# Basic Statistics


| | |
|---|---|
| **Document Number:** | P1708R9 |
| **Author:** | Richard Dosselmann: **dosselmr@uregina.ca** |
| **Contributors:** | Michael Chiu: chiu@cs.toronto.edu,<br>Guy Davidson: guy.davidson@hatcat.com<br>Oleksandr Koval: oleksandr.koval.dev@gmail.com<br>Larry Lewis: Larry.Lewis@sas.com<br>Johan Lundburg: lundberj@gmail.com<br>Jens Maurer: Jens.Maurer@gmx.net<br>Eric Niebler: eniebler@fb.com<br>Phillip Ratzloff: phil.ratzloff@sas.com<br>Vincent Reverdy: vreverdy@illinois.edu<br>John True<br>Michael Wong: michael@codeplay.com |
| **Date:** | October 2024 (mailing) |
| **Project:** | ISO JTC1/SC22/WG21: Programming Language C++ |
| **Audience:** | LEWG, SG6 |

# Contents

# 0 Revision History

**P1708R0**

- `https://github.com/cplusplus/papers/issues/475`

**P1708R1**

- An accumulator object is proposed to allow for the computation of statistics in a **single** pass over a sequence of values.

**P1708R2**

- Reformatted using LaTeX.
- A (possible) return to functions is proposed following discussions of the accumulator object of the previous version.

**P1708R3**

- **Geometric mean** is proposed, since it exists in Calc, Excel, Julia, MATLAB, Python, R and Rust.
- **Harmonic mean** is proposed, since it exists in Calc, Excel, Julia, MATLAB, PHP, Python, R and Rust.
- **Weighted means**, **median**, **mode**, **variances** and **standard deviations** are proposed, since they exist (with the exception of mode) in MATLAB and R.
- **Quantile** is proposed, since it is more generic than median and exists in Calc (percentile), Excel (percentile), Julia, MATLAB, PHP (percentile), R and SQL (percentile).
- **Skewness** is proposed, since it exists in Calc, Excel, Julia, MATLAB, PHP, R, Rust, SAS and SQL and was recommended as part of a presentation to SAS corporation.
- **Kurtosis** is proposed, since it exists in Calc, Excel, Julia, MATLAB, PHP, R, Rust, SAS and SQL and was recommended as part of a presentation to SAS corporation.
- Both **functions** and **accumulator objects** are proposed, since they (largely) have distinct purposes.
- **Iterator pairs** are replaced by **ranges**, since ranges simplify predicates (as comparisons and projections).

**P1708R4**

- Parameter `data_t` (corresponding to values `population_t` and `sample_t`) of **variance** and **standard deviation** are replaced by **delta degrees of freedom**, since this is done in Python (NumPy).
- In the case of a **quantile** (or median), specific methods of interpolation between adjacent values is proposed, since this is done in Python (NumPy).
- `stats_error`, previously a constant, is replaced by a **class**.

**P1708R5**

- **Quantile** (and **median**) and **mode** are deferred to a future proposal, given ongoing unresolved issues relating to these statistics.
- `stats_error`, an **exception**, is removed, since (C++) math funtions do not throw exceptions.
- The ability to create **custom** accumulator objects is proposed, since this is done in Boost Accumulators.
- `stats_result_t` is introduced so as to simplify (function) signatures.
- Various errors in statistical formulas are corrected.
- Various functions, objects (classes) and parameters are renamed so as to be more meaningful.
- Various technical errors relating to ranges and execution policy are corrected.

**P1708R6**

- **SG6** voted unanimously to forward to **LEWG** on April 14, 2022.
- `stats_result_t` is removed, since return type is deduced from projection.
- **Accumulator** objects are revised so as to be simpler and allow for parallel implementations.
- `stat_accum` and `weighted_stat_accum` are removed, since they are no longer needed.
- **Concepts** are removed so as to allow for **custom** data types.
- **Projections** are removed, since **views** already offer such functionality.
- Numerous functions and classes are renamed so as to be more meaningful.
- Reformatted so as to fulfill the specification style guidelines and **standardese**.

**P1708R7**

- **Unweighted** and **weighted** functions are combined so as to take advantage of **overloading**.
- The presentation of formulas is simplified.
- **Derivations** of skewness and kurtosis formulas are given.
- The wording of the technical specifications is updated.
- Further reformatted so as to fulfill the specification style guidelines and **standardese**.

**P1708R8**

- Statistics are reordered as first, second, third and fourth moments.
- **Constructors** are no longer **`explicit`**.
- `value` member function of accumulator objects is simplified.
- The name `stats` is replaced by the more meaningful name `statistics`.

**P1708R9**

- **LEWG** voted 14 / 4 / 5 / 0 / 1 to back **statistics** in C++ on March 20, 2024 in Tokyo.
- **Unweighted** and **weighted** variance (and standard deviation) are updated.
- **Unweighted** skewness and kurtosis are updated.
- **Weighted** skewness and kurtosis are removed, since they are highly specialized.
- Function overloads are introduced so as to easily allow the **data type** of a statistic to be changed.
- **Convenience** functions to simultaneously compute mean and variance (or standard deviation) are introduced.
- Accumulator objects are **aggregated** so as to reduce the run-time complexity of the (simultaneous) computation of multiple statistics.
- Various functions and parameters are renamed so as to be more meaningful.

# 1 Introduction

This document proposes an extension to the C++ library, to support **basic statistics**.

## 2 Motivation and Scope

Basic statistics, **not** presently found in the standard (including the special math library), frequently arise in **scientific** and **industrial**, as well as **general**, applications [1, 2, 3]. These functions do exist in Python [4], the foremost competitor to C++ in the area of **machine learning**, along with Calc [5], Excel [6], Julia [7], Maple [8], Mathematica [9], MATLAB [10], NumPy [11], Pandas [12], PHP [13], R [14], Rust [15], SAS [16], SciPy [17], SPSS [18], Stata [19] and SQL [20]. Further need for such functions has been identified as part of **SG19** (machine learning) [21].

This is not the first proposal to move statistics in C++. In 2004, a number of statistical distributions were proposed in [22]. Additional distributions followed in 2006 [23]. Statistical distributions ultimately appeared in the C++11 standard [24]. Distributions, along with statistical tests, are also found in Boost [25]. A C library, GNU Scientific Library [26], further includes support for statistics, special functions and histograms.

**Five** statistics are defined in this proposal. Two (univariate) statistics, specifically **percentile** (along with **quantile** and **median**) and **mode**, are **not** included in this proposal. These more involved statistics are deferred to a **future** proposal. Like existing entities of the (C++) standard library, this proposal specifies only the interface of functions and objects, meaning that a variety of implementations are possible. This enables a vendor to favor accuracy [27] over performance for instance. An **implementation**, released under the **MIT** license [28], is available at `https://github.com/dosselmann/statistics`

### 2.1 Mean

The *arithmetic mean* [29, 30], denoted $\mu$, of the $n \geq 1$ values $x_1, x_2, \ldots, x_n$ of a population [29], and $\bar{x}$ in the case of a sample [29], is defined as

$$\mu = \bar{x} = \frac{1}{n} \sum_{i=1}^{n} x_i. \tag{1}$$

The arithmetic mean is found in the **ISO** $3534-1{:}2006$ [31] standard and Python [4]. The *weighted arithmetic mean* [30, 32, 33, 34], for weights $w_1, w_2, \ldots, w_n$, is defined as

$$\mu_w = \bar{x}_w = \frac{1}{\sum_{i=1}^{n} w_i} \sum_{i=1}^{n} w_i x_i. \tag{2}$$

The weighted mean is found in Python [4]. The *geometric mean* [29], having a number of applications in science and technology [1] and found in Python [4], is defined as

$$\left( \prod_{i=1}^{n} x_i \right)^{\frac{1}{n}} \tag{3}$$

and the *weighted geometric mean* [32] is defined as

$$\left( \prod_{i=1}^{n} x_i^{w_i} \right)^{\left( \sum_{i=1}^{n} w_i \right)^{-1}}. \tag{4}$$

The *harmonic mean* [29] of positive values $x_i > 0$, also having many applications in science and technology [2] and found in Python [4], is defined as

$$\left( \frac{1}{n} \sum_{i=1}^{n} \frac{1}{x_i} \right)^{-1} \tag{5}$$

and the *weighted harmonic mean* [35] is defined as

$$\frac{\sum_{i=1}^{n} w_i}{\sum_{i=1}^{n} \frac{w_i}{x_i}}. \tag{6}$$

The weighted harmonic mean is found in Python [4]. Each of the arithmetic, geometric and harmonic means can be (accurately) computed in **linear** time [36]. When computing the associated sums of these means, and indeed any sum in this proposal, robust methods [37, 38] ought to be considered.

## 2.2 Variance

The population *variance* [39, 40, 41] of $n \geq 1$ values is

$$\sigma^2 = \frac{1}{n} \sum_{i=1}^{n} (x_i - \mu)^2 \tag{7}$$

and an unbiased estimator [40, 42] of the sample *variance* [29, 41, 43] of $n \geq 2$ values is

$$s^2 = \frac{1}{n-1} \sum_{i=1}^{n} (x_i - \bar{x})^2 . \tag{8}$$

The population and sample variance are found in the **ISO** $3534-1{:}2006$ standard and Python [4]. A population *weighted variance* [33] is

$$\sigma_w^2 = \frac{1}{\sum_{i=1}^{n} w_i} \sum_{i=1}^{n} w_i (x_i - \bar{x}_w)^2 \tag{9}$$

and an unbiased estimator of this sample weighted variance [44] is

$$s_{w,1}^2 = \frac{1}{\sum_{i=1}^{n} w_i - 1} \sum_{i=1}^{n} w_i (x_i - \bar{x}_w)^2 . \tag{10}$$

There are at least two other definitions of sample weighted variance (and standard deviation) in addition to that of $s_{w,1}^2$ of Equation (10). Note that $s_{w,1}^2$ is the version implemented in MATLAB [45], as well as an R package [46]. The first [47, 48] of these two is

$$s_{w,2}^2 = \frac{1}{\frac{\hat{n}-1}{\hat{n}} \sum_{i=1}^{n} w_i} \sum_{i=1}^{n} w_i (x_i - \bar{x}_w)^2 , \tag{11}$$

where $\hat{n} \geq 1$ is the number of non-zero weights $w_i$. A second definition of sample weighted variance [30, 49], also used in the case of *reliability weights* [50], is

$$s_{w,3}^2 = \frac{\sum_{i=1}^{n} w_i}{(\sum_{i=1}^{n} w_i)^2 - \sum_{i=1}^{n} w_i^2} \sum_{i=1}^{n} w_i (x_i - \bar{x}_w)^2 . \tag{12}$$

As there appears to be no common definition of (sample) weighted variance, **functions** to compute this statistic are not proposed. Instead, an **accumulator object** that can compute this statistic is proposed. Moving on, variance (and standard deviation) is computed using the terms $1/n$ and $1/(n-1)$. Other terms might be used instead [50], $1/(n-1.5)$ as an example [51, 52, 53]. To allow for such terms, this proposal, like NumPy [54], Pandas [55] and SciPy [56], enables one to specify *delta degrees of freedom* [54], a value subtracted from $n$. Variance (and standard deviation) can be computed in **linear** time [36, 57, 58].

## 2.3 Standard Deviation

The *standard deviation* [29, 41] of $n \geq 2$ values, denoted $s$, is defined as the square root of the variance. The population and sample standard deviation are found the **ISO** $3534-1{:}2006$ standard and Python [4]. Likewise, the *weighted standard deviation* [34], denoted $s_w$, is defined as the square root of the weighted variance.

## 2.4 Skewness

The population *skewness* [29, 59, 60, 61], a measure of the **asymmetry** [29] of $n \geq 1$ values, is

$$g_s = \frac{1}{n\sigma^3} \sum_{i=1}^{n} (x_i - \mu)^3 \tag{13}$$

and an unbiased estimator of the sample skewness [59], an *adjusted Fisher-Pearson standardized moment coefficient* [62, 63], of $n \geq 3$ values is

$$G_s = \frac{\sqrt{n(n-1)}}{n-2} g_s. \tag{14}$$

The population and sample skewness are found in the **ISO** $3534-1{:}2006$ standard. Skewness (and kurtosis) can be used to check if a dataset is unbalanced, as well as detect outliers [64]. Skewness (and kurtosis) can be computed in **linear** time [36, 65]. Weighted skewness [30, 66] is highly specialized and is therefore not considered in this proposal.

### 2.5 Kurtosis

The population *kurtosis* [29, 61, 67], a measure of the "**tailedness**" [68] of $n \geq 1$ values, is

$$g_k = \frac{1}{n\sigma^4} \sum_{i=1}^{n} (x_i - \mu)^4 \tag{15}$$

and the population *excess kurtosis* [60, 67] is

$$\hat{g}_k = \frac{1}{n\sigma^4} \sum_{i=1}^{n} (x_i - \mu)^4 - 3. \tag{16}$$

An unbiased estimator of the sample kurtosis [68, 69, 70, 71], itself too an adjusted Fisher-Pearson standardized moment coefficient [67], of $n \geq 4$ values, is

$$G_k = \frac{n(n+1)}{(n-1)(n-2)(n-3)s^4} \sum_{i=1}^{n} (x_i - \bar{x})^4 \tag{17}$$

and an unbiased estimator of the sample excess kurtosis [70] is

$$\hat{G}_k = \frac{n(n+1)}{(n-1)(n-2)(n-3)s^4} \sum_{i=1}^{n} (x_i - \bar{x})^4 - \frac{3(n-1)^2}{(n-2)(n-3)}. \tag{18}$$

The population and sample kurtosis are found in the **ISO** $3534-1{:}2006$ standard. Weighted kurtosis [30, 66] is highly specialized and is therefore not considered in this proposal.

## 3 Impact on the Standard

This proposal is a pure **library** extension.

## 4 Design Decisions

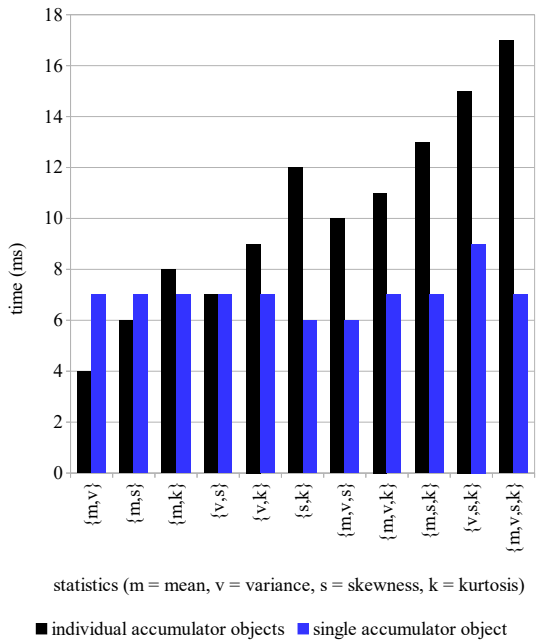
The discussions of the following sections address the concerns that have been raised in regards to this proposal.

### 4.1 Functions vs. Accumulator Objects

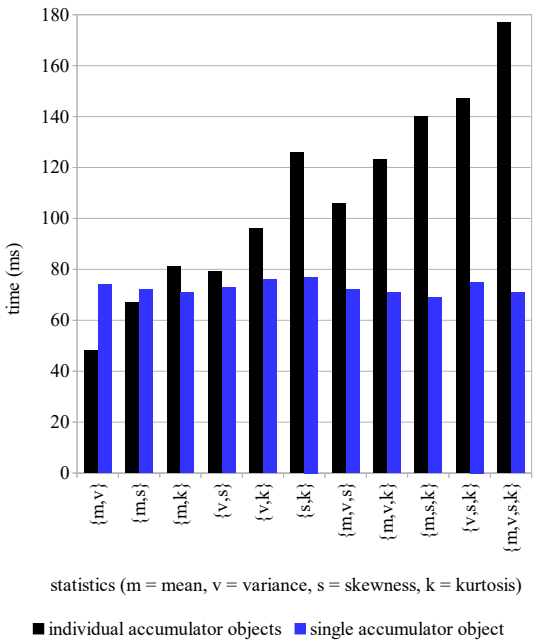
Perhaps the most significant concern stemming from this proposal is that of functions versus (accumulator) objects. In the first incarnation of this proposal, namely P1708R0, functions were (exclusively) proposed. **Functions** are useful when one wishes to compute a **single** statistic. **Objects** were introduced in P1708R1 and P1708R2, which allow a user to (efficiently) compute **more than one** statistic in a **single** pass over the values, an idea borrowed from Boost Accumulators [72]. Like Boost Accumulators, a programmer has the ability to create **custom** objects. As a result, a user can compute any of the proposed statistics, along with any custom statistics, in a single pass over the values. Given that each of these two paradigms has merit, with functions again being most useful in the case of the computation of a single statistic and objects being more attractive in instances in which multiple statistics are computed, the decision has been made to incorporate **both** such models into this proposal. Users are thus able to choose the approach that best fits with their design rather than being forced to use one of two paradigms.

Up until P1708R9, individual objects were proposed for each of the statistics of Section 2. These statistics are computed in much the same way [36], meaning that a large number of repeated computations is unfortunately performed. Individual objects have thus been **merged** for the sake of improved run-time performance. This improvement is seen in each of the four plots of Figure 1. As the four plots illustrate, there is little difference in the performance of individual objects and a single (merged) object in the case of the (simultaneous) computation of two statistics over random `double` values. Individual objects lag however in situations involving three or four statistics. The experiments of Figure 1 were carried out on a four-core Acer Intel® Core™ i5-4440 CPU running at 3.10 GHz with 16 GB of memory.

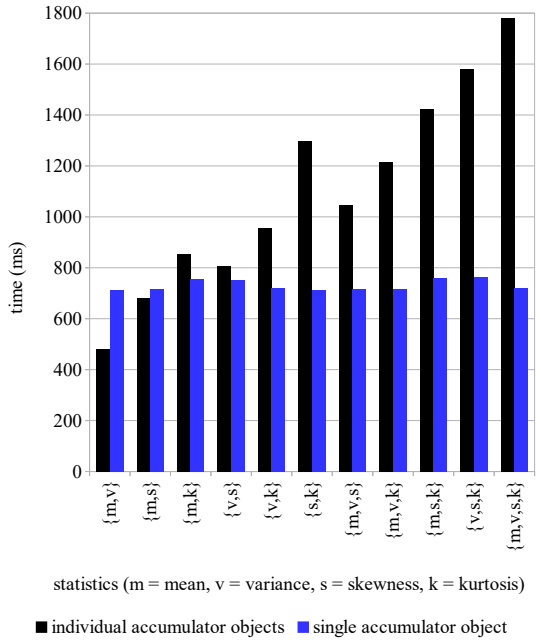
It has further been suggested that the **min** and **max** statistics be incorporated into the single object, given the routine need for such operations. Additional plots, shown in Figure 2, demonstrate the lag associated with the computation of the min and max. This lag stems from manner in which these two statistics are implemented. As shown in Example 5 of Appendix A, a costly Boolean check is needed during each accumulation to determine whether or not a given accumulation is the first such accumulation. An alternate design might require that a user invoke a special accumulate function the first time that an object is accumulated, which would result in a rather unusual design. These two statistics are therefore not proposed. The experiments of Figure 2 were carried out on the same hardware as those of Figure 1.



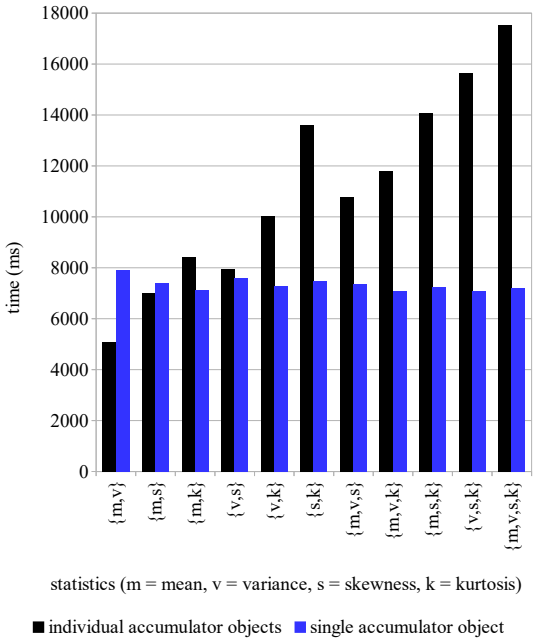

(a)




(b)




(c)




(d)

Figure 1: Run-time performance comparision of the computation of two, three and four statistics using individual accumulator objects versus a single accumulator object over $N$ random **`double`** values; (a) $N = 10^6$; (b) $N = 10^7$; (c) $N = 10^8$; (d) $N = 10^9$

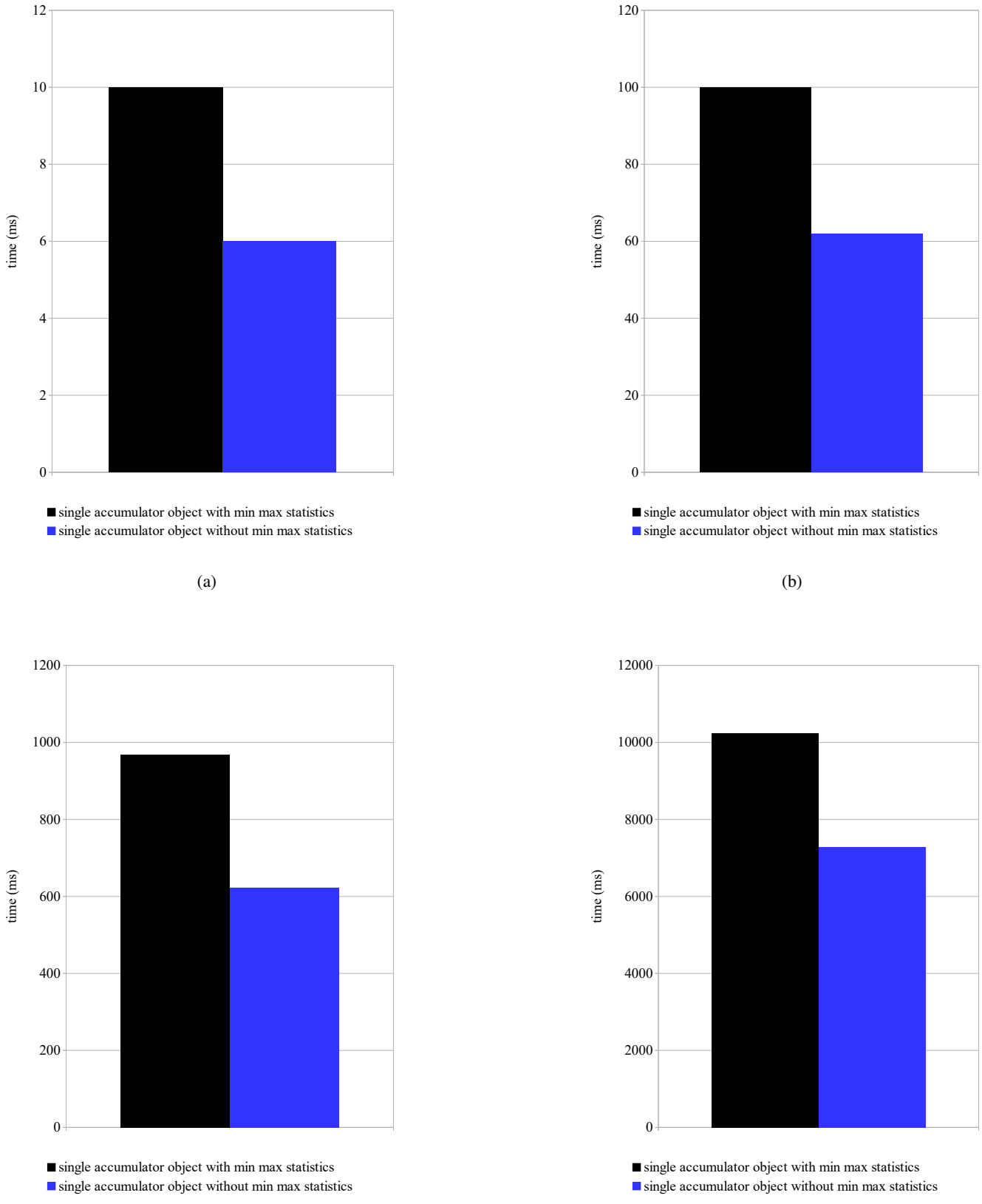



Figure 2: Run-time performance comparision of the computation of single accumulator objects with and without min and max operations over $N$ random **`double`** values; (a) $N = 10^6$; (b) $N = 10^7$; (c) $N = 10^8$; (d) $N = 10^9$

## 4.2 Trimmed Mean

The issue of a trimmed mean is raised in [73]. A $p\%$ *trimmed* mean [74] is one in which each of the $(p/2)\%$ **highest** and **lowest** values (of a **sorted** range) are excluded from the computation of that mean. This feature would require that the values of a given range either be **presorted** or **sorted** as part of the computation of a mean. As an author, Phillip Ratzloff feels (a sentiment that was echoed by the author of [73]) that one might handle this (and other similar) matter via **ranges**, specifically by using a statement of the form

```
auto m = data | std::ranges::sort | trim(p) | std::mean;
```

## 4.3 Special Values

Much like the question of the trimmed mean of the previous section, special values, such as $\pm\infty$ and **NaN**, are readily addressed using **ranges**, a motivating factor for the introduction of ranges into this proposal. As a result, a programmer might handle such values using, as an example, a statement of the form

```
auto m = data | std::ranges::filter([](auto x) { return !std::isnan(x); }) | std::mean;
```

## 4.4 Projections

The functions and objects of P1708R3, P1708R4 and P1708R5 employ projections as a means of accessing individual components of aggregate entities. Given that such functionality is available through the use of **views**, projections have been removed, thereby yielding simpler functions and objects. This is much like the approach suggested in Sections 4.2 and 4.3. An example that demonstrates the use of views is presented in Appendix A.

## 4.5 Concepts

Much like `std::complex`, the proposed (template) functions and objects are defined for each of the (C++) **arithmetic** types, **except** for **`bool`**. Also like `std::complex`, the effect of instantiating the templates for any other type is unspecified. A programmer can therefore attempt to use **custom** types with the proposed functions and objects. It is felt that the added flexibility afforded by not using **concepts** to strictly limit functions and objects to arithmetic types is in the interest of the C++ community. In fact, several concerned parties reached out to the author of this proposal in regard to this matter, all of whom suggested that this flexible approach be taken. Note that concepts are still employed in the case of **execution policy**, namely `std::is_execution_policy_v`, in which a fixed set of policies exists.

## 4.6 Header and Namespace

Early versions of this proposal, specifically P1708R0, P1708R1 and P17082, request that the proposed functions and objects be placed into the `<numeric>` header. Since P1708R3, it has instead been suggested that the function and objects be placed into a (new) **header** `<statistics>`, just as was done with the rational arithmetic of `<ratio>`, probability distributions of `<random>`, bit operations of `<bit>` and constants of `<numbers>`. Like rational arithmetic, probability distributions, bit operations and constants, basic statistics fit into the existing `std` **namespace**.

## 4.7 `std::expected`

The prospect of returning an `std::expected` object from the `value` member function of accumulator objects has been raised. Specifically, it has been suggested to do so in the case that there are too few elements in a range to which a particular accumulator is applied. Presently, `std::expected` objects are not used (elsewhere) in this proposal, such as in the case of a NaN, nor among the mathematical functions of the C++ library. It would therefore be asymmetric to have some situations return an `std::expected` object but not others.

## 4.8 Weighted Variance

As noted, there are multiple definitions of (sample) weighted variance, meaning that functions to compute this statistic are not considered in this proposal. Instead, (any desired definition of) weighted variance may be (indirectly) computed using the weighted **accumulator object** of Section 5, which computes the weighted second central moment [75], along with each of $\hat{n}$, $\sum_{i=1}^{n} w_i$ and $\sum_{i=1}^{n} w_i^2$. Accordingly, $\sigma_w^2$ may be computed as

```
std::weighted_accumulator<double> acc;

acc(R, W); // accumulate values of range R weighted by weights of range W

auto w        = static_cast<double>(acc.sum_of_weights());
auto sigma2_w = 1/w * acc.second_central_moment();
```

and $s^2_{w,1}$ may be computed as

```
auto w     = static_cast<double>(acc.sum_of_weights());
auto s2_w1 = 1/(w-1) * acc.second_central_moment();
```

Likewise, $s^2_{w,2}$ may be computed as

```
auto n     = static_cast<double>(acc.non_zero_count());
auto w     = static_cast<double>(acc.sum_of_weights());
auto s2_w2 = 1 / ((n-1)/n * w) * acc.second_central_moment();
```

Lastly, the computation of $s^2_{w,3}$ might look something like

```
auto w     = static_cast<double>(acc.sum_of_weights());
auto w2    = static_cast<double>(acc.sum_of_squared_weights());
auto s2_w3 = w / (w*w - w2) * acc.second_central_moment();
```

# 5 Technical Specifications

The templates of the classes and functions specified in this section are defined for each of the arithmetic types, except for **bool**. The effect of instantiating the templates for any other type is unspecified. Parallel function overloads follow the requirements of [algorithms.parallel].

## 5.1 Header <statistics> synopsis [statistics.syn]

```
#include <execution>

namespace std {

// functions

template<class T, ranges::input_range R>
constexpr auto mean(R&& r) -> T;

template<ranges::input_range R>
constexpr auto mean(R&& r) -> std::ranges::range_value_t<R>;

template<class T, ranges::input_range R, ranges::input_range W>
constexpr auto mean(R&& r, W&& w) -> T;

template<ranges::input_range R, ranges::input_range W>
constexpr auto mean(R&& r, W&& w) -> std::ranges::range_value_t<R>;

template<class ExecutionPolicy, class T, ranges::input_range R>
requires std::is_execution_policy_v<std::remove_cvref_t<ExecutionPolicy>>
auto mean(ExecutionPolicy&& policy, R&& r) -> T;

template<class ExecutionPolicy, ranges::input_range R>
requires std::is_execution_policy_v<std::remove_cvref_t<ExecutionPolicy>>
auto mean(ExecutionPolicy&& policy, R&& r) -> std::ranges::range_value_t<R>;
```

```
template<class ExecutionPolicy, class T, ranges::input_range R, ranges::input_range W>
requires std::is_execution_policy_v<std::remove_cvref_t<ExecutionPolicy>>
auto mean(ExecutionPolicy&& policy, R&& r, W&& w) -> T;

template<class ExecutionPolicy, ranges::input_range R, ranges::input_range W>
requires std::is_execution_policy_v<std::remove_cvref_t<ExecutionPolicy>>
auto mean(ExecutionPolicy&& policy, R&& r, W&& w) -> std::ranges::range_value_t<R>;

template<class T, ranges::input_range R>
constexpr auto geometric_mean(R&& r) -> T;

template<ranges::input_range R>
constexpr auto geometric_mean(R&& r) -> std::ranges::range_value_t<R>;

template<class T, ranges::input_range R, ranges::input_range W>
constexpr auto geometric_mean(R&& r, W&& w) -> T;

template<ranges::input_range R, ranges::input_range W>
constexpr auto geometric_mean(R&& r, W&& w) -> std::ranges::range_value_t<R>;

template<class ExecutionPolicy, class T, ranges::input_range R>
requires std::is_execution_policy_v<std::remove_cvref_t<ExecutionPolicy>>
auto geometric_mean(ExecutionPolicy&& policy, R&& r) -> T;

template<class ExecutionPolicy, ranges::input_range R>
requires std::is_execution_policy_v<std::remove_cvref_t<ExecutionPolicy>>
auto geometric_mean(ExecutionPolicy&& policy, R&& r) -> std::ranges::range_value_t<R>;

template<class ExecutionPolicy, class T, ranges::input_range R, ranges::input_range W>
requires std::is_execution_policy_v<std::remove_cvref_t<ExecutionPolicy>>
auto geometric_mean(ExecutionPolicy&& policy, R&& r, W&& w) -> T;

template<class ExecutionPolicy, ranges::input_range R, ranges::input_range W>
requires std::is_execution_policy_v<std::remove_cvref_t<ExecutionPolicy>>
auto geometric_mean(
  ExecutionPolicy&& policy, R&& r, W&& w) -> std::ranges::range_value_t<R>;

template<class T, ranges::input_range R>
constexpr auto harmonic_mean(R&& r) -> T;

template<ranges::input_range R>
constexpr auto harmonic_mean(R&& r) -> std::ranges::range_value_t<R>;

template<class T, ranges::input_range R, ranges::input_range W>
constexpr auto harmonic_mean(R&& r, W&& w) -> T;

template<ranges::input_range R, ranges::input_range W>
constexpr auto harmonic_mean(R&& r, W&& w) -> std::ranges::range_value_t<R>;

template<class ExecutionPolicy, class T, ranges::input_range R>
requires std::is_execution_policy_v<std::remove_cvref_t<ExecutionPolicy>>
auto harmonic_mean(ExecutionPolicy&& policy, R&& r) -> T;

template<class ExecutionPolicy, ranges::input_range R>
requires std::is_execution_policy_v<std::remove_cvref_t<ExecutionPolicy>>
auto harmonic_mean(ExecutionPolicy&& policy, R&& r) -> std::ranges::range_value_t<R>;

template<class ExecutionPolicy, class T, ranges::input_range R, ranges::input_range W>
```

```
requires std::is_execution_policy_v<std::remove_cvref_t<ExecutionPolicy>>
auto harmonic_mean(ExecutionPolicy&& policy, R&& r, W&& w) -> T;

template<class ExecutionPolicy, ranges::input_range R, ranges::input_range W>
requires std::is_execution_policy_v<std::remove_cvref_t<ExecutionPolicy>>
auto harmonic_mean(
  ExecutionPolicy&& policy, R&& r, W&& w) -> std::ranges::range_value_t<R>;

template<class T, ranges::input_range R>
constexpr auto variance(R&& r, T ddof = T(1)) -> T;

template<ranges::input_range R>
constexpr auto variance(
  R&& r, std::ranges::range_value_t<R> ddof = std::ranges::range_value_t<R>(1)) ->
  std::ranges::range_value_t<R>;

template<class ExecutionPolicy, class T, ranges::input_range R>
requires std::is_execution_policy_v<std::remove_cvref_t<ExecutionPolicy>>
auto variance(ExecutionPolicy&& policy, R&& r, T ddof = T(1)) -> T;

template<class ExecutionPolicy, ranges::input_range R>
requires std::is_execution_policy_v<std::remove_cvref_t<ExecutionPolicy>>
auto variance(
  ExecutionPolicy&& policy,
  R&& r,
  std::ranges::range_value_t<R> ddof = std::ranges::range_value_t<R>(1)) ->
    std::ranges::range_value_t<R>;

template<class T, ranges::input_range R>
constexpr auto standard_deviation(R&& r, T ddof = T(1)) -> T;

template<ranges::input_range R>
constexpr auto standard_deviation(
  R&& r, std::ranges::range_value_t<R> ddof = std::ranges::range_value_t<R>(1)) ->
    std::ranges::range_value_t<R>;

template<class ExecutionPolicy, class T, ranges::input_range R>
requires std::is_execution_policy_v<std::remove_cvref_t<ExecutionPolicy>>
auto standard_deviation(ExecutionPolicy&& policy, R&& r, T ddof = T(1)) -> T;

template<class ExecutionPolicy, ranges::input_range R>
requires std::is_execution_policy_v<std::remove_cvref_t<ExecutionPolicy>>
auto standard_deviation(
  ExecutionPolicy&& policy,
  R&& r,
  std::ranges::range_value_t<R> ddof = std::ranges::range_value_t<R>(1)) ->
    std::ranges::range_value_t<R>;

template<class T, ranges::input_range R>
constexpr auto mean_variance(R&& r, T ddof = T(1)) -> std::pair<T,T>;

template<ranges::input_range R>
constexpr auto mean_variance(
  R&& r, std::ranges::range_value_t<R> ddof = std::ranges::range_value_t<R>(1)) ->
    std::pair<std::ranges::range_value_t<R>, std::ranges::range_value_t<R>>;

template<class ExecutionPolicy, class T, ranges::input_range R>
requires std::is_execution_policy_v<std::remove_cvref_t<ExecutionPolicy>>
```

```
auto mean_variance(ExecutionPolicy&& policy, R&& r, T ddof = T(1)) -> std::pair<T,T>;

template<class ExecutionPolicy, ranges::input_range R>
requires std::is_execution_policy_v<std::remove_cvref_t<ExecutionPolicy>>
auto mean_variance(
  ExecutionPolicy&& policy,
  R&& r,
  std::ranges::range_value_t<R> ddof = std::ranges::range_value_t<R>(1)) ->
    std::pair<std::ranges::range_value_t<R>, std::ranges::range_value_t<R>>;

template<class T, ranges::input_range R>
constexpr auto mean_standard_deviation(R&& r, T ddof = T(1)) -> std::pair<T,T>;

template<ranges::input_range R>
constexpr auto mean_standard_deviation(
  R&& r, std::ranges::range_value_t<R> ddof = std::ranges::range_value_t<R>(1)) ->
    std::pair<std::ranges::range_value_t<R>, std::ranges::range_value_t<R>>;

template<class ExecutionPolicy, class T, ranges::input_range R>
requires std::is_execution_policy_v<std::remove_cvref_t<ExecutionPolicy>>
auto mean_standard_deviation(ExecutionPolicy&& policy, R&& r, T ddof = T(1)) ->
  std::pair<T,T>;

template<class ExecutionPolicy, ranges::input_range R>
requires std::is_execution_policy_v<std::remove_cvref_t<ExecutionPolicy>>
auto mean_standard_deviation(
  ExecutionPolicy&& policy,
  R&& r,
  std::ranges::range_value_t<R> ddof = std::ranges::range_value_t<R>(1)) ->
    std::pair<std::ranges::range_value_t<R>, std::ranges::range_value_t<R>>;

template<class T, ranges::input_range R>
constexpr auto skewness(R&& r, bool sample=true) -> T;

template<ranges::input_range R>
constexpr auto skewness(R&& r, bool sample=true) -> std::ranges::range_value_t<R>;

template<class ExecutionPolicy, class T, ranges::input_range R>
requires std::is_execution_policy_v<std::remove_cvref_t<ExecutionPolicy>>
constexpr auto skewness(ExecutionPolicy&& policy, R&& r, bool sample=true) -> T;

template<class ExecutionPolicy, ranges::input_range R>
requires std::is_execution_policy_v<std::remove_cvref_t<ExecutionPolicy>>
constexpr auto skewness(
  ExecutionPolicy&& policy, R&& r, bool sample=true) -> std::ranges::range_value_t<R>;

template<class T, ranges::input_range R>
constexpr auto kurtosis(R&& r, bool sample=true, bool excess=true) -> T;

template<ranges::input_range R>
constexpr auto kurtosis(R&& r, bool sample=true, bool excess=true) ->
  std::ranges::range_value_t<R>;

template<class ExecutionPolicy, class T, ranges::input_range R>
requires std::is_execution_policy_v<std::remove_cvref_t<ExecutionPolicy>>
constexpr auto kurtosis(
  ExecutionPolicy&& policy, R&& r, bool sample=true, bool excess=true) -> T;
```

```
template<class ExecutionPolicy, ranges::input_range R>
requires std::is_execution_policy_v<std::remove_cvref_t<ExecutionPolicy>>
constexpr auto kurtosis(
  ExecutionPolicy&& policy, R&& r, bool sample=true, bool excess=true) ->
    std::ranges::range_value_t<R>;

// accumulator objects

template<class T>
class unweighted_accumulator
{
public:
  constexpr unweighted_accumulator() noexcept;

  constexpr size_t count() const noexcept;

  constexpr void operator()(const T& x);

  template<ranges::input_range R> constexpr void operator()(R&& r);

  constexpr auto second_central_moment() const noexcept -> T;
  constexpr auto third_central_moment()  const noexcept -> T;
  constexpr auto fourth_central_moment() const noexcept -> T;

  constexpr auto mean()                                       const noexcept -> T;
  constexpr auto variance(T ddof = T(1))                      const noexcept -> T;
  constexpr auto standard_deviation(T ddof = T(1))            const noexcept -> T;
  constexpr auto skewness(bool sample=true)                   const noexcept -> T;
  constexpr auto kurtosis(bool sample=true, bool excess=true) const noexcept -> T;
};

template<class T, class W = T>
class weighted_accumulator
{
public:
  constexpr weighted_accumulator() noexcept;

  constexpr W      sum_of_weights()         const noexcept;
  constexpr W      sum_of_squared_weights() const noexcept;
  constexpr size_t non_zero_count()         const noexcept;

  constexpr void operator()(const T& x, const W& w);

  template<ranges::input_range R, ranges::input_range W>
  constexpr void operator()(R&& r, W&& w);

  constexpr auto second_central_moment() const noexcept -> T;

  constexpr auto mean() const noexcept -> T;
};

}
```

## 5.2 Functions

If, first, any of the values of the ranges `r` or `w` of the functions specified in this section is a NaN, $\infty$ or $-\infty$, secondly, NaN, $\infty$ or $-\infty$ occurs, or, thirdly, overflow or underflow occurs, which might even occur in the case of finite ranges of values, the function returns an unspecified value.

### 5.2.1 Mean Functions

```
template<class T, ranges::input_range R>
constexpr auto mean(R&& r) -> T;

template<ranges::input_range R>
constexpr auto mean(R&& r) -> std::ranges::range_value_t<R>;

template<class T, ranges::input_range R, ranges::input_range W>
constexpr auto mean(R&& r, W&& w) -> T;

template<ranges::input_range R, ranges::input_range W>
constexpr auto mean(R&& r, W&& w) -> std::ranges::range_value_t<R>;

template<class ExecutionPolicy, class T, ranges::input_range R>
requires std::is_execution_policy_v<std::remove_cvref_t<ExecutionPolicy>>
auto mean(ExecutionPolicy&& policy, R&& r) -> T;

template<class ExecutionPolicy, ranges::input_range R>
requires std::is_execution_policy_v<std::remove_cvref_t<ExecutionPolicy>>
auto mean(ExecutionPolicy&& policy, R&& r) -> std::ranges::range_value_t<R>;

template<class ExecutionPolicy, class T, ranges::input_range R, ranges::input_range W>
requires std::is_execution_policy_v<std::remove_cvref_t<ExecutionPolicy>>
auto mean(ExecutionPolicy&& policy, R&& r, W&& w) -> T;

template<class ExecutionPolicy, ranges::input_range R, ranges::input_range W>
requires std::is_execution_policy_v<std::remove_cvref_t<ExecutionPolicy>>
auto mean(ExecutionPolicy&& policy, R&& r, W&& w) -> std::ranges::range_value_t<R>;
```

1. *Preconditions*: `r` and `w` are ranges of finite values, where `r` has at least 1 value and the length of `r` is less than or equal to the length of `w`.
2. *Returns*: The (weighted) arithmetic mean of the values of `r` (weighted by the corresponding values of `w`) if the preconditions have been met and an unspecified value otherwise.
3. *Complexity*: Linear in `ranges::distance(r)`.

### 5.2.2 Geometric Mean Functions

```
template<class T, ranges::input_range R>
constexpr auto geometric_mean(R&& r) -> T;

template<ranges::input_range R>
constexpr auto geometric_mean(R&& r) -> std::ranges::range_value_t<R>;

template<class T, ranges::input_range R, ranges::input_range W>
constexpr auto geometric_mean(R&& r, W&& w) -> T;

template<ranges::input_range R, ranges::input_range W>
constexpr auto geometric_mean(R&& r, W&& w) -> std::ranges::range_value_t<R>;

template<class ExecutionPolicy, class T, ranges::input_range R>
requires std::is_execution_policy_v<std::remove_cvref_t<ExecutionPolicy>>
auto geometric_mean(ExecutionPolicy&& policy, R&& r) -> T;

template<class ExecutionPolicy, ranges::input_range R>
requires std::is_execution_policy_v<std::remove_cvref_t<ExecutionPolicy>>
```

```
auto geometric_mean(ExecutionPolicy&& policy, R&& r) -> std::ranges::range_value_t<R>;

template<class ExecutionPolicy, class T, ranges::input_range R, ranges::input_range W>
requires std::is_execution_policy_v<std::remove_cvref_t<ExecutionPolicy>>
auto geometric_mean(ExecutionPolicy&& policy, R&& r, W&& w) -> T;

template<class ExecutionPolicy, ranges::input_range R, ranges::input_range W>
requires std::is_execution_policy_v<std::remove_cvref_t<ExecutionPolicy>>
auto geometric_mean(
  ExecutionPolicy&& policy, R&& r, W&& w) -> std::ranges::range_value_t<R>;
```

1. *Preconditions*: `r` and `w` are ranges of finite values, where `r` has at least 1 value and the length of `r` is less than or equal to the length of `w`, and, if the product of the values of `r` is negative, then `ranges::distance(r)` is odd.
2. *Returns*: The (weighted) geometric mean of the values of `r` (weighted by the corresponding values of `w`) if the preconditions have been met and an unspecified value otherwise.
3. *Complexity*: Linear in `ranges::distance(r)`.

#### 5.2.3 Harmonic Mean Functions

```
template<class T, ranges::input_range R>
constexpr auto harmonic_mean(R&& r) -> T;

template<ranges::input_range R>
constexpr auto harmonic_mean(R&& r) -> std::ranges::range_value_t<R>;

template<class T, ranges::input_range R, ranges::input_range W>
constexpr auto harmonic_mean(R&& r, W&& w) -> T;

template<ranges::input_range R, ranges::input_range W>
constexpr auto harmonic_mean(R&& r, W&& w) -> std::ranges::range_value_t<R>;

template<class ExecutionPolicy, class T, ranges::input_range R>
requires std::is_execution_policy_v<std::remove_cvref_t<ExecutionPolicy>>
auto harmonic_mean(ExecutionPolicy&& policy, R&& r) -> T;

template<class ExecutionPolicy, ranges::input_range R>
requires std::is_execution_policy_v<std::remove_cvref_t<ExecutionPolicy>>
auto harmonic_mean(ExecutionPolicy&& policy, R&& r) -> std::ranges::range_value_t<R>;

template<class ExecutionPolicy, class T, ranges::input_range R, ranges::input_range W>
requires std::is_execution_policy_v<std::remove_cvref_t<ExecutionPolicy>>
auto harmonic_mean(ExecutionPolicy&& policy, R&& r, W&& w) -> T;

template<class ExecutionPolicy, ranges::input_range R, ranges::input_range W>
requires std::is_execution_policy_v<std::remove_cvref_t<ExecutionPolicy>>
auto harmonic_mean(
  ExecutionPolicy&& policy, R&& r, W&& w) -> std::ranges::range_value_t<R>;
```

1. *Preconditions*: `r` and `w` are ranges of finite values, where `r` has at least 1 value, the length of `r` is less than or equal to the length of `w` and all of the values of `r` are positive.
2. *Returns*: The (weighted) harmonic mean of the values of `r` (weighted by the corresponding values of `w`) if the preconditions have been met and an unspecified value otherwise.
3. *Complexity*: Linear in `ranges::distance(r)`.

#### 5.2.4 Variance Functions

```
template<class T, ranges::input_range R>
constexpr auto variance(R&& r, T ddof = T(1)) -> T;

template<ranges::input_range R>
constexpr auto variance(
  R&& r, std::ranges::range_value_t<R> ddof = std::ranges::range_value_t<R>(1)) ->
  std::ranges::range_value_t<R>;

template<class ExecutionPolicy, class T, ranges::input_range R>
requires std::is_execution_policy_v<std::remove_cvref_t<ExecutionPolicy>>
auto variance(ExecutionPolicy&& policy, R&& r, T ddof = T(1)) -> T;

template<class ExecutionPolicy, ranges::input_range R>
requires std::is_execution_policy_v<std::remove_cvref_t<ExecutionPolicy>>
auto variance(
  ExecutionPolicy&& policy,
  R&& r,
  std::ranges::range_value_t<R> ddof = std::ranges::range_value_t<R>(1)) ->
    std::ranges::range_value_t<R>;
```

1. *Preconditions*: `r` is a range of finite values, where `r` has at least 1 value, and `ddof` is not equal to the length of `r`.
2. *Returns*: The variance of the values of `r` if the preconditions have been met and an unspecified value otherwise.
3. *Complexity*: Linear in `ranges::distance(r)`.

#### 5.2.5 Standard Deviation Functions

```
template<class T, ranges::input_range R>
constexpr auto standard_deviation(R&& r, T ddof = T(1)) -> T;

template<ranges::input_range R>
constexpr auto standard_deviation(
  R&& r, std::ranges::range_value_t<R> ddof = std::ranges::range_value_t<R>(1)) ->
    std::ranges::range_value_t<R>;

template<class ExecutionPolicy, class T, ranges::input_range R>
requires std::is_execution_policy_v<std::remove_cvref_t<ExecutionPolicy>>
auto standard_deviation(ExecutionPolicy&& policy, R&& r, T ddof = T(1)) -> T;

template<class ExecutionPolicy, ranges::input_range R>
requires std::is_execution_policy_v<std::remove_cvref_t<ExecutionPolicy>>
auto standard_deviation(
  ExecutionPolicy&& policy,
  R&& r,
  std::ranges::range_value_t<R> ddof = std::ranges::range_value_t<R>(1)) ->
    std::ranges::range_value_t<R>;
```

1. *Preconditions*: `r` is a range of finite values, where `r` has at least 1 value, and `ddof` is not equal to the length of `r`.
2. *Returns*: The standard deviation of the values of `r` if the preconditions have been met and an unspecified value otherwise.
3. *Complexity*: Linear in `ranges::distance(r)`.

### 5.2.6 Combined Mean and Variance and Standard Deviation Functions

```
template<class T, ranges::input_range R>
constexpr auto mean_variance(R&& r, T ddof = T(1)) -> std::pair<T,T>;

template<ranges::input_range R>
constexpr auto mean_variance(
  R&& r, std::ranges::range_value_t<R> ddof = std::ranges::range_value_t<R>(1)) ->
    std::pair<std::ranges::range_value_t<R>, std::ranges::range_value_t<R>>;

template<class ExecutionPolicy, class T, ranges::input_range R>
requires std::is_execution_policy_v<std::remove_cvref_t<ExecutionPolicy>>
auto mean_variance(ExecutionPolicy&& policy, R&& r, T ddof = T(1)) -> std::pair<T,T>;

template<class ExecutionPolicy, ranges::input_range R>
requires std::is_execution_policy_v<std::remove_cvref_t<ExecutionPolicy>>
auto mean_variance(
  ExecutionPolicy&& policy,
  R&& r,
  std::ranges::range_value_t<R> ddof = std::ranges::range_value_t<R>(1)) ->
    std::pair<std::ranges::range_value_t<R>, std::ranges::range_value_t<R>>;

template<class T, ranges::input_range R>
constexpr auto mean_standard_deviation(R&& r, T ddof = T(1)) -> std::pair<T,T>;

template<ranges::input_range R>
constexpr auto mean_standard_deviation(
  R&& r, std::ranges::range_value_t<R> ddof = std::ranges::range_value_t<R>(1)) ->
    std::pair<std::ranges::range_value_t<R>, std::ranges::range_value_t<R>>;

template<class ExecutionPolicy, class T, ranges::input_range R>
requires std::is_execution_policy_v<std::remove_cvref_t<ExecutionPolicy>>
auto mean_standard_deviation(ExecutionPolicy&& policy, R&& r, T ddof = T(1)) ->
  std::pair<T,T>;

template<class ExecutionPolicy, ranges::input_range R>
requires std::is_execution_policy_v<std::remove_cvref_t<ExecutionPolicy>>
auto mean_standard_deviation(
  ExecutionPolicy&& policy,
  R&& r,
  std::ranges::range_value_t<R> ddof = std::ranges::range_value_t<R>(1)) ->
    std::pair<std::ranges::range_value_t<R>, std::ranges::range_value_t<R>>;
```

1. *Preconditions*: `r` is a range of finite values, where `r` has at least 1 value, and `ddof` is not equal to the length of `r`.
2. *Returns*: The mean and variance (or standard deviation) of the values of `r` if the preconditions have been met and an unspecified value otherwise.
3. *Complexity*: Linear in `ranges::distance(r)`.

### 5.2.7 Skewness Functions

```
template<class T, ranges::input_range R>
constexpr auto skewness(R&& r, bool sample=true) -> T;

template<ranges::input_range R>
constexpr auto skewness(R&& r, bool sample=true) -> std::ranges::range_value_t<R>;
```

```
template<class ExecutionPolicy, class T, ranges::input_range R>
requires std::is_execution_policy_v<std::remove_cvref_t<ExecutionPolicy>>
constexpr auto skewness(ExecutionPolicy&& policy, R&& r, bool sample=true) -> T;

template<class ExecutionPolicy, ranges::input_range R>
requires std::is_execution_policy_v<std::remove_cvref_t<ExecutionPolicy>>
constexpr auto skewness(
  ExecutionPolicy&& policy, R&& r, bool sample=true) -> std::ranges::range_value_t<R>;
```

1. *Preconditions*: `r` is a range of finite values, where `r` has at least 3 values if `sample` is **`true`** and 1 otherwise.
2. *Returns*: An unbiased sample estimator of the skewness of the values of `r` if `sample` is **`true`** and the population skewness otherwise, if the preconditions have been met and an unspecified value otherwise.
3. *Complexity*: Linear in `ranges::distance(r)`.

#### 5.2.8 Kurtosis Functions

```
template<class T, ranges::input_range R>
constexpr auto kurtosis(R&& r, bool sample=true, bool excess=true) -> T;

template<ranges::input_range R>
constexpr auto kurtosis(R&& r, bool sample=true, bool excess=true) ->
  std::ranges::range_value_t<R>;

template<class ExecutionPolicy, class T, ranges::input_range R>
requires std::is_execution_policy_v<std::remove_cvref_t<ExecutionPolicy>>
constexpr auto kurtosis(
  ExecutionPolicy&& policy, R&& r, bool sample=true, bool excess=true) -> T;

template<class ExecutionPolicy, ranges::input_range R>
requires std::is_execution_policy_v<std::remove_cvref_t<ExecutionPolicy>>
constexpr auto kurtosis(
  ExecutionPolicy&& policy, R&& r, bool sample=true, bool excess=true) ->
    std::ranges::range_value_t<R>;
```

1. *Preconditions*: `r` is a range of finite values, where `r` has at least 4 values if `sample` is **`true`** and 1 otherwise.
2. *Returns*: An unbiased sample estimator of the kurtosis of the values of `r` if `sample` is **`true`** and the population kurtosis otherwise, if the preconditions have been met and an unspecified value otherwise.
3. *Complexity*: Linear in `ranges::distance(r)`.

### 5.3 Accumulator Objects

The accumulator objects specified in this section are trivially copyable if `T` is trivially copyable. If, first, any of the values of `x` or `w` of the **`operator`**`()` is a NaN (Not a Number), $\infty$ or $-\infty$, secondly, NaN, $\infty$ or $-\infty$ occurs, or, thirdly, overflow or underflow occurs, which might even occur in the case of finite ranges of values, the functions `second_central_moment`, `third_central_moment`, `fourth_central_moment`, `mean`, `variance`, `standard_deviation`, `skewness` and `kurtosis` return an unspecified value. The `mean`, `variance`, `standard_deviation`, `skewness` and `kurtosis` functions of the accumulator objects shall return the same values as the `mean`, `variance`, `standard_deviation`, `skewness` and `kurtosis` functions, respectively, of Section 5.2.

#### 5.3.1 Unweighted Accumulator Object

```
template<class T>
class unweighted_accumulator
{
```

```
public:
  constexpr unweighted_accumulator() noexcept;

  constexpr size_t count() const noexcept;

  constexpr void operator()(const T& x);

  template<ranges::input_range R> constexpr void operator()(R&& r);

  constexpr auto second_central_moment() const noexcept -> T;
  constexpr auto third_central_moment()  const noexcept -> T;
  constexpr auto fourth_central_moment() const noexcept -> T;

  constexpr auto mean()                                          const noexcept -> T;
  constexpr auto variance(T ddof = T(1))                         const noexcept -> T;
  constexpr auto standard_deviation(T ddof = T(1))               const noexcept -> T;
  constexpr auto skewness(bool sample=true)                      const noexcept -> T;
  constexpr auto kurtosis(bool sample=true, bool excess=true) const noexcept -> T;
};
```

```
constexpr unweighted_accumulator() noexcept;
```

1. *Effects*: An unweighted accumulator object is constructed.
2. *Complexity*: Constant.

```
constexpr size_t count() const noexcept;
```

1. *Returns*: The number of invocations of the accumulator object.
2. *Complexity*: Constant.

```
constexpr void operator()(const T& x);
```

1. *Preconditions*: `x` is a finite value.
2. *Effects*: The value of `x` is accumulated.
3. *Complexity*: Constant.

```
template<ranges::input_range R> constexpr void operator()(R&& r);
```

1. *Preconditions*: `r` is a range of finite values.
2. *Effects*: The values of `r` are accumulated.
3. *Complexity*: Linear in `ranges::distance(r)`.

```
constexpr auto second_central_moment() const noexcept -> T;
constexpr auto third_central_moment()  const noexcept -> T;
constexpr auto fourth_central_moment() const noexcept -> T;
```

1. *Preconditions*: At least 2 (or 3 or 4) invocations of **`operator()`**.
2. *Returns*: The second (or third or fourth) central moment of the accumulated values if the preconditions have been met and an unspecified value otherwise.

3. *Complexity*: Constant.

```
constexpr auto mean() const noexcept -> T;
```

1. *Preconditions*: At least 1 invocation of **operator**().
2. *Returns*: The arithmetic mean of the accumulated values if the preconditions have been met and an unspecified value otherwise.
3. *Complexity*: Constant.

```
constexpr auto variance(T ddof = T(1)) const noexcept -> T;
```

1. *Preconditions*: At least 1 invocation of **operator**() and ddof is not equal to the number of invocations.
2. *Returns*: The variance of the accumulated values if the preconditions have been met and an unspecified value otherwise.
3. *Complexity*: Constant.

```
constexpr auto standard_deviation(T ddof = T(1)) const noexcept -> T;
```

1. *Preconditions*: At least 1 invocation of **operator**() and ddof is not equal to the number of invocations.
2. *Returns*: The standard deviation of the accumulated values if the preconditions have been met and an unspecified value otherwise.
3. *Complexity*: Constant.

```
constexpr auto skewness(bool sample=true) const noexcept -> T;
```

1. *Preconditions*: At least 3 invocations of **operator**() if sample is **true** and 1 otherwise.
2. *Returns*: An unbiased sample estimator of the skewness of the accumulated values if sample is **true** and the population skewness otherwise, if the preconditions have been met and an unspecified value otherwise.
3. *Complexity*: Constant.

```
constexpr auto kurtosis(bool sample=true, bool excess=true) const noexcept -> T;
```

1. *Preconditions*: At least 4 invocations of **operator**() if sample is **true** and 1 otherwise.
2. *Returns*: An unbiased sample estimator of the kurtosis of the accumulated values if sample is **true** and the population kurtosis otherwise, if the preconditions have been met and an unspecified value otherwise.
3. *Complexity*: Constant.

#### 5.3.2 Weighted Accumulator Object

```
template<class T, class W = T>
class weighted_accumulator
{
public:
  constexpr weighted_accumulator() noexcept;

  constexpr W      sum_of_weights()         const noexcept;
  constexpr W      sum_of_squared_weights() const noexcept;
  constexpr size_t non_zero_count()         const noexcept;

  constexpr void operator()(const T& x, const W& w);
```

```
  template<ranges::input_range R, ranges::input_range W>
  constexpr void operator()(R&& r, W&& w);

  constexpr auto second_central_moment() const noexcept -> T;

  constexpr auto mean() const noexcept -> T;
};
```

```
constexpr weighted_accumulator() noexcept;
```

1. *Effects*: A weighted accumulator object is constructed.
2. *Complexity*: Constant.

```
constexpr W sum_of_weights() const noexcept;
```

1. *Returns*: The sum of the accumulated weights of the accumulator object.
2. *Complexity*: Constant.

```
constexpr W sum_of_squared_weights() const noexcept;
```

1. *Returns*: The sum of the squares of the accumulated weights of the accumulator object.
2. *Complexity*: Constant.

```
constexpr size_t non_zero_count() const noexcept;
```

1. *Returns*: The number of non-zero accumulated weights of the accumulator object.
2. *Complexity*: Constant.

```
constexpr void operator()(const T& x, const W& w);
```

1. *Preconditions*: `x` and `w` is a finite value.
2. *Effects*: The value of `x` weighted by `w` is accumulated.
3. *Complexity*: Constant.

```
template<ranges::input_range R, ranges::input_range W>
constexpr void operator()(R&& r, W&& w);
```

1. *Preconditions*: `r` and `w` are ranges of finite values, where the length of `r` is less than or equal to the length of `w`.
2. *Effects*: The values of `r` weighted by the corresponding values of `w` are accumulated.
3. *Complexity*: Linear in `ranges::distance(r)`.

```
constexpr auto second_central_moment() const noexcept -> T;
```

1. *Preconditions*: At least 2 invocations of **`operator()`**.
2. *Returns*: The second weighted central moment of the accumulated values if the preconditions have been met and an unspecified value otherwise.
3. *Complexity*: Constant.

```
constexpr auto mean() const noexcept -> T;
```

1. *Preconditions*: At least 1 invocation of **`operator()`**.
2. *Returns*: The weighted arithmetic mean of the accumulated weighted values if the preconditions have been met and an unspecified value otherwise.
3. *Complexity*: Constant.

# 6 Acknowledgements

Michael Wong's work is made possible by Codeplay Software Ltd., ISOCPP Foundation, Khronos and the Standards Council of Canada. The authors of this proposal wish to further thank the members of SG19 for their contributions. Additional thanks are extended to Jolanta Opara, along with Axel Naumann of CERN.

# References

[1] Geometric mean. Wikipedia, accessed 24 Mar. 2024.
https://en.wikipedia.org/wiki/Geometric_mean.

[2] Harmonic mean. Wikipedia, accessed 24 Mar. 2024.
https://en.wikipedia.org/wiki/Harmonic_mean.

[3] Petar Ĉisar and Sanja Maravić Ĉisar. Skewness and kurtosis in function of selection of network traffic distribution. *Acta Polytechnica Hungarica*, 7(2), 2020.

[4] statistics - mathematical statistics functions, python. Python, accessed 14 Apr. 2020.
https://docs.python.org/3/library/statistics.html.

[5] Documentation/How Tos/Calc: Statistical functions. Apache OpenOffice, accessed 23 May 2020.
https://wiki.openoffice.org/wiki/Documentation/How_Tos/Calc:_Statistical_functions.

[6] Statistical functions (reference). Microsoft, accessed 23 May 2020.
https://support.office.com/en-us/article/statistical-functions-reference-624dac86-a375-4435-bc25-76d659719ffd.

[7] Statistics. Julia, accessed 23 May 2020.
https://docs.julialang.org/en/v1/stdlib/Statistics/.

[8] Statistics. Maplesoft, accessed 25 Feb. 2024.
https://www.maplesoft.com/support/help/category.aspx?cid=1010.

[9] Numerical operations on data. Mathematica, accessed 25 Feb. 2024.
https://reference.wolfram.com/language/tutorial/NumericalOperationsOnData.html#7135.

[10] Computing with descriptive statistic. MathWorks, accessed 23 May 2020.
https://www.mathworks.com/help/matlab/data_analysis/descriptive-statistics.html.

[11] Statistics. NumPy, accessed 25 Feb. 2024.
https://numpy.org/doc/stable/reference/routines.statistics.html.

[12] How to calculate summary statistics. pandas, accessed 25 Feb. 2024.
https://pandas.pydata.org/docs/getting_started/intro_tutorials/06_calculate_statistics.html#.

[13] Statistics. php, accessed 23 May 2020.
https://www.php.net/manual/en/book.stats.php.

[14] stats. RDocumentation, accessed 23 May 2020.
https://www.rdocumentation.org/packages/stats/versions/3.6.2.

[15] Crate statistical. Rust, accessed 23 May 2020.
https://docs.rs/statistical/1.0.0/statistical/.

[16] The SURVEYMEANS procedure. sas, accessed 11 Jun. 2020.
https://support.sas.com/documentation/cdl/en/statug/65328/HTML/default/viewer.htm#statug_surveymeans_details06.htm.

[17] Statistical functions (scipy.stats). SciPy, accessed 25 Feb. 2024.
https://docs.scipy.org/doc/scipy/reference/stats.html.

[18] Statistical functions. IBM, accessed 28 Aug. 2020.
https://www.ibm.com/support/knowledgecenter/SSLVMB_sub/statistics_reference_project_ddita/spss/base/syn_transformation_expressions_statistical_functions.html.

[19] summarize - summary statistics. stata, accessed 25 Feb. 2024.
https://www.stata.com/manuals13/rsummarize.pdf.

[20] Aggregate functions (Transact-SQL). Microsoft, accessed 23 May 2020.
https://docs.microsoft.com/en-us/sql/t-sql/functions/aggregate-functions-transact-sql?view=sql-server-ver15.

[21] Michael Wong et al. P1415r1: SG19 machine learning layered list. ISO JTC1/SC22/WG21: Programming Language C++, accessed 9 Aug. 2020.
http://open-std.org/JTC1/SC22/WG21/docs/papers/2019/p1415r1.pdf.

[22] Paul Bristow. A proposal to add mathematical functions for statistics to the C++ standard library. JTC 1/SC22/WG14/N1069, WG21/N1668, accessed 12 Jun. 2020.
http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1069.pdf.

[23] Walter E. Brown et al. Random number generation in C++0X: A comprehensive proposal, version2. WG21/N2032 = J16/06/0102, accessed 13 Jun. 2020.
www.open-std.org/jtc1/sc22/wg21/docs/papers/2006/n2032.pdf.

[24] Pseudo-random number generation. cppreference.com, accessed 13 Jun. 2020.
https://en.cppreference.com/w/cpp/numeric/random.

[25] Nikhar Agrawal et al. Chapter 5. Statistical distributions and functions, Boost: C++ libraries, accessed 12 Jun. 2020.
https://www.boost.org/doc/libs/1_73_0/libs/math/doc/html/dist.html.

[26] GNU scientific library. GNU Operating System, accessed 13 Jun. 2020.
https://www.gnu.org/software/gsl/doc/html/index.html#.

[27] Raymond Chen. On finding the average of two unsigned integers without overflow. Microsoft, accessed 22 Feb. 2022.
https://devblogs.microsoft.com/oldnewthing/20220207-00/?p=106223.

[28] The MIT license. open source initiative, accessed 23 Mar. 2024.
https://opensource.org/license/mit.

[29] Martha L. Abell, James P. Braselton, and John A. Rafter. *Statistics with Mathematica*. Academic Press, 1999.

[30] Lorenzo Rimoldini. Weighted skewness and kurtosis unbiased by sample size. arXiv, Apr. 2013.
https://arxiv.org/abs/1304.6564.

[31] ISO 3534-1:2006: Statistics vocabulary and symbols - Part 1: General statistical terms and terms used in probability. ISO, Oct. 2006.
https://www.iso.org/standard/40145.html#:~:text=ISO%203534%2D1%3A2006%20defines,limited%20number%20of%20these%20terms.

[32] Alan Anderson. *Statistics for Dummies*. John Wiley & Sons, 2014.

[33] Ken Black, Kenneth Urban Black, Ignacio Castillo, Amy Goldlist, and Timothy Edmunds. *Essentials of Business Statistics*. John Wiley & Sons Canada, 2018.

[34] Godfrey Beddardm. *Applying Maths in the Chemical and Biomolecular Sciences: An Example-based Approach*. OUP Oxford, 2009.

[35] Naval Bajpai. *Business Statistics*. Pearson, 2009.

[36] Philippe Pébay, Timothy B. Terriberry, Hemanth Kolla, and Janine Bennett. Numerically stable, scalable formulas for parallel and online computation of higher-order multivariate central moments with arbitrary weights. *Computational Statistics*, 31(4):1305–1325, 2016.

[37] John Michael McNamee. A comparison of methods for accurate summation. *ACM SIGSAM Bulletin*, 38(1), Mar. 2004.

[38] Johan Hoffman. *Methods in Computational Science*. Society for Industrial and Applied Mathematics, 2021.

[39] Variance. Wikipedia, accessed 24 Mar. 2024.
https://en.wikipedia.org/wiki/Variance.

[40] Karl-Rudolf Koch. *Parameter Estimation and Hypothesis Testing in Linear Models*. Springer, 1999.

[41] Anurag Pande and Brian Wolshon, editors. *Traffic Engineering Handbook*. Wiley, seventh edition, 2016.

[42] Michael J. Panik. *Advanced Statistics from an Elementary Point of View*. Elsevier Science, 2005.

[43] Kandethody M. Ramachandran and Chris P. Tsokos. *Mathematical Statistics with Applications*. Elsevier Science, 2009.

[44] Charlie Amatutti. Statistical mean & business uses. bizfluent, accessed 2 Mar. 2024.
https://bizfluent.com/info-8031040-statistical-mean-business-uses.html.

[45] var. MathWorks, accessed 2 Mar. 2024.
https://www.mathworks.com/help/matlab/ref/var.html.

[46] weighted.var: Weighted univariate variance coping with missing values. RDocumentation, accessed 2 Mar. 2024.
R package: https://www.rdocumentation.org/packages/modi/versions/0.1.2/topics/weighted.var.

[47] Weightedstdev (weighted standard deviation of a sample). MicroStrategy, accessed 2 Mar. 2024.
https://www2.microstrategy.com/producthelp/current/FunctionsRef/Content/FuncRef/WeightedStDev__weighted_standard_deviat

[48] Weighted standard deviation. National Institute of Standards and Technology: U.S. Department of Commerce, accessed 2 Mar. 2024.
https://www.itl.nist.gov/div898/software/dataplot/refman2/ch2/weightsd.pdf.

[49] Pawel Cichosz. *Data Mining Algorithms: Explained Using R*. Wiley, 2014.

[50] Weighted arithmetic mean. Wikipedia, accessed 26 Dec. 2022.
https://en.wikipedia.org/wiki/Weighted_arithmetic_mean#Weighted_sample_variance.

[51] Unbiased estimation of standard deviation. Wikipedia, accessed 22 May 2021.
https://en.m.wikipedia.org/wiki/Unbiased_estimation_of_standard_deviation.

[52] John Gurland and Ram C. Tripathi. A simple approximation for unbiased estimation of the standard deviation. *The American Statistician*, 25(4):30–32, Oct. 1971.

[53] Cain Mckay. *Probability and Statistics*. EDTECH, 2019.

[54] numpy.var. NumPy, accessed 22 May 2021.
https://numpy.org/doc/stable/reference/generated/numpy.var.html.

[55] pandas.dataframe.var. pandas, accessed 25 Feb. 2024.
https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.var.html.

[56] scipy.stats.tvar. SciPy, accessed 3 Mar. 2024.
https://docs.scipy.org/doc/scipy/reference/generated/scipy.stats.tvar.html#scipy.stats.tvar.

[57] Algorithms for calculating variance. Wikipedia, accessed 19 Oct. 2019.
https://en.wikipedia.org/wiki/Algorithms_for_calculating_variance.

[58] Algorithms for calculating variance. Project Gutenberg Self Publishing Press, accessed 23 Aug. 2020.
http://www.self.gutenberg.org/articles/Algorithms_for_calculating_variance.

[59] Skewness. Wikipedia, accessed 25 Feb. 2024.
https://en.wikipedia.org/wiki/Skewness.

[60] Barry H. Cohen. *Explaining Psychological Statistics*. Wiley, 2013.

[61] Rex B. Kline. *Principles and Practice of Structural Equation Modeling*. Guilford Publications, 2023.

[62] Paul J. Mitchell. Experimental design and statistical analysis for pharmacology and the biomedical sciences, 2022.

[63] scipy.stats.skew. SciPy.org, accessed 24 May 2021.
https://docs.scipy.org/doc/scipy/reference/generated/scipy.stats.skew.html.

[64] Tom Keldenich. What is skewness and kurtosis? - Everything you need to know now. Inside Machine Learning, accessed Feb. 19, 2024.
https://inside-machinelearning.com/en/skewness-and-kurtosis/.

[65] Computing skewness and kurtosis in one pass. John D. Cook Consulting, accessed 20 Aug. 2020.
https://www.johndcook.com/blog/skewness_kurtosis/.

[66] Richard Dosselmann. Basic Statistics, accessed 3 Mar. 2024.
https://open-std.org/JTC1/SC22/WG21/docs/papers/2023/p1708r8.pdf.

[67] Kurtosis. Wikipedia, accessed 29 Dec. 2022.
https://en.wikipedia.org/wiki/Kurtosis.

[68] Kurtosis. Wikipedia, accessed 29 May 2021.
https://en.wikipedia.org/wiki/Kurtosis.

[69] James Wu and Stephen Coggeshall. *Foundations of Predictive Analytics*. CRC Press, 2012.

[70] Kurtosis formula. macroption, accessed 24 May 2021.
https://www.macroption.com/kurtosis-formula/.

[71] Ken A. Aho. *Foundational and Applied Statistics for Biologists Using R*. CRC Press, 2016.

[72] Eric Niebler. Chapter 1. Boost.Accumulators. Boost: C++ Libraries, accessed 14 Sept. 2019.
https://www.boost.org/doc/libs/1_71_0/doc/html/accumulators.html.

[73] Jolanta Opara. P2119R0 feedback on P1708: Simple statistical functions. JTC1/SC22/WG21, accessed 14 Apr. 2020.
http://open-std.org/JTC1/SC22/WG21/docs/papers/2020/p2119r0.html.

[74] James A. Rosenthal. *Statistics and Data Interpretation for Social Work*. Springer, 2012.

[75] Central moment. Wikipedia, accessed 6 Jul. 2024.
https://en.wikipedia.org/wiki/Central_moment.

[76] Walter E. Brown, Axel Naumann, and Edward Smith-Rowland. Mathematical special functions for C++17, v4. JTC1.22.32 Programming Language C++, WG21 P0226R0, accessed 12 Jun. 2020.
www.open-std.org/jtc1/sc22/wg21/docs/papers/2016/p0226r0.pdf.

# Appendix A Examples

**Example 1** The following example showcases the use of **mean**, **variance** and **standard deviation** functions.

```
struct PRODUCT {
  float price;
  int   quantity;
};

std::array<PRODUCT, 5> A = { {{5.0f, 1}, {1.7f, 2}, {9.2f, 5}, {4.4f, 7}, {1.7f, 3}} };
auto A_ = A
  | std::views::transform([](const auto& product) { return product.price; })
  | std::ranges::to<std::vector<float>>();
std::array<float, 5> W = { { 2.0f, 2.0f, 1.0f, 3.0f, 5.0f } };

std::cout << "mean = " << std::mean(std::execution::par, A_, W);
std::cout << "\nvariance = " << std::variance(A_);
std::cout << "\nstandard deviation = " << std::standard_deviation(A_);
```

**Example 2** The following example showcases the use of a **mean** and **variance** function.

```
std::list<int> L = { 8, 6, 12, 3, 5 };

auto [mean, variance] = std::mean_variance<float>(L);

std::cout << "mean = "       << mean;
std::cout << "\nvariance = " << variance;
```

**Example 3** The following example showcases the use of a **kurtosis** function.

```
std::vector<double> v = { 2.0, 3.0, 5.0, 7.0, 11.0, 13.0, 17.0, 19.0 };

std::cout << "kurtosis = " << std::kurtosis(v);
```

**Example 4** The following example showcases the use of an accumulator object.

```
std::list<double> L = { 1., 2., 2., 2., 3., 3., 3. };

std::unweighted_accumulator<double> acc;

for (const auto& x : L)
  acc(x);

std::cout << "mean = "                 << acc.mean();
std::cout << "\nvariance = "           << acc.variance(0);
std::cout << "\nstandard deviation = " << acc.standard_deviation();
std::cout << "\nskewness = "           << acc.skewness(false);
std::cout << "\nkurtosis = "           << acc.kurtosis();
```

**Example 5** The following example showcases the use of a **customized** unweighted accumulator object.

```
// custom unweighted accumulator

class custom_unweighted_accumulator : public std::unweighted_accumulator<double>
{
public:
  constexpr custom_unweighted_accumulator() noexcept { first_ = true; sum_sq_=0; }

  constexpr void operator()(const double& x)
  {
    unweighted_accumulator::operator()(x);

    if (first_)
    {
      min_   = max_ = x;
      first_ = false;
    }
    else
    {
      min_ = std::min(min_, x);
      max_ = std::max(max_, x);
    }

    sum_sq_ += x*x;
  }

  constexpr double min()    const noexcept { return min_; }
  constexpr double max()    const noexcept { return max_; }
  constexpr double sum_sq() const noexcept { return sum_sq_; }

private:
  bool   first_;
  double min_, max_;
  double sum_sq_;
};

// ...
```

```
std::vector<double> L = { 17.2, -14.27, 19.22, 13.56, -0.01, 2.6 };

custom_unweighted_accumulator acc;

for (const auto& x: L)
  acc(x);

std::cout << "mean = "             << acc.mean();
std::cout << "\nvariance = "       << acc.variance();
std::cout << "\nmax = "            << acc.max();
std::cout << "\nmin = "            << acc.min();
std::cout << "\nsum of squares = " << acc.sum_sq();
```