Compress Time Series with Smaller Error Tolerances

Juntao Yu[1], Fangyu Wu[2], Huanyu Zhao[3], Shiting Wen[1], Tongliang Li[4], and Chaoyi Pang[1]✉

[1] NingboTech University, Ningbo, China
luvox__@outlook.com, wensht@nbt.edu.cn, chaoyi.pang@qq.com
[2] Xi'an Jiaotong-Liverpool University, Suzhou, China
fangyu.wu02@xjtlu.edu.cn
[3] Donghua University, Shanghai, China
zhaohuanyu@163.com
[4] Hebei Academy of Sciences, Shijiazhuang, China
litongliang@tom.com

Abstract. Despite the wealth of time series data collected across various fields, its sheer volume presents significant challenges for efficient usage and management in database systems. By compressing time series data within a predefined error tolerance for each individual data point, it can greatly boost data storage capacity, accelerate data transmission rates, and, most importantly, enable high-quality analytical tasks. In this paper, we introduce novel algorithms for compressing floating-point time series data, ensuring that each decoded data point stays within a predefined error bound. Our proposed methods are based on XOR encoding, a widely adopted approach used by leading lossless floating-point compression techniques, and are further optimized by selecting either the *"shortest"* XORed expressions or the *shortest* bits on *"mantissa+flags"* the permissible tolerance range. Extensive experiments on widely used datasets demonstrate that our algorithms significantly outperform many state-of-the-art max-error bound compression algorithms, in both compression ratio and execution time. Specifically, for smaller tolerable errors, our algorithms achieve compressed data sizes that are approximately 45% and 65% of those produced by leading lossless and lossy floating-point compression algorithms respectively, while incurring only 75% and 30% of the execution costs in compression and decompression when compared to those competitors.

Keywords: Time series data· Lossless/Lossy compression· Quality guaranteed lossy compression

1 Introduction

The exponential growth of time series data has become a significant challenge in fields such as healthcare and finance. Efficient management and storage of this data are critical, and compression algorithms play a key role in reducing storage and transmission costs while preserving essential data characteristics for analysis. Compression algorithms can be broadly categorized into two types: lossy and lossless.

Lossy compression reduces data size by allowing a predefined error tolerance. A common approach is max-error bound (L_∞-bound) compression, ensuring each data point stays within a specified error range. Techniques include hierarchical methods like wavelet compression [11] and non-hierarchical methods like Piecewise Linear Approximation (PLA) [13,14].

Wavelet compression decomposes time series into approximation and detail coefficients, selectively retaining coefficients for compression. PLA, on the other hand, approximates time series with line segments. Recent advancements, such as Sim-Piece [3] and Mix-Piece [4], improve PLA by grouping similar segments, achieving excellent compression ratios and execution speeds.

Lossless compression retains all original data points, ensuring exact reconstruction after decompression. A common technique is *delta encoding*, which encodes differences between successive data points, resulting in minimal variations suitable for further encoding. XOR-based encoding, a variant of delta encoding, captures bit-wise differences between consecutive values, enabling significant storage reduction with low computational cost. Gorilla [12], a brilliant lossless algorithm, uses XOR-based encoding for efficient floating-point time-series compression. Chimp [7] optimizes XORed leading zeros, improving compression and decompression speeds. Elf [6] further enhances efficiency by maximizing XORed trailing zeros through a pre-treatment step.

In real-world applications, data collected in volatile environments is often inherently lossy and prone to errors, with imprecise representations due to numerical limitations in computers.

In this paper, we propose two novel lossy compression algorithms with max-error bound for floating-point time series: SXC (Shortest XOR Compression) and SMC (Shorter Mantissa with smaller flags bits Compression). Both algorithms use XOR-based encoding, typically associated with lossless compressions, ensure each decoded data point stays within a predefined error tolerance, making them lossy with a max-error bound. To the best of our knowledge, SXC and SMC are the first such XOR-based compression algorithms in the context of max-error bound. The key contributions of this paper can be summarized as follows:

⧫ We propose two novel XOR-based encoding max-error bound compression algorithms SXC and SMC. SXC is designed to use the shortest bits of XORed results within the imposed tolerant error. SMC is constructed to encode the minimized expression bits of flags and mantissa within the tolerant error.

⧫ Our proposed algorithms SXC and SMC have linear-time complexity and $O(1)$ space complexity. Compared with many existing methods, including Sim-Piece etc., SXC and SMC are very efficient in real applications, ensuring faster execution and lower memory footprint.

⧫ SXC and SMC are compared with the state-of-the-art lossy and lossless algorithms on 90 widely used datasets. The extensive test results indicate that SXC and SMC exhibit the most superior performances on compression ratio, compression/decompression time under smaller error tolerances: The compressed data size of SMC and SXC is about 2/3 of that leading compression algorithms, while incurring only 75% and 30% of the execution costs in compression and decompression when compared to those competitors.

The rest of the paper is organized as follows. Section 2 is the preliminaries; Section 3 delves into related works; Section 4 presents of our algorithms; Section 5 is the experimental results, followed by conclusion in Section 6.

2 Preliminaries

This section will introduce concepts used in this paper. We will cover key definitions, relevant theories, and any necessary background information for easier understanding of this paper.

2.1 Floating-Point Time Series

A floating-point time series consists of collected data points arranged in ascending time order. It is denoted as $TS = \langle p_1, p_2, \ldots, p_i, \ldots \rangle$ or $TS = \langle (t_1, v_1), (t_2, v_2), \ldots, (t_i, v_i), \ldots \rangle$. At point $p_i = (t_i, v_i)$, the floating-point value v_i was recorded at the specific timestamp t_i.

Efficient compression of floating-point time series often involves compress timestamps and floating-point values separately. However, in this paper, we assume that $t_i - t_{i-1} = t_{i+1} - t_i$ holds for $i = 1 \ldots n-1$, which indicates that timestamps do not need to be compressed or storaged.

This paper concentrates on the compression of double-precision (floating-point) values, generally abbreviated as "double values". According to the IEEE 754 Standard [1], a double value v is stored using 64 binary bits. Among these bits, 1 bit is designated for the sign, 11 bits are allocated for the exponent, and 52 bits for the mantissa, as illustrated in Equation 1. Specifically, the exponent is denoted as $Exp(v) = \boldsymbol{e} = \langle e_1, e_2, \ldots, e_{11} \rangle$, and the mantissa as $\boldsymbol{m} = \langle m_1, m_2, \ldots, m_{52} \rangle$.

$$v = (-1)^s \times 2^{\sum_{i=1}^{11} e_i \times 2^{11-i} - 1023} \times (1 + \sum_{i=1}^{52} m_i \times 2^{-i}). \tag{1}$$

2.2 Erasing and Re-erasing

For binary representation, Buff [10] first introduced the erasing operation, which aims to ignore less significant bits in the mantissa. Elf [6] enhanced this idea by applying erasing as a preprocessing step before XOR, significantly increasing trailing zeros in the XOR result. This improvement boosts compression efficiency when used with Gorilla-like XOR compressors [6].

Due to the limitations of floating-point representation in computers, many decimal fractions, such as 0.9, cannot be precisely represented in binary form. For example, the binary representation of 0.9 corresponds to the decimal value

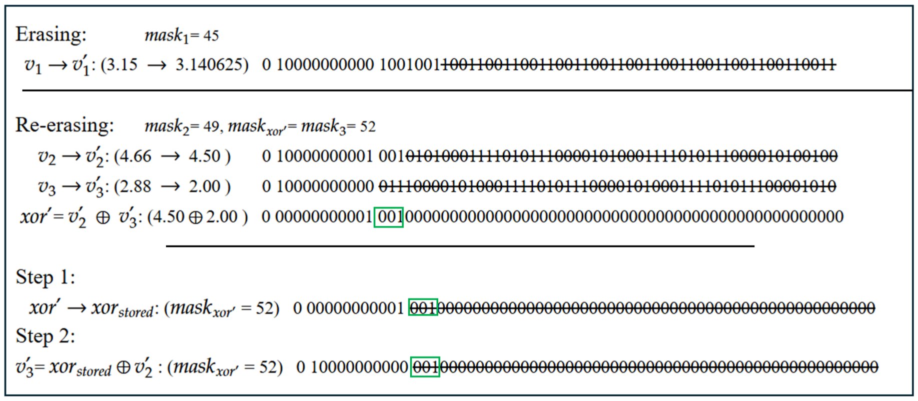
Erasing: $mask_1 = 45$
$v_1 \rightarrow v_1'$: (3.15 → 3.140625) 0 10000000000 1001001~~100110011001100110011001100110011001100110011~~
Re-erasing: $mask_2 = 49$, $mask_{xor'} = mask_3 = 52$
$v_2 \rightarrow v_2'$: (4.66 → 4.50) 0 10000000001 001~~0101000111101011100001010001111010111000010100100~~
$v_3 \rightarrow v_3'$: (2.88 → 2.00) 0 10000000000 ~~0111000010100011110101110000101000111101011100001010~~
$xor' = v_2' \oplus v_3'$: (4.50 ⊕ 2.00) 0 00000000001 0010000000000000000000000000000000000000000000000000
Step 1:
$xor' \rightarrow xor_{stored}$: ($mask_{xor'} = 52$) 0 00000000001 ~~0010000000000000000000000000000000000000000000000000~~
Step 2:
$v_3' = xor_{stored} \oplus v_2'$: ($mask_{xor'} = 52$) 0 10000000000 ~~0010000000000000000000000000000000000000000000000000~~

Fig. 1: Our erasing and re-erasing.

0.9000000000000000222. . . , which is only considered equal to 0.9 within a certain precision. However, within a given error tolerance, we can introduce deliberate representation errors to reduce the size of a double value. As shown in the following example, for a positive double value $v = 3.15$ and a error tolerance like 0.01, a straightforward approach would be to set certain bits in its binary representation to 0, thereby reducing the value of v.

$$\begin{aligned}
3.15 &= (11.0010011001100110011001100110011\ldots\ldots)_2 \\
&\approx (11.0010011001100110011001100110011)_2 \\
&< (11.001001\cancel{1001100110011001100110011})_2 + 10^{-2} \leq 3.140625 + 10^{-2}, \qquad (2) \\
3.15 &= 3.14\cancel{0625} + 10^{-2}.
\end{aligned}$$

By this method, we can represent 3.15 with 3.140625 within a given error of 0.01, required much less bits in its binary expression. This serves as the basic prototype of our "erasing" operation that we would like to introduce. As shown in Figure 1, our erasing operation transforms $v_1 = 3.15$ into $v_1' = 3.140625$ by setting several consecutive bits to '0' in the tail of the binary double representation. The parts were set to 0 are considered to have no contribution to the XOR operation, and the length of the zeros is referred to as the "mask", which satisfies mask≤ 52 for double values.

As shown in Figure 1, after performing $v_2' \oplus v_3'$, the parts that we previously considered to have no contribution in v_3 reemerge in the corresponding parts of the xor', as listed in the highlighted area. This undermines our goal of reducing storage space. To rectify this, we implement Step 1, which involves $xor' \rightarrow xor_{stored}$, to carry out the "erasing" of this part and decrease the storage requirements for the xor'. However, this action results in the reappearance of this part when recovering v_3' from xor_{stored}, calculated as $xor_{stored} \oplus v_2'$. To address this issue, we execute the "erasing" once more in Step 2, which is "erasing" v_3'. We refer to this two-step process of "erasing" as "re-erasing".

Through erasing and re-erasing, we can greatly reduce the storage requirements for the XOR values while recovering the desired values in a lossless manner.

And we can store the "mask" using just 4 bits hashing, which will be explained in Section 4.

2.3 Feasible Region

For a given double value v and an error tolerance δ, as shown in Figure 2, the range of selectable values of $[v^-, v^+] = [v - \delta, v + \delta]$ is called tolerate region. By performing $v^- \oplus v^+$, we can clearly identify the (longest) changed bits (or shortest mantissa) within the interval $[v^-, v^+]$. Let $bin(v)$ be the binary form of v expressed as.

$$bin(v) = \pm(b_h(v) \cdots b_1(v)b_0(v).b_{-1}(v) \cdots b_{-t}(v))_2 = (\sum_{i=-t}^{h} b_i(v) \times 2^i)_{10} \tag{3}$$

where $b_i(v) \in \{0, 1\}$ for $i = -t, \cdots, h$. Let $b_{-j}(v^- \oplus v^+)$ be the first left most bit of $v^- \oplus v^+$ satisfying $b_{-j}(v^- \oplus v^+) = 1$ and $b_{-i}(v^- \oplus v^+) = 0$ for $i > j$. Then feasible region $F(v, \delta)$ is $[0, 2^{-j}]$, indicating that the shortest mantissa of $[v^-, v^+]$ has j bits mostly.

Since the change bit can be on the Sign part, Exponent part, or Mantissa part, we have the following explanations:

Sign: To avoid storage overhead, we simply select "0" whenever $0 \in [v^-, v^+]$ as illustrated in Case 1. In this case, the feasible region is simply defined as "0".
Mantissa: Illustrated by the two extreme cases of Case 2.1 and Case 2.2.
Exponent: This case is very similar to Case 2 and illustrated in Case 3.

3 Related Work

There are two categories of compression methods, lossy compression algorithms and lossless compression algorithms.

Lossy Floating-Point Compression Floating-point data compression is a challenging task due to its complex storage format. Recently, various lossy floating-point data compression methods[9,10] have been proposed to address this challenge, aiming to reduce storage space while preserving precision. Liu et al. proposed Buff [10] that uses a decomposed columnar storage and encoding methods to provide effective compression with SIMD support. However, Buff focuses on converting double-precision numbers to single-precision numbers, without considering double-precision floating-point numbers. Liang et al. [8] further designed a prediction-based compression method SZ3 for double-precision floating-point numbers. By separating and abstracting stages in the prediction-based compression model, allowing efficient random access while preserving high precision. However, SZ3 requires the use of certain lossless compression algorithms during the compression process, which increases the algorithm's complexity and computational overhead.

Another category of lossy floating-point compression methods is based on Piece-wise Linear Approximation (PLA) for compression [3,14,4]. Recently, a

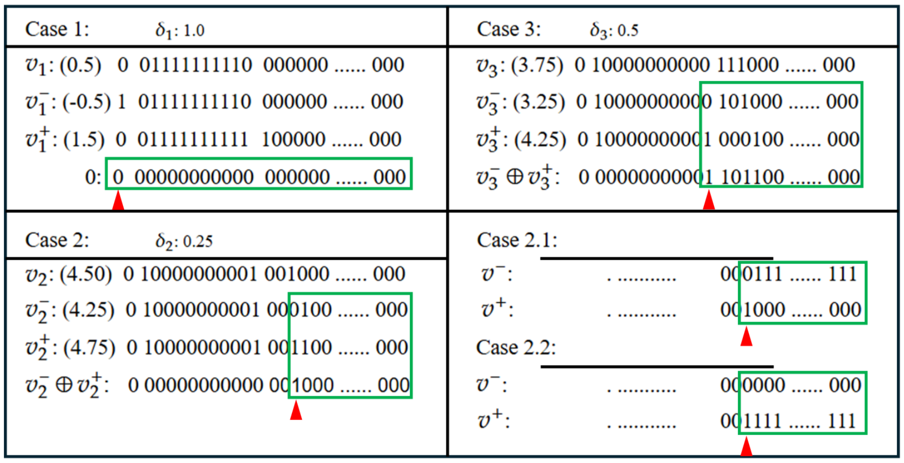
Case 1: δ_1: 1.0
v_1: (0.5) 0 01111111110 000000 000
v_1^-: (-0.5) 1 01111111110 000000 000
v_1^+: (1.5) 0 01111111111 100000 000
0: 0 00000000000 000000 000
Case 3: δ_3: 0.5
v_3: (3.75) 0 10000000000 111000 000
v_3^-: (3.25) 0 10000000000 101000 000
v_3^+: (4.25) 0 10000000001 000100 000
$v_3^- \oplus v_3^+$: 0 00000000001 101100 000
Case 2: δ_2: 0.25
v_2: (4.50) 0 10000000001 001000 000
v_2^-: (4.25) 0 10000000001 000100 000
v_2^+: (4.75) 0 10000000001 001100 000
$v_2^- \oplus v_2^+$: 0 00000000000 001000 000
Case 2.1:
v^-: 000111 111
v^+: 001000 000
Case 2.2:
v^-: 000000 000
v^+: 001111 111

Fig. 2: Example of feasible region.

novel lossy compression algorithm Sim-Piece [3] is designed to recompress grouped PLA segments to enhance efficiency in compression ratio and running time. This work is then improved by Sim-Piece+ and Mix-Piece subsequently, resulting in excellent performances in compression ration and running time.

Lossless Floating-Point Compression Building on Chimp [7], Elf and its extention Elf+ [6,5] introduced an erasing operation [5] as a preprocessing step to enhance XOR efficiency by replacing less significant mantissa bits with zeros, greatly reducing stored XOR results.

The Elf algorithm incorporates a preprocessing step known as "erasing", which aims to eliminate less significant bits in the data before applying XOR operations. This step results in XOR values that typically capture the differences between consecutive data points becoming shorter. Consequently, this reduction in the length of XOR values decreases storage requirements, as fewer bits are needed for each XOR result. However, Elf and Elf+ do not effectively manage the specified error, as they are designed to operate within the minimum precision of the given number. Moreover, if the minimum precision is considered as the error, Elf's "erasing" strategy covers only "half" of the error interval, meaning the erased value is always smaller than the original value.

Table 1: Meanings of symbols and terms.

Symbol	Meaning
δ	error bound on data points
$p_i = (v_i, t_i)$	a time series data point with a value of v_i and a timestamp of t_i
$TS_n = \langle p_1, p_2, \ldots, p_n \rangle$	a time series of length n
$v_i^+ = v_i + \delta$ $(v_i^- = v_i - \delta)$	max (min) allowable value of v_i
$Exp(v_i)/Mas(v_i)$	the exponent/mantissa part of v_i
$F(v,\delta)/\lvert F(v,\delta)\rvert$	feasible region derived from $[v^-, v^+]$ and its length

4 Algorithm

In this section, we will discuss the two proposed algorithms SXC and SMC. Table 1 summarizes the symbols frequently used in this paper[5].

4.1 General Idea

Let u and v be two double values, and δ be an error tolerance. The technique problems of our proposed algorithms SXC and SMC are formulated as follows:

SXC: Selects a $v' \in [v^-, v^+]$ such that the significant bits of $u \oplus v'$ are the shortest. Let γ denote the obtained shortest number of significant bits.

As detailed in Section 2.3, Suppose that the feasible region $F(v, \delta) = [0, 2^{-j}]$. Based on Equation 3, we set $b_i(v') = b_i(v)$ for $i > j$, $b_j(v') = 1$, and $b_i(v') = b_i(u)$ for $i < j$. Then, without considering changed exponent, the shortest the number of significant bits γ_1 of $u \oplus v'$ is the second shortest at least; γ_2 will be explained in the next. That is,

$$|u \oplus v'| - 1 \leq \gamma_1 \leq |u \oplus v'|, \qquad 1 \leq \gamma_2 \leq 12. \tag{4}$$

For the changed exponent, we set $b_j(v') = 0$ for $1 \leq j \leq 52$. Then, iterate through $[Exp(v^-) + 1, Exp(v^+)]$, making $Exp(v')$ to be any number within the interval. The shortest significant bits γ_2 of $u \oplus v'$ in this case is in Equation 4.

The shortest XOR result can be obtained from γ_1 and γ_2.

SMC: Selects a $v' \in [v^-, v^+]$ with the shortest mantissa that can be efficiently implemented with minimized flags' bits.

From the obtained feasible region $F(v, \delta) = [0, 2^{-j}]$, we can easily find v' from $[v^-, v^+]$ with the shortest mantissa by setting $b_i(v') = b_i(v)$ for $i > j$, $b_j(v') = 1$, and $b_i(v') = 0$ for $i < j$. The issues on minimizing flags' bits and efficiency optimization will be discussed in Section 4.3.

Since SXC and SMC rely on the correlation or trend between adjacent values for compression, their applicability is relatively limited for string compression with highly variable patterns.

4.2 SXC Algorithm

The implementation process of SXC is described as follows.

- Based on Equation 3, if $b_i(u) = b_i(p)$ for any $p \in [v^-, v^+]$ and $i <= j$, we set $v' = v$, mask= j, re-erasing= False, such that $b_i(v') = b_i(u)$ for $i < j$. Therefore, $b_i(v' \oplus u) = 0$ for $i < j$.
- Otherwise, $b_i(u) < b_i(v^-)$ or $b_i(u) > b_i(v^+)$ for $i \geq j$. We can set $v' = v^+$, mask= j- 1, re-erasing= True. Therefore, $xor_{stored} = u \oplus v'(erasing)$, $b_i(xor_{stored}) =$ "$100 \ldots 0$ for $i \geq j$ and its length is the shortest, which is 1.

[5] For simplicity, we have adopted some of the notations used in Elf [6].

For Case (1), it is evident that, the feasible region in $xor = u \oplus v$ has already been set to 0, thereby achieving the shortest $u \oplus v$.

For Case (2), it may involve changing the exponent of $xor = u \oplus v$ and the cases discussed in the following.

Exponent Unchanged We only need to focus on the mantissa part. In this case, there are $b_i(u) < b_i(v^-)$ or $b_i(u) > b_i(v^+)$ for $i \leq j$, with $b_j(v^+) = 1$ and $b_j(v^-) = 0$, respectively.For these two cases, the feasible regions are as follows, where ?' represents 0', '1', or the absence of that bit:

$$\begin{aligned} v_1^+ &: 1\ \sout{???\ ?\ ??????\ldots} & u_2' &: 1\ ???\ 1\ ??????\ldots \\ v_1^- &: 0\ ???\ 1\ ??????\ldots & v_2^+ &: 1\ \sout{???\ 0\ ??????\ldots} \\ u_1' &: 0\ ???\ 0\ ??????\ldots & v_2^- &: 0\ ???\ ?\ ??????\ldots \\ xor_1' = v_1^+ \oplus u_1' &: 1\ \sout{???\ ?\ ??????\ldots} & xor_2' = v_2^+ \oplus u_2' &: 0\ \sout{???\ 1\ ??????\ldots} \\ v_1' = xor_1' \oplus u_1' &: 1\ \sout{???\ ?\ ??????\ldots} & v_2' = xor_2' \oplus u_2' &: 1\ \sout{???\ 1\ ??????\ldots} \end{aligned} \tag{5}$$

The $u \oplus v'$ will always contain at least one '1'. However, if we select any v' with "100...000"$\leq b_i(v') \leq b_i(v^+)$, then perform erasing with mask= j-1, we can obtain $u \oplus v'$ with $\gamma = 1$, which is the shortest.

Exponent Changed First, for given three numbers u, p, q with $Exp(q) = Exp(p) + 1$ and $Mas(p) = Mas(q) = 0$, for any $p' \in [p, q)$, there always ensures that $p \oplus u$ performing erasing with mask= 52 is shortest than any $u \oplus p'$. Therefore, we can divide this situation into two parts:

▲ Suppose there is a number p with $Exp(p) = Exp(v^-) - 1$. Then, the numbers in $[p, v^+]$ can all be represented by numbers with all 0s mantissas. By comparing these different XOR values, we can obtain the shortest XOR value for this case, denoted as xor_1.

▲ For the remaining part, i.e., $[v^-, p)$, the handling method is similar to the approach in **Exponent Unchanged**. Then we can obtain the shortest XOR in this case, denoted as xor_2.

Finally, by comparing xor_1 and xor_2, we select the shortest one. The pseudocode of the SXC is shown in Algorithm 1.

Encoding scheme We improve Chimp encoding scheme [7] to better accommodate the characteristics of the XOR values generated by the algorithm, as illustrated in Figure 3. Specifically, for the representation of leading zeros, we extend the concept from Chimp and further refined it to address additional scenarios arising from the variable count of leading zeros. Specifically, the Count of Leading Zeros (CLZ) is expressed as follows:

$$CLZ = \lfloor CLZ_{original}/2 \rfloor * 2 \tag{6}$$

According to our algorithm, when $|v| < \delta$, we choose $v' = 0$. The subsequent erasing and re-erasing operations will also result in xor_{stored} being 0, which is the same as the XOR when $v = u$. Therefore, we set mask= 63 to distinguish between these cases and we use 2 flag bits to represent these four cases.

Algorithm 1: SXC Algorithm

```
if $Exp(v^-) \neq Exp(v^+)$ then
    p ← $Shortest\_Exp(v^-, v^+)$
    if $Mas(v^-) \leq Mas(u)$ then
        $v' \leftarrow v^-$
        mask ← 52
        re-erasing ← True
    else if $Mas(v^-) = 0$ then
        return p
    else
        pos ← $First\_0(v^-)$
        mask ← 63-pos
        // $v^-$to $v'$
        q ← $Transfer(v^-, mask)$
        re-erasing ← True
    $xor_1 \leftarrow p \oplus u$
    $xor_2 \leftarrow (q \oplus u)$// erasing
    $v' \leftarrow Shortest\_Xor(xor_1, xor_2)$
    return $v'$
```

Algorithm 1: SXC Algorithm

```
if $Exp(v^-) = Exp(v^+)$ then
    $v' \leftarrow v^+$
    // for any $i \geq j$
    if
     $b_i(v^-) \leq b_i(u) \leq b_i(v^+)$
     then
        mask ← $j$
    else
        mask ← $j - 1$
        re-erasing← True
$v' \leftarrow v'$// erasing
return $v'$
```

Algorithm 2: SMC Algorithm

```
$v' \leftarrow v^+$
if $Exp(v^-) \neq Exp(v^+)$ then
    $mask \leftarrow j$
else
    $mask \leftarrow j - 1$
$v \leftarrow v'$// erasing
return $v'$
```

When $xor_{stored} \neq 0$, in experiment conditions, the precision of the data typically does not exceed 10 digits, while the precision of newly generated numbers by the algorithm consistently remains below 8 digits. Based on research findings[10], it is established that these data representations require a maximum of 31 significant digits in mantissa. Consequently, we opt to employ a bias value of 21 to represent the length of mantissa, which is "mask-21" and need 5 bits to represent it. Then, we use 1 bit to indicate whether re-erasing occurs. Next, we write the number of leading zeros in xor_{stored} as represented by Equation 6, followed by writing the remaining part, namely the center bits.

4.3 SMC Algorithm

During the research of the SXC algorithm, we discovered that, similar to greedy algorithms, the current optimal solution is not necessarily globally optimal. Furthermore, although SXC generates the shortest XOR values, it requires a relatively large number of flag bits for storage. To address this, we propose the SMC algorithm, which is encoded using shorter mantissa and balanced with a smaller total number of flag bits. This approach focuses on XOR mantissa and further optimizes the encoding scheme, resulting in improved efficiency.

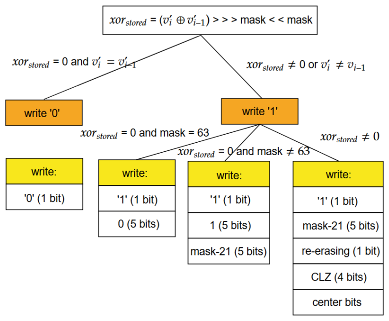
xor_{stored} = $(v'_i \oplus v'_{i-1})$ > > > mask < < mask
xor_{stored} = 0 and $v'_i = v'_{i-1}$
$xor_{stored} \neq 0$ or $v'_i \neq v_{i-1}$
write '0'
write '1'
xor_{stored} = 0 and mask = 63
xor_{stored} = 0 and mask ≠ 63
$xor_{stored} \neq 0$
write:
'0' (1 bit)
write:
'1' (1 bit)
0 (5 bits)
write:
'1' (1 bit)
1 (5 bits)
mask-21 (5 bits)
write:
'1' (1 bit)
mask-21 (5 bits)
re-erasing (1 bit)
CLZ (4 bits)
center bits

Fig. 3: SXC compressor.

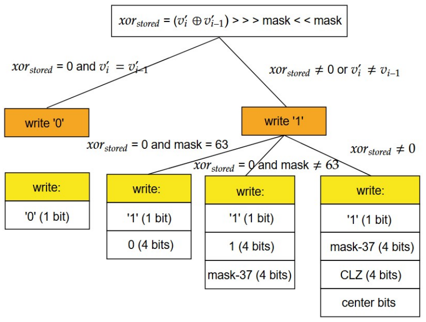
xor_{stored} = $(v'_i \oplus v'_{i-1})$ > > > mask < < mask
xor_{stored} = 0 and $v'_i = v'_{i-1}$
$xor_{stored} \neq 0$ or $v'_i \neq v_{i-1}$
write '0'
write '1'
xor_{stored} = 0 and mask = 63
xor_{stored} = 0 and mask ≠ 63
$xor_{stored} \neq 0$
write:
'0' (1 bit)
write:
'1' (1 bit)
0 (4 bits)
write:
'1' (1 bit)
1 (4 bits)
mask-37 (4 bits)
write:
'1' (1 bit)
mask-37 (4 bits)
CLZ (4 bits)
center bits

Fig. 4: SMC compressor.

Exponent Unchanged The SXC algorithm selects the shortest mantissa in such cases but incurs an additional 1-bit overhead to indicate whether re-erasing is required. Upon analyzing SXC's strategy, we observed that the two choices for the mantissa length differ by only 1 bit. By opting for the slightly longer mantissa directly, the space saved by eliminating the re-erasing flag bit offsets the additional cost of the longer mantissa. This adjustment not only simplifies the operation but also significantly reduces computational overhead, leading to considerable time savings.

Exponent Changed In this scenario, it is straightforward to identify at least one value with all '0's mantissa. Given our focus on achieving shorter XOR mantissa, we avoid the complexity of determining the optimal choice among them. Instead, we simplify the process by directly selecting v^+ and setting its mantissa to zero. This approach streamlines the operation while maintaining efficiency. The pseudocode of the SMC is shown in Algorithm 2.

Encoding scheme As showin in Figure 4, to limit the length of the XOR mantissa, we use a bias value of 37 (referred to as "mask-37"), which requires 4 bits to represent, for larger error tolerances. For smaller error tolerances, we retain the original bias value of 21, which needs 5 bits. Additionally, the re-erasing flag bit has been removed to further improve the encoding scheme. Based on Section 1 and Section 2, SXC achieved shortest XOR mantissa in Case $xor_{stored} \neq 0$ in Section 1. In this case, the trailing zero length of the XOR generated by SMC can be at most 1 bit shorter than that of SXC. However, SMC does not need to store "re-erasing", which means when considering the storage for trailing zeros and "re-erasing" together, the space required by SMC is no more than that required by SXC. In other words, shorter trailing zeros and flag bits storage.

5 Experiments

In this section, SXC and SMC are evaluated against Elf+, Sim-Piece+, and Mix-Piece with considerable improvements in compression ratios and execution speeds across 90 diverse datasets. On smaller error tolerances, SMC outperforms the other algorithms with up to 17.7% improvement in compression ratio, 10.67 times faster in compression speed, and 7.65 times faster in decompression speed.

5.1 Datasets and Experimental Setting

Datasets To evaluate SXC and SMC, we selected a wide range of datasets from the UCR time series archive [2] (Table 2). For consistency, we set the data block size to 500 points and applied the same "ignore" strategy across all algorithms, minimizing the impact of different strategies. Elf [6] also used this approach, loading data in blocks and ignoring the remainder.

Baselines

We compare SXC and SMC with the lossy methods Sim-Piece+ [3] and Mix-Piece [4], as well as the lossless method Elf+ [5]. These are the latest optimized versions of Sim-Piece [3] and Elf [6], ensuring a fair comparison.

Among the most recent max-error bound compression algorithms, Sim-Piece+ [3] and Mix-Piece [4] stand out for their excellent performance. To ensure concise and convincing experimental evaluations, we selected these two algorithms for comparison, along with Elf+ [5], a recent lossless algorithm known for its outstanding compression ratio, as a benchmark.

Metrics Performance is evaluated based on compression ratios, execution time, Mean Absolute Error (MAE), and Root Mean Square Error (RMSE):

Mean Absolute Error (MAE) measures the average absolute difference between compressed and original values, indicating how closely the compressed data matches the original. Lower MAE values signify better performance.

Root Mean Square Error (RMSE) is the square root of the average squared differences, penalizing larger errors more heavily. Lower RMSE values indicate better performance, especially when minimizing large errors is critical.

$$\mathrm{MAE} = \frac{1}{n}\sum_{i=1}^{n}|v_i - v_i'|, \quad \mathrm{RMSE} = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(v_i - v_i')^2}, \quad \mathrm{CR} = \frac{\mathrm{CompressedSize}}{\mathrm{OrigionalSize}} \tag{7}$$

In Equation 7, v_i and v_i' denote the original and compressed values, respectively, and n is the total number of values.The Compression Ratio (CR) is the ratio of compressed to original data size, where a smaller CR indicates more efficient space savings.

Settings All algorithms were implemented in Java (JDK 1.8) and executed on a desktop with an AMD (TM) R7 4800H CPU and 16G memory. During the testing process, each algorithm was run up to 10 times to obtain the averaged results. The maximum error tolerances are set to be 0.005%, 0.01%, 0.05%, and 0.1% of the range of values of the test datasets, respectively.

Table 2: Order numbers and lengths of our 90 time series datasets.

Data set (Order number)	**length**	**Data set(Order number)**	**length**
ACSF1(1)	146000	Lightning2(46)	38220
Adiac(2)	68640	Lightning7(47)	22330
AllGestureWiimoteX(3)	150000	Mallat(48)	56320
AllGestureWiimoteY(4)	150000	MedicalImages(49)	37719
AllGestureWiimoteZ(5)	150000	MelbournePedestrian(50)	28656
CBF(6)	3840	MiddlePhalanxOutlineAgeGroup(51)	32000
ChlorineConcentration(7)	77522	MiddlePhalanxOutlineCorrect(52)	48000
Computers(8)	180000	MiddlePhalanxTW(53)	31920
CricketX(9)	117000	MixedShapesRegular(54)	512000
CricketY(10)	117000	MixedShapesSmall(55)	102400
CricketZ(11)	117000	NonInvasiveFetalECGThorax2(56)	1350000
DistalPhalanxAgeGroup(12)	32000	OSULeaf(57)	85400
DistalPhalanxCorrect(13)	48000	PhalangesOutlinesCorrect(58)	144000
DistalPhalanxTW(14)	32000	Phoneme(59)	219136
DodgerLoopDay(15)	22464	PigAirwayPressure(60)	208000
DodgerLoopGame(16)	5760	PigArtPressure(61)	208000
DodgerLoopWeekend(17)	5760	PigCVP(62)	208000
Earthquakes(18)	164864	Plane(63)	15120
ECG200(19)	9600	PowerCons(64)	25920
ECG5000(20)	70000	ProximalPhalanxOutlineAgeGroup(65)	32000
EOGHorizontalSignal(21)	452500	ProximalPhalanxOutlineCorrect(66)	48000
EOGVerticalSignal(22)	452500	ProximalPhalanxTW(67)	32000
EthanolLevel(23)	882504	RefrigerationDevices(68)	270000
FaceAll(24)	73360	Rock(69)	56880
FaceFour(25)	8400	ScreenType(70)	270000
FacesUCR(26)	26200	SemgHandGenderCh2(71)	450000
FiftyWords(27)	121500	SemgHandMovementCh2(72)	675000
Fish(28)	81025	SemgHandSubjectCh2(73)	675000
FordA(29)	1800500	ShapeletSim(74)	10000
FordB(30)	1818000	ShapesAll(75)	307200
FreezerRegular(31)	45150	SmallKitchenAppliances(76)	270000
GestureMidAirD1(32)	74880	StarLightCurves(77)	1024000
GestureMidAirD2(33)	74880	Strawberry(78)	144055
GestureMidAirD3(34)	74880	SwedishLeaf(79)	64000
GesturePebbleZ1(35)	60060	SyntheticControl(80)	18000
GesturePebbleZ2(36)	66430	ToeSegmentation1(81)	11080
GunPointAgeSpan(37)	20250	TwoPatterns(82)	128000
GunPointMaleFemale(38)	20250	UWaveGestureLibraryX(83)	282240
GunPointOldYoung(39)	20400	UWaveGestureLibraryY(84)	282240
Ham(40)	46979	UWaveGestureLibraryZ(85)	282240
HandOutlines(41)	2709000	Wafer(86)	152000
Haptics(42)	169260	WordSynonyms(87)	72090
HouseTwenty(43)	80000	WormsTwoClass(88)	162900
InsectWingbeatSound(44)	56320	Worms(89)	162900
LargeKitchenAppliances(45)	270000	Yoga(90)	127800

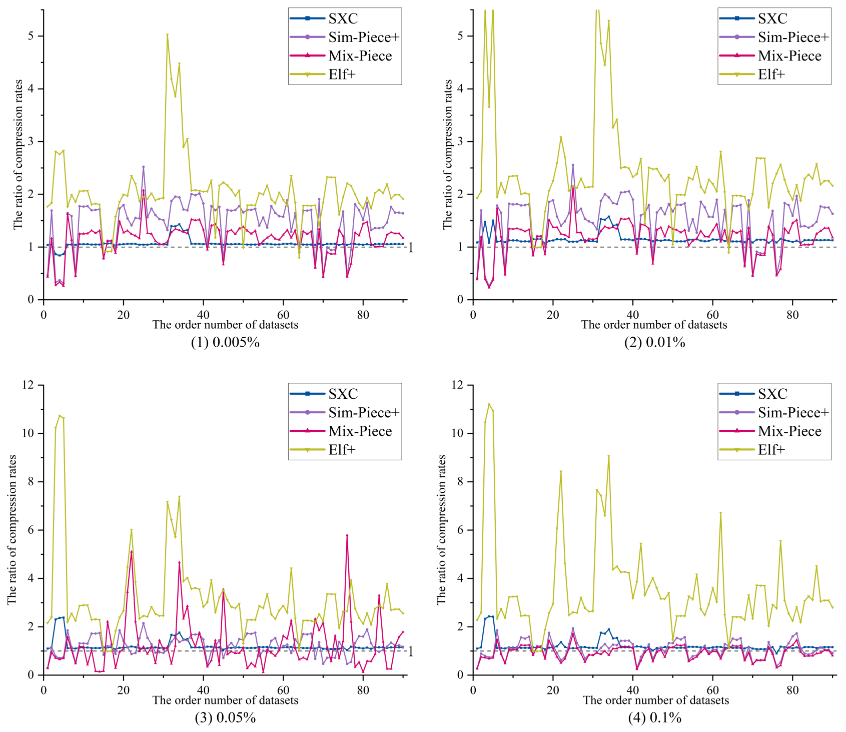

Fig. 5: Compression ratios of four algorithms divided by the compression ratio of SMC under four errors.

5.2 Compression Ratio and Data Quality

Comparisons on compression ratios are depicted at Figure 5 and Table 3. As the error tolerance increases to 0.1%, SMC reaches a Compression Ratio (CR) of 0.152, slightly higher than Mix-Piece 0.149, but still the second best among the methods. With further increases in error, Sim-Piece+ and Mix-Piece demonstrate stronger performance, outperforming SMC and SXC in more datasets and showing greater reductions in CR.

While Sim-Piece+ is less prominent at smaller error levels, it surpasses SXC at 0.1% error tolerance. Mix-Piece exhibits variability across datasets but maintains a better average CR than Sim-Piece+. Elf+, a lossless algorithm, performs comparably to Sim-Piece+ on some datasets and even outperforms certain lossy algorithms at smaller error tolerances.

Overall, CRs of the five algorithms show distinctive trends in various datasets. SMC and Mix-Piece demonstrate clear advantages in specific datasets, while Sim-Piece+, Mix-Piece and Elf+ are more sensitive to the dataset characteristics. In contrast, the CR of SXC and SMC are relatively stable.

Table 3: Compression ratio, compression time and decompression time across five algorithms under four error tolerances. Alg 1-5 refers to SMC, SXC, Sim-Piece+, Mix-Piece, and Elf+.

δ	Compression Ratio					Compression Time					Decompression Time				
	Alg 1	Alg 2	Alg 3	Alg 4	Alg 5	Alg 1	Alg 2	Alg 3	Alg 4	Alg 5	Alg 1	Alg 2	Alg 3	Alg 4	Alg 5
0.005%	**0.232**	0.246	0.352	0.274	0.457	**18.00**	62.66	69.52	205.08	25.51	**8.41**	9.06	59.52	73.25	18.27
0.01%	**0.204**	0.231	0.317	0.248		**18.91**	62.25	63.08	189.14		**7.88**	9.05	52.13	66.68	
0.05%	**0.168**	0.194	0.215	0.178		**16.27**	62.64	56.81	134.14		**7.51**	8.00	43.05	54.42	
0.1%	0.152	0.176	0.171	**0.149**		**6.62**	65.22	51.19	109.5		**7.49**	7.87	38.03	45.14	

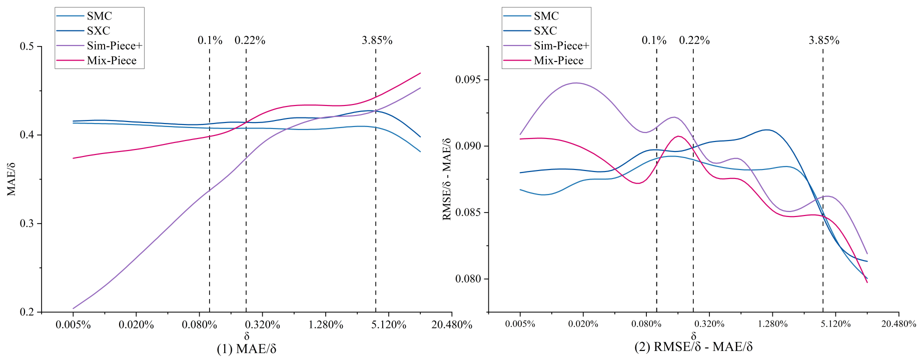

Fig. 6: MAE and RMSE- MAE of three lossy compression algorithm.

On certain datasets, SMC and SXC demonstrate particularly strong performance, raising questions about their impact on data quality. To evaluate this, we conducted experiments measuring MAE and RMSE for the four lossy algorithms across error tolerances from 0.005% to 10.24%, as shown in Figure 6.

The trends of MAE and RMSE-MAE are shown in plots (1) and (2), respectively. SMC and SXC exhibit stable MAE across varying error tolerances, indicating a uniform error distribution after compression. In contrast, Sim-Piece+ shows a significant increase in MAE as error tolerance grows, especially at higher thresholds (e.g., 3.85%), reflecting its sensitivity to small errors. Mix-Piece, while more stable than Sim-Piece+, still experiences rising MAE with increasing error tolerance, highlighting limitations of its PLA-based strategy. Plot (2) reveals that SMC and SXC are less affected by extreme errors, with relatively flat curves, whereas Sim-Piece+ and Mix-Piece exhibit larger fluctuations, particularly at smaller δ values, indicating greater sensitivity to extreme errors.

At error tolerances of 0.22% and 3.85%, Mix-Piece and Sim-Piece+ surpass SMC and SXC in MAE, with their MAE increasing rapidly beyond 3.85%. This trend aligns with changes in average compression ratios, as larger MAE values in PLA-based algorithms often correlate with better compression rates.

In conclusions, SMC and SXC demonstrate high stability across error tolerances. Figure 6 showing their resilience to extreme errors and strong generalization ability. These qualities make them suitable for a wide range of applications. While Mix-Piece is more stable than Sim-Piece+, both are limited by smaller error tolerances, where they are more susceptible to extreme errors. Therefore, SMC and SXC excel in tasks requiring stability and robustness, whereas Sim-Piece+ and Mix-Piece may offer advantages in specific low-error scenarios but lack robustness.

5.3 Execution Time

Table 3 shows the average compression and decompression times for the five algorithms across error tolerances from 0.005% to 0.1%, measured in μs in each block.

SMC consistently achieves the shortest compression times, outperforming other algorithms. This advantage persists at larger error tolerances, where Mix-Piece and Sim-Piece+ exhibit significantly longer compression times. SXC's advantage diminishes and is eventually surpassed by Sim-Piece+. Elf+, a lossless algorithm, requires more time for compression (e.g., 25.51 μs at $\delta = 0.005\%$), reflecting the higher computational cost of lossless methods.

On the decompression side, SMC and SXC maintain high efficiency, significantly lower decompression times compared to Sim-Piece+, Mix-Piece, and Elf+. For example, at $\delta = 0.005\%$,SMC takes 8.41 μs, while Mix-Piece requires 59.52 μs, and Elf+ takes 18.25 μs. SMC and SXC show stable decompression times across error tolerances, highlighting their efficiency. Sim-Piece+ and Mix-Piece exhibit higher decompression times, especially as error tolerance increases. Elf+, despite being a lossless algorithm, demonstrates efficient decompression times, unlike typical lossless methods that require more time for complex reconstruction.

In summary, SMC and SXC are the most time-efficient algorithms in both compression and decompression, while Mix-Piece and Sim-Piece+ show significantly slower performance, especially in compression. Elf+, though offering lossless compression and competitive speed, highlights the trade-off between compression ratio and speed inherent to lossless methods.

6 Conclusion

In this paper, we propose two novel lossy floating-point data compression algorithms that leverage XOR encoding, providing impressed results with quality assurance. Compared to existing established and effective lossy and XOR-based lossless floating-point compression algorithms, SXC algorithm achieves better space savings than many algorithms under smaller error tolerances. Meanwhile, SMC algorithm excels in compression ratio across smaller error tolerances and maintains stable, high performances. Furthermore, SMC consistently delivers the best compression and decompression speeds. In addition to impressive compression ratios, stable in MAE and RMSE values, SXC and SMC have demonstrated strong reliability and robustness.

Acknowledgments. We would like to thank the authors of [6,3,4] for generously giving us their source code for the comparison tests in this paper. We also thank the data providers of [2] for the testing data sets. This research was supported by Zhejiang Provincial Natural Science Foundation of China under Grant No. LY24F020021, and the Ningbo Science and Technology Special Projects under Grant No. 2024Z263, 2024H030, and 2022Z235, and Suzhou Science and Technology Development Planning Programme under Grant No. ZXL2023176.

References

1. Ieee standard for floating-point arithmetic. IEEE Std 754-2019 (Revision of IEEE 754-2008) pp. 1–84 (2019)
2. Dau, H.A., Keogh, E., Kamgar, K., Yeh, C.C.M., Zhu, Y., Gharghabi, S., Ratanamahatana, C.A., Yanping, Hu, B., Begum, N., Bagnall, A., Mueen, A., Batista, G., Hexagon-ML: The ucr time series classification archive (October 2018), `https://www.cs.ucr.edu/~eamonn/time_series_data_2018/`, last accessed 2024/5/20
3. Kitsios, X., Liakos, P., Papakonstantinopoulou, K., Kotidis, Y.: Sim-piece: Highly accurate piecewise linear approximation through similar segment merging. Proc. VLDB Endow. **16**(8), 1910–1922 (2023)
4. Kitsios, X., Liakos, P., Papakonstantinopoulou, K., Kotidis, Y.: Flexible grouping of linear segments for highly accurate lossy compression of time series data. VLDB J. **33**, 1569–1589 (2024), `https://api.semanticscholar.org/CorpusID:271234506`
5. Li, R., Li, Z., Wu, Y., Chen, C., Guo, S., Zhang, M., Zheng, Y.: Erasing-based lossless compression method for streaming floating-point time series. arXiv preprint arXiv:2306.16053(2023) (2023)
6. Li, R., Li, Z., Wu, Y., Chen, C., Zheng, Y.: Elf: Erasing-based lossless floating-point compression. Proc. VLDB Endow. **16**(7), 1763–1776 (2023)
7. Liakos, P., Papakonstantinopoulou, K., Kotidis, Y.: Chimp: efficient lossless floating point compression for time series databases. Proc. VLDB Endow. **15**(11), 3058–3070 (2022)
8. Liang, X., Zhao, K., Di, S., Li, S., Underwood, R., Gok, A.M., Tian, J., Deng, J., Calhoun, J.C., Tao, D., Chen, Z., Cappello, F.: Sz3: A modular framework for composing prediction-based error-bounded lossy compressors. IEEE Transactions on Big Data **9**(2), 485–498 (2023)
9. Lindstrom, P.: Fixed-rate compressed floating-point arrays. IEEE Transactions on Visualization and Computer Graphics **20**(12), 2674–2683 (2014)
10. Liu, C., Jiang, H., Paparrizos, J., Elmore, A.J.: Decomposed bounded floats for fast compression and queries. Proc. VLDB Endow. **14**(11), 2586–2598 (2021)
11. Pang, C., Zhang, Q., Zhou, X., Hansen, D.P., Wang, S., Maeder, A.J.: Computing unrestricted synopses under maximum error bound. Algorithmica **65**(1), 1–42 (2013)
12. Pelkonen, T., Franklin, S., Teller, J., Cavallaro, P., Huang, Q., Meza, J., Veeraraghavan, K.: Gorilla: a fast, scalable, in-memory time series database. Proc. VLDB Endow. **8**(12), 1816–1827 (2015)
13. Xie, Q., Pang, C., Zhou, X., Zhang, X., Deng, K.: Maximum error-bounded piecewise linear representation for online stream approximation. VLDB J. **23**(6), 915–937 (2014)
14. Zhao, H., Pang, C., Kotagiri, R., Pang, C.K., Deng, K., Yang, J., Li, T.: An optimal online semi-connected pla algorithm with maximum error bound. IEEE Trans. on Knowl. and Data Eng. **34**(1), 164–177 (2022)