Eur. Phys. J. Spec. Top. (2026) 235:9425–9439
https://doi.org/10.1140/epjs/s11734-026-02389-6

THE EUROPEAN PHYSICAL JOURNAL SPECIAL TOPICS

Regular Article

An optimized Python library for analyzing dynamical systems with recurrence microstates

Gabriel Vinicius Ferreira[1,2,a], Felipe Eduardo Lopes da Cruz[1,2], Gabriel Marghoti[1,2], Thiago de Lima Prado[1,2,b], Sergio Roberto Lopes[1,2], Nobert Marwan[3,4,5], and Jürgen Kurths[3,6]

[1] Physics Department, Federal University of Paraná, Curitiba, Paraná 81531-990, Brazil
[2] Interdisciplinary Center for Science, Technology and Innovation (CICTI), Federal University of Paraná, Curitiba, Brazil
[3] Potsdam Institute for Climate Impact Research (PIK), Member of the Leibniz Association, 14412 Potsdam, Germany
[4] Institute for Geosciences, University of Potsdam, 14476 Potsdam, Germany
[5] Institute for Physics and Astronomy, University of Potsdam, 14476 Potsdam, Germany
[6] Humboldt Universität zu Berlin, 10099 Berlin, Germany

Received 23 October 2025 / Accepted 27 April 2026 / Published online 3 June 2026
© The Author(s) 2026

Abstract Recurrence analysis techniques based on recurrence plots (RPs) have become a standard tool for the study of dynamical systems, offering means to quantify nonlinear characteristics from time series and to apply machine-learning methods in their analysis. Among recent developments, recurrence microstates analysis (RMA) is emerging as a complementary approach, enabling the estimation of typical quantifiers from recurrence quantification analysis (RQA) using the microstate distributions, as well as the use of these distributions as input features for machine-learning models. In this work, we introduce a Python library designed to provide the computational efficiency required for the optimized application of RMA techniques. The proposed methodology and performance benchmarks demonstrate a substantial reduction in computation time, making the library sustainable and energy efficient, especially for large-scale data analysis, and aligned with green computing principles.

1 Introduction

Recurrence microstates analysis (RMA) is emerging as an efficient tool for studying dynamical systems. One important advantage is the possibility to estimate typical quantifiers of recurrence quantification analysis (RQA) [1] at low computational costs. The distribution of microstates can also be used as input features for simple machine-learning models—i.e. multi-layer perceptrons (MLPs) [2]—instead of convolutional neural networks (CNNs), which are the standard approach when working directly with recurrence plots (RPs) [3–5]. Furthermore, RMA enables the analysis of stochastic behavior using triangle motifs, as proposed by Hirata [6], or through the disorder quantification proposed by Flauzino et al. [7]. It also facilitates the extraction of underlying dynamical properties from univariate or multivariate time series, such as the geometry of attractors in terms of involution symmetry quantification [8].

RMA is based on recurrence plots (RPs), an approach introduced by Eckmann et al. [9] for studying recurrences in time series, enabling the qualitative visualization and analysis of dynamical systems. The RPs and their quantification (recurrence quantification analysis, RQA) [10] have already been applied in many scientific and technological fields, such as climate [11], health [12, 13], physics [14], and even image analysis [15]. The RMA is a completely new approach that aims to extracts information from the distribution of small structures identified within the RP. These structures, referred to as recurrence microstates or recurrence motifs, were first introduced by Corso et al. [16] to define recurrence entropy.

[a] e-mail: gabriel.vferreira@icloud.com
[b] e-mail: thiago@fisica.ufpr.br (corresponding author)

Given a trajectory $\vec{x}_i \in \mathbb{R}^d$, where $i \in \{1, 2, ..., K\}$, d denotes the dimension of the coordinate space, and K is the length of the analyzed time series, the RP is constructed from the recurrence matrix $\mathbf{R}$, defined as

$$R_{i,j} = \Theta(\varepsilon - \|\vec{x}_i - \vec{x}_j\|), \tag{1}$$

where Θ is the Heaviside function, $\|\cdot\|$ denotes an appropriate norm, and ε is the threshold parameter that defines the maximum distance at which two states are considered ε-recurrent.

Based on $\mathbf{R}$, a motif is a small structure composed of ones (recurrences) and zeros (non-recurrences) with a specific constrained shape. For example, for a given position $(m, n) \in \{1, 2, ..., K - (k-1)\}^2$, where k denotes the motif size, a square-shaped motif can be defined as

$$\mathbf{M}(m, n) = \begin{pmatrix} R_{m,n} & \cdots & R_{m,n+k-1} \\ \vdots & \ddots & \vdots \\ R_{m+k-1,n} & \cdots & R_{m+k-1,n+k-1} \end{pmatrix}. \tag{2}$$

Initially, these recurrence motifs were defined using square-shaped templates. However, as the method evolved, it became evident that variations in motif shape can be used to extract different types of information from the RP. For instance, triangular motifs have been employed to analyze stochastic properties [6], while diagonal and line motifs are useful for estimating determinism (DET) and laminarity (LAM) [17].

Moreover, the strategy used to define the positions (m, n) from which each motif is extracted can vary depending on the research objective. For example, in the computation of involution symmetry [8], a column-wise sampling approach is adopted to obtain the required information.

Motivated by the growing development of the RMA field, we adapted the methodology implemented in our Julia package, `RecurrenceMicrostatesAnalysis.jl` [17], into a Python library that enables the use of RMA techniques with an optimized computational process. This library was written in `C++`, using `pybind11` [18] as a binding interface to integrate the code with the Python interpreter. Despite the increased complexity of the source code, we preserved our original goal of designing a user-friendly library for recurrence microstates analysis (RMA).

2 Theoretical basis

The Python library is organized into two components:

(i) **Core**—responsible for computing the distributions; and
(ii) **Quantification functions**—which use the distributions to calculate RQA.

2.1 Motifs

The main objective of our method as a fast computational process is to compute the probability distributions without explicitly constructing the RP. Building an RP can be computationally expensive, especially for long time series. Therefore, calculating the recurrence motif distributions directly from the trajectory provides an efficient and memory-conscious alternative.

This approach avoids the overhead of allocating large memory blocks—as an RP for a time series of length K contains $K \times K$ elements—and reduces computation time, since in most cases, only about 5% of the total number of motifs in the RP need to be processed for RMA applications. As a result, the library aligns with the principles of green computing [19], minimizes both memory usage and energy consumption, contributing to more environmentally sustainable scientific computing.

With this in mind, the core of the library aims to compute each motif from a trajectory and count the occurrences of each identified motif to construct a histogram. Since a motif is a binary structure—a small structure from an RP—it is possible to represent each motif as a binary number formed by linearizing its elements. Using, as an example, a square 2×2 motif, defined as

$$\begin{pmatrix} \xi_{0,0} & \xi_{0,1} \\ \xi_{1,0} & \xi_{1,1}, \end{pmatrix} \tag{3}$$

where each element is defined by $\xi_{p,q} := R_{m+p,n+q}$ for given indices (m, n) as in Eq. (2), we can represent this motif as the binary number $\xi_{0,0}\xi_{0,1}\xi_{1,0}\xi_{1,1}$.

Since it is uncommon in computation to use binary values directly as indices—because doing so can make the code harder to understand—it is good practice to convert these binary representations to decimal numbers. The library uses an interface `IShape` to define how this conversion process is performed. Each supported motif shape inherits from this interface, using object properties to simplify the implementation.

When a shape is defined, it is necessary to calculate the number of elements inside the motif's constrained structure to define the power vector. For a square-shaped motif, this value is equal to $\sigma = k \times k$. From it, we construct the power vector, $\mathbf{P}$, as a vector with σ elements written as

$$P_s = 2^{s-1}, \quad s \in \{1, 2, \ldots, \sigma\}. \tag{4}$$

This process is done in the class constructor, such as demonstrated into Listing (1), where the function `compute_area()` is responsible for computing σ.

```
Square::Square ([arguments]) : IShape([arguments]) {
    //      Compute the motif area.
    motif_area = compute_area();

    //      Define and compute the power vector.
    power_vector = std::vector(motif_area, 1);
    for (int i = 0; i < motif_area; i++) {
        power_vector[i] = static_cast<int>(std::pow(2, i));
    }
}
```

Listing 1. Constructor of the square motif constrained shape into C++ library source.

The process of constructing a motif and identifying it as a decimal number is handled by the function `get_index([arguments])`. This function receives a position (m, n) provided by the sampling structure and constructs the motif from the trajectory based on this position, following the corresponding combinations of (p, q). For example, using $k = 3$, we can define square, triangular, and diagonal motifs constrained as

$$\begin{pmatrix} \xi_{0,0} & \xi_{0,1} & \xi_{0,2} \\ \xi_{1,0} & \xi_{1,1} & \xi_{1,2} \\ \xi_{2,0} & \xi_{2,1} & \xi_{2,2} \end{pmatrix}, \quad \begin{pmatrix} \xi_{0,0} & \xi_{0,1} & \xi_{0,2} \\ & \xi_{1,1} & \xi_{1,2} \\ & & \xi_{2,2} \end{pmatrix}, \quad \begin{pmatrix} \xi_{0,0} & & \\ & \xi_{1,1} & \\ & & \xi_{2,2} \end{pmatrix}. \tag{5}$$

The function associates each of these positions with an index s of the power vector $\mathbf{P}$, that is, using $\xi_s \leftrightarrow \xi_{p,q}$, we can rewrite the matrices from Eq. (5) as

$$\begin{pmatrix} \xi_1 & \xi_2 & \xi_3 \\ \xi_4 & \xi_5 & \xi_6 \\ \xi_7 & \xi_8 & \xi_9 \end{pmatrix}, \quad \begin{pmatrix} \xi_1 & \xi_2 & \xi_4 \\ & \xi_3 & \xi_5 \\ & & \xi_6 \end{pmatrix}, \quad \begin{pmatrix} \xi_1 & & \\ & \xi_2 & \\ & & \xi_3 \end{pmatrix}. \tag{6}$$

Based on this structure, we compute the index I that represents the motif as

$$I(m, n) = \sum_{s=1}^{\sigma} R_{m+p_s, n+q_s} P_s, \tag{7}$$

where each (p_s, q_s) pair is associated with index s, as defined by the motif's constrained shape, exemplified in Eqs. (5) and (6). There are $N = 2^{\sigma}$ possible distinct combinations of recurrence within a motif structure. Accordingly, we define N distinct motif constrained structures, denoted as $\mathbf{M}^{(I)}$, where I is the structure identifier and is given by Eq. (7) for each possible structure, in such a way that $I \in \{0, 1, \ldots, N-1\}$.

In the library context, it is necessary to define each of these structures within the logic of the function `get_index([arguments])`, such as the example of Listing (2), where the index of a square-shaped motif is computed. It is important to note that this code includes an implementation for correct memory access to a `NumPy` array [20], since it is the input format passed to the library.

```
\\       [arguments]
\\  idx: define the pair (m,n)
\\  itr: memory pre-allocated to compute (m+p,n+q)
const ssize_t Square::get_index(const std::vector<ssize_t> &idx,
    std::vector<ssize_t> &itr) const {
    ssize_t index = 0;
    //  Copy values from idx to itr
    std::ranges::copy(idx, itr.begin());

    for (const int m : power_vector) {
        ssize_t x_idx = 0;
        ssize_t y_idx = 0;

        for (ssize_t i = 1; i < x_info.ndim; i++) {
            x_idx += itr[i-1] * x_strides[i-1]; //  Calculate
                the position in the memory used by NumPy's array
        }

        for (ssize_t i = 1; i < y_info.ndim; i++) {
            y_idx += itr[(i-1] + (x_info.ndim - 1)] *
                y_strides[i-1]; //  Same thing
        }

        //  Compute the recurrence
        if (recurrence->compute(x, y, fd_shape, fd_stride,
            x_idx, y_idx))
            index += m;

        //  Change the values (p,q)
        itr[0]++;
        for (ssize_t k = 0; k < structure.size() - 1; k++) {
            if (itr[k] > idx[k] + (structure[k] - 1)) {
                itr[k] = idx[k];
                itr[k + 1]++;
                continue;
            }
            break;
        }
    }

    return index;
}
```

Listing 2. Function in charge to construct a square motif from the trajectory and converts it to a decimal representation.

Each position (m, n) is defined by classes that inherit the interface `ISampling`. Usually, we use the sampling mode `Random`—that selects (m, n) randomly from the RP—, or `Full`—that takes all motifs contained into the RP—, although there are other sampling strategies. To improve the library speed, we use asynchronous processing at this stage, splitting the work between the available threads. With this goal, it is defined a task function is defined to compute (m, n) within a specific range. For example, Listing (3) shows this for square-shaped motifs. This task function is called by a function `run([number of threads])`, which divides the work between the available threads. The function `run` passes the `begin` and `end` values to the task function, defining the range of positions that each thread will process.

```
//      [arguments]
//  begin and end are used to define where the task needs to
    start, and finish. In this case, we just define the number
    of samples by end - begin
std::vector<double> Random::task(const ssize_t begin, const
    ssize_t end) const {
    //  Allocate memory. (N = numb_motifs())
    std::vector<double> result(shape->numb_motifs(), 0);

    std::vector<ssize_t> idx(structure_size, 0);
    std::vector<ssize_t> itr(structure_size, 0);

    //  Random generator
    static std::random_device rd;
    static std::mt19937 gen(rd());

    //  Compute each sample
    for (ssize_t i = begin; i <= end; i++) {
        for (ssize_t s = 0; s < rp_structure.size(); ++s) {
            idx[s] = std::uniform_int_distribution <ssize_t>(1,
                rp_structure[s])(gen);
        }

        //  Call get_index() from shape
        result[shape->get_index(idx, itr)]++;
    }

    return result;
}
```

Listing 3. Function in charge to define each motif position (m,n), used as base to build it.

2.2 Recurrence quantification analysis (RQA)

Recurrence quantification analysis (RQA) is a powerful tool used to analyze nonlinear time series. Quantifiers such as recurrence rate (RR), determinism (DET), and laminarity (LAM) are usually used to quantify statistics over internal structures of the RP.

The simplest quantifier to define when working with RMA is the recurrence entropy [16], given by

$$RME = -\sum_{I=1}^{N} p_I \ln(p_I), \tag{8}$$

where p_I is the probability of observing a specific motif I in the RP.

The RR is the overall density of points within an RP, and is calculated as [10]

$$RR = \frac{1}{K^2} \sum_{i,j=1}^{K} R_{i,j}. \tag{9}$$

When working with RMA, we can estimate RR when considering that it is the mean of the individual RR values of each motif. We define it as

$$RR \approx \sum_{I=1}^{N} p_I^{(k)} RR_I, \tag{10}$$

where RR_I is the recurrence rate of a motif constrained structure, calculated as

$$RR_I = \frac{1}{\sigma} \sum_{s=1}^{\sigma} M_{p_s,q_s}^{(I)}, \tag{11}$$

denoting $\mathbf{M}^{(I)}$ as the motif constrained structure.

The computational implementation follows a similar logic, but converting the identifier I to a binary number and uses it to compute the local recurrence rate, RR_I, as demonstrate in Listing (4).

```cpp
//  Compute the local recurrence rate (RR_I)
inline double local_rr(ssize_t n, const ssize_t hv) {
    ssize_t sum_bits = 0;
    while (n > 0) {
        sum_bits += n & 1;      //  Sum 1 if the low bit is 1.
        n >>= 1;                //  Move n 1 bit to right.
    }

    return static_cast<double>(sum_bits) /
        static_cast<double>(hv);
}

//  Compute the recurrence rate from a recurrence motifs
    distribution
inline double recurrence_rate(const std::vector<double>& dist) {
    double result = 0.0;

    //          Get the "motif constrained structure area"
    const auto area = static_cast<ssize_t>
        (std::floor(log2(dist.size())));

    for (int i = 0; i < dist.size(); ++i) {
        const double rr_i = local_rr(i, area);
        result += rr_i * dist[i];
    }
    //          Return the recurrence rate
    return result;
}
```

Listing 4. Function to compute the recurrence rate from a recurrence motifs distribution.

The DET is a rate of points that are part of diagonal lines. It is calculated as [10]

$$DET = \frac{\sum_{\ell=\ell_{min}}^{K} \ell H(\ell)}{\sum_{\ell=1}^{K} \ell H(\ell)}, \tag{12}$$

where $H(\ell)$ is the number of diagonal lines of the length ℓ, calculated as

$$H(\ell) = \sum_{i,j=1}^{K-\ell-1} (1 - R_{i,j})(1 - R_{i+\ell+1,j+\ell+1}) \prod_{k=1}^{\ell} R_{i+k,j+k}. \tag{13}$$

Typically, the minimum length ℓ_{min} is set to 2, which is the simplest choice.

The LAM is similar to the DET, but quantify the proportion of points that are part of horizontal (or vertical) lines, rather than diagonal lines. It is calculated as [10]

$$LAM = \frac{\sum_{v=v_{min}}^{K} vH(v)}{\sum_{v=1}^{K} vH(v)}, \tag{14}$$

where $H(v)$ is the number of lines of length v, defined as

$$H(v) = \sum_{i=1}^{K-v-1} \sum_{j=1}^{K} (1 - R_{i,j})(1 - R_{i+v+1,j}) \prod_{k=1}^{v} R_{i+k,j}. \tag{15}$$

The process to compute DET and LAM using RMA was introduced by da Cruz et al. [1] who expand the terms of $H(l)$ and $H(v)$, making it is possible to write

$$DET \approx 1 - \frac{\mathbf{d} \cdot (\mathcal{R}^{(3)}\mathbf{p}^{(3)})}{RR}, \tag{16}$$

$$LAM \approx 1 - \frac{\mathbf{v} \cdot (\mathcal{R}^{(3)}\mathbf{p}^{(3)})}{RR}, \tag{17}$$

where $\mathcal{R}^{(k)}$ is used to obtain terms of the type

$$\sum_{i,\,j=1}^{K-N+1} \frac{R_{i+a,\,j+b} \ \ldots \ R_{i+y,\,j+z}}{(K-N+1)^2} \tag{18}$$

from the motif's distribution $\mathbf{p}^{(k)}$, and, $\mathbf{d}$ and $\mathbf{v}$ are used to select the terms necessary to calculate the histograms $H(\ell)$ and $H(v)$ of Eqs. (13) and (14) with $\ell = v = 1$.

The library also implements the optimization methodology using diagonal and line motifs [17], which simplifies the computation. This means that DET can be estimated as

$$DET \approx 1 - \frac{p_2^{(3)}}{RR}, \tag{19}$$

where $\bar{p}^{(3)}$ needed to be a distribution of diagonal-shaped motifs with $k = 3$, and the needed motif

$$\begin{pmatrix} 0 & & \\ & 1 & \\ & & 0 \end{pmatrix}, \tag{20}$$

corresponds to $I = 2$.

Although it is possible to generalize it for any $\ell_{min} \geq 2$, the function is currently defined for $\ell_{min} = 2$ to simplify the initial implementation, while it can be adapted it in the future to any suitable configuration. The computational implementation is shown in Listing (5).

```
inline double determinism(const py::array_t<double> &x, const
    py::object &threshold, const unsigned int threads = 0) {
    //          Compute the distribution
    //  - k = 3
    //  - sampling rate = 5% (0.05)
    const py::array_t<double> dist = distribution(x, threshold,
        3, 0.05, threads, SamplingMode::Random,
        ShapeName::Diagonal);

    const py::buffer_info info = dist.request();
    const auto ptr = static_cast<double *>(info.ptr);

    //          Compute the recurrence rate
    const double rr = recurrence_rate(std::vector(ptr, ptr +
        info.shape[0]));

    //          Compute determinism
    return 1 - ((1 / rr) * ptr[2]);
}
```

Listing 5. Function to compute the determinism from a trajectory.

The methodology to estimate LAM with RMA is similar to that used to DET, but uses line-shaped motifs instead of diagonal motifs [1, 17]. Therefore, LAM is estimated as

$$LAM \approx 1 - \frac{p_2^{(3)}}{RR}, \tag{21}$$

where $\vec{p}^{(3)}$ needed to be a distribution of line-shaped motifs with $k = 3$, and the needed motif

$$\begin{pmatrix} 0 & 1 & 0 \end{pmatrix}, \tag{22}$$

which also corresponds to $I = 2$. The computational implementation is shown in Listing (6).

```
inline double laminarity(const py::array_t<double> &x, const
    py::object &threshold, const unsigned int threads = 0) {
    //          Compute the distribution
    //  - k = {lines: 3, columns: 1} (vertical lines)
    //  - sampling rate = 5% (0.05)
    const py::array_t<double> dist = distribution(x, x,
        threshold, std::vector{3, 1}, 0.05, threads);

    const py::buffer_info info = dist.request();
    const auto ptr = static_cast<double *>(info.ptr);
    //          Compute the recurrence rate
    const double rr = recurrence_rate(std::vector(ptr, ptr +
        info.shape[0]));

    //          Compute laminarity
    return 1 - ((1 / rr) * ptr[2]);
}
```

Listing 6. Function to compute the laminarity from a trajectory.

3 The library

The library `recurrence_microstates` is available as an open-source project at GitHub https://github.com/DynamicsUFPR/recurrence-microstates-python and has been designed to be fast and easy-to-use. In this section, we will present how to use the library and its performance results, including the RQA estimation error, and the library computation time.

3.1 Examples

As examples, we use five paradigmatic dynamical systems. A uniform white noise, a logistic map [21], a generalized Bernoulli shifted map [22], the Lorenz system [23], and the Rössler system [24].

The logistic map describes a population growing purely exponentially (for $r > 1$), and is written as [21]

$$x_{n+1} = r x_n (1 - x_n), \tag{23}$$

where r is the population growth factor. To induce chaotic dynamics in our tests, we set $r = 4$.

The generalized Bernoulli shifted map is defined as [22]

$$x_{n+1} = \beta x_n \ \mathbf{mod}\ 1, \tag{24}$$

where **mod** is the operation that restricts x_{n+1} and x_n to values between 0 and 1, and β is the stretching factor that we arbitrarily choose as $\beta = 4.99$, since for this value the map exhibits chaos behavior.

The Lorenz system was originally developed to model atmosphere convection [22, 23], and is described by the three differential equations

$$\frac{\mathrm{d}x}{\mathrm{d}t} = \sigma(y - x), \tag{25}$$

$$\frac{\mathrm{d}y}{\mathrm{d}t} = x(\rho - z) - y, \tag{26}$$

$$\frac{\mathrm{d}z}{\mathrm{d}t} = xy - bz, \tag{27}$$

where ρ is the Rayleigh number, σ is the Prandtl number, and b is a geometrically determined constant. Here, we fix $\rho = 28$, $\sigma = 10$, and $b = 8/3$, which is a standard parameter configuration that produces chaotic behavior.

The Rössler system is also defined by a set of three differential equations [22, 24]

$$\frac{\mathrm{d}x}{\mathrm{d}t} = -y - z, \tag{28}$$

$$\frac{\mathrm{d}y}{\mathrm{d}t} = x + ay, \tag{29}$$

$$\frac{\mathrm{d}z}{\mathrm{d}t} = b + z(x - c), \tag{30}$$

where we set $a = 0.1$, $b = 0.1$, and $c = 18.0$, a well-known parameter set that leads to chaotic dynamics.

The library uses `NumPy`'s arrays as input [20], where the data must be provided in a matrix form, such as

$$\begin{matrix} x_{1,1} & x_{1,2} & \cdots & x_{1,K} \\ x_{2,1} & x_{2,2} & \cdots & x_{2,K} \\ \vdots & \vdots & \ddots & \vdots \\ x_{d,1} & x_{d,2} & \cdots & x_{d,K} \end{matrix}, \tag{31}$$

where each row corresponds to a system component and each column corresponds to a time step.

3.1.1 Computing recurrence motifs distributions

The computation of recurrence motifs distributions from a trajectory is performed by the function `distribution(input, threshold, k)`. This function can be called as illustrated in Listing (7), where we apply the RMA library to a uniform distribution, using a threshold $\varepsilon = 0.27$ that is a tabulated value for a uniform distribution [25], and $k = 2$. It is important to note that when $d = 1$ the library accepts a vector with one-dimension, interpreting it as a matrix $1 \times K$.

```
>>> import numpy as np
>>> import recurrence_microstates as rma

>>> data = np.random.uniform(0, 1, 1000).astype(np.float64)
[array([8.70785113e-01, 0.74211064e-01, ..., 9.68336117e-01]]

>>> dist = rma.distribution(data, 0.27, 2)
[array([0.11052104, 0.05160321, 0.05240481, 0.08192385,
        0.05442886, 0.07997996, 0.08166333, 0.03024048,
        0.0501002 , 0.08098196, 0.08563126, 0.0287976 ,
        0.08478958, 0.02907816, 0.02715431, 0.0707014 ])]
```

Listing 7. Application of the library to a uniform distribution.

By default, the library applies a sampling rate of 5% using random sampling and square-shaped motifs. These settings can be modified through optional parameters, as shown in Listing (8). A more detailed description of all available options is provided in the library's documentation on GitHub.

```
>>> import numpy as np
>>> import recurrence_microstates as rma

>>> data = np.random.uniform(0, 1, 1000).astype(np.float64)
[array([8.70785113e-01, 0.74211064e-01, ..., 9.68336117e-01]]

//      Shapes are given by rma.Shape
>>> dist = rma.distribution(data, 0.27, 2, sampling = 0.1, shape
    = rma.Shape.Triangle)

//      Sampling modes are given by rma.SamplingMode
>>> dist = rma.distribution(data, 0.27, 2, sampling_mode =
    rma.SamplingMode.Full)
array([0.1109558 , 0.05370736, 0.05279153, 0.08189972,
       0.05279153, 0.08189972, 0.08010613, 0.02999797,
       0.05113422, 0.0804889 , 0.08436364, 0.02872242,
       0.08436364, 0.02872242, 0.02747091, 0.0705841 ])
```

Listing 8. Application of the library to a uniform distribution using triangle-shaped motifs and 10% as sampling rate, and square-shaped motifs with a full sampling mode.

3.1.2 Computing RQA

The implementation of RQA computations was designed for ease of use. In this context, we define the threshold `th` as the value that maximizes the recurrence microstates entropy (RME) [26].

The RME is calculated from a motif distribution using Eq. (8), and is implemented in the library by the function `entropy`. An example of its usage for a given time series `data` is shown in Listing (9).

```
>>> dist = rma.distribution(data, th, 3)
array([1.37550201e-02, 3.21285141e-03, 2.61044177e-03, ...,
    2.16867470e-03, 1.98795181e-03, 7.53012048e-03])

>>> rme = rma.entropy(dist)
8.38083237742761
```

Listing 9. Application of the library to compute the RME.

The recurrence rate (RR) is also computed from a motif distribution using the functions shown in Listing (4). In Python, the RR can be calculated as illustrated in Listing (10) for a given time series `data`.

```
>>> dist = rma.distribution(data, th, 3)
array([1.92369478e-02, 2.40963855e-03, 3.03212851e-03,
    2.69076305e-03, ..., 1.80722892e-03, 1.58634538e-03,
    1.05220884e-02])

>>> rr = rma.recurrence_rate(dist)
0.4593529674252569
```

Listing 10. Application of the library to compute the RR.

DET can be estimated directly from a time series using the function `determinism`, which implements the calculation with diagonal-shaped motifs, as introduced in Sect. 2.2. The estimation of LAM is analogous, but uses line-shaped motifs. Both quantifiers are computed with a motif size $k = 3$, which corresponds to $\ell_{min} = v_{min} = 2$. Listing (11) illustrates how to apply the library to estimate these values for a given time series `data`.

```
>>> det = rma.determinism(x, th)
0.7088809297572325

>>> lam = rma.laminarity(x, th)
0.7056565920892619
```

Listing 11. Application of the library to estimate DET and LAM

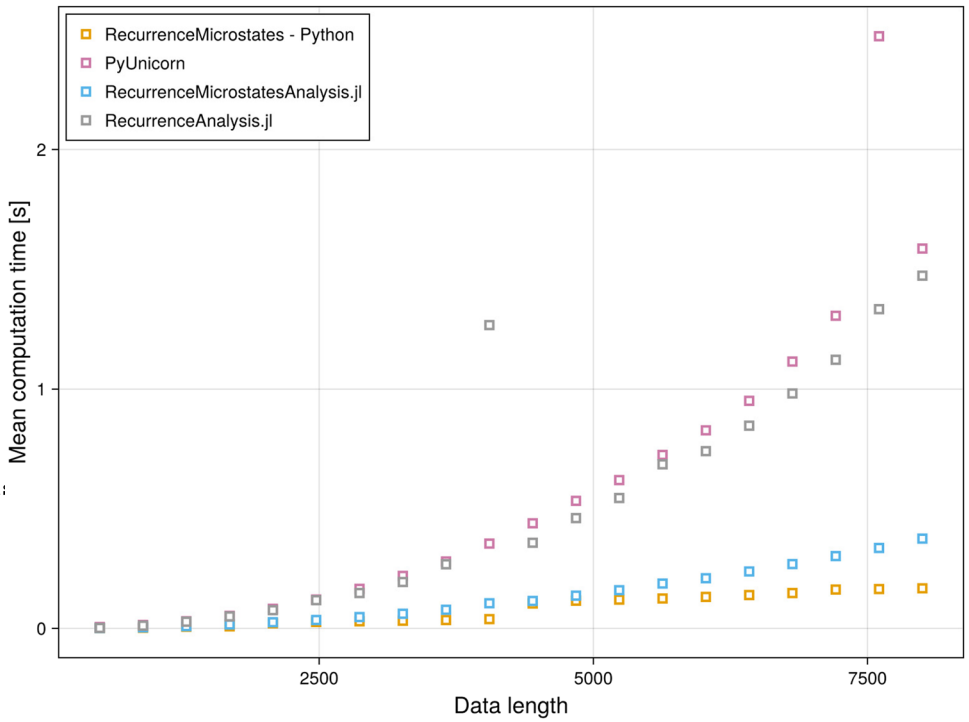


Fig. 1 Mean computation time as a function of the data length K. The values represent the average of a distribution of tests, each computed using a fixed input for the corresponding length K. The reported times include the computation of both determinism and laminarity from the time series. Four libraries are compared: `Recurrence Microstates` (Python, the library introduced in this work), `PyUnicorn`, `RecurrenceMicrostatesAnalysis` and `RecurrenceAnalysis.jl`

3.2 Performance

The library was developed in `C++` to ensure high computational performance. We validate it by comparison with the well-known libraries `PyUnicorn` [27] and `RecurrenceAnalysis.jl` [29], as well as with `RecurrenceMicrostatesAnalysis.jl` [17], which implements the same methodology in Julia.

To analyze the mean computation time, we used the `timeit` function in Python and the `BenchmarkTools.jl` [28] library in Julia to benchmark the libraries on the same dataset of a uniform distribution. The results are shown in Figure 1, which highlights the Python library's strong performance, exhibiting the shortest mean computation time among the tests. The observed scaling is close to $O(n)$, although the algorithm is inherently $O(n^2)$, as expected for an implementation based on recurrence plots, which requires computing $K \times K$ recurrences.

It is also important to evaluate the library's efficiency in estimating RQA quantifiers. To do so, we compare the values estimated using the Python library with those calculated using libraries that provide well-established implementations of the standard RQA computation approach—such as `PyUnicorn` [27] and `RecurrenceAnalysis.jl` [29], as well as the library `RecurrenceMicrostatesAnalysis.jl` [17](abbreviated as `RMA.jl`), which uses a similar RMA-based approach to estimate DET and LAM. This comparison is carried out using the relative error between the RQA measures obtained from the reference libraries and those estimated by the Python library.

Figure 2 shows the relative error in estimating DET, comparing results from the Python library with those from the other libraries referenced in the figure legend. Panels (a)–(c) demonstrate excellent performance, with errors below 0.5% for $K \approx 917$ in the uniform distribution, $K \approx 546$ for the logistic map, and $K \approx 1364$ for the generalized Bernoulli-shift map. Panels (d) and (e) show similarly good results when comparing the Python library with `RecurrenceAnalysis.jl` and `RecurrenceMicrostatesAnalysis.jl`. However, a considerable error is observed when comparing with `PyUnicorn`. This discrepancy is known to occur when passing a multivariate time series (with $d > 1$) to `PyUnicorn` and only appears in such comparative scenarios.

Figure 3 is analogous to Fig. 2, but for the estimation of LAM. Panels (a)–(c) also show excellent performance, with errors below 0.5% for $K \approx 843$ in the uniform distribution and the generalized Bernoulli-shift map, and $K \approx 769$ for the logistic map. Similarly, panels (d) and (e) show good agreement when comparing with `RecurrenceAnalysis.jl` and `RecurrenceMicrostatesAnalysis.jl`, but a considerable error is observed when comparing with `PyUnicorn`, due to the same issue with multivariate time series ($d > 1$).

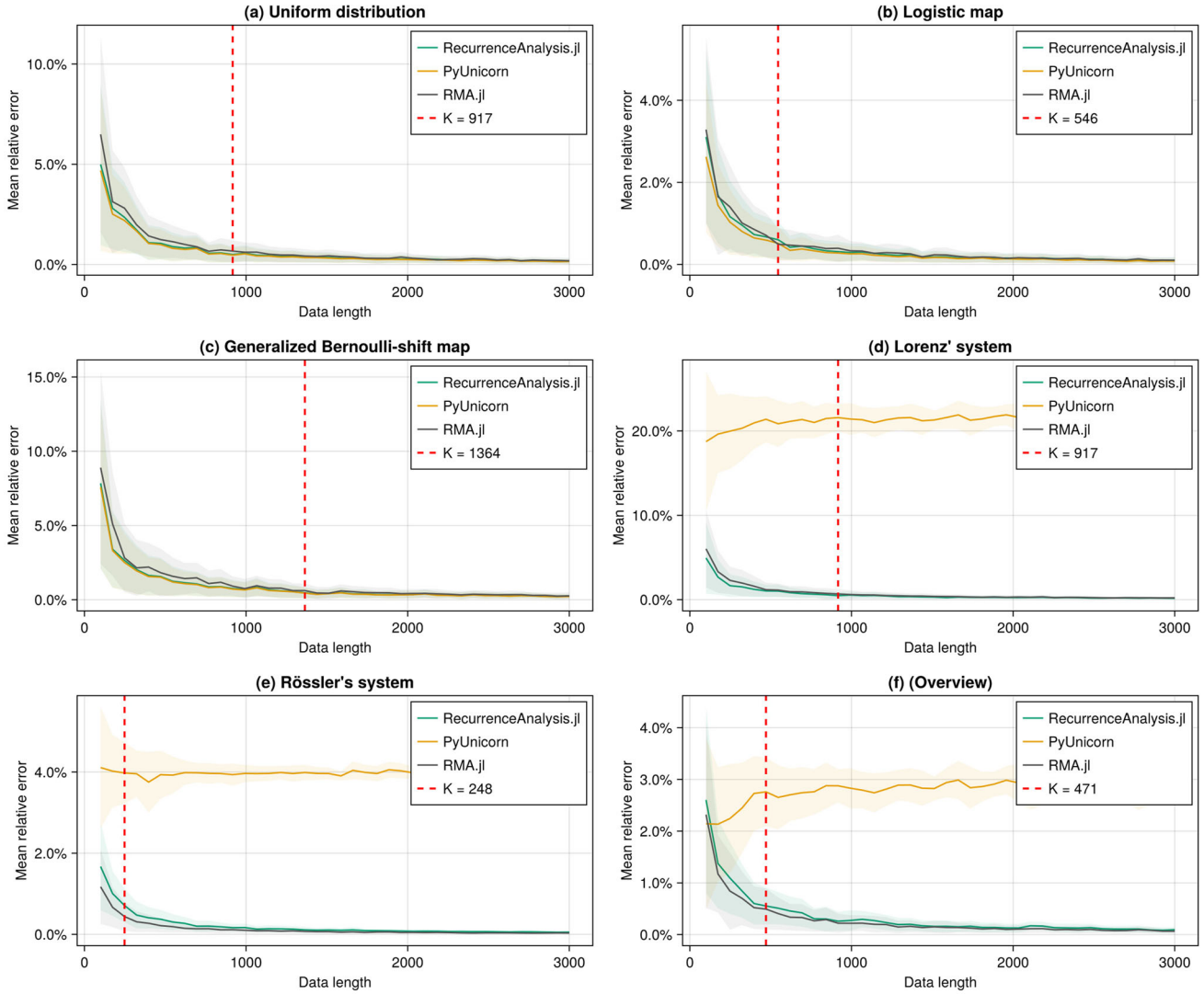


Fig. 2 Mean relative error of DET estimation using the Python library, compared with other libraries that compute DET. Each curve represents the error for a given comparison between the Python library and the library indicated in the legend. The vertical red dashed line indicates the data length at which the first error falls below 0.5%. Panels **(a)**–**(e)** show the error for each specific system, while panel **(f)** provides an overall summary across all systems

4 Conclusion

In this work, we presented the library `Recurrence Microstates` for Python. This library had been developed in `C++` to grant a good computational performance, with a integration with Python done by the library `pybind11` [18], and shows to be a efficient library for computing recurrence motifs probability distribution and estimating quantifiers.

The library provides a robust approach for analyzing large data series using RMA, demonstrating superior performance compared to other available libraries. In addition, it offers a user-friendly interface for configuring motif shapes and sampling modes. Tests conducted on three discrete and two continuous paradigmatic dynamical systems indicate that the estimation of RQA quantifiers produces very small errors relative to values obtained with alternative methods.

Consequently, `Recurrence Microstates` for Python demonstrates itself to be an optimal approach for analyzing data with RMA, combining improved computational efficiency with alignment to green computing principles.

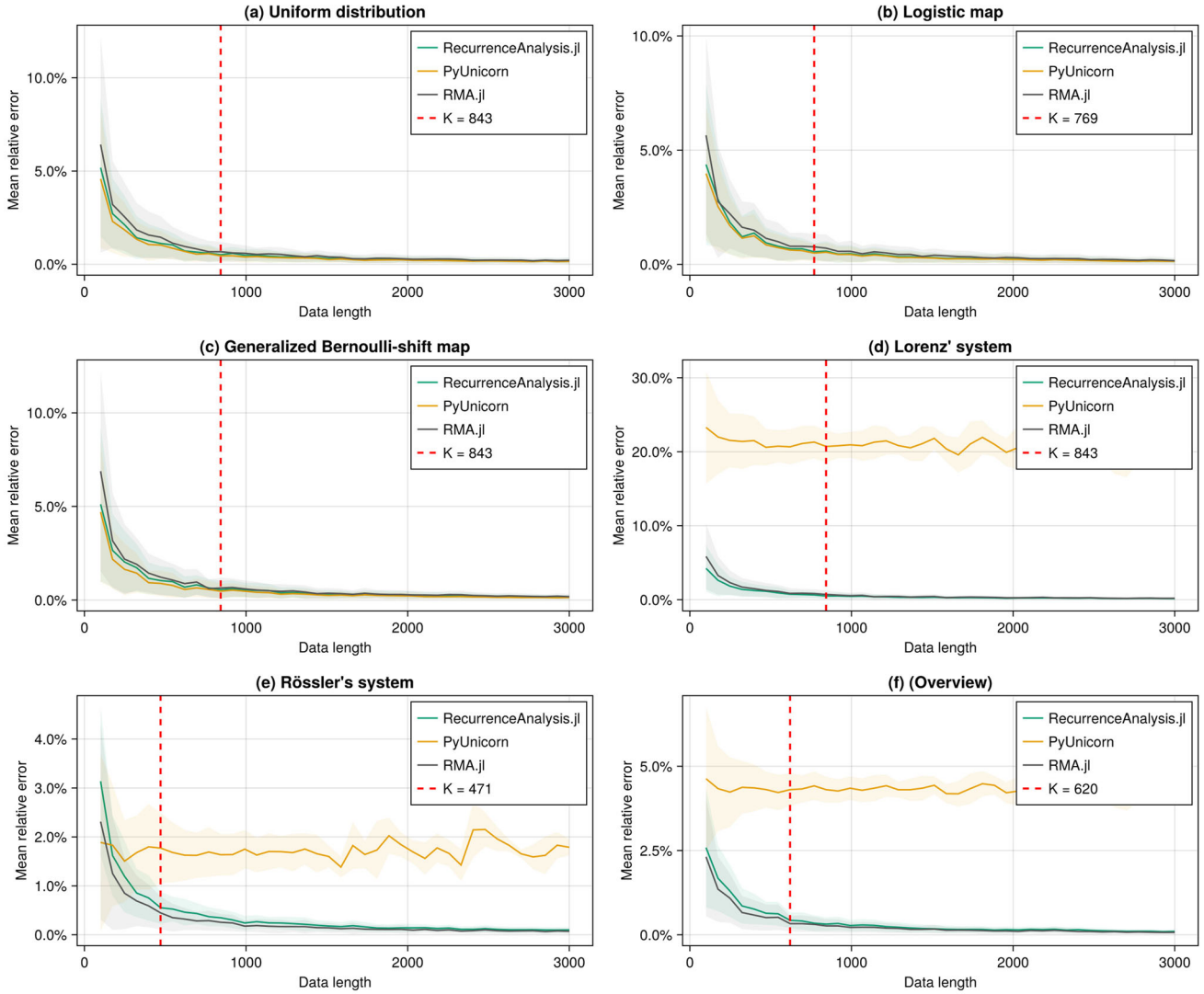


Fig. 3 Mean relative error of LAM estimation using the Python library, compared with other libraries that compute LAM. Each curve represents the error for a given comparison between the Python library and the library indicated in the legend. The vertical red dashed line indicates the data length at which the first error falls below 0.5%. Panels **a**–**e** show the error for each specific system, while panel **f** provides an overall summary across all systems

Acknowledgements This work is supported by the following Brazilian research agency: Conselho Nacional de Desenvolvimento Científico e Tecnológico (CNPq), Grants Nos. 308441/2021-4, 407072/2022-5, 305189/2022-0, 408254/2022-0, 300064/2023-3 and 443575/2024-0. This research is supported by INCT-NeuroComp (CNPq Grant 408389/2024-9). Coordenação de Aperfeiçoamento de Pessoal de Nível Superior (CAPES), Grants Nos. 88881.895032/2023-01.

Author contribution statement

G. V. Ferreira: conceptualization, methodology, software, validation, formal analysis, investigation, data curation, writing—original draft, visualization. F. E. L. da Cruz: conceptualization, methodology, software, validation, formal analysis, investigation, data curation, writing—original draft, visualization. G. Marghoti: conceptualization, methodology, software, validation, formal analysis, investigation, data curation, writing—original draft, visualization. T. L. Prado: conceptualization, methodology, resources, writing—review and editing, visualization, supervision, project administration, funding acquisition. S. R. Lopes: conceptualization, methodology, resources, writing—review and editing, visualization, supervision, project administration, funding acquisition. N. Marwan: conceptualization, resources, writing—review and editing, visualization, supervision, project administration, funding acquisition. J. Kurths: conceptualization, resources, writing—review and editing, visualization, supervision, project administration, funding acquisition.

Funding The Article Processing Charge (APC) for the publication of this research was funded by the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES) (ROR identifier: 00x0ma614).

Data Availability The data that support the findings of this study are available within the article and the corresponding webpages.

Declarations

Conflict of 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.

Open Access This article is licensed under a Creative Commons Attribution 4.0 International License, which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence, and indicate if changes were made. The images or other third party material in this article are included in the article's Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the article's Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder. To view a copy of this licence, visit http://creativecommons.org/licenses/by/4.0/.

References

1. F.E.L. da Cruz, T.L. Prado, S.R. Lopes, N. Marwan, J. Kurths, Phys. Rev. E **111**, 4 (2025). https://doi.org/10.1103/physreve.111.044212
2. G.S. Spezzatto, J.V.V. Flauzino, G. Corso, B.R.R. Boaretto, E.E.N. Macau, T.L. Prado, S.R. Lopes, Chaos **34**(7), 073140 (2024). https://doi.org/10.1063/5.0203801
3. E. Garcia-Ceja, Z. Uddin, J. Torresen, Procedia Comput. Sci. **130**, 157–163 (2018). https://doi.org/10.1016/j.procs.2018.04.025
4. B.M. Mathunjwa, Y.-T. Lin, C.-H. Lin, M.F. Abbod, J.-S. Shieh, Biomed. Signal Process. Control **64**, 102262 (2021). https://doi.org/10.1016/j.bspc.2020.102262
5. L. Lober, M.S. Palmero, F.A. Rodrigues, Phys. Rev. E **112**, 014210 (2025). https://doi.org/10.1103/wz7j-lzvs
6. Y. Hirata, Commun. Nonlinear Sci. Numer. Simul. **94**, 105552 (2021). https://doi.org/10.1016/j.cnsns.2020.105552
7. J.V.V. Flauzino, T.L. Prado, N. Marwan, J. Kurths, S.R. Lopes, Phys. Rev. Lett. **135**, 097401 (2025). https://doi.org/10.1103/1y98-x33s
8. G. Marghoti, T.L. Prado, S.R. Lopes, Y. Hirata, Phys. Rev. E **110**, 2 (2024). https://doi.org/10.1103/PhysRevE.110.024203
9. J.-P. Eckmann, S.O. Kamphorst, D. Ruelle, Europhys. Lett. **4**, 973–977 (1987). https://doi.org/10.1209/0295-5075/4/9/004
10. N. Marwan, M.C. Romano, M. Thiel, J. Kurths, Phys. Rep. **438**(5), 237–329 (2007). https://doi.org/10.1016/j.physrep.2006.11.001
11. T. Westerhold, N. Marwan, A.J. Drury, D. Liebrand, C. Agnini, E. Anagnostou, J.S.K. Barnet, S.M. Bohaty, D. De Vleeschouwer, F. Florindo et al., Science **369**(6509), 1383–1387 (2020). https://doi.org/10.1126/science.aba6853
12. H. González, O. Infante, H. Pérez-Grovas, M.V. Jose, C. Lerma, Med. Eng. Phys. **35**(2), 178–187 (2013). https://doi.org/10.1016/j.medengphy.2012.04.013
13. I.B.S. Ferré, G. Corso, G.Z.S. Lima, S.R. Lopes, M.A. Leocadio-Miguel, L.G.S. França, T.L. Prado, J.F. Araújo, PLoS ONE **19**(10), e0298703 (2024). https://doi.org/10.1371/journal.pone.0298703
14. M.S. Palmero, I.L. Caldas, I.M. Sokolov, Chaos **32**, 11 (2022). https://doi.org/10.1063/5.0102424
15. T.L. Prado, P.P. Galuzio, S.R. Lopes, R.L. Viana, Chaos **24**, 1 (2014). https://doi.org/10.1063/1.4861895
16. G. Corso, T.L. Prado, G.Z.S. Lima, J. Kurths, S.R. Lopes, Chaos **28**, 8 (2018). https://doi.org/10.1063/1.5042026
17. G.V. Ferreira, F.E.L. da Cruz, G. Marghoti, T.L. Prado, S.R. Lopes, N. Marwan, J. Kurths, Chaos [accepted, not published yet] (2025)
18. Pybind11, computer software available at GitHub. https://github.com/pybind/pybind11
19. B. Bruers, M. Cruces, M. Demleitner, G. Duckeck, M. Düren, N. Eich, T. Enßlin, J. Erdmann, M. Erdmann, P. Fackeldey, C. Felder, B. Fischer, S. Fröse, S. Funk, M. Gasthuber, A. Grimshaw, D. Hadasch, M. Hannemann, A. Kappes, R. Kleinemühl, O.M. Kozlov, T. Kuhr, M. Lupberger, S. Neuhaus, P. Niknejadi, J. Reindl, D. Schindler, A. Schneidewind, F. Schreiber, M. Schumacher, K. Schwarz, A. Streit, R.F. von Cube, R. Walker, C. Walther, S. Wozniewski, K. Zhou, Eur. Phys. J. Spec. Top. (2024). https://doi.org/10.1140/epjs/s11734-024-01436-4
20. C.R. Harris, K.J. Millman, S.J. van der Walt, R. Gommers, P. Virtanen, D. Cournapeau, E. Wieser, J. Taylor, S. Berg, N.J. Smith, R. Kern, M. Picus, S. Hoyer, M.H. van Kerkwijk, M. Brett, A. Haldane, J.F. del Río, M. Wiebe, P. Peterson, P. Gérard-Marchant, K. Sheppard, T. Reddy, W. Weckesser, H. Abbasi, C. Gohlke, T.E. Oliphant, Nature **585**(7825), 357–362 (2020). https://doi.org/10.1038/s41586-020-2649-2
21. R.M. May, Nature **261**, 459–467 (1976). https://doi.org/10.1038/261459a0

22. K.T. Alligood, T.D. Sauer, J.A. Yorke, *Chaos: An Introduction to Dynamical Systems* (Springer, New York, 1996). https://doi.org/10.1007/0-387-22492-0_3
23. E.N. Lorenz, J. Atmos. Sci. **20**(2), 130–141 (1963). https://doi.org/10.1175/1520-0469(1963)020<0130:DNF>2.0.CO;2
24. O.E. Rössler, Phys. Lett. A **57**(5), 397–398 (1976). https://doi.org/10.1016/0375-9601(76)90101-8
25. T.L. Prado, G. Corso, G.Z.S. Lima, R.C. Budzinski, B.R.R. Boaretto, F.A.S. Ferrari, E.E.N. Macau, S.R. Lopes, Chaos **30**(4), 043123 (2020). https://doi.org/10.1063/1.5125921
26. T.L. Prado, V.S. Machado, G. Corso, G.Z.S. Lima, S.R. Lopes, Nonlinear Dyn. **112**, 1141–1152 (2024). https://doi.org/10.1007/s11071-023-09063-9
27. J.F. Donges, J. Heitzig, B. Beronov, M. Wiedermann, J. Runge, Q.Y. Feng, L. Tupikina, V. Stolbova, R.V. Donner, N. Marwan, H.A. Dijkstra, J. Kurths, Chaos **25**(11), 113101 (2015). https://doi.org/10.1063/1.4934554
28. J. Chen, J. Revels, arXiv:1608.04295 (2016). https://doi.org/10.48550/arXiv.1608.04295
29. G. Datseris, J. Open Source Softw. **3**(23), 598 (2018). https://doi.org/10.21105/joss.00598