaotic dynamics. Additionally, fractal boundaries decrease the predictability of the final state [150–152], i.e., to which basin the initial condition belongs. For a complete review of fractal boundaries in dynamical systems, we refer the reader to Ref. [153] and references therein.

### 5.1. Survival probability

The escape times, i.e., the time it takes for a trajectory to escape through one of the exits also tell us important information regarding the underlying dynamics. Given the escape time, we compute the survival probability, $P(n)$, that corresponds to the fraction of initial conditions that have not escaped until the $n$th iteration. It is defined as,

$$P(n) = \frac{N_{\text{surv}}(n)}{M}, \tag{58}$$

where $M$ is the total number of initial conditions and $N_{\text{surv}}$ is the number of initial conditions that have not escaped until the $n$th iteration. In a hyperbolic and strongly chaotic system, the survival probability decays exponentially as $P(n) \sim \exp(-\kappa n)$, where $\kappa$ is known as the escape rate. However, in Hamiltonian systems, the existence of trapping regions leads to a slower escape rate: instead of an exponential decay, then the decay can be a power-law [154] or a stretched exponential [143].

To illustrate this feature, we consider the Leonel map [155,156], defined as

$$\begin{aligned} y_{n+1} &= y_n + \epsilon \sin\left(x_n\right), \\ x_{n+1} &= x_n + \frac{1}{|y_{n+1}|^{\gamma}} \mod 2\pi, \end{aligned} \tag{59}$$

where $\epsilon > 0$ is the nonlinearity parameter and $\gamma > 0$ controls the speed of the divergence of the $x$ coordinate in the limit $y \to 0$. This mapping has the interesting feature of exhibiting chaotic regions for small, but nonzero, perturbation values ($\epsilon \ll 1$) due to the divergent behavior of the second term in the $y$ equation. For small values of $y$, $1/|y|^{\gamma} \to \infty$ and $x_{n+1}$ and $x_n$ becomes uncorrelated, thus generating chaotic behavior. As $y$ increases, becomes slower and slower, and regular regions can be found in phase space [ Fig. 12(a)]. The transition from integrability ($\epsilon = 0$) to non-integrability ($\epsilon \neq 0$) has been investigated for this system and researchers have shown that the transition is characterized by a second-order phase transition. Additionally, the diffusion of chaotic orbits in the system is scaling invariant, i.e., the diffusion can be characterized by a homogeneous function of the parameters $\epsilon$ and $\gamma$. By introducing symmetric exits, located at the horizontal lines $y = \pm y_{\text{esc}}$, one can show that the survival probability is also scaling invariant as long as the phase space region $(x, y) \in [0, 2\pi] \times [-y_{\text{esc}}, y_{\text{esc}}]$ does not contain any stability islands [157].

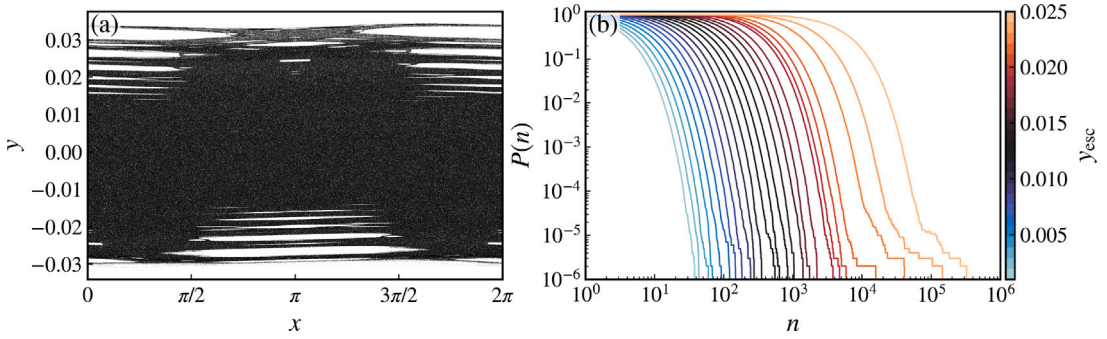


**Fig. 12.** Demonstration of the use of the `escape_analysis` with `escape="exiting"` and `survival_probability` methods of the `DiscreteDynamicalSystem` class of `pynamicalsys` for the Leonel map [Eq. (59)] for different escape regions defined by $y_{esc}$. Execution times (a) 378 ms and (b) 7 min 54 s.

The escape analysis can be done using the `escape_analysis` method from the `DiscreteDynamicalSystem` class of `pynamicalsys`:

```
obj.escape_analysis(u, max_time, exits, parameters=None, escape="entering", hole_size=None)
```

Here, the argument `max_time` determines the maximum iteration time to check for escape. If this time is reached and the trajectory has not escaped, the simulation is stopped. The argument `exits` defines the exits of the system. The shape of this argument depends on the `escape` argument. If `escape="entering"`, it means that the trajectories escape upon entering a predefined region, i.e., by reaching a hole in the system. However, if the escape happens when trajectories leave a predefined region, such as in our current example, then this argument should be set to `escape="exiting"`. The argument `hole_size` defines the size of the hole when `escape="entering"`.

Here, we focus on `escape="exiting"` and we discuss the other case in Section 5.2. The argument `exits` defines the boundaries of the $d$-dimensional phase-space box and has the format where each sublist specifies the lower and upper bounds along one coordinate axis. The `escape_analysis` method returns two values: `escape_side`, which indicates the side through which the trajectory escaped (with `-1` meaning no escape was detected), and `escape_time`, which is the time it took for the trajectory to escape (equal to `max_time` if no escape occurred). In general, escape can happen through any side of the box: 0 (1) corresponds to the left (right) side, 2 (3) to the bottom (top), and the pattern continues in higher dimensions. In the specific example considered here, the $x$ direction is periodic, so if we define `exits = [[0, 2 * np.pi], [-y_esc, y_esc]]`, escape can only occur in the $y$ direction. Escapes through the $x$ boundaries (i.e., `escape_side=0` for $x < 0$ or `escape_side=1` for $x > 2\pi$) are excluded due to the periodicity in $x$.

```
        exits = [[x_ini, x_end], [y_ini, y_end], [z_ini, z_end]]
```

The following code snippet demonstrates the use of the `escape_analysis` method. It calculates the escape and survival probability for different $y_{esc}$ for random initial conditions defined in the interval $(x, y) \in [0, 2\pi] \times [-1 \times 10^{-14}, 1 \times 10^{-14}]$ [Fig. 12(b)]:

```
>>> import numpy as np
>>> from pynamicalsys import DiscreteDynamicalSystem as dds
>>> ds = dds(model="leonel map")
>>> ds.info["parameters"]
["eps", "gamma"]
>>> eps, gamma = 1e-3, 1.0  # Define the parameters
>>> parameters = [eps, gamma]
>>> max_time = 1000000  # Maximum time
>>> num_ic = 1000000  # Number of initial conditions
>>> np.random.seed(13)  # Seed for reproducibility
>>> x_range = (0, 2 * np.pi, num_ic)  # Limits in x for the initial conditions
>>> y_range = (-1e-14, 1e-14, num_ic)  # Limits in y for the initial conditions
>>> x = np.random.uniform(*x_range)  # Create the random initial conditions
>>> y = np.random.uniform(*y_range)
>>> y_esc = np.logspace(np.log10(1e-3), np.log10(0.025), 25)  # Define the escape region
>>> x_esc = (0, 2 * np.pi)
>>> sp, times = [], []  # Empty list to store the survival probability and times
>>> for i in range(len(y_esc)):
...     exit = np.array([[x_esc[0], x_esc[1]], [-y_esc[i], y_esc[i]]])
...     escape = [ds.escape_analysis([x[j], y[j]], max_time, exit, parameters=parameters, escape="exiting")
     for j in range(num_ic)]
```

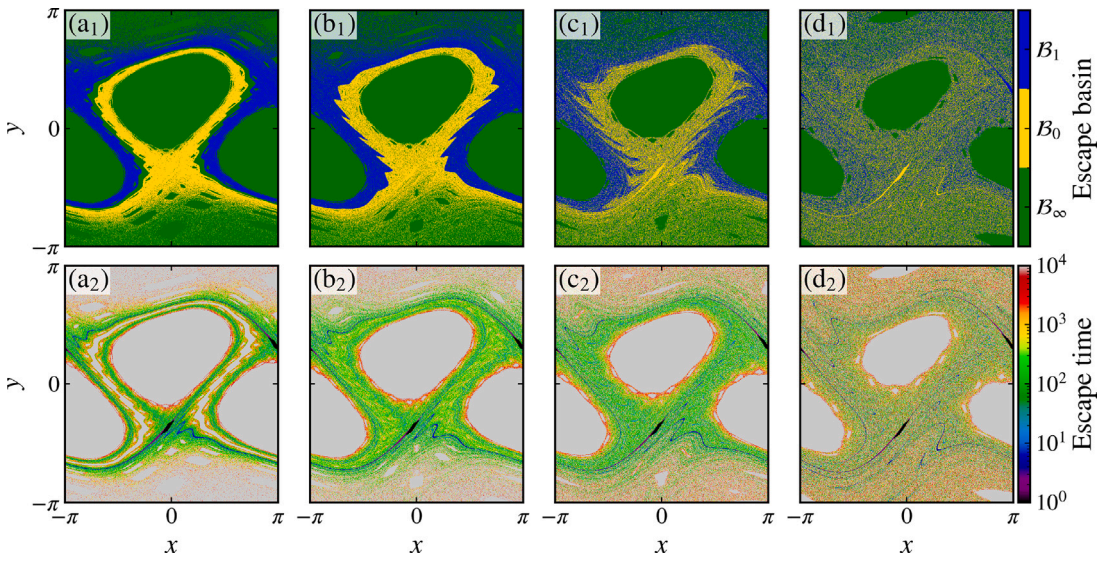


**Fig. 13.** Demonstration of the use of the `escape_analysis` method with `escaping="entering"` for the Weiss map [Eq. (60)] with different values of $k$, namely, (a) $k = 0.50$, (b) $k = 0.55$, (c) $k = 0.60$, and (d) $k = 0.70$. Execution time (total): 3 min 39 s.

```
...     escape = np.array(escape, dtype=np.int32)
...     time, survival_probability = ds.survival_probability(escape[:, 1], escapes[i, :, 1].max())
...     sp.append(survival_probability)
...     times.append(time)
```

The exponential decay is evident for almost all survival probability curves. The last three curves, i.e., the largest $y_{\text{esc}}$ values, exhibit a small power-law tail for large values of $n$. This is characteristic of systems that exhibit the stickiness effect.

### 5.2. Escape basins

Due to the strong sensitivity to initial conditions, chaotic systems often exhibit escape basins with an intertwined pattern and fractal basin boundaries. We exemplify this with a two-dimensional, area-preserving nontwist map, which describes the advection of passive scalars [158,159]. The map is defined by the following equations:

$$\begin{aligned} y_{n+1} &= y_n - k\sin(x_n), \\ x_{n+1} &= x_n + k\left(y_{n+1}^2 - 1\right) \bmod 2\pi, \end{aligned} \tag{60}$$

where, $k > 0$ is the nonlinearity parameter. This map is called nontwist due to the violation of the twist condition $\partial x_{n+1}/\partial y_n \neq 0$. Indeed, by calculating this derivative, we obtain

$$\frac{\partial x_{n+1}}{\partial y_n} = 2k y_{n+1}. \tag{61}$$

The twist condition is violated for $y_{n+1} = 0$. Nontwist systems exhibit nonmonotonic rotation number profiles, which leads to the phenomenon of degeneracy, i.e., two or more distinct stability islands with the same rotation number. For the Weiss map, due to the quadratic dependence of $x_{n+1}$ on $y_n$, there are two sets of islands with the same rotation number [160].

Additionally, nontwist systems exhibit a robust transport barrier, called the shearless curve. The shearless curve corresponds to a local extremum of the rotation number profile and it prevents global transport: it divides the phase space into two distinct and unconnected domains. Numerous studies on the breakup of the shearless curve have been done and we refer the reader to Refs. [149,161–168] and references therein for more details on two-dimensional, area-preserving nontwist systems.

Contrary to the approach we took on the previous section, now we introduce two holes of width 0.2 in the phase space of the system, centered at the points $(x, y) = (0.0, -1.1)$ and $(x, y) = (\pi - 0.1, 1.0)$ [160]. The `exits` argument now corresponds to the centers of the holes and the `hole_size` theirs the widths. Also, we must modify the `escape` argument: `escape="entering"`. But first, we need to define the mapping function as this system is not built-in within the `DiscreteDynamicalSystem` class:

```
>>> import numpy as np
>>> from numba import njit
```

```
>>> from pynamicalsys import DiscreteDynamicalSystem as dds
>>> @njit
>>> def weiss_map(u, parameters):
...     k = parameters[0]
...     x, y = u
...     y_new = y - k * np.sin(x)
...     x_new = (x + k * (y_new ** 2 - 1) + np.pi) % (2 * np.pi) - np.pi
...     return np.array([x_new, y_new])
>>> ds = dds(mapping=weiss_map, system_dimension=2, number_of_parameters=1)
```

To generate the escape basins in this case, we consider a $1000 \times 1000$ grid of initial conditions uniformly distributed on $(x, y) \in [-\pi, \pi]^2$ and iterate each one of them up to $10^4$ times. The following code snippet illustrates the calculation of the escape basins for four different values of $k$, namely, $k = 0.5$, $k = 0.55$, $k = 0.60$, and $k = 0.70$ (Fig. 13):

```
>>> import itertools
>>> centers = [[0.0, -1.1], [np.pi - 0.1, 1.0]]
>>> hole_size = 0.2
>>> grid_size = 1000
>>> ks = [0.5, 0.55, 0.60, 0.70]
>>> total_time = 10000
>>> x_range = (-np.pi, np.pi, grid_size)
>>> y_range = (-np.pi, np.pi, grid_size)
>>> X = np.linspace(*x_range)
>>> Y = np.linspace(*y_range)
>>> escapes = np.zeros((len(ks), grid_size, grid_size, 2))
>>> for i, k in enumerate(ks):
...     escape = []
...     for x, y in itertools.product(X, Y):
...         escape.append(ds.escape_analysis([x, y], total_time, centers, parameters=k, hole_size=size_exit)
    )
...     escape = np.array(escape).reshape(grid_size, grid_size, 2)
...     escapes[i, :, :, :] = escape
```

In this case, the first output of the `escape_analysis` method can only be -1, 0, or 1. If the initial condition has not reached one of the exits until $10^4$ iterations, it returns $-1$ and we color the point green. If, however, the initial condition has reached the first (second) exit, it returns 0 (1) and we color the point yellow (blue). The first row of Fig. 13 corresponds to the escape basins while the second row corresponds to the escape times. For $k = 0.5$, the shearless curve is still present in the phase space of the system, i.e., the blue and yellow basins do not mix. As we increase the perturbation, the shearless curve is broken and we observe an intermixing of the basins. To understand the effect of the parameter $k$ on the size of the basins, we can compute the basin stability [169], which is simply the relative proportion of each basin, Fig. 14(a). The stability of the green basin, which corresponds to the points that do not escape, for being in trapping regions or inside KAM islands, with the increase of the nonlinearity parameter these islands are broken, having the areas reduced leading to a decrease of the stability of this basin. The stability of the yellow and blue basins increases with the decrease of the regular regions, both basins present similar values for the basin stability, which indicate a strong mixing and intertwined basins.

To quantify the uncertainty of the final state caused by the fractal structure of the basin boundaries, we compute the basin entropy and boundary entropy as well as the dimension of the boundary using the uncertainty fraction method [150–152]. Regarding the basin entropy, the method was introduced by Daza et al. [170] and it consists of dividing the escape basin, characterized by the presence of $N_e$ distinguishable asymptotic states, into a fine mesh of $N \times N$ boxes. Each box contains a set of initial conditions that leads to a certain asymptotic state, which in our case can be escaping through one of the exits or never escaping. We label these asymptotic states from 1 to $N_e$. For each box $i$, we associate a probability $p_{ij}$ of the asymptotic state $j$ to be present in the box and define the Shannon entropy of the $i$th box as

$$S_i = -\sum_{j=1}^{n_i} p_{ij} \log p_{ij}, \tag{62}$$

where $n_i \in [1, N_e]$ is the number of asymptotic states present in the box. The total basin entropy is obtained by averaging over all boxes:

$$S_b = \frac{1}{N^2} \sum_{i=1}^{N^2} S_i. \tag{63}$$

The $S_b$ quantity is a measure of the complexity of the escape basin as a whole, with higher values indicating more complex basins. This methodology also allows us to compute the uncertainty of the final state associated with the basin boundary. To do this, we follow the same procedure to obtain Eq. (63), but considering only the $N_b$ boxes that contain more than one asymptotic state, i.e., those that intersect multiple basins. Then the basin boundary entropy is

$$S_{bb} = \frac{1}{N_b} \sum_{i=1}^{N^2} S_i = \frac{N S_b}{N_b}. \tag{64}$$

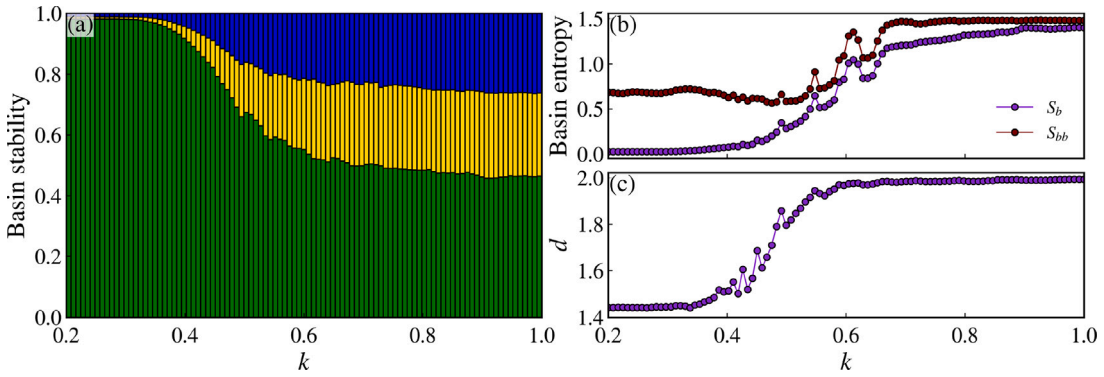


**Fig. 14.** (a) The basin stability and (b) and (c) the demonstration of the use of the `basin_entropy` and `uncertainty_fraction` methods of the `BasinMetrics` class of `pynamicalsys`. Execution time: 2 h 2 min 23 s.

A sufficient but not necessary condition for the boundary to be fractal is $S_{bb} > \log 2$. If we consider the logarithm in the base 2, this condition becomes $S_{bb} > 1$. Fractal boundaries are those with a non-integer value for their dimension. In our case, we have a two-dimensional basin and a smooth boundary is characterized by $d = 1$, while a fractal boundary exhibits $d > 1$. We compute the dimension of the boundary using the uncertainty fraction method [150–152]. Similarly to the basin entropy method, given an escape basin and an uncertainty $\epsilon$, for each point, we test whether small perturbations along each coordinate axis by $\epsilon$ remain in the same basin. Specifically, given a point $(x_i, y_i)$, we evaluate whether the perturbed points $(x_i \pm \epsilon, y_i)$ and $(x_i, y_i \pm \epsilon)$, one at a time, converge to the same asymptotic state as the reference point. If at least one of these four perturbed points belongs to a different basin, the reference point is classified as $\epsilon$-uncertain. The uncertainty fraction $f(\epsilon)$ is defined as the ratio of $\epsilon$-uncertain points to the total number of points in the basin. By varying $\epsilon$, we obtain the dependence of $f(\epsilon)$ on $\epsilon$.

For a smooth boundary, $f(\epsilon) \sim \epsilon$, whereas for fractal boundaries the uncertainty fraction scales with $\epsilon$ as a power law: $f(\epsilon) \sim \epsilon^{\alpha}$, where $\alpha$ is the uncertainty exponent. The uncertainty exponent, $\alpha$, and the dimension of the boundary, $d$, are related through the following equation [151]:

$$d = D - \alpha, \tag{65}$$

where $D$ is the dimension of the basin. In our case, $D = 2$ and a fractal boundary is characterized by $\alpha \in (0, 1)$.

Both entropies and the uncertainty fraction can be calculated using the `basin_entropy` and `uncertainty_fraction` methods, respectively, of the `BasinMetrics` class of `pynamicalsys`. To instantiate this class, you simply pass as an argument the two-dimensional array representing the basin you wish to quantify:

```
>>> basin = np.random.randint(1, 4, size=(1000, 1000))  # Example basin
>>> from pynamicalsys import BasinMetrics
>>> bm = BasinMetrics(basin)
```

The signatures of the methods are as follows:

```
obj.basin_entropy(n, log_base=np.e)
obj.uncertainty_fraction(x, y, epsilon_max=0.1, epsilon_min=None, n_eps=100)
```

In the `basin_entropy` method, the argument `n` specifies the size of each box in the $N \times N$ grid that covers the basin, and `log_base` sets the base of the logarithm used in the entropy calculation. This method returns a list containing the values of $S_b$ and $S_{bb}$.

In the `uncertainty_fraction` method, `x` and `y` are two-dimensional arrays that define the coordinates of the basin and must have the same shape as the basin. The arguments `epsilon_max` and `epsilon_min` specify the maximum and minimum values of the uncertainty $\epsilon$, in the same units as `x` and `y`. If `epsilon_min` is not provided, the method automatically determines an appropriate value based on the resolution of the basin. For instance, if the basin has a resolution of 0.001, setting `epsilon_min` below this value is not meaningful, and in general, it is recommended to omit this argument. The final argument, `n_eps`, determines the number of uncertainty values sampled between `epsilon_min` and `epsilon_max`. This method returns two arrays: the sampled values of $\epsilon$ and the corresponding values of the uncertainty fraction $f(\epsilon)$. The uncertainty exponent, and consequently the dimension, can be obtained by performing a least-squares fit of $\log \epsilon$ versus $\log f(\epsilon)$.

In Fig. 14(b) and (c), we show the dependence of the basin entropy, $S_b$, and basin boundary entropy $S_{bb}$, and the dimension $d$ on the parameter $k$. Immediately we notice that the boundary is fractal, i.e., $d > 1$, for all values of $k$ while the fractality condition of $S_{bb} > \log_2 2$ is only satisfied for $k > 0.6$. This is consistent with our expectations, as the latter condition is sufficient but not necessary for fractality, as previously discussed. We also note that, with a few exceptions where $S_b$, $S_{bb}$, and $d$ oscillate, all quantities increase with $k$. This is also consistent with our expectations. As the nonlinearity parameter increases, the size of the stability islands diminishes and the basins become more and more mixed (Fig. 13). Moreover, for $k \to 1$, the dimension $d$ tends to 2 and $S_b$ and $S_{bb}$ become closer and closer. This is an indication of riddled basins [171], i.e., basins in which every neighborhood of a point in one basin contains points belonging to other basins, making the system extremely sensitive to initial conditions. In terms of the uncertainty fraction, a value of $d \approx 2$ implies $\alpha \approx 0$. In other words, no matter how much we reduce the uncertainty $\epsilon$, the number of uncertain points does not decrease.

## 6. Continuous-time dynamical systems

### 6.1. System definition and basic simulation

All the examples so far were concerning discrete-time dynamical systems. In this section, we discuss how to use the `pynamicalsys` package to analyze continuous-time systems. Similarly to the discrete-time system definition, we create our dynamical system object using the `ContinuousDynamicalSystem` class:

```
>>> from pynamicalsys import ContinuousDynamicalSystem as cds
```

The class takes on five arguments: `model`, `equations_of_motion`, `jacobian`, `system_dimension`, and `number_of_parameters`. Analogously to the `DiscreteDynamicalSystem` class, it comes with a few built-in systems. To check all the built-in systems, run

```
>>> cds.available_models()
['lorenz system', 'henon heiles', 'rossler system', '4d rossler system']
```

Thus, to create an object of the Lorenz system, given by the following equations

$$\begin{aligned} \dot{x} &= \sigma(y - x), \\ \dot{y} &= x(\rho - z) - y, \\ \dot{z} &= xy - \beta z, \end{aligned} \tag{66}$$

where $\sigma$, $\rho$, and $\beta$ are the parameters of the system, you proceed as

```
>>> ds = cds(model="lorenz system")
```

In the case of continuous-time systems, one should also inform which numerical integrator one would like to use. Currently, the `ContinuousDynamicalSystem` class supports the 4th order Runge–Kutta method with fixed time step (RK4) and the Dormand–Prince method with adaptive time step, which it a Runge–Kutta method of order 5 with an embedded method of order 4 (RK45). A detailed discussion of these methods is provided in the Appendix B. To check the available integrators, run

```
>>> cds.available_integrators()
['rk4', 'rk45']
```

The numerical integrator is set using the `integrator` method from the `ContinuousDynamicalSystem` class:

```
obj.integrator(integrator, time_step=0.01, atol=0.000001, rtol=0.001)
```

where `integrator` is a string corresponding to the name of the integrator (e.g. `integrator="rk4"`), `time_step` is the time step used in the RK4 method and `atol` and `rtol` are the absolute and relative tolerance used in the RK45 method. In case the `integrator` method is not called, the `ContinuousDynamicalSystem` methods use the RK4 method with a time step of $\Delta t = 0.01$ by default.

The following code snippet demonstrates how to calculate the trajectory of the Lorenz system using both numerical integrators available:

```
>>> from pynamicalsys import ContinuousDynamicalSystem as cds
>>> ds = cds(model="lorenz system")
>>> ds.info["parameters"]
["sigma", "rho", "beta"]
>>> total_time = 80.0 # Total integration time
>>> u = [0., 0.1, 0.] # Initial condition (x, y, z)
```

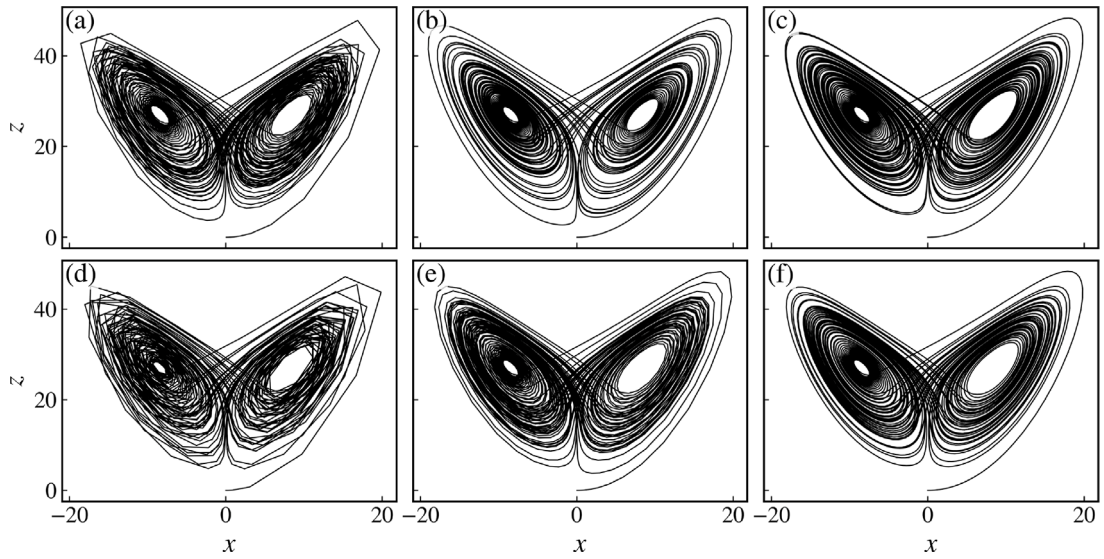


**Fig. 15.** Trajectories for the Lorenz system [Eq. (66)] with total integration time of 80 time units, $\sigma = 10$, $\rho = 28$, $\beta = 8/3$, and with initial condition $(x_0, y_0, z_0) = (0.0, 0.1, 0.0)$ using the (a)–(c) RK4 method and using the (d)–(f) RK45 method with different precisions, namely, (a) $\Delta t = 0.05$, (b) $\Delta t = 0.01$, (c) $\Delta t = 0.005$, (d) $(\text{atol}, \text{rtol}) = (10^{-6}, 10^{-3})$, (e) $(\text{atol}, \text{rtol}) = (10^{-8}, 10^{-5})$, and (f) $(\text{atol}, \text{rtol}) = (10^{-10}, 10^{-7})$. Execution time: 77.7 ms.

```
>>> parameters = [10, 28, 8/3] # (sigma, rho, beta)
>>> ds.integrator("rk4", time_step=0.01)
>>> trajectory = ds.trajectory(u, total_time, parameters=parameters)
>>> trajectory.shape
(8000, 4)
>>> ds.integrator("rk45", atol=1e-8, rtol=1e-5)
>>> trajectory = ds.trajectory(u, total_time, parameters=parameters)
>>> trajectory.shape
(2388, 4)
```

The `trajectory` method returns a two-dimensional array of shape `(sample_size, system_dimension + 1)`. The first column contains the time samples, while the remaining columns represent the system's coordinates at each corresponding time, i.e., `trajectory[:, 0] = t`, `trajectory[:, 1] = x(t)`, and so on. Fig. 15 shows example of trajectories for the Lorenz system [Eq. (66)] using both RK4 (top row) and RK45 (bottom row) methods with different time steps and absolute and relative tolerances.

### 6.2. User-defined systems

The `ContinuousDynamicalSystem` class also allows user defined systems. The equations of motion function signature should be `dudt = f(t, u, parameters)`, i.e., given the time `t`, the state vector `u`, and the parameters, the function returns the time derivative of the state vector. Let us demonstrate how one should proceed using the Rössler system as example. The equations of the system are

$$\begin{aligned} \dot{x} &= -(y + z), \\ \dot{y} &= x + ay, \\ \dot{z} &= b + z(x - c), \end{aligned} \tag{67}$$

where $a$, $b$, and $c$ are the parameters of the system. The function should then be defined as follows:

```
>>> import numpy as np
>>> from numba import njit
>>> @njit
>>> def rossler_system(t, u, parameters):
...     a, b, c = parameters
...     x, y, z = u
...     dx = -(y + z)
...     dy = x + a * y
...     dz = b + z * (x - c)
...     return np.array([dx, dy, dz])
```

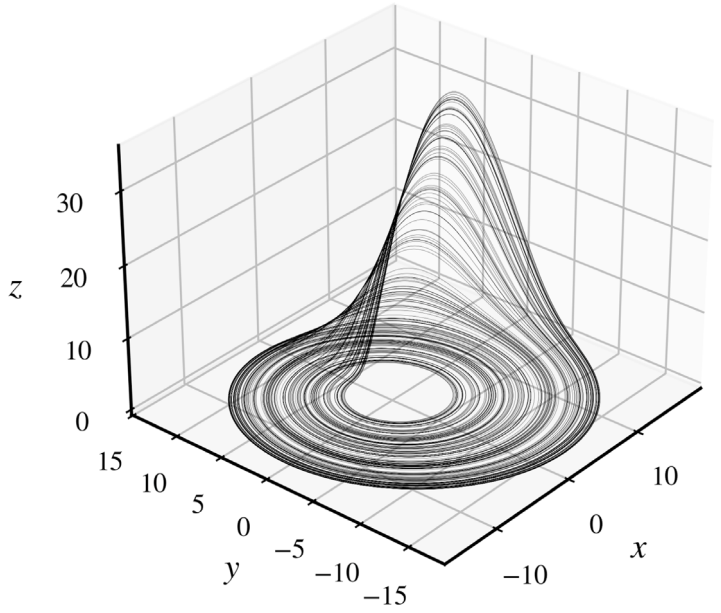


**Fig. 16.** Trajectory for the Rössler system [Eq. (67)] with total integration time of 2000 time units and transient time of 1000 time units. The parameters and initial condition are, respectively, $(a, b, c) = (0.15, 0.20, 10.0)$ and $(x_0, y_0, z_0) = (0.1, 0.1, 0.1)$. The trajectory was generated using the RK4 method with a time step of $\Delta t = 0.005$. Execution time: 245 ms.

To create the dynamical system object, instead of informing the `model` argument, we inform the `equations_of_motion`, `system_dimension`, and `number_of_parameters`:

```
>>> ds = cds(equations_of_motion=rossler_system, system_dimension=3, number_of_parameters=3)
```

and we can generate a trajectory, for example, for the Rössler system (Fig. 16):

```
>>> ds.integrator("rk4", time_step=0.005) # Define the integrator
>>> u = [0.1, 0.1, 0.1] # Initial condition
>>> total_time = 2000 # Total time
>>> transient_time = 1000 # Transient time to be discarded
>>> parameters = [0.15, 0.20, 10.0] # (a, b, c)
>>> trajectory = ds.trajectory(u, total_time, parameters=parameters, transient_time=transient_time)
>>> trajectory.shape
(200000, 4)
```

### 6.3. Chaotic indicators

The `ContinuousDynamicalSystem` class also offers numerical techniques to detect chaotic behavior. Let us begin with the LEs (cf. Section 3.1). Given a $d$-dimensional continuous-time dynamical system $\dot{\mathbf{x}} = \mathbf{f}(\mathbf{x}(t), t)$, where $\mathbf{f} : \mathbb{R}^d \to \mathbb{R}^d$ is a smooth vector field, the Lyapunov spectrum $\lambda_1 \geq \lambda_2 \geq \cdots \geq \lambda_d$ is defined under Oseledec's multiplicative ergodic theorem [55]. Let $J(\mathbf{x}) = D\mathbf{f}(\mathbf{x})$ be the Jacobian matrix of the vector field $\mathbf{f}$ evaluated at $\mathbf{x}$. The linearized dynamics around a trajectory $\mathbf{x}(t)$ is governed by the variational equation

$$\frac{\mathrm{d}\mathbf{y}}{\mathrm{d}t} = J(\mathbf{x}(t))\mathbf{y}, \tag{68}$$

where $\mathbf{y}(t)$ is a tangent vector. Let $\Phi(t, \mathbf{x}_0)$ denote the fundamental matrix solution to this equation so that $\Phi(t, \mathbf{x}_0)$ describes the evolution of tangent vectors along the trajectory starting at $\mathbf{x}_0$. Oseledec's theorem states that for almost every initial condition $\mathbf{x}_0$, the following limit exists:

$$\Lambda(\mathbf{x}_0) = \lim_{t\to\infty} \left[\Phi(t, \mathbf{x}_0)^T \Phi(t, \mathbf{x}_0)\right]^{1/2t}. \tag{69}$$

The LEs are then defined via the eigenvalues of $\Lambda(\mathbf{x}_0)$, and are expressed as

$$\lambda_i = \lim_{t\to\infty} \frac{1}{t} \log \left\| \Phi(t, \mathbf{x}_0)\mathbf{v}_i \right\|, \tag{70}$$

where $\mathbf{v}_i$ are the eigenvectors of $\Lambda(\mathbf{x}_0)$.

Similarly to the discrete-time case, the direct computation of the matrix $\Phi(t, \mathbf{x}_0)$ becomes numerically unstable for large $t$, as it becomes dominated by the most expanding direction. This problem is addressed by the reorthonormalization of the tangent vectors based on a QR decomposition: Given an initial orthonormal basis of tangent vectors encoded in a matrix $Q_0 \in \mathbb{R}^{d\times d}$, we evolve $Q_0$ along the variational equations. For a short time step $\Delta t$, the basis evolves according to

$$A_1 = \Phi(\Delta t, \mathbf{x}_0)Q_0. \tag{71}$$

We then perform a QR decomposition of $A_1$:

$$A_1 = Q_1 R_1, \tag{72}$$

where $Q_1$ is orthonormal and $R_1$ is upper triangular. The matrix $Q_1$ is the updated orthonormal basis, and $R_1$ contains the stretching information over the interval $[0, \Delta t]$. Repeating this process at each time step, we evolve

$$A_k = \Phi(\Delta t, \mathbf{x}(k-1)\Delta t)Q_{k-1}, \tag{73}$$

and compute

$$A_k = Q_k R_k. \tag{74}$$

After $n$ steps (total time $t = n\Delta t$), the accumulated linearized evolution is

$$\Phi(t, \mathbf{x}_0) = Q_n \mathcal{R}_n Q_0^{-1}, \tag{75}$$

where $\mathcal{R}_n = R_n R_{n-1} \cdots R_1$ is the product of the upper-triangular matrices.

Now consider the symmetric positive-definite matrix

$$\Phi(t, \mathbf{x}_0)^T \Phi(t, \mathbf{x}_0) = Q_0^{-T} \mathcal{R}_n^T \mathcal{R}_n Q_0^{-1}, \tag{76}$$

which is similar to $\mathcal{R}_n^T \mathcal{R}_n$, meaning that they share the same eigenvalues. In the long-time limit, the eigenvalues of $\mathcal{R}_n^T \mathcal{R}_n$ thus approximate those of $\Phi(t, \mathbf{x}_0)^T \Phi(t, \mathbf{x}_0)$, and therefore yield the LEs.

Since the diagonal elements $r_{ii}^{(j)}$ of each $R_j$ represent the local expansion or contraction along the $i$th orthonormal direction during the $j$th interval, the LEs can be computed as time averages of their logarithms [cf. Eq. (15)]:

$$\lambda_i = \lim_{n\to\infty} \frac{1}{n\Delta t} \sum_{j=1}^{n} \log \left| r_{ii}^{(j)} \right|. \tag{77}$$

The LEs for continuous-time systems are computed using the `lyapunov` method from the `ContinuousDynamicalSystem` class of `pynamicalsys`:

```
obj.lyapunov(u, total_time, parameters=None, transient_time=None, return_history=False, seed=13, log_base=np.e, method="QR", endpoint=True)
```

The non-optional arguments are the initial condition `u` and the total integration time `total_time`. If the system depends on parameter values, they can be passed using the `parameters` argument. The `transient_time` specifies the duration to discard before starting the Lyapunov exponent calculation. If `return_history=False`, the method returns only the final values of the LEs. When `return_history=True`, it returns their values at each sample time. The `seed` argument sets the initial tangent vectors randomly, and by default, the logarithm in Eq. (77) is the natural logarithm. You can change the logarithm base by specifying it with the `log_base` argument. The `lyapunov` method uses the modified Gram–Schmidt algorithm for the QR decomposition. For improved numerical stability, you can instead use Householder reflections by setting `method="QR_HH"`. Finally, setting `endpoint=True` ensures that the LEs at the final time are included in the returned history.

In the following, we calculate the LEs for the Lorenz [Eq. (66)] and Rössler [Eq. (67)] systems as well as for a 4-dimensional Rössler system [172], defined as

$$\begin{aligned} \dot{x} &= -(y+z), \\ \dot{y} &= x + ay + w, \\ \dot{z} &= b + xz, \\ \dot{w} &= -cz + dw, \end{aligned} \tag{78}$$

which displays hyperchaos for the parameters $(a, b, c, d) = (0.25, 3.0, 0.5, 0.05)$. We choose these three systems as examples for the Lyapunov exponent calculation following the seminal work of Wolf et al. [58]. The following code snippet demonstrates the use of the `lyapunov` method from the `ContinuousDynamicalSystem` class for the three mentioned systems which are built-in within the `ContinuousDynamicalSystem` class:

**Table 1**
The average LEs computed over 50 randomly chosen initial conditions using the `lyapunov` method of the `ContinuousDynamicalSystem` class in `pynamicalsys`, with a total integration time of $T = 2.0 \times 10^4$ and a transient time of $t = 1.0 \times 10^4$. Note that in the case of the Lorenz and Rössler systems, there is no fourth Lyapunov exponent as they are three-dimensional systems and this is denoted in the table by —. Execution time: 13 min 15 s.

| | Lorenz system | Rössler system | 4D Rössler system |
|---|---|---|---|
| $\lambda_1$ | 2.168 ± 0.002 | 0.128 ± 0.002 | 0.161 ± 0.004 |
| $\lambda_2$ | −0.0002 ± 0.0002 | 0.0001 ± 0.0002 | 0.031 ± 0.002 |
| $\lambda_3$ | −32.463 ± 0.002 | −14.139 ± 0.003 | −0.0002 ± 0.0003 |
| $\lambda_4$ | – | – | −35.8 ± 0.6 |

```
>>> from pynamicalsys import ContinuousDynamicalSystems as cds
>>> total_time = 2000 # Total integration time
>>> transient_time = 1000 # Discarded time
>>> ds = cds(model="lorenz system")
>>> u = [0., 0.1, 0.] # Initial condition
>>> parameters = [16, 45.92, 4] # (sigma, rho, beta)
>>> ds.lyapunov(u, total_time, parameters=parameters, transient_time=transient_time, log_base=2)
>>> array([ 2.16623420e+00, -5.95950415e-04, -3.24609067e+01])
>>> ds = cds(model="rossler system")
>>> u = [0.1, 0.1, 0.1] # Initial condition
>>> parameters = [0.15, 0.20, 10.0] # (a, b, c)
>>> ds.lyapunov(u, total_time, parameters=parameters, transient_time=transient_time, log_base=2)
array([ 1.28949638e-01,  1.49355076e-03, -1.41369173e+01])
>>> ds = cds(model="4d rossler system")
>>> u = [-20., 0, 0., 15.] # Initial condition
>>> parameters = [0.25, 3.0, 0.5, 0.05] # (a, b, c, d)
>>> ds.lyapunov(u, total_time, parameters=parameters, transient_time=transient_time, log_base=2)
array([ 1.72511105e-01,  3.63638738e-02,  4.73195157e-03, -3.88752036e+01])
```

Since the code snippet above does not specify the numerical integrator, the integration is performed with the RK4 method and a default time step of $10^{-2}$. To obtain more accurate estimates of the LEs, we repeat the computation for 50 randomly chosen initial conditions, where each component differs by at most 0.001. We choose to calculate the LEs using a base-2 logarithm to ensure consistency with Wolf's results reported in Table I of Ref. [58]. The integration is carried out over a total time of $T = 2.0 \times 10^4$, discarding a transient of $t = 1.0 \times 10^4$. Table 1 reports the mean LEs for the three systems studied, along with their standard deviations, showing that the values obtained with `pynamicalsys` are in close agreement with Wolf's results.

Another chaotic indicator of paramount importance is the generalization of the smaller alignment index (SALI) and generalized alignment index (GALI), the linear dependence index (LDI). For a discussion of these measures, see Section 3.2. Since both $\text{GALI}_k$ and $\text{LDI}_k$ yield the same conclusion, we have chosen to implement only the SALI and LDI in the `ContinuousDynamicalSystem` class:

```
obj.SALI(u, total_time, k, parameters=None, transient_time=None, return_history=False, seed=13, threshold=1e
    ↪ -16, endpoint=True)
obj.LDI(u, total_time, parameters=None, transient_time=None, return_history=False, seed=13, threshold=1e-16,
    ↪  endpoint=True)
```

The `threshold` parameter is the numerical tolerance for stopping the calculation (default is $10^{-16}$). To illustrate the computation of the SALI and LDI's, we use the Lorenz system as an example [173]. We fix the parameters at $(\sigma, \rho, \beta) = (10, 33.3, 8/3)$ and use the initial condition $(x, y, z) = (0.0, 0.1, 0.0)$. The following code snippet demonstrates the use of both `SALI` and `LDI` method from the `ContinuousDynamicalSystem` class:

```
>>> from pynamicalsys import ContinuousDynamicalSystem as cds
>>> total_time = 20000 # Total integration time
>>> transient_time = 10000 # Discarded time
>>> ds = cds(model="lorenz system") # Create the dynamical system object
>>> ds.integrator("rk45", atol=1e-10, rtol=1e-8) # Set the numerical integrator to RK45
>>> u = [0.0, 0.1, 0.0] # Initial condition
>>> parameters = [10, 33.3, 8/3] # (sigma, rho, beta)
>>> lyapunov = ds.lyapunov(u, total_time, parameters=parameters, transient_time=transient_time,
    return_history=True)
>>> sali = ds.SALI(u, total_time, parameters=parameters, transient_time=transient_time, return_history=True)
>>> ldi_2 = ds.LDI(u, total_time, 2, parameters=parameters, transient_time=transient_time, return_history=
    True)
>>> ldi_3 = ds.LDI(u, total_time, 3, parameters=parameters, transient_time=transient_time, return_history=
    True)
```

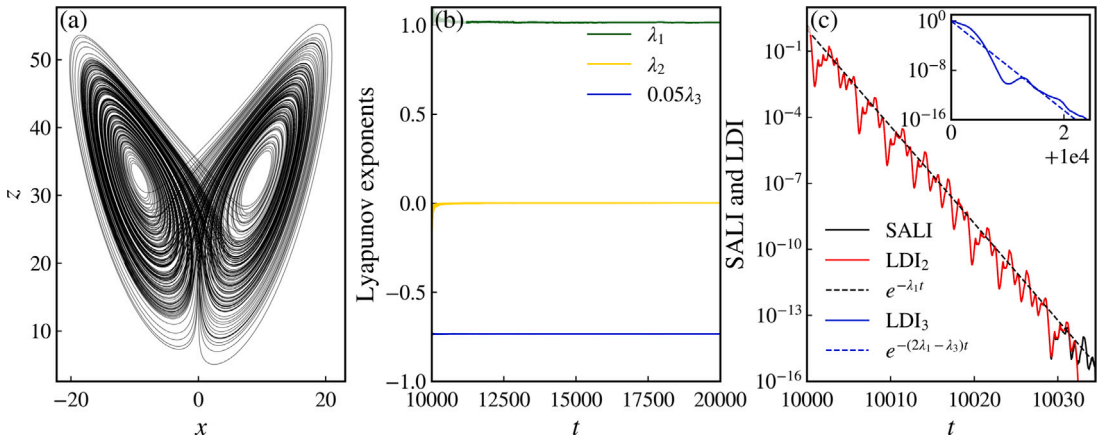


**Fig. 17.** (a) The trajectory for the Lorenz system [Eq. (66)] and demonstration of the use of the (b) `lyapunov` (c) and the `SALI` and `LDI` methods of the `ContinuousDynamicalSystem` class of `pynamicalsys`. Execution times: (a) 2.42 s, (b) 4.31 s, and (c) 2.33 s (SALI), 2.39 s ($LDI_2$), and 2.35 s ($LDI_3$).

Fig. 17 shows the time series of the LEs, along with the SALI, $LDI_2$, and $LDI_3$ indicators. As seen in the plot, SALI and $LDI_2$ closely overlap, while $LDI_3$ decays significantly faster toward zero. All three indicators exhibit exponential decay, following the exponential trends indicated by the dashed lines [81,173].

### 6.4. Basins of attraction

Multistability is a common property of nonlinear dynamical systems, characterized by the coexistence of two or more attractors within the same phase space [174–178]. Each attractor has an associated basin of attraction, defined as the set of initial conditions that asymptotically converge to it under the system dynamics. Mapping these basins provides essential information about the system's long term behavior, stability, and sensitivity to perturbations [179]. The geometry of the basins can be highly complex [153,180], ranging from smooth, well separated regions to intricately interwoven or even fractal structures, which reflect the presence of chaotic saddles and complex invariant manifolds. Computing and visualizing basins of attraction therefore serves as a fundamental tool for characterizing multistability, understanding transitions between attractors, and identifying regions of parameter space where qualitative changes in dynamics occur.

As a first example of a dynamical system exhibiting multistability, we consider the well-known Duffing oscillator, described by

$$\ddot{x} + \delta\dot{x} - \alpha x + \beta x^3 = \gamma\cos(\omega t), \tag{79}$$

where $\delta > 0$ is the damping coefficient, $\alpha \in \mathbb{R}$ and $\beta \in \mathbb{R}$ are the coefficients of the linear and nonlinear restoring forces, respectively, and $\gamma > 0$ and $\omega > 0$ denote the amplitude and frequency of the external driving force. Because of its double-well potential and the presence of periodic forcing, the Duffing oscillator naturally exhibits multiple coexisting attractors whose basins of attraction can display highly intricate structures, making it a paradigmatic system for studying multistability [181–183]. Multistability, however, is not unique to the Duffing oscillator. It has been observed in a wide variety of physical, chemical, and biological systems, including resonator systems [184], coupled chemical models [185], laser models [186], systems with time-delay [187], and neural networks [188], to cite a few.

Before we demonstrate how to use `pynalicalsys` to compute the basins of attraction, let us first discuss two other alternatives for visualizing attractors in phase space. In this section, we have computed the trajectories $\mathbf{x}(t)$ and plotted them directly in phase space (Figs. 15–17). However, continuous trajectories can be difficult to interpret, especially in higher dimensions. To simplify their analysis, it is common to reduce the dimensionality of the system by recording the state only when it intersects a chosen lower-dimensional surface, known as a Poincaré section. A Poincaré section is typically defined by a plane or hypersurface in phase space (for instance, $x = 0$ with $\dot{x} > 0$), and each time the trajectory crosses this surface, the phase-space coordinates are recorded. This produces a discrete set of points that captures the essential structure of the attractor.

A closely related idea is the stroboscopic map, which is particularly useful for systems with periodic forcing. Instead of sampling the trajectory when it crosses a geometric surface, the stroboscopic map samples the system state at regular time intervals synchronized with the driving period, for example at $t = nT$ where $T = 2\pi/\omega$ is the forcing period. This produces a discrete map of points in phase space that reveals the long-term dynamics in a way analogous to a Poincaré section but tailored to time-periodic systems. Both the Poincaré section and the stroboscopic map can be computed using `pynamicalsys`. In the following we discuss the computation of the stroboscopic map only and we leave the Poincaré section for Section 6.5. Thus, the stroboscopic map can be computed using the `stroboscopic_method` from the `ContinuousDynamicalSystem` class of `pynamicalsys`:

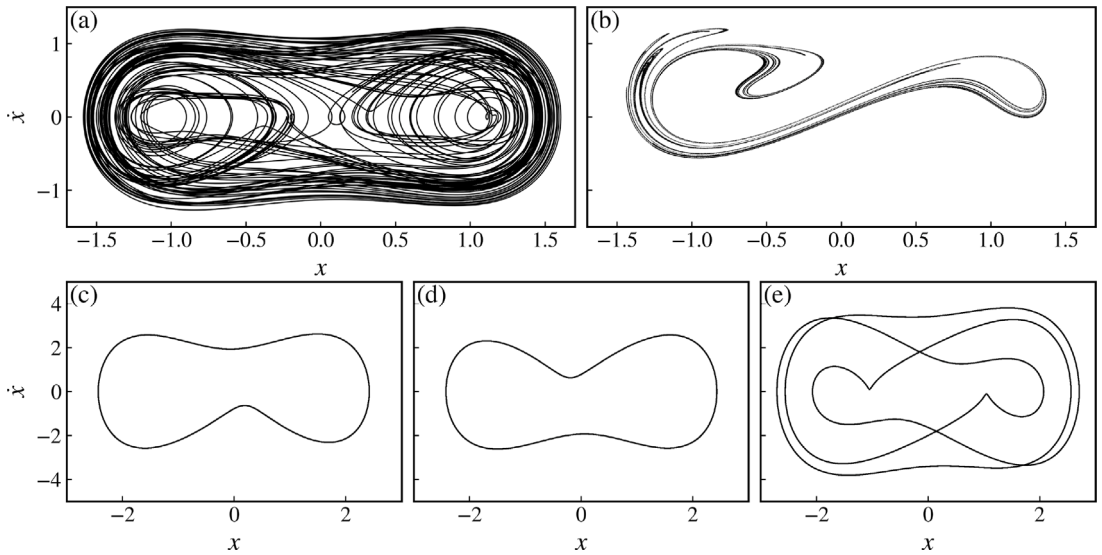


**Fig. 18.** (a) A chaotic trajectory for the Duffing oscillator [Eq. (79)] and (b) demonstration of the use of the `stroboscopic_map` method of the `ContinuousDynamicalSystem` class of `pynamicalsys` with $\gamma = 0.425$. Panels (c)–(e) show three coexisting periodic attractors for the Duffing oscillator with $\gamma = 3$. The other parameters are $\delta = 0.2$, $\alpha = \beta = 1$, and $\omega = 1.1$. Execution times: (a) 853 ms, (b) 1 min 30 s, (c)–(e): 1.88 s.

```
obj.stroboscopic_map(u, num_samples, sampling_time, parameters=None, transient_time=None)
```

Here, `u` can either be a single initial condition or an ensemble of initial conditions. The parameters `num_samples` and `sampling_time` determines the number of samples, i.e., the number of points in the map, and the interval between each point, respectively. The following code snipped demonstrate how to construct the stroboscopic map for the Duffing oscillator in a chaotic regime with parameters $\delta = 0.2$, $\alpha = \beta = 1$, $\gamma = 0.425$, and $\omega = 1.1$ and initial condition $(x_0, \dot{x}_0) = (1, 0)$ [Fig. 18(a) and (b)]:

```
>>> import numpy as np
>>> from pynamicalsys import ContinuousDynamicalSystem as cds
>>> ds = cds(model="duffing")  # Instanciate the class for the Duffing oscillator
>>> ds.info["parameters"]  # Obtain the order of the parameters
['delta', 'alpha', 'beta', 'gamma', 'omega']
>>> delta, alpha, beta, gamma, omega = 0.2, 1, 1, 0.425, 1.1  # Define the parameters
>>> parameters = [delta, alpha, beta, gamma, omega]  # Define the parameter list
>>> u0 = [1, 0]  # Initial condition
>>> T = 2 * np.pi / omega  # Forcing period
>>> sample_time = 100 * T  # Time to store the trajectory after the transient
>>> transient_time = 100 * T  # Transient time
>>> total_time = sample_time + transient_time  # Total integration time
>>> num_samples = 500000  # Number of points in the stroboscopic map
>>> trajectory = ds.trajectory(u0, total_time, parameters=parameters, transient_time=transient_time)
>>> strobe_map = ds.stroboscopic_map(u0, num_samples, T, parameters=parameters, transient_time=
    transient_time)
```

As for the multistable feature of the Duffing oscillator, for the parameters $\delta = 0.2$, $\alpha = \beta = 1$, $\gamma = 3$, and $\omega = 1.1$, there are three coexisting periodic attractors shown in Fig. 18(c)–(e). To construct the basin of attraction, we use the `basin_of_attraction` method from the `ContinuousDynamicalSystem` class of `pynamicalsys`:

```
obj.basin_of_attraction(u, num_intersections, parameters=None, transient_time=None, map_type="SM",
    ↪ section_index=None, section_value=None, crossing=None, sampling_time=None, eps=0.05, min_samples=1)
```

This method calculates either the stroboscopic map (`map_type="SM"`) or the Poincaré section (`map_type="PS"`) and uses the `DBSCAN` algorithm from `scikit-learn` [189] to group points that are close to each other in phase space, with `eps` defining the neighborhood radius and `min_sample` specifying the minimum number of points to form a cluster. Each cluster corresponds to a distinct attractor, and initial conditions whose trajectories end up in the same cluster are considered to belong to the same basin of attraction. Fig. 19(a) shows the basin of attraction for the Duffing oscillator [Eq. (79)] for the parameters $\delta = 0.2$, $\alpha = \beta = 1$, $\gamma = 3$, and $\omega = 1.1$. The purple region corresponds to the attractor shown in Fig. 18(c), the green region to the attractor in Fig. 18(d),

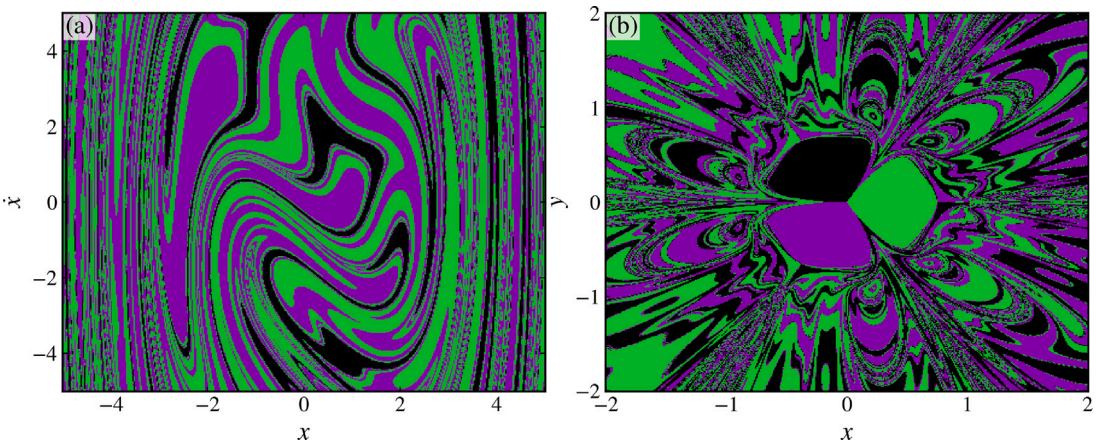


**Fig. 19.** Demonstration of the use of the `basin_of_attraction` method of the `ContinuousDynamicalSystem` class of `pynamicalsys` for the (a) Duffing oscillator [Eq. (79)] with parameters $\gamma = 3$, $\delta = 0.2$, $\alpha = \beta = 1$, $\gamma = 0.425$, and $\omega = 1.1$ and for the (b) magnetic pendulum [Eq. (80)]. Each color represents a different asymptotic attractor. Execution times: (a) 30 min 18 s and (b) 25 min 23 s. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

and the black region to the attractor in Fig. 18(e). The following code snippet demonstrates the use of the `basin_of_attraction` method:

```
>>> import numpy as np
>>> from pynamicalsys import ContinuousDynamicalSystem as cds
>>> ds = cds(model="duffing")  # Instanciate the class for the Duffing oscillator
>>> delta, alpha, beta, gamma, omega = 0.2, 1, 1, 3, 1.1  # Define the parameters
>>> parameters = [delta, alpha, beta, gamma, omega]  # Define the parameter list
>>> T = 2 * np.pi / omega  # Sampling time for the stroboscopic map
>>> transient_time = 100 * T  # Initial transient to discard
>>> num_samples = 100  # Number of points in the stroboscopic map
>>> grid_size = 500  # Number of points in the basin of attraction
>>> x_range, v_range = (-5, 5), (-5, 5)  # Ranges in x and v = dotx
>>> x = np.linspace(*x_range, grid_size)  # Uniformly distributed points in x
>>> v = np.linspace(*v_range, grid_size)  # Uniformly distributed points in v
>>> x, v = np.meshgrid(x, v, indexing="ij")  # Create a 2D mesh of points
>>> u0 = np.zeros((grid_size, grid_size, 2))  # Store the initial conditions in a 2D array
>>> u0[:, :, 0] = x
>>> u0[:, :, 1] = v
>>> u0 = u0.reshape(grid_size ** 2, 2)
>>> basin = ds.basin_of_attraction(u0, num_samples, parameters=parameters, transient_time=transient_time,
    sampling_time=T)
>>> basin.shape
(25000,)
>>> basin = basin.reshape(grid_size, grid_size)  # Reshape into a 2D array to match the grid
```

As a second example of a dynamical system with more than two coexisting attractors, we consider a magnetic pendulum [190–192]. The system consists of three identical magnets placed at the vertices of an equilateral triangle with unit edge length. A pendulum with an iron bob is suspended from above the triangle's center by a massless rod. The bob experiences gravity, attractive magnetic forces, and air drag. We assume the pendulum rod is long compared to the distance between the magnets, enabling a small-angle approximation. The interaction between the bob and each magnet follows an inverse-square law, as if the magnets were point sources. In dimensionless form, the equations of motion are

$$\begin{aligned}\ddot{x} &= \omega_0 x^2 - \alpha\dot{x} + \sum_{i=1}^{3} \frac{X_i - x}{D_i(X_i, Y_i, t)^3}, \\ \ddot{y} &= \omega_0 y^2 - \alpha\dot{y} + \sum_{i=1}^{3} \frac{Y_i - y}{D_i(X_i, Y_i, t)^3},\end{aligned} \tag{80}$$

where $(X_i, Y_i)$ are the coordinates of the $i$th magnet, $\omega_0$ is the natural frequency of oscillation, $\alpha$ is the damping coefficient due to air resistance, and $D_i(X_i, Y_i, t) = \sqrt{(X_i - x)^2 + (Y_i - y)^2 + d^2}$ and $d$ are the distances from the pendulum bob to the $i$th magnet and to the magnets plane, respectively. This system has three fixed-point attractors located at the positions of the magnets. The coordinates of the magnets are $(X_1, Y_1) = (1/\sqrt{3}, 0)$ (green), $(X_2, Y_2) = (-1/2\sqrt{3}, -0.5)$ (purple), and $(X_3, Y_3) = (-1/2\sqrt{3}, 0.5)$ (black).

To visualize their basins of attraction, we consider a grid of 500 × 500 initial conditions uniformly distributed over the $(x, y)$ plane with $\dot{x}_0 = \dot{y}_0 = 0$. The resulting basin of attraction is shown in Fig. 19(b) and the following code snippet demonstrates how to obtain it:

```
>>> import numpy as np
>>> from pynamicalsys import ContinuousDynamicalSystem as cds
>>> ds = cds(model="magnetic pendulum")  # Instanciate the class for the magnetic pendulum
>>> ds.info["parameters"]  # Obtain the order of the parameters
['omega', 'alpha', 'd', 'X1', 'Y1', 'X2', 'Y2', 'X3', 'Y3']
>>> omega, d, alpha = 0.5, 0.3, 0.2  # Define the parameters
>>> X1, Y1 = 1 / np.sqrt(3), 0.0
>>> X2, Y2 = - 1 / (2 * np.sqrt(3)), -0.5
>>> X3, Y3 = - 1 / (2 * np.sqrt(3)), 0.5
>>> parameters = [omega, alpha, d, X1, Y1, X2, Y2, X3, Y3]
>>> T = 1 / omega  # Sampling time
>>> transient_time = 100 * T  # Initila time to discard
>>> num_samples = 50  # Number of samples on the stroboscopic map
>>> grid_size = 500  # Size of the basin of attraction
>>> x_range, y_range = (-2, 2), (-2, 2)  # Ranges in x and y
>>> x = np.linspace(*x_range, grid_size)  # Uniformly distribuited points in x
>>> y = np.linspace(*y_range, grid_size)  # Uniformly distribuited points in x
>>> x, y = np.meshgrid(x, y, indexing='ij')  # Create a 2D mesh of points
>>> u0 = np.zeros((grid_size, grid_size, 4))  # Store the initial conditions in a 2D array
>>> u0[:, :, 0] = x
>>> u0[:, :, 2] = y
>>> u0 = u0.reshape(grid_size ** 2, 4)
>>> basin = ds.basin_of_attraction(u0, num_samples, parameters=parameters, transient_time=transient_time,
    sampling_time=T)
>>> basin.shape
(25000,)
>>> basin = basin.reshape(grid_size, grid_size)  # Reshape into a 2D array to match the grid
```

### 6.5. Hamiltonian systems

Up to this point in this section, we have considered continuous-time dynamical systems of the form $\dot{\mathbf{x}} = \mathbf{f}(\mathbf{x}, t)$, where the evolution of the state $\mathbf{x}$ is governed by the vector field $\mathbf{f}$. While this formulation is general and valid for any continuous-time system, depending on the nature of the governing equations, different subclasses arise. A particularly important subclass is formed by Hamiltonian systems, which describe the evolution of conservative mechanical systems in terms of the $N$-dimensional generalized canonical coordinates $\mathbf{q} = (q_1, q_2, \ldots, q_N)$ and momentum $\mathbf{p} = (p_1, p_2, \ldots, p_N)$ for a $2N$-dimensional Hamiltonian system with $N$ degrees of freedom. These systems are described by a scalar function $H(\mathbf{p}, \mathbf{q}, t)$, the Hamiltonian, and the time evolution of the state vector $(\mathbf{p}, \mathbf{q})$ is given by Hamilton's equations:

$$\begin{aligned} \dot{q}_i &= \frac{\partial H}{\partial p_i}, \\ \dot{p}_i &= -\frac{\partial H}{\partial q_i}, \end{aligned} \tag{81}$$

with $i = 1, 2, \ldots, N$. For an autonomous Hamiltonian, i.e., a Hamiltonian function that does not depend explicitly on time ($\partial H / \partial t = 0$), $H \equiv H(\mathbf{p}, \mathbf{q})$ is a conserved quantity and if the kinetic energy $T$ of the system is quadratic on the velocities, $H$ is the total mechanical energy of the system: $H = T + V$, where $V$ is the potential energy. One important feature of Hamiltonian systems is that it has a symplectic structure [55] thus preserving the infinitesimal phase space volume. Following that, Liouville's theorem states that the phase space volume of a closed surface is preserved under the time evolution in a Hamiltonian system [46,55]. Therefore, the existence of attractors and repellers are prohibited in a Hamiltonian system.

Classical examples of Hamiltonian systems include the three-body problem [2] (or any gravitational interaction problem), which exhibits complex gravitational interactions and chaotic behavior, the double pendulum [193,194], known for its sensitive dependence on initial conditions, the Atwood machine [195,196], a simple mechanical system demonstrating energy conservation, and the spring pendulum [197,198], which couples linear and rotational motion and exhibits nonlinear dynamics.

Due to the symplectic structure of Hamiltonian systems, special care must be taken when numerically integrating their equations of motion. Conventional integration schemes, such as the classical fourth-order Runge–Kutta (RK4) method, do not preserve the symplectic structure. Although RK4 can provide high local accuracy for short time intervals, its lack of symplecticity leads to a systematic drift in conserved quantities (such as the Hamiltonian) over long integration times. This drift manifests as artificial energy gain or loss and distortion of the phase-space structure. To correctly capture the behavior of Hamiltonian systems, the equations of motion must be integrated using methods designed to preserve the symplectic form ensuring that invariants such as the Hamiltonian are conserved over long integration times. These methods are called symplectic integrators and, in this paper, we consider the second-order velocity-Verlet integrator [199] (VV2) and the fourth-order Yoshida integrator [200] (SVY4), which uses the VV2 scheme as its underlying second-order step, to integrate both the equations of motion and the variational equations [201]. A detailed discussion of these methods is provided in the Appendix B.

The `HamiltonianSystem` class is analogous to the `ContinuousDynamicalSystem` class. To create our Hamiltonian system object, we import the class using

```
from pynamicalsys import HamiltonianSystem as HS
```

and instantiate the class by informing either the built-in model, via the `model` parameter, or by informing the gradient of $T$ with respect to $\mathbf{p}$ ($\partial T/\partial \mathbf{p}$) and the gradient of $V$ with respect to $\mathbf{q}$ ($\partial V/\partial \mathbf{q}$) via the `grad_T` and `grad_V` parameters, respectively. Additionally, we need to inform the number of degrees of freedom (`degrees_of_freedom`) and the number of parameters (`number_of_parameters`). For the Lyapunov exponents, SALI, and LDI calculations, it is also necessary to inform the Hessian matrix of $T$ and $V$ ($\partial^2 T/\partial \mathbf{p}^2$ and $\partial^2 V/\partial \mathbf{q}^2$) via the `hess_T` and `hess_V` parameters, respectively. The following code snippet illustrates how to list the available built-in models and integrators:

```
    >>> HS.available_models()
    ['henon heiles']
    >>> HS.available_integrators()
    ['vv2', 'svy4']
```

Here, `'vv2'` corresponds to the second-order velocity-Verlet integrator, while `'svy4'` refers to the fourth-order Yoshida integrator, which uses the velocity-Verlet scheme as its underlying second-order step. The numerical integrator is set using the `integrator` method from the `HamiltonianSystem` class:

```
obj.integrator(integrator, time_step=0.01)
```

In case the `integrator` method is not called, the `HamiltonianSystem` class methods use the SVY4 method with a time step of $\Delta = 0.01$ by default.

Although the Hénon–Heiles Hamiltonian [202] is provided as a built-in option within the `HamiltonianSystem` class, it is instructive to demonstrate how an user can define a custom Hamiltonian system. The Hénon–Heiles model is a two-degree-of-freedom system given by

$$H(x, y, p_x, p_y) = \frac{1}{2}\left(p_x^2 + p_y^2\right) + \frac{1}{2}\left(x^2 + y^2\right) + x^2 y - \frac{y^3}{3}. \tag{82}$$

This Hamiltonian is separable, $H = T(p_x, p_y) + V(x, y)$, with gradients of the kinetic and potential energy terms given by

$$\frac{\partial T}{\partial \mathbf{p}} = \begin{pmatrix} p_x \\ p_y \end{pmatrix}, \qquad \frac{\partial V}{\partial \mathbf{q}} = \begin{pmatrix} x + 2xy \\ y + x^2 - y^2 \end{pmatrix}. \tag{83}$$

The Hessians of the kinetic and potential energy terms are then

$$\frac{\partial^2 T}{\partial \mathbf{p}^2} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}, \qquad \frac{\partial^2 V}{\partial \mathbf{q}^2} = \begin{pmatrix} 1 + 2y & 2x \\ 2x & 1 - 2y \end{pmatrix}. \tag{84}$$

The gradients and Hessians of $T$ and $V$ for the Hénon–Heiles Hamiltonian can now be implemented as Python functions. In the example below, `henon_heiles_grad_T` returns the gradient of the kinetic energy with respect to the generalized momenta $\mathbf{p}$, and `henon_heiles_grad_V` returns the gradient of the potential energy with respect to the generalized coordinates $\mathbf{q}$. Analogously, one could define additional functions (e.g., `henon_heiles_hess_T` and `henon_heiles_hess_V`) to return the Hessians of $T$ and $V$:

```
>>> import numpy as np
>>> from numba import njit
>>> @njit
>>> @njit
>>> def henon_heiles_grad_T(p, parameters=None):
...     return np.array([p[0], p[1]])
>>> @njit henon_heiles_hess_T(p, parameters=None):
...     return np.array([[1.0, 0.0], [0.0, 1.0]])
>>> @njit
>>> def henon_heiles_grad_V(q, parameters=None):
...     q0, q1 = q[0], q[1]
...     dV_dq0 = q0 * (1.0 + 2.0 * q1)
...     dV_dq1 = q1 + q0 * q0 - q1 * q1
...     return np.array([dV_dq0, dV_dq1])
>>> @njit
>>> def henon_heiles_hess_V(q, parameters=None):
...     q0, q1 = q[0], q[1]
...     H00 = 1.0 + 2.0 * q1
...     H01 = 2.0 * q0
...     H11 = 1.0 - 2.0 * q1
...     return np.array([[H00, H01], [H01, H11]])
```

Note that even though the Hénon–Heiles Hamiltonian itself does not depend on any additional parameters, the function signatures include a parameters argument. This ensures compatibility with the general integrator interface, which passes user-defined parameter arrays to all gradient functions. Then, to create the Hamiltonian system object, we proceed as follows:

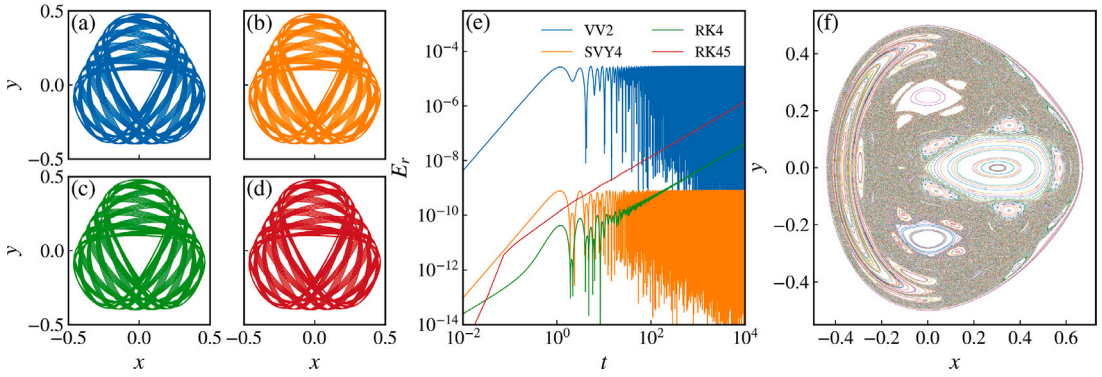


**Fig. 20.** (a)–(d) Trajectories for the Hénon–Heiles Hamiltonian [Eq. (82)] using different numerical integrators, namely, VV2, SVY4, RK4, and RK45, respectively. The energy and initial conditions used are $(E, x, y, p_y) = (1/8, 0, 0.1, 0)$ and $p_x$ is calculated from $p_x = p_x(E, x, y, p_y)$. (e) The relative energy error $E_r$ as a function of time for these four integration scheme. (f) Demonstration of the use of the `poincare_section` method of the `HamiltonianSystem` class of `pynamicalsys`. Execution times: (a) 251 ms, (b) 723 ms, (c) 838 ms, (d) 484 ms, and (f) 1 min 31 s.

```
>>> from pynamicalsys import HamiltonianSystem as HS
>>> hs = HS(grad_T=henon_heiles_grad_T, grad_V=henon_heiles_grad_V, hess_T=henon_heiles_hess_T, hess_V=
    henon_heiles_hess_V, degrees_of_freedom=2, number_of_parameters=0)
```

The following code snippet demonstrates how to calculate a trajectory for the Hénon–Heiles system using both VV2 and SVY4 integrators. We choose a total energy of $E_0 = 1/8$ and initial condition $(x, y, p_y) = (0, 0.1, 0)$ with $p_x$ being calculated from $p_x = p_x(x, y, p_y, E)$ [Fig. 20(a) and (b)]:

```
>>> E_0 = 1 / 8  # Total energy of the system
>>> x, y, py = 0, 0.1, 0  # Define the initial condition
>>> px = np.sqrt(2 * (E_0 - x**2 * y + y**3 / 3) - x**2 - y**2 - py**2)
>>> q = np.array([x, y])
>>> p = np.array([px, py])
>>> total_time = 300
>>> ds.integrator("vv2", time_step=0.01)
>>> trajectory_vv2 = hs.trajectory(q, p, total_time)
>>> ds.integrator("svy4", time_step=0.01)
>>> trajectory_svy4 = hs.trajectory(q, p, total_time)
```

To compare these trajectories with nonsymplectic integrators, we compute the same trajectories using the RK4 integrator with time step $\Delta t = 0.01$ and the RK45 integrator with absolute and relative tolerances of atol = rtol = $10^{-10}$ [Fig. 20(c) and (d)] and calculate the relative energy error

$$E_r(t) = \frac{|E(t) - E_0|}{E_0} \tag{85}$$

as a function of time [Fig. 20(e)] up to $T = 10^4$. We find that $E_r$ for both RK4 and RK45 initially remains smaller than $E_r$ for the VV2 integrator, but gradually increases with time, while $E_r$ for VV2 stays bounded and below $10^{-4}$. As expected, the SVY4 scheme exhibits the smallest relative error, remaining bounded and below $10^{-9}$. Although the trajectories produced by nonsymplectic integrators appear visually similar to those obtained with symplectic schemes for short integration times ($T = 300$), this agreement is only superficial. Over longer time intervals, the systematic energy drift inherent to nonsymplectic methods leads to a cumulative phase error and eventual divergence of the trajectories from the true Hamiltonian flow. In contrast, symplectic integrators constrain the energy oscillations and preserve the qualitative structure of phase space even over very long integrations, making them more reliable for studying long-term dynamics.

The Hénon–Heiles Hamiltonian is a system with a four-dimensional phase space. This high dimensionality makes direct visualization of trajectories difficult. In Section 6.4, we discussed two common techniques for reducing the dimensionality of the phase space: the Poincaré section and the stroboscopic map. While the latter has already been introduced in Section 6.4, we now demonstrate how to obtain a Poincaré section using `pynamicalsys`. We employ here the `HamiltonianSystem` class, although the procedure is analogous for the `ContinuousDynamicalSystem` class. The `poincare_section` method of the `HamiltonianSystem` class extracts the points at which a trajectory intersects a given lower-dimensional surface of section, thereby reducing the dimensionality of the flow and revealing its geometric structure:

```
obj.poincare_section(q, p, num_intersections, parameters=None, section_index=0, section_value=0, crossing=1)
```

Here `q` and `p` are the initial generalized coordinates and momenta of the trajectory, and `num_intersections` specifies how many intersection points with the section should be computed. The optional `parameters` argument allows passing system parameters if needed. The `section_index` selects which phase-space coordinate is used to define the section, while `section_value` specifies the value of that coordinate at which the section is taken (by default, the hyperplane is $q_0 = 0$). The `crossing` parameter determines the direction of crossing: `1` restricts to upward crossings, `-1` to downward crossings, and `0` to all crossings regardless of direction. The method returns an array with the time of the crossings and the phase-space points lying on the chosen Poincaré section, which can then be plotted or further analyzed.

The following code snippet demonstrate the use of the `poincare_section` method for an ensemble of initial conditions with energy $E = 1/8$ and $x = 0$ [Fig. 20(f)]. We choose as our Poincaré section the plane $x = 0$ with $\dot{x} > 0$:

```
>>> import numpy as np
>>> from pynamicalsys import HamiltonianSystem as HS
>>> hs = HS(model="henon heiles")  # Use the built-in model
>>> num_ic = 200  # Number of initial conditions
>>> dof = 2  # Number of degrees of freedom
>>> q, p = np.zeros((num_ic, dof)), np.zeros((num_ic, dof))  # Arrays to store the initial conditions
>>> E_ref = 1 / 8  # Total energy of the system
>>> x = 0  # Define the initial condition
>>> y_range = (-0.5, 0.5)  # Select initial condition within these ranges
>>> py_range = (-0.5, 0.5)
>>> np.random.seed(13)  # Set the seed for reproducibility
>>> for i in range(num_ic):  # Generate the valid initial conditions
...     while True:
...         py = np.random.uniform(*py_range)
...         y = np.random.uniform(*y_range)
...         px = 2 * (E_ref - x**2 * y + y**3 / 3) - x**2 - y**2 - py**2
...         if px > 0:
...             q[i] = [x, y]
...             p[i] = [np.sqrt(px), py]
...             break
>>> num_intersections = 5000  # Number of points in the Poincaré section
>>> PS = hs.poincare_section(q, p, num_intersections)  # Generate the Poincaré section
>>> PS.shape
(200, 5000, 5)
```

Since the `HamiltonianSystem` class shares the same workflow as the `ContinuousDynamicalSystem` class, we have opted not to include separate demonstrations of the `stroboscopic_map`, `lyapunov`, `SALI`, and `LDI` methods. These methods follow the same usage patterns as those already illustrated in Sections 6.3 and 6.4. We refer the reader to the documentation page [203] for further details.

## 7. Conclusions

In this paper, we have introduced `pynamicalsys`, an open-source Python module for the analysis of both discrete- and continuous-time dynamical systems. The module implements a variety of methods to analyze and quantify the dynamical behavior of dynamical systems. These include trajectory, Poincaré section, stroboscopic map, and bifurcation diagram computation, Lyapunov exponent estimation, the smaller alignment index (SALI) and the linear dependence index (LDI), and other indicators of chaotic behavior. Additionally, for the discrete-time systems, it provides tools for periodic orbit detection and the computation of their invariant manifolds, as well as escape analysis and basin quantification. All methods are built on top of Numpy and Numba, ensuring high performance and efficiency. The `DiscreteDynamicalSystem`, `ContinuousDynamicalSystem`, and `HamiltonianSystem` classes come with several built-in models ready to use, however, they are not limited to the built-in ones. The definition of custom mapping functions and equations of motion is extremely easy and straightforward.

We have provided a description, literature review, and mathematical description of the principal methods and classes of `pynamicalsys`'s module. Additionally, the complete documentation is available in Ref. [203], with a more in-depth discussion and beginner-friendly language along with the API (Application Programming Interface) reference. The Jupyter notebook used to generate and plot all the data used in this paper is available in the Supplementary Material as well as in Ref. [204]. We are committed to keep implementing new features and including them in new versions of `pynamicalsys`.

Even though there has been a paradigm shift in the scientific community regarding the public sharing of code and data [205], many researchers remain reluctant [206], and some do not comply with their own published data sharing statements [207]. This has led to reproducibility issues in many scientific publications [208,209]. Therefore, we hope that `pynamicalsys` contributes to making research in nonlinear dynamics more accessible and reproducible, and helps shift the prevailing culture of withholding code and data.

While `pynamicalsys` shares some functionality with established toolkits such as MATCONT [210], DynamicalSystems.jl [211, 212], and JITCode [213], its scope and design philosophy differ. In contrast to MATCONT's emphasis on advanced continuation methods within a proprietary MATLAB environment, `pynamicalsys` offers a lightweight, fully open-source Python alternative that supports both discrete and continuous systems through a unified API. Compared to DynamicalSystems.jl, it targets the larger and rapidly growing Python user community, benefiting from Python's widespread adoption across scientific disciplines. With Numba acceleration, `pynamicalsys` achieves competitive performance without requiring a compiled-language toolchain. Unlike JITCode, which focuses solely on fast ODE integration, `pynamicalsys` integrates trajectory computation, chaos detection, and bifurcation analysis into a single package, aiming to balance flexibility performance and usability for research and teaching in nonlinear dynamics. We are currently exploring the incorporation of JITCode into the continuous system class to further enhance integration performance.

There are also some other open-source projects related to numerical methods in nonlinear dynamics research, such as the `pyunicorn` [122], `ordpy` [214], `tisean` [215], and `powerlaw` [216].

## CRediT authorship contribution statement

**Matheus Rolim Sales:** Writing – review & editing, Writing – original draft, Visualization, Validation, Supervision, Software, Project administration, Methodology, Investigation, Formal analysis, Data curation, Conceptualization. **Leonardo Costa de Souza:** Writing – review & editing, Writing – original draft, Visualization, Validation, Methodology, Investigation, Formal analysis, Data curation, Conceptualization. **Daniel Borin:** Writing – review & editing, Writing – original draft, Visualization, Validation, Methodology, Investigation, Formal analysis, Data curation. **Michele Mugnaine:** Writing – review & editing, Writing – original draft, Visualization, Validation, Methodology, Investigation, Formal analysis, Data curation. **José Danilo Szezech:** Writing – review & editing, Writing – original draft, Visualization, Validation, Methodology, Investigation, Formal analysis. **Ricardo Luiz Viana:** Writing – review & editing, Writing – original draft, Visualization, Validation, Methodology, Investigation, Formal analysis. **Iberê Luiz Caldas:** Writing – review & editing, Writing – original draft, Visualization, Validation, Methodology, Investigation, Formal analysis. **Edson Denis Leonel:** Writing – review & editing, Writing – original draft, Visualization, Validation, Methodology, Investigation, Formal analysis. **Chris G. Antonopoulos:** Writing – review & editing, Writing – original draft, Visualization, Validation, Methodology, Investigation, Formal analysis.

## Code availability

The source code to reproduce the results presented in this paper is freely available in the GitHub repository at Ref. [217].

## Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

## Acknowledgments

This work was supported by the São Paulo Research Foundation (FAPESP, Brazil), under Grant Nos. 2019/14038-6, 2021/09519-5, 2023/08698-9, 2023/16146-6, 2024/09208-8, 2024/03570-7, 2024/06248-9, 2024/06749-8, 2024/14825-6, 2024/20417-8, and 2024/05700-5, by the National Council for Scientific and Technological Development (CNPq, Brazil), under Grant Nos. 301318/2019-0, 301019/2019-3, 304616/2021-4, 403120/2021-7, 309670/2023-3, 304398/2023-3, and 443575/2024-0. It was also supported by the Coordination of Superior Level Staff Improvement (CAPES, Brazil), under Grant No. 88881.895032/2023-01. We also would like to thank the 105 Group Science for fruitful discussions.

## Appendix A. Benchmarks

In this section, we show the average execution times for the main functionalities of the `DiscreteDynamicalSystem` (Table A.2) and `ContinuousDynamicalSystem` (Table A.3) classes for typical values of total iteration time, $N = 1 \times 10^6$, and total integration time, $T = 1.0 \times 10^4$ time units. The benchmarks were obtained on a MacBook Air equipped with an Apple M4 chip, featuring a 10-core CPU. The timing measurements for the methods were performed in a Jupyter Notebook using the `%%timeit` magic command with the options `-n 5 -r 10`, i.e., 5 loops per run and 10 runs in total, reporting the mean ± standard deviation over the 10 runs.

Overall, the execution times for both discrete and continuous dynamical systems increase with the system's dimensionality and the computational complexity of the method. In the `DiscreteDynamicalSystem` class, the `trajectory` method is consistently the fastest, with runtimes under 60 ms even for 4D systems. In contrast, `lyapunov` calculations are significantly more expensive, rising from about 94 ms in 1D to nearly 3.8 s in 4D. This higher cost is primarily due to the repeated QR decompositions required for the base vectors. Notably, the `lyapunov` method is far more efficient for 2D systems than for higher-dimensional cases, thanks to the analytical approach described in Section 3.1.2. Among the chaotic indicators, `SALI` is generally the most efficient. Although SALI and $LDI_2$ are equivalent, $LDI_2$ is considerably slower because it requires computing a singular value decomposition at each

**Table A.2**
Benchmark execution times for the main functionalities of the `DiscreteDynamicalSystem` class in `pynamicalsys`. All simulations were performed using a total iteration time of $N = 1 \times 10^6$ and a transient time of $N_t = 5 \times 10^5$, except for the `recurrence_time_entropy` method, for which we used $N = 5.1 \times 10^5$ and $N_t = 5 \times 10^5$.

| Class method | 1D | 2D area-preserving | 2D dissipative | 3D | 4D |
|---|---|---|---|---|---|
| `trajectory` | 21 ms ± 6.33 ms | 35.9 ms ± 1.14 ms | 44.5 ms ± 923 µs | 44.5 ms ± 786 µs | 56.6 ms ± 605 µs |
| `lyapunov` | 94.2 ms ± 3.94 ms | 345 ms ± 3.2 ms | 197 ms ± 2.21 ms | 1.28 s ± 12.6 ms | 3.77 s ± 168 ms |
| `recurrence_time_entropy` | 142 ms ± 7.12 ms | 78.8 ms ± 330 s | 133 ms ± 9.93 ms | 110 ms ± 5.11 ms | 98.2 ms ± 554 µs |
| `SALI` | – | 125 µs ± 17.1 µs | 20.8 ms ± 87.4 µs | 20.1 ms ± 185 µs | 4.65 ms ± 143 µs |
| $\text{LDI}_2$ | – | 1.47 ms ± 591 µs | 209 ms ± 7.2 ms | 225 ms ± 4.03 ms | 126 ms ± 2.24 ms |
| $\text{LDI}_3$ | – | – | – | 200 ms ± 7.88 ms | 59.9 ms ± 805 µs |
| $\text{LDI}_4$ | – | – | – | – | 13.6 ms ± 1.08 ms |

**Table A.3**
Benchmark execution times for the main functionalities of the `ContinuousDynamicalSystem` class in `pynamicalsys`. All simulations were performed using a total integration time of $T = 1.0 \times 10^4$ time units and a transient time of $t = 5.0 \times 10^3$ time units using the RK4 method with a time step of $\Delta t = 0.01$.

| Class method | Lorenz system | Rössler system | 4D Rössler system |
|---|---|---|---|
| `trajectory` | 574 ms ± 3.36 ms | 596 ms ± 50.6 ms | 595 ms ± 5.95 ms |
| `lyapunov` | 2.18 s ± 63 ms | 2.17 s ± 74.9 ms | 2.82 s ± 54.8 ms |
| `SALI` | 206 ms ± 61.6 ms | 314 ms ± 56 ms | 296 ms ± 2.88 ms |
| $\text{LDI}_2$ | 344 ms ± 23.8 ms | 1.92 s ± 31.1 ms | 2.07 s ± 31.5 ms |
| $\text{LDI}_3$ | 189 ms ± 2.68 ms | 201 ms ± 2.12 ms | 1.21 s ± 23.6 ms |
| $\text{LDI}_4$ | – | – | 194 ms ± 8.81 ms |

**Table A.4**
Benchmark execution times for the main functionalities of the `HamiltonianSystem` class in `pynamicalsys` (third and fourth column) in comparison with the main functionalities of the `ContinuousDynamicalSystem` class (second column) for the Hénon–Heiles Hamiltonian system. All simulations were performed using a total integration time of $T = 1.0 \times 10^4$ time units and a time step of $\Delta t = 0.01$. For the Poincaré section, we considered 5000 intersections.

| Class method | 4th order Runge–Kutta (RK4) | 2nd order velocity–Verlet (VV2) | 4th order Yoshida (SVY4) |
|---|---|---|---|
| `trajectory` | 837 ms ± 12.3 ms | 216 ms ± 9.23 ms | 699 ms ± 71.6 ms |
| `poincare_section` | 933 ms ± 17.2 ms | 682 ms ± 7.76 ms | 2.16 s ± 19 ms |
| `lyapunov` | 16 s ± 128 ms | 6.18 s ± 31.7 ms | 13.5 s ± 104 ms |
| `SALI` | 211 ms ± 2.31 ms | 318 ms ± 20.5 ms | 576 ms ± 24.9 ms |
| $\text{LDI}_2$ | 1.83 s ± 23.4 ms | 3.65 s ± 14.4 ms | 2.57 s ± 13.9 ms |
| $\text{LDI}_3$ | 799 ms ± 10.9 ms | 1.35 s ± 3.99 ms | 1.57 s ± 4.78 ms |
| $\text{LDI}_4$ | 330 ms ± 3.44 ms | 668 ms ± 2.61 ms | 805 ms ± 18.1 ms |

step. These benchmarks highlight that, unless LEs are specifically needed, distinguishing between chaotic and regular solutions can be performed much more efficiently using SALI or LDI.

For the `ContinuousDynamicalSystem` class, all methods are costlier than their discrete counterparts due to the numerical integration of ODEs. The `trajectory` method runs in roughly 0.57 s–0.60 s regardless of dimension, while `lyapunov` calculations take around 2.2 s for three-dimensional systems and 2.82 s for the 4D case. Among the chaotic indicators, `SALI` remains faster than most `LDI` variants, though the runtime gap is smaller than in the discrete case. Higher-order `LDI` computations again exhibit a noticeable increase in cost with dimension. These results show that the implemented methods are efficient for large-scale simulations, with runtimes well below a few seconds even for high-dimensional cases, making them suitable for extensive parameter sweeps and statistical studies.

For the `HamiltonianSystem` class (Table A.4), the execution times depend strongly on the chosen integration scheme. As expected, the second-order velocity-Verlet (VV2) integrator yields the fastest `trajectory` and `lyapunov` calculations (roughly four times faster than the standard RK4 implementation) while still maintaining symplecticity. The fourth-order Yoshida (SVY4) integrator strikes a balance between speed and accuracy, outperforming RK4 for some tasks but remaining slower than VV2. For Poincaré section computations, VV2 again provides the lowest runtime, whereas SVY4 is the most expensive because it requires multiple force evaluations per step. The chaotic indicators show a similar trend: `SALI` remains the most efficient overall, while higher-order `LDI` variants grow progressively more costly. Notably, the differences between integrators become especially pronounced for `lyapunov` calculations, where RK4 takes about 16 s while VV2 completes the same task in just over 6 s. These benchmarks demonstrate that symplectic integrators, particularly VV2, offer substantial performance gains for Hamiltonian systems while preserving the geometric structure of phase space, making them well-suited for long-term integrations and large-scale parameter studies.

## Appendix B. Numerical integrators

*Fourth-order Runge–Kutta method*

The fourth-order Runge–Kutta method is one of the most widely used algorithms to numerically solve initial-value problems of the form

$$\begin{aligned} &\dot{\mathbf{y}}(t) = \mathbf{f}(t, \mathbf{y}(t)), \\ &\mathbf{y}(t_0) = \mathbf{y}_0. \end{aligned} \tag{B.1}$$

For a given time step $h$, the RK4 scheme advances the solution from $\mathbf{y}_n$ at $t_n$ to $\mathbf{y}_{n+1}$ at $t_{n+1} = t_n + h$ via four intermediate slope evaluations:

$$\begin{aligned} \mathbf{k}_1 &= \mathbf{f}\big(t_n, \mathbf{y}_n\big), \\ \mathbf{k}_2 &= \mathbf{f}\Big(t_n + \frac{h}{2}, \mathbf{y}_n + \frac{h}{2}\mathbf{k}_1\Big), \\ \mathbf{k}_3 &= \mathbf{f}\Big(t_n + \frac{h}{2}, \mathbf{y}_n + \frac{h}{2}\mathbf{k}_2\Big), \\ \mathbf{k}_4 &= \mathbf{f}\big(t_n + h, \mathbf{y}_n + h\mathbf{k}_3\big), \end{aligned} \tag{B.2}$$

and the approximation of the state at $t_{n+1}$ is then given by

$$\mathbf{y}_{n+1} = \mathbf{y}_n + \frac{h}{6}\big(\mathbf{k}_1 + 2\mathbf{k}_2 + 2\mathbf{k}_3 + \mathbf{k}_4\big). \tag{B.3}$$

This method achieves fourth-order accuracy in the local truncation error, $\mathcal{O}(h^5)$, and third-order accuracy for the global error, $\mathcal{O}(h^4)$.

*Adaptive 4th/5th-order Runge–Kutta method (Dormand–Prince)*

The adaptive Runge–Kutta method combines two formulas of different orders to control the local truncation error and adjust the time step automatically. A widely used implementation is the fourth/fifth-order Dormand–Prince method (RK45). Like the classical RK4, RK45 solves initial-value problems of the form given by Eq. (B.1), but in each step it computes two approximations of the solution: a fifth-order estimate $\mathbf{y}^{(5)}_{n+1}$ and a fourth-order estimate $\mathbf{y}^{(4)}_{n+1}$. The difference between these two estimates provides an error estimate:

$$\mathbf{e}_{n+1} = \mathbf{y}^{(5)}_{n+1} - \mathbf{y}^{(4)}_{n+1}. \tag{B.4}$$

In a single step, the method evaluates seven intermediate stages $\mathbf{k}_i$ according to the Dormand–Prince coefficients $c_i$ and $a_{ij}$:

$$\mathbf{k}_i = \mathbf{f}\left(t_n + c_i h, \mathbf{y}_n + h\sum_{j=0}^{i-1} a_{ij}\mathbf{k}_j\right), \tag{B.5}$$

where $i = 0, 1, \ldots, 6$ and $h$ is the current time step. The fourth- and fifth-order approximations are then obtained as

$$\begin{aligned} \mathbf{y}^{(4)}_{n+1} &= \mathbf{y}_n + h\sum_{i=0}^{6} b^{(4)}_i \mathbf{k}_i, \\ \mathbf{y}^{(5)}_{n+1} &= \mathbf{y}_n + h\sum_{i=0}^{6} b^{(5)}_i \mathbf{k}_i, \end{aligned} \tag{B.6}$$

where $b^{(4)}_i$ and $b^{(5)}_i$ are the corresponding weights.

The element-wise error is normalized using absolute and relative tolerances (atol and rtol):

$$\mathbf{e}_{\text{norm}} = \frac{\left|\mathbf{y}^{(5)}_{n+1} - \mathbf{y}^{(4)}_{n+1}\right|}{\text{atol} + \text{rtol}\cdot\max\left(\left|\mathbf{y}_n\right|, \left|\mathbf{y}^{(5)}_{n+1}\right|\right)}, \tag{B.7}$$

and the error is then computed as $\text{err} = \max(\mathbf{e}_{\text{norm}})$. If $\text{err} < 1$, the step is accepted. Otherwise, the step is rejected and recomputed with a smaller time step. The next time step is then adapted according to

$$h_{\text{new}} = h\cdot\min\left(\max\left(0.9\,\text{err}^{-0.25}, 0.1\right), 2.0\right). \tag{B.8}$$

The new time step is scaled according to the error estimate but bounded to the interval $[0.1\,h,\ 2.0\,h]$ to prevent excessively large or small changes.

The adaptive RK45 method is highly efficient for general ordinary differential equations because it automatically increases the step size when the solution varies slowly and decreases it when rapid changes occur.

*Second-order velocity–Verlet integrator*

Consider an autonomous Hamiltonian system with $N$ degrees of freedom and a separable Hamiltonian

$$H(\mathbf{p},\mathbf{q}) = T(\mathbf{p}) + V(\mathbf{q}), \tag{B.9}$$

where $\mathbf{q}$ and $\mathbf{p}$ are the generalized coordinates and momenta, respectively. The equations of motion are given by Hamilton's equations

$$\begin{aligned} \dot{q}_i &= \frac{\partial H}{\partial p_i}, \\ \dot{p}_i &= -\frac{\partial H}{\partial q_i}, \end{aligned} \tag{B.10}$$

for $i = 1, 2, \ldots, N$. The velocity-Verlet integrator [199] in the kick–drift–kick form advances the system from time $t_n$ to $t_{n+1} = t_n + h$ through the following sequence:

*half-kick of momentum*:

$$\mathbf{p}_{n+\frac{1}{2}} = \mathbf{p}_n - \frac{h}{2}\frac{\partial H}{\partial \mathbf{q}}, \tag{B.11}$$

*drift of position*:

$$\mathbf{q}_{n+1} = \mathbf{q}_n + h\frac{\partial H}{\partial \mathbf{p}}, \tag{B.12}$$

*half-kick of momentum*:

$$\mathbf{p}_{n+1} = \mathbf{p}_{n+\frac{1}{2}} - \frac{h}{2}\frac{\partial H}{\partial \mathbf{q}}. \tag{B.13}$$

This integrator is second-order accurate, with a local truncation error of $\mathcal{O}(h^3)$ and a global error of $\mathcal{O}(h^2)$. Its symplectic nature ensures the preservation of phase-space volume, making it particularly suitable for long-term integration of Hamiltonian systems. The algorithm is time-reversible, and although the Hamiltonian is not exactly conserved at each step, the energy error remains bounded, typically exhibiting small oscillations around the true value without secular drift. The kick–drift–kick decomposition corresponds directly to the standard velocity-Verlet scheme but emphasizes the separation of momentum and position updates, which is convenient for numerical implementation.

For the calculation of the Lyapunov exponents, one also needs to numerically integrate the variational equations. This integration must be carried out using a symplectic integrator [201]. To rewrite the variational equations in a form suitable for the same symplectic scheme, let us define a trajectory in the $2N$-dimensional phase space by the vector $\mathbf{x}(t) = (\mathbf{q}(t), \mathbf{p}(t))$. Hamilton's equations [Eq. (B.10)] can then be expressed in matrix form as

$$\dot{\mathbf{x}} = \begin{pmatrix} \frac{\partial H}{\partial \mathbf{p}} & -\frac{\partial H}{\partial \mathbf{q}} \end{pmatrix}^T = \mathbf{J}_{2N} \cdot \mathbf{D}_H(\mathbf{x}(t)), \tag{B.14}$$

where

$$\mathbf{D}_H(\mathbf{x}(t)) = \begin{pmatrix} \frac{\partial H(\mathbf{x}(t))}{\partial q_1} & \frac{\partial H(\mathbf{x}(t))}{\partial q_2} & \cdots & \frac{\partial H(\mathbf{x}(t))}{\partial q_N} & \frac{\partial H(\mathbf{x}(t))}{\partial p_1} & \frac{\partial H(\mathbf{x}(t))}{\partial p_2} & \cdots & \frac{\partial H(\mathbf{x}(t))}{\partial p_N} \end{pmatrix}^T \tag{B.15}$$

is the gradient of the Hamiltonian evaluated at $\mathbf{x}(t)$, and

$$\mathbf{J}_{2N} = \begin{pmatrix} \mathbf{0}_N & \mathbf{I}_N \\ -\mathbf{I}_N & \mathbf{0}_N \end{pmatrix}, \tag{B.16}$$

with $\mathbf{I}_N$ the $N \times N$ identity matrix and $\mathbf{0}_N$ the $N \times N$ zero matrix. An initial deviation vector $\mathbf{v}(0) = (\delta\mathbf{q}(0), \delta\mathbf{p}(0))$ from an orbit $\mathbf{x}(t)$ evolves in the tangent space according to the variational equations

$$\dot{\mathbf{v}} = \left[\mathbf{J}_{2N} \cdot \mathbf{D}_H^2(\mathbf{x}(t))\right] \cdot \mathbf{v}, \tag{B.17}$$

where $\mathbf{D}_H^2(\mathbf{x}(t))$ is the Hessian of the Hamiltonian evaluated at $\mathbf{x}(t)$:

$$\mathbf{D}_H^2(\mathbf{x}(t)) = \begin{pmatrix} \frac{\partial^2 H}{\partial \mathbf{q}^2} & \frac{\partial^2 H}{\partial \mathbf{q}\partial \mathbf{p}} \\ \frac{\partial^2 H}{\partial \mathbf{p}\partial \mathbf{q}} & \frac{\partial^2 H}{\partial \mathbf{p}^2} \end{pmatrix} = \begin{pmatrix} \mathbf{H}_{qq} & \mathbf{H}_{qp} \\ \mathbf{H}_{pq} & \mathbf{H}_{pp} \end{pmatrix}. \tag{B.18}$$

Eq. (B.17) is linear in $\mathbf{v}$ with time-dependent coefficients given by

$$\mathbf{J}_{2N} \cdot \mathbf{D}_H^2(\mathbf{x}(t)) = \begin{pmatrix} \mathbf{0}_N & \mathbf{I}_N \\ -\mathbf{I}_N & \mathbf{0}_N \end{pmatrix} \begin{pmatrix} \mathbf{H}_{qq} & \mathbf{H}_{qp} \\ \mathbf{H}_{pq} & \mathbf{H}_{pp} \end{pmatrix} = \begin{pmatrix} \mathbf{H}_{pq} & \mathbf{H}_{pp} \\ -\mathbf{H}_{qq} & -\mathbf{H}_{qp} \end{pmatrix}. \tag{B.19}$$

Splitting Eq. (B.17) into the components $\delta\mathbf{q}(t)$ and $\delta\mathbf{p}(t)$, we obtain

$$\begin{aligned} \delta\dot{\mathbf{q}}(t) &= \mathbf{H}_{pq} \cdot \delta\mathbf{q} + \mathbf{H}_{pp} \cdot \delta\mathbf{p}, \\ \delta\dot{\mathbf{p}}(t) &= -\mathbf{H}_{qq} \cdot \delta\mathbf{q} - \mathbf{H}_{qp} \cdot \delta\mathbf{p}. \end{aligned} \tag{B.20}$$

Since the Hamiltonian [Eq. (B.9)] is separable, $\mathbf{H}_{qp} = \mathbf{H}_{pq} \equiv \mathbf{0}$, and the variational equations reduce to

$$\begin{aligned} \delta\dot{\mathbf{q}}(t) &= \mathbf{H}_{pp} \cdot \delta\mathbf{p}, \\ \delta\dot{\mathbf{p}}(t) &= -\mathbf{H}_{qq} \cdot \delta\mathbf{q}. \end{aligned} \tag{B.21}$$

These equations can be integrated together with the main trajectory using the velocity-Verlet integrator:

*half-kick of momentum*:

$$\delta\mathbf{p}_{n+\frac{1}{2}} = \delta\mathbf{p}_n - \frac{h}{2}\mathbf{H}_{qq}(\mathbf{q}_n) \cdot \delta\mathbf{q}_n, \tag{B.22}$$

*drift of position*:

$$\delta\mathbf{q}_{n+1} = \delta\mathbf{q}_n + h\,\mathbf{H}_{pp}(\mathbf{p}_{n+\frac{1}{2}}) \cdot \delta\mathbf{p}_{n+\frac{1}{2}}, \tag{B.23}$$

*half-kick of momentum*:

$$\delta\mathbf{p}_{n+1} = \delta\mathbf{p}_{n+\frac{1}{2}} - \frac{h}{2}\mathbf{H}_{qq}(\mathbf{q}_{n+1}) \cdot \delta\mathbf{q}_{n+1}. \tag{B.24}$$

*Fourth-order Yoshida integrator*

The fourth-order Yoshida integrator [200] is a composition of three second-order symplectic steps (velocity-Verlet) with carefully chosen coefficients $\alpha$ and $\beta$, which cancel lower-order errors and yield an overall fourth-order method. Let us consider again the separable Hamiltonian function, given by Eq. (B.9). Let $h$ be the full time step. The coefficients $\alpha$ and $\beta$ are defined by

$$\alpha = \frac{1}{2 - 2^{1/3}}, \qquad \beta = -\frac{2^{1/3}}{2 - 2^{1/3}}. \tag{B.25}$$

The integrator advances the system from $t_n$ to $t_{n+1} = t_n + h$ via three consecutive velocity-Verlet updates:

*first velocity-Verlet step with step $\alpha h$*:

$$(\mathbf{q}^{(1)}, \mathbf{p}^{(1)}) = \mathrm{VV}(\mathbf{q}_n, \mathbf{p}_n, \alpha h), \tag{B.26}$$

*second velocity-Verlet step with step $\beta h$*:

$$(\mathbf{q}^{(2)}, \mathbf{p}^{(2)}) = \mathrm{VV}(\mathbf{q}^{(1)}, \mathbf{p}^{(1)}, \beta h), \tag{B.27}$$

*third velocity-Verlet step with step $\alpha h$*:

$$(\mathbf{q}_{n+1}, \mathbf{p}_{n+1}) = \mathrm{VV}(\mathbf{q}^{(2)}, \mathbf{p}^{(2)}, \alpha h), \tag{B.28}$$

where $\mathrm{VV}(\mathbf{q}, \mathbf{p}, \Delta t)$ denotes a single velocity-Verlet step of size $\Delta t$. This composition results in a fourth-order symplectic integrator that preserves phase-space volume and exhibits bounded energy errors over long times. The same sequence of steps can be applied to variational vectors $\delta\mathbf{q}$ and $\delta\mathbf{p}$ for the computation of Lyapunov exponents.

## Data availability

I have shared the source code to reproduce all the results in GitHub and in the Supplementary Material.

## References


[1] de Laplace PS. Essai philosophique sur les probabilités. Bachelier; 1825.
[2] Poincaré H. Les méthodes nouvelles de la Mécanique céleste. Les méthodes nouvelles de la Mécanique céleste, vol. 2, Gauthier-Villars; 1893.
[3] Lorenz EN. Deterministic nonperiodic flow. J Atmos Sci 1963;20:130–41.
[4] Smale S. Diffeomorphisms with many periodic points. Columbia University, Department of Mathematics; 1963.
[5] Smale S. Differentiable dynamical systems. Bull Am Math Soc 1967;73(6):747–817.
[6] Sinai YG. Spectral properties of ergodic dynamical systems. Dokl Akad Nauk SSSR 1963;1235–7.
[7] Sinai YG. On the foundations of the ergodic hypothesis for a dynamical system of statistical mechanics. Dokl Akad Nauk SSSR 1963;1261–4.
[8] Li T-Y, Yorke JA. Period three implies chaos. Am Math Mon 1975;82(10):985–92.
[9] Feigenbaum MJ. Quantitative universality for a class of nonlinear transformations. J Stat Phys 1978;19(1):25–52.
[10] Feigenbaum MJ. The universal metric properties of nonlinear transformations. J Stat Phys 1979;21(6):669–706.
[11] Chirikov BV. A universal instability of many-dimensional oscillator systems. Phys Rep 1979;52(5):263–379.
[12] Morrison PJ, Greene JM. Noncanonical Hamiltonian density formulation of hydrodynamics and ideal magnetohydrodynamics. Phys Rev Lett 1980;45:790–4.
[13] Cary JR, Littlejohn RG. Noncanonical Hamiltonian mechanics and its application to magnetic field line flow. Ann Physics 1983;151(1):1–34.
[14] Escande DF. Contributions of plasma physics to chaos and nonlinear dynamics. Plasma Phys Control Fusion 2016;58:113001.
[15] Chian AC-L, Muñoz PR, Rempel EL. Edge of chaos and genesis of turbulence. Phys Rev E 2013;88:052910.
[16] Ecke RE. Chaos, patterns, coherent structures, and turbulence: Reflections on nonlinear science. Chaos: Interdiscip J Nonlinear Sci 2015;25:097605.
[17] Contopoulos G. Order and chaos in dynamical astronomy. Heidelberg: Springer Berlin; 2002.
[18] Lefebvre JH, Goodings DA, Kamath MV, Fallen EL. Predictability of normal heart rhythms and deterministic chaos. Chaos: An Interdiscip J Nonlinear Sci 1993;3:267–76.

[19] Lombardi F. Chaos theory, heart rate variability, and arrhythmic mortality. Circulation 2000;101(1):8–10.
[20] Ferreira BB, de Paula AS, Savi MA. Chaos control applied to heart rhythm dynamics. Chaos Solitons Fractals 2011;44(8):587–99.
[21] Blasius B, Stone L. Chaos and phase synchronization in ecological systems. Int J Bifurc Chaos 2000;10(10):2361–80.
[22] Stiefs D, Venturino E, Feudel U. Evidence of chaos in eco-epidemic models. 2009.
[23] Labos E. Chaos and neural networks. Boston, MA: Springer US; 1987, p. 195–206.
[24] Rabinovich MI, Abarbanel HDI, Huerta R, Elson R, Selverston AI. Self-regularization of chaos in neural systems: experimental and theoretical results. IEEE Trans Circuits Syst I 1997;44(10):997–1005.
[25] Potapov A, Ali M. Robust chaos in neural networks. Phys Lett A 2000;277(6):310–22.
[26] Guckenheimer J, Oliva RA. Chaos in the Hodgkin–Huxley model. SIAM J Appl Dyn Syst 2002;1(1):105–14.
[27] Peters E. Fractal market analysis: applying chaos theory to investment and economics. Wiley Finance, Wiley; 1994.
[28] Hibbert B, Wilkinson IF. Chaos theory and the dynamics of marketing systems. J Acad Mark Sci 1994;22(3):218–33.
[29] Ma J-h, Chen Y-s. Study for the bifurcation topological structure and the global complicated character of a kind of nonlinear finance system(i). Appl Math Mech 2001;22(11):1240–51.
[30] Klioutchnikov I, Sigova M, Beizerov N. Chaos theory in finance. Procedia Comput Sci 2017;119:368–75, 6th International Young Scientist Conference on Computational Science, YSC 2017 01-03 2017, Kotka, Finland.
[31] Szumiński W. Integrability analysis of chaotic and hyperchaotic finance systems. Nonlinear Dynam 2018;94(1):443–59.
[32] Baptista MS. Cryptography with chaos. Phys Lett A 1998;240(1):50–4.
[33] Kocarev L. Chaos-based cryptography: a brief overview. IEEE Circuits Syst Mag 2001;1(3):6–21.
[34] Kocarev L, Lian S. Chaos-based cryptography: theory, algorithms and applications. Studies in computational intelligence, Springer Berlin Heidelberg; 2011.
[35] Pareschi F, Setti G, Rovatti R. A fast chaos-based true random number generator for cryptographic applications. In: 2006 proceedings of the 32nd European solid-state circuits conference. 2006, p. 130–3.
[36] Yu F, Li L, Tang Q, Cai S, Song Y, Xu Q. A survey on true random number generators based on chaos. Discrete Dyn Nat Soc 2019;2019(1):2545123.
[37] Deng Q, Wang C, Sun Y, Yang G. Delay difference feedback memristive map: Dynamics, hardware implementation, and application in path planning. IEEE Trans Circuits Syst I Regul Pap 2025;1–11.
[38] Wang C, Li Y, Yang G, Deng Q. A review of fractional-order chaotic systems of memristive neural networks. Mathematics 2025;13(10).
[39] Tang Y, Kurths J, Lin W, Ott E, Kocarev L. Introduction to focus issue: When machine learning meets complex systems: Networks, chaos, and nonlinear dynamics. Chaos: Interdiscip J Nonlinear Sci 2020;30:063151.
[40] Panahi S, Kong L-W, Moradi M, Zhai Z-M, Glaz B, Haile M, Lai Y-C. Machine learning prediction of tipping in complex dynamical systems. Phys Rev Res 2024;6:043194.
[41] Boaretto BRR, Budzinski RC, Rossi KL, Prado TL, Lopes SR, Masoller C. Discriminating chaotic and stochastic time series using permutation entropy and artificial neural networks. Sci Rep 2021;11(1):15789.
[42] Spezzatto GS, Flauzino JVV, Corso G, Boaretto BRR, Macau EEN, Prado TL, Lopes SR. Recurrence microstates for machine learning classification. Chaos: Interdiscip J Nonlinear Sci 2024;34:073140.
[43] Lober L, Palmero MS, Rodrigues FA. Parameter inference in nonlinear dynamical systems via recurrence plots and convolutional neural networks. Phys Rev E 2025;112:014210.
[44] Yang Y, Li H. Neural ordinary differential equations for robust parameter estimation in dynamic systems with physical priors. Appl Soft Comput 2025;169:112649.
[45] Lam SK, Pitrou A, Seibert S. Numba: A LLVM-based Python jit compiler. In: Proceedings of the second workshop on the LLVM compiler infrastructure in HPC. 2015, p. 1–6.
[46] Lichtenberg AJ, Lieberman MA. Regular and chaotic dynamics. Applied mathematical sciences, vol. 38, Springer-Verlag; 1992.
[47] Zaslavsky GM, Edelman M. Hierarchical structures in the phase space and fractional kinetics: I. Classical systems. Chaos: Interdiscip J Nonlinear Sci 2000;10:135–46.
[48] Zaslavsky GM. Dynamical traps. Phys D: Nonlinear Phenom 2002;168–169:292–304.
[49] Manos T, Robnik M. Survey on the role of accelerator modes for anomalous diffusion: The case of the standard map. Phys Rev E - Stat Nonlinear Soft Matter Phys 2014;89(2):1–12.
[50] Harsoula M, Karamanos K, Contopoulos G. Characteristic times in the standard map. Phys Rev E 2019;99.
[51] Sales MR, Mugnaine M, Szezech Jr JD, Viana RL, Caldas IL, Marwan N, Kurths J. Stickiness and recurrence plots: An entropy-based approach. Chaos: Interdiscip J Nonlinear Sci 2023;33:033140.
[52] Souza LC, Sales MR, Mugnaine M, Szezech JD, Caldas IL, Viana RL. Chaotic escape of impurities and sticky orbits in toroidal plasmas. Phys Rev E 2024;109:015202.
[53] Hénon M. A two-dimensional mapping with a strange attractor. Comm Math Phys 1976;50(1):69–77.
[54] Mugnaine M, Sales MR, Szezech JD, Viana RL. Dynamics, multistability, and crisis analysis of a sine-circle nontwist map. Phys Rev E 2022;106:034203.
[55] Ott E. Chaos in dynamical systems. 2nd ed.. Cambridge University Press; 2002.
[56] Shimada I, Nagashima T. A numerical approach to ergodic problem of dissipative dynamical systems. Progr Theoret Phys 1979;61:1605–16.
[57] Benettin G, Galgani L, Giorgilli A, Strelcyn J-M. Lyapunov characteristic exponents for smooth dynamical systems and for Hamiltonian systems; a method for computing all of them. Meccanica 1980;15:9–20.
[58] Wolf A, Swift JB, Swinney HL, Vastano JA. Determining Lyapunov exponents from a time series. Phys D: Nonlinear Phenom 1985;16(3):285–317.
[59] Eckmann JP, Ruelle D. Ergodic theory of chaos and strange attractors. Rev Modern Phys 1985;57:617–56.
[60] Wang L, Benenti G, Casati G, Li B. Ratchet effect and the transporting islands in the chaotic sea. Phys Rev Lett 2007;99:244101.
[61] Celestino A, Manchein C, Albuquerque HA, Beims MW. Ratchet transport and periodic structures in parameter space. Phys Rev Lett 2011;106:234101.
[62] Lopes SR, Szezech JD, Pereira RF, Bertolazzo AA, Viana RL. Anomalous transport induced by nonhyperbolicity. Phys Rev E 2012;86.
[63] Rolim Sales M, Mugnaine M, Leonel ED, Caldas IL, Szezech J, José D. Shrinking shrimp-shaped domains and multistability in the dissipative asymmetric kicked rotor map. Chaos: Interdiscip J Nonlinear Sci 2024;34:113129.
[64] Press WH, Teukolsky SA, Vetterling WT, Flannery BP. Numerical recipes: the art of scientific computing. numerical recipes: the art of scientific computing. 3rd ed.. Cambridge University Press; 2007.
[65] Gallas JAC. Structure of the parameter space of the Hénon map. Phys Rev Lett 1993;70:2714–7.
[66] Gallas JAC. Dissecting shrimps: results for some one-dimensional physical models. Phys A 1994;202(1):196–223.
[67] Manchein C, Beims MW. Conservative generalized bifurcation diagrams. Phys Lett A 2013;377:789–93.
[68] Contopoulos G. Orbits in highly perturbed dynamical systems, 111. Nonperiodic orbits. Astron J 1971;76:147.
[69] Karney CFF. Long-time correlations in the stochastic regime. Phys D: Nonlinear Phenom 1983;8(3):360–80.
[70] Meiss JD, Cary JR, Grebogi C, Crawford JD, Kaufman AN, Abarbanel HDI. Correlations of periodic, area-preserving maps. Phys D: Nonlinear Phenom 1983;6(3):375–84.
[71] Chirikov BV, Shepelyansky DL. Correlation properties of dynamical chaos in Hamiltonian systems. Phys D: Nonlinear Phenom 1984;13(3):395–400.
[72] Efthymiopoulos C, Contopoulos G, Voglis N, Dvorak R. Stickiness and cantori. J Phys A: Math Gen 1997;30(8167).

[73] Zaslavsky GM. Chaos, fractional kinetics, and anomalous transport. Phys Rep 2002;371(6):461–580.
[74] Altmann EG, Motter AE, Kantz H. Stickiness in Hamiltonian systems: From sharply divided to hierarchical phase space. Phys Rev E 2006;73:026207.
[75] Cristadoro G, Ketzmerick R. Universality of algebraic decays in Hamiltonian systems. Phys Rev Lett 2008;100:184101.
[76] Contopoulos G, Harsoula M. Stickiness in chaos. Int J Bifurc Chaos 2008;18(10):2929–49.
[77] Contopoulos G, Harsoula M. Stickiness effects in conservative systems. Int J Bifurc Chaos 2010;20(07):2005–43.
[78] Szezech JD, Lopes SR, Viana RL. Finite-time Lyapunov spectrum for chaotic orbits of non-integrable Hamiltonian systems. Phys Lett A 2005;335:394–401.
[79] Skokos C, Antonopoulos C, Bountis TC, Vrahatis MN. Smaller alignment index (SALI): Determining the ordered or chaotic nature of orbits in conservative dynamical systems. In: Libration point orbits and applications. World Scientific; 2003, p. 653–64.
[80] Skokos C, Antonopoulos C, Bountis TC, Vrahatis MN. Detecting order and chaos in Hamiltonian systems by the SALI method. J Phys A: Math Gen 2004;37:6269–84.
[81] Skokos C, Bountis TC, Antonopoulos C. Geometrical properties of local dynamics in Hamiltonian systems: The generalized alignment index (GALI) method. Phys D: Nonlinear Phenom 2007;231:30–54.
[82] Voglis N, Contopoulos G, Efthymiopoulos C. Detection of ordered and chaotic motion using the dynamical spectra. Celest Mech Dyn Astron 1999;73(1):211–20.
[83] Antonopoulos C, Bountis T. Detecting order and chaos by the linear dependence index (LDI) method. Romai J 2006;2(2):1–13.
[84] Das S, Dock CB, Saiki Y, Salgado-Flores M, Sander E, Wu J, Yorke JA. Measuring quasiperiodicity. vol. 114, EPL; 2016.
[85] Das S, Saiki Y, Sander E, Yorke JA. Quantitative quasiperiodicity. Nonlinearity 2017;30:4111–40.
[86] Das S, Yorke JA. Super convergence of ergodic averages for quasiperiodic orbits. Nonlinearity 2018;31:491–501.
[87] Sander E, Meiss J. Birkhoff averages and rotational invariant circles for area-preserving maps. Phys D: Nonlinear Phenom 2020;411:132569.
[88] Meiss JD, Sander E. Birkhoff averages and the breakdown of invariant tori in volume-preserving maps. Phys D: Nonlinear Phenom 2021;428.
[89] Duignan N, Meiss JD. Distinguishing between regular and chaotic orbits of flows by the weighted Birkhoff average. Phys D: Nonlinear Phenom 2023;449:133749.
[90] Cornfeld IP, Sossinskii AB, Fomin SV, Sinai YG. Ergodic theory. Springer New York; 2012.
[91] Sales MR, Mugnaine M, Viana RL, Caldas IL, Szezech JD. Unpredictability in Hamiltonian systems with a hierarchical phase space. Phys Lett A 2022;431:127991.
[92] Souza LC, Mathias AC, Caldas IL, Elskens Y, Viana RL. Fractal and wada escape basins in the chaotic particle drift motion in tokamaks with electrostatic fluctuations. Chaos: Interdiscip J Nonlinear Sci 2023;33:083132.
[93] Eckmann JP, Kamphorst SO, Ruelle D. Recurrence plots of dynamical systems. Europhys Lett 1987;4:973.
[94] Marwan N, Carmen Romano M, Thiel M, Kurths J. Recurrence plots for the analysis of complex systems. Phys Rep 2007;438(5):237–329.
[95] Zbilut JP, Zaldivar-Comenges J-M, Strozzi F. Recurrence quantification based Liapunov exponents for monitoring divergence in experimental data. Phys Lett A 2002;297(3):173–81.
[96] Kraemer KH, Donner RV, Heitzig J, Marwan N. Recurrence threshold selection for obtaining robust recurrence characteristics in different embedding dimensions. Chaos: Interdiscip J Nonlinear Sci 2018;28:085720.
[97] Thiel M, Carmen Romano M, Kurths J, Meucci R, Allaria E, Arecchi FT. Influence of observational noise on the recurrence quantification analysis. Phys D: Nonlinear Phenom 2002;171(3):138–52.
[98] Schinkel S, Dimigen O, Marwan N. Selection of recurrence threshold for signal detection. Eur Phys J Spec Top 2008;164:45–53.
[99] Ngamga EJ, Nandi A, Ramaswamy R, Romano MC, Thiel M, Kurths J. Recurrence analysis of strange nonchaotic dynamics. Phys Rev E 2007;75:036222.
[100] Ngamga EJ, Buscarino A, Frasca M, Fortuna L, Prasad A, Kurths J. Recurrence analysis of strange nonchaotic dynamics in driven excitable systems. Chaos: Interdiscip J Nonlinear Sci 2008;18:013128.
[101] Webber CL, Zbilut JP. Dynamical assessment of physiological systems and states using recurrence plot strategies. J Appl Physiol 1994;76(2):965–73.
[102] Gao JB. Recurrence time statistics for chaotic systems and their applications. Phys Rev Lett 1999;83:3178–81.
[103] Shockley K, Butwill M, Zbilut JP, Webber CL. Cross recurrence quantification of coupled oscillators. Phys Lett A 2002;305(1):59–69.
[104] Marwan N, Thiel M, Nowaczyk NR. Cross recurrence plot based synchronization of time series. Nonlinear Process Geophys 2002;9(3/4):325–31.
[105] Marwan N, Wessel N, Meyerfeldt U, Schirdewan A, Kurths J. Recurrence-plot-based measures of complexity and their application to heart-rate-variability data. Phys Rev E 2002;66:026702.
[106] Marwan N. A historical review of recurrence plots. Eur Phys J Spec Top 2008;164:3–12.
[107] Gao J, Cai H. On the structures and quantification of recurrence plots. Phys Lett A 2000;270(1):75–87.
[108] Zou Y, Pazó D, Romano MC, Thiel M, Kurths J. Distinguishing quasiperiodic dynamics from chaos in short-time series. Phys Rev E 2007;76:016210.
[109] Ngamga EJ, Senthilkumar DV, Prasad A, Parmananda P, Marwan N, Kurths J. Distinguishing dynamics using recurrence-time statistics. Phys Rev E 2012;85:026217.
[110] Slater NB. The distribution of the integers $n$ for which $\{\theta n\} < \phi$. Math Proc Cambridge Philos Soc 1950;46(4):525–34.
[111] Slater NB. Gaps and steps for the sequence $n\theta \mod 1$. Math Proc Cambridge Philos Soc 1967;63(4):1115–23.
[112] Mayer DH. On the distribution of recurrence times in nonlinear systems. Lett Math Phys 1988;16:139–43.
[113] Zou Y, Thiel M, Romano MC, Kurths J. Characterization of stickiness by means of recurrence. Chaos: Interdiscip J Nonlinear Sci 2007;17:043101.
[114] Simile Baroni R, Egydio de Carvalho R, Castaldi B, Furlanetto B. Time recurrence analysis of a near singular billiard. Math Comput Appl 2019;24(2):50.
[115] Abud CV, Caldas IL. On Slater's criterion for the breakup of invariant curves. Phys D: Nonlinear Phenom 2015;308:34–9.
[116] Hermes JDV, dos Reis MA, Caldas IL, Leonel ED. Break-up of invariant curves in the Fermi-Ulam model. Chaos Solitons Fractals 2022;162:112410.
[117] Huggler YH, Hermes JDV, Leonel ED. Application of the Slater criteria to localize invariant tori in Hamiltonian mappings. Chaos: Interdiscip J Nonlinear Sci 2022;32:093125.
[118] Kraemer KH, Marwan N. Border effect corrections for diagonal line based recurrence quantification analysis measures. Phys Lett A 2019;383(34):125977.
[119] Little MA, McSharry PE, Roberts SJ, Costello DA, Moroz IM. Exploiting nonlinear recurrence and fractal scaling properties for voice disorder detection. BioMed Eng OnLine 2007;6:23.
[120] Baptista MS, Ngamga EJ, Pinto PRF, Brito M, Kurths J. Kolmogorov–Sinai entropy from recurrence times. Phys Lett A 2010;374(9):1135–40.
[121] Gabrick EC, Sales MR, Sayari E, Trobia J, Lenzi EK, Borges FS, Szezech Jr JD, Iarosz KC, Viana RL, Caldas IL, Batista AM. Fractional dynamics and recurrence analysis in cancer model. Braz J Phys 2023;53:145.
[122] Donges JF, Heitzig J, Beronov B, Wiedermann M, Runge J, Feng QY, Tupikina L, Stolbova V, Donner RV, Marwan N, Dijkstra HA, Kurths J. Unified functional network and nonlinear time series analysis for complex systems science: The pyunicorn package. Chaos: Interdiscip J Nonlinear Sci 2015;25:113101.
[123] Harle M, Feudel U. Hierarchy of islands in conservative systems yields multimodal distributions of FTLEs. Chaos Solitons Fractals 2007;31(1):130–7.
[124] Hurst HE. Long-term storage capacity of reservoirs. Trans Am Soc Civ Eng 1951;116(1):770–99.
[125] Geweke J, Porter-Hudak S. The estimation and application of long memory time series models. J Time Series Anal 1983;4(4):221–38.
[126] Peng C-K, Buldyrev SV, Havlin S, Simons M, Stanley HE, Goldberger AL. Mosaic organization of DNA nucleotides. Phys Rev E 1994;49:1685–9.
[127] Alessio E, Carbone A, Castelli G, Frappietro V. Second-order moving average and scaling of stochastic time series. Eur Phys J B 2002;27(2):197–200.
[128] Zhang H-Y, Feng Z-Q, Feng S-Y, Zhou Y. Typical algorithms for estimating hurst exponent of time sequence: A data analyst's perspective. IEEE Access 2024;12:185528–56.

[129] Mandelbrot BB, Wallis JR. Noah, joseph, and operational hydrology. Water Resour Res 1968;4(5):909–18.
[130] Mandelbrot BB, Wallis JR. Robustness of the rescaled range R/S in the measurement of noncyclic long run statistical dependence. Water Resour Res 1969;5(5):967–88.
[131] Borin D. Hurst exponent: A method for characterizing dynamical traps. Phys Rev E 2024;110.
[132] Alligood KT, Sauer T, Yorke JA. Chaos: an introduction to dynamical systems. New York: Springer; 2000.
[133] Péntek A, Toroczkai Z, Tél T, Grebogi C, Yorke JA. Fractal boundaries in open hydrodynamical flows: Signatures of chaotic saddles. Phys Rev E 1995;51:4076–88.
[134] Birkhoff GD. Dynamical systems. vol. 9, American Mathematical Soc; 1927.
[135] Hobson D. An efficient method for computing invariant manifolds of planar maps. J Comput Phys 1993;104(1):14–22.
[136] Greene JM. Two-dimensional measure-preserving mappings. J Math Phys 1968;9:760–8.
[137] Greene JM. A method for determining a stochastic transition. J Math Phys 1979;20:1183–201.
[138] Piña E, Jiménez Lara L. On the symmetry lines of the standard mapping. Phys D: Nonlinear Phenom 1987;26(1):369–78.
[139] Henon M, Heiles C. The applicability of the third integral of motion: Some numerical experiments. Astron J 1964;69:73.
[140] Aguirre J, Vallejo JC, Sanjuán MAF. Wada basins and chaotic invariant sets in the Hénon-Heiles system. Phys Rev E 2001;64:066208.
[141] Aguirre J, Vallejo JC, Sanjuan MA. Wada basins and unpredictability in Hamiltonian and dissipative systems. Internat J Modern Phys B 2003;17(22-24 II):4171–5.
[142] Custódio MS, Beims MW. Intrinsic stickiness and chaos in open integrable billiards: Tiny border effects. Phys Rev E 2011;83:056201.
[143] Dettmann CP, Leonel ED. Escape and transport for an open bouncer: Stretched exponential decays. Phys D: Nonlinear Phenom 2012;241:403–8.
[144] Vallejo JC, Nieto AR, Seoane JM, Sanjuán MAF. Fast and slow escapes in forced chaotic scattering: The Newtonian and the relativistic regimes. Phys Rev E 2025;111:024212.
[145] Mugnaine M, Batista AM, Caldas IL, Szezech JD, Viana RL. Ratchet current in nontwist Hamiltonian systems. Chaos: Interdiscip J Nonlinear Sci 2020;30:093141.
[146] Nieto AR, Capeáns R, Sanjuán MAF. Systematic search for islets of stability in the standard map for large parameter values. Nonlinear Dynam 2024;4.
[147] Rolim Sales M, Borin D, de Souza LC, Szezech Jr JD, Viana RL, Caldas IL, Leonel ED. Ratchet current and scaling properties in a nontwist mapping. Chaos Solitons Fractals 2024;189:115614.
[148] Rolim Sales M, Borin D, da Costa DR, Szezech J, Danilo José, Leonel ED. An investigation of escape and scaling properties of a billiard system. Chaos: Interdiscip J Nonlinear Sci 2024;34:113122.
[149] Simile Baroni R, de Carvalho RE, Szezech Junior JD, Caldas IL. Transport barriers and directed transport in the rational standard nontwist map. Phys Rev E 2025;111:034203.
[150] Grebogi C, McDonald SW, Ott E, Yorke JA. Final state sensitivity: An obstruction to predictability. Phys Lett A 1983;99(9):415–8.
[151] McDonald SW, Grebogi C, Ott E, Yorke JA. Fractal basin boundaries. Phys D: Nonlinear Phenom 1985;17(2):125–53.
[152] Grebogi C, Ott E, Yorke JA. Chaos, strange attractors, and fractal basin boundaries in nonlinear dynamics. Science 1987;238(4827):632–8.
[153] Aguirre J, Viana RL, Sanjuán MAF. Fractal structures in nonlinear dynamics. Rev Modern Phys 2009;81:333–86.
[154] Altmann EG, Tél T. Poincaré recurrences and transient chaos in systems with leaks. Phys Rev E 2009;79:016204.
[155] de Oliveira JA, Bizão RA, Leonel ED. Finding critical exponents for two-dimensional Hamiltonian maps. Phys Rev E 2010;81:046212.
[156] Leonel ED, de Oliveira JA, Saif F. Critical exponents for a transition from integrability to non-integrability via localization of invariant tori in the Hamiltonian system. J Phys A 2011;44:302001.
[157] Borin D, Livorati ALP, Leonel ED. An investigation of the survival probability for chaotic diffusion in a family of discrete Hamiltonian mappings. Chaos Solitons Fractals 2023;175:113965.
[158] Weiss JB. Transport and mixing in traveling waves. Phys Fluids A: Fluid Dyn 1991;3:1379–84.
[159] Pierrehumbert RT. Large-scale horizontal mixing in planetary atmospheres. Phys Fluids A: Fluid Dyn 1991;3:1250–60.
[160] Souza LC, Mathias AC, Haerter P, Viana RL. Basin entropy and shearless barrier breakup in open non-twist Hamiltonian systems. Entropy 2023;25:1142.
[161] del Castillo-Negrete D, Greene JM, Morrison PJ. Area preserving nontwist maps: periodic orbits and transition to chaos. Phys D: Nonlinear Phenom 1996;91(1):1–23.
[162] del Castillo-Negrete D, Greene JM, Morrison PJ. Renormalization and transition to chaos in area preserving nontwist maps. Phys D: Nonlinear Phenom 1997;100(3):311–29.
[163] Mathias AC, Mugnaine M, Santos MS, Szezech JD, Caldas IL, Viana RL. Fractal structures in the parameter space of nontwist area-preserving maps. Phys Rev E 2019;100:052207.
[164] Szezech Jr JD, Caldas IL, Lopes SR, Viana RL, Morrison PJ. Transport properties in nontwist area-preserving maps. Chaos: Interdiscip J Nonlinear Sci 2009;19:043108.
[165] Mugnaine M, Mathias AC, Santos MS, Batista AM, Szezech JD, Viana RL. Dynamical characterization of transport barriers in nontwist Hamiltonian systems. Phys Rev E 2018;97:012214.
[166] Szezech JD, Caldas IL, Lopes SR, Morrison PJ, Viana RL. Effective transport barriers in nontwist systems. Phys Rev E 2012;86:036206.
[167] Mugnaine M, Caldas IL, Szezech JD, Viana RL, Morrison PJ. Shearless effective barriers to chaotic transport induced by even twin islands in nontwist systems. Phys Rev E 2024;110:044201.
[168] Grime GC, Caldas IL, Viana RL, Elskens Y. Effective transport barriers in the biquadratic nontwist map. Phys Rev E 2025;111:014219.
[169] Menck PJ, Heitzig J, Marwan N, Kurths J. How basin stability complements the linear-stability paradigm. Nat Phys 2013;9(2):89–92.
[170] Daza A, Wagemakers A, Georgeot B, Guéry-Odelin D, Sanjuán MAF. Basin entropy: a new tool to analyze uncertainty in dynamical systems. Sci Rep 2016;6(1):31416.
[171] Daza A, Wagemakers A, Sanjuán MAF. Classifying basins of attraction using the basin entropy. Chaos Solitons Fractals 2022;159:112112.
[172] Rossler OE. An equation for hyperchaos. Phys Lett A 1979;71(2):155–7.
[173] Moges H, Manos T, Racoveanu O, Skokos C. On the behavior of the generalized alignment index (GALI) method for dissipative systems. Int J Bifurc Chaos 2025;35.
[174] Feudel U, Grebogi C. Multistability and the control of complexity. Chaos: Interdiscip J Nonlinear Sci 1997;7:597–604.
[175] Feudel U. Complex dynamics in multistable systems. Int J Bifurc Chaos 2008;18(06):1607–26.
[176] Pisarchik AN, Feudel U. Control of multistability. Phys Rep 2014;540(4):167–218, Control of multistability.
[177] Pisarchik AN, Hramov AE. Multistability in physical and living systems. Cham: Springer; 2022.
[178] Bi M-X, Fan H, Yan X-H, Lai Y-C. Folding state within a hysteresis loop: Hidden multistability in nonlinear physical systems. Phys Rev Lett 2024;132:137201.
[179] Daza Á, Wagemakers A, Sanjuán MAF. Multistability and unpredictability. Phys Today 2024;77:44–50.
[180] Wagemakers A. Basins of attraction: A dynamical zoo. Int J Bifurc Chaos 2025;35(11):2530024.
[181] Arecchi FT, Badii R, Politi A. Generalized multistability and noise-induced jumps in a nonlinear dynamical system. Phys Rev A 1985;32:402–8.
[182] Englisch V, Lauterborn W. Regular window structure of a double-well duffing oscillator. Phys Rev A 1991;44:916–24.
[183] Prando Livorati AL, Paganotti Faber A, Borin D. Asymptotic convergence for the dynamics of a Duffing-like oscillator under scaling analyses. Chaos: Interdiscip J Nonlinear Sci 2025;35:013108.

[184] Leutcho GD, Gandubert G, Woodward L, Blanchard F. Electric-field-biased control of irregular oscillations via multistability in a nonlinear terahertz meta-atom. Chaos Solitons Fractals 2025;198:116586.
[185] Ngonghala CN, Feudel U, Showalter K. Extreme multistability in a chemical model system. Phys Rev E 2011;83:056206.
[186] Meucci R, Marc Ginoux J, Mehrabbeik M, Jafari S, Clinton Sprott J. Generalized multistability and its control in a laser. Chaos: Interdiscip J Nonlinear Sci 2022;32:083111.
[187] Kim S, Park SH, Ryu CS. Multistability in coupled oscillator systems with time delay. Phys Rev Lett 1997;79:2911–4.
[188] Cheng C-Y, Lin K-H, Shih C-W. Multistability in recurrent neural networks. SIAM J Appl Math 2006;66(4):1301–20.
[189] Pedregosa F, Varoquaux G, Gramfort A, Michel V, Thirion B, Grisel O, Blondel M, Prettenhofer P, Weiss R, Dubourg V, Vanderplas J, Passos A, Cournapeau D, Brucher M, Perrot M, Duchesnay E. Scikit-learn: Machine learning in Python. J Mach Learn Res 2011;12:2825–30.
[190] Peitgen H-O, Jürgens H, Saupe D, Feigenbaum MJ. Chaos and fractals: new frontiers of science. second ed.. Springer New York, NY; 2004.
[191] Tél T, Gruiz M. Chaotic dynamics: an introduction based on classical mechanics. Cambridge University Press; 2006.
[192] Motter AE, Gruiz M, Károlyi G, Tél T. Doubly transient chaos: Generic form of chaos in autonomous dissipative systems. Phys Rev Lett 2013;111:194101.
[193] Shinbrot T, Grebogi C, Wisdom J, Yorke JA. Chaos in a double pendulum. Am J Phys 1992;60:491–9.
[194] Stachowiak T, Okada T. A numerical analysis of chaos in the double pendulum. Chaos Solitons Fractals 2006;29(2):417–22.
[195] Tufillaro NB, Abbott TA, Griffiths DJ. Swinging Atwood's machine. Am J Phys 1984;52:895–903.
[196] Szumiński W, Maciejewski AJ. Dynamics and integrability of the swinging Atwood machine generalisations. Nonlinear Dynam 2022;110(3):2101–28.
[197] Broucke R, Baxa PA. Periodic solutions of a spring-pendulum system. Celest Mech 1973;8(2):261–7.
[198] Szumiński W, Maciejewski AJ. Dynamics and non-integrability of the double spring pendulum. J Sound Vib 2024;589:118550.
[199] Verlet L. Computer experiments on classical fluids, I. Thermodynamical properties of Lennard-Jones molecules. Phys Rev 1967;159:98–103.
[200] Yoshida H. Construction of higher order symplectic integrators. Phys Lett A 1990;150:262–8.
[201] Skokos C, Gerlach E. Numerical integration of variational equations. Phys Rev E 2010;82:036704.
[202] Hénon M, Heiles C. The applicability of the third integral of motion: Some numerical experiments. Astron J 1964;69:73.
[203] Rolim Sales M. Pynamicalsys documentation. 2025, [Accessed 16 June 2025].
[204] Rolim Sales M. Supplementary material. 2025.
[205] Besançon L, Peiffer-Smadja N, Segalas C, Jiang H, Masuzzo P, Smout C, Billy E, Deforet M, Leyrat C. Open science saves lives: lessons from the covid-19 pandemic. BMC Med Res Methodol 2021;21(1):117.
[206] Gomes DGE, Pottier P, Crystal-Ornelas R, Hudgins EJ, Foroughirad V, Sánchez-Reyes LL, Turba R, Martinez PA, Moreau D, Bertram MG, Smout CA, Gaynor KM. Why don't we share data and code? Perceived barriers and benefits to public archiving practices. Proc R Soc B: Biol Sci 2022;289(1987):20221113.
[207] Gabelica M, Bojčić R, Puljak L. Many researchers were not compliant with their published data sharing statement: a mixed-methods study. J Clin Epidemiol 2022;150:33–41.
[208] Baker M. 1 500scientists lift the lid on reproducibility. Nature 2016;533(7604):452–4.
[209] Miyakawa T. No raw data, no science: another possible source of the reproducibility crisis. Mol Brain 2020;13(1):24.
[210] Dhooge A, Govaerts W, Kuznetsov YA, Meijer HG, Sautois B. New features of the software MatCont for bifurcation analysis of dynamical systems. Math Comput Model Dyn Syst 2008;14(2):147–75.
[211] Datseris G. Dynamicalsystems. jl: A Julia software library for chaos and nonlinear dynamics. J Open Source Softw 2018;3:598.
[212] Datseris G, Parlitz U. Nonlinear dynamics: a concise introduction interlaced with code. Cham, Switzerland: Springer Nature; 2022.
[213] Ansmann G. Efficiently and easily integrating differential equations with JiTCODE, JiTCDDE, and JiTCSDE. Chaos: Interdiscip J Nonlinear Sci 2018;28:043116.
[214] Pessa AAB, Ribeiro HV. Ordpy: A python package for data analysis with permutation entropy and ordinal network methods. Chaos: Interdiscip J Nonlinear Sci 2021;31.
[215] Hegger R, Kantz H, Schreiber T. Practical implementation of nonlinear time series methods: The TISEAN package. Chaos: Interdiscip J Nonlinear Sci 1999;9:413–35.
[216] Alstott J, Bullmore E, Plenz D. Powerlaw: A python package for analysis of heavy-tailed distributions. PLoS One 2014;9:1–11.
[217] Rolim Sales M. Mrolims/pynamicalsys. 2025.