Numerically Stable Parallel Computation of (Co-)Variance

Erich Schubert
Heidelberg University
Heidelberg, Germany
schubert@informatik.uni-heidelberg.de

Michael Gertz
Heidelberg University
Heidelberg, Germany
gertz@informatik.uni-heidelberg.de

ABSTRACT

With the advent of big data, we see an increasing interest in computing correlations in huge data sets with both many instances and many variables. Essential descriptive statistics such as the variance, standard deviation, covariance, and correlation can suffer from a numerical instability known as "catastrophic cancellation" that can lead to problems when naively computing these statistics with a popular textbook equation. While this instability has been discussed in the literature already 50 years ago, we found that even today, some high-profile tools still employ the instable version.

In this paper, we study a popular incremental technique originally proposed by Welford, which we *extend* to weighted covariance and correlation. We also discuss strategies for further improving numerical precision, how to compute such statistics online on a data stream, with exponential aging, with missing data, and a batch parallelization for both high performance and numerical precision.

We demonstrate when the numerical instability arises, and the performance of different approaches under these conditions. We showcase applications from the classic computation of variance as well as advanced applications such as stock market analysis with exponentially weighted moving models and Gaussian mixture modeling for cluster analysis that all benefit from this approach.

CCS CONCEPTS

• **Mathematics of computing** → **Mathematical software performance**; • **Applied computing** → *Mathematics and statistics*; • **Information systems** → *Data stream mining*;

ACM Reference Format:
Erich Schubert and Michael Gertz. 2018. Numerically Stable Parallel Computation of (Co-)Variance. In *SSDBM '18: 30th International Conference on Scientific and Statistical Database Management, July 9–11, 2018, Bozen-Bolzano, Italy*. ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3221269.3223036

1 INTRODUCTION

Statistical moments—the mean, variance, skewness, and kurtosis—are popular quantities to describe the shape of a probability distribution and are heavily used in descriptive statistics. Covariance and Pearson correlation extend this concept to measure the joint variability and linear dependence of multiple variables.

SSDBM '18, July 9–11, 2018, Bozen-Bolzano, Italy

Textbooks will often introduce variance as

$$\mathrm{Var}(X) = \mathrm{E}[(X-\mu)^2] = \mathrm{E}[X^2] - \mathrm{E}[X]^2 \tag{1}$$

where $\mathrm{E}[f]$ denotes the expected value of an expression f, and $\mu = \mathrm{E}[X]$ is the arithmetic mean.

The latter form (albeit mathematically correct) is problematic when used with *floating point* arithmetics. We will refer to this method as the "textbook" algorithm, as it is overly popular in textbooks despite its known deficiencies. As long as $\mathrm{E}[X]^2 \ll \mathrm{E}[X^2]$ (for example, on centered data with $\mu = 0 = \mathrm{E}[X]$), the problem will not surface. But once we have a data set where $\mathrm{E}[X^2] \approx \mathrm{E}[X]^2$, computing the right-hand side will cause a substantial loss of precision. In Figure 1, we give a toy example, assuming 4 decimal digits of precision; the commonly used IEEE-754 formats offer 7–8 decimal digits / 23+1 bits with single precision and 15–16 decimal digits / 52+1 bits with double precision. In the first example, $\mathrm{E}[X] \ll \mathrm{E}[X^2]$ and the result is correct in the maximum possible 4 digits. In the second example, both values agree on the first four digits, and computing the true difference would require a higher numerical precision. The resulting difference in this example is 0, the actual variance is lost. Because of rounding, we may even get a small negative variance in the worst case, which can then cause a NaN (not-a-number) error when computing the square root [12].

If we use $\mathrm{Var}(X) = \mathrm{E}[(X-\mu)^2] = \mathrm{E}\left[(X-\mathrm{E}[X])^2\right]$ instead (the "two pass algorithm"), we can obtain a more precise result. But computing this equation requires two passes over the data, the first pass to compute $\mu = \mathrm{E}[X]$. For small sized data and in main memory, this may well be acceptable, but once the data needs to be read from disk the runtime will usually double. On a live stream of data, this version cannot be used without storing the entire stream.

"Online" algorithms aim at integrating new data and *updating an existing result* without having to recompute everything. For example, we can "online" compute the mean of a data set $X' = X \cup \{x_{k+1}\} = \{x_1, \ldots, x_k\} \cup \{x_{k+1}\}$ based on the previous mean μ_k using $\mu_{k+1} = \frac{1}{k+1}(k \cdot \mu_x + x_{k+1})$. The term "online" comes from the idea of keeping a database "online" (available to serve requests) rather than having to perform an "offline" computation while no further changes to the database are permitted.

$\mathrm{E}[X^2]$		0	.	1	2	3	4	3	7	4		0	.	1	2	3	4	3	7	6	2
$-\,\mathrm{E}[X]^2$	-	0	.	0	0	0	1	2	3	4	-	0	.	1	2	3	4	1	5	2	1
$=\mathrm{Var}(X)$	=	0	.	1	2	3	3	1	4	0	=	0	.	0	0	0	0	0	0	0	0

Figure 1: Toy example showing *catastrophic cancellation* when using $\mathrm{E}[X^2] - \mathrm{E}[X]^2$, assuming four decimal digits of precision. In the left example, the result is correct in all four digits of precision. But in the right example, the first four digits cancel out completely, leaving no remaining known digits, and all precision is lost.

The objective of this paper is to discuss *numerically stable* computation of *variance, correlation and covariance* for scientific databases. *Weighted* computations will allow us to take uncertainty and importance of data points into account, such as the trade volume of a stock. Using *single pass* algorithms, we can perform the computations incrementally in an "*online*" fashion, and together with weighted computations in the same framwork we can also use it in *parallel* processing and *distributed* databases.

We will first do a detailed survey of literature in Section 2. In Section 3, we then introduce and proof a *general form for weighted (co-)variance*, from which we can derive many of the optimizations found in the earlier literature by specializing it to unweighted data and variance. In Section 4, we discuss how to use the general form for scenarios such as weighted Pearson correlation, exponentially weighted moving streaming models, covariance matrixes as used, e.g., in Gaussian Mixture Modeling (GMM) clustering, and how to use this to in parallel processing and distributed databases. In Section 5 we demonstrate the numeric precision, runtime, and applicability of several approaches in different settings.

2 RELATED WORK

Algorithms for calculating the variance have a long history in Computer Science, and the related work search for this paper turned out to be an interesting dive into the history of statistical computing, with most of the discussion happening in the Communications of the ACM and Technometrics in the 1970s.

We will first survey approaches mostly following chronological order, before we present our more general result in Section 3.

Most of the algorithms we discuss here have asymptotic complexity $O(n)$. The main differences are in the so-called constant factors, which are ignored in asymptotic analysis. For example, the two-pass algorithm which first computes the mean $\mu_k := \frac{1}{k}\sum_{i=1}^{k} x_i$, then in a second pass the squared deviations $S_k := \sum_{i=1}^{k}(x_i - \mu_k)^2$, is also of $O(n)$ complexity, but needs to access memory twice. In the following, the interest is therefore in (a) using only a single pass over the data, (b) keeping the number of operations for each data point small, and (c) having a high numerical accuracy.

Welford [27] shows that given k data points $x_1 \dots x_k$ one can update the running sum of squared deviations $S_k := \sum_{i=1}^{k}(x_i - \mu_k)^2$ and the running average $\mu_k := \frac{1}{k}\sum_{i=1}^{k} x_i$ with an additional new data point x_{k+1} using the simple algorithm:

$$\mu_{k+1} = \tfrac{k}{k+1}\mu_k + \tfrac{1}{k+1}x_{k+1} \tag{2}$$

$$S_{k+1} = S_k + \tfrac{k}{k+1}(x_{k+1} - \mu_k)^2 \tag{3}$$

The variance can then be simply computed using $\mathrm{Var}(X) = \frac{1}{k}S_k$. A similar approach was previously used by Box and Hunter [3] to update the change in a residual sum. This approach avoids the aforementioned catastrophic cancellation of the textbook algorithm, and it is much closer in precision to the two-pass algorithm. Welford also includes an equations for the sum of products and suggests how to compute higher-order moments. Knuth [12] gives a very simple unweighted variant of this algorithm (attributed to Welford):

$$\mu_{k+1} = \mu_k + \tfrac{1}{k+1}(x_{k+1} - \mu_k) \tag{4}$$

$$S_{k+1} = S_k + (x_{k+1} - \mu_k)(x_{k+1} - \mu_{k+1}) \tag{5}$$

Neely [15] compared three methods for computing the mean, four for computing the standard deviation, and four for computing the correlation along with the benefits of using double precision. Except for the textbook algorithm and Welford's algorithm, the tested algorithms all require two passes. The main contribution of Neely is to use a first pass to estimate the mean, and an error term based on this "trial mean" M_1.

$$M_1 = \tfrac{1}{k}\sum_{i=1}^{k} x_i$$

$$\mu_k = \tfrac{1}{k}\sum_{i=1}^{k} x_i + \tfrac{1}{k}\sum_{i=1}^{k}(x_i - M_1) \tag{6}$$

$$\mathrm{Var}_k = \tfrac{1}{k}\sum_{i=1}^{k}(x_i - M_1)^2 - \tfrac{1}{k^2}\Big(\sum_{i=1}^{k}(x_i - M_1)\Big)^2 \tag{7}$$

With perfect math, we would have $M_1 = \mu$, and the additional terms would be 0. But with floating point it can aggregate some rounding errors, in an approach that shares ideas with Kahan summation [10]. Rodden [18] suggests to use Neely's approach with integers to avoid rounding errors for many practical applications at that time. Van Reeken [26] found Welford's algorithm to perform better than evaluated by Neely, when implemented with different rounding, and suggests a slight variation of the equations:

$$\mu_{k+1} = \mu_k + \tfrac{1}{k+1}(x_{k+1} - \mu_k) \tag{8}$$

$$S_{k+1} = S_k + (x_{k+1} - \mu_k)^2 - \tfrac{1}{k+1}(x_{k+1} - \mu_k)^2 \tag{9}$$

Youngs and Cramer [30] reproduce the results of Neely, adding one additional online algorithm for computing the variance (and sum-of-products), which we will study in detail below (c.f. Equation 28). In their experiments, this approach is as accurate as the two-pass algorithm, but much faster. Ling [14] compares several algorithms, including Westford's, Neely's, Rodden's, van Reeken's, and Youngs and Cramer's on many different data distributions, but prefers the textbook algorithm with double precision. Hanson [9] uses the Givens transformation to derive similar update equations for the (weighted) mean and variance, and points out that one may also use negative weights to "remove" data points from the statistic. West [28] formalizes Hanson's approach for weights ω_i and the weight sum $\Omega_i = \sum_i \omega_i$ as:

$$\widehat{x}_{k+1} = \tfrac{1}{\Omega_{k+1}}\left(\Omega_k \widehat{x}_k + \omega_{k+1}x_{k+1}\right)$$

$$XX_{k+1} = XX_k + \tfrac{\omega_k \Omega_k}{\Omega_{k+1}}(x_{k+1} - \widehat{x}_k)^2 \tag{10}$$

Cotton [7] attempts to simplify the approach of Hanson, by reintroducing the problematic textbook equation to compute the variance from the sum-of-squares (as discussed by Chan and Lewis [6]), and with a subtle typo in the variance equation.

West [28] then simplifies Hanson's equations to reduce the number of multiplicative operations.

$$\delta_{k+1} = x_{k+1} - \widehat{x}_n$$

$$\delta'_{k+1} = \delta_{k+1} \cdot \omega_{k+1}/\Omega_{k+1}$$

$$\widehat{x}_{k+1} = \widehat{x}_k + \delta'_{k+1}$$

$$XX_{k+1} = XX_k + \delta_{k+1} \cdot \Omega_k \cdot \delta'_{k+1} \tag{11}$$

It can be seen as an optimized and weighted version of Welford's approach, and offers similar benefits as Van Reeken's [26].

Chan and Lewis [6] compare the textbook algorithm, the two-pass algorithm, West's algorithm and Cotton's algorithm with respect to runtime and accuracy. They note that Cottons algorithm should never be used, as it has the same limitations as the textbook algorithm without offering benefits. This is also confirmed in our experiments, as shown in Section 5.1.

Chan et al. [4, 5] discuss error bounds for the textbook and the two-pass algorithms. They also include a small modification of the two-pass algorithm that further improves numerical precision, which they attribute to a suggestion by Åke Björck, but which we could already find in Neely [15] (c.f. Equation 7). The main contribution of Chan et al. is a pairwise approach that aggregates the data in a binary tree to reduce losses. This is one of the few approaches that is in not in $O(n)$, but the tree-based aggregation will need $O(n \log n)$ operations.

While Welford [27] already mentioned higher moments, Terriberry [25] gives explicit equations for skewness and kurtosis. Pébay [17] shows how to derive compact update formulas for higher moments as well as covariance for the unweighted case. Equations for the incremental (but not parallel) computation of the weighted Pearson correlation coefficient were previously published in the Appendix of the PhD thesis by Schubert [20].

The reason why we revisit this long-studied (and apparently somewhat forgotten, as we will see below) task is due to improvements in CPU architectures, and the applicability of these techniques for parallel and distributed processing. Multiplications in CPUs have become a lot faster, while divisions remain much slower.[1] Memory locality becomes an increasingly important contributor to computation time, which can have a major impact on two-pass algorithms unless we can keep the entire data in memory.

The widespread use of the numerically instable equation in many tools can put scientific results into question. We could find the problematic textbook approach both in widespread relational databases (e.g., PostgreSQL 10.2), specialized scientific databases (e.g., RasDaMan 9.4), as well as scientific computing platforms such as Apache Spark MLLib 2.2.1, RapidMiner 8.1.0, and KNIME 3.5.2, confirming earlier observations for database systems [11]. MySQL 8.0.4 uses the Knuth-Welford version for variance, but does not include covariance at all. Numpy 1.14.1 uses the two-pass algorithm, but appears to create an unnecessary copy of the data. GNU R 3.4.3 uses four passes: one to count non-missing values, one for an initial mean, a second pass to refine the mean as used by Neely, and one pass for variance without the Neely adjustment. The implementation in ELKI used to use the stable two-pass algorithm, and since ELKI 0.6.0 [1] uses a numerically stable single-pass method as discussed in this article. None of the implementations we could find would make use of AVX vectorization of modern CPUs, which could increase runtime performance substantially.

In this paper, we introduce a general, *weighted* covariance form, which not only allows integrating single samples (as in most of the earlier work), but which can combine the results from arbitrary subsets. Although it is in spirit with the earlier work, these either do not support weights, or do not support covariance, while our form has both. One can use this for parallelization with multiple threads and cores, but also for distributed computation. Covariance can, of course, be specialized to variance, and we can also compute weighted Pearson correlation with this approach.

[1] E.g., on Intel Skylake, according to Fog [8], AVX divisions have a latency of 13-14 cycles and a reciprocal throughput of 8, while multiplication and fused-multiply-add ($a \cdot b + c$ in a single instruction) have a latency of 4 cycles and a reciprocal throughput of as little as 0.5 (i.e., the CPU can process two such instructions in a single clock cycle due to pipelining). Thus, multiplications can be about 4 to 5 times faster than divisions.

Symbol	Meaning
A,B,P	partitions of the data set
$AB = A \sqcup B$	union of disjoint partitions A and B
$A \sqcup \{b\}$	partition A plus a single $b \notin A$
$X = \{x_1, \ldots\}$,Y,V	variables
$\Omega = \{\omega_1, \ldots\}$	weights
$V_P = \sum_{i \in P} \omega_i v_i$	weighted sum of V over partition P
$\Omega_P = \sum_{i \in P} \omega_i$	weight sum of partition P
$\widehat{v}_P = \frac{1}{\Omega_P} \sum_{i \in P} \omega_i v_i$	weighted mean of V over partition P
$\widehat{vw}_P = \frac{1}{\Omega_P} \sum_{i \in P} \omega_i v_i w_i$	weighted sum of products
$VW_P = \sum_{i \in P} \omega_i (v_i - \frac{1}{\Omega_P} V_P)(w_i - \frac{1}{\Omega_P} W_P)$	weighted sum of deviation products

Table 1: Notation used in the general form

3 UPDATEABLE WEIGHTED (CO-)VARIANCE

In this section, we will derive a very general form that supports both multiple variables (X, Y), multiple partitions (A, B), and weights (Ω). In Table 1 we provide a summary of the most important notations. After deriving the general form in a few variations in Section 3.1, we will discuss specializations and optimizations for partitions containing a single point, as used in online algorithms, in Section 3.2. In the following Section 4 we will then show how to use the general form for more specific use cases.

We use A, B, $AB = A \sqcup B$ and the wildcard $P \in \{A, B, AB\}$ to denote partitions of the data. Our equations assume disjoint partitions of samples, and use $A \sqcup B$ to emphasize that we assume a disjoint union. $X = \{x_i\}$ and $Y = \{y_i\}$ denote data variables (with wildcards $V, W \in \{X, Y\}$), $\Omega = \{\omega_i\}$ is the weight variable. With a subscript, we use $V_P := \sum_{i \in P} \omega_i v_i$ to denote the weighted sum over variable V of all points in partition P, and the corresponding weight sum is $\Omega_P := \sum_{i \in P} \omega_i$. We denote the sum of deviation products as $VW_P = \sum_{i \in P} \omega_i (v_i - \frac{1}{\Omega_P} V_P)(w_i - \frac{1}{\Omega_P} W_P)$.

One can compute the mean and covariance from this by:

$$\widehat{v}_P = \tfrac{1}{\Omega_P} V_P \quad = \tfrac{1}{\Omega_P} \sum\nolimits_{i \in P} \omega_i v_i \tag{12}$$

$$\mathrm{Cov}(V, W)_P = \tfrac{1}{\Omega_P} VW_P = \tfrac{1}{\Omega_P} \sum\nolimits_{i \in P} \omega_i (v_i - \widehat{v}_P)(w_i - \widehat{w}_P) \tag{13}$$

3.1 Derivation of the General Form

In this section, we derive the general equations Equation 21 and Equation 22 below. A reader who is not interested in the derivation but only the application can skip to these results, and should still be able to follow on below.

We will first show some basic properties that we will require later. Because we require A and B to be disjoint, we have:

$$V_{AB} = \sum\nolimits_{i \in A} \omega_i v_i + \sum\nolimits_{i \in B} \omega_i v_i = V_A + V_B \tag{14}$$

$$\Omega_{AB} = \sum\nolimits_{i \in A} \omega_i \quad + \sum\nolimits_{i \in B} \omega_i \quad = \Omega_A + \Omega_B \tag{15}$$

We can also trivially prove the update equation for the mean:

$$\widehat{v}_{AB} = \frac{\Omega_A}{\Omega_{AB}}\widehat{v}_A + \frac{\Omega_B}{\Omega_{AB}}\widehat{v}_B = \frac{1}{\Omega_{AB}}(\Omega_A\widehat{v}_A + \Omega_B\widehat{v}_B) \quad (16)$$

$$= \widehat{v}_A + \frac{\Omega_B}{\Omega_{AB}}(\widehat{v}_B - \widehat{v}_A) \quad (17)$$

The König–Huygens formula then gives the "textbook" equality:

$$\mathrm{Cov}(V,W)_P = \frac{1}{\Omega_P}\sum_{i\in P}\omega_i(v_i - \widehat{v}_P)(w_i - \widehat{w}_P) \quad (18)$$

$$= \frac{1}{\Omega_P}\sum_{i\in P}\omega_i\left(v_i w_i - v_i\widehat{w}_P - \widehat{v}_P w_i + \widehat{v}_P\widehat{w}_P\right)$$

$$= \left(\frac{1}{\Omega_P}\sum_{i\in P}\omega_i v_i w_i\right) - \widehat{v}_P\widehat{w}_P - \widehat{v}_P\widehat{w}_P + \widehat{v}_P\widehat{w}_P$$

$$= \widehat{vw}_P - \widehat{v}_P\widehat{w}_P \quad (19)$$

$$\mathrm{Var}(V)_P = \mathrm{Cov}(V,V)_P = \widehat{(v^2)}_P - (\widehat{v}_P)^2 \quad (20)$$

This proves that the "textbook" algorithm is mathematically correct except for aforementioned floating point numerical problems due to catastrophic cancellation unless $\frac{1}{\Omega_P}VV_P \gg (\frac{1}{\Omega_P}V_P)^2$.

We will now use this form to derive alternative equations to compute (co-)variance. The key idea is to decompose P into two partitions A and B. One can either decompose P into a partition A and a single point $\{b\}$ to get an online algorithm, or into two arbitrary partitions to obtain a parallelization and distribution friendly approach that allows divide-and-conquer strategies.

To prove the general form, we first establish the following identity. We derive this in "backwards" direction, which is easier to do. The proof relies on $\frac{\Omega_{AB}}{\Omega_A} = \frac{\Omega_A+\Omega_B}{\Omega_A} = 1+\frac{\Omega_B}{\Omega_A}$, since $\Omega_{AB} = \Omega_A + \Omega_B$.

$$\begin{aligned}
&\frac{1}{\Omega_A}V_A W_A + \frac{1}{\Omega_B}V_B W_B - \frac{\Omega_A\Omega_B}{\Omega_{AB}}(\widehat{v}_A - \widehat{v}_B)(\widehat{w}_A - \widehat{w}_B)\\
&= \frac{1}{\Omega_A}V_A W_A + \frac{1}{\Omega_B}V_B W_B - \frac{\Omega_A\Omega_B}{\Omega_{AB}}(\widehat{v}_A\widehat{w}_A - \widehat{v}_A\widehat{w}_B - \widehat{v}_B\widehat{w}_A + \widehat{v}_B\widehat{w}_B)\\
&= \frac{1}{\Omega_{AB}}\Big(\frac{\Omega_{AB}}{\Omega_A}V_A W_A + \frac{\Omega_{AB}}{\Omega_B}V_B W_B\\
&\qquad - \frac{\Omega_B}{\Omega_A}V_A W_A + V_A W_B + V_B W_A - \frac{\Omega_A}{\Omega_B}V_B W_B\Big)\\
&= \frac{1}{\Omega_{AB}}(V_A W_A + V_A W_B + V_B W_A + V_B W_B)\\
&= \frac{1}{\Omega_{AB}}V_{AB}W_{AB} = \Omega_{AB}\widehat{v}_{AB}\widehat{w}_{AB}
\end{aligned}$$

Using this identity backwards in Equation 19, we can now show:

$$\begin{aligned}
VW_{AB} &= \Omega_{AB}\,\mathrm{Cov}(V,W)_{AB} = \Omega_{AB}\widehat{vw}_{AB} - \Omega_{AB}\widehat{v}_{AB}\widehat{w}_{AB}\\
&= \sum_{i\in AB}\omega_i v_i w_i\\
&\quad - \frac{1}{\Omega_A}V_A W_A - \frac{1}{\Omega_B}V_B W_B + \frac{\Omega_A\Omega_B}{\Omega_{AB}}(\widehat{v}_A - \widehat{v}_B)(\widehat{w}_A - \widehat{w}_B)\\
&= \sum_{i\in A}\omega_i v_i w_i - \frac{1}{\Omega_A}V_A W_A + \sum_{i\in B}\omega_i v_i w_i - \frac{1}{\Omega_B}V_B W_B\\
&\quad + \frac{\Omega_A\Omega_B}{\Omega_{AB}}(\widehat{v}_A - \widehat{v}_B)(\widehat{w}_A - \widehat{w}_B)
\end{aligned}$$

$$VW_{AB} = VW_A + VW_B + \frac{\Omega_A\Omega_B}{\Omega_{AB}}(\widehat{v}_A - \widehat{v}_B)(\widehat{w}_A - \widehat{w}_B) \quad (21)$$

or, equivalently, if we use the sum V_P instead of the mean $\widehat{v}_P$:

$$VW_{AB} = VW_A + VW_B + \frac{(\Omega_B V_A - \Omega_A V_B)(\Omega_B W_A - \Omega_A W_B)}{\Omega_A\Omega_B\Omega_{AB}} \quad (22)$$

Equation 21 allows us to combine the results of two partitions A and B, if we know $(VW_A, \widehat{v}_A, \widehat{w}_A, \Omega_A)$ and $(VW_B, \widehat{v}_B, \widehat{w}_B, \Omega_B)$, or, alternatively, $(VW_A, V_A, W_A, \Omega_A)$ and $(VW_B, V_B, W_B, \Omega_B)$. This means that just these four values for each partition provide sufficient statistics to compute the covariance of the combined data $AB = A \sqcup B$, and can be aggregated easily. In a distributed system, it means we need just a tiny amount of communication compared to transferring the entire data for analysis. By repeated application, we can also combine the covariances of multiple partitions.

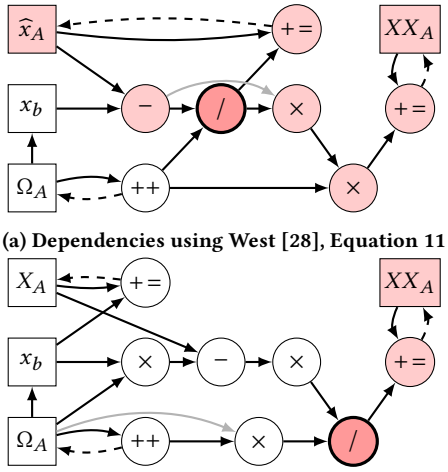


Figure 2: Dependencies of computations in two different equations to compute the variance

3.2 Optimizations and Specializations

Note that for a single point b (i.e., $B = \{b\}$) the statistics are simply $(0, v_b, w_b, \omega_b)$ and for $V = W = X$, Equation 21 simplifies to Equation 10 as used by Hanson [9].

Because $\frac{\Omega_A}{\Omega_{AB}}(\widehat{v}_A - \widehat{v}_B) + 0 = \frac{\Omega_A}{\Omega_{AB}}(\widehat{v}_A - \widehat{v}_B) + \frac{\Omega_B}{\Omega_{AB}}(\widehat{v}_B - \widehat{v}_B) = \widehat{v}_{AB} - \widehat{v}_B$, we can rewrite Equation 21 to either of:

$$VW_{AB} = VW_A + VW_B + \Omega_B(\widehat{v}_{AB} - \widehat{v}_B)(\widehat{w}_A - \widehat{w}_A) \quad (23)$$

$$= VW_A + VW_B + \Omega_B(\widehat{v}_A - \widehat{v}_B)(\widehat{w}_{AB} - \widehat{w}_B) \quad (24)$$

$$= VW_A + VW_B + \Omega_A(\widehat{v}_A - \widehat{v}_{AB})(\widehat{w}_A - \widehat{w}_B) \quad (25)$$

$$= VW_A + VW_B + \Omega_A(\widehat{v}_A - \widehat{v}_B)(\widehat{w}_A - \widehat{w}_{AB}) \quad (26)$$

which requires one multiplication and one division less. Again, specializing this for $X = Y$, a single new point $B = \{b\}$, with unit weight $\Omega_B = \omega_b = 1$ we get a variant equivalent to Equation 5 by Welford / Knuth:

$$XX_{A\sqcup\{b\}} = XX_A + \left(\widehat{x}_{A\sqcup\{b\}} - x_b\right)\left(\widehat{x}_A - x_b\right) \quad (27)$$

Youngs and Cramer [30] use the sum V_P instead of the mean $\widehat{v}_P$, but they only consider variance with $V = W = X$ and $B = \{b\}$ being a single new data point at a time. Equation 22 then simplifies to:

$$XX_{A\sqcup\{b\}} = XX_A + \frac{1}{\Omega_A(\Omega_A+1)}(X_A - \Omega_A x_b)^2 \quad (28)$$

This variant turned out to be unexpectedly effective, outperforming closely related methods by a factor of 2, as we will see in the experimental evaluation. The substantial performance benefit of this approach cannot be explained just with the number of math operations (it requires one *additional* multiplication compared to West's approach), but it is supposedly due to pipelining capabilities of modern CPUs. The slowest operation is the division, and in the approaches by Welford, West, and others the division is done early, and subsequent operations depend on the result. In Youngs and Cramer's approach, the division is done last, and none of the

other values depend on the previous value of XX_A. Therefore, we assume the CPU can apply pipelining here much more effectively. In Figure 2a we illustrate the dependency graph for West's method (Equation 11), while Figure 2b is that of the approach of Youngs and Cramer (Equation 28). Colors indicate computation that depends on the (slow) division, arrows indicate computational dependencies. In West's approach, almost all values (except the data and the weights) depend on the outcome of the division, and we have a cyclic depedency between the division and the running mean $\widehat{x}_b$. In Youngs and Cramer's approach, only the sum of squares XX_b depends on the division operation. Because of this, we suggest to use the sums X_P, Y_P instead of the means $\widehat{x}_P$, $\widehat{y}_P$, and to prefer Equation 22 over Equation 21, but this may require additional benchmarking. In particular, when additional operations are necessary to load or compute the input values x_i—that can also be pipelined by the CPU—such benefits can easily disappear.

4 USAGE EXAMPLES

We will now discuss how to use these equations in different scenarios. First of all, we discuss the weighted form of Pearson correlation, and how we can compute the correlation of long series of sensor data in Section 4.1. Secondly, we discuss how to extend this to exponentially weighted moving averages of streaming data in Section 4.2. The third scenario detailed is clustering with Gaussian Mixture Modeling, which requires weighted covariance computations, in Section 4.3. We then discuss parallel computation with vectorization in Section 4.4 and with distributed parallelization in Section 4.5. Last but not least, we discuss options to further increase numerical precision in Section 4.6. In the following Section 5 we then experimentally study precision and runtime performance.

4.1 Weighted Pearson Correlation Coefficient

The Pearson product-moment correlation coefficient [16], often denoted as PPMCC, PCC or Pearson's r, is a classic statistical measure of linear dependence (correlation). It has been applied to various problems successfully, and "for a 100-year-old index it is remarkably unaffected by the passage of time" [19]. It can be interpreted as standardized covariance, or as standardized slope of the regression line, but also in several other ways [19].

Weighted Pearson correlation can be defined consistent with the classic Pearson product-moment correlation coefficient:

$$\rho_\omega(X,Y) = \frac{\mathrm{Cov}_\omega(X,Y)}{\sqrt{\mathrm{Cov}_\omega(X,X)\,\mathrm{Cov}_\omega(Y,Y)}} = \frac{\mathrm{Cov}_\omega(X,Y)}{\sqrt{\mathrm{Var}_\omega(X)\,\mathrm{Var}_\omega(Y)}} \tag{29}$$

Where $\mathrm{Cov}_\omega(X,Y)$ is the weighted covariance, and $\mathrm{Var}_\omega(X) = \mathrm{Cov}_\omega(X,X)$ is the weighted variance.

Weighted Pearson correlation, just like regular Pearson correlation, is in the range of $[-1;+1]$, and can be used as a dissimilarity function either via $1-\rho_\omega$ or via $1-\rho_\omega^2$. The key difference between these two is that in the latter, a perfect negative correlation ($\rho=-1$) also yields a distance of 0, which may be sometimes desirable.

Pearson correlation is not generally a metric distance. This is easy to see in particular as it obviously is not defined when a vector has variance 0.[2] However, if the vectors are not constant and are standardized to zero mean $\sum_i x_i = 0$ and unit variance $\sum_i x_i^2 = 1$, the formula simplifies to $\rho(X,Y) = \mathrm{Cov}(X,Y)$. This standardization does not change correlation, but with this preconditions it turns out to be a variant of Euclidean distance:

$$d_{\text{Euclidean}}(x,y) = \sqrt{\textstyle\sum_i (x_i - y_i)^2} = \sqrt{\textstyle\sum_i x_i^2 + \sum_i y_i^2 - 2\sum_i x_i \cdot y_i}$$
$$= \sqrt{2n - 2\textstyle\sum_i x_i \cdot y_i} = \sqrt{2n(1-\rho(X,Y))}$$

where the last line requires standardized data: $\sum_i x_i^2 = 1 = \sum_i y_i^2$.

Therefore, if we disallow constant vectors, (weighted-) Pearson correlation can be converted into a *metric* by using

$$d_{\text{Pearson}}(x,y) = \sqrt{1-\rho_\omega(X,Y)}$$

For data mining, it will often be more efficient to *store the standardized vectors*, and use Euclidean distance. It is worth emphasizing that with the transformed data, we can efficiently find the most correlated vectors using various data indexes such as k-d-trees and R-trees. Doing so requires an additional rescaled copy of the data, but allows using the simpler Euclidean distance during mining.

In this particular scenario, we can assume the vectors to be short enough for the two-pass algorithms; so while the incremental method can be used, it probably only offers no benefits here. When processing fewer, but much longer vectors such as time series, the new method becomes more interesting, in particular, when we look at live streaming time series such as sensor or stock market data. Preprocessing the data in two passes is then no longer possible, and we need to be able to update Pearson correlation on the fly.

For each sensor, we maintain the sum X (or the average $\widehat{x}$), and pairwise sum of squared deviations XY for any two sensors X and Y. We can incrementally update the statistics to maintain correlation information. The proposed approach does not require keeping the entire time series in memory and can be updated easily, while also offering good numerical behavior. The textbook approach based on $\mathrm{E}[X\cdot Y] - \mathrm{E}[X]\cdot\mathrm{E}[Y]$ also does not require to keep the entire series in memory, but is numerically problematic for non-central data.

For the actual use on sensors, it may also be interesting to "age" the historic data, and put more weight on new data as it arrives. We will discuss this in the following subsection.

4.2 Exponentially Weighted Moving Correlation

For an exponentially weighted moving average (EWMA), we weight each point in a series $i=1,\ldots,n$ with an exponentially decreasing weight $\omega_i = \alpha\cdot(1-\alpha)^{n-i}$. For an infinite series, the total weight then is approximately 1. Because this weighting scheme can be defined with the recurrence $\omega_{i-1} = (1-\alpha)\cdot\omega_i$, we can efficiently compute this by applying the discount factor $1-\alpha$ each step to the old data sums (i.e., Ω_A, X_A, XX_A, etc.; but not to averages like $\widehat{x}_A$), and add the new data always with weight α, rather than recomputing all weights when we add a new data point. To use this scheme with our approach, we can either use fixed weights $\Omega_A = 1-\alpha$ and $\Omega_B = \alpha$, or we simply apply the discounting factor to the current weight of the old data prior to executing the update step, which handles the beginning of the series better.

We can choose α using different formulations, for example, the half-life time t: $\alpha = 1 - \exp(\log(\frac{1}{2})/t)$, where t is the number of

[2] We could define correlation with a constant to be 0, but it this is not always satisfactory. But what is then the correlation of two constant variables? For the purpose of obtaining a metric distance, we would need to define the correlation of two constants as 1 if they are the same constant to satisfy reflexivity. Undefined correlation may often be the more appropriate answer.

iterations until the weight of a point has decreased to half the initial weight. For example, if we set $t = 4$, we get $\alpha \approx 0.159$ which subsequently decreases to 0.134, 0.113, 0.095 and 0.080 – so after 4 iterations, the initial weight has reduced to half.

SigniTrend [22] and SPOTHOT [23] apply EWMA and exponentially weighted moving variance (EWMVar) to analyze the frequency of word patterns on streaming data. With our weighted covariance formulation, we can analogously define exponentially weighting moving correlation (EWMCorr).

This analysis is interesting on streaming data to monitor changes in correlation of variables, for example, stock market charts. A sudden departure from a correlation, or the sudden increase in correlation of variables, can indicate important changes in the market. With our approach, we can furthermore incorporate, e.g., the trading volume as weights into our analysis. Let $\widehat{t}$ be a long-running average volume of the stock, then we can instead choose $\alpha' = 1 - (1-\alpha)^{t/\widehat{t}}$ to base the learning rate on "average" trading days rather than calendar days, and the moving average will react slower to low-volume trades and faster if we see an increased trading volume. We show experiments with this weighting scheme in Section 5.2.

4.3 Parallel Computation of Covariance Matrixes for Gaussian Mixture Modeling

We can now use the equations from Section 3.1 to parallelize Gaussian Mixture Modeling (GMM) without using two passes over the data, without losing numerical precision. In clustering, we cannot assume that all clusters are close to 0, but rather we must assume that $\mathrm{E}[X]^2 \ll \mathrm{E}[X^2]$ if we want well-separated clusters. Therefore, numerical precision is very likely to suffer.

For example, Apache Spark MLLib 2.2.1 uses the numerically problematic textbook approach in its GMM code. Since version 0.6.0, ELKI [1] uses the numerically stable single-pass method discussed here, prior to that it used the two-pass algorithm. None of the implementations we investigated would make use of AVX vectorization.

To compute the covariance matrix, we need to compute the mean $\widehat{x}$ for each variable X, and the sum of squared deviations $XY = \sum_i \omega_i (x_i - \widehat{x})(y_i - \widehat{y})$ for all pairs of variables X and Y. Because of symmetry, it is sufficient to track an upper (or lower) triangular matrix (including the diagonal), and the sum of weights $\Omega = \sum_i \omega_i$ will be the same for the entire matrix, so we only need to update this once (unless we also need to support missing values).

To add a data point to our summary, we can use this simplified *update step* derived from Equation 24:

$$
\begin{array}{lll}
 & \Omega_{A\cup\{i\}} \leftarrow \Omega_A + \omega_i & \} \text{ Upd. weights} \\
\forall_X & \delta_x \leftarrow x_i - \widehat{x}_A & \\
\forall_X & \widehat{x}_{A\cup\{i\}} \leftarrow \widehat{x}_A + \frac{\omega_i}{\Omega_{A\cup\{i\}}} \cdot \delta_x & \} \text{ Upd. means} \\
\forall_Y & \delta'_y \leftarrow y_i - \widehat{y}_{A\cup\{i\}} & \\
\forall_{X,Y} & XY_{A\cup\{i\}} \leftarrow XY_A + \omega_i \cdot \delta_x \cdot \delta'_y & \} \text{ Upd. deviations}
\end{array}
$$

Because $\omega_i / \Omega_{A\cup\{i\}}$ is shared across all variables, it is beneficial to compute this factor only once, and the pipelining effects observed in Figure 2 do not apply to the multivariate case.

For *parallelization*, we can compute XY_A and Ω_A on separate partitions A, for example, using multiple registers, multiple cores, or multiple nodes of a cluster. One can then merge the results using Equation 24 using the following *merge step*:

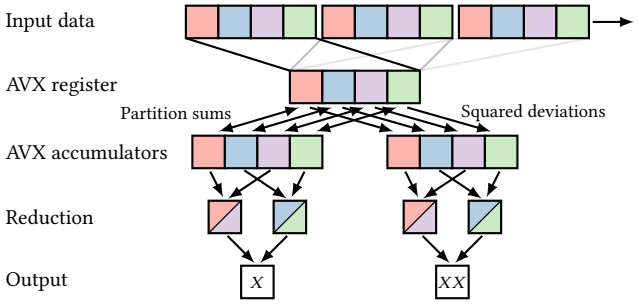


Figure 3: Data flow in AVX computation: all operations are performed 4× parallel (with AVX-512: 8×), then for the final output, the four partitions are combined in a final step (some arrows $X \rightarrow XX$ in the reduction are omitted for readability).

$$
\begin{array}{lll}
 & \Omega_{AB} \leftarrow \Omega_A + \Omega_B & \} \text{ Upd. weights} \\
\forall_X & \delta_x \leftarrow \widehat{x}_B - \widehat{x}_A & \\
\forall_X & \widehat{x}_{AB} \leftarrow \widehat{x}_A + \frac{\Omega_B}{\Omega_{AB}} \cdot \delta_x & \} \text{ Upd. means} \\
\forall_Y & \delta'_y \leftarrow \widehat{y}_B - \widehat{y}_{AB} & \\
\forall_{X,Y} & XY_{AB} \leftarrow XY_A + XY_B + \Omega_B \cdot \delta_x \cdot \delta'_y & \} \text{ Upd. dev.}
\end{array}
$$

From the resulting XY, one can obtain (co-)variance using the equation $\mathrm{Cov}(X, Y) = XY/\Omega$ at any time.

4.4 Vectorization

The key idea for vectorization is to use SIMD (single instruction, multiple data) instructions to compute several partitions in parallel and then merge the results. One can employ different strategies:

- Using SIMD instructions such as AVX one can process 4 double-precision floating point numbers at once using 4 parallel accumulators (with AVX-512, we will get a width of 8 doubles). The resulting partitions can then be reduced to a single result at the end. This is illustrated in Figure 3.
- Multiple registers can be used to get a larger width. AVX provides 16 registers (AVX-512 has 32), but one also need some register for intermediate values. Using 4×2 registers for mean and variance leaves half of the registers available, but allows processing 16 partitions in parallel.
- Micro-batches of, e.g., 16 or 32 values depending on memory cache line size can be processed using the two-pass approach using SIMD instructions. These results are then aggregated as in the first approach described, but we need to run this approach less often, and may get a slight improvement in precision from the two-pass stage.

In our experiments below, we usually observe a slight increase in precision using these approaches. Given that we use 256 bits instead of 64 bits to perform our computations until we merge these in the very end, it is not surprising that we see a slight increase in numerical precision compared to the non-vectorized versions. Furthermore, this approach uses fused-multiply-add (FMA, $a \cdot b + c$) operations that internally can use higher precision.

For integration into a scientific database, the AVX approach, however, means a considerable increase in effort: these instructions require that memory is aligned on 32 byte boundaries, and that values are stored consecutively in memory (e.g., in a column-oriented storage). On a row-oriented storage—which is often more appropriate for dense vector data in data mining—the data reorganization likely is too expensive to make the AVX approach worthwhile.

This kind of computation could also be used on a GPU. However, not all GPUs may provide full double-precision accuracy. Nvidia compute capability 1.2 and below, for example, only include single-precision, and does not guarantee IEEE 754 accuracy for all operations [29]. With modern compute capability 2.0, the accuracy should be comparable. However, in many cases the cost to transfer the data to the GPU may be non-negligible for such a low-cost computation, so that GPU approaches will likely only be beneficial when the data *already* is stored in the GPU memory.

4.5 Distributed Parallelization

When processing very large data sets, we can also use the same idea to distribute the computation onto multiple nodes. Each of these nodes can independently run a two-pass or the online algorithm to compute the statistics. Only a minimum amount of communication is necessary to transmit the summaries (one tuple of the form $(VW_P, V_P, W_P, \Omega_P)$ for each partition P) to the coordination node, where the results can be easily combined. Alternatively, summaries can also be combined in a tree structure as recommended by Chan et al. [4], and possibly by multiple workers.

Only when we had a column partitioning of the data (where columns reside on different nodes), we would need to stream the data from one node to another to be able to compute correlations. To the best of our knowledge, column store systems will prefer horizontal partitioning across nodes, and vertical partitioning inside. However, when joining tables from different nodes, such situations can nevertheless arise. The online algorithm is still applicable then, but the join itself will likely require the data from one node to be streamed to the other.

4.6 Further Improvements to Accuracy

Neither our approach, nor the two-pass algorithm, are safe from numerical problems, just like any other floating point computation. In fact, a data set with just three points is enough to cause problems: on 3, 10^{100}, -10^{100} we will already fail to compute the correct mean (which obviously should be 1, but we will usually get a result of 0).

If additional precision is needed, we can equip all our accumulators (each Ω, X, XY) with additional overflow registers and use Kahan summation [10] or the adaptive Shewchuk approach [24] to compensate for lost digits with additional accumulators. A similar modification can be done to the textbook and the two-pass algorithm to improve precision.

Furthermore, we can employ a trial mean to improve numerical precision, as used by Neely [15]. In this approach, we compute the deviation from a guessed value; and if our guess is good enough, this can improve numerical precision. For example, we can compute the mean using $\mu = \eta + \frac{1}{n}\sum_{i=1}^{n}(x_i - \eta)$, which for $\eta = 0$ gives the usual mean. If $\eta = \mu$, the last term will sum to 0 in exact math, but due to floating point rounding this will often not hold in practice. If η is close to μ, this will improve numerical precision. Chan et al. [5] recommend to "shift the data as well as possible". This can be in particular beneficial if we have pre-sorted data, where numerical precision is more problematic than on random data, and can guess η for example by using the median.

In our experiments shown in the next section, we found that we can usually improve precision by about four decimals, and with most algorithms we get close to the double precision limit. From a statistical point of view, this extra precision may, however, be questionable, because our input sample is finite. The relative standard error of the variance is $\approx \sqrt{2/n}$ [2], so to warrant a precision of d digits, we should use about $2 \cdot 10^{2 \cdot d}$ samples, twice as many as to estimate the mean. Thus, except for the textbook approach, any of the algorithms we evaluated is precise enough unless our input data as in the example above has a very high dynamic range, causing a loss of precision already when computing the sum.

5 EXPERIMENTS

In this section, we first study numeric precision and runtime of different methods for computing the mean and variance in Section 5.1. In particular, we demonstrate that the popular textbook equation $E[X^2] - E[X]^2$ should not be used for computing the variance. Not only has this version a low precision under certain – not uncommon – conditions, but there also exist alternatives with much better precision at little additional runtime cost.

In the second experiment, in Section 5.2, we apply the weighted online algorithm to stock charts, where the weights are chosen based on the current trading volume.

Next, we benchmark different implementations of clustering with Gaussian Mixture Modeling (GMM). Again, we observe numerical problems with the textbook covariance $E[X \cdot Y] - E[X] \cdot E[Y]$, and suggest to not use it. Instead, either the two-pass algorithm should be used, or an online algorithm as introduced in Section 4.3. We demonstrate that a very popular distributed machine learning toolkit suffers from these numerical issues, and should be updated to use the equations introduced in this paper instead.

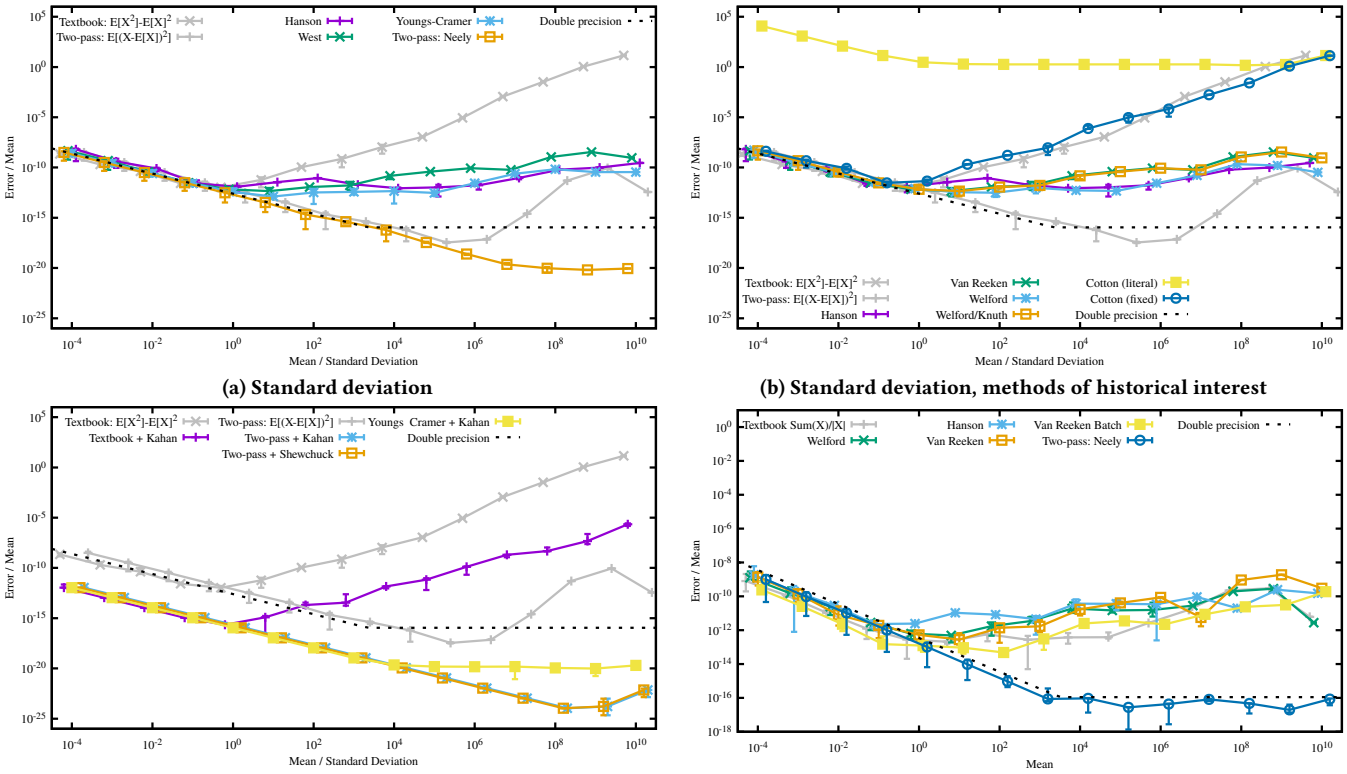
5.1 Runtime and Accuracy

We evaluated different approaches on synthetic data of 100 million samples generated from a normal distribution with fixed variance $\sigma^2 = 1$ and varying mean $\mu \in \{10^{-4} \dots 10^{10}\}$. For obvious reasons, the ratio σ^2/μ is of interest. We considered five permutations of the data: a fixed random order, ascending, descending, ascending by the deviation from μ and descending by the deviation from μ. For each method, we only use the worst ordering for evaluation. Sorted order caused substantially larger errors for most methods, as this causes many of the small values to be lost due to rounding errors. Each experiment was repeated for 11 different random seeds.
We compare various methods that can be grouped as follows:

(1) Incremental algorithms, including the *Textbook* approach using $\mathrm{E}[X^2] - \mathrm{E}[X]^2$ which can, for example, be found in the Boost C++ library as `tag::lazy_variance`. The approaches by *Welford*, also in the variant of *Knuth*, and the versions of *Van Reeken*, *Hanson*, *West*, and *Youngs and Cramer*. *Boost* also includes a variation of Hanson as `tag::variance`. For *Cotton* we include both the literal version and a version with the fix. As expected,

Table 2: Runtime and precision of different equations to computing the mean and variance of 100.000.000 doubles using the GCC 7 compiler and an Intel Core i7-7560U @2.4GHz CPU. Aggregate over 11 runs, worst data ordering only.

Method	Variant	Runtime (s)				Precision (decimal digits)			
Mean		Min	Mean	Median	Max	Best	Mean	Median	Worst
Boost	double	752.25	752.47	752.37	753.05	12.605	10.574	10.539	8.222
Hanson	double	752.23	752.44	752.35	752.88	12.605	10.574	10.539	8.222
Van Reeken	double	919.37	919.62	919.52	920.20	12.848	10.715	10.746	8.569
Welford	double	920.78	922.86	922.96	923.64	12.726	10.834	10.828	8.529
Textbook	double	168.14	168.28	168.25	168.54	14.298	11.376	11.571	8.680
Van Reeken (Minibatch)	double	91.88	92.20	92.18	92.68	13.388	11.541	11.588	9.143
Two-pass (Neely)	double	336.75	337.02	336.93	337.62	17.866	14.490	16.012	8.748
Van Reeken	AVX	86.87	87.11	87.11	87.30	13.874	11.450	11.478	9.598
Textbook	AVX	45.22	45.74	45.50	47.15	14.525	11.949	12.545	9.526
Van Reeken	AVX×8	44.27	44.86	44.57	46.29	14.732	12.417	12.582	10.023
Textbook	AVX×8	46.67	47.23	46.98	48.66	15.535	13.107	13.644	10.477
Textbook	Kahan	668.68	668.90	668.75	669.77	17.866	16.441	16.333	15.835
Textbook	Shewchuk	1109.61	1343.99	1134.27	2485.95	17.866	16.441	16.333	15.835
Van Reeken	Kahan	1112.33	1112.98	1113.01	1115.08	17.866	16.501	16.374	15.643
Variance									
Cotton (literal)	double	1256.43	1257.34	1257.10	1260.67	-0.076	-3.949	-3.257	-11.135
Cotton (fixed)	double	921.88	924.09	924.01	927.39	13.010	3.415	5.035	-11.127
Textbook	double	168.85	168.97	168.93	169.22	12.848	4.086	6.150	-11.153
Van Reeken	double	943.66	944.55	944.29	946.14	13.224	7.441	8.787	-0.963
West	double	933.17	934.42	934.42	935.67	13.224	7.441	8.787	-0.963
Welford / Knuth	double	929.17	929.97	929.93	931.18	13.224	7.441	8.787	-0.963
Boost	double	940.36	943.00	942.90	945.25	12.432	7.793	8.700	-0.451
Hanson	double	766.45	766.71	766.58	767.37	13.370	7.868	8.700	-0.451
Welford	double	943.51	951.54	952.72	955.09	13.064	8.072	9.019	0.479
Youngs & Cramer	double	212.20	**212.53**	**212.49**	213.31	12.840	8.284	9.588	0.454
Proposed Minibatch	double	202.50	202.78	202.73	204.33	14.256	9.299	10.184	0.805
Two-pass	double	336.78	337.02	336.93	337.38	14.135	10.168	12.383	1.045
Two-pass (Neely)	double	337.15	337.41	337.32	338.08	14.135	12.372	12.485	10.042
Textbook	AVX	48.61	49.04	48.86	50.14	5.932	-0.469	1.393	-12.599
Welford / Knuth	AVX	94.75	95.12	95.17	95.44	13.815	8.287	9.622	-0.805
Youngs & Cramer	AVX	65.40	66.30	66.16	67.35	14.301	8.838	10.187	0.516
Welford / Knuth	AVX×4	50.93	**51.47**	**51.22**	52.83	14.041	8.892	11.306	-0.547
Youngs & Cramer	AVX×4	49.82	52.88	52.98	54.85	14.353	9.524	10.600	0.805
Welford / Knuth (Minibatch)	AVX	54.47	55.32	55.01	62.55	12.284	8.263	10.490	-0.547
Youngs & Cramer (Minibatch)	AVX	55.18	56.52	56.60	57.36	13.268	8.948	10.254	0.805
Two-pass	AVX	91.52	92.42	92.15	94.85	13.685	10.817	13.122	1.051
Two-pass	AVX×4	91.04	92.17	91.74	95.31	14.239	11.907	13.595	1.108
Two-pass (Neely)	AVX	96.88	97.69	97.36	99.81	13.685	12.851	13.123	10.365
Two-pass (Neely)	AVX×4	89.07	90.47	89.75	95.90	14.375	13.240	13.591	10.707
Textbook	Kahan	668.70	668.89	668.82	669.27	15.955	8.653	10.450	-4.430
Youngs & Cramer	Kahan	692.21	692.44	692.39	692.97	15.955	14.538	15.955	9.570
Two-pass	Kahan	1337.38	1337.77	1337.54	1338.58	15.955	15.630	15.955	12.006
Two-pass	Shewchuk	2174.92	4000.18	3741.58	6601.13	15.955	15.630	15.955	12.006
Two-pass (Neely)	Kahan	1337.37	1337.74	1337.56	1338.75	15.955	15.941	15.955	15.654



(a) Standard deviation

(b) Standard deviation, methods of historical interest

(c) Standard deviation, high-precision methods

(d) Mean

Figure 4: Estimation errors

this version is slower than the textbook version, but it is just as instable.

(2) Algorithms requiring multiple passes, in particular the canonical *two-pass* approach, also with *Neely*'s adjustment, and a *mini-batch* version using the two-pass approach only on mini-batches of 16 values at a time, then aggregating the results incrementally with Equation 23 as explained in Section 4.4, but without AVX.

(3) AVX vectorized versions from Section 4.4 of the incremental algorithms of Welford / Knuth, and Youngs and Cramer. The AVX×4 variant uses four registers and processes 16 partitions in parallel. The partitions are in the end aggregated into a single result using a binary tree of pairwise aggregations, similar to Chan et al. [4].

(4) AVX vectorized versions of the two-pass algorithm.

(5) High precision versions using Kahan summation [10] or the Shewchuk algorithm [24] for aggregation, but regular double precision for multiplication and the mean.

To compute the reference values, we also used the Shewchuk algorithm with two passes, but using simulated (slow) quadruple precision floats, to compute the reference value accurate to a significant with 112+1 bits (i.e., 34 decimal digits), which is sufficient to evaluate the maximum 15.955 decimal digits of a double. Scores larger than this are possible, because our input data also is only double precision. All evaluated implementations use *double* precision internally (and a double approximation of the mean), and the resulting rounding errors cause an additional error in the variance. The Kahan and Shewchuk based versions use the high-precision technique only in the accumulators, not for the other computations.

Table 2 gives the results of this experiment (minimum, maximum, mean, and the median-of-medians for each method). We first evaluate the precision of the *mean*, while the lower part of the table evaluates the precision of the *variance*. The table is sorted by the mean within each group. This experiment used 10^8 samples, so the statistically meaningful precision will only be around 4 digits. Furthermore, we must only treat the numbers as a rough indication of which methods to prefer: The reported mean and median depend very much on the choice of σ^2/μ, and on patterns in the input data, and will not translate to a guaranteed improvement on actual data. In fact, on many real data sets with a variance much larger than the mean, even the textbook algorithm will give near-optimal precision. This experiment yields a number of interesting observations:

- The textbook algorithm is the fastest, twice as fast as the two-pass algorithm, but it also is by far the least accurate.
- The two-pass algorithm, in particular with Neely's modification, offers the highest accuracy, but it may not be applicable in all scenarios because of the second pass needed. Performance-wise, the two-pass algorithm is surprisingly competitive, supposedly because it does not involve slow division operations for each data point.
- Online algorithms vary surprisingly a lot both in precision and runtime. The better approaches of Hanson, Welford, and Youngs and Cramer will typically give twice as many digits of precision as the textbook approach.

- Youngs and Cramer's has a surprising performance benefit over the alternatives, and clearly is the go-to method for a true streaming approach. As explained in Section 3.2, we attribute this to pipelining benefits of modern CPUs.
- If we can afford to buffer some data points, the proposed mini-batch algorithm improves both runtime and precision compared to the best streaming approach.
- Parallelization with AVX yields speedups of a factor of 2 to 4, but requires much more implementation efforts, for example, because of the required memory alignment. When data is not already in a consecutive array of doubles, they probably are not beneficial to use.
- The parallelized AVX approaches also yield a small gain in precision, because they postpone merging the individual partitions to the end, and keep 4-16 values in parallel.
- The approaches using Kahan and Shewchuk compensation during aggregation are accurate to the limits of double precision, except for the textbook algorithm (as we implement only a high-precision aggregation, not multiplication). However, they are also 2 to 3 times slower, and the additional gain in accuracy, as explained in Section 4.6, is supposedly less than the statistical uncertainty of the data overall.

With respect to numeric precisions, our results experimentally confirm the findings by West [28] and Youngs and Cramer [30]. Figure 4 visualizes the relative error of the different approaches dependant on the ratio of standard deviation to the mean. Lines are offset slightly on the x axis to reduce overlap, all x values are exactly powers of 10. As long as the standard deviation is much larger than the mean, none of the methods has problems, and we can estimate variance accurately to about 12 decimal digits (which supposedly is due to the unavoidable square in the variance definition). As we decrease the variance, we have fewer and fewer significant digits in our deviation from the mean, the black dashed line indicates a theoretical limit due to double precision. While we can get precision much below this line using, e.g., Neely's adjustment, Kahan compensation or Shewchuk summation, precision below this line is not very well founded. At the right end, when our standard deviation is about 10^{-10}, our input random numbers will only differ from the mean with very few bits in the significand. The important area of the plot is in the center, where the standard deviation is 2–4 orders of magnitude smaller than the mean. In this range, the textbook method using $\mathrm{E}[X^2]-\mathrm{E}[X]^2$ will give us results that are incorrect by a substantial factor. For example, if we have just two points, 1 ± 10^{-7} (with the true variance 10^{-14}), $\mathrm{E}[X^2]$ will be about $1+1.021405\cdot10^{-14}$, because of floating point errors. After subtracting $\mathrm{E}[X]^2=1$, we get an estimated variance with only two significant digits of precision. At around 10^{-11}, we get $\mathrm{E}[X^2]=\mathrm{E}[X]^2$ with respect to double precision, and the variance computed by the textbook approach is always 0. Chan et al. [4, 5] give theoretical error bounds for different approaches that can help explain the shape of the curves that we observe in this experiment.

In conclusion of this experiment, we suggest to use the two-pass algorithm with Neely's adjustment when applicable, and otherwise resort to a mini-batch algorithm or the variant of Youngs and Cramer for unweighted data. For weighted data, Equation 22 may offer similar pipelining benefits over Equation 21, but supposedly

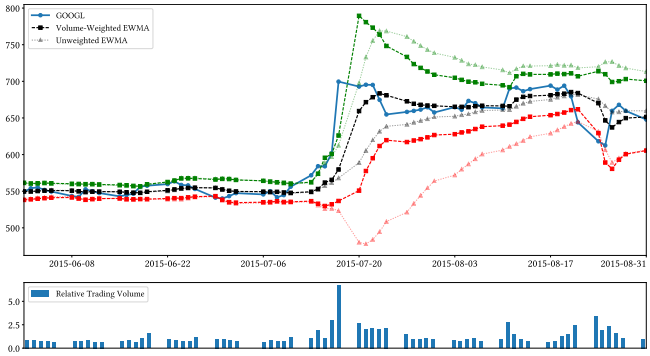


Figure 5: Google stock during the "one day rally" 2015. Our volume-weighted EWMA variant (dashed lines) reacts quicker due to the increased trading volume, while it does not differ much from a regular EWMA model (dotted lines) during the average trading before and after the rally. Bounds give the stock mean ± two standard deviations, similar to Bollinger Bands but with exponentially decreasing weight.

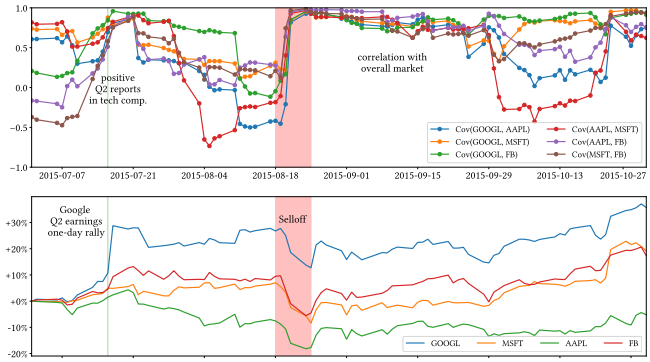


Figure 6: August 2015 stock market selloff. We observe an increase in correlation of Tech stocks at the positive Q2 quarterly reports, and during the overall stock market selloff August 18-25 and the subsequent days of market nervousness.

with a smaller benefit because of the additional processing needed for the weights. If possible, a AVX vectorization is worth exploring, but the integration into existing scientific databases may be difficult because of memory alignment and data layout.

5.2 Stock Market Analysis

We applied the weighting discussed at the end of Section 4.2 to daily stock market data from four technology companies. We weight data volume such that it has a half-life time of three average trading days. In Figure 5, we can see this weighting scheme in effect. During the one-day rally following Google's 2015 Q2 quarterly profit report, the stock value increased by 25%. During this period, we see a much increased trading volume, and because of this our approach reacts faster than the traditional exponentially weighted moving average.

In Figure 6, we study the correlation of stocks during the August 2015 stock market selloff that peaked on Friday, August 21 and Monday, August 24. On Monday, international stocks followed suit in China (-8%), India (-6%), and Europe (-3%). The following days, we observe a strong correlation of all stocks, due to this

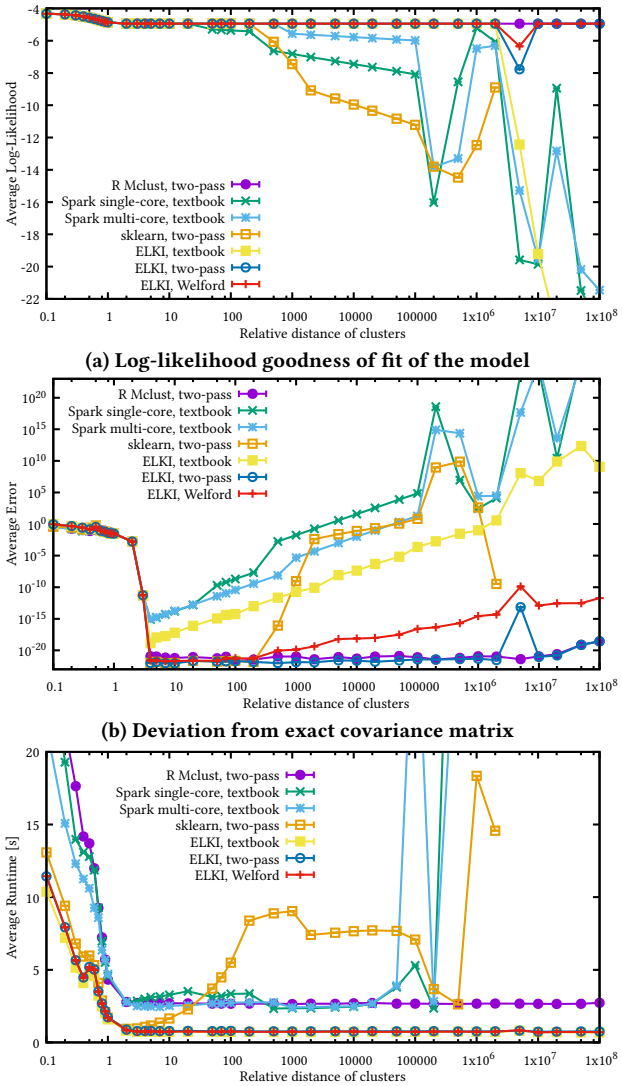
overall market trend. This shows that it is, unfortunately, not that trivial to identify correlations in stocks that are *not* caused by general market developments, and such an analysis will require more careful decorrelation to be useful. In the beginning of the time series, we can see Google's one day rally again, and an overall high correlation in the tech stocks after their positive quarterly reports. Inbetween, we can see a moderate negative correlation of Apple with the other three stocks, a strong correlation between Google and Facebook, and a moderate positive correlation of these two with Microsoft. During this time, Apple already exhibits a negative development, while the other two continue to exhibit a positive development. Another non-trivial part of such an analysis is handling the time lag: by nature, rolling averages have to lag the data substantially. So while our approach enables interesting – and online – analysis of such data, it is far from being ready to be deployed without further studies.

5.3 Gaussian Mixture Modeling

We studied the quality and runtime of Gaussian Mixture Modeling using the popular Expectation-Maximization (EM) algorithm. We chose a very simple scenario: two Gaussian clusters in $\mathbb{R}^3$, with 50.000 points each, standard deviations $\frac{4}{3}$, 1, and $\frac{3}{4}$, and a random rotation to have a unique covariance matrix each. We vary only the placement of the two clusters by shifting the cluster centers diagonally away from the mean in order to provoke numerical instabilities. For a cluster center distance of < 2, the clusters will still overlap with a non-trivial amount, but at a distance of > 10 this data set is supposedly trivial to cluster, because the separation of the clusters is much larger than their diameters. We run each algorithm 11 times, and report the averages in the following.

We evaluate four different implementations here: the R "mclust" package, which probably is the most widely known implementation. The core of this package is written in Fortran, and it uses a two-pass algorithm. The Python "sklearn" version uses numpy, and also a two-pass algorithm. To make results more comparable, we disabled regularization and used random initialization as with the other implementations. Spark ML is of interest, because it uses the numerically problematic textbook approach to allow parallel processing. We run it single-core, and with all 8 threads supported by our i7-3770 CPU. ELKI uses a Welford-based incremental approach, closely related to Section 4.3. We add additional implementations to ELKI 0.7.1 [21] that use the two-pass, respectively the textbook algorithms. The modular Java architecture of ELKI made it easy to add the variants, keeping the remainder of the code unchanged. This way we can study the differences in isolation, which is a best-practise for benchmarking algorithms, and not implementations [13].

We evaluate the quality using two different metrics: Figure 7a evaluates the average log-likelihood of the Gaussian model, which is a goodness-of-fit measure internally used by the algorithm itself. The sudden changes we see in most curves are due to some of the 11 samples failing to find the best solution, but we can also see a gradual degradation in the Spark and sklearn implementations. In Figure 7b we instead evaluate the difference between the covariance matrix found by the algorithm, and the "true" covariance matrix computed during data generation with a high-precision method.



(a) Log-likelihood goodness of fit of the model

(b) Deviation from exact covariance matrix

(c) Runtime of Gaussian Mixture Modeling

Figure 7: Numerical issues in Gaussian Mixture Modeling

The results here are similar. Finally, in Figure 7c we compare the run-time of the different implementations (we tried our best to choose the same convergence thresholds, but nevertheless these numbers can only serve as a rough indicator.)

As long as the clusters are close to each other (with a distance of < 5), all implementations work well. After this, we can see the three textbook algorithms lose precision in the covariance matrix (Figure 7b) quickly. It is not obvious why the ELKI implementation nevertheless has a few digits of precision more than the Spark version. The loss in precision is approximately linear in log-log-space, as expected. The Welford-based approach also loses some precision, but at a much slower rate, while the two-pass implementations of R and ELKI remain close to the precision limit.

The numpy implementation exhibits erradic behavior once the clusters become too separated. We do not have an explanation for this behaviour. We can see the runtime increase substantially after a separation of > 2, at > 500 the numerical precision suddenly

drops (likely because the method often no longer finding the correct separation), and at $5 \cdot 10^6$ the implementation begins to fail with an exception. At this extreme separation, we do observe hickups in the ELKI implementations, too.

We can see two groups—possibly by chance—that exhibit similar runtime performance in Figure 7c, with the ELKI and numpy implementations outperforming the R and Spark implementations usually by a factor of 2–3. The only multi-core implementation (Spark) only reduces the runtime by a surprisingly small bit. Initially, the textbook implementation in ELKI is the fastest method, and in this range also with neglibile loss in accuracy. But the additional cost of the two-pass and Welford approaches is probably worth the gain in precision in other situations.

In conclusion of this experiment, we note that (i) the textbook approach should only be used when clusters are very close to the origin. At a distance of 10 standard deviations we already see a numeric loss in the precision of the covariance matrix, so it is supposedly not safe for clustering data sets with many clusters. (ii) the two-pass algorithm is a safe choice, with the best precision, and little performance overhead. (iii) for distributed systems such as Spark, an incremental approach as discussed in Section 4.3 is beneficial and will provide sufficient accuracy, while only requiring a single pass over the data. But we may require extremely large data sets, that no longer fit into the memory of a modern server, for this to be beneficial.

6 CONCLUSIONS AND FUTURE WORK

While numerical stability has been a concern for a long time, we found that modern tools such as the PostgreSQL database, scientific databases such as RasDaMan, and the computation platform Apache Spark still use numerically unstable computation methods, for example, in GMM clustering. In *closed-source* software, we will usually not even know which variant is used.[3] Our findings agree with earlier results by West [28]: if possible, the two-pass algorithm often is a good choice, also because of its simplicity. Only when we can visit data only once (such as in streaming data, or when we desire to use exponentially weighted moving averages) or when it is expensive to visit twice (e.g., because it is too large to keep in memory, or residing in a distributed storage) then incremental algorithms such as ours are beneficial. As seen in our experiments, it can sometimes be beneficial to perform the division last as in Youngs and Cramer's [30] approach, if the division cannot be shared for multiple variables. In this article, we provide optimized equations to merge covariance information from multiple partitions in a numerically reliable way, but that can be aggregated in a single-pass over the data. Using these equations, we can improve numerical precision of many methods with little additional computational overhead. We also suggest AVX and GPU parallelization and mini-batches to further improve precision and runtime.

Besides numerical stability and support for parallelization, our approach also allows for different weighting schemes, making this approach applicable for interesting analysis of correlation of times series, using exponentially weighted correlation.

[3] A simple test that fails, e.g., on Microsoft SQL and HyPer is: `SELECT VAR(x) FROM (SELECT 1000000000 AS x UNION SELECT 1000000001 AS x) t`.
PostgreSQL is correct for `VAR_SAMP(x)`$= \frac{1}{2}$ but returns 0 for `COVAR_SAMP(x,x)`.

REFERENCES

[1] E. Achtert, H.-P. Kriegel, E. Schubert, and A. Zimek. 2013. Interactive Data Mining with 3D-Parallel-Coordinate-Trees. In *Proc. SIGMOD*. 1009–1012. https://doi.org/10.1145/2463676.2463696

[2] S. Ahn and J. A. Fessler. 2003. *Standard errors of mean, variance, and standard deviation estimators.* Technical Report. EECS, University of Michigan.

[3] G. E. P. Box and J. S. Hunter. 1959. Condensed Calculations for Evolutionary Operation Programs. *Technometrics* 1, 1 (1959), 77–95.

[4] T. F. Chan, G. H. Golub, and R. J. LeVeque. 1982. Updating Formulae and a Pairwise Algorithm for Computing Sample Variances. In *COMPSTAT 1982*, H. Caussinus, P. Ettinger, and R. Tomassone (Eds.). 30–41. https://doi.org/10.1007/978-3-642-51461-6_3

[5] T. F. Chan, G. H. Golub, and R. J. LeVeque. 1983. Algorithms for Computing the Sample Variance: Analysis and Recommendations. *The American Statistician* 37, 3 (1983), 242–247. https://doi.org/10.2307/2683386

[6] T. F. Chan and J. G. Lewis. 1979. Computing Standard Deviations: Accuracy. *Commun. ACM* 22, 9 (1979), 526–531. https://doi.org/10.1145/359146.359152

[7] I. W. Cotton. 1975. Remark on Stably Updating Mean and Standard Deviation of Data. *Commun. ACM* 18, 8 (1975), 458. https://doi.org/10.1145/360933.360981

[8] A. Fog. 2017. *Instruction tables: Lists of instruction latencies, throughputs and micro-operation breakdowns for Intel, AMD and VIA CPUs.* Technical Report. Copenhagen University College of Engineering.

[9] R. J. Hanson. 1975. Stably Updating Mean and Standard Deviation of Data. *Commun. ACM* 18, 1 (1975), 57–58. https://doi.org/10.1145/360569.360662

[10] W. Kahan. 1965. Pracniques: further remarks on reducing truncation errors. *Commun. ACM* 8, 1 (1965), 40. https://doi.org/10.1145/363707.363723

[11] N. Kamat and A. Nandi. 2016. A Closer Look at Variance Implementations In Modern Database Systems. *SIGMOD Record* 45, 4 (2016), 28–33. https://doi.org/10.1145/3092931.3092936

[12] D. E. Knuth. 1981. *The Art of Computer Programming, Volume II: Seminumerical Algorithms, 2nd Edition.* Addison-Wesley.

[13] H.-P. Kriegel, E. Schubert, and A. Zimek. 2017. The (black) art of runtime evaluation: Are we comparing algorithms or implementations? *KAIS* 52, 2 (2017), 341–378. https://doi.org/10.1007/s10115-016-1004-2

[14] R. F. Ling. 1974. Comparison of Several Algorithms for Computing Sample Means and Variances. *J. Amer. Statist. Assoc.* 69, 348 (1974), 859–866. https://doi.org/10.2307/2286154

[15] P. M. Neely. 1966. Comparison of Several Algorithms for Computation of Means, Standard Deviations and Correlation Coefficients. *Commun. ACM* 9, 7 (1966), 496–499. https://doi.org/10.1145/365719.365958

[16] K. Pearson. 1901. On lines and planes of closest fit to systems of points in space. *Lond. Edinb. Dubl. Phil. Mag.* 2, 6 (1901), 559–572.

[17] P. Pébay. 2008. *Formulas for robust, one-pass parallel computation of covariances and arbitrary-order statistical moments.* Technical Report SAND2008-6212. Sandia National Laboratories.

[18] B. E. Rodden. 1967. Error-free methods for statistical computations. *Commun. ACM* 10, 3 (1967), 179–180. https://doi.org/10.1145/363162.363205

[19] J. L. Rodgers and W. A. Nicewander. 1988. Thirteen Ways to Look at the Correlation Coefficient. *The American Statistician* 42, 1 (1988), 59–66.

[20] E. Schubert. 2013. *Generalized and Efficient Outlier Detection for Spatial, Temporal, and High-Dimensional Data Mining.* Ph.D. Dissertation. Ludwig-Maximilians-Universität München, Munich, Germany.

[21] E. Schubert, A. Koos, T. Emrich, A. Züfle, K. A. Schmid, and A. Zimek. 2015. A Framework for Clustering Uncertain Data. *Proc. VLDB Endowment* 8, 12 (2015), 1976–1979. https://doi.org/10.14778/2824032.2824115

[22] E. Schubert, M. Weiler, and H.-P. Kriegel. 2014. SigniTrend: Scalable Detection of Emerging Topics in Textual Streams by Hashed Significance Thresholds. In *Proc. SIGKDD*. 871–880. https://doi.org/10.1145/2623330.2623740

[23] E. Schubert, M. Weiler, and H.-P. Kriegel. 2016. SPOTHOT: Scalable Detection of Geo-spatial Events in Large Textual Streams. In *Proc. SSDBM*. https://doi.org/10.1145/2949689.2949699

[24] J. R. Shewchuk. 1997. Adaptive Precision Floating-Point Arithmetic and Fast Robust Geometric Predicates. *Discrete & Computational Geometry* 18, 3 (1997), 305–368. https://doi.org/10.1007/PL00009321

[25] T. B. Terriberry. 2008. Computing higher-order moments online. (2008). Technical Note, http://people.xiph.org/~tterribe/notes/homs.html.

[26] A. J. Van Reeken. 1968. Letters to the editor: Dealing with Neely's algorithms. *Commun. ACM* 11, 3 (1968), 149–150. https://doi.org/10.1145/362929.362961

[27] B. P. Welford. 1962. Note on a Method for Calculating Corrected Sums of Squares and Products. *Technometrics* 4, 3 (1962), 419–420. https://doi.org/10.2307/1266577

[28] D. H. D. West. 1979. Updating mean and variance estimates: an improved method. *Commun. ACM* 22, 9 (1979), 532–535. https://doi.org/10.1145/359146.359153

[29] N. Whitehead and A. Fit-Florea. 2014. *Precision and Performance: Floating Point and IEEE 754 Compliance for NVIDIA GPUs.* TB-06711-001_v6.5. NVIDIA.

[30] E. A. Youngs and E. M. Cramer. 1971. Some Results Relevant to Choice of Sum and Sum-of-Product Algorithms. *Technometrics* 13, 3 (1971), 657–665. https://doi.org/10.1080/00401706.1971.10488826