On a workaround for an overflow in streaming process mining

Reynald Eugénie, Averil Prost, Seth Taylor, Rita Zantout

On a workaround for an overflow in streaming process mining

Reynald Eugénie, Averil Prost, Seth Taylor and Rita Zantout

15-19 May 2023
Semaine d'Étude Maths-Entreprise (SEME), Pointe-à-Pitre, Guadeloupe

Contents

This document serves as a report of a SEME project held in Pointe-à-Pitre, Guadeloupe, during June 2023. The authors wish to warmly thank the AMIES, the organisers and SoftBridge Technology for their confidence and efficient support.

1 Introduction

1.1 Context

It is a sacred rule in programming that each action must leave a trace. Computers and communication devices are constantly recording their states and various failures in *log files*, with a high level of detail. These flows of information stay widely unexploited, due to their technical complexity and overwhelming volume. However, they contain invaluable *a posteriori* observations of the interactions of applications, in particular regarding the eventual flaws or malfunctions.

The intelligent treatment of log files is the starting point of *process mining*. This recent field is developed both in the academic community and in the industry. In the latter case, the target technique is the *streaming process mining*, in which the generated data is immediately handled. This is the main service offered by Softbridge Technology, through a platform implemented in Java.

One of the aims of Softbridge is to detect latency in a process. The proposed method implies to approximate the distribution of durations of the events contributing to this process, such that outliers may be identified. However, the choice of an on-line treatment of incoming data puts a constraint on the representation of these durations, since *each event must be processed only once*. Consequently, some basic-but-efficient methods such as histogram representations are ill-adapted, since the range of datas is not known *a priori*. Instead, one can use a specific class of parametric representations, under the assumption that the parameters may be updated on-line. It is the case, for instance, of the generic estimators of the mean and standard deviation of a given probability law (see Section 1.2).

This project arises from a technical difficulty encountered in the implementation of the parametric approach. The duration of a process, denoted x_i, is expressed in milliseconds (following the classical POSIX representation). Some processes may run up to 10 months, and the associated x_i becomes so large that x_i^2 overflows the memory allocated by the primitive type `long` of Java. The language implements a class `BigInteger` that virtually achieves unbounded precision, but at the expense of a greater computational cost and possible loss of portability. The purpose of this document is to find the most efficient workaround, respecting the constraints of streaming estimation, using only primitive types, and maximizing the precision.

1.2 Formulation of the problem

In the sequel, the processes will be observed only through their duration in milliseconds, generically denoted x_i. Suppose that N observations are available, and denote

$$\mu_N = \frac{1}{N}\sum_{i=1}^{N} x_i\,, \qquad S_N = \frac{1}{N}\sum_{i=1}^{N} x_i^2\,,$$

as the mean and mean of the squares respectively. Our aim is to parametrize the distribution of processes by either a normal or a log-normal law. Recall that the normal and log-normal distributions are given by

$$f[\mu,\sigma](x) = \frac{1}{\sqrt{2\pi}\sigma}\exp\left(-\frac{(x-\mu)^2}{2\sigma^2}\right), \qquad \text{and} \qquad g[\mu,\sigma](x) = \frac{1}{x\sqrt{2\pi}\sigma}\exp\left(-\frac{(\ln(x)-\mu)^2}{2\sigma^2}\right),$$

An on-line estimation of the relevant moments to compute and plot the distribution is given by the following algorithm:

Algorithm 1: Naive algorithm

```
1  Set N = 0, μ_0 = 0 and S_0 = 0.
2  while the program is running do
3  |  Get a new duration x_{N+1}.
4  |  Update the mean by μ_{N+1} ≔ (N × μ_N + x_{N+1}) /(N + 1).
5  |  Update the sum of squares by S_{N+1} ≔ (N × S_N + x²_{N+1}) /(N + 1).
6  |  if the law is evaluated then
7  |  |  Compute σ_{N+1} = √(S_{N+1} − μ²_{N+1}).
8  |  |  Compute and plot the chosen distribution.
9  |  end
10 end
```

In practice, $N \cdot S_N$ is the value stored and input into the formula for the standard deviation σ. This poses two issues associated with overflow in this algorithm, both occurring at line 5.

Problem 1. *If the process has duration $x_i > 2^{32} - 1$ then overflow will occur when squaring.*

Problem 2. *The sum of squares $N \cdot S_N$ can become unbounded and will cause overflow if stored directly.*

2 Proposed solutions

In theory, Java has a non-primitive type called `BigInteger` which allows for an adaptivity from the `int` type into a `long` type when an overflow is detected. In practice however, this solution is not satisfactory since this type is not generic and its storage in memory is not necessarily controlled. Furthermore, Softbridge seeks a solution methodology which is independent of the language and equipment. We thus propose two solutions which address the problems posed by Problems 1 and 2 which use only integer or floating-point encodings. Each method is explained and detailed in the following sections. We then characterize the possible advantages and drawbacks realized using these approaches.

2.1 Emulation of the behavior of `BigInteger`

The first proposed approach consists in emulating the operations of the `BigInteger` type using only operations on the `int` type until the variance is accessed. The central idea behind this approach is to use modular arithmetic to have a larger representation of an integer greater than $2^{63}-1$. We consider an emulation of the behavior of the `BigInteger` type of Java, by decomposing the duration using two values A, B, each stored as primitive types.

Following the Java convention of signed natural numbers, we define the following sets.

Definition 1 (Int64)**.** *A natural number x admits a representation as an `Int64` if it belongs to $[\![-2^{63}, 2^{63}-1]\!]$.*

Definition 2 (`MyBigInt`)**.** *A natural number x admits a representation as a `MyBigInt` if it is written as $x = A \times 2^e + B$, where A, B are Int64, and $e \in [\![6, 24]\!]$.*

The range defining the exponent will be clarified based on the operation carried out using this solution. Note that, the set of `MyBigInt` does not coincide with the set of numbers represented in Java as `BigInteger`.

Assumption 1 (Range of incoming data)**.** *In the following, we will assume that the incoming values $(x_i)_i$ are all exactly represented as Int64.*

As a consequence of Assumption 1, the mean values $(\mu_N)_N$ are all exactly represented in 64-bits integers.

Assumption 2 (Range of the squares)**.** *We will also assume that at any given N, the sum $\sum_{i=1}^{N} x_i^2$ has a representation as a* `MyBigInt`*.*

Let // denote the integer division, i.e. $a//b = \lfloor a/b \rfloor$, and mod be the modulo operation (for completeness, let us point that we consider the multiplication to have priority upon the modulo, and denote $ab \bmod c = (ab) \bmod c$).

Remark 1 (Range of validity of the assumptions)**.** *Assumptions 1 and 2 are based on the following computations: a process lasting 4 years will take*

$$\alpha = 4 \times 365 \times 24 \times 3600 \times 1000 = 126144000000 \simeq 2^{36} \ ms$$

to complete. This number still admits a representation as an 64-bits integer. On the other hand, the achievable precision of a `MyBigInt` *is of the order of magnitude of $(2^{2e}-1)2^{2e} \simeq 2^{4e}$. Hence, in the worst case where all the processes last 4 years and taking $e = 24$, the number of processes N such that $N \times \alpha$ overflows the* `MyBigInt` *representation is at least*

$$2^{4\times e} // \alpha^2 + 1 = 4\,979\,048.$$

In comparison, the number of 4-year processes needed for the sum of squares to overflow the `double` *representation is around 10^{285}. We consider this to be a reasonable amount.*

The manipulation of `MyBigInt` relies on the following lemma.

Lemma 1 (Operations)**.** *Let $x \in \mathbb{R}^+$ be given by $x = a2^e + b$, where a and b belong to $[\![0, 2^e - 1]\!]$. Then its square x^2 may be written as $A \times 2^{2e} + B$, where*

$$A = a^2 + \left(2ab2^e + b^2\right) // 2^{2e}, \qquad \text{and} \qquad B = (2ab2^e \bmod 2^{2e} + b^2) \bmod 2^{2e}. \tag{1}$$

Let now $x_i = A_i 2^{2e} + B_i$, with $A_i, B_i \in [\![0, 2^{2e} - 1]\!]$ and $i \in \{0, 1\}$.

1) *The sum $x_0 + x_1$ writes $A2^{2e} + B$, where $A = A_0 + A_1 + (B_0 + B_1)//2^{2e}$ and $B = (B_0 + B_1) \bmod 2^{2e}$.*
2) *Assume that $x_0 \geqslant x_1$, and denote $s = 0$ if $B_0 \geqslant B_1$, and $s = 1$ if $B_0 < B_1$. The difference $x_0 - x_1$ writes as $A2^{2e} + B$, where $A = A_0 - A_1 - s$, and $B = B_0 - B_1 + s\,2^{2e}$.*
3) *Let $n \in \mathbb{N}$ such that nB_0 can be represented as an* `Int64`*. Then $y = nx_0$ may be represented as $C2^{2e} + D$, where $C = nA_0 + (nB_0)//2^{2e}$, and $D = (nB_0) \bmod 2^{2e}$.*

Remark 2 (Choice of the exponent)**.** *When calculating $2a_ib_i2^e$, there is a risk of overflowing. We can circumvent this issue by considering an adapted value for the exponent e. If we assume that any given duration is less than four year ($\sim 2^{37}$) then we have that $a \in [0, 2^{37-e}] \implies a^2 \in [0, 2^{74-2e}]$. Since we require $74 - 2e < 63$, as a lower bound we have $e > 5$. As an upper bound, we note that since $a \in [0, 2^{37-e}]$ and $b \in [0, 2^e - 1]$ then $2 \cdot a \cdot b \cdot 2^e \in [0, 2^{38+e}]$ hence we require $e < 25$. In practice we take $e \in [6, 24]$.*

Proof. Notice that

$$x^2 = (a2^e + b)^2 = a^2 2^{2e} + 2ab2^e + b^2.$$

Recall that $ab \bmod c = ((a \bmod c)(b \bmod c)) \bmod c$. The expression (1) follows from

$$B = x^2 \bmod 2^{2e} = \left(2ab2^e + b^2\right) \bmod 2^{2e} = \left((2ab2^e) \bmod 2^{2e} + b^2\right) \bmod 2^{2e},$$

and, since all the terms are nonnegative,

$$A = x^2 // 2^{2e} = \left\lfloor \frac{a^2 2^{2e} + 2ab2^e + b^2}{2^{2e}} \right\rfloor = a^2 + \left(2ab2^e + b^2\right) // 2^{2e}.$$

Item 1) is straightforward. Item 2) may be treated by disjunction, and Item 3) follows the same reasoning as Item 1). □

To completely re-implement the naive algorithm Algorithm 1, one needs to have an approximation of the square root of a natural number $x = A2^{2e} + B$. Our implementation amounts to compute

$$\sqrt{x} \simeq \sqrt{A + B/2^{2e}} \times 2^e, \tag{2}$$

where the first part is encoded as a `double`. A lot of submethods may arise from this point, by considering the order of operations and the point to which the encoding falls back into double. The current implementation turns B into a double, then performs the division $B/2^{2e}$ in floating-point, and converts A to a double on the way.

Remark 3 (Alternative)**.** *Historically, we considered an approximation as* $\sqrt{A_0 * 2^{2e}/(A_0+1) + B_0/(A_0+1)} \times \sqrt{A_0+1}$ *in order to reduce the magnitude of the stored numbers, but the results turn out to be unsatisfactory (see Section 3.2).*

Altogether, our first solution strategy can be implemented as the following pseudocode algorithm:

Algorithm 2: Emulation of `BigInteger` algorithm

Set $N = 0$, $\mu_0 = 0$ and $S_0 = 0$.
while *the program is running* **do**
  Get a new duration x_{N+1}.
  Update the mean by $\mu_{N+1} \coloneqq (N \times \mu_N + x_{N+1})/(N+1)$.
  Decompose x_{N+1}^2 in `MyBigInt` representation.
  Perform the sum $S_{N+1} \coloneqq S_N + x_{N+1}^2$ in `MyBigInt` representation using the formula (1).
  if *the law is evaluated* **then**
    Decompose $(N+1) \times \mu_{N+1}^2$ in the `MyBigInt` representation (using a truncation).
    Perform the difference $A2^{2e} + B \coloneqq S_{N+1} - (N+1)\mu_{N+1}^2$ in `MyBigInt`.
    if $A = 0$ **then**
      Compute directly the square root of B,
    else
      Compute the square root as in (2).
    end
    Divide by $\sqrt{N+1}$ to obtain σ.
    Compute and plot the chosen distribution.
  end
end

2.2 Re-scaling using floating-point encoding

As a second proposed solution, as initially hinted by Softbridge, we consider a dynamic re-scaling of the sum of squares. Let $\lambda > 0$ denote our re-scaling parameter and let $\sigma_\lambda^2 \coloneqq \lambda\sigma^2$. A re-scaled distribution takes the equivalent form

$$f[\mu,\sigma](x) = \frac{1}{\sqrt{2\pi}\sigma}\exp\left(-\frac{(x-\mu)^2}{2\sigma^2}\right) = \frac{\sqrt{\lambda}}{\sqrt{2\pi}\sigma_\lambda}\exp\left(-\lambda\frac{(x-\mu)^2}{2\sigma_\lambda^2}\right) =: f_\lambda[\mu,\sigma_\lambda](x).$$

In the above expression, λ may be dynamically chosen small in order to keep the value σ_λ in ranges of values that may be represented in floating-point encoding (see 4) with sufficiently high precision. Note here that the terms λx_i^2 have no reason to stay natural numbers and that by taking $\lambda = 1$ we recover the case where each number is simply turned into a float.

Remark 4 (Double-Precision)**.** *The IEEE standard floating-point encoding of double precision uses 64 bits, broken down into the following sections:*

- *a first sign bit s.*
- *11 bits of exponent, encoding a number $p \in [\![0, 2^{11}-1]\!]$.*
- *52 bits of mantissa, that we will label in decreasing order as $b_{51}, b_{50}, b_{49}, \cdots, b_0$, encoding a multiplier $m \in [1,2[$ by $m = 1 + \sum_{i=1}^{52} b_{52-i}2^{-i}$.*

The floating-point double will then be represented as

$$x = (-1)^s \times m \times 2^{p-1023} = (-1)^s \times \left(1 + \sum_{i=1}^{52} b_{52-i}2^{-i}\right) \times 2^{p-1023}.$$

We will say that a real number x is encodable *if it may exactly be represented in double floating-point encoding. Given a fixed exponent p, the encodable values are equally distributed in the interval $[-(2-2^{-52})2^{p-1023}, (2-2^{-52})2^{p-1023}]$, with a step of $2^{-52} \times 2^{p-1023}$.*

Re-scaled operations

The computation of the mean remains unchanged, however the computation of the sum of squares becomes

$$S_{N+1}^\lambda = \frac{N \times S_N^\lambda + \lambda\, x_{N+1}^2}{N+1} = \frac{N}{N+1} \times S_N^\lambda + \frac{1}{N+1}\left(\sqrt{\lambda}\, x_{N+1}\right)^2.$$

The computation of the re-scaled deviation at step N will be

$$\sigma_\lambda^2 = S_N^\lambda - (\sqrt{\lambda} \times \mu_N)^2.$$

The last modification of the naive algorithm will intervene in the evaluation of the law, that has to take into account the parameter λ. The modification will depend on the particular expression of the distribution. The correct expression of the re-scaled normal and log-normal laws are the following:

$$f_\lambda[\mu, \sigma_\lambda](x) \coloneqq \frac{\sqrt{\lambda}}{\sigma_\lambda \sqrt{2\pi}} \exp\left(-\lambda \frac{(x-\mu)^2}{2\sigma_\lambda^2}\right) \qquad \text{and} \qquad g_\lambda[\mu, \sigma_\lambda](x) \coloneqq \frac{\sqrt{\lambda}}{x\sigma_\lambda \sqrt{2\pi}} \exp\left(-\lambda \frac{(\ln(x)-\mu)^2}{2\sigma_\lambda^2}\right). \tag{3}$$

Implementation as an on-line algorithm

Given the wide range of possible durations, the choice of λ is somewhat arbitrary. We consider two possible implementations:

- Fixing λ *a priori*, allowing us to detect whether a small or a large value of λ is well-suited.
- Update λ dynamically as the inverse of the current maximum of x_i, or the current maximum of x_i^2, reducing the order of magnitude of the data.

If we denote λ_N as the choice of λ at step N, an algorithm for the proposed second solution strategy is as follows:

Algorithm 3: Re-scaling algorithm

```
Set N = 0, μ_0 = 0, S_0 = 0 and λ_0 = 1.
while the program is running do
    Get a new duration x_{N+1}.
    Update the mean by μ_{N+1} ≔ N/(N+1) × μ_N + x_{N+1}/(N+1).
    Update the scaling parameter λ_{N+1} (while still storing λ_N).
    Update the sum of squares by S_{N+1}^{λ_{N+1}} ≔ S_N^{λ_N} × λ_{N+1}/λ_N + λ_{N+1} x_{N+1}^2.
    if the law is evaluated then
        Compute σ_{N+1} = sqrt(S_{N+1}^{λ_{N+1}}/(N+1) − λ_{N+1} μ_{N+1}^2).
        Compute and plot the re-scaled chosen distribution as in (3).
    end
end
```

3 Numerical results

3.1 Testing environment

In order to assess the different proposed solutions, we designed a range of test cases. We divide them into deterministic test cases, in which every parameter is analytically known at every time, at the expense of being non-realistic, and random test cases that mimic the industrial context. A test case is determined by

1. a method to provide the sequence $(x_i)_{i\in\mathbb{N}}$,
2. a set of expected parameters μ_N and σ_N in function of the number N of available processes.

Let $\hat{\mu}_N$ (resp. $\hat{\sigma}_N$) be the numerical approximation of μ_N (resp. σ_N). As a measure of the error for the proposed algorithms, we consider the following indicators:

- a measure of the overflow, equal to 0 if the program is completed without overflowing, and $+\infty$ otherwise,
- the relative error in mean $e_\mu \coloneqq |\hat{\mu}_N - \mu_N| / \mu_N$,
- the relative error in standard deviation $e_\sigma \coloneqq |\hat{\sigma}_N - \sigma_N| / \sigma_N$,

The reported results do not show problems of overflow, and we do not mention it.

Deterministic test cases

As deterministic test cases, we consider the following two trials

1. the constant test case: $x_i = \overline{x}$ a fixed value. The expected parameters are $\mu_N = \overline{x}$ and $\sigma_N = 0$ at any step N.
2. the cyclical test case: the values of x_i will cycle through a given finite set $\overline{X} = \{\overline{x}_1, \overline{x}_2, \cdots, \overline{x}_m\}$. Let $\overline{\mu}$ be the mean of $\overline{X}$, and $\overline{S} = \frac{1}{m}\sum_{i=1}^{m} \overline{x}_i^2$. The expected parameters are given by

$$\mu_N = \frac{1}{N}\sum_{i=1}^{N} x_i = \frac{N//m}{N} m\overline{\mu} + \frac{1}{N}\sum_{i=1}^{N\%m} \overline{x}_i, \qquad \text{and} \qquad \sigma_N = \sqrt{\frac{1}{N}\sum_{i=1}^{N} x_i^2 - \mu_N^2} = \sqrt{\frac{N//m}{N} m\overline{S} + \frac{1}{N}\sum_{i=1}^{N\%m} \overline{x}_i^2 - \mu_N^2}.$$

Random test cases

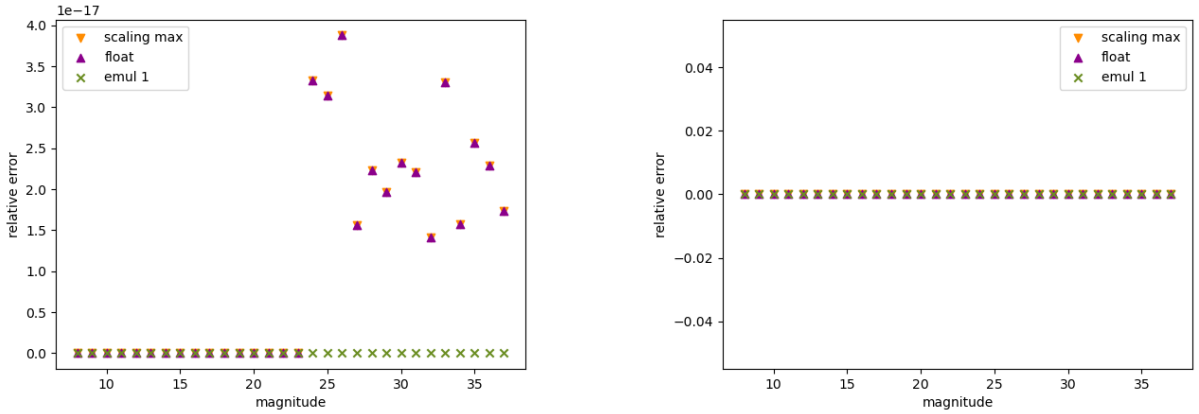
In the random case, the samples x_i are drawn from a given law. Consequently, the values of μ_N and σ_N will only be approximated by their numerical counterparts for a large number of samples. We keep the law fixed with respect to N, so that the expected parameters $\mu = \mu_N$ and $\sigma = \sigma_N$ are independent of N. We consider the following two trials.

1. The uniform discrete distribution $\mathcal{U}(1, K)$ between 1 ms and K ms, where K is even and is taken as a large duration of ≈ 10 months. The expected parameters are given by $\mu = (1 + K)/2$, and the standard deviation is $\sigma = \sqrt{\frac{k^2-1}{12}}$.
2. The normal distribution $\mathcal{N}(\mu, \sigma)$ for known mean and standard deviation.

3.2 Comparison of methods

In all the following, the number of simulated processes is taken as $N = 100$ for deterministic test cases, and $N = 10\,000$ for random ones. We compare the different methods against an implementation using `BigInteger`. The plotted errors are computed as the mean of relative errors over the last 10 iterations.

We begin by some very simple deterministic cases to assess the method. The first test case (represented in Figure 1) considers a constant sequence of durations $x_i \equiv \overline{x}$. The value of $\overline{x}$ is set to 2^i, with i varying in $[\![8, 38]\!]$ in order to make the sum of squares S overflow the `long` representation.



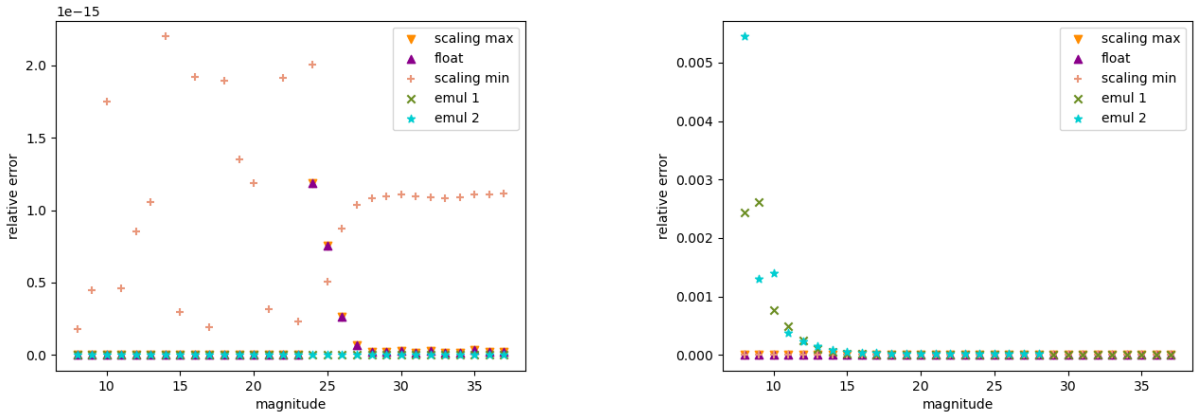
(a) (b)

Figure 1: Relative error in S (a) and σ (b), constant test case.

The x axis represents the natural number $i \in [\![8, 28]\!]$ such that $\overline{x} = 2^i$. The y axis shows the relative error on, respectively, the sum of the squares $S = \sum_{i=1}^{N} x_i^2$, and the standard deviation $\sigma = \sqrt{S/N - \mu^2}$. The method ***`scaling max`*** *corresponds to the choice of $\lambda_N = \max(\lambda_{N-1}, x_N)$, with $\lambda_0 = 0$. The method* ***`float`*** *corresponds to the constant choice of $\lambda = 1$. The method* ***`emul 1`*** *corresponds to the Algorithm 2, with the approximation of the square root performed as in (2). Unrepresented methods are giving far larger errors (see Tables 1 and 2).*

The second test case (see Figure 2) considers a sequence x_i alternating between 1 and the value 2^i. The i is chosen increasingly as before.

To conclude with analytically known test cases, the Figure 3 considers a periodic sequence over a fixed set of values, whose maximum is again set to 2^i, with i varying as before.



(a) (b)

Figure 2: Relative error in S (a) and σ (b), alternating in $\{1, 2^i\}$.

The series of data labeled ***`scaling min`*** *is an instance of the scaling algorithm for a fixed value $\lambda = 1e-12$. The series* ***`emul 2`*** *refers to the emulation of* ***`BigInteger`*** *behavior, with an implementation of the square root with a division by the largest factor (see Remark 3). The remaining labels are identical to Figure 1.*

To get closer to reality, we also considered random test cases. The error represented in the following figures is computed relatively to the mean and standard deviation of the laws that we sample, and the reader may keep in mind that part of the error may be due to the random sampling. Figure 4 presents the sampling of the uniform distribution in $[1, 2^i]$, with i increasing as previously.

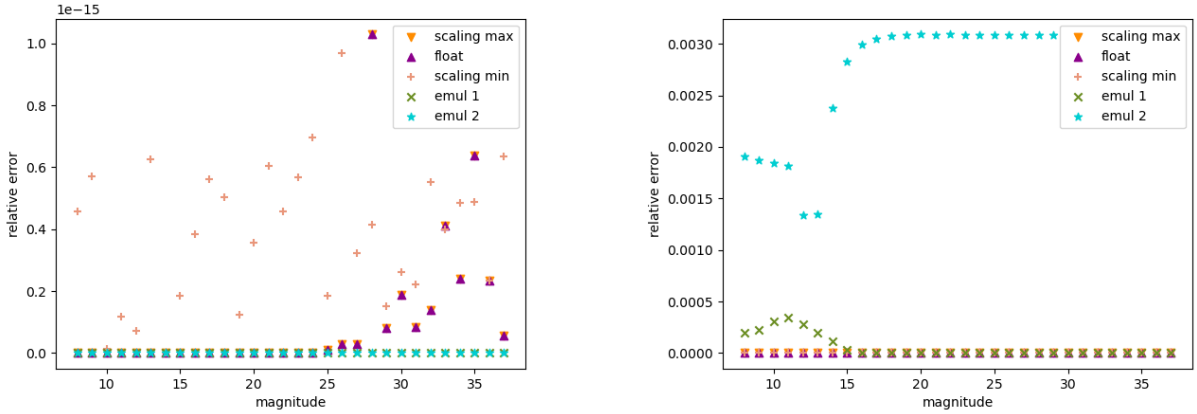
Figure 5 shows the sampling of a normal law of increasing mean and fixed standard deviation at 200. This case is the most relevant in the given context.

At first sight, it seems natural that the scaling method should give better results for a suitably adapted parameter λ, that would keep the stored quantities into reasonable intervals. To assess this, Figure 6 represents the variation of the relative error of S and σ for a given λ, ranging from 1 to 10^{-38}. The test case is chosen to be the alternation between 1 and 2^8, so that the error is not influenced by a random sampling.

The comparison is summarized with better precision in the following tables, in which the numbering of simulation coincide with that of the corresponding figures.

Simulation n°	error	scaling max	float	scaling min	emul 1	emul 2
1	max	3.88e-17	3.88e-17	4.61e-16	0.00e+00	0.00e+00
1	mean	1.12e-17	1.12e-17	4.30e-16	0.00e+00	0.00e+00
2	max	1.19e-15	1.19e-15	2.20e-15	0.00e+00	0.00e+00
2	mean	8.33e-17	8.33e-17	1.05e-15	0.00e+00	0.00e+00
3	max	1.03e-15	1.03e-15	9.70e-16	0.00e+00	0.00e+00
3	mean	1.06e-16	1.06e-16	3.87e-16	0.00e+00	0.00e+00
4	max	2.24e-02	2.24e-02	2.24e-02	0.00e+00	0.00e+00
4	mean	6.98e-03	6.98e-03	6.98e-03	0.00e+00	0.00e+00
5	max	7.64e-03	7.64e-03	7.64e-03	0.00e+00	0.00e+00
5	mean	6.20e-04	6.20e-04	6.20e-04	0.00e+00	0.00e+00

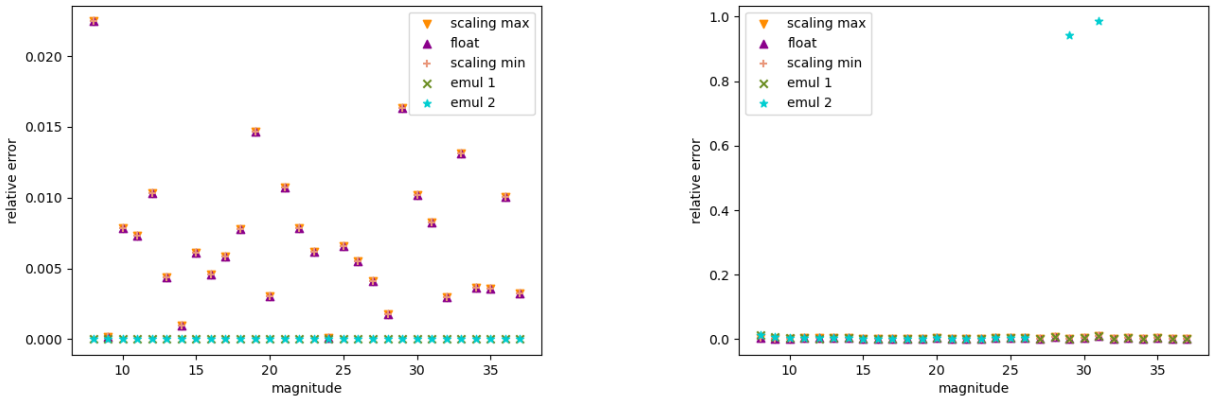
Table 1: Errors for the sum of squares S



(a) (b)

Figure 3: Relative error in S (a) and σ (b) alternating in a broader range of values.

For each $x \in [\![8, 37]\!]$*, the values are taken periodically in the set* $\{1000, 2^x, 6000, 3000, 5 \times 2^{x//2}\}$.

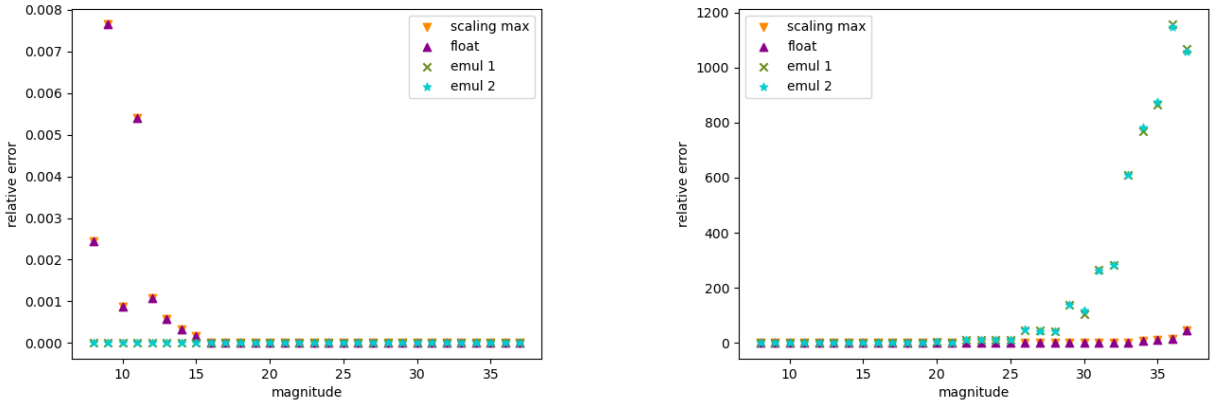


(a) (b)

Figure 4: Relative error in S (a) and σ (b), uniform distribution in $[1, 2^i]$.

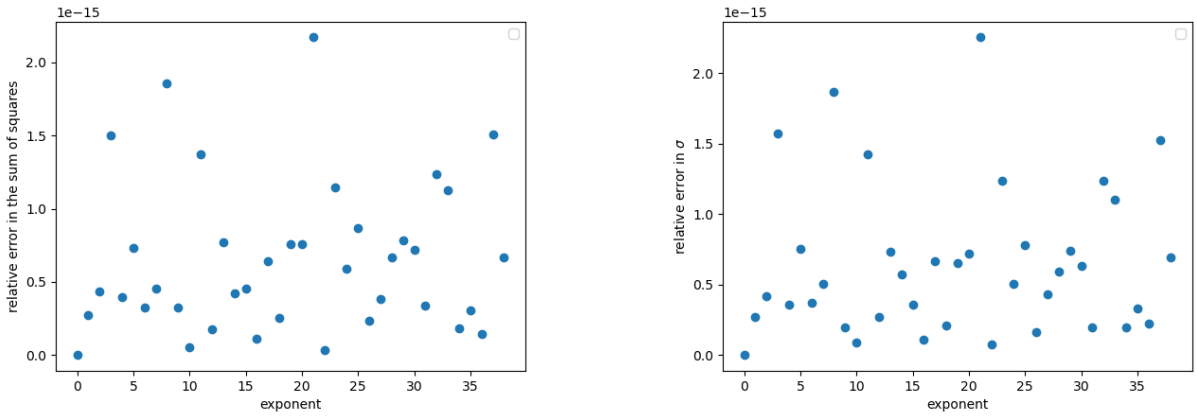
Simulation n°	error	scaling max	float	scaling min	emul 1	emul 2
1	max	0.00e+00	0.00e+00	2.90e+03	0.00e+00	0.00e+00
1	mean	0.00e+00	0.00e+00	1.93e+02	0.00e+00	0.00e+00
2	max	2.17e-07	2.17e-07	2.17e-07	2.61e-03	nan
2	mean	2.17e-08	2.17e-08	2.17e-08	2.25e-04	nan
3	max	9.21e-08	9.21e-08	9.21e-08	3.41e-04	nan
3	mean	5.26e-09	5.26e-09	5.26e-09	5.70e-05	nan
4	max	9.00e-03	9.00e-03	9.00e-03	1.35e-02	nan
4	mean	2.85e-03	2.85e-03	2.85e-03	3.56e-03	nan
5	max	4.75e+01	4.73e+01	5.49e+01	1.16e+03	1.15e+03
5	mean	2.82e+00	2.82e+00	3.14e+00	1.82e+02	1.82e+02

Table 2: Errors for the standard deviation σ



(a) (b)

Figure 5: Relative error in S (a) and σ (b), normal distribution with mean 2^i and standard deviation 200.



(a) (b)

Figure 6: Relative error in S (a) and σ (b), alternating test case and varying λ.

The scaling parameter λ is chosen as 10^{-x}, where $x \in [\![0, -38]\!]$ lies on the first axis.

Conclusion

Our project explored two methods to avoid overflowing in a stream process mining used by Softbridge technologies, namely

1. emulating `BigInteger` with `long` types: decompose a large number in order to store it as $A \times 2^e + B$, where A and B may be represented as long values,
2. re-scaling the parameters: store a scaled version of the parameters, thus converting any `long` to `double` at the beginning of the process.

In regard of the first method, we observed that we could represent the sum of the square without any loss of information. However, for the computation of σ, there is a non-negligible loss of precision associated with the square root operation. We have tested many options to calculate the square root, and presented the results for some factorisations. Improving the computation of the square root may allow more precise results of the σ.

On the second method, the loss of precision occurs in the computation of the sum of the squares. Nevertheless, this loss is mitigated when calculating the standard deviation, showing better results than method 1. We also observed that the choice of the scaling parameter λ does not have a significant influence on the results.

In conclusion, our advice to the Softbridge company would be to turn to a `double` encoding. This choice is by far the most easily implemented, and the above study shows that it may outperform more refined methods.