Eur. Phys. J. Spec. Top.
https://doi.org/10.1140/epjs/s11734-026-02338-3

THE EUROPEAN PHYSICAL JOURNAL SPECIAL TOPICS

Regular Article

AccRQA library: accelerating recurrence quantification analysis

Karel Adámek[1,a], Jan Novotný[1], Norbert Marwan[2], and Radim Pánis[1]

[1] Institute of Physics in Opava, Silesian University in Opava, Bezručovo nám. 13, 74601 Opava, Czech Republic
[2] Postdam Institute for Climate Impact Research, Telegrafenberg A 31, 14473 Postdam, Germany

Received 26 January 2026 / Accepted 20 April 2026
© The Author(s) 2026

Abstract Recurrence quantification analysis is a powerful tool for identifying and quantifying patterns in dynamical systems, widely used across many disciplines. The rapid growth of data in these fields demands more efficient techniques for analysis. We present AccRQA, a high-performance library available in Python, R, and C/C++, which utilizes novel, scalable, and portable parallel algorithms. AccRQA is parallelized using OpenMP and can leverage NVIDIA GPUs when available, providing portability across computational platforms (CPUs, GPUs) and user environments (PC, HPC), thus offering flexibility between exploration and systematic mapping of a vast parameter space. AccRQA supports long time series and efficient computations for different embedding dimensions m and delay τ with minimal memory requirements. We also present performance benchmarks demonstrating an average 10 × speed-up and at least a 6 × speed-up compared to state-of-the-art RQA packages on both CPUs and GPUs.

1 Introduction

Zbilut and Webber [1] introduced recurrence quantification analysis (RQA) as a tool for studying complex systems and showed its capability to uncover nonlinear dynamics. RQA is now a widely recognized method for examining dynamics in time series produced by underlying systems by quantifying recurring patterns and structures [2, 3]. Its versatility has led to applications in many scientific and engineering fields.

In physics, Guimarães-Filho et al. [4] used RQA to study turbulent fluctuations in the plasma edge of a tokamak and highlighted how RQA can uncover dynamics missed by traditional methods. In astrophysics, RQA has also been applied to analyze time series produced by blazars [5] and to distinguish between known blazar groups based on timing signals from gamma-ray observations [6]. In the field of theoretical physics, RQA has been used to investigate the chaotic motion of charged particles in Keplerian accretion disks [7].

RQA is utilized in climate science as well [8, 9], for example, Westerhold et al. [8] utilized RQA to identify and classify four distinct climate states in Earth's Cenozoic climate record and enabled the authors to quantify climate predictability. RQA has also been adopted in cognitive psychology, where RQA enabled researchers to systematically identify and classify different decision-making strategies [10] or text reading fluency [11]. Economics, too, has benefited from RQA's insights. Orlando and Zimatore [12] demonstrated that RQA is suitable for studying business cycles in macroeconomics and could be used for early detection of recessions. In engineering, RQA has been applied to fault diagnosis in mechanical systems [13–16].

In all these scientific fields, the volume and velocity of data are increasing with more advanced sensors and data-gathering techniques [17–20]. This growth in data places increasing demands on RQA packages, which must not only process larger datasets but also do so faster than before to keep up with incoming data. Moreover, RQA is memory-intensive, as its memory requirements scale with the square of the data size unless optimized. Furthermore, energy efficiency and conservation are essential aspects of scientific computing and package development [21]. Trefethen and Thiyagalingam [22] and Adámek et al. [23] have investigated possible energy savings on CPUs and GPUs, concluding that, without dynamic voltage and frequency scaling, the most energy-efficient algorithm is the fastest and most optimized. Therefore, a high-performance, optimized, and memory-efficient RQA package is necessary to address increasing data volumes and improve energy efficiency.

[a] e-mail: karel.adamek@slu.cz (corresponding author)

Published online: 13 May 2026

The current landscape of RQA packages is diverse, with varying performance, support for programming languages, computational platforms (CPU, GPU), and different memory limitations, thus potentially excluding measures unless, for example, downsampling is used. Among many available packages, the ones we refer to in this work are `CRQA` [24–26], which is a gold standard package for R but consumes a lot of memory, `pyunicorn` [27] is a Python package which again has a large memory footprint but is fast, especially for small datasets, RecurrenceAnalysis.jl [28] is a RQA package for Julia, and lastly `PyRQA` [29], which is a parallel Python package accelerated by OpenCL. There are also packages for MATLAB [30]. A review of these and recent trends in the development of recurrence analysis can be found in Marwan and Kraemer [3].

In this work, we present AccRQA, a library for calculating RQA measures of time series, designed to process ever-increasing volumes of data generated in these disciplines with a minimal memory footprint. The AccRQA library is written in C/C++ and parallelized using OpenMP and CUDA, and is accessible in multiple languages (C/C++, Python, R) via bindings to a C/C++ backend. Acceleration with NVIDIA GPUs is optional, and NVIDIA GPUs are not required for any features of the AccRQA library.

We also describe a set of novel parallel algorithms for calculating RQA measures. Our algorithms are based on stencil operations, and we demonstrate their implementation on NVIDIA GPUs using CUDA. The AccRQA library is available on GitHub.[1]

The main advantages of the AccRQA library are:

- **Speed**—AccRQA is faster than the current state-of-the-art (PyRQA, pyunicorn)
- **Memory efficient**—memory footprint is at most 4× size of the time series
- **Processing at scale**—ability to process long time series with more than 1 million samples
- **Portability**—available on Windows, Linux, macOS
- **Multiple language support**—C, Python, R
- **Hardware independent**—supporting CPUs and optional acceleration using NVIDIA GPUs

The AccRQA library was thoroughly tested against the CRQA package (R) [25, 31] and was already used in a modification of RQA called averaged recurrence quantification analysis (ARQA) that eliminates the need for selecting a fixed recurrence threshold by calculating RQA measures across a range of thresholds [32].

This work is divided as follows: RQA measures and parameters are described in Sect. 2, followed by Sect. 3, which describes our algorithms and their implementation in the AccRQA library. The performance of the AccRQA library and comparison to other packages is shown in Sect. 4. The design of the library, its functions, the data format of the results, and other implementation details are summarized in Sect. 5. The paper is summarized in Sect. 6.

2 RQA measures

To describe the recurrence plot (RP), RQA uses a set of quantities [9, 33] that were developed over time [1, 34, 35].

$$R(i,\, j) = \Theta(\epsilon - ||\vec{x}(i) - \vec{x}(j)||)\,, \tag{1}$$

where Θ is a Heaviside function, ϵ is threshold, and $x(i)$ is a sample from time series under investigation. These measures are recurrence rate RR, determinism DET, laminarity LAM, averaged diagonal line length L, trapping time TT, longest diagonal line $L_{\max}$, longest vertical line $V_{\max}$, and entropy ENTR. These are defined as

$$\mathrm{RR} = \frac{1}{N^2}\sum_{i,\,j=1}^{N} R(i,\, j) = \sum_{l=1}^{N} lP(l)\,, \tag{2}$$

$$\mathrm{DET} = \frac{\sum_{l=l_{\min}}^{N} lP(l)}{\sum_{l=1}^{N} lP(l)}\,, \tag{3}$$

$$\mathrm{LAM} = \frac{\sum_{v=v_{\min}}^{N} vP(v)}{\sum_{v=1}^{N} vP(v)}\,, \tag{4}$$

$$L = \frac{\sum_{l=l_{\min}}^{N} lP(l)}{\sum_{l=l_{\min}}^{N} P(l)}\,, \tag{5}$$

[1] GitHub: https://github.com/KAdamek/AccRQA.

$$\mathrm{TT} = \frac{\sum_{v=v_{\min}}^{N} vP(v)}{\sum_{v=v_{\min}}^{N} P(v)}\,, \tag{6}$$

$$L_{\max} = \max(l)\,, \tag{7}$$

$$V_{\max} = \max(v)\,, \tag{8}$$

$$\mathrm{ENTR} = -\sum_{l=l_{\min}}^{N} p(l)\ln p(l)\,, \tag{9}$$

where N is the length of the time series that is being analyzed, $R(i, j)$ is the element of the recurrence plot at i and j, l, v is the length of diagonal and vertical line, respectively, $P()$ is a histogram of line lengths, $p(l) = P(l)/\sum_{l=l_{\min}}^{N} P(l)$ is a probability that diagonal line has exactly length l and, and lastly $l_{\min}$, $v_{\min}$ are the minimal lengths considered as line in DET, LAM measures.

Each of the RQA measures describes specific features of the recurrence plot and the underlying dynamics of the system:

- **Recurrence rate** (RR)**:** Measures how often the system's states recur by calculating the density of recurrence points in the plot, indicating how frequently similar states are revisited.
- **Determinism** (DET)**:** Quantifies the predictability of the system by measuring the percentage of recurrence points that form diagonal lines. Diagonal lines suggest that the system follows a predictable trajectory over time.
- **Laminarity** (LAM)**:** Reflects the proportion of recurrence points that form vertical lines, indicating periods when the system remains in a specific state or is trapped in an intermittent behavior.
- **Averaged diagonal line length** (L)**:** Represents the average length of diagonal lines in the plot, providing insight into how long the system exhibits predictable behavior before diverging.
- **Trapping time** (TT)**:** Measures the average length of vertical lines, showing how long the system stays in particular states, indicating stability or laminar phases.
- **Longest diagonal line** ($L_{\max}$)**:** The length of the longest diagonal line reveals the longest time over which the system behaves predictably.
- **Longest vertical line** ($V_{\max}$)**:** The length of the longest vertical line indicates the longest duration the system spends in a particular state without changing.
- **Entropy** (ENTR)**:** Captures the complexity of the system by calculating the distribution of diagonal line lengths, with higher Entropy indicating greater diversity in the behavior of the system.

Together, these measures offer a comprehensive view of the properties of the dynamics of the system, including its regularity, predictability, stability, and complexity.

2.1 Embedding and time delay

Nonlinear time-series analysis provides a fresh perspective on understanding physical systems. Unlike traditional stochastic approaches that mainly deal with randomness and probabilities, nonlinear methods emphasize connections to the actual physical properties of the system. These methods can detect underlying patterns in seemingly random one-dimensional data, pointing to deterministic chaos. Deterministic chaos describes a behavior that looks unpredictable at first glance but actually follows precise rules due to the system's inherent nonlinear characteristics. In such systems, the future behavior is entirely determined by the starting conditions, even though it might appear chaotic or noise-like.

A key part of nonlinear time-series analysis is reconstructing the state space. This technique allows us to explore the behavior of a complex system by reconstructing its dynamics from just a single observed time series. The concept of "reconstruction" focuses on maintaining the original system's structural features, which helps us in understanding the underlying patterns [36].

One popular method for reconstructing the state space is called delay coordinate embedding [37]. It transforms the original one-dimensional time series into a higher-dimensional space using time-delayed vectors. This is done by taking an m-dimensional vector formed from the original series values, spaced out by a delay τ, and can be written as:

$$\mathbf{X}(t) = [\mathbf{y}(t), \mathbf{y}(t-\tau), \mathbf{y}(t-2\tau), \ldots, \mathbf{y}(t-(m-1)\tau)].$$

The choice of τ is important in practice, especially when dealing with real-world data that is often noisy and limited in quantity. If τ is too small, the values in the reconstructed vectors will be closely related, causing the points to cluster near the main diagonal of the reconstructed space and potentially obscuring meaningful patterns. Conversely, if τ is too large, the points may scatter too much, and the reconstructed state space might not accurately reflect the original system's dynamics.

The choice of the embedding dimension m is equally important. If m is too small, the trajectories in the reconstructed space may overlap, as the projection fails to fully unfold the system's dynamics. Conversely, if m is too large, redundant coordinates make the reconstruction sparse and less reliable due to limited data. Hence, the embedding dimension should be selected to capture the system's essential dynamics while avoiding redundant dimensionality.

Since real-world data often contains noise and other imperfections, such as missing data, the algorithms used to estimate m and τ may not always yield accurate values. For this reason, it can be valuable to calculate results across different choices of τ and m and observe how these variations affect the computed RQA measures. By examining the stability of RQA measures across a range of embedding parameters, one can gain insights into the robustness of the reconstructed dynamics and better understand the underlying behavior of the system. This approach helps to identify parameter settings that consistently capture meaningful patterns, even in the presence of noise.

Our library supports customizing both the time delay τ and the embedding m via the interface, as well as batch processing across multiple values of τ and m.

3 Algorithms

To calculate RQA measures, we first need to evaluate the recurrence plot (RP) according to Eq. 1 before calculating any of the RQA measures. However, storing RP is memory-intensive, especially for longer series, as it scales with the square of its length. Furthermore, it introduces a memory bandwidth bottleneck, slowing the calculation of RQA measures. Therefore, both of our algorithms for calculating RQA measures evaluate individual recurrent plot elements (RPEs) $R(i, j)$ on the fly, saving memory and avoiding a memory bandwidth bottleneck. RP is, thus, never evaluated and stored as a whole.

From the perspective of algorithm design and implementation, we can divide the RQA measures into those that can be calculated using local operations, which are operations that act on a single recurrence plot element (RPE) and its immediate surroundings, and those that require global operations, such as determining line lengths. Among the local measures, we can include RR, DET, LAM, L, and TT. Measures that require global operations are $L_{\max}$, $V_{\max}$, and ENTR. Since global operations require more communication and synchronization among individual threads, they are usually slower than local operations.

Therefore, in the AccRQA library, we have implemented two ways of calculating RQA measures. The first implementation, based on local operations, calculates RR, DET, LAM, L, and TT, while the second, based on enumerating the histogram of lengths $P(l)$, calculates all RQA measures.

The following algorithms were developed using CUDA and OpenMP frameworks.

3.1 Global approach

We see that the histogram of line lengths $P(l)$ in the diagonal or vertical direction is featured in all RQA measures listed in Sect. 2. Thus, an algorithm for quick, scalable calculation of the length histogram is necessary.

A high-performance, parallel and scalable algorithm must identify a line, where the line starts and ends, and determine its length in parallel. Therefore, we have developed an algorithm that has two stages. In the first stage, two filters are applied, resulting in two arrays identifying the starting and ending elements of all lines. In the second stage, these two arrays are rearranged so that the same array index contains the start and end positions of the same line. These positions are then subtracted, resulting in the length of each line (vertical or diagonal) contained in the recurrence plot. Both stages are parallel in nature as is necessary for GPUs.

To identify the start and the end of each line, we have used two simple filters. This is shown in Fig. 1. These filters can be applied to each RPE independently and in parallel. The beginning of the line is characterized by one empty RPE (having value 0) followed by one occupied RPE (value 1). Similarly, the end is characterized by an occupied RPE followed by an empty RPE. If the criteria for any of the two filters are fulfilled, the array index of the RPE is stored in an array containing start or end indices, respectively. To preserve the order of identified start/end indices, the start/end indices are stored at the same relative position as the index of the RPE the thread is examining.

We handle the border conditions by inserting a single zero RPE at the beginning and end of each RP line. This ensures that the beginning of a line starting with the first RPE is detected, and that the end of a line ending with the last RPE is also detected. The two extra elements are shown in Fig. 1 using dotted lines. These zero elements

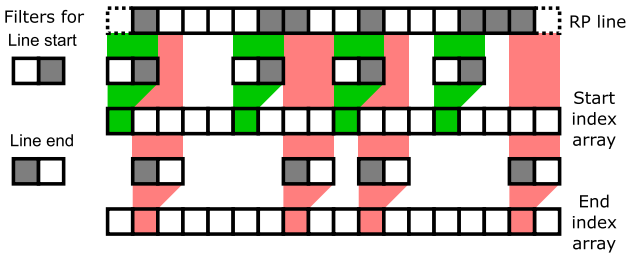


Fig. 1 First stage: filters identify the beginning/end of each line and store indexes into appropriate arrays

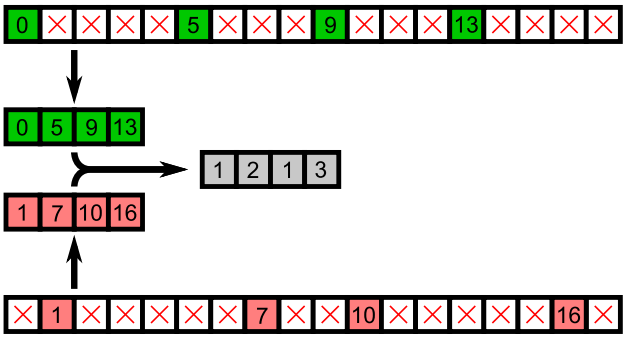


Fig. 2 Second stage: two arrays containing beginning/end indices are first compacted by removing empty elements. Then the two related indices for each line (start index and end index) are subtracted, giving the line length

do not prevent us in principle from performing border effect correction [38], as the indices of the start and end of the line are retained.

The calculation of line lengths by element-wise subtraction of a start index from an end index requires ordered and compact arrays. The arrays are ordered by virtue of how they are created. However, they will contain empty elements, and the start and end indices will not be aligned at the same position to be subtracted. To correct this, we perform a compact operation on both arrays as shown in Fig. 2.

Algorithm 1 Pseudo-code for calculating global RQA metrics

Require: $RP, m, \tau, l_{\min}$
$P(\cdot) \leftarrow 0$ ▷ Histogram initialization
$B[\cdot] \leftarrow 0$ ▷ Beginning array
$E[\cdot] \leftarrow 0$ ▷ Ending array
for all $line \in RP$ **do**
  $B[\cdot] \leftarrow \text{DetectLineStart}(line)$ ▷ Populate start/end arrays based on line
  $E[\cdot] \leftarrow \text{DetectLineEnd}(line)$
  $B[\cdot] \leftarrow \text{Compact}(B[\cdot])$ ▷ Compactification of the start/end arrays
  $E[\cdot] \leftarrow \text{Compact}(E[\cdot])$
  $L \leftarrow E[\cdot] - B[\cdot]$ ▷ Calculate line length L
  $P(L) \leftarrow P(L) + 1$ ▷ Add line of length L to the histogram
end for

The compact operation eliminates empty (neutral) elements and aligns occupied elements at the beginning of the array while preserving the order of elements. The parallel implementation of compact operation [39] on GPUs uses a prefixed sum or a scan algorithm [40]. First, a new array is created containing a predicate, that is, zero for empty elements and one for occupied elements. Then a prefix sum is calculated on the predicate that results in an address where the elements that fulfill the predicate (in this case, occupied RPE) should be moved to. After both start and end arrays are compacted, we can perform element-wise subtraction and obtain line lengths. Pseudo-code is shown in Algorithm 1.

When we have the lengths of all lines, we add them to a histogram of all lengths $P(l)$.

3.2 Local approach

Local algorithm is based on the assumption that we do not need to calculate length of each line to produce sums $S_{\mathrm{t}} = \sum_{l=1}^{N} lP(l)$, $S_{\mathrm{n}} = \sum_{l=l_{\min}}^{N} P(l)$, and $S_{\mathrm{lmin}} = \sum_{l=l_{\min}}^{N} lP(l)$ as long as we can distinguish if RPE belongs to

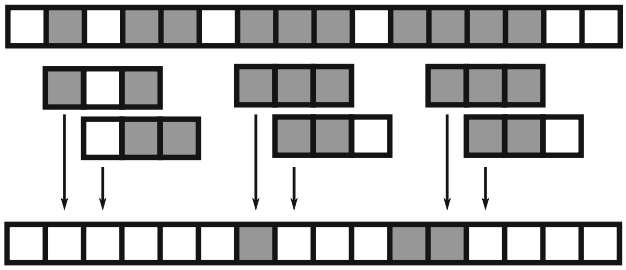


Fig. 3 Boxcar stage: boxcar filter eliminates lines shorter then $l_{\min}$ and shortens other lines by $l_{\min} - 1$. In this example, $l_{\min} = 3$

lines longer or equal to $l_{\min}$. This can be achieved by first eliminating all lines shorter than $l_{\min}$ and then counting all occupied RPEs.

To eliminate lines shorter than $l_{\min}$, we invoke a boxcar filter of size $l_{\min}$ and apply it to the recurrent plot. Then RPE is changed to empty if the boxcar value is smaller than $l_{\min}$. The situation is shown in Fig. 3. This also has an unfortunate effect of cutting the end of a line by a $l_{\min} - 1$ RPEs, which needs to be compensated for. Compensation is done by detecting the beginning of a line by the same filter as in the global case and multiplication of the beginning RPE by $l_{\min}$. This process will not overcompensate as lines shorter than $l_{\min}$ were already eliminated, and the same amount of RPEs were cut off from remaining lines.

Since the above-mentioned algorithm is spatially local, we divide the RP matrix into submatrices and process each submatrix in parallel. The pseudo-code is shown in Algorithm 2.

Algorithm 2 Pseudo-code for calculating local RQA measures

Require: $RP, m, \tau, l_{\min}$
$S_t \leftarrow 0$ ▷ Accumulator for $S_{\mathrm{t}} = \sum_{l=1}^{N} lP(l)$
$S_{lmin} \leftarrow 0$ ▷ Accumulator for $S_{\mathrm{lmin}} = \sum_{l=l_{\min}}^{N} lP(l)$
$S_n \leftarrow 0$ ▷ Accumulator for $S_{\mathrm{n}} = \sum_{l=l_{\min}}^{N} P(l)$
for all $r \in RP$ **do**
    $S_t \leftarrow S_t + r$
    $r \leftarrow \mathrm{BoxcarFilter}(l_{\min}, r)$ ▷ Eliminates lines shorter than $l_{\min}$
    if DetectLineStart(r) **then** ▷ We must handle start of the line differently
        $S_{lmin} \leftarrow S_{lmin} + r \cdot l_{\min}$ ▷ Boxcar filter correction
        $S_n \leftarrow S_n + 1$ ▷ Number of lines increased by 1
    end if
end for
$\mathrm{ReductionSum}(S_t, S_{lmin}, S_n)$ ▷ Gather contributions from other threads

4 Performance

We evaluated the performance of AccRQA and compared it with other publicly available libraries, specifically: `PyRQA` (8.1.0) [29], `pyunicorn` (0.8.2) [27] and `RecurrenceAnalysis.jl` (2.1.4) [28]. Each library was tested on the CPU and GPU, depending on the capabilities provided by the package.[2]

We divided our performance test into two separate parts. The first focuses on the overall performance, measuring execution time across a range of time-series lengths and comparing the libraries on both GPU and CPU architectures. The second part provides a more detailed analysis of AccRQA's performance, exploring how performance varies with the RQA parameters, including the embedding m, the time delay τ, and threshold ϵ. The range of parameters m and τ used in the benchmark is based on an RQA setup employed in various fields: Medicine [41–45]; Chemistry [46–48]; Psychology [25, 49]; Linguistics [50, 51]; Astronomy [52, 53]; Geology/Seismology [54, 55].

To evaluate both everyday and high-performance usage scenarios, we tested all libraries on two categories of hardware: server-grade CPUs and GPUs typically used in HPC environments and consumer-grade systems, such as standard laptops. The summary of the hardware and software used is presented in Table 1 for CPUs and in Table 2 for GPU accelerators.

[2] For other available libraries, see the above-mentioned references.

Table 1 Hardware and software specifications for the selected server and laptop CPUs used

	Server CPUs		Laptop CPUs	
Name	AMD 7601	AMD 7352	Intel i7-1065G7	AMD Ryzen 5800H
Architecture	EPYC 1st (Naples)	EPYC 2nd (Rome)	10th Intel i7 (Ice Lake)	AMD's Zen 3 (Cezanne)
Base core frequency (MHz)	2200	2300	1300	3200
Base memory frequency (MHz)	2667 (DDR4)	2667 (DDR4)	3733	3200
Cores	2x32	2x24	4	8
TDP (W)	180	155	15	45
GCC/MSVC version	11.4.0	8.5.0	13.3	13.3
Python version	3.10.12	3.9.7	3.12.3	3.12.3
OpenMP version	4.5	4.5	4.5	4.5

Please note the Server CPUs (e.g., AMD 7601, AMD 7352) are dual-socket configurations

Table 2 Hardware and software specifications for the selected NVIDIA GPUs range from consumer-grade (GTX 1070, GTX 1650) to data center/high-performance-class accelerators (V100, A100, H100) used in our performance test

	Server GPUs			Laptop GPUs	
Name	V100-SXM2-32GB	A100-PCIE-40GB	H100 NVL	GTX 3070	GTX 1650
Architecture	Tesla (GV100)	Ampere (GA100)	Hopper (GH100)	Ampere (GA104)	Turing (TU117)
Boost vore frequency (MHz)	1530	1410	1980	1730	1665
Base memory frequency (MHz)	877 (HBM2)	1215 (HBM2e)	1315 (HBM3)	2002 (GDDR5)	2001 (GDDR5)
SM vount	80	108	114	46	14
CUDA cores	5120	6912	14592	5888	896
Tensor vore	640	432	528	184	None
FP32 performance (TFLOPS)	14	19.5	51	20.3	3
TDP (W)	300	250	700	150	75
GCC/MSVC version	7.5.0	14.2.0	13.2.0	13.3	13.3
NVCC version	12.8	12.6	12.6	12.6	12.6
Python version	3.11.3	3.12.3	3.13.1	3.12.3	3.12.3
Driver version	570.124.06	560.35.03	560.35.03	560.81	538.92

4.1 Measurement methodology

In our benchmarks, we have attempted to replicate a typical workflow for processing data with RQA. We have generated a long time series, consisting of more than 1 million points, and then extracted time series of increasing length, up to 1 million points. The time series is generated from the Rössler system, with standard parameters ($a = 0.25$, $b = 0.25$, and $c = 4.0$) and sampling time $\Delta t = 0.01$. This setup replicates the dataset used in Ref. [3], allowing for continuity and comparability.

Each time series of length N from 200 up to 1 million, increasing in steps to provide equidistant points along a logarithmic axis, was processed once using preselected values of embedding dimension, time delay, and threshold ($m = 3$, $\tau = 6$, $\epsilon = 1.2$), and the execution time for each length was recorded. This process was repeated multiple times to ensure that the standard deviation of the measurements did not exceed approximately 5 %. The final execution time was then computed as the mean of the measured times for each sequence length. Alternating the

time-series length and its content, i.e., changing the sequence length between runs rather than repeating the same process N times before switching to a new length, helps ensure that no data remains cached in the CPU or GPU memory and provides more consistent performance measurements. This methodology was used for each tested library and computational platform.

4.2 Comparison to the state-of-the-art

We have compared the performance of the AccRQA library to the state-of-the-art package pyRQA, which utilizes OpenCL for parallelisation, pyunicorn, which is a C-compiled Python package, RecurrenceAnalysis.jl in Julia and CRQA in R. For the benchmarks, we have used server-grade NVIDIA GPUs and AMD CPUs commonly used for scientific calculations, as well as off-the-shelf laptop hardware. NVIDIA A100 and H100 GPUs represent server-grade NVIDIA GPUs, which are widely used. Similarly, the AMD 7601 and AMD 7352 are server-grade CPUs commonly found in HPC environments. For comparison with older architectures, we have also included the NVIDIA V100 GPU (release 2017). To demonstrate that GPU acceleration is a benefit even on off-the-shelf hardware, we have also included NVIDIA gaming cards, specifically an Ampere generation (2020) GTX 3070 and a Turing generation (2018) GTX 1650, along with laptop CPUs, namely AMD Ryzen 5800H and Intel i7-1065G7.

The comparison of execution times in log-scale on server-grade hardware is shown in Fig. 4. AccRQA performance on GPUs and CPUs is shown in separate plots, but the best-performing hardware from the other plot is shown as a

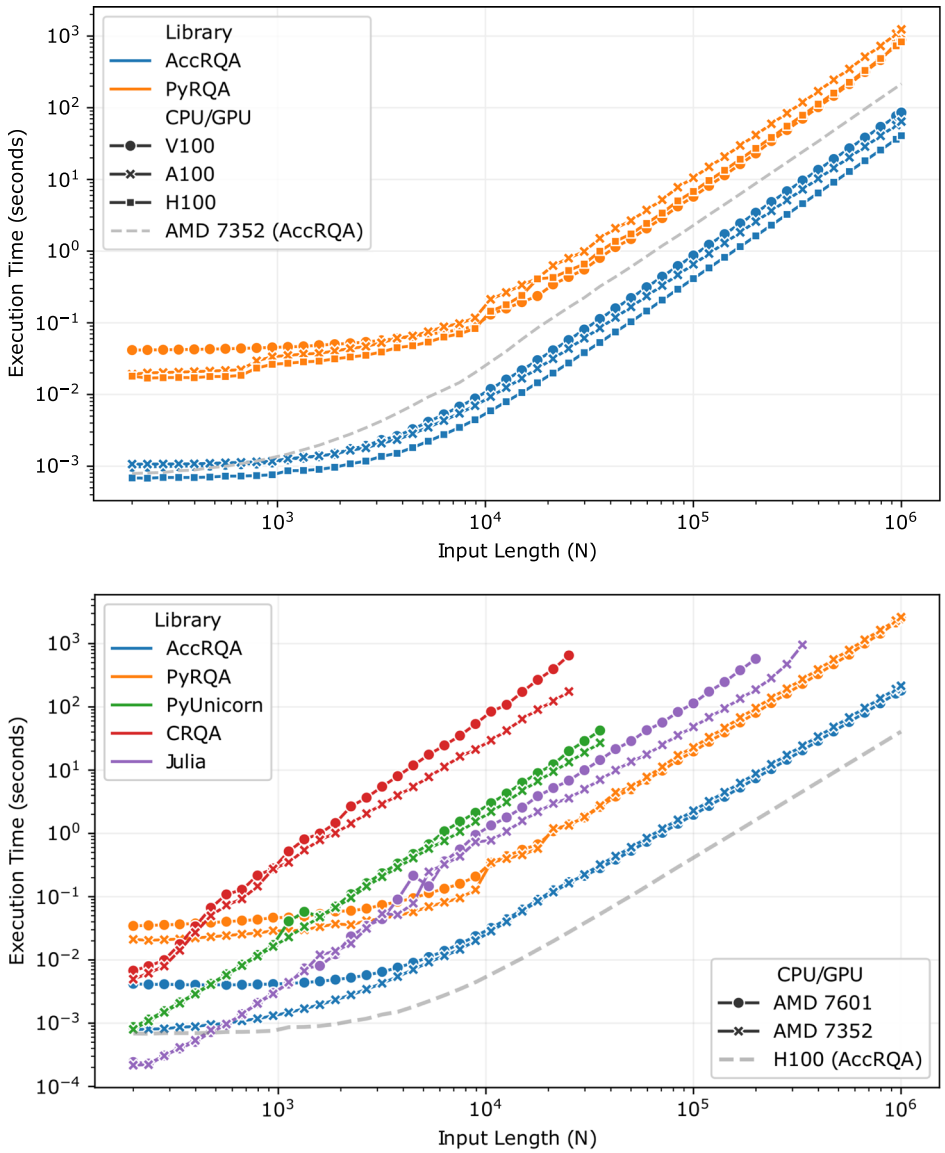


Fig. 4 Dependence of execution time on the input time series length for the RQA libraries AccRQA (blue), PyRQA (orange), pyunicorn (green), RecurrenceAnalysis.jl (purple), and CRQA (red). The upper panels display results obtained on server-class GPUs, while the lower panels correspond to server-class CPUs. For clarity, additional reference curves for AccRQA (in grey) are included to facilitate direct comparisons between CPU and GPU

grey dashed line for comparison. We can see that our AccRQA library on server GPUs achieves higher performance compared to pyRQA and provides a consistent 15 × speed-up on NVIDIA A100 and H100 GPUs across all time-series lengths. When we compare PyRQA on the NVIDIA H100 GPU to AccRQA on the highest-performing CPU (AMD 7352), AccRQA still achieves a 3 × speed-up. The exception is the NVIDIA V100 GPU, where AccRQA runs 6 × faster compared to PyRQA.

Moving to server-CPUs, we can see that AccRQA is faster than PyRQA and deliver at least 6x speed-up with an average speed-up of 10 × for both CPUs. The pyunicorn manages to be faster on AMD 7601 until a time series of 500 samples in length. The Julia package RecurrenceAnalysis.jl is faster than our AccRQA library for series length below 600 samples. This is due to the time costs associated with thread creation in OpenMP. In contrast, on AMD 7352, AccRQA is on par with pyunicorn even for short time series.

The performance of the off-the-shelf hardware is shown in Fig. 5. For comparison, the best-performing server-grade hardware for other libraries and AccRQA is shown for reference. We can see that the performance of AccRQA on off-the-shelf hardware is comparable to, or even higher than, PyRQA's performance when running on server-grade CPUs or GPUs. Similarly, for RecurrenceAnalysis.jl, which is faster for short series, but it is overtaken by AccRQA for series longer than 400 samples. Therefore, AccRQA can deliver on laptop hardware what state-of-the-art libraries could deliver in an HPC environment.

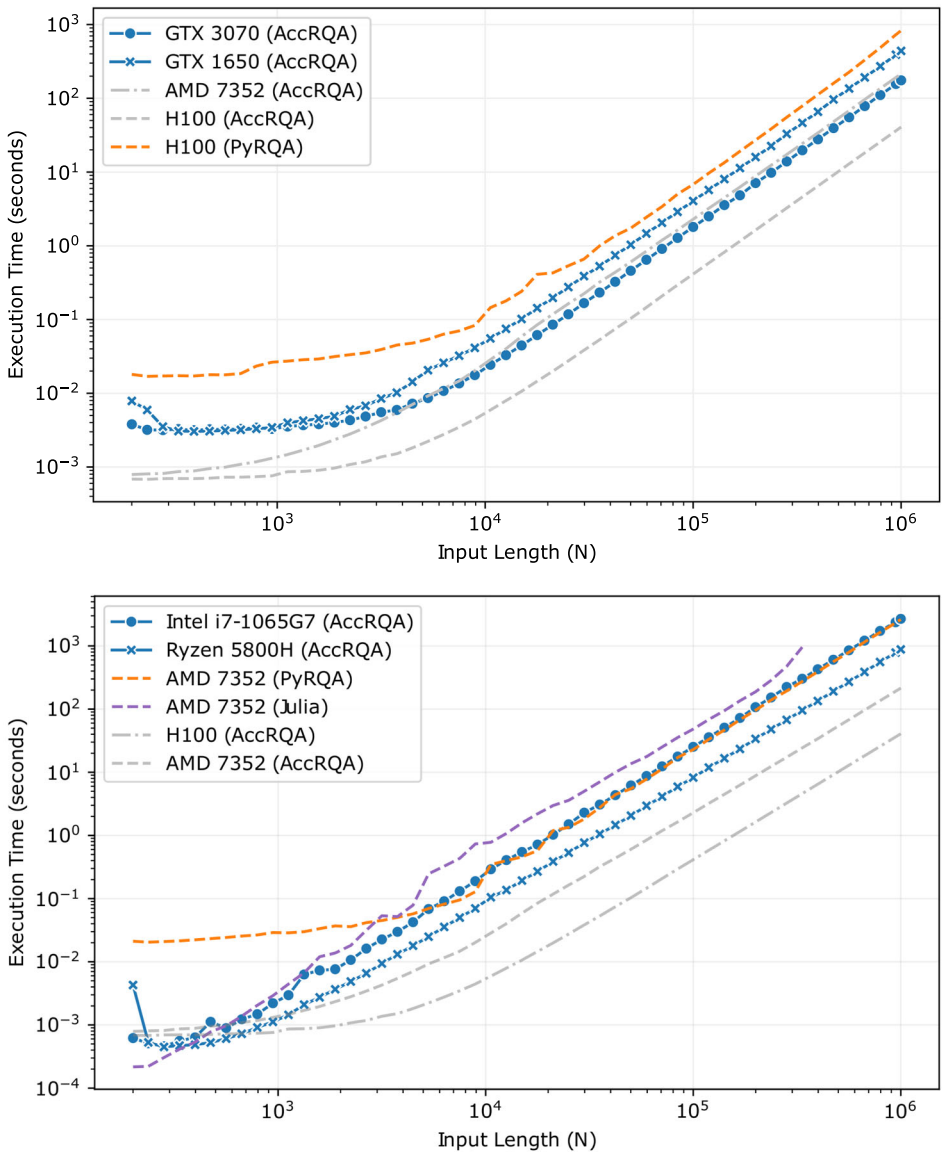


Fig. 5 Dependence of execution time on the input time series length for the RQA libraries AccRQA (blue) on the laptop-class GPU (top) and CPU (bottom). For clarity, additional reference curves for AccRQA (grey), PyRQA (orange), and RecurrenceAnalysis.jl (purple) are included to facilitate direct CPU/GPU comparisons

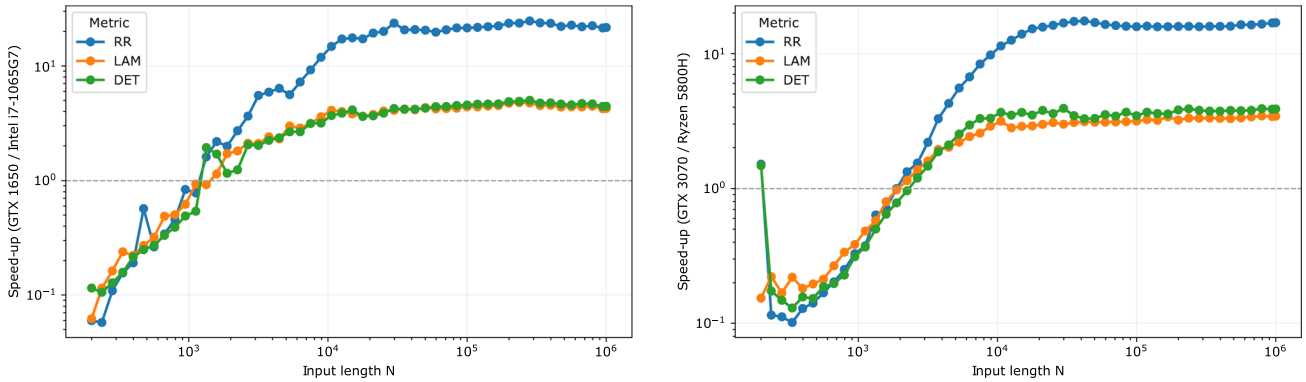


Fig. 6 Speed-up achieved by using a laptop GPU compared to a laptop CPU

To demonstrate that GPU acceleration is useful even on consumer-grade hardware, particularly on laptop GPUs, we have compared GPU and CPU performance on each laptop in Fig. 6. It shows that even on consumer-grade hardware, GPU acceleration achieves good speed-ups. The GPU speed-up is on par with the CPU at over 1000 samples. Calculating LAM and DET for lengths of over 5000 samples using GPU acceleration results in approximately a 2 × speed-up and more than a 3 × speed-up for even longer time series. For the RR, we can speed up the calculation significantly (at least 15 × on tested hardware) using GPUs.

This speed-up is useful as the recurrence rate (RR) is commonly employed to determine an appropriate recurrence threshold ϵ, such as finding values that will yield neither too many recurrences nor too few. Following Webber Jr and Zbilut [56] and Marwan et al. [9], ϵ is typically adjusted so that RR remains within 1–5 %, or up to 10 %, ensuring comparable recurrence densities across embeddings and datasets. Thiel et al. [57] further suggested inspecting RR (ϵ) to locate the range where the recurrence plot is neither too sparse nor saturated. This RR-based normalization has also been applied in behavioral and cognitive studies, for example by Coco and Dale [25], who search for to achieve RR = 2–5 % to standardize recurrence density across participants. In simpler applications, such as change or anomaly detection—RR alone can reveal whether a system remains within previously visited states or exhibits transitions and it may serve as a fast preprocessing measure before performing full RQA with DET and LAM on selected data subsets.

4.3 Global vs. local algorithm

The AccRQA library also contains, as described in Sect. 3, two methods for calculating RQA measures: one that calculates RQA measures, including ENTR, using global operations, and one that does not calculate ENTR. In comparison with state-of-the-art packages, we calculate ENTR as the other packages do. However, if one does not require the ENTR measure, a faster algorithm could be used. The speed-up that could be achieved on our benchmark dataset is shown in Fig. 7. By omitting ENTR, we can get an additional 30–40% reduction in execution time. While for the short time series, the gain is negligible, for long time series, it can significantly shorten the time to result. The total speed-up will depend on RQA parameters, as discussed in the next section.

4.4 Effect of parameters on execution time

Phase space reconstruction and RQA has four parameters that can affect the performance of the AccRQA package. These are threshold ϵ, delay τ, embedding m, and $l_{\min}/v_{\min}$. The simplest case is dependency on $l_{\min}/v_{\min}$ parameters. For the global algorithm, which evaluates the length histogram, the change in $l_{\min}/v_{\min}$ does not require recalculation of RP as all information for new values of RQA measures is contained in the histogram itself, and no reevaluation of recurrence plot (RP) is required. In contrast, the local algorithm must reevaluate RP to calculate RQA measures.

For all other parameters, we have run benchmarks for different values of each parameter and their combination. The results are shown in Fig. 8. There, we can see that the execution time is fairly insensitive to the value of the delay τ. However, at this point, we need to separate the execution time for the global algorithm that calculates ENTR from that of the local algorithm that does not. What has a noticeable effect on a global algorithm are combinations of threshold and embedding, where execution time can change by almost 50%. This comes down to the histogram of line lengths, whose execution time depends on the number of lengths added to it (bottom of Fig. 8). In contrast, the local algorithm does not suffer from these fluctuations as it does not calculate the histogram directly, boosting its speed-up over the global algorithm to 2 × or more in many cases.

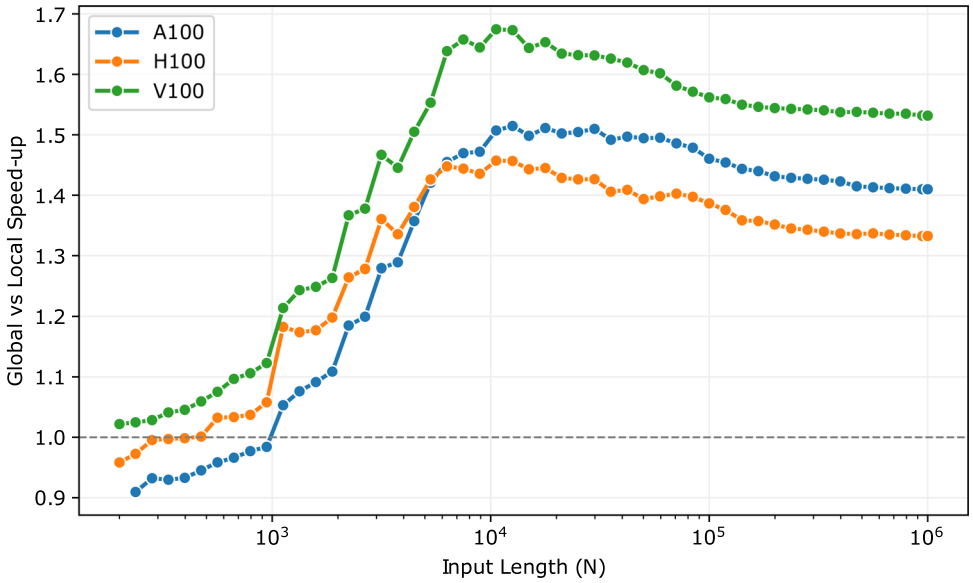


Fig. 7 Speed-up achieved between the global algorithm that calculates ENTR and the local algorithm that does not calculate line length histogram

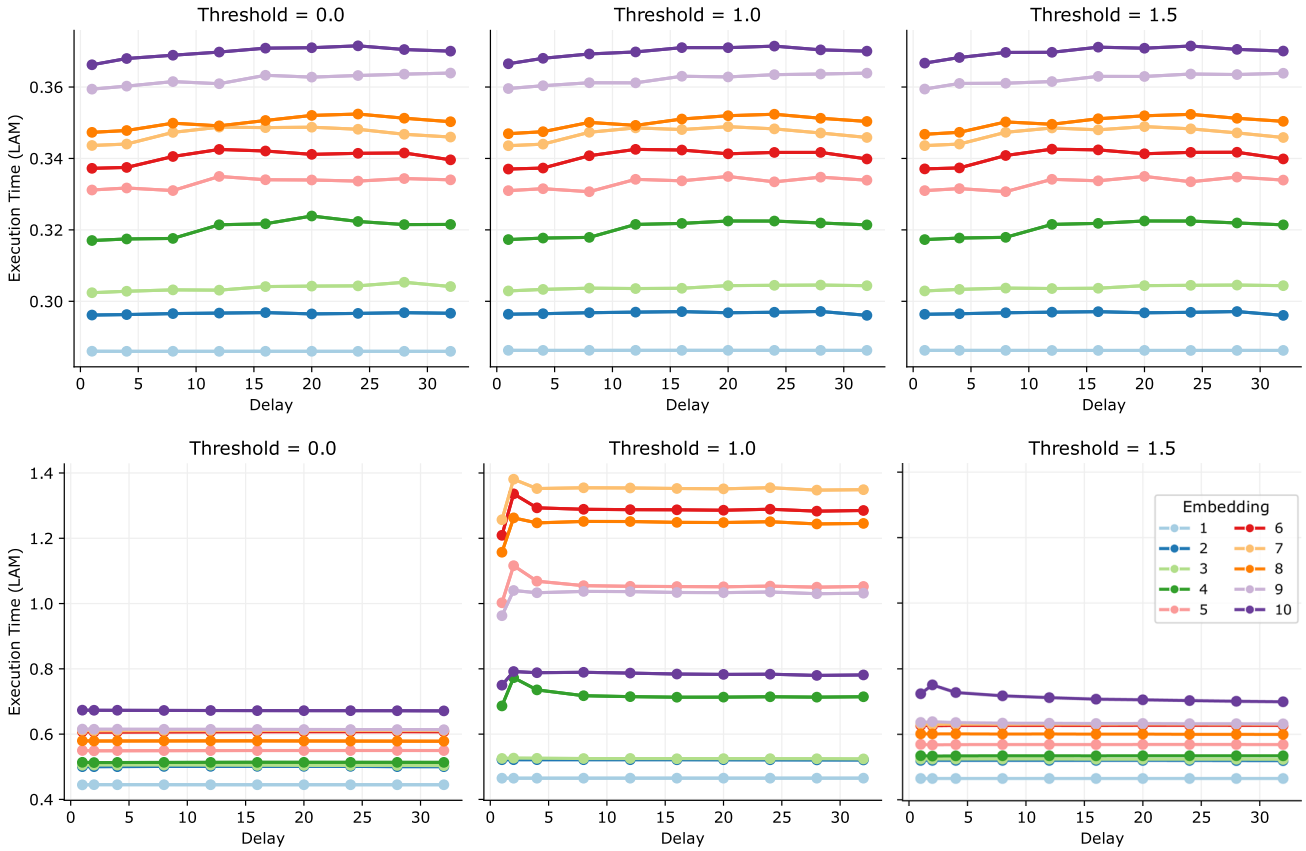


Fig. 8 Effects of threshold ϵ, delay τ, and embedding m on the execution time in the case of the local algorithm not computing the line length histogram (first row) and the global algorithm calculating ENTR (second row)

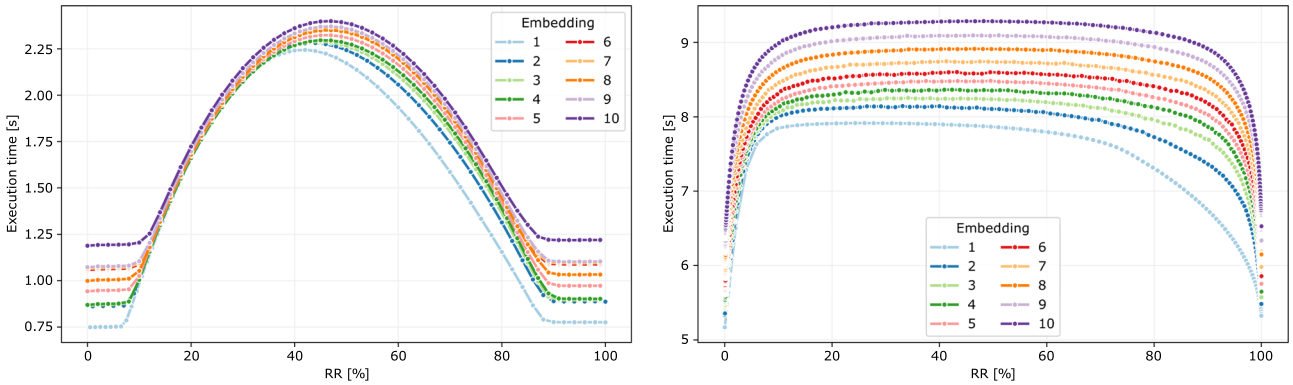


Fig. 9 Effects of the threshold via the recurrence rate (RR) in %, and embedding m with delay $\tau = 2$ on the total execution time for AccRQA (left) and PyRQA (right)

As increases in execution time of up to 50% are dramatic, we have compared AccRQA's increases with those suffered by the PyRQA package. This comparison is shown in Fig. 9, and we can see that PyRQA sees similar increases as AccRQA.

5 Library overview

AccRQA library is available on GitHub (https://github.com/KAdamek/AccRQA). The AccRQA library can be used from `C/C++`, `Python` through the AccRQA Python package (https://pypi.org/project/accrqa/) and `R` through the R source package, which is available on GitHub (https://github.com/KAdamek/AccRQA/releases). The repository includes the C/C++ code of the library itself, an example C/C++ application, Python package files along with Python scripts that demonstrate the use of the AccRQA library, and an R package with examples of how to use it.

The AccRQA interface is designed to be invariant across different floating-point precisions (`FP32` and `FP64`) and to be platform-independent (CPUs and NVIDIA GPUs). The computational platform can be specified during function invocation. If the selected computational platform is unavailable, the library defaults to using the CPU. This enables users to develop or test their data workflow or pipeline locally on a CPU or consumer-grade accelerator, and to seamlessly transfer it to an HPC environment to utilize additional computational resources without modifying the code.

The AccRQA library supports CPUs and NVIDIA GPUs, with support for AMD GPUs planned for the future. Dependencies of the AccRQA library were kept to a minimum to increase portability. The C/C++ library requires `cmake>=3.13`, `gcc>=8.1` and optional dependency is `CUDA>=7.0`. The Python package further depends on `NumPy>=1.7.0`, `Pandas>=1.4.3` and for installation `setuptools>=63.1.0`. The `R` package depends on `Rcpp`, `tidyr`, and `dplyr`.

Documentation is available on GitHub pages (https://kadamek.github.io/AccRQA/index.html), which includes a description of the library functions and instructions on how to install the library. For reporting bugs, issues, and requests, GitHub issues can be used.

5.1 Functions

The functionality of the AccRQA library can be accessed using three programming languages: `Python`, `R`, and `C/C++`. Interfaces in those languages are similar, although there are some exceptions.

AccRQA library offers three main functions which provide calculation of RR alone, LAM and associated measures (TT, $V_{\max}$, $\mathrm{ENTR}_{\mathrm{LAM}}$, RR) and DET and associated measures (L, $L_{\max}$, $\mathrm{ENTR}_{\mathrm{DET}}$, RR).

5.1.1 Support classes

In Python, there are two supporting classes that facilitate the selection of the distance metric and the computational platform. These are:

- `accrqa.accrqaDistance(distance_type)`—selects the distance metric that can be either *euclidian* for l^2 norm or *maximal* for l^∞ norm.

- `accrqa.accrqaCompPlatform(comp_platform)`—selects computational platform that can be either *cpu* for CPUs or *nv_gpu* for NVIDIA GPUs.

5.1.2 Recurrence rate (RR)

If the recurrence rate is the only RQA measure required, then AccRQA offers a function that only calculates RR with the benefit that calculation of RR alone is faster than getting it along with other RQA measures. To calculate RR, use:

Python:`accrqa.RR(input_data, tau_values, emb_values, threshold_values, distance_type, comp_platform, tidy_data=True),`

R:`accrqa_RR(input_data, tau_values, emb_values, threshold_values, distance_type = "euclidean", comp_platform),`

5.1.3 Laminarity (LAM)

To calculate LAM and associated measures (TT, $V_{\max}$, $\mathrm{ENTR}_{\mathrm{LAM}}$, RR), use:

Python:`accrqa.LAM(input_data, tau_values, emb_values, vmin_values, threshold_values, distance_type, calculate_ENTR=True, comp_platform, tidy_data=True),`

R:`accrqa_LAM(input_data, tau_values, emb_values, vmin_values, threshold_values, distance_type = "euclidean", calculate_ENTR = TRUE, comp_platform)`

5.1.4 Determinism (DET)

To calculate DET and associated measures (L, $L_{\max}$, $\mathrm{ENTR}_{\mathrm{DET}}$, RR), use:

Python:`accrqa.DET(input_data, tau_values, emb_values, lmin_values, threshold_values, distance_type, calculate_ENTR=True, comp_platform, tidy_data=True),`

R:`accrqa_DET(input_data, tau_values, emb_values, lmin_values, threshold_values, distance_type = "euclidean", calculate_ENTR = TRUE, comp_platform),`

where `input_data` is the input time series, `tau_values` is an array of delays, `emb_values` is an array of embedding values, `lmin_values` is an array of minimal lengths, `threshold_values` is an array of threshold values, `distance_type` is the type of distance metric used for calculating RP, `calculate_ENTR` enables calculation of $L_{\max}$, $\mathrm{ENTR}_{\mathrm{DET}}$ (enabled by default), `comp_platform` is the computational platform used for calculations. For Python, there is an additional argument `tidy_data`, which is an optional switch to enable or disable output in tidy data format.

5.1.5 Recurrence plot (RP)

AccRQA also supports the calculation of the recurrence plot. To get RP, use:

Python:`accrqa.RP(input_data, tau, emb, threshold, distance_type),`

R:`accrqa_RP(input_data, tau, emb, threshold, distance_type),`

where `tau` is an integer value of the delay, `emb` is an integer value of the embedding, `threshold` is a floating point value of the threshold.

5.1.6 Target RR

To calculate threshold associated with RR, use:

Python:`accrqa.RR_target(input_data, tau, emb, target_RR, distance_type, epsilon, comp_platform, max_iter, threshold_min, threshold_max),`

R:`accrqa_RR_target(input_data, tau, emb, target_RR, epsilon, distance_type, comp_platform, max_iter, threshold_min, threshold_max, n_threshold, verbose),`

Table 3 Table representation of given data

Delay	Embedding	$L_{\min}$	Threshold	DET	L	$L_{\max}$	ENTR	RR
1	1	2	0.5	0.933	4.941	9.0	1.571	0.900
1	1	3	0.5	0.888	5.333	9.0	1.319	0.900
1	2	2	0.5	0.970	4.333	8.0	1.319	0.827
1	2	3	0.5	0.850	5.181	8.0	0.967	0.827
2	1	2	0.5	0.933	4.941	9.0	1.571	0.900
2	1	3	0.5	0.888	5.333	9.0	1.319	0.900
2	2	2	0.5	0.851	4.181	7.0	0.967	0.843
2	2	3	0.5	0.629	6.800	7.0	0.636	0.843

where `target_RR` is the target value of recurrence rate, `epsilon` is the floating-point precision of the target RR and RR resulting from the threshold found by the function, `max_iter` is the maximum number of iterations, `threshold_min` and `threshold_max` is a range of threshold values to be searched. Furthermore, in R, `n_threshold` specifies the number of thresholds to search in each iteration, and `verbose` enables debug messages.

5.2 Data format

The AccRQA library provides an interface for multiple languages with different philosophies and operational approaches. Therefore, data formats used by the AccRQA, to some extent, depend on the language environment. In general, we aim to adopt principles of tidy data [58]. The exception to this is C/C++ interfaces, which use an array-of-structures approach.

Tidy data principals are [58]:

1. Each variable forms a column.
2. Each observation forms a row.
3. Each type of observational unit forms a table.

The result of a single run of the AccRQA function (either in `Python` or in `R`) to calculate DET in the form of a tidy data table is shown in Table 3. The code producing this result is

```
tau_values = np.array([1,2], dtype=np.intc)

emb_values = np.array([1,2], dtype=np.intc)

lmin_values = np.array([2,3], dtype=np.intc)

vmin_values = np.array([2,3], dtype=np.intc)

threshold_values = np.array([0.5], dtype=np.float32)

input_data = np.random.rand(10)

input_data = input_data.astype('float32')

rqa.DET(

    input_data, tau_values, emb_values, lmin_values, threshold_values,

    distance_type='maximal', calculate_ENTR = True, tidy_data = True

)
```

In different languages, the result is represented using different packages.

In the case of `C/C++`, the data must be accessed in a specific order, as the `C/C++` interface returns a raw pointer.

```
for(tau_idx=0; tau_idx<num_tau; tau_idx++){
    for(emb_idx=0; emb_idx<num_emb; emb_idx++){
        for(lmin_idx=0; lmin_idx<num_lmin; lmin_idx++){
            for(thr_idx=0; thr_idx<num_threshold; thr_idx++){
                pos = tau_idx*num_lmin*num_emb*num_threshold +
                    + emb_idx*num_lmin*num_threshold +
                    + lmin_idx*num_threshold +
                    + thr_idx;
                DET  = rqa_measures[pos + 0]
                L    = rqa_measures[pos + 1]
                Lmax = rqa_measures[pos + 2]
                ENTR = rqa_measures[pos + 3]
                RR   = rqa_measures[pos + 4]
            }
        }
    }
}
```

For LAM measures, the structure is similar.

6 Conclusion

This paper has introduced the AccRQA library, a high-performance, portable, and memory-efficient software solution for conducting recurrence quantification analysis (RQA). AccRQA addresses the growing computational demands of modern scientific research by offering novel parallel algorithms that are accelerated using both multi-core CPUs and NVIDIA GPUs.

The performance benchmarks demonstrate that AccRQA offers higher performance than existing state-of-the-art libraries. On server-grade hardware, AccRQA achieves an average 10 × speed-up and at least 6 × speed-up compared to PyRQA and pyunicorn using CPUs. On NVIDIA GPUs, AccRQA is 15 × faster than PyRQA (NVIDIA A100, H100). This performance extends to consumer-grade hardware, making RQA more accessible even on large datasets.

Beyond its speed, AccRQA is designed for practical use. It features a minimal memory footprint, enabling the analysis of time series exceeding 1 million data points on standard hardware. It is portable across operating systems (Linux, Windows, macOS), programming languages (C/C++, Python, R) and computational platforms (CPUs, GPUs). This, combined with a tidy data output format, ensures that AccRQA can be easily integrated into many scientific workflows and makes transitions from consumer-grade hardware to HPC environment seamless.

In summary, AccRQA is a fast and versatile tool in the computational toolbox available for nonlinear time-series analysis. Providing a fast, efficient, and accessible library makes RQA techniques available to researchers across disciplines, from physics and climate science to economics and psychology, enabling them to tackle larger datasets and more complex problems.

Acknowledgements The creation of this article was supported by the grant program Vouchers for Universities in the Moravian-Silesian Region (Operational Program Just Transformation, registration number CZ.10.03.01/00/23_042/0000390), within the project, Spectral Graph: a new data structure for Monte Carlo Radiative Transport. The authors would also like to acknowledge the use of the University of Oxford Advanced Research Computing

(ARC) [59] facility in carrying out this work. This work was supported by the internal Grant IGS/19/2024 of the Silesian University in Opava. The authors were supported by the Research Center for Theoretical Physics and Astrophysics, Institute of Physics in Opava, Silesian University in Opava.

Funding Open access publishing supported by the institutions participating in the CzechELib Transformative Agreement.

Data availability statement GitHub: https://github.com/KAdamek/AccRQA PyPi: https://pypi.org/project/accrqa/ R source package: https://github.com/KAdamek/AccRQA/releases Documentation: https://kadamek.github.io/AccRQA/index.html Benchmarking script and data: https://github.com/KAdamek/AccRQA/releases

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. J.P. Zbilut, C.L. Webber, Embeddings and delays as derived from quantification of recurrence plots. Phys. Lett. A **171**(3), 199–203 (1992). https://doi.org/10.1016/0375-9601(92)90426-M
2. N. Marwan, A historical review of recurrence plots. Eur. Phys. J. Spec. Top. **164**(1), 3–12 (2008). https://doi.org/10.1140/epjst/e2008-00829-1
3. N. Marwan, K.H. Kraemer, Trends in recurrence analysis of dynamical systems. Eur. Phys. J. Spec. Top. **232**(1), 5–27 (2023). https://doi.org/10.1140/epjs/s11734-022-00739-8
4. Z. Guimarães-Filho, I.L. Caldas, R.L. Viana, I.C. Nascimento, Y.K. Kuznetsov, J. Kurths, Recurrence quantification analysis of turbulent fluctuations in the plasma edge of tokamak Chauffage Alfvén Brésilien tokamak. Phys. Plasmas **17**, 012303 (2010)
5. S. Britzen, M. Zajaček, L.V. Popović, C. Fendt, A. Tramacere, I.N. Pashchenko, F. Jaron, R. Pánis, L. Petrov, M.F. Aller, H.D. Aller, A ring accelerator? Unusual jet dynamics in the IceCube Candidate PKS 1502+106. Mon. Not. R. Astron. Soc. **503**(3), 3145–3178 (2021). https://doi.org/10.1093/mnras/stab589
6. G. Bhatta, R. Pánis, Z. Stuchlík, Deterministic aspect of the γ-ray variability in blazars. Astrophys. J. **905**(2), 160 (2020). https://doi.org/10.3847/1538-4357/abc625
7. R. Pánis, M. Kološ, Z. Stuchlík, Determination of chaotic behaviour in time series generated by charged particle motion around magnetized Schwarzschild black holes. Eur. Phys. J. C (2019). https://doi.org/10.1140/epjc/s10052-019-6961-7
8. T. Westerhold, N. Marwan, A.J. Drury, D. Liebrand, C. Agnini, E. Anagnostou, J.S.K. Barnet, S.M. Bohaty, D.D. Vleeschouwer, F. Florindo, T. Frederichs, D.A. Hodell, A.E. Holbourn, D. Kroon, V. Lauretano, K. Littler, L.J. Lourens, M. Lyle, H. Pälike, U. Röhl, J. Tian, R.H. Wilkens, P.A. Wilson, J.C. Zachos, An astronomically dated record of earth's climate and its predictability over the last 66 million years. Science **369**(6509), 1383–1387 (2020). https://doi.org/10.1126/science.aba6853
9. N. Marwan, M. Carmen Romano, M. Thiel, J. Kurths, Recurrence plots for the analysis of complex systems. Phys. Rep. **438**(5), 237–329 (2007). https://doi.org/10.1016/j.physrep.2006.11.001
10. E.N. McCormick, L.M. Blaha, C. Gonzalez, Exploring dynamic decision making strategies with recurrence quantification analysis, in *Proceedings of the Annual Meeting of the Cognitive Science Society*, vol. 42 (2020). https://escholarship.org/uc/item/6fj218hh. Accessed 1 Mar 2025
11. S. Wallot, G. Hollis, M. Rooij, Connected text reading and differences in text reading fluency in adult readers. PLoS ONE **8**(8), 1–22 (2013). https://doi.org/10.1371/journal.pone.0071914
12. G. Orlando, G. Zimatore, Recurrence quantification analysis of business cycles. Chaos Solitons Fract. **110**, 82–94 (2018). https://doi.org/10.1016/j.chaos.2018.02.032
13. B. Wang, W. Qiu, X. Hu, W. Wang, A rolling bearing fault diagnosis technique based on recurrence quantification analysis and bayesian optimization SVM. Appl. Soft Comput. **156**, 111506 (2024). https://doi.org/10.1016/j.asoc.2024.111506
14. T. Haj Mohamad, Y. Chen, Z. Chaudhry, C. Nataraj, Gear fault detection using recurrence quantification analysis and support vector machine. J. Softw. Eng. Appl. (2018). https://doi.org/10.4236/jsea.2018.115012
15. A. Syta, J. Czarnigowski, P. Jakliński, N. Marwan, Detection and identification of cylinder misfire in small aircraft engine in different operating conditions by linear and non-linear properties of frequency components. Measurement **223**, 113763 (2023). (**10.1016/j.measurement.2023.113763**)
16. V. Nair, G. Thampi, R.I. Sujith, Intermittency route to thermoacoustic instability in turbulent combustors. J. Fluid Mech. **756**, 470–487 (2014). https://doi.org/10.1017/jfm.2014.468

17. M. Volk, D. Staegemann, S. Bosse, R. Häusler, K. Turowski, Approaching the (big) data science engineering process, in *Proceedings of the 5th International Conference on Internet of Things, Big Data and Security—IoTBDS* (SciTePress, Setúbal, Portugal, 2020), pp. 428–435. https://doi.org/10.5220/0009569804280435
18. L. Clissa, M. Lassnig, L. Rinaldi, How big is big data? A comprehensive survey of data production, storage, and streaming in science and industry. Front. Big Data (2023). https://doi.org/10.3389/fdata.2023.1271639
19. J.T. Overpeck, G.A. Meehl, S. Bony, D.R. Easterling, Climate data challenges in the 21st century. Science **331**(6018), 700–702 (2011). https://doi.org/10.1126/science.1197869
20. P. Clarke, P. Coveney, A. Heavens, J. Jäykkä, B. Joachimi, A. Karastergiou, N. Konstantinidis, A. Korn, R. Mann, J. McEwen et al., Big data in the physical sciences: challenges and opportunities. ATI Scoping Report (2016)
21. B. Knowles, Computing and climate change. ACM TechBrief (2021)
22. A.E. Trefethen, J. Thiyagalingam, Energy-aware software: challenges, opportunities and strategies. J. Comput. Sci. **4**(6), 444–449 (2013). https://doi.org/10.1016/j.jocs.2013.01.005. (**Scalable Algorithms for Large-Scale Systems Workshop (ScalA2011), Supercomputing 2011**)
23. K. Adámek, J. Novotný, J. Thiyagalingam, W. Armour, Efficiency near the edge: increasing the energy efficiency of ffts on gpus for real-time edge computing. IEEE Access **9**, 18167–18182 (2021)
24. M.I. Coco, D. Mønster, G. Leonardi, R. Dale, S. Wallot, Unidimensional and multidimensional methods for recurrence quantification analysis with crqa. R J. **13**, 112–130 (2021). https://doi.org/10.32614/RJ-2021-062
25. M.I. Coco, R. Dale, Cross-recurrence quantification analysis of categorical and continuous time series: an r package. Front. Psychol. (2014). https://doi.org/10.3389/fpsyg.2014.00510
26. S. Wallot, Multidimensional cross-recurrence quantification analysis (mdcrqa)—a method for quantifying correlation between multivariate time-series. Multivar. Behav. Res. **54**(2), 173–191 (2019). https://doi.org/10.1080/00273171.2018.1512846
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, Unified functional network and nonlinear time series analysis for complex systems science: the pyunicorn package. Chaos **25**(11), 113101 (2015). https://doi.org/10.1063/1.4934554
28. H.D. Rosario, RecurrenceAnalysis.jl. GitHub (2016). https://github.com/JuliaDynamics/RecurrenceAnalysis.jl. Accessed 1 Mar 2026
29. T. Rawald, M. Sips, N. Marwan, Pyrqa-conducting recurrence quantification analysis on very long time series efficiently. Comput. Geosci. **104**, 101–108 (2017). https://doi.org/10.1016/j.cageo.2016.11.016
30. N. Marwan, K.H. Krämer, pucicu/rp: Recurrence plot and recurrence quantification analysis implementation for MATLAB. Zenodo (2022). https://doi.org/10.5281/zenodo.6325324
31. M.I. Coco, D. Mønster, G. Leonardi, R. Dale, S. Wallot, J.D. Dixon, J.C. Nash, A. Paxton. Crqa: cross recurrence quantification analysis for categorical and continuous time series. (2025). https://doi.org/10.32614/CRAN.package.crqa. R package version 2.0.7
32. R. Pánis, K. Adámek, N. Marwan, Averaged recurrence quantification analysis: method omitting the recurrence threshold choice. Eur. Phys. J. Spec. Top. **232**(1), 47–56 (2023). https://doi.org/10.1140/epjs/s11734-022-00686-4
33. N. Marwan, N. Wessel, U. Meyerfeldt, A. Schirdewan, J. Kurths, Recurrence-plot-based measures of complexity and their application to heart-rate-variability data. Phys. Rev. E **66**, 026702 (2002). https://doi.org/10.1103/PhysRevE.66.026702
34. C.L. Webber, J.P. Zbilut, Dynamical assessment of physiological systems and states using recurrence plot strategies. J. Appl. Physiol. **76**(2), 965–973 (1994). https://doi.org/10.1152/jappl.1994.76.2.965
35. J.P. Zbilut, C.L. Webber, Recurrence quantification analysis: Introduction and historical context. Int. J. Bifurc. Chaos **17**(10), 3477–3481 (2007). https://doi.org/10.1142/S0218127407019238
36. F. Takens, Detecting strange attractors in turbulence, in *Dynamical Systems and Turbulence, Warwick 1980*. ed. by D. Rand, L.-S. Young (Springer, Berlin, 1981), pp.366–381
37. H. Kantz, T. Schreiber, *Nonlinear Time Series Analysis*, 2nd edn. (Cambridge University Press, Cambridge, 2003). https://doi.org/10.1017/CBO9780511755798
38. K.H. Kraemer, N. Marwan, Border effect corrections for diagonal line based recurrence quantification analysis measures. Phys. Lett. A **383**(34), 125977 (2019). https://doi.org/10.1016/j.physleta.2019.125977
39. D. Horn, Stream reduction operations for gpgpu applications, in *GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation (Gpu Gems)* (Addison-Wesley Professional, Boston, MA, 2005), pp. 573–589
40. W.D. Hillis, G.L. Steele, Data parallel algorithms. Commun. ACM **29**(12), 1170–1183 (1986). https://doi.org/10.1145/7902.7903
41. N. Marwan, Y. Zou, N. Wessel, M. Riedl, J. Kurths, Estimating coupling directions in the cardiorespiratory system using recurrence properties. Philos. Trans. A Math. Phys. Eng. Sci. **371**(1997), 20110624 (2013)
42. I. Konvalinka, D. Xygalatas, J. Bulbulia, U. Schjødt, E.-M. Jegindø, S. Wallot, G.V. Orden, A. Roepstorff, Synchronized arousal between performers and related spectators in a fire-walking ritual. Proc. Natl. Acad. Sci. **108**(20), 8514–8519 (2011). https://doi.org/10.1073/pnas.1016955108
43. P. Melillo, M. Bracale, L. Pecchia, Nonlinear heart rate variability features for real-life stress detection. Case study: students under stress due to university examination. Biomed. Eng. Online **10**(1), 96 (2011)
44. M.A. Lopes, J. Zhang, D. Krzemiński, K. Hamandi, Q. Chen, L. Livi, N. Masuda, Recurrence quantification analysis of dynamic brain networks. Eur. J. Neurosci. **53**(4), 1040–1059 (2021). https://doi.org/10.1111/ejn.14960

45. C.I. Ledesma-Ramírez, J.J. Hernández-Gloria, E. Bojorges-Valdez, O. Yanez-Suarez, O. Piña-Ramírez, Recurrence quantification analysis during a mental calculation task. Chaos **33**(6), 063154 (2023)
46. C. Manetti, M.-A. Ceruso, A. Giuliani, C.L. Webber, J.P. Zbilut, Recurrence quantification analysis as a tool for characterization of molecular dynamics simulations. Phys. Rev. E **59**, 992–998 (1999). https://doi.org/10.1103/PhysRevE.59.992
47. H. Castellini, L. Romanelli, Applications of recurrence quantified analysis to study the dynamics of chaotic chemical reaction. Phys. A **342**(1), 301–307 (2004). https://doi.org/10.1016/j.physa.2004.06.028. (**Proceedings of the VIII Latin American Workshop on Nonlinear Phenomena**)
48. J.P. Zbilut, A. Giuliani, C.L. Webber, A. Colosimo, Recurrence quantification analysis in structure-function relationships of proteins: an overview of a general methodology applied to the case of TEM-1 beta-lactamase. Protein Eng. **11**(2), 87–93 (1998). https://doi.org/10.1093/protein/11.2.87
49. K. Kodama, D. Shimizu, R. Dale, K. Sekine, An approach to aligning categorical and continuous time series for studying the dynamics of complex human behavior. Front. Psychol. **12**, 614431 (2021). https://doi.org/10.3389/fpsyg.2021.614431
50. K.K. Nawas, A. Shahina, K. Balachandar, P. Maadeshwaran, N.G. Devanathan, N. Kumar, A.N. Khan, Recurrence plot embeddings as short segment nonlinear features for multimodal speaker identification using air, bone and throat microphones. Sci. Rep. **14**, 12513 (2024). https://doi.org/10.1038/s41598-024-62406-3
51. S. Wallot, G. Leonardi, Analyzing multivariate dynamics using cross-recurrence quantification analysis (CRQA), diagonal-cross-recurrence profiles (DCRP), and multidimensional recurrence quantification analysis (MdRQA)—a tutorial in R. Front. Psychol. **9**, 2232 (2018)
52. T. Kovács, Recurrence network analysis of exoplanetary observables. Chaos **29**(7), 071105 (2019). https://doi.org/10.1063/1.5109564
53. R.A. Phillipson, P.T. Boyd, A.P. Smale, M.S. Vogeley, Complex variability of Kepler agn revealed by recurrence analysis. Mon. Not. R. Astron. Soc. **497**(3), 3418–3439 (2020). https://doi.org/10.1093/mnras/staa2069
54. T. Chelidze, T. Matcharashvili, Complexity of seismic process; measuring and applications—a review. Tectonophysics **431**(1), 49–60 (2007). https://doi.org/10.1016/j.tecto.2006.05.029
55. M. Lin, G. Zhao, G. Wang, Recurrence quantification analysis for detecting dynamical changes in earthquake magnitude time series. Int. J. Mod. Phys. C **26**(07), 1550077 (2015). https://doi.org/10.1142/S0129183115500771
56. C.L. Webber Jr., J.P. Zbilut, Recurrence quantification analysis of nonlinear dynamical systems, in *Tutorials in Contemporary Nonlinear Methods for the Behavioral Sciences* (National Science Foundation, Alexandra, VA, USA, 2005), pp. 26–94
57. M. Thiel, M. Romano, J. Kurths, How much information is contained in a recurrence plot? Phys. Lett. A **330**(5–6), 343–349 (2004). https://doi.org/10.1016/j.physleta.2004.07.050
58. H. Wickham, Tidy data. J. Stat. Softw. (2014). https://doi.org/10.18637/jss.v059.i10
59. A. Richards, University of Oxford Advanced Research Computing. Zenodo (2015). https://doi.org/10.5281/zenodo.22558