entropy

MDPI

Article

Teaching Ordinal Patterns to a Computer: Efficient Encoding Algorithms Based on the Lehmer Code

Sebastian Berger [1,*], Andrii Kravtsiv [1], Gerhard Schneider [1] and Denis Jordan [2]

[1] Department of Anaesthesiology and Intensive Care, Klinikum rechts der Isar der Technischen Universität München, 81675 Munich, Germany; andrii.kravtsiv@tum.de (A.K.); gerhard.schneider@tum.de (G.S.)
[2] Institute of Geomatics Engineering, University of Applied Sciences and Arts Northwestern Switzerland, 4132 Muttenz, Switzerland; denis.jordan@fhnw.ch
* Correspondence: sebastian.berger@tum.de

Received: 25 July 2019; Accepted: 18 October 2019; Published: 21 October 2019

Abstract: Ordinal patterns are the common basis of various techniques used in the study of dynamical systems and nonlinear time series analysis. The present article focusses on the computational problem of turning time series into sequences of ordinal patterns. In a first step, a numerical encoding scheme for ordinal patterns is proposed. Utilising the classical Lehmer code, it enumerates ordinal patterns by consecutive non-negative integers, starting from zero. This compact representation considerably simplifies working with ordinal patterns in the digital domain. Subsequently, three algorithms for the efficient extraction of ordinal patterns from time series are discussed, including previously published approaches that can be adapted to the Lehmer code. The respective strengths and weaknesses of those algorithms are discussed, and further substantiated by benchmark results. One of the algorithms stands out in terms of scalability: its run-time increases linearly with both the pattern order and the sequence length, while its memory footprint is practically negligible. These properties enable the study of high-dimensional pattern spaces at low computational cost. In summary, the tools described herein may improve the efficiency of virtually any ordinal pattern-based analysis method, among them quantitative measures like permutation entropy and symbolic transfer entropy, but also techniques like forbidden pattern identification. Moreover, the concepts presented may allow for putting ideas into practice that up to now had been hindered by computational burden. To enable smooth evaluation, a function library written in the C programming language, as well as language bindings and native implementations for various numerical computation environments are provided in the supplements.

Keywords: Lehmer code; ordinal patterns; symbolic dynamics; permutation entropy; symbolic transfer entropy

1. Introduction

The article *Permutation Entropy: A Natural Complexity Measure for Time Series* by Christoph Bandt and Bernd Pompe [1] pioneered a novel approach towards nonlinear time series analysis. In essence, the time series of interest is embedded in an m-dimensional phase space, then each delay vector is discretised according to the ordinal ranks among its m components. This procedure yields a sequence of symbols synonymously called order patterns or ordinal patterns, whereas the parameter m is either referred to as the embedding dimension, or simply the *order* of the ordinal pattern. Permutation entropy (PeEn) is in turn defined as the Shannon entropy [2,3] of a marginal probability distribution of such ordinal patterns.

Comprehensive overviews on the many applications of PeEn are given in [4–6]. Following the initial publication of 2002 [1], numerous extensions and modifications of PeEn have been devised, for instance, the methods proposed in [7–10]. Apart from Shannon entropy, other information-theoretic measures have been applied to ordinal pattern distributions, among them conditional entropy [11], mutual information [12], and transfer entropy [13,14]. Recurrence plots [15] and various correlation functions [16] were also transferred to the so-called ordinal pattern space [17]. On a more abstract level, ordinal patterns were tightly integrated into the general theory of discrete dynamics. A thorough introduction to such topics is provided in the book *Permutation Complexity in Dynamical Systems* by José Amigó [18].

1.1. Efficient but Infeasible?

Besides its conceptual simplicity and its robustness against certain forms of measurement noise, computational efficiency is likely one of the most-cited advantages of the Bandt–Pompe approach towards time series analysis [1,4,6–11,14,15,19–24]. However, this well-acclaimed run-time behaviour is not a specific property of ordinal analysis, but constitutes a feature of discrete dynamics in general. By matter of principle, coarse-graining the phase space of a dynamical system can radically reduce the computational cost of its analysis because quantisation turns continuous probability densities into discrete probability masses. A concise example (intentionally unrelated to ordinal patterns) can be found in [25], wherein Andreas Kaiser and Thomas Schreiber address the intricacies of estimating transfer entropy from continuously-valued time series, as compared to the far simpler discrete case.

Before the computational benefits of symbolisation can take any effect in ordinal analysis, the (usually real-valued) input data need to be converted into sequences of discrete ordinal patterns. Somewhat paradoxically, extracting ordinal patterns from time series is computationally a lot heavier than literature commonly suggests. Determining a single ordinal pattern of order m requires a total of $(m^2 - m)/2$ pairwise comparisons, resulting in a computational complexity of $O(m^2)$. In other words, "the computation time increases rapidly with m"—as has been pointed out by Matthäus Staniek and Klaus Lehnertz [26], the creators of symbolic transfer entropy [14]. In a similar context, Amigó stated that "there is no substitute for substantial computational effort when [the order m] becomes sufficiently large", and further conjectured that working with ordinal patterns beyond approximately $m = 12$ may likely be "computationally unfeasible" [18].

Besides computational complexity, another closely related issue is the spatial complexity of ordinal analysis: how should ordinal patterns best be represented in the digital domain, and what amount of extra memory is required to obtain that representation? Because a total of $m!$ different ordinal patterns of order m exist (see Section 2.2), their memory footprint scales at a super-exponential rate of $O(m!)$. Although posing a computational challenge to the investigator, this combinatorial explosion also has a beautiful application in the study of complex dynamics. The increasing spatial complexity of ordinal patterns will at some point transcend the irregularity producible by any chaotic dynamics, which gives rise to the notion of so-called forbidden ordinal patterns. This term describes patterns that do exist in theory, but cannot be generated under the particular evolution rule of a given dynamical system. Their presence or absence can therefore guide the distinction between complex determinism and mere randomness [18,27,28]. In the words of Amigó and colleagues: "Chaos [...] cannot cope with a super-exponentially growing manifold such as that of order patterns" [27].

Against this backdrop, and especially considering the semantic gap between "high efficiency" and downright "infeasibility", it seems justified to elaborate on the computational pitfalls and algorithmic possibilities of encoding ordinal patterns. Literature on this subject is still rather scarce. To the best of our knowledge, there is only one group of researchers who have published on the topic until now, which is the team led by Karsten Keller. In their white paper on ordinal analysis, Karsten Keller, Mathieu Sinn and Jan Emonds proposed a numerical encoding scheme for ordinal patterns [17], which Valentina Unakafova and Karsten Keller subsequently optimised for speed of execution [21]. The results of their work enable the efficient extraction of ordinal patterns from time series for the most commonly used pattern orders [29] that is, for $m \in \{2, 3, \ldots, 9\}$. (To avoid ambiguity, please note that the Keller group prefers a different definition of the pattern order, according to which the ordinal pattern of an m-dimensional vector is of order $d = m - 1$.) Expanding on the aforementioned work of the Keller group [17,21], the present article proposes to use the Lehmer code for mapping ordinal patterns onto non-negative integer values, and demonstrates that simple, efficient and versatile algorithms result from this modification.

1.2. Structure of the Article

Section 2 discusses a few simple (yet rather uncommon) mathematical concepts and notations, including the definition of ordinal patterns used throughout the present article. On that basis, Section 3 discusses the Lehmer code as an advantageous numerical representation for ordinal patterns, and presents a closed-form solution for their encoding. A comparable approach, originally proposed in [17], is also summarised and put into context. Section 4 then expands on these concepts, describing three different algorithms for transforming univariate time series into sequences of ordinal patterns—among them, the aforementioned solution by Unakafova and Keller [21]. The main computational challenges of turning those algorithms into run-time efficient code are addressed in Section 5, and possible optimisation techniques are suggested. Both aspects are immediately substantiated by run-time measurements. The hurried reader can safely skip to the concluding Section 6, which should provide enough information to put the ideas of the article into action by utilising the supplementary software library. That being said, readers concerned with the inner workings of ordinal pattern encoding (and those who lack blind trust in foreign code) are invited to follow through the article in its entirety. Let us start with some mathematical underpinnings.

2. Preliminaries

2.1. Iversonian Brackets

Throughout this article, we will be using a highly convenient, if slightly uncommon notational convention, called the Iversonian bracket [30,31]. In terms of computer science, the Iversonian bracket represents a data type conversion from Boolean to integer: for a given logical expression L, it holds that

$$[L] = \begin{cases} 0, & \text{if } L \text{ is false,} \\ 1, & \text{if } L \text{ is true.} \end{cases} \tag{1}$$

For example, the number of positive elements in a finite time series $\{x_1, x_2, \ldots, x_N\}$ can compactly be written as $\sum_{t=1}^{N} [x_t > 0]$.

2.2. Ordinal Patterns

Any m-tuple $(x_1, x_2, \ldots, x_m) \in X^m$ of pairwise distinct elements from a totally ordered set X has a unique ordinal pattern. This abstract entity describes how the tuple's elements relate to one another in terms of position and rank order. For example, the ordinal pattern of the tuple $(17, 7, 8) \in \mathbb{N}^3$ is fully specified by:

"There are three elements, the first is the greatest, the second is the least." (2)

This same ordinal pattern applies to any tuple (x_1, x_2, x_3) for which the order relations $x_2 < x_3 < x_1$ hold. By contrast, each permutation of the elements $\{x_1, x_2, x_3\}$ yields a different ordinal pattern, so any given m-tuple of pairwise distinct elements $(x_1, x_2, \ldots, x_m) \in X^m$ has exactly one out of $m!$ different ordinal patterns. We here call the tuple length $m \in \{2, 3, \ldots\}$ the order of the set of ordinal patterns $\Omega_m = \{\Pi_1, \Pi_2, \ldots, \Pi_{m!}\}$. Any such pattern $\Pi_i \in \Omega_m$ can formally be denoted by a distinct permutation function

$$\sigma \colon \mathbb{N} \to \mathbb{N}, \qquad \text{such that} \qquad x_{\sigma(1)} < x_{\sigma(2)} < \cdots < x_{\sigma(m)},$$

or, equivalently, by its inverse function

$$\sigma^{-1}\colon \mathbb{N} \to \mathbb{N}, \qquad \text{such that} \qquad \sigma^{-1}(i) < \sigma^{-1}(j) \iff x_i < x_j.$$

Intuitively, the permutation function sorts the tuple, whereas its inverse function assigns a unique rank to each element. Variants of both notations coexist in literature [1,14,16–18,23]. For the scope of the current article, we choose to represent ordinal patterns as m-tuples of ranks $(\lambda_1, \lambda_2, \ldots, \lambda_m) \in \mathbb{N}^m$, where $\lambda_i = \sigma^{-1}(i)$.

The strict limitation to pairwise distinct elements is usually dropped in practise. The common approach is to stipulate $\lambda_i < \lambda_j$ for any pair of values $x_i = x_j$ if their order of appearance is $i < j$, and vice versa. (Depending on the amplitude distribution of the input data [22], it may be advisable to use a more sophisticated technique of resolving tied values during data pre-processing, that is, prior to the rank analysis considered here.) Adopting this simple convention, we arrive at the following definition.

Definition 1. *For any given m-tuple* $(x_1, x_2, \ldots, x_m) \in X^m$ *from a totally ordered set X, its ordinal pattern* $\Pi_i \in \Omega_m = \{\Pi_1, \Pi_2, \ldots, \Pi_{m!}\}$ *of order m is the unique m-tuple of ranks* $(\lambda_1, \lambda_2, \ldots, \lambda_m) \in \{1, 2, \ldots, m\}^m$, *such that*

$$\forall i, j \in \{1, 2, \ldots, m\}\colon\ \lambda_i < \lambda_j \iff \big(x_i < x_j \ \vee\ (x_i = x_j \ \wedge\ i < j)\big). \tag{3}$$

Note that, under this definition, any given $\Pi_i \in \Omega_m$ does not only represent an ordinal pattern, but rather *is* the ordinal pattern itself. To motivate this, let us further introduce the function

$$\begin{gathered} \mathrm{op}\colon X^m \to \Omega_m \subset \mathbb{N}^m, \\ (x_1, x_2, \ldots, x_m) \mapsto (\lambda_1, \lambda_2, \ldots, \lambda_m), \end{gathered} \tag{4}$$

which enables statements like $\Pi_i = \mathrm{op}(x_1, x_2, \ldots, x_m)$. Then, due to Definition 1, the rather curious expression $\mathrm{op}(\Pi_i)$ is actually well-defined: it is the ordinal pattern of the ordinal pattern Π_i, which simply is the ordinal pattern of an m-tuple of pairwise distinct positive integers. Moreover, it is easily confirmed that $\mathrm{op}(\Pi_i) = \Pi_i$ for all $\Pi_i \in \Omega_m$, which implies that the function $\mathrm{op} = \mathrm{op} \circ \mathrm{op}$ is an idempotence—the ordinal pattern of another ordinal pattern is that other ordinal pattern.

2.3. Ordinal Processes and Markov Chains

The fundamental idea presented by Bandt and Pompe is to transform a given time series $\{x_t\}$ with time indices $t \in \{1, 2, \ldots\}$ into a sequence of discrete symbols $\{\Pi_t\}$ prior to any further processing. This approach builds upon the delay embeddings used in dynamical systems theory. Using a fixed pattern order $m \in \{2, 3, \ldots\}$ and a time lag $\tau \in \{1, 2, \ldots\}$, one creates the sequence

$$\{\Pi_t\} \qquad \text{with} \qquad \Pi_t = \mathrm{op}(x_t, x_{t+\tau}, \ldots, x_{t+(m-1)\tau}), \tag{5}$$

that is, the ordinal patterns of the consecutive delay vectors of $\{x_t\}$. In the light of Definition 1, this transformation combines delay embeddings with a nonlinear form of vector quantisation. For this reason, some authors prefer to call the pattern order m the embedding dimension of the ordinal pattern.

Assuming that the time series $\{x_t\}$ is a realisation of some stochastic process, it makes sense to postulate that its corresponding sequence of ordinal patterns $\{\Pi_t\}$ originates from an underlying stochastic process as well. In particular, this process is time-discrete, and its state space is the set of ordinal patterns Ω_m. Keller, Sinn, and Emonds coined the term ordinal process for this concept [17]. The most decisive property of an ordinal process of order $m > 2$ is that its random variables can never be independent. In this respect, observe that, for any time t, the m-dimensional delay vectors

$$(x_t, x_{t+\tau}, \ldots, x_{t+(m-1)\tau}) \qquad \text{and} \qquad (x_{t+\tau}, x_{t+2\tau}, \ldots, x_{t+m\tau})$$

overlap in $m-1$ out of m values. Consequently, with Π_t already fixed, an ordinal process cannot draw $\Pi_{t+\tau}$ from its full state space Ω_m, but merely from a subset of cardinality $m!\,/\,(m-1)! = m$. Keller and colleagues regard this property as the very definition [17] of ordinal processes, by contrast with any other process drawing from the state space Ω_m. As already mentioned, ordinal processes of order $m = 2$ constitute an exception in this regard because $(x_t, x_{t+\tau})$ and $(x_{t+\tau}, x_{t+2\tau})$ overlap in merely one value, such that their ordinal patterns have disjoint order relations. Consistently, it holds that $2! = 2$.

For the time lag $\tau = 1$, an ordinal process is a first-order Markov chain. Because of the inter-pattern dependencies just described, its corresponding transition matrix $\mathbf{T}$ is sparse, containing no more than m positive entries per row. Assuming $m = 3$, for instance, the matrix cannot be less sparse than

$$\mathbf{T} = \begin{array}{c} \\ (1,2,3) \\ (1,3,2) \\ (2,1,3) \\ (2,3,1) \\ (3,1,2) \\ (3,2,1) \end{array} \begin{array}{c} \begin{array}{cccccc} (1,2,3) & (1,3,2) & (2,1,3) & (2,3,1) & (3,1,2) & (3,2,1) \end{array} \\ \left(\begin{array}{cccccc} p_{1,1} & p_{1,2} & 0 & p_{1,4} & 0 & 0 \\ 0 & 0 & p_{2,3} & 0 & p_{2,5} & p_{2,6} \\ p_{3,1} & p_{3,2} & 0 & p_{3,4} & 0 & 0 \\ 0 & 0 & p_{4,3} & 0 & p_{4,5} & p_{4,6} \\ p_{5,1} & p_{5,2} & 0 & p_{5,4} & 0 & 0 \\ 0 & 0 & p_{6,3} & 0 & p_{6,5} & p_{6,6} \end{array} \right) \end{array}.$$

In the general case of $\tau > 1$, an ordinal process behaves like τ such Markov chains interleaved. By way of further illustration, a state diagram for the order $m = 3$ (and necessarily, the time lag $\tau = 1$) is depicted in Figure 1.

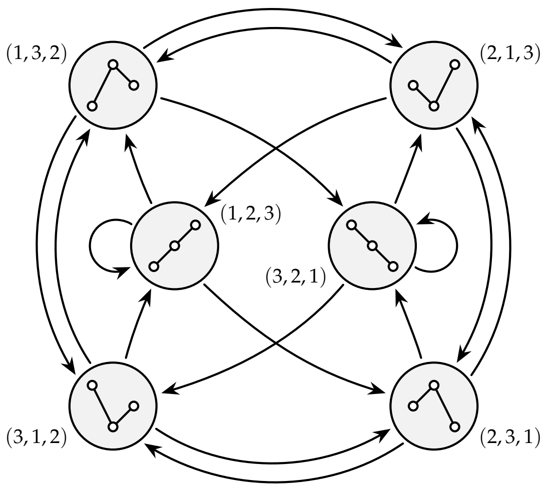

Figure 1. State diagram for an ordinal process of order $m = 3$ and time lag $\tau = 1$, which can be interpreted as a first-order Markov chain. Its transition probabilities depend on the underlying process, and some can be zero. However, no other transitions than the ones depicted are possible because consecutive patterns overlap in two out of three values.

3. Ordinal Patterns in the Digital Domain

In the last section, we defined ordinal patterns as m-tuples of ranks $(\lambda_1, \lambda_2, \ldots, \lambda_m) \in \Omega_m \subset \mathbb{N}^m$, and thereby also established an easily interpretable means of notation. Human interpretability is, however, not a primary concern when storing data in computer memory. Different requirements then prevail, and render the rank representation rather cumbersome. Before we discuss suitable encodings for the digital domain, let us look at these requirements.

Assume that we want to store an ordinal pattern $(\lambda_1, \lambda_2, \ldots, \lambda_m)$ of order m in the main memory of a computer. The naïve solution (for any $m < 256$) would then be to use an array of m consecutive bytes, each holding a particular rank λ_i. This approach is disadvantageous in several respects, the most prominent being:

1. While there are $m!$ distinct ordinal patterns of order m, a block of m bytes can take on 256^m different states. Due to $m! \ll 256^m$ for small m, the memory footprint of the above encoding is far from optimal.
2. Testing a pair of ordinal patterns for equality requires up to m byte-wise comparisons, which is particularly detrimental to the run-time of operations like sorting and searching.
3. Counting distinct pattern occurrences in a sequence of ordinal patterns requires an associative array that provides a map from each possible tuple of ranks to its respective counter variable.

Those shortcomings can be overcome by representing the patterns $\Omega_m = \{\Pi_1, \Pi_2, \ldots, \Pi_{m!}\}$ using non-negative integers $\{0, 1, \ldots, m! - 1\}$ that is, by establishing a bijective map,

$$\begin{aligned} \mathrm{enc}\colon \Omega_m &\to \mathbb{N}_0, \\ \Pi_i &\mapsto i - 1. \end{aligned} \tag{6}$$

Such a map could readily be implemented in software by means of a lookup table. However, encoding a pattern would then require up to $m!$ iterations of that table, with each iteration involving up to m integer comparisons. Fortunately, literature knows better ways of enumerating permutations, and, thus, of encoding ordinal patterns numerically.

3.1. The Keller–Sinn–Emonds Code

Against this backdrop, Keller, Sinn, and Emonds [17] proposed a closed-form solution that directly maps a given m-tuple of values $(x_1, x_2, \ldots, x_m) \in X^m$ onto its ordinal pattern in numerical representation, that is, the authors suggested a function of the form

$$\mathrm{enc} \circ \mathrm{op} = \mathrm{sym}\colon X^m \to \mathbb{N}_0.$$

The principle is as follows: given an ordinal pattern $\Pi_i = (\lambda_1, \lambda_2, \ldots, \lambda_m) \in \Omega_m$ of order m, one interprets its ranks $(\lambda_1, \lambda_2, \ldots, \lambda_m)$ as a permutation of the integers $\{1, 2, \ldots, m\}$, and in turn obtains its right inversion counts

$$(r_1, r_2, \ldots, r_m) \qquad \text{where} \qquad r_i = \sum_{j=i}^{m} [\lambda_i > \lambda_j]. \tag{7}$$

For any fixed pattern order m, there are a total of $m!$ pairwise distinct tuples $(r_1, r_2, \ldots, r_m)$, and each corresponds to a particular ordinal pattern $\Pi_i \in \Omega_m$. Any such tuple of inversion counts can then bijectively be mapped onto a distinct non-negative integer

$$n = \sum_{i=1}^{m} r_i \, \frac{m!}{(m-i+1)!} \qquad \text{such that} \qquad n \in \{0, 1, \ldots, m! - 1\}. \tag{8}$$

With $r_m = 0$ for the rightmost right inversion count, and $\lambda_i > \lambda_j$ if and only if $x_i > x_j$ (see Definition 1), the ordinal pattern symbolisation function

$$\begin{aligned} \mathrm{sym}^*\colon X^m &\to \mathbb{N}_0, \\ (x_1, x_2, \ldots, x_m) &\mapsto \sum_{i=1}^{m-1} \left(\frac{m!}{(m-i+1)!} \sum_{j=i+1}^{m} [x_i > x_j] \right) \end{aligned} \tag{9}$$

follows immediately. This function maps any m-tuple of values $(x_1, x_2, \ldots, x_m) \in X^m$ onto a distinct numerical representation of its ordinal pattern. As discussed in the beginning of the current section, this is highly advantageous in computational terms. Still, a minor drawback of Equation (9) is that factorial functions and integer division operations are involved, which are computationally expensive in general. As suggested in [21], this can be resolved by computing the weights

$$w_i = \frac{m!}{(m-i+1)!} \qquad \text{for} \qquad i \in \{1, 2, \ldots, m-1\}$$

only once in advance, and looking them up during the actual encoding. Alternatively, the encoding given by Equation (8) can be modified, as will be discussed in the following.

3.2. The Lehmer Code

Named in appreciation of Derrick Lehmer, the Lehmer code assigns a unique non-negative integer $n \in \{0, 1, \ldots, m! - 1\}$ to each permutation of a set of m elements. The mathematical foundations of this problem had already been studied in the 19th century [32], and Lehmer incorporated them into his work on algorithms for combinatorial computing, published as *Teaching Combinatorial Tricks to a Computer* [33] in 1960. Lehmer's approach towards enumerating permutations is conceptually similar to the solution that Keller and colleagues [17] proposed for the encoding of ordinal patterns. Given some permutation $(\lambda_1, \lambda_2, \ldots, \lambda_m)$, which could be the ranks of an ordinal pattern $\Pi_i \in \Omega_m$ without loss of generality, its corresponding set of right inversion counts $(r_1, r_2, \ldots, r_m)$ in terms of Equation (7) are obtained. Then, and by contrast with Equation (8), a unique integer representation of that permutation is calculated according to

$$n = \sum_{i=1}^{m} r_i \, (m-i)! \qquad \text{such that} \qquad n \in \{0, 1, \ldots, m! - 1\}. \tag{10}$$

In other words, the tuple of right inversion counts $(r_1, r_2, \ldots, r_m)$ is interpreted as an m-digit factoradic numeral of the form "$r_1 r_2 \cdots r_m$" [33]. In analogy with Section 3.1, but utilising the Lehmer code instead, the ordinal pattern of an m-tuple $(x_1, x_2, \ldots, x_m) \in X^m$ can therefore be extracted and encoded by computing the function

$$\begin{aligned} \operatorname{sym}\colon X^m &\to \mathbb{N}_0, \\ (x_1, x_2, \ldots, x_m) &\mapsto \sum_{i=1}^{m-1} \Big((m-i)! \sum_{j=i+1}^{m} \big[x_i > x_j \big] \Big). \end{aligned} \tag{11}$$

With a view to software implementation, Equation (11) still provides opportunity for optimisation. In its current form, the factorial $(m-i)!$ needs to be re-evaluated for each iteration of the outer sum. In general, calculating a (non-trivial) factorial requires a sequence of multiplications, but, due to the specific structure of Equation (11), these multiplications are avoidable here in their entirety. Taking advantage of the fundamental recurrence relation $k! = k(k-1)!$, the value of $\operatorname{sym}(x_1, x_2, \ldots, x_m)$ can be computed recursively by initialising $n_0 = 0$, and successively iterating

$$n_i = (m-i)(n_{i-1} + r_i) \qquad \text{with} \qquad r_i = \sum_{j=i+1}^{m} \big[x_i > x_j \big]. \tag{12}$$

The recursion terminates after iteration $i = m-1$, and yields $n_{m-1} = \operatorname{sym}(x_1, x_2, \ldots, x_m)$ as the result. The arithmetical equivalence with Equation (11) can be proven by mathematical induction, and is also evident from the following example.

Example 1. *Let $(r_1, r_2, \ldots, r_6) \in R_6$ denote the right inversion counts to the ordinal pattern of a given tuple $(x_1, x_2, \ldots, x_6)$. According to Equation (11), the numerical representation of this ordinal pattern of order $m = 6$ is obtained by computing*

$$\begin{aligned} \mathrm{sym}(x_1, x_2, \ldots, x_6) &= 5 \times 4 \times 3 \times 2 \times r_1 \\ &+ 4 \times 3 \times 2 \times r_2 \\ &+ 3 \times 2 \times r_3 \\ &+ 2 \times r_4 \\ &+ r_5 \,. \end{aligned}$$

Iterating the recurrence relation given by Equation (12) for the same right inversion counts $(r_1, r_2, \ldots, r_6)$ and the same pattern order $m = 6$, we obtain

$$n_5 = \mathrm{sym}(x_1, x_2, \ldots, x_6) = (((r_1 \times 5 + r_2) \times 4 + r_3) \times 3 + r_4) \times 2 + r_5.$$

Not only are both solutions arithmetically identical, but the recursive approach also requires considerably fewer multiplications—quod erat illustrandum.

The recursion given by Equation (12) thus enables a remarkably simple algorithm for extracting and storing ordinal patterns in computer memory, as is demonstrated by the pseudocode of the following Algorithm 1.

Algorithm 1. Given an m-tuple $(x_1, x_2, \ldots, x_m) \in X^m$ of elements from a totally ordered set X, a distinct numerical representation $n \in \{0, 1, \ldots, m! - 1\}$ of its ordinal pattern of order m can be obtained as outlined by the following pseudocode.

```
function encode_pattern
input
        $(x_1, x_2, \ldots, x_m) \in X^m$ with $m \in \{2, 3, \ldots\}$
output
        $n \in \{0, 1, \ldots, m! - 1\}$
begin
        $n \leftarrow 0$
        for $i \leftarrow 1$ to $m - 1$ do
                for $j \leftarrow i + 1$ to $m$ do
                        $n \leftarrow n + [x_i > x_j]$
                end
                $n \leftarrow (m - i)\, n$
        end
        return n
end.
```

The computational complexity of Algorithm 1 in dependence of the pattern order m is $O(m^2)$, as is further substantiated in Section 5.1.

Compared to the encoding scheme originally proposed in [17] and summarised in Section 3.1, adopting the Lehmer code allows for algorithms that can be implemented without relying on either factorial functions and division operations, or on lookup tables. Moreover, the encoding resulting from Equation (11) provides for an intuitive enumeration of ordinal patterns: figuratively speaking, the Lehmer code preserves the lexicographic sorting order of the permutations it encodes [33]. When applied to ordinal patterns, their tuples of ranks, tuples of inversion counts, as well as the resulting numerical codes are all consistently sorted. The reader may find Example 2 instructive in this respect.

Example 2. *Consider the $m! = 24$ ordinal patterns $(\lambda_1, \lambda_2, \lambda_3, \lambda_4) \in \Omega_4$ of order $m = 4$. Each of them has a distinct tuple of right inversion counts (r_1, r_2, r_3), and its corresponding numerical representation n is obtained by interpreting (r_1, r_2, r_3) as the digits of a factoradic numeral, in particular, $n = 6r_1 + 2r_2 + r_3$. As shown in Table 1, the tuples of ranks, tuples of inversion counts, and numerical codes all obey the same lexicographic sorting order.*

Table 1. The ranks $(\lambda_1, \lambda_2, \lambda_3, \lambda_4)$, right inversion counts (r_1, r_2, r_3), and numerical representations n for the $m! = 24$ ordinal patterns of order $m = 4$.

Ranks				**Inversions**			**Code**	**Ranks**				**Inversions**			**Code**
λ_1	λ_2	λ_3	λ_4	r_1	r_2	r_3	n	λ_1	λ_2	λ_3	λ_4	r_1	r_2	r_3	n
1	2	3	4	0	0	0	0	3	1	2	4	2	0	0	12
1	2	4	3	0	0	1	1	3	1	4	2	2	0	1	13
1	3	2	4	0	1	0	2	3	2	1	4	2	1	0	14
1	3	4	2	0	1	1	3	3	2	4	1	2	1	1	15
1	4	2	3	0	2	0	4	3	4	1	2	2	2	0	16
1	4	3	2	0	2	1	5	3	4	2	1	2	2	1	17
2	1	3	4	1	0	0	6	4	1	2	3	3	0	0	18
2	1	4	3	1	0	1	7	4	1	3	2	3	0	1	19
2	3	1	4	1	1	0	8	4	2	1	3	3	1	0	20
2	3	4	1	1	1	1	9	4	2	3	1	3	1	1	21
2	4	1	3	1	2	0	10	4	3	1	2	3	2	0	22
2	4	3	1	1	2	1	11	4	3	2	1	3	2	1	23

4. Encoding Time Series Data

Any form of ordinal pattern-based time series analysis requires that sequences of elements from a totally ordered set X (usually, the real numbers $\mathbb{R}$) be converted into sequences of ordinal patterns. Given a finite time series $\{x_t\}$, with $x_t \in X$ and $t \in \{1, 2, \ldots, N\}$, we select a pattern order $m \geqslant 2$ and time lag $\tau \geqslant 1$, and subsequently obtain $\{\Pi_t\}$, where

$$\Pi_t = \mathrm{op}(x_t, x_{t+\tau}, \ldots, x_{t+(m-1)\tau}) \qquad \text{for all} \qquad t \in \{1, 2, \ldots, N-(m-1)\tau\}. \tag{13}$$

In doing so, we assign to each ordinal pattern Π_t the time index t of the leftmost of its underlying tuple elements. Consequently, the last $(m-1)\tau$ indices of the time series $\{x_t\}$ do not reference any ordinal pattern, and the resulting pattern sequence $\{\Pi_t\}$ thus comprises of $N-(m-1)\tau$ elements. To perform this transformation in software, the encoding approach described in the previous section can directly be utilised.

4.1. The Straightforward Approach (Plain Algorithm)

Algorithm 1 maps any given m-tuple $(x_1, x_2, \ldots, x_m) \in X^m$ of elements from a totally ordered set X onto a non-negative integer $n \in \{0, 1, \ldots, m!-1\}$, such that the value

$$n \;=\; \mathrm{enc}(\mathrm{op}(x_1, x_2, \ldots, x_m)) \;=\; \mathrm{sym}(x_1, x_2, \ldots, x_m)$$

uniquely identifies the ordinal pattern $\mathrm{op}(x_1, x_2, \ldots, x_m) = (\lambda_1, \lambda_2, \ldots, \lambda_m) \in \Omega_m$ of the m-tuple. This encoding strategy is easily expanded, so as to turn an entire time series $\{x_t\}$ into numerical representations $\{n_t\}$ of its ordinal pattern sequence $\{\Pi_t\}$, whereby

$$n_t = \mathrm{enc}(\Pi_t) = \mathrm{sym}(x_t, x_{t+\tau}, \ldots, x_{t+(m-1)\tau}) \qquad \text{for all} \qquad t \in \{1, 2, \ldots, N-(m-1)\tau\}. \tag{14}$$

The extension of Algorithm 1 merely requires an additional loop and proper indexing. It results in the following algorithm.

In analogy with Algorithm 1, the computational complexity of Algorithm 2 is $O(m^2)$ with regard to the pattern order m. It is $O(N)$ in dependence of the sequence length N, and $O(1)$ for any choice of the time lag τ (also see Sections 5.1 and 5.8).

Algorithm 2. Plain Algorithm. To transform a finite time series $\{x_t\}$ of elements from a totally ordered set X into a sequence of non-negative integers $\{n_t\}$, select a pattern order $m \geqslant 2$ and a time lag $\tau \geqslant 1$, and proceed according to the pseudocode below. The value $n_t \in \{0, 1, \ldots, m! - 1\}$ then uniquely identifies the ordinal pattern Π_t of order m, which is extracted from the time series $\{x_t\}$ at time index t using the time lag τ. The function `encode_pattern` is specified in Algorithm 1.

```
function encode_sequence
input
        $m \in \mathbb{N}$ with $m \geqslant 2$
        $\tau \in \mathbb{N}$ with $\tau \geqslant 1$
        $\{x_t\}$ with $x_t \in X$ and $t \in \{1, 2, \ldots, N\}$
output
        $\{n_t\}$ with $n_t \in \{0, 1, \ldots, m! - 1\}$ and $t \in \{1, 2, \ldots, N - (m-1)\tau\}$
begin
        for $t \leftarrow 1$ to $N - (m-1)\tau$ do
                $n_t \leftarrow$ encode_pattern$(x_t, x_{t+\tau}, \ldots, x_{t+(m-1)\tau})$
        end
end.
```

4.2. An Optimised Encoding Strategy (Overlap Algorithm)

Algorithm 2 still contains some redundant operations, and does therefore not scale too well with the pattern order m. This aspect can be targeted by further optimisation. Bandt and Pompe had already hinted at this possibility in their seminal publication on ordinal patterns, suggesting there was "an extremely fast algorithm where each pair of values need to be compared only once" [1]. Keller, Sinn, and Emonds further elaborated on this matter, and demonstrated that the overlap property described in Section 2.2 can be exploited computationally [17]. The algorithm described in the following builds upon the same fundamental idea, but additionally uses the recursive Lehmer encoding proposed in Section 3.2.

Written out in its entirety, Algorithm 2 converts a time series $\{x_t\}$, indexed by $t \in \{1, 2, \ldots, N\}$, into a sequence of non-negative integers $\{n_t\}$, such that

$$n_t = \sum_{i=1}^{m-1} \left((m-i)! \sum_{j=i+1}^{m} \left[x_{t+(i-1)\tau} > x_{t+(j-1)\tau} \right] \right) \qquad \text{for all} \qquad t \in \{1, 2, \ldots, N - (m-1)\tau\}. \tag{15}$$

As derived in Section 3.2, each evaluation of the inner sum of Equation (15) yields one of the $m-1$ non-trivial right inversion counts to the ordinal pattern Π_t. The encoding can hence be reformulated in terms of

$$n_t = \sum_{i=1}^{m-1} (m-i)! \times r_{t,i} \qquad \text{where} \qquad r_{t,i} = \sum_{j=i+1}^{m} \left[x_{t+(i-1)\tau} > x_{t+(j-1)\tau} \right]. \tag{16}$$

Now recall from Section 2.3 that any two ordinal patterns $\Pi_{t-\tau}$ and Π_t at a distance equalling the time lag τ share all but one of their underlying time series values. Consequently, the inversion counts to the patterns $\Pi_{t-\tau}$ and Π_t are strongly interrelated as well. In particular, it is easily confirmed that

$$n_t = \sum_{i=1}^{m-1} (m-i)! \times r_{t,i} \qquad \text{where} \qquad r_{t,i} = \left[x_{t+(i-1)\tau} > x_{t+(m-1)\tau} \right] + r_{t-\tau,i+1}. \tag{17}$$

This recurrence relation is highly advantageous in computational terms. Let us assume that, in the overall process of encoding a time series $\{x_t\}$, the inversion counts to the pattern $\Pi_{t-\tau}$ have just been determined. If those are kept in memory for τ more iterations, then encoding the pattern Π_t merely requires $m-1$ additional comparisons. In turn, each such comparison yields one of the inversion counts to the pattern Π_t, which can then be reused to efficiently encode $\Pi_{t+\tau}$ another τ time steps ahead. Merely the first τ ordinal patterns $\{\Pi_1, \Pi_2, \ldots, \Pi_\tau\}$ require additional consideration because obviously, none of them has a neighbour located τ time steps earlier. While $\{\Pi_{1-\tau}, \Pi_{2-\tau}, \ldots, \Pi_0\}$ are hence undefined, all but the leftmost of their respective inversion counts still formally exist. More precisely,

$$\mathbf{R}_{\text{init}} = \begin{pmatrix} r_{1-\tau,2} & r_{1-\tau,3} & \cdots & r_{1-\tau,m-1} \\ r_{2-\tau,2} & r_{2-\tau,3} & \cdots & r_{2-\tau,m-1} \\ \vdots & \vdots & \ddots & \vdots \\ r_{0,2} & r_{0,3} & \cdots & r_{0,m-1} \end{pmatrix} \tag{18}$$

are all well-defined, and suffice to calculate $\{n_1, n_2, \ldots, n_\tau\}$ in terms of Equation (17). The procedure of encoding a time series thus comprises of two stages: (1) obtain the initial inversion counts $\mathbf{R}_{\text{init}}$, then (2) iterate Equation (17) for all t, starting from $t = 1$. Each such iteration yields an encoded pattern n_t, as well as a corresponding set of inversion counts. The latter are temporarily buffered in memory, reused τ iterations later, and then become obsolete. The entire process is summarised in Algorithm 3.

Algorithm 3. Overlap Algorithm. To transform a finite time series $\{x_t\}$ of elements $x_t \in X$ into a sequence of non-negative integers $\{n_t\}$ representing the ordinal patterns $\{\Pi_t\}$ of the time series, select a pattern order $m \geqslant 2$ and time lag $\tau \geqslant 1$. Then, proceed as follows.

```
function encode_sequence
input
        $\{x_t\}$   with $x_t \in X$ and $t \in \{1,2,\ldots,N\}$
        $m \in \mathbb{N}$ with $m \geqslant 2$
        $\tau \in \mathbb{N}$ with $\tau \geqslant 1$
output
        $\{n_t\}$   with $n_t \in \{0,1,\ldots,m!-1\}$ and $t \in \{1,2,\ldots,N-(m-1)\tau\}$
locals
        $\{r_{i,j}\}$   with $i \in \{1,2,\ldots,\tau\}$ and $j \in \{1,2,\ldots,m\}$
begin
        $r_{i,j} \leftarrow 0$ for all $i \in \{1,2,\ldots,\tau\}$ and all $j \in \{1,2,\ldots,m\}$

        /* Obtain initial right inversion counts $\mathbf{R}_{\text{init}}$ */
        for $t \leftarrow 1$ to $\tau$ do
                for $i \leftarrow 1$ to $m-2$ do
                        for $j \leftarrow i+1$ to $m-1$ do
                                $r_{t,i+1} \leftarrow r_{t,i+1} + [x_{t+(i-1)\tau} > x_{t+(j-1)\tau}]$
                        end
                end
        end

        /* Extract and encode ordinal patterns */
        $i \leftarrow 1$
        for $t \leftarrow 1$ to $N-(m-1)\tau$ do
                for $j \leftarrow 1$ to $m-1$ do
                        $r_{i,j} \leftarrow r_{i,j+1} + [x_{t+(j-1)\tau} > x_{t+(m-1)\tau}]$
                        $n_t \leftarrow (m-j)(n_t + r_{i,j})$
                end
                $i \leftarrow (i \bmod \tau) + 1$ /* Increment circular buffer index */
        end
end.
```

Notice that we favour readability over efficiency in the pseudocode of Algorithm 3. Actually, a smaller buffer $\{r_{i,j}\}$ with $i \in \{1, \ldots, \tau\}$ and $j \in \{2, \ldots, m-1\}$ will suffice for an actual implementation because the recurrence relation does not depend on $r_{i,1}$ at all, whereas $r_{i,m} = 0$ always. In addition, the modulo operation used for circular buffer indexing may impose a considerable run-time penalty. Both aspects are addressed in the reference implementation of Algorithm 3, which is provided by the `ordpat_encode_overlap` function in the supplementary file `ordpat.c`.

Independent of such details of implementation, and let aside the initialisation of $\mathbf{R}_{\text{init}}$, the asymptotic computational complexity of Algorithm 3 is $O(N)$ for the sequence length N, and a constant $O(1)$ for the time lag parameter τ. By contrast with Algorithm 2, however, Algorithm 3 offers a complexity of $O(m)$ with regard to the pattern order m (see Section 5.1 for details).

4.3. The Unakafova–Keller Approach (Lookup Algorithm)

Algorithm 3 is based on the fact that two ordinal patterns $\Pi_{t-\tau}$ and Π_t overlap in all but one of their underlying time series values. Utilising the same interrelation, Valentina Unakafova and Karsten Keller proposed [21] a different encoding strategy that (by contrast with our Algorithm 3, as well as their own previous work [17]) does not depend on buffering inversion counts. The approach is compellingly simple: as described in Section 2.3, an ordinal pattern $\Pi_{t-\tau}$ of order m can only have m different succeeding patterns Π_t at τ time steps distance. Consequently, if the ordinal pattern $\Pi_{t-\tau} = \operatorname{op}(x_{t-\tau}, x_t, \ldots, x_{t+(m-2)\tau})$ is known in advance, then the value of the expression

$$\lambda_{t,m} = m - \sum_{i=1}^{m-1} \left[x_{t+(i-1)\tau} > x_{t+(m-1)\tau} \right] \tag{19}$$

uniquely determines the pattern $\Pi_t = \operatorname{op}(x_t, x_{t+\tau}, \ldots, x_{t+(m-1)\tau})$. In connection with Definition 1, the decisive variable $\lambda_{t,m} \in \{1, 2, \ldots, m\}$ is easily identified as the rightmost rank of the ordinal pattern $\Pi_t = (\lambda_{t,1}, \lambda_{t,2}, \ldots, \lambda_{t,m})$. As is visualised in Figure 2, each value that $\lambda_{t,m}$ can take on represents one of the m different ordinal patterns Π_t that may possibly follow after a particular pattern $\Pi_{t-\tau}$.

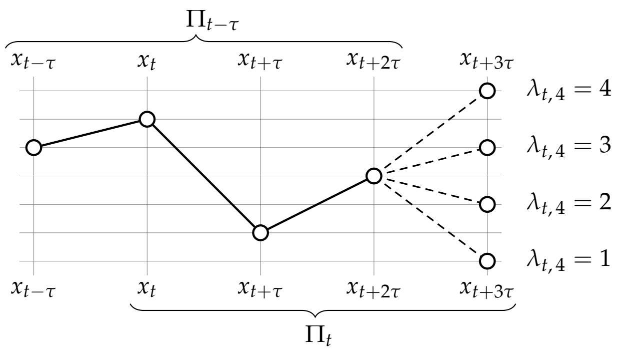

Figure 2. Assume pattern order $m = 4$, without loss of generality. For any fixed ordinal pattern $\Pi_{t-\tau}$, its succeeding pattern $\Pi_t = (\lambda_{t,1}, \lambda_{t,2}, \lambda_{t,3}, \lambda_{t,4})$ at τ time steps distance has merely one degree of freedom: its rightmost rank $\lambda_{t,4}$.

These considerations essentially imply that a surjective map $(\Pi_{t-\tau}, \lambda_{t,m}) \mapsto \Pi_t$ exists for any pattern order m and time lag τ. Likewise, there has to be a well-defined surjection

$$\begin{gathered} \mathbb{N}_0 \times \mathbb{N} \to \mathbb{N}_0, \\ (n_{t-\tau}, \lambda_{t,m}) \mapsto n_t \end{gathered} \tag{20}$$

for ordinal patterns represented numerically. Note that the particular encoding function $\operatorname{enc}\colon \Omega_m \to \mathbb{N}_0$ in terms of Equation (6) does not make a difference in this regard—as long as it is bijective, such that each ordinal pattern is assigned a unique numerical label. In their original publication [21], Unakafova and Keller used the encoding given by Equation (9). To implement Equation (20), the authors rely on a lookup table holding $m! \times m$ entries, in particular

$$\mathbf{L}_m = \begin{pmatrix} L_{1,1} & \cdots & L_{1,m} \\ \vdots & \ddots & \vdots \\ L_{m!,1} & \cdots & L_{m!,m} \end{pmatrix} \quad \text{such that} \quad n_t = L_{i,j} \quad \text{for} \quad i = n_{t-\tau} + 1 \quad \text{and} \quad j = \lambda_{t,n}. \tag{21}$$

Encoding a time series $\{x_t\}$, with $t \in \{1, 2, \ldots, N\}$, is then a matter of computing the numerical codes $\{n_1, n_2, \ldots, n_\tau\}$ for the first τ patterns by direct evaluation, and subsequently iterating Equation (20) to obtain the remaining symbols $\{n_{\tau+1}, n_{\tau+2}, \ldots, n_{N-(m-1)\tau}\}$. Algorithm 4 describes the process in full detail.

Algorithm 4. Lookup Algorithm. To transform a finite time series $\{x_t\}$ into a sequence of non-negative integers $\{n_t\}$ representing its ordinal patterns, select a pattern order $m \geqslant 2$ and time lag $\tau \geqslant 1$. In addition, prepare a lookup table $\{L_{i,j}\}$ according to Equation (21) that matches the `encode_pattern` function to be used. This can be the sym-function given by Equation (11), the sym*-function of Equation (9), or any other bijective map yielding $n_t \in \{0, 1, \ldots, m! - 1\}$. Then, proceed as follows.

```
function encode_sequence
input
          $\{x_t\}$  with $x_t \in X$ and $t \in \{1, 2, \ldots, N\}$
          $m \in \mathbb{N}$ with $m \geqslant 2$
          $\tau \in \mathbb{N}$ with $\tau \geqslant 1$
output
          $\{n_t\}$  with $n_t \in \{0, 1, \ldots, m! - 1\}$ and $t \in \{1, 2, \ldots, N - (m-1)\tau\}$
locals
          $\{L_{i,j}\}$ with $L_{i,j} \in \{0, 1, \ldots, m! - 1\}$ and $i \in \{1, 2, \ldots, m!\}$ and $j \in \{1, 2, \ldots, m\}$
begin
          $\{L_{i,j}\} \leftarrow$ load_lookup_table($m$)

          /* Encode first $\tau$ ordinal patterns */
          for $t \leftarrow 1$ to $\tau$ do
                    $n_t \leftarrow$ encode_pattern($x_t, x_{t+\tau}, \ldots, x_{t+(m-1)\tau}$)
          end

          /* Encode all remaining patterns */
          for $t \leftarrow \tau + 1$ to $N - (m-1)\tau$ do
                    $row \leftarrow n_{t-\tau} + 1$
                    $col \leftarrow 1$
                    for $i \leftarrow 1$ to $m - 1$ do
                              $col \leftarrow col + [x_{t+(i-1)\tau} > x_{t+(m-1)\tau}]$
                    end
                    $n_t \leftarrow L_{row,col}$
          end
end.
```

In terms of computational complexity, the asymptotic behaviour of Algorithm 4 is identical to that of Algorithm 3. Putting aside the initialisation steps, its computational complexity is $O(N)$ for the sequence length N, $O(m)$ for the pattern order m, and $O(1)$ with regard to the time lag τ. In practice, however, the run-time properties of the two algorithms can differ substantially. On the one hand, Algorithm 4 requires significantly fewer computational operations per ordinal pattern (see Table 2), as is the very purpose of using a lookup table. On the other hand, this reduction in computational complexity does come at a price—as will be further discussed in Section 5.7.

5. Implementation and Run-Time Performance

Three encoding algorithms of varying complexity have been presented in the previous section. Those will henceforth also be referred to by the plain algorithm (Algorithm 2), the overlap algorithm (Algorithm 3), and the lookup algorithm (Algorithm 4). Each algorithm comes with strengths and weaknesses, and raises different implementational challenges. Thus, the following section is intended to guide the reader in selecting and implementing the most appropriate algorithm for a particular execution platform and task. In particular, we will consider GNU Octave [34], Matlab (The Mathworks, Natick, MA, USA), NumPy/Python [35], and the C programming language.

5.1. Theoretical Computational Complexities

The computational complexities of the plain, overlap, and lookup algorithm (Algorithms 2–4) differ substantially in dependence of the pattern order m. This can be derived from the basic operation counts provided in Table 2.

Table 2. Number of operations required to encode a single ordinal pattern of order m, using either the plain algorithm (Algorithm 2), the overlap algorithm (Algorithm 3), or the lookup algorithm (Algorithm 4). Early initialisation operations have not been considered.

Algorithm	Add	Multiply	Compare	Increment	Assign	Modulo	Total
plain	$\frac{m^2+3m-2}{2}$	$m-1$	m^2-1	$\frac{m^2+m-2}{2}$	$\frac{m^2+3m}{2}$	0	$\frac{5m^2+9m-8}{2}$
overlap	$9m-8$	$6m-6$	$2m-2$	m	$2m$	1	$20m-15$
lookup [21]	$6m-3$	$2m-1$	$2m-2$	$m-1$	$m+3$	0	$12m-4$

Following from the total operation counts given in Table 2, the plain algorithm has an asymptotic computational complexity of $O(m^2)$. By contrast, the overlap and lookup algorithm both scale linearly with the pattern order, that is, their complexity is $O(m)$. However, the lookup algorithm avoids many of the computations that the overlap algorithm has to perform. Therefore, the theoretical computational complexity of the three algorithms decreases from the plain algorithm to the overlap algorithm, and again from the overlap algorithm to the lookup algorithm.

In practice, however, the way that an abstract algorithm is translated into actual software can make a big difference for its run-time efficiency. Taking into account application-dependent requirements and platform-specific peculiarities is essential in this regard. Therefore, the rest of this section focusses on those practical aspects of implementing the three encoding algorithms.

5.2. Memory Alignment

The algorithms described in Section 4 represent the ordinal patterns $\Omega_m = \{\Pi_1, \Pi_2, \dots, \Pi_{m!}\}$ by distinct integers $\{0, 1, \dots, m! - 1\}$, thereby providing a bijective map $\Pi_i \mapsto i - 1$ as stipulated by the enc-function of Equation (6). The resulting symbols are highly memory-efficient, theoretically requiring a mere $\log_2 m!$ bit per pattern. Note that $\log_2 m!$ also is the entropy of a uniform distribution of $m!$ elements, and, thus, the maximum entropy an ordinal pattern distribution of order m can possibly attain. Putting aside data compression techniques, no other encoding can be more compact (as is assured by Shannon's source coding theorem [2,3]).

In practice, it makes sense to align ordinal patterns to byte boundaries, which can be accomplished by dedicating an integer power of 2 (but at least 8) bits to each ordinal pattern, such that the resulting bit width per pattern is

$$w_\mathrm{b} = 2^k \geqslant \log_2 m! \qquad \text{where} \qquad k \in \{3, 4, \dots\}. \tag{22}$$

Any digital processor equipped with 64-bit integer registers will therefore handle ordinal patterns of order $m \leqslant 20$ natively—that is, at hardware efficiency. For reference, Table 3 lists the maximum pattern orders that fit the primitive data types available on standard computer systems.

Table 3. Maximum pattern orders representable by standard integer and floating point data types.

Data Type	Significant Bits w_b	Maximum Order m
`uint8`	8	5
`uint16`	16	8
`uint32`	32	12
`uint64`	64	20
`binary32` (`single`)	24	10
`binary64` (`double`)	53	18

Although ordinal patterns and their numerical representations are intrinsically integral, Table 3 also references two IEEE 754 floating point formats [36,37]. Those were included because computation environments like NumPy/Python, GNU Octave, and Matlab by default use the `binary64` floating point data format. Previously known as `double` [36], this format features an effective mantissa length of 53 bit [37], and can therefore represent a total of 2^{53} distinct non-negative values at integer precision [38]. If this limit is exceeded, mantissa truncation will silently cause unexpected results (like the erroneous $2^{53} = 2^{53} + 1$), and distinct patterns will falsely be labelled as identical. When working with patterns of order $m > 18$ in such computation environments, this bug-inviting peculiarity must be kept in mind.

Independent of the software platform used, ordinal patterns of order $m > 20$ require additional thought because current general-purpose processors do not provide native support for integers wider than 64 bits. Therefore, each pattern of order $m > 20$ has to be stored as an array of integers, and all arithmetical and logical operations need to be emulated in software. Those matters will be further discussed in Section 5.6.

5.3. Run-Time Test Environment

All performance testing was done on a conventional x86-64 laptop computer, equipped with an Intel Core i7-5600U processor (Intel Corporation, Santa Clara, CA, USA) and 8 GB of random access memory (RAM). An Arch Linux distribution of the GNU/Linux operating system was used, running the default kernel (`linux`, 5.2.arch2-1) and C standard library (`glibc`, 2.29-3). Pre-built binary packages of GNU Octave (`octave`, 5.1.0-4), Python 3 (`python`, 3.7.3-1), NumPy (`python-numpy`, 1.16.4-1), and FFTW (`fftw, 3.3.8-1`), as well as their respective dependencies were installed from the official repositories of the distributor. The Linux version of the Matlab 2018b release was used. All source code written in the C programming language was compiled using the GNU Compiler Collection (`gcc`, 8.3.0-2). The parameters `-march=x86-64 -mtune=generic -O3` were selected to allow for heavy compiler optimisation, while not relying on any model-specific features of the targeted processor.

5.4. Test Signal Generation

Based on the fact that ordinal patterns of any order are uniformly distributed in white noise [18], sequences of independent and uniformly distributed pseudo-random numbers were used to test the performance of the algorithms. This choice ascertains that all ordinal pattern transitions (see Figure 1) appear at the same relative frequency, such that each possible path of execution is taken equally often. In addition, to test for a possible dependency between the run-time of the algorithms and the ordinal complexity of the input signal, low-pass filtered (and thus, self-correlated) noise of various bandwidths was incorporated into the test procedure where appropriate. Filtering was performed by zeroing bins in the Discrete Fourier Transform (DFT) of the signal.

To maintain reproducible test signals across all software environments, the xorshift random number generator by George Marsaglia [39] was used with a word size of 32 bits and the standard shift parameters $(13, 17, 5)$. In this configuration, xorshifting produces a pseudo-random sequence of period length $2^{32} - 1$, and elements drawn from $\{1, 2, \ldots, 2^{32} - 1\}$. Normalisation (as in zero-mean or unit-variance) was omitted, considering that ordinal patterns are invariant to order-preserving transformations anyway [1]. However, the integer-valued test signals were stored in `binary64` floating point representation, as this is the expected input format in ordinal pattern analysis.

5.5. The Plain Algorithm (Algorithm 2)

The plain algorithm is the simplest among the three encoding strategies considered in this manuscript. As it makes no effort to avoid redundant operations, it may at first glance seem generally inferior to the more sophisticated overlap and lookup algorithms (Algorithms 3 and 4). Quite the contrary, the plain algorithm may actually be preferable when numerical scripting languages like GNU Octave, Matlab or NumPy/Python shall be used to encode ordinal patterns. The reason is that, by contrast with the recursive Algorithms 3 and 4, the plain algorithm allows for a vectorised implementation that avoids loops.

Compared to pre-compiled languages like C, scripting languages are relatively slow at iterating loops. This clearly manifests if we implement the plain algorithm in a straightforward manner, which then requires three levels of nested loops. See the `encode_plain` functions in the supplementary files `encode_plain.m` and `ordpat.py`, as well as the `ordpat_encode_plain` function in `ordpat.c`, respectively. As shown in Table 4, the run-time efficiency varies by orders of magnitude across different execution environments.

Table 4. Computation time (median of 20 trials) for turning 3.6×10^5 samples of uniform white noise into a sequence of ordinal patterns of order m, using the time lag $\tau = 1$. Straightforward iterative implementations of Algorithm 2 were tested in various computation environments.

Order	**Computation Time** (ms)			
m	**GNU Octave**	**NumPy/Python**	**Matlab**	**C**
2	7.9×10^3	2.4×10^3	1.1×10^1	7.8×10^{-1}
3	2.0×10^4	5.7×10^3	2.2×10^1	1.6×10^0
4	3.6×10^4	1.0×10^4	3.5×10^1	2.8×10^0
5	5.6×10^4	1.5×10^4	5.5×10^1	4.3×10^0
6	8.1×10^4	2.1×10^4	8.5×10^1	6.0×10^0
7	1.1×10^5	2.9×10^4	1.2×10^2	8.0×10^0
8	1.4×10^5	3.7×10^4	1.6×10^2	1.0×10^1
9	1.8×10^5	4.6×10^4	2.0×10^2	1.3×10^1

Those differences are due to the iterative nature of the plain algorithm, which forces the Octave and Python language interpreters to translate the same sequence of instructions over and over again, for each and every loop iteration. Consistently, the Matlab just-in-time compiler performs better, but is in turn outperformed by the machine code of the fully-optimising C compiler. To mitigate the performance penalty inherent to numerical scripting languages, a programming technique known as vectorisation can be applied in many cases. In a nutshell, vectorisation is about avoiding element-wise operations in favour of high-level instructions acting on blocks of data, like vectors (hence the name), matrices, and tensors. Vectorising the plain algorithm is a bit tricky, but the performance gain is well worth the effort. The approach is best explained by means of a practical example. We will be using Matlab code here, but the concepts translate to other programming environments as well. Recall from Equation (11) that the map

$$(x_1, x_2, \ldots, x_m) \mapsto \sum_{i=1}^{m-1} \Big((m-i)! \sum_{j=i+1}^{m} [x_i > x_j] \Big) \tag{23}$$

is the basis of the plain algorithm, and yields the ordinal pattern of the m-tuple $(x_1, x_2, \ldots, x_m)$ in its numerical representation. Now consider that, for any fixed pattern order m, the result of this function can be rewritten without relying on summation signs. Assuming the order $m = 5$, for example, the mathematical expression

$$\begin{aligned}
& 24 \times \big([x_1 > x_2] + [x_1 > x_3] + [x_1 > x_4] + [x_1 > x_5]\big) \\
+ \; & \;\,6 \times \big(\qquad\qquad\quad [x_2 > x_3] + [x_2 > x_4] + [x_2 > x_5]\big) \\
+ \; & \;\,2 \times \big(\qquad\qquad\qquad\qquad\quad\; [x_3 > x_4] + [x_3 > x_5]\big) \\
+ \; & \;\,1 \times \big(\qquad\qquad\qquad\qquad\qquad\qquad\qquad\; [x_4 > x_5]\big)
\end{aligned}$$

is admittedly more tedious, but arithmetically equivalent to the compact formulation in Equation (23). The point here is that this expression can directly be translated into a single Matlab instruction, namely

```
    24 * ( (x1 > x2) + (x1 > x3) + (x1 > x4) + (x1 > x5) ) ...
+    6 * (             (x2 > x3) + (x2 > x4) + (x2 > x5) ) ...
+    2 * (                         (x3 > x4) + (x3 > x5) ) ...
+    1 * (                                     (x4 > x5) );
```

Let us assume that the time series to be analysed is represented by a $N \times 1$ vector `input` on the Matlab workspace. Obviously, if we initialise the variables

```
x1 = input(1); x2 = input(2); x3 = input(3); x4 = input(4); x5 = input(5);
```

and call the above instruction, we obtain the ordinal pattern of the vector `input(1:5)` in its numerical representation. This is an optimisation technique called loop unrolling. Furthermore, consider that, in numerical scripting languages, most basic operations are not limited to scalar values, but can in principle handle arrays of arbitrary dimension as their operands. If we thus set `x1`, …, `x5` to the delay vectors

```
x1 = input(1:end-4);
x2 = input(2:end-3);
x3 = input(3:end-2);
x4 = input(4:end-1);
x5 = input(5:end-0);
```

instead, we can obtain from `input` its full sequence of ordinal patterns of order $m = 5$ and lag $\tau = 1$ by calling a single Matlab instruction. Arbitrary time lags $\tau \geqslant 1$ can in turn be realised by using the generalised delay vectors

```
x1 = input(1 + 0*lag : end - 4*lag);
x2 = input(1 + 1*lag : end - 3*lag);
x3 = input(1 + 2*lag : end - 2*lag);
x4 = input(1 + 3*lag : end - 1*lag);
x5 = input(1 + 4*lag : end - 0*lag);
```

in conjunction with the exact same Matlab expression. As a side note, the operation is also applicable to multidimensional data structures like matrices and tensors, such that multivariate time series can as well be encoded by means of a single invocation.

The downside of this approach is that each pattern order m requires a dedicated piece of code. When working with low pattern orders, implementation by hand may be acceptable because it still yields comprehensible code—as is demonstrated by the `symbolise.m` function provided in the supplements of [40]. For higher pattern orders, though, another convenient feature offered by scripting languages should rather be utilised. GNU Octave, Matlab and NumPy/Python all support self-modifying code, which is source code that can dynamically modify its own sequence of instructions at run-time. Supporting languages provide built-in functions like `eval` or `exec` for this purpose, which take a string as their input argument and hand it over to the execution engine for evaluation. Such language facilities can neatly be utilised to obtain an efficient, universal implementation of the plain algorithm: we just have to write a function that dynamically creates the appropriate vectorised code for a given pattern order m and time lag τ, then executes it. Consider the functions `encode_vectorised` in the supplementary files `encode_vectorised.m` and `ordpat.py` for reference. As can be seen from Table 5, the optimisation yields a tremendous increase in run-time efficiency.

Table 5. Computation time (median of 20 trials) for turning 3.6×10^5 samples of uniform white noise into a sequence of ordinal patterns of order m, using the time lag $\tau = 1$. Vectorised implementations of Algorithm 2 were tested in various computation environments. The results of Table 4 (relating to iterative implementations of Algorithm 2) were replicated for ease of comparison.

Order	**Computation Time** (ms)						
m	**GNU Octave**		**NumPy/Python**		**Matlab**		**C**
	loops	**vectors**	**loops**	**vectors**	**loops**	**vectors**	**loops**
2	7.9×10^3	1.5×10^0	2.4×10^3	7.9×10^{-1}	1.1×10^1	3.8×10^0	7.8×10^{-1}
3	2.0×10^4	6.5×10^0	5.7×10^3	1.3×10^0	2.2×10^1	9.0×10^0	1.6×10^0
4	3.6×10^4	1.4×10^1	1.0×10^4	2.0×10^0	3.5×10^1	1.2×10^1	2.8×10^0
5	5.6×10^4	2.0×10^1	1.5×10^4	2.9×10^0	5.5×10^1	1.5×10^1	4.3×10^0
6	8.1×10^4	3.3×10^1	2.1×10^4	5.3×10^0	8.5×10^1	1.8×10^1	6.0×10^0
7	1.1×10^5	4.2×10^1	2.9×10^4	6.8×10^0	1.2×10^2	2.3×10^1	8.0×10^0
8	1.4×10^5	5.7×10^1	3.7×10^4	8.5×10^0	1.6×10^2	2.7×10^1	1.0×10^1
9	1.8×10^5	6.8×10^1	4.6×10^4	1.3×10^1	2.0×10^2	3.2×10^1	1.3×10^1

For any of the numerical computation environments considered, using a vectorised version of the plain algorithm allows for encoding millions of ordinal patterns in milliseconds, without having to rely on pre-compiled external libraries. Most noteworthy, the vectorised NumPy/Python implementation actually outperformed the pre-compiled C code in the majority of cases. This hints at the amount of sophistication put into the development of the free and open source NumPy/Python framework.

5.6. The Overlap Algorithm (Algorithm 3)

The prerequisite for vectorising an algorithm is that all input data be available in advance, such that they can be passed to the software in parallel. Therefore, and by contrast with the plain algorithm considered above, recursive solutions like the overlap algorithm (Algorithm 3) cannot be fully vectorised, but inevitably require some sort of iteration. Due to the reasons given in Section 5.5, implementing the overlap algorithm in a numerical scripting language thus defeats its very purpose, which is the efficient evaluation of Equation (15). This can be demonstrated by benchmarking a Matlab implementation of the overlap algorithm against a vectorised Matlab implementation of the plain algorithm (Algorithm 2). The code for both implementations is provided in the supplements. While the overlap algorithm should be superior in theory, a vectorised implementation of the plain algorithm can actually be faster under practical conditions, as can be observed in Figure 3.

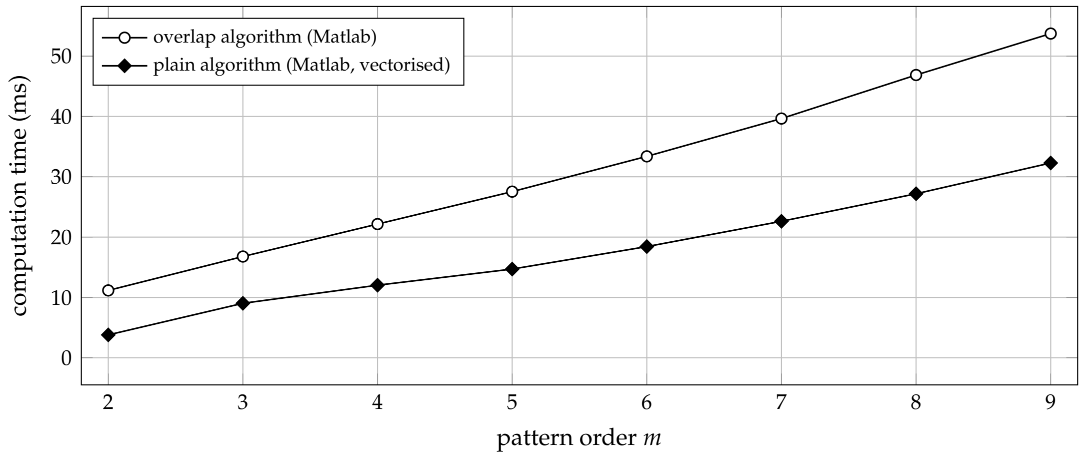

Figure 3. Computation time (median of 20 trials) for transforming 3.6×10^5 samples of uniform white noise into a sequence of ordinal patterns of order m. The lag was set to a constant $\tau = 1$, and the Matlab functions `encode_vectorised` and `encode_overlap` from the supplementary files `encode_vectorised.m` and `encode_overlap.m` were used for the simulation.

To benefit from the overlap algorithm in terms of efficiency, it is therefore highly advisable to use a compiled programming language instead. In the following, we shall exclusively be concerned with implementing the overlap algorithm in the C programming language. For pattern orders up to $m = 20$, the pseudocode of Algorithm 3 can directly be translated into C code (assuming 64-bit integer support on the targeted platform). Admittedly, a few minor tweaks are still possible, but those are easily understood from the reference implementation, that is, from the `ordpat_encode_overlap` function provided in the supplementary `ordpat.c` file. Typical run-time performances achieved by C implementations of the plain versus the overlap algorithm are depicted in Figure 4.

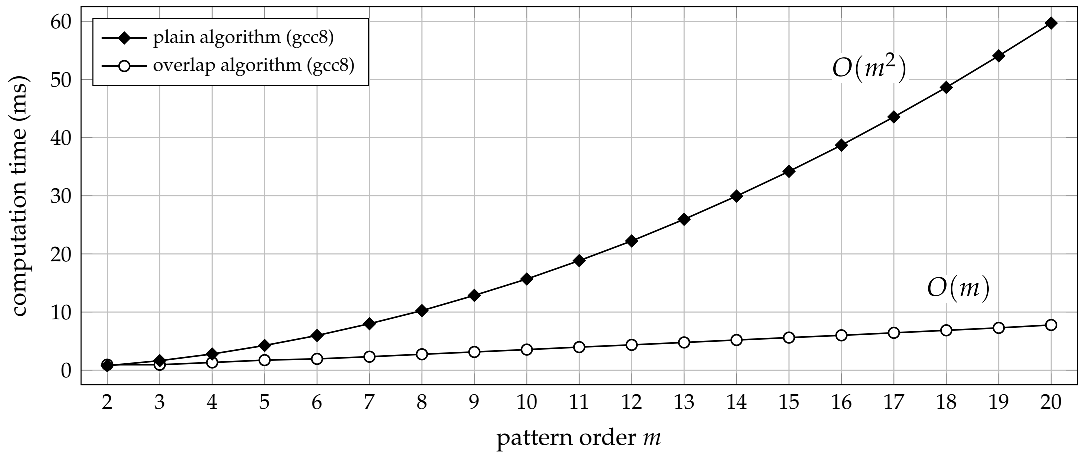

Figure 4. Computation time (median of 20 trials) for transforming 3.6×10^5 samples of uniform white noise into a sequence of ordinal patterns of order m. The time lag was set to $\tau = 1$, and the C functions `ordpat_encode_plain` and `ordpat_encode_overlap` from the supplementary file `ordpat.c` were used for the simulation. The run-time complexity of the plain algorithm is $O(m^2)$, whereas the overlap algorithm generally scales at $O(m)$. For $m = 2$, there is no advantage over the plain algorithm: all order relations are then disjoint, such that no overlap can be exploited (see Section 2.3).

The aforementioned limitation to pattern orders $m \leqslant 20$ results from the relation $20! < 2^{64} < 21!$, which implies that patterns beyond $m = 20$ cannot be represented by 64-bit integers (see Section 5.2). Reconsidering the pseudocode of Algorithm 3, it is easily confirmed that the only instruction actually affected by this limitation is

$$n_t \leftarrow (m - j)(n_t + r_{i,j}) \tag{24}$$

in line 27 of the listing: if the variable n_t is limited to 64 bits of accuracy, it will eventually overflow for pattern orders $m > 20$. Fortunately, it merely takes

1. an arbitrary-precision integer representation for the variable n_t,
2. a function that adds a non-negative integer to n_t, and
3. a function that multiplies n_t with a non-negative integer

to overcome this upper boundary. Although books have been written about the details of arbitrary precision arithmetic [41,42], a simplistic approach will fully satisfy the current application. For an arbitrary pattern of order m, its maximum bit width is known in advance, and amounts to $\log_2 m!$ bit. To ease subsequent analyses, it is advisable to keep the width constant across all patterns of a given order, such that we do not have to be concerned with dynamic memory reallocation. Any operation applied to an arbitrary-precision variable n_t will then result in a fixed-length sequence of machine instructions, each acting on a fraction of the overall data. To minimise the instruction count, it thus makes sense to allocate an integer multiple of the machine word size per pattern. On contemporary hardware platforms, a total of

$$d = \lceil \log_2(m!)/64 \rceil \tag{25}$$

consecutive 64-bit words should be used for each ordinal pattern. Given a sequence of N ordinal patterns of order m, its resulting in-memory representation is then a 64-bit unsigned integer array comprised of $d \times N$ elements. The addition and multiplication functions required for the evaluation of Equation (24) should ideally act "in-place" on the array-valued operand n_t, while their respective second operands can safely be restricted to 32-bit unsigned integers. This is based on the conjecture that $m < 2^{32}$ will not constitute a limitation in practice. The pair of arithmetical functions then boil down to a straightforward addition-with-carry, as well as a schoolbook multiplication approach. For reference, see the functions `add_mp` and `multiply_mp`, as well as the resulting multi-precision implementation of the overlap algorithm called `ordpat_encode_overlap_mp` (all to be found in the supplementary file `ordpat.c`). As is to be expected, the multi-precision approach introduces a certain performance penalty when compared to the standard implementation. This overhead is depicted in Figure 5 for the range of pattern orders supported by both variants. The run-time behaviour of `ordpat_encode_overlap_mp` for a wider range of pattern orders is in turn visualised in Figure 6.

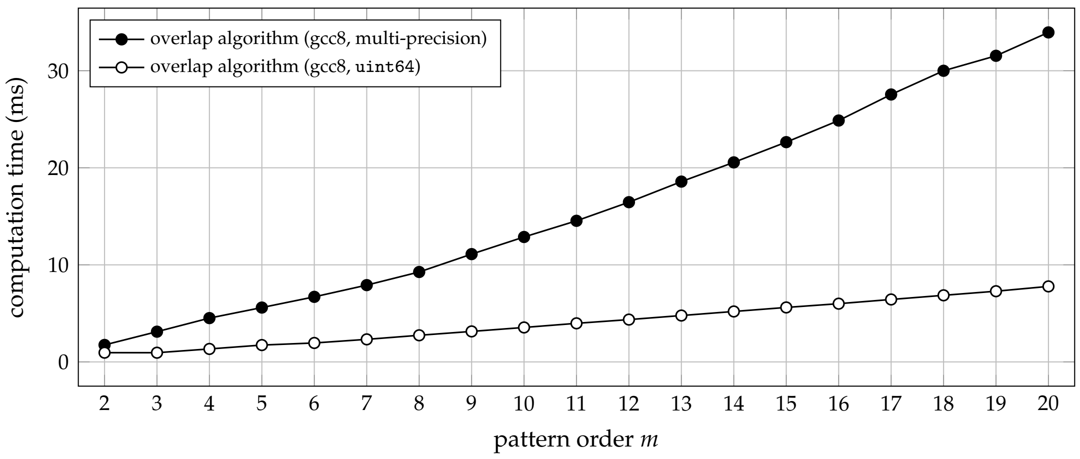

Figure 5. Computation time (median of 20 trials) for transforming 3.6×10^5 samples of uniformly distributed white noise into a sequence of ordinal patterns of pattern order m, using the constant time lag $\tau = 1$. The C functions `ordpat_encode_overlap` and `ordpat_encode_overlap_mp` from the supplementary file `ordpat.c` were used for the simulation. The arbitrary-precision arithmetic used in the `ordpat_encode_overlap_mp` function increases the overall run-time complexity, and the timing is less stable than for strictly hardware-based arithmetic operations.

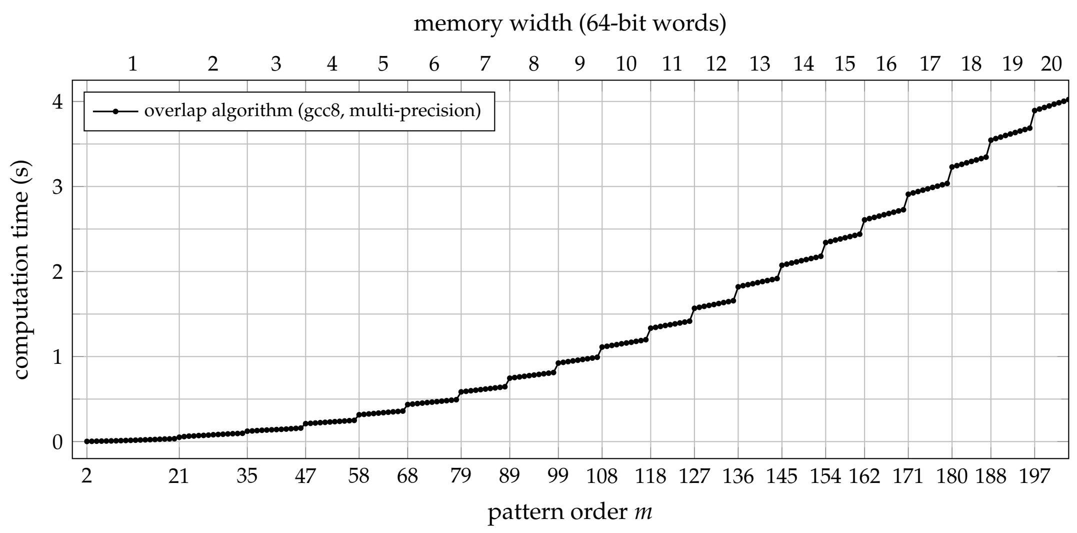

Figure 6. Computation time (median of 20 trials) for transforming 3.6×10^5 samples of uniformly distributed white noise into a sequence of ordinal patterns of order m. The time lag was set to $\tau = 1$, and the C function `ordpat_encode_overlap_mp` (an arbitrary-precision implementation of the overlap algorithm) from the supplementary file `ordpat.c` was used for the simulation. The memory required per pattern is growing with m in a stepwise manner, increasing by one 64-bit word at each vertical grid line. The computational cost in turn rises linearly with the number of memory words to be iterated for each pattern, which shows as distinct jumps in the graph. Independent of that, the run-time complexity also increases linearly with the pattern order m as such. Both effects combined explain the parabolic envelope of the curve depicted.

As can be seen from Figures 5 and 6, some of the run-time efficiency of the overlap algorithm has to be traded off to allow for pattern orders $m > 20$, which require multi-precision (and thus, multi-iteration) integer arithmetic. Although the overall computational complexity then scales with $O(m^2)$ again, the absolute run-time of the approach is still acceptable: encoding a one-hour sequence of data sampled at 100 Hz will take less than one second of processing time for orders as high as $m = 100$, for which an inconceivable number of $100! \approx 9.3 \times 10^{157}$ distinct ordinal patterns do formally exist.

5.7. The Lookup Algorithm (Algorithm 4)

In analogy with the overlap algorithm (Algorithm 3) discussed in Section 5.6, the lookup algorithm (Algorithm 4) cannot be fully vectorised due to its recursive nature. Regarding run-time efficiency, numerical scripting languages are thus at a considerable disadvantage compared to compiled programming languages. We therefore implemented the lookup algorithm in the C programming language to enable meaningful comparison with Algorithm 3 (see the `ordpat_encode_lookup` function provided in the supplementary `ordpat.c` file). Still, native implementations for numerical scripting languages are provided in the supplements for the sake of completeness. Those are meant to allow the reader a quick confirmation of the aforementioned performance drop.

In theory, the lookup algorithm (Algorithm 4) is computationally more efficient than the overlap algorithm (Algorithm 3). For each pattern to be encoded, the overlap algorithm has to calculate an integer representation $n \in \{0, 1, \ldots, m! - 1\}$ from a tuple of inversion counts $(r_1, r_2, \ldots, r_m)$, whereas the lookup algorithm can fetch the result of this operation from memory. This reduces the number of computational operations per ordinal pattern (see Table 2).

In practice, however, the overall run-time of a piece of software is not exclusively determined by the number of operations it performs, but also depends on memory requirements and memory access patterns. In this regard, and as described in Section 4.3, Algorithm 4 requires a lookup table of $m! \times m$ elements, each holding the numerical representation of a particular ordinal pattern of order m. Thus, the size of the lookup table increases rapidly with the pattern order: conservatively assuming

$\log_2 m!$ bit of storage space per pattern, the table size exceeds a gigabyte for $m = 11$, and occupies more than five terabytes of memory for the order $m = 14$. Therefore, memory access times quickly become prohibitive as the pattern order increases. For this reason, Unakafova and Keller stated that the applicability of their algorithm be limited to the pattern orders most commonly used [21], and provided precomputed lookup tables for $m \in \{2, 3, \ldots, 9\}$. Nevertheless, the principal problem of memory access times still persists for low pattern orders. In cases where the lookup table is too large to entirely reside in the processor's internal cache, the overall run-time efficiency of Algorithm 4 strongly depends on the nature of the input data. Time series of high ordinal complexity will then result in frequent cache misses. In other words: if the time series contains many different ordinal patterns, the processor will frequently have to reload different parts of the lookup table from main memory into cache, which stalls the processor and thus slows down the encoding process. This circumstance can be demonstrated by feeding low-pass filtered noise of increasing bandwidth to the lookup algorithm, which yields results as presented in Figure 7.

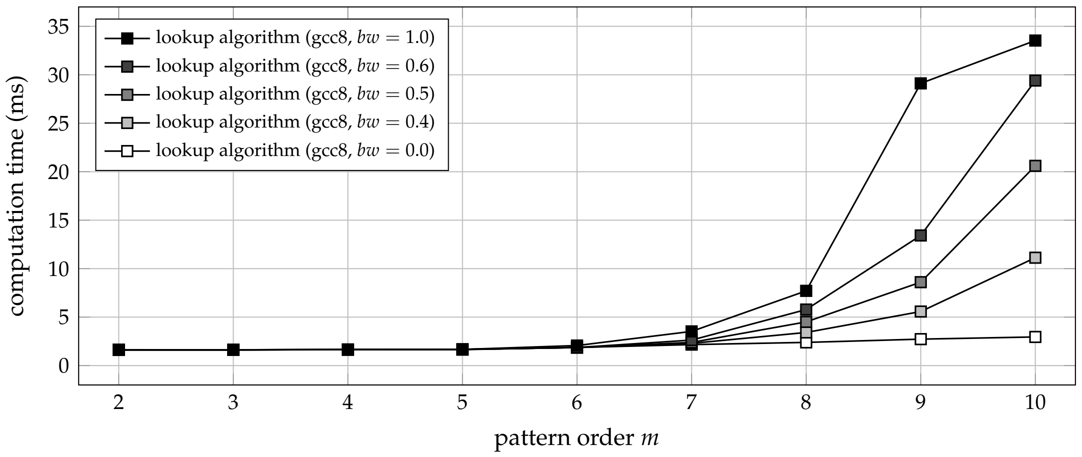

Figure 7. Computation time (median of 20 trials) for transforming 3.6×10^5 samples of uniform white noise, low-pass filtered to various relative bandwidths bw, into sequences of ordinal patterns of order m. The time lag was set to a constant $\tau = 1$, and the C functions `ordpat_encode_lookup` and `ordpat_encode_overlap` from the supplementary file `ordpat.c` were used for the simulation. The time required for loading lookup tables from mass storage into main memory was not taken into account. Filtering to $bw = 0.0$ results in an all-zero input signal, whereas $bw = 1.0$ results in white noise. The computation time increases not only with the pattern order m, but also with the ordinal complexity of the input signal.

It is therefore hard to draw a general conclusion on the run-time efficiency of the lookup algorithm. Suffice it to say that, for input data of low ordinal complexity, the lookup algorithm may outperform the overlap algorithm, as can be substantiated by using an all-zero time series as the test signal. In this idealised case, the algorithm will look up the exact same ordinal pattern again and again, and will therefore not run into cache contention issues, as is demonstrated in Figure 8.

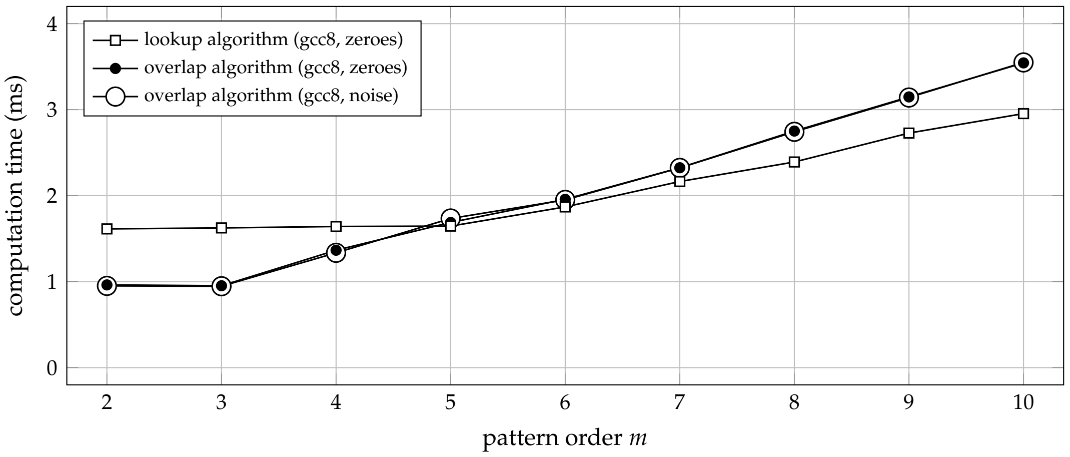

Figure 8. Computation time (median of 20 trials) for transforming 3.6×10^5 samples of data into sequences of ordinal patterns of order m. Either zeroes or uniform white noise were used as input data. The time lag was set to a constant $\tau = 1$, and the C functions `ordpat_encode_lookup` and `ordpat_encode_overlap` from the supplementary file `ordpat.c` were used for the simulation. The time required for loading lookup tables from mass storage into main memory was not taken into account. For an all-zero input signal, no cache contention will occur, and the lookup algorithm can outperform the overlap algorithm as the pattern order m (and thus the computational workload for the overlap algorithm) increases.

Figure 8 also shows that the performance of the overlap algorithm is independent from the input data. It remains stable for both extreme cases: sequences of zeroes, as well as white noise. The benchmark presented in Figure 8 also conveys the impression that the overlap algorithm may be at an advantage for $m \in \{2, 3, 4\}$ and any type of input signal. A possible explanation could be that addressing and accessing the lookup table in cache still imposes some constant delay, causing the processor to stall for the pattern orders with the lowest computational workload. However, considering that under the above conditions both algorithms achieve a data throughput of more than 1 GB per second, we did not study this effect any further: loading input data from mass storage will likely take a lot longer than the actual processing times considered here.

5.8. Sequence Length and Time Lag

In the previous simulations, signals of a fixed $N = 3.6 \times 10^5$ samples length have been processed, using the constant time lag $\tau = 1$ throughout. A remaining question therefore is how the plain, overlap and lookup algorithms (Algorithms 2–4) scale with regard to the length N of the input sequence, as well as the time lag τ under practical conditions. Fortunately, those aspects are qualitatively identical for all three algorithms, and their run-time behaviour is consistent with theoretical expectation: let aside the τ additional steps required to initialise the overlap and lookup algorithms, each of the algorithms is repeated once per ordinal pattern to be encoded, so doubling the amount of input data will coarsely double the computational effort. In addition, the data to be encoded are iterated in a linear manner. Therefore, neither inordinate cache misses nor incorrect branch prediction should pose a problem in theory. Simulation confirms that the run-time of all three algorithms scales linearly with the sequence length, as can be observed in Figure 9.

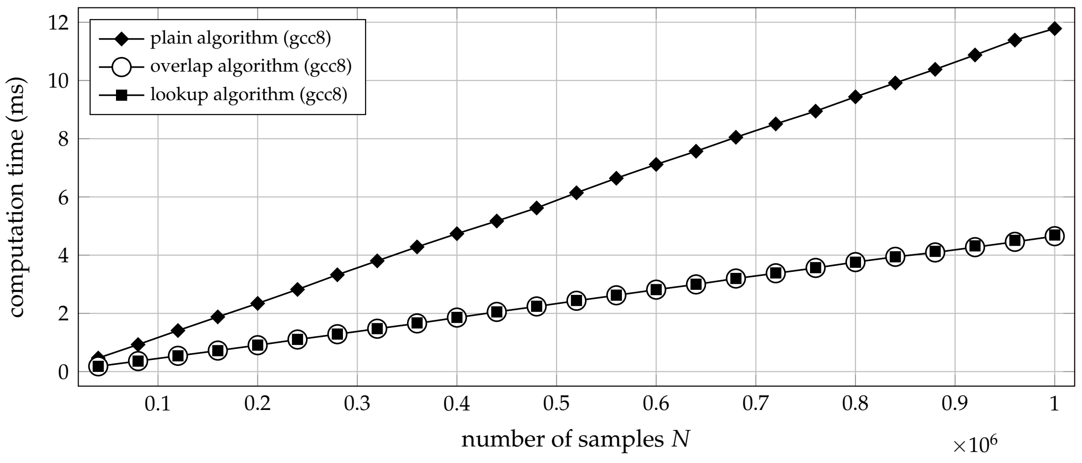

Figure 9. Computation time (median of 20 trials) for transforming N samples of uniform white noise into a sequence of ordinal patterns, using order $m = 5$ and time lag $\tau = 1$. The respective C functions `ordpat_encode_plain`, `ordpat_encode_overlap` and `ordpat_encode_lookup` from the supplementary file `ordpat.c` were tested. The time required for loading lookup table data from mass storage into main memory was not taken into account. The order $m = 5$ was selected so as to operate the `ordpat_encode_lookup` function at its sweet spot with regard to cache utilisation. In good approximation, the computation time then increases linearly with the sequence length N for all three algorithms.

The relation between the time lag τ and the overall run-time is even simpler. With respect to computational effort, the time lag should not make any difference because, for all three algorithms, the value of τ is predominantly used to calculate memory addresses, where its absolute value cannot have any influence on the computational workload. On the other hand, τ determines the stride of the (otherwise linear) memory access pattern. Therefore, increasing the time lag could theoretically be detrimental to the cache performance. Under test conditions, however, the choice of τ had no noticeable influence on run-time efficiency: consider the measurements depicted in Figure 10.

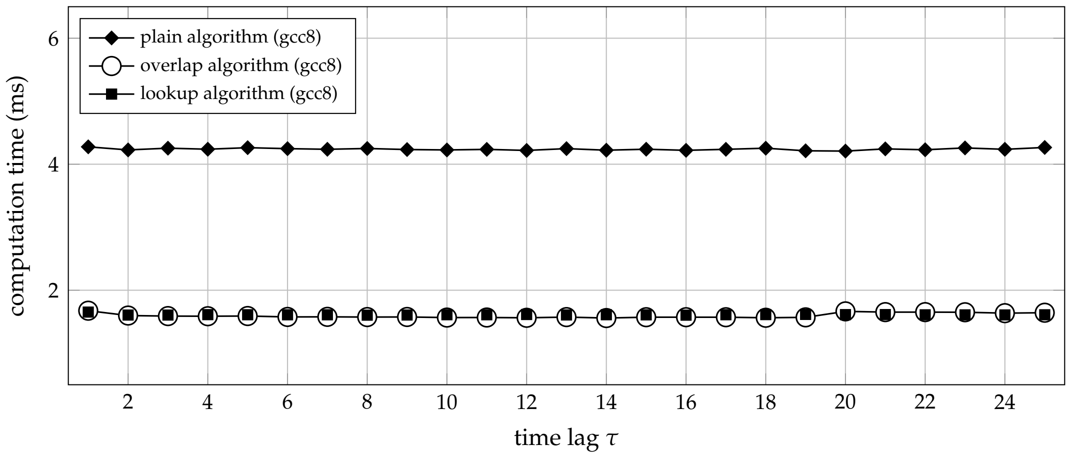

Figure 10. Computation time (median of 20 trials) for transforming 3.6×10^5 samples of uniform white noise into a sequence of ordinal patterns, using the fixed order $m = 5$ and increasing time lags τ. The respective C functions `ordpat_encode_plain`, `ordpat_encode_overlap` and `ordpat_encode_lookup` from the supplementary file `ordpat.c` were tested. The time required for loading lookup table data from mass storage into main memory was not taken into account. The order $m = 5$ was selected so as to operate the `ordpat_encode_lookup` function at its sweet spot with regard to cache utilisation. The simulations did not reveal any noticeable dependency between the time lag τ and the computation time.

6. Conclusions

Three different algorithms were discussed that all analyse the ordinal patterns (see Definition 1) of a given time series, and encode them in a computationally advantageous way, such that the ordinal patterns $\{\Pi_1, \Pi_2, \ldots, \Pi_{m!}\}$ of order m are compactly represented by the set of non-negative integers $\{0, 1, \ldots, m! - 1\}$ in a one-to-one manner. The theoretical foundations for this encoding were adopted from the Lehmer code [33], a classical approach in computational combinatorics (see Section 3).

6.1. Picking the Right Tool for the Job

From a theoretical perspective, the plain algorithm (Algorithm 2) has the highest computational complexity, followed by the overlap algorithm (Algorithm 3), and in turn followed by the lookup algorithm (Algorithm 4), which is the least computationally complex among the three (see Section 5.1). In practice, however, the algorithms presented are complementary with regard to their scope of application, and each can be worth considering.

Being fully vectorisable, the plain algorithm (Algorithm 2) is a particularly good choice for computational environments like Matlab, GNU Octave or NumPy/Python, and its efficiency should suffice most standard applications of ordinal pattern analysis (see Table 5).

By contrast, the overlap algorithm (Algorithm 3) constitutes a general-purpose solution, providing high data throughput over a wide range of pattern orders m, while only requiring a small amount of extra memory. To achieve suitable run-time performance, it needs to be implemented in a compiled programming language, though. This is not an actual limitation in practice because virtually any high-level scripting language can link against pre-compiled library functions. Under this paradigm of execution, the overlap algorithm clearly outperforms the plain algorithm. Implementing the overlap algorithm in the C programming language is straightforward, and provides plenty of opportunity for platform-specific optimisation: as demonstrated in Section 5.6, arbitrary-precision arithmetic can easily be incorporated to enable pattern orders $m > 20$, and (although not considered in this manuscript) single-instruction-multiple-data (SIMD) processing could be employed to further boost the run-time performance on supporting architectures. With regard to real-time applications running on specialised embedded systems, it may be worth mentioning that the algorithm does not depend on floating-point arithmetic, and merely uses a few extra bytes of working memory on top of its input/output buffers.

As with the overlap algorithm, the lookup algorithm (Algorithm 4) should ideally be implemented in a compiled programming language to maximise its performance. By matter of principle, it has a narrower scope of application, though. Depending on a lookup table of $m! \times m$ elements, its memory requirements currently limit the algorithm to pattern orders $m \in \{2, 3, \ldots, 10\}$. For the same reason, its run-time performance varies with the nature of the input data (see Figure 7). When analysing time series of comparatively low ordinal complexity, the lookup algorithm may outperform the overlap algorithm. On the other hand, time series of higher ordinal complexity will result in frequent cache misses, and may lead to a substantial drop in the overall run-time. Thus, if performance is critically important, both algorithms should be tested for the particular kind of data to be analysed.

6.2. Final Remarks

The general aim of the present article is to improve the run-time performance of known methods in ordinal pattern analysis, and to foster the development of new applications, so far hindered by computational limitations. To that end, the publication is supplemented by a cross-platform software library that supports various programming languages commonly used in scientific computing, namely NumPy/Python, GNU Octave, Matlab, as well as the C programming language. The library includes reference implementations for all algorithmic variants considered here, and is provided under the permissive terms of a 3-clause Berkeley Software Distribution (BSD) license.

Mapping time series onto sequences of ordinal patterns is essential to the methodology, but constitutes only the first step of ordinal pattern analysis. Other aspects often include the estimation of (possibly multi-dimensional) probability masses. As soon as the pattern order m increases beyond a certain point, those can pose a computational challenge in their own right. Thus, a follow-up article discussing such matters is currently under preparation.

Last but not least, it has to be stated clearly that the present work is exclusively concerned with computational feasibility as such, and in no way with the actual relevance of high pattern orders. Most prominently, the part on multi-precision arithmetic in Section 5.6 was written in the hope that it will be useful to researchers further exploring the possibilities of ordinal pattern analysis. That being said, the authors do not endorse the extension of well-established analysis methods to unreasonably high pattern orders: for a fixed sequence length N, not only computational efficiency, but also statistical validity may vanish quite rapidly as the pattern order m increases.

Supplementary Materials: The supplementary code is available online at http://www.mdpi.com/1099-4300/21/10/1023/s1, and will be developed further at https://github.com/seb-berger/libordpat.

Author Contributions: Conceptualization, S.B.; formal analysis, S.B., A.K. and D.J.; investigation, S.B. and A.K.; methodology, S.B.; software, S.B.; supervision, G.S. and D.J.; validation, S.B. and A.K.; visualization, S.B.; writing–original draft preparation, S.B.; writing–review and editing, S.B., A.K., G.S., and D.J.

Funding: This work was supported by the Deutsche Forschungsgemeinschaft (DFG) and the Technische Universität München (TUM) in the framework of the Open Access Publishing Program.

Acknowledgments: The authors thank the anonymous referees for their time, effort, and valuable contributions. S.B. thanks Mila Dorji for testing the supplementary code on macOS, Andreas Ranft for many inspiring discussions, and Kristine Kellermann for constant support and encouragement.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Bandt, C.; Pompe, B. Permutation Entropy: A Natural Complexity Measure for Time Series. *Phys. Rev. Lett.* **2002**, *88*, 174102. [CrossRef]
2. Shannon, C.E. A Mathematical Theory of Communication. *Bell Syst. Tech. J.* **1948**, *27*, 379–423. [CrossRef]
3. Shannon, C.E. A Mathematical Theory of Communication. *Bell Syst. Tech. J.* **1948**, *27*, 623–656. [CrossRef]
4. Zanin, M.; Zunino, L.; Rosso, O.A.; Papo, D. Permutation Entropy and Its Main Biomedical and Econophysics Applications: A Review. *Entropy* **2012**, *14*, 1553–1577. [CrossRef]
5. Amigó, J.M.; Keller, K.; Kurths, J. Recent Progress in Symbolic Dynamics and Permutation Complexity. *Eur. Phys. J. Spec. Top.* **2013**, *222*, 241–247. [CrossRef]
6. Amigó, J.M.; Keller, K.; Unakafova, V.A. Ordinal symbolic analysis and its application to biomedical recordings. *Philos. Trans. R. Soc. A Math. Phys. Eng. Sci.* **2014**, *373*, 20140091. [CrossRef]
7. Fadlallah, B.; Chen, B.; Keil, A.; Príncipe, J. Weighted-permutation entropy: A complexity measure for time series incorporating amplitude information. *Phys. Rev. E* **2013**, *87*, 022911. [CrossRef]
8. Li, D.; Li, X.; Liang, Z.; Voss, L.J.; Sleigh, J.W. Multiscale permutation entropy analysis of EEG recordings during sevoflurane anesthesia. *J. Neural Eng.* **2010**, *7*, 046010. [CrossRef]
9. Morabito, F.C.; Labate, D.; La Foresta, F.; Bramanti, A.; Morabito, G.; Palamara, I. Multivariate Multi-Scale Permutation Entropy for Complexity Analysis of Alzheimer's Disease EEG. *Entropy* **2012**, *14*, 1186–1202. [CrossRef]
10. Azami, H.; Escudero, J. Amplitude-aware permutation entropy: Illustration in spike detection and signal segmentation. *Comput. Methods Programs Biomed.* **2016**, *128*, 40–51. [CrossRef]
11. Unakafov, A.M.; Keller, K. Conditional entropy of ordinal patterns. *Phys. D Nonlinear Phenom.* **2014**, *269*, 94–102. [CrossRef]
12. King, J.R.; Sitt, J.D.; Faugeras, F.; Rohaut, B.; El Karoui, I.; Cohen, L.; Naccache, L.; Dehaene, S. Information sharing in the brain indexes consciousness in noncommunicative patients. *Curr. Biol.* **2013**, *23*, 1914–1919. [CrossRef]
13. Schreiber, T. Measuring Information Transfer. *Phys. Rev. Lett.* **2000**, *85*, 461–464. [CrossRef]
14. Staniek, M.; Lehnertz, K. Symbolic Transfer Entropy. *Phys. Rev. Lett.* **2008**, *100*, 158101. [CrossRef]
15. Groth, A. Visualization of coupling in time series by order recurrence plots. *Phys. Rev. E* **2005**, *72*, 046220. [CrossRef]

16. Bandt, C.; Shiha, F. Order patterns in time series. *J. Time Ser. Anal.* **2007**, *28*, 646–665. [CrossRef]
17. Keller, K.; Sinn, M.; Emonds, J. Time Series From the Ordinal Viewpoint. *Stochastics Dyn.* **2007**, *7*, 247–272. [CrossRef]
18. Amigó, J.M. *Permutation Complexity in Dynamical Systems*; Springer: Berlin/Heidelberg, Germany, 2010. [CrossRef]
19. Cao, Y.; Tung, W.w.; Gao, J.B.; Protopopescu, V.A.; Hively, L.M. Detecting dynamical changes in time series using the permutation entropy. *Phys. Rev. E* **2004**, *70*, 046217. [CrossRef]
20. Frank, B.; Pompe, B.; Schneider, U.; Hoyer, D. Permutation entropy improves fetal behavioural state classification based on heart rate analysis from biomagnetic recordings in near term fetuses. *Med. Biol. Eng. Comput.* **2006**, *44*, 179. [CrossRef]
21. Unakafova, V.A.; Keller, K. Efficiently Measuring Complexity on the Basis of Real-World Data. *Entropy* **2013**, *15*, 4392–4415. [CrossRef]
22. Zunino, L.; Olivares, F.; Scholkmann, F.; Rosso, O.A. Permutation entropy based time series analysis: Equalities in the input signal can lead to false conclusions. *Phys. Lett. A* **2017**, *381*, 1883–1892. [CrossRef]
23. Dickten, H.; Lehnertz, K. Identifying delayed directional couplings with symbolic transfer entropy. *Phys. Rev. E* **2014**, *90*, 062706. [CrossRef]
24. Unakafov, A.M.; Keller, K. Change-Point Detection Using the Conditional Entropy of Ordinal Patterns. *Entropy* **2018**, *20*, 709. [CrossRef]
25. Kaiser, A.; Schreiber, T. Information transfer in continuous processes. *Phys. D Nonlinear Phenom.* **2002**, *166*, 43–62. [CrossRef]
26. Staniek, M.; Lehnertz, K. Parameter selection for permutation entropy measurements. *Int. J. Bifurc. Chaos* **2007**, *17*, 3729–3733. [CrossRef]
27. Amigó, J.M.; Kocarev, L.; Szczepanski, J. Order patterns and chaos. *Phys. Lett. A* **2006**, *355*, 27–31. [CrossRef]
28. Amigó, J.M.; Zambrano, S.; Sanjuán, M.A.F. True and false forbidden patterns in deterministic and random dynamics. *Europhys. Lett.* **2007**, *79*, 50001. [CrossRef]
29. Riedl, M.; Müller, A.; Wessel, N. Practical considerations of permutation entropy. *Eur. Phys. J. Spec. Top.* **2013**, *222*, 249–262. [CrossRef]
30. Iverson, K.E. *A Programming Language*; Wiley: New York, NY, USA, 1962.
31. Knuth, D.E. Two Notes on Notation. *Am. Math. Mon.* **1992**, *99*, 403. [CrossRef]
32. Laisant, C.A. Sur la numération factorielle, application aux permutations. *Bull. la Société Mathématique Fr.* **1888**, *16*, 176–183. [CrossRef]
33. Lehmer, D.H. Teaching combinatorial tricks to a computer. In *Proceedings of Symposia in Applied Mathematics*; Bellman, R.; Hall, M., Jr., Eds.; American Mathematical Society: Providence, RI, USA, 1960; Volume 10, pp. 179–193. [CrossRef]
34. Eaton, J.W. GNU Octave and reproducible research. *J. Process Control* **2012**, *22*, 1433–1438. [CrossRef]
35. Oliphant, T.E. Python for Scientific Computing. *Comput. Sci. Eng.* **2007**, *9*, 10–20. [CrossRef]
36. *IEEE 754-1985: IEEE Standard for Binary Floating-Point Arithmetic*; IEEE Computer Society: New York, NY, USA, 1985. [CrossRef]
37. *IEEE 754-2008: IEEE Standard for Floating-Point Arithmetic*; IEEE Computer Society: New York, NY, USA, 2008. [CrossRef]
38. Goldberg, D. What every computer scientist should know about floating-point arithmetic. *ACM Comput. Surv.* **1991**, *23*, 5–48. [CrossRef]
39. Marsaglia, G. Xorshift RNGs. *J. Stat. Softw.* **2003**, *8*, 1–6. [CrossRef]
40. Berger, S.; Schneider, G.; Kochs, E.F.; Jordan, D. Permutation Entropy: Too Complex a Measure for EEG Time Series? *Entropy* **2017**, *19*, 692. [CrossRef]
41. St. Denis, T.; Rose, G. *BigNum Math: Implementing Cryptographic Multiple Precision Arithmetic*; Syngress Publishing: Rockland, MA, USA, 2006. [CrossRef]
42. Brent, R.P.; Zimmermann, P. *Modern Computer Arithmetic*; Cambridge University Press: Cambridge, UK, 2010. [CrossRef]