Basic Statistics

Document Number:	P1708R10
Author:	Richard Dosselmann: **dosselmr@uregina.ca**
Contributors:	Michael Chiu: chiu@cs.toronto.edu,
	Guy Davidson: guy.davidson@hatcat.com
	Pete Isensee: pjisensee@gmail.com
	Oleksandr Koval: oleksandr.koval.dev@gmail.com
	Larry Lewis: Larry.Lewis@sas.com
	Johan Lundburg: lundberj@gmail.com
	Jens Maurer: Jens.Maurer@gmx.net
	Eric Niebler: eniebler@fb.com
	Phillip Ratzloff: phil.ratzloff@sas.com
	Vincent Reverdy: vreverdy@illinois.edu
	John True
	Michael Wong: michael@codeplay.com
Date:	September 2025 (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 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 objects 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.
- Overloaded functions 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.

- Overloaded functions to allow the **data type** of a statistic to be changed are removed, since this can (instead) be done by way of **views**.
- Mean and variance (and standard deviation) convenience functions are changed so as to return mean and variance (and standard deviation) as a **`struct`**, rather than a pair of values, so as to make such return values more meaningful, as suggested in [1].
- Kurtosis functions are changed so as to take parameters `sample` and `excess` as a **`struct`**, rather than individual values, so as to make such parameters more meaningful, as suggested in [1].
- **Accumulator** object is deferred to a future proposal so as to allow for further study.
- **Covariance** is introduced from P2681 [2].

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 [3, 4, 5]. These functions do exist in Python [6], the foremost competitor to C++ in the area of **machine learning**, along with Apache Commons Math [7], Calc [8], Excel [9], Julia [10], Maple [11], Mathematica [12], MATLAB [13], NumPy [14], Pandas [15], PHP [16], R [17], Rust [18], SAS [19], SciPy [20], SPSS [21], Stata [22] and SQL [23]. Further need for such functions has been identified as part of **SG19** (machine learning) [24].

This is not the first proposal to move statistics in C++. In 2004, a number of statistical distributions were proposed in [25]. Additional distributions followed in 2006 [26]. Statistical distributions ultimately appeared in the C++11 standard [27]. Statistical distributions and functions are also found in Boost [28]. A C library, GNU Scientific Library [29], further includes support for statistics, special functions and histograms.

Six statistics are defined in this proposal. Two additional statistics, specifically **median** (along with **quantile**) and **mode**, are **not** included in this proposal. These more involved statistics are deferred to a **future** proposal. Proposed functions are similar to (existing) numeric operations, which includes `std::accumulate` [30] for instance, and those of numeric arrays, specifically `std::valarray` [31], member functions such as `std::valarray::sum` [32]. Like existing entities of the (C++) standard library, this proposal only specifies an interface, meaning that a variety of implementations are possible. This enables a vendor to favor accuracy [33] over performance for instance. An **implementation**, released under the **MIT** license [34], is available at `https://github.com/dosselmann/statistics`

2.1 Mean

The *arithmetic mean* [35, 36], denoted μ, of the $n \geq 1$ values $x_1, x_2, \ldots, x_n$ of a population [35], and $\bar{x}$ in the case of a sample [35], 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** $3534-1{:}2006$ [37] standard and Python [6]. The *weighted arithmetic mean* [36, 38, 39, 40], 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 [6]. The *geometric mean* [35], having a number of applications in science and technology [3] and found in Python [6], is defined as

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

and the *weighted geometric mean* [38] 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* [35] of positive values $x_i > 0$, also having many applications in science and technology [4] and found in Python [6], is defined as

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

and the *weighted harmonic mean* [41] 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 [6]. Each of the arithmetic, geometric and harmonic means can be (accurately) computed in **linear** time [42]. When computing the associated sums of these means, and indeed any sum in this proposal, robust methods [43, 44] ought to be considered.

2.2 Variance

The population *variance* [45, 46, 47] of $n \geq 1$ values is defined as

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

and an unbiased estimator [46, 48] of the sample *variance* [35, 47, 49] 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 [6]. As there appears to be no common definition of (sample) weighted variance (and standard deviation), as further discussed in Section 4.11, weighted variance is not considered in this proposal. 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 [42, 57, 58].

2.3 Standard Deviation

The *standard deviation* [35, 47] of $n \geq 2$ values, denoted s, is defined as the square root of the variance. The population and sample standard deviation are found in the **ISO** $3534-1{:}2006$ standard and Python [6].

2.4 Skewness

The population *skewness* [35, 59, 60, 61], a measure of the **asymmetry** [35] of $n \geq 1$ values, is defined as

$$g_s = \frac{1}{n\sigma^3} \sum_{i=1}^{n} (x_i - \mu)^3 \tag{9}$$

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

$$G_s = \frac{\sqrt{n\,(n-1)}}{n-2} g_s. \tag{10}$$

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 [42, 65]. Weighted skewness [36, 66] is highly specialized and is, therefore, not considered in this proposal.

2.5 Kurtosis

The population *kurtosis* [35, 61, 67], a measure of the "**tailedness**" [67] of $n \geq 1$ values, is defined as

$$g_k = \frac{1}{n\sigma^4} \sum_{i=1}^{n} (x_i - \mu)^4 \tag{11}$$

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

$$\hat{g}_k = \frac{1}{n\sigma^4}\sum_{i=1}^{n}(x_i - \mu)^4 - 3. \tag{12}$$

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

$$G_k = \frac{n(n+1)}{(n-1)(n-2)(n-3)s^4}\sum_{i=1}^{n}(x_i - \bar{x})^4 \tag{13}$$

and an unbiased estimator of the sample excess kurtosis [69] is

$$\hat{G}_k = \frac{n(n+1)}{(n-1)(n-2)(n-3)s^4}\sum_{i=1}^{n}(x_i - \bar{x})^4 - \frac{3(n-1)^2}{(n-2)(n-3)}. \tag{14}$$

The population and sample kurtosis are found in the **ISO** $3534-1{:}2006$ standard. Weighted kurtosis [36, 66] is highly specialized and is, therefore, not considered in this proposal.

2.6 Covariance

The **population** [35] *covariance* [71], a measure of the **joint variability** of the (two sets of) values $x_1, x_2, \ldots, x_n$ and $y_1, y_2, \ldots, y_n$ ($n \geq 1$), is defined as

$$\sigma_{xy} = \frac{1}{n}\sum_{i=1}^{n}(x_i - \mu_x)(y_i - \mu_y), \tag{15}$$

where μ_x and μ_y are the (arithmetic) **population** means of the values $x_1, x_2, \ldots, x_n$ and $y_1, y_2, \ldots, y_n$, respectively. The **sample** [35] covariance [71] ($n \geq 2$) is defined as

$$s_{xy} = \frac{1}{n-1}\sum_{i=1}^{n}(x_i - \bar{x})(y_i - \bar{y}), \tag{16}$$

where $\bar{x}$ and $\bar{y}$ are the **sample** means of the values. The sample covariance is found in the **ISO** $3534-1{:}2006$ [37] standard and Python [6].

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 Special Values

Special values may sometimes turn up in a range. This includes values such as $\pm\infty$ and **NaN**. Special values may also arise during the computation of a statistic, for instance the mean of an empty range (see Section 4.2), which results in a division by zero. The matter of special values is raised in [1]. Checks for special values require that a branching statement be evaluated for each value of a range, a costly operation. Following **precedent**, `std::accumulate` and `std::val_array::sum` as examples, the proposed functions do not check for special values. When special values do occur in a range, functions return an **unspecified** value. Depending on the particular (vendor) implementation, the proposed functions may call `<cmath>` functions that might, at times, raise floating-point exception flags, such as `FE_DIVBYZERO`. Following [72], in which many of the `<cmath>` functions are marked `constexpr`, though they may raise floating-point exception flags, the proposed functions are marked as `constexpr`.

From a user's perspective, special values 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::isfinite(x); }) | std::mean;
```

4.2 Insufficient Values

It may be the case that there are too few values in a range over which a particular statistic is computed, a matter that is also brought up in [1]. The prospect of returning an `std::expected` object has been suggested. Presently, `std::expected` objects are not used (elsewhere) in this proposal, such as in the case of a special value. It would, therefore, be asymmetric to have some situations return an `std::expected` object, but not others. There is **precedent** in the case of too few values, in particular `std::val_array::sum`, namely "[i]f the `std::valarray` is empty, the behavior is undefined" [32]. In place of undefined behavior, this proposal requires that a value be returned in the event that there are too few values in a range, albeit an **unspecified** value. The following is an example of such a situation, namely

```
std::list<int> L;
auto average = std::mean(L);
```

4.3 Accuracy

(Numerical) accuracy is discussed in [1]. C++ makes no guarantees about the accuracy of a floating-point calculation, a fact that applies to the functions of this proposal. As noted in [1], accuracy may be related to the order in which the values of a range are processed. Following **precedent**, namely `std::val_array::sum`, in which, "[t]he order in which the elements are processed by this function is unspecified" [32], the functions of this proposal make no guarantee about the order in which the values of a range are processed. Vendors may wish to provide documentation regarding the recommended usage of functions, suggesting, for instance, that a range be sorted in order to maximize the accuracy of the computation of a mean.

The author and (several of the) contributors of this proposal recognize the problems associated with accuracy (as well as special values), problems that also arise in many contexts outside of this proposal. Given the complicated nature of such problems, and so as to not burden vendors, this proposal does not attempt to resolve such problems. The author and contributors welcome future proposals to tackle such problems.

4.4 Overflow and Underflow

Depending on the nature of a given implementation, the computation of a statistic may result in an (unexpected) overflow or underflow. This might occur for example during the computation of the mean of a range of **`double`** values, each equal to `DBL_MAX`, again depending on manner in which the mean is computed. In the interest of reducing the burden on vendors, this proposal does not insist that every range of **finite** values yield a well-defined value. Instead, this proposal requires that the (numerical) range of values for which overflow and underflow might occur be **implemention** defined, with an **unspecified** value returned when overflow and underflow occurs.

4.5 Casting

A frequent question is one of the ability to cast, generally **promote**, the values of a range. The mean of a range of integers, for instance, is often a (strictly) floating-point value. Earlier versions of this proposal sought to (implicitly) convert a statistic, and, thus, return type of a function, to a floating-point value. Given that this behavior might be **unexpected** on the part of a user, the proposed functions do not perform any such conversion. Another option, template overloads of functions were proposed in 1709R9, one for which there is no precedent. A user wishing to promote the type of a range may do so (explicitly) using a **view**, perhaps something of the form

```
auto data_ = data
  | std::views::transform([](const auto& value) { return static_cast<float>(value); })
  | std::ranges::to<std::vector<float>>();
```

in which a range `data` of values of type **`int`** is cast, by way of a view, to a range of values of type **`float`**. A (complete) example that demonstrates the use of views is presented in Appendix A.

It is suggested in [1] that the proposed functions take an additional **initialization** parameter, in the same way as the function `std::accumulate` [30] for example. The suggested function is given as

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

This sort of design could allow a user to compute a statistic in **stages**. While this approach is natural in the case of a sum, in which a parameter is added to a given sum, things are more involved in the case of the statistics of this proposal. Take as an example again, the mean of a range of values, this time the mean of (any) ten values, denoted (as a set) $S = \{x_1, x_2, \ldots, x_{10}\}$. Further suppose that one wishes to compute the mean of S in two stages, first using the values $S_1 = \{x_1, x_2, x_3, x_4\}$ and then $S_2 = \{x_5, x_6, x_7, x_8, x_9, x_{10}\}$. The (sample) mean of S_1, computed first, is

$$\bar{x}_1 = \frac{x_1 + x_2 + x_3 + x_4}{4}. \tag{17}$$

In the second stage, the (overall) mean is computed as

$$\bar{x} = \frac{4\bar{x}_1 + x_5 + x_6 + x_7 + x_8 + x_9 + x_{10}}{4 + 6} \tag{18}$$

$$= \frac{4\left(\frac{x_1 + x_2 + x_3 + x_4}{4}\right) + x_5 + x_6 + x_7 + x_8 + x_9 + x_{10}}{4 + 6} \tag{19}$$

$$= \frac{x_1 + x_2 + x_3 + x_4 + x_5 + x_6 + x_7 + x_8 + x_9 + x_{10}}{10}. \tag{20}$$

As seen in the computations above, the first mean, namely $\bar{x}_1$, must be effectively "reversed", that is multiplied by $|S_1| = 4$, before it can be combined with the values of S_2. To allow for this sort of computation of a mean in stages, a function would need to take as an additional parameter $|S_1|$, a value that might not be available. Things are more involved yet in the case of the standard deviation for example, in which a square root would additionally need to be "reversed". Given the involved nature of such computations, this proposal does not recommend nor propose that functions include an initialization parameter.

4.6 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. A contributor, Phillip Ratzloff feels (a sentiment that was echoed by the author of [73]) that one might handle this matter in much the same way as the special values of Section 4.1, specifically by using a statement of the form

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

4.7 Projections

The functions 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. This is much like the approach suggested in Sections 4.1 and 4.6.

4.8 Zipped Ranges

Several of the functions of this proposal take both a range of values and a (corresponding) range of weights. It is natural to suggest that the two ranges be replaced by a (single) zipped range. Not all functions, however, support weighted variants. Ranges of values and weights are, thus, separated so as to clearly indicate which functions (and, therefore, statistics) support weighted variants and which do not.

4.9 Concepts

Much like `std::complex`, the proposed (template) functions 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. It is felt that the added flexibility afforded by not using **concepts** to strictly limit functions 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.

4.10 Header and Namespace

Early versions of this proposal, specifically P1708R0, P1708R1 and P17082, request that the proposed functions be placed into the `<numeric>` header. Since P1708R3, it has instead been suggested that the functions 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**.

4.11 Weighted Variance

The population *weighted variance* [39] of $n \geq 1$ values is defined as

$$\sigma_w^2 = \frac{1}{\sum_{i=1}^{n} w_i} \sum_{i=1}^{n} w_i \left(x_i - \bar{x}_w\right)^2 \tag{21}$$

and an unbiased estimator of the sample weighted variance [75] is

$$s_{w,1}^2 = \frac{1}{\sum_{i=1}^{n} w_i - 1} \sum_{i=1}^{n} w_i \left(x_i - \bar{x}_w\right)^2 . \tag{22}$$

There are at least two other definitions of sample weighted variance in addition to that of $s_{w,1}^2$. Note that $s_{w,1}^2$ is the version implemented in MATLAB [76], as well as an R package [77]. The first [78, 79] of these two is

$$s_{w,2}^2 = \frac{1}{\frac{\hat{n}-1}{\hat{n}} \sum_{i=1}^{n} w_i} \sum_{i=1}^{n} w_i \left(x_i - \bar{x}_w\right)^2 , \tag{23}$$

where $\hat{n} \geq 1$ is the number of non-zero weights w_i. A second definition of sample weighted variance [36, 80], also used in the case of *reliability weights* [50], is

$$s_{w,3}^2 = \frac{\sum_{i=1}^{n} w_i}{(\sum_{i=1}^{n} w_i)^2 - \sum_{i=1}^{n} w_i^2} \sum_{i=1}^{n} w_i \left(x_i - \bar{x}_w\right)^2 . \tag{24}$$

As there appears to be no common definition of (sample) weighted variance, this statistic is not considered in this proposal.

A future proposal may consider this statistic, perhaps indirectly as a function which computes the weighted second central moment [81], along with each of $\hat{n}$, $\sum_{i=1}^{n} w_i$ and $\sum_{i=1}^{n} w_i^2$. Thus, σ_w^2 could be computed as

```
// accumulate values of range R weighted by weights of range W
auto [second_central_moment, non_zero_count, w, w_sq] = std::weighted_variance(R, W);

auto sigma2_w = 1/w * second_central_moment;
```

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

```
auto s2_w1 = 1/(w-1) * second_central_moment;
```

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

```
auto s2_w2 = 1 / ((non_zero_count-1)/n_zero_count * w) * second_central_moment;
```

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

```
auto s2_w3 = w / (w*w - w_sq) * second_central_moment;
```

5 Technical Specifications

The templates of the 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<ranges::input_range R>
constexpr auto mean(R&& r) -> std::ranges::range_value_t<R>;
```

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

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

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

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

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

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

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

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

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, 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<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, 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>
struct mean_variance_result { T mean, variance; };

template<class T>
struct mean_standard_deviation_result { T mean, standard_deviation; };

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)) ->
    mean_variance_result<std::ranges::range_value_t<R>>;

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)) ->
    mean_variance_result<std::ranges::range_value_t<R>>;

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)) ->
    mean_standard_deviation_result<std::ranges::range_value_t<R>>;

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)) ->
    mean_standard_deviation_result<std::ranges::range_value_t<R>>;

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

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>;

struct kurtosis_parameters { bool sample = true; bool excess = true; };

template<ranges::input_range R>
constexpr auto kurtosis(R&& r, kurtosis_parameters params = {}) ->
  std::ranges::range_value_t<R>;

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, kurtosis_parameters params = {}) ->
    std::ranges::range_value_t<R>;

template<ranges::input_range R1, ranges::input_range R2>
constexpr auto covariance(
  R1&& r1, R2&& r2,
  std::common_type_t<std::iter_value_t<R1>, std::iter_value_t<R2>> ddof = 1) ->
    std::common_type_t<std::iter_value_t<R1>, std::iter_value_t<R2>>;

template<class ExecutionPolicy, ranges::input_range R1, ranges::input_range R2>
requires std::is_execution_policy_v<std::remove_cvref_t<ExecutionPolicy>>
constexpr auto covariance(
  ExecutionPolicy&& policy,
  R1&& r1, R2&& r2,
  std::common_type_t<std::iter_value_t<R1>, std::iter_value_t<R2>> ddof = 1) ->
    std::common_type_t<std::iter_value_t<R1>, std::iter_value_t<R2>>;
}
```

5.2 Functions

The functions specified in this section return an unspecified value:

- If any of the values of the ranges `r`, `r1`, `r2` or `w` is a NaN, ∞ or $-\infty$.
- If a NaN, ∞ or $-\infty$ occurs during the evaluation of a function.
- If overflow or underflow occurs during the evaluation of a function, which might even occur in the case of finite ranges of values, where the (numerical) range of values for which overflow or underflow might occur is implementation defined.

5.2.1 Mean Functions

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

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

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

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

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

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

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, ranges::input_range R, ranges::input_range Weights>
requires std::is_execution_policy_v<std::remove_cvref_t<ExecutionPolicy>>
auto harmonic_mean(ExecutionPolicy&& policy, R&& r, Weights&& 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<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, 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<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, 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 Mean and Variance and Standard Deviation Convenience Functions

```
template<class T>
struct mean_variance_result { T mean, variance; };

template<class T>
struct mean_standard_deviation_result { T mean, standard_deviation; };

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)) ->
    mean_variance_result<std::ranges::range_value_t<R>>;

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)) ->
    mean_variance_result<std::ranges::range_value_t<R>>;

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)) ->
    mean_standard_deviation_result<std::ranges::range_value_t<R>>;

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

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, where the specific unbiased sample estimator of the skewness is implementation defined.
3. *Complexity*: Linear in `ranges::distance(r)`.

5.2.8 Kurtosis Functions

```
struct kurtosis_parameters { bool sample = true; bool excess = true; };

template<ranges::input_range R>
constexpr auto kurtosis(R&& r, kurtosis_parameters params = {}) ->
  std::ranges::range_value_t<R>;

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, kurtosis_parameters params = {}) ->
    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, where the specific unbiased sample estimator of the kurtosis is implementation defined.
3. *Complexity*: Linear in `ranges::distance(r)`.

5.2.9 Covariance Functions

```
template<ranges::input_range R1, ranges::input_range R2>
constexpr auto covariance(
  R1&& r1, R2&& r2,
  std::common_type_t<std::iter_value_t<R1>, std::iter_value_t<R2>> ddof = 1) ->
    std::common_type_t<std::iter_value_t<R1>, std::iter_value_t<R2>>;

template<class ExecutionPolicy, ranges::input_range R1, ranges::input_range R2>
requires std::is_execution_policy_v<std::remove_cvref_t<ExecutionPolicy>>
constexpr auto covariance(
  ExecutionPolicy&& policy,
```

```
R1&& r1, R2&& r2,
std::common_type_t<std::iter_value_t<R1>, std::iter_value_t<R2>> ddof = 1) ->
  std::common_type_t<std::iter_value_t<R1>, std::iter_value_t<R2>>;
```

1. *Preconditions*: `r1` and `r2` are ranges of finite values, where `r1` has at least 1 value, the length of `r1` is less than or equal to the length of `r2`, and `ddof` is not equal to `ranges::distance(r)`.
2. *Returns*: The covariance of the values of `r1` and `r2` if the preconditions have been met and an unspecified value otherwise.
3. *Complexity*: Linear in `ranges::distance(r)`.

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] Oliver J. Rosten and Mark Hoemmen. Remarks on basis statistics, P1708r9. Programming Language C++, Library Working Group, accessed 25 Jan. 2025. https://www.open-std.org/JTC1/SC22/WG21/docs/papers/2024/p3495r0.pdf.

[2] Richard Dosselmann. More basic statistics. ISO JTC1/SC22/WG21: Programming Language C++, accessed 17 May 2025. https://open-std.org/JTC1/SC22/WG21/docs/papers/2023/p2681r1.pdf.

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

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

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

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

[7] Statistics. Apache, accessed 4 May 2025. https://commons.apache.org/proper/commons-math/userguide/stat.html.

[8] Documentation/how tos/calc: Statistical functions. Apache OpenOffice, accessed 23 May 2020. https://wiki.openoffice.org/wiki/Documentation/How_Tos/Calc:_Statistical_functions.

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

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

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

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

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

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

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

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

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

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

[19] 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.

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

[21] 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.

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

[23] 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.

[24] 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.

[25] 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.

[26] 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.

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

[28] Nikhar Agrawal et al. Math toolkit 4.2.1. Boost: C++ Libraries, accessed 12 Jun. 2020.
https://www.boost.org/doc/libs/latest/libs/math/doc/html/index.html.

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

[30] std::accumulate. cppreference.com, accessed 15 Mar. 2025.
https://en.cppreference.com/w/cpp/algorithm/accumulate.

[31] Standard library header <valarray>. cppreference.com, accessed 15 Mar. 2025.
https://en.cppreference.com/w/cpp/header/valarray.

[32] std::valarray<T>::sum. cppreference.com, accessed 15 Mar. 2025.
https://en.cppreference.com/w/cpp/numeric/valarray/sum.

[33] 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.

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

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

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

[37] ISO 3534-1:2006(en): Statistics - Vocabulary and symbols - Part 1: General statistical terms and terms used in probability. ISO, Oct. 2006.
https://www.iso.org/obp/ui/en/#iso:std:iso:3534:-1:ed-2:v2:en.

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

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

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

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

[42] 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.

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

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

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

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

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

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

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

[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. ISO JTC1/SC22/WG21: Programming Language C++, 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] James Wu and Stephen Coggeshall. *Foundations of Predictive Analytics*. CRC Press, 2012.

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

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

[71] Peter Goos and David Meintrup. *Statistics with JMP: Graphs, Descriptive Statistics and Probability*. Wiley, 2015.

[72] Oliver J. Rosten. More constexpr for <cmath> and <complex>. Programming Language C++, Library Working Group, accessed 15 Mar. 2025.
https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2023/p1383r2.pdf.

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

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

[77] 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.

[78] Weightedstdev (weighted standard deviation of a sample). MicroStrategy, accessed 2 Mar. 2024.
https://www2.microstrategy.com/producthelp/current/FunctionsRef/Content/FuncRef/WeightedStDev_weighted_standard_deviation_of_a_sa.htm#:~:textĀ%20weighted%20standard%20deviation%20allows,other%20values%20in%20a%20sample.

[79] 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.

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

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

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_);
```

```
std::cout << "\nweighted mean = "           << std::mean(std::execution::par, A_, W);
std::cout << "\ngeometric mean = "          << std::geometric_mean(A_);
std::cout << "\nweighted geometric mean = " << std::geometric_mean(A_, W);
std::cout << "\nharmonic mean = "           << std::harmonic_mean(A_);
std::cout << "\nweighted harmonic mean = "  << std::harmonic_mean(A_, W);
std::cout << "\nvariance = "                << std::variance(A_);
std::cout << "\nstandard deviation = "      << std::standard_deviation(A_);
std::cout << "\nskewness = "                << std::skewness(A_);
std::cout << "\nkurtosis = "                << std::kurtosis(A_);
```

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

```
std::list<float> L = { 8.0f, 6.0f, 12.0f, 3.0f, 5.0f };

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

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

Example 3 The following example showcases the use of the **skewness** and **kurtosis** functions.

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

std::cout << "skewness = "   << std::skewness(v, false);
std::cout << "\nkurtosis = " << std::kurtosis(v, { .sample=false, .excess=true });
```

Example 4 The following example showcases the use of a **covariance** function.

```
std::vector<double> v1 = {  2.0,  3.0,  5.0,  7.0,  11.0,  13.0,  17.0,  19.0 };
std::vector<double> v2 = { -2.0, -3.0, -5.0, -7.0, -11.0, -13.0, -17.0, -19.0 };

std::cout << "covariance = " << std::covariance(v1, v2);
```