Simple Statistics

Document Number:	P1708R6
Author:	Richard Dosselmann: **dosselmr@cs.uregina.ca**
Contributors:	Michael Chiu: chiu@cs.toronto.edu, Guy Davidson: guy.davidson@hatcat.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 Michael Wong: michael@codeplay.com
Date:	March 15, 2022 (mailing)
Project:	ISO JTC1/SC22/WG21: Programming Language C++
Audience:	SG6, SG19, WG21, LEWG

Contents

0 Revision History

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 freestanding 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 **freestanding** 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, classes and parameters are renamed so as to be more meaningful.
- Various technical errors relating to ranges and execution policy are corrected.

- `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**.

1 Introduction

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

2 Motivation and Scope

Simple statistical functions, **not** presently found in the standard (including the special math library), frequently arise in **scientific** and **industrial**, as well as **general**, applications. These functions do exist in Python [1], the foremost competitor to C++ in the area of **machine learning**, along with Calc [2], Excel [3], Julia [4], MATLAB [5], PHP [6], R [7], Rust [8], SAS [9], SPSS [10] and SQL [11]. Further need for such functions has been identified as part of **SG19** (machine learning) [12].

This is not the first proposal to move statistics in C++. In 2004, a number of statistical distributions were proposed in [13]. Additional distributions followed in 2006 [14]. Statistical distributions ultimately appeared in the C++11 standard [15]. Distributions, along with statistical tests, are also found in Boost [16]. A series of special mathematical functions later followed as part of the C++17 standard [17]. A C library, GNU Scientific Library [18], further includes support for statistics, special functions and histograms.

Five statistics are defined in this proposal. Two (important) statistics, specifically **quantile** (and **median**) and **mode**, are **not** included in this proposal. These more involved statistics are deferred to a **future** proposal.

2.1 Mean

The *arithmetic mean* [19] of the values $x_1, x_2, \ldots, x_n$ ($n \geq 1$), denoted μ or $\bar{x}$ in the case of a **population** [19] or **sample** [19], respectively, is defined as

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

The *weighted arithmetic mean* [20, 21], for weights $w_1, w_2, \ldots, w_n \geq 0$, denoted μ^* or $\bar{x}^*$ in the case of a population or sample, respectively, is defined as

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

The *geometric mean* [19] of the non-negative values $x_i \geq 0$ is defined as

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

and the *weighted geometric mean* [20] 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* [19] of the positive values $x_i > 0$ is defined as

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

and the *weighted harmonic mean* [22] is defined as

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

Each of the arithmetic, geometric and harmonic means can be computed in **linear** time using Equations (1) and (2), (3) and (4), and (5) and (6), respectively. When computing the associated sums of these means, and indeed any sum in this proposal, robust methods [23] ought to be considered.

2.2 Skewness

The **population** *skewness* [19], a measure of the **symmetry** [19] of the values ($n \geq 3$), is defined as

$$\frac{1}{n\sigma^3}\sum_{i=1}^{n}(x_i-\mu)^3 \tag{7}$$

and the **sample** *skewness* [19] is defined as

$$\frac{n}{(n-1)(n-2)s^3}\sum_{i=1}^{n}(x_i-\bar{x})^3, \tag{8}$$

where σ and s are defined in Section 2.5. The *weighted* **population** *skewness* [21] is defined as

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

and the *weighted* **sample** *skewness* [21] is defined as

$$\frac{\left(\sum_{i=1}^{n} w_i\right)^2}{s^3\left(\left(\sum_{i=1}^{n} w_i\right)^3 - 3\sum_{i=1}^{n} w_i\sum_{i=1}^{n} w_i^2 + 2\sum_{i=1}^{n} w_i^3\right)}\sum_{i=1}^{n} w_i(x_i-\bar{x})^3. \tag{10}$$

Skewness (and kurtosis) can be computed in **linear** time [24]. When scaled by the factor

$$\frac{\sqrt{n(n-1)}}{n-2}, \tag{11}$$

skewness becomes the *adjusted Fisher-Pearson standardized moment coefficient* [25].

2.3 Kurtosis

The *Fisher* [26] or *excess* [27] **population** *kurtosis* [27], a measure of the "**tailedness**" [28] of the values ($n \geq 4$), is defined as

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

and the (Fisher) **sample** *kurtosis* [21] is defined as

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

The *weighted* (Fisher) **population** *kurtosis* [21] is defined as

$$\frac{1}{\sigma^4\sum_{i=1}^{n} w_i}\sum_{i=1}^{n} w_i(x_i-\mu)^4 - 3 \tag{14}$$

and the *weighted* (Fisher) **sample** *kurtosis* [21] is defined as

$$\frac{\left(\sum_{i=1}^{n} w_i\right)^2 \left(\left(\sum_{i=1}^{n} w_i\right)^4 - 4\sum_{i=1}^{n} w_i \sum_{i=1}^{n} w_i^3 - 3\left(\sum_{i=1}^{n} w_i^2\right)^2\right)}{W\sigma^4} \sum_{i=1}^{n} w_i (x_i - \mu)^4$$
$$- \frac{3\sum_{i=1}^{n} w_i^2 \left(\left(\sum_{i=1}^{n} w_i\right)^4 - 2\left(\sum_{i=1}^{n} w_i\right)^2 \sum_{i=1}^{n} w_i^2 + 4\sum_{i=1}^{n} w_i \sum_{i=1}^{n} w_i^3 - 3\left(\sum_{i=1}^{n} w_i^2\right)^2\right)}{W},$$

where

$$W = \left(\left(\sum_{i=1}^{n} w_i\right)^2 - \sum_{i=1}^{n} w_i^2\right)\left(\left(\sum_{i=1}^{n} w_i\right)^4 - 6\left(\sum_{i=1}^{n} w_i\right)^2 \sum_{i=1}^{n} w_i^2 + 8\sum_{i=1}^{n} w_i \sum_{i=1}^{n} w_i^3 + 3\left(\sum_{i=1}^{n} w_i^2\right)^2 - 6\sum_{i=1}^{n} w_i^4\right). \tag{15}$$

The *Pearson* [26] **population** *kurtosis* [27] is defined as

$$\frac{1}{n\sigma^4} \sum_{i=1}^{n} (x_i - \bar{x})^4 \tag{16}$$

and the (Pearson) **sample** *kurtosis* is defined as

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

The *weighted* (Pearson) **population** *kurtosis* is defined as

$$\frac{1}{\sigma^4 \sum_{i=1}^{n} w_i} \sum_{i=1}^{n} w_i (x_i - \mu)^4 \tag{18}$$

and the *weighted* (Pearson) **sample** *kurtosis* is defined as

$$\frac{\left(\sum_{i=1}^{n} w_i\right)^2 \left(\left(\sum_{i=1}^{n} w_i\right)^4 - 4\sum_{i=1}^{n} w_i \sum_{i=1}^{n} w_i^3 - 3\left(\sum_{i=1}^{n} w_i^2\right)^2\right)}{W\sigma^4} \sum_{i=1}^{n} w_i (x_i - \mu)^4. \tag{19}$$

2.4 Variance

The **population** *variance* [19] ($n \geq 1$), denoted σ^2, is defined as

$$\frac{1}{n} \sum_{i=1}^{n} (x_i - \mu)^2 \tag{20}$$

and the **sample** *variance* [19] ($n \geq 2$), denoted s^2, is defined as

$$\frac{1}{n-1} \sum_{i=1}^{n} (x_i - \bar{x})^2. \tag{21}$$

The *weighted* **population** *variance* [21, 29] is defined as

$$\frac{\sum_{i=1}^{n} w_i (x_i - \mu)^2}{\sum_{i=1}^{n} w_i} \tag{22}$$

and the *weighted* **sample** *variance* [21, 29] is defined as

$$\frac{\sum_{i=1}^{n} w_i}{\left(\sum_{i=1}^{n} w_i\right)^2 - \sum_{i=1}^{n} w_i^2} \sum_{i=1}^{n} w_i (x_i - \mu)^2. \tag{23}$$

Variance (and standard deviation) can be computed in **linear** time [30, 31]. Population and sample variance (and standard deviation) are computed using factors of $1/n$ and $1/(n-1)$, respectively. Other factors might be used instead, $1/(n-1.5)$ as an example [32, 33]. To allow for such factors, this proposal, like NumPy [34], enables one to specify *delta degrees of freedom* [34], a value subtracted from n.

2.5 Standard Deviation

The **population** *standard deviation* [19] ($n \geq 1$), denoted σ, is defined as the square root of the population variance. The **sample** *standard deviation* [19] ($n \geq 2$), denoted s, is defined as the square root of the sample variance. Likewise, the *weighted* **population** *standard deviation* is defined as the square root of the weighted population variance and the *weighted* **sample** *standard deviation* [35] is defined as the square root of the weighted sample variance.

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 Freestanding Functions vs. Accumulator Objects

Perhaps the most significant concern stemming from this proposal is that of freestanding functions versus accumulator objects. In the first incarnation of this proposal, namely P1708R0, freestanding functions were (exclusively) proposed. Freestanding **functions** are useful when one wishes to compute a **single** statistic. Accumulator 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 [36]. Like Boost Accumulators, a programmer has the ability to create **custom** accumulator objects. Given that each of these two paradigms has merit, with freestanding 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. Note that the skewness, kurtosis, variance and standard deviation accumulator objects do not include overloaded constructors to accept a (precomputed) mean (or standard deviation in the case of the skewness and kurtosis), as this might ultimately require that an implementation include a (rather inefficient) branching statement or virtual function call for each value (of a range). And, like existing items 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 [37] over performance for instance.

4.2 Trimmed Mean

The issue of a trimmed mean is raised in [38]. A $p\%$ *trimmed mean* [39] 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 [38]) 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 !isnan(x); }) | std::mean;
```

4.4 Concepts

Much like `std::complex`, the proposed (template) functions and accumulator 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 authors of this proposal in regards 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.

4.5 Projections

The functions and accumulator 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 in this version of the proposal, thereby yielding simpler functions and accumulator objects. An example that demonstrates the use of views is presented in Section A.

4.6 Function and Accumulator Object Names

The formulations of four statistics, namely skewness, kurtosis, variance and standard deviation, differ slightly, depending on whether they are computed over a population or sample. Skewness and kurtosis additionally take on multiple forms. One option, pursued in earlier versions of this proposal, is to select from among these variants using enumerated values. Seeing that there are few variants, it is felt that it is better to instead create unique functions and accumulator objects for each variant. The same is true of weighted variants of functions, which were previously proposed to be overloaded variants of non-weighted functions. As a consequence, all functions and accumulator objects follow a **standard** naming convention, one that includes (descriptive) terms such as `weighted` and `sample`. This standard naming scheme allows for a more intuitive interface.

4.7 Header and Namespace

Early versions of this proposal, specifically P1708R0, P1708R1 and P17082, request that the proposed functions and accumulator objects be placed into the `<numeric>` header. Since P1708R3, it is instead suggested that the functions and objects be placed into a (new) **header** `<stats>`, 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, simple statistics fit into the existing `std` **namespace**.

5 Technical Specifications

The templates of the functions and classes 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.

5.1 Header `<stats>` synopsis [stats.syn]

```
#include <execution>

namespace std {

// freestanding mean functions
template<ranges::input_range R>
constexpr auto mean(R&& r) -> ranges::iterator_t<R>::value_type;

template<ranges::input_range R, ranges::input_range W>
constexpr auto weighted_mean(R&& r, W&& w) -> ranges::iterator_t<R>::value_type;

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

template<class ExecutionPolicy, ranges::forward_range R, ranges::forward_range W>
requires std::is_execution_policy_v<std::remove_cvref_t<ExecutionPolicy>>
auto weighted_mean(ExecutionPolicy&& policy, R&& r, W&& w) ->
  ranges::iterator_t<R>::value_type;

// freestanding geometric mean functions
template<ranges::input_range R>
constexpr auto geometric_mean(R&& r) -> ranges::iterator_t<R>::value_type;

template<ranges::input_range R, ranges::input_range W>
constexpr auto weighted_geometric_mean(R&& r, W&& w) -> ranges::iterator_t<R>::value_type;
```

```
template<class ExecutionPolicy, ranges::forward_range R>
requires std::is_execution_policy_v<std::remove_cvref_t<ExecutionPolicy>>
auto geometric_mean(ExecutionPolicy&& policy, R&& r) -> ranges::iterator_t<R>::value_type;

template<class ExecutionPolicy, ranges::forward_range R, ranges::forward_range W>
requires std::is_execution_policy_v<std::remove_cvref_t<ExecutionPolicy>>
auto weighted_geometric_mean(ExecutionPolicy&& policy, R&& r, W&& w) ->
  ranges::iterator_t<R>::value_type;

// freestanding harmonic mean functions
template<ranges::input_range R>
constexpr auto harmonic_mean(R&& r) -> ranges::iterator_t<R>::value_type;

template<ranges::input_range R, ranges::input_range W>
constexpr auto weighted_harmonic_mean(R&& r, W&& w) -> ranges::iterator_t<R>::value_type;

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

template<class ExecutionPolicy, ranges::forward_range R, ranges::forward_range W>
requires std::is_execution_policy_v<std::remove_cvref_t<ExecutionPolicy>>
auto weighted_harmonic_mean(ExecutionPolicy&& policy, R&& r, W&& w) ->
  ranges::iterator_t<R>::value_type;

// freestanding skewness functions
template<ranges::input_range R>
constexpr auto unadjusted_population_skewness(R&& r) -> ranges::iterator_t<R>::value_type;

template<ranges::input_range R>
constexpr auto unadjusted_sample_skewness(R&& r) -> ranges::iterator_t<R>::value_type;

template<ranges::input_range R>
constexpr auto adjusted_fisher_pearson_population_skewness(R&& r) ->
  ranges::iterator_t<R>::value_type;

template<ranges::input_range R>
constexpr auto adjusted_fisher_pearson_sample_skewness(R&& r) ->
  ranges::iterator_t<R>::value_type;

template<ranges::input_range R, ranges::input_range W>
constexpr auto weighted_unadjusted_population_skewness(R&& r, W&& w) ->
  ranges::iterator_t<R>::value_type;

template<ranges::input_range R, ranges::input_range W>
constexpr auto weighted_unadjusted_sample_skewness(R&& r, W&& w) ->
  ranges::iterator_t<R>::value_type;

template<ranges::input_range R, ranges::input_range W>
constexpr auto weighted_adjusted_fisher_pearson_population_skewness(R&& r, W&& w) ->
  ranges::iterator_t<R>::value_type;

template<ranges::input_range R, ranges::input_range W>
constexpr auto weighted_adjusted_fisher_pearson_sample_skewness(R&& r, W&& w) ->
  ranges::iterator_t<R>::value_type;

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

```
requires std::is_execution_policy_v<std::remove_cvref_t<ExecutionPolicy>>
constexpr auto unadjusted_population_skewness(ExecutionPolicy&& policy, R&& r) ->
  ranges::iterator_t<R>::value_type;

template<class ExecutionPolicy, ranges::input_range R>
requires std::is_execution_policy_v<std::remove_cvref_t<ExecutionPolicy>>
constexpr auto unadjusted_sample_skewness(ExecutionPolicy&& policy, R&& r) ->
  ranges::iterator_t<R>::value_type;

template<class ExecutionPolicy, ranges::input_range R>
requires std::is_execution_policy_v<std::remove_cvref_t<ExecutionPolicy>>
constexpr auto adjusted_fisher_pearson_population_skewness(
  ExecutionPolicy&& policy, R&& r) -> ranges::iterator_t<R>::value_type;

template<class ExecutionPolicy, ranges::input_range R>
requires std::is_execution_policy_v<std::remove_cvref_t<ExecutionPolicy>>
constexpr auto adjusted_fisher_pearson_sample_skewness(ExecutionPolicy&& policy, R&& r) ->
  ranges::iterator_t<R>::value_type;

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

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

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

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

// freestanding kurtosis functions
template<ranges::input_range R>
constexpr auto fisher_population_kurtosis(R&& r) -> ranges::iterator_t<R>::value_type;

template<ranges::input_range R>
constexpr auto fisher_sample_kurtosis(R&& r) -> ranges::iterator_t<R>::value_type;

template<ranges::input_range R>
constexpr auto pearson_population_kurtosis(R&& r) -> ranges::iterator_t<R>::value_type;

template<ranges::input_range R>
constexpr auto pearson_sample_kurtosis(R&& r) -> ranges::iterator_t<R>::value_type;

template<ranges::input_range R, ranges::input_range W>
constexpr auto weighted_fisher_population_kurtosis(R&& r, W&& w) ->
  ranges::iterator_t<R>::value_type;

template<ranges::input_range R, ranges::input_range W>
constexpr auto weighted_fisher_sample_kurtosis(R&& r, W&& w) ->
```

```
  ranges::iterator_t<R>::value_type;

template<ranges::input_range R, ranges::input_range W>
constexpr auto weighted_pearson_population_kurtosis(R&& r, W&& w) ->
  ranges::iterator_t<R>::value_type;

template<ranges::input_range R, ranges::input_range W>
constexpr auto weighted_pearson_sample_kurtosis(R&& r, W&& w) ->
  ranges::iterator_t<R>::value_type;

template<class ExecutionPolicy, ranges::input_range R>
requires std::is_execution_policy_v<std::remove_cvref_t<ExecutionPolicy>>
constexpr auto fisher_population_kurtosis(ExecutionPolicy&& policy, R&& r) ->
  ranges::iterator_t<R>::value_type;

template<class ExecutionPolicy, ranges::input_range R>
requires std::is_execution_policy_v<std::remove_cvref_t<ExecutionPolicy>>
constexpr auto fisher_sample_kurtosis(ExecutionPolicy&& policy, R&& r) ->
  ranges::iterator_t<R>::value_type;

template<class ExecutionPolicy, ranges::input_range R>
requires std::is_execution_policy_v<std::remove_cvref_t<ExecutionPolicy>>
constexpr auto pearson_population_kurtosis(ExecutionPolicy&& policy, R&& r) ->
  ranges::iterator_t<R>::value_type;

template<class ExecutionPolicy, ranges::input_range R>
requires std::is_execution_policy_v<std::remove_cvref_t<ExecutionPolicy>>
constexpr auto pearson_sample_kurtosis(ExecutionPolicy&& policy, R&& r) ->
  ranges::iterator_t<R>::value_type;

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

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

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

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

// freestanding variance functions
template<ranges::input_range R>
constexpr auto variance(R&& r, typename ranges::iterator_t<R>::value_type ddof) ->
  ranges::iterator_t<R>::value_type;

template<ranges::input_range R, ranges::input_range W>
constexpr auto weighted_population_variance(R&& r, W&& w) ->
  ranges::iterator_t<R>::value_type;
```

```
template<ranges::input_range R, ranges::input_range W>
constexpr auto weighted_sample_variance(R&& r, W&& w) ->
  ranges::iterator_t<R>::value_type;

template<class ExecutionPolicy, ranges::forward_range R>
requires std::is_execution_policy_v<std::remove_cvref_t<ExecutionPolicy>>
auto variance(
  ExecutionPolicy&& policy, R&& r, typename ranges::iterator_t<R>::value_type ddof) ->
    ranges::iterator_t<R>::value_type;

template<class ExecutionPolicy, ranges::forward_range R, ranges::forward_range W>
requires std::is_execution_policy_v<std::remove_cvref_t<ExecutionPolicy>>
auto weighted_population_variance(ExecutionPolicy&& policy, R&& r, W&& w) ->
  ranges::iterator_t<R>::value_type;

template<class ExecutionPolicy, ranges::forward_range R, ranges::forward_range W>
requires std::is_execution_policy_v<std::remove_cvref_t<ExecutionPolicy>>
auto weighted_sample_variance(ExecutionPolicy&& policy, R&& r, W&& w) ->
  ranges::iterator_t<R>::value_type;

// freestanding standard deviation functions
template<ranges::input_range R>
constexpr auto standard_deviation(
  R&& r, typename ranges::iterator_t<R>::value_type ddof) ->
    ranges::iterator_t<R>::value_type;

template<ranges::input_range R, ranges::input_range W>
constexpr auto weighted_population_standard_deviation(R&& r, W&& w) ->
  ranges::iterator_t<R>::value_type;

template<ranges::input_range R, ranges::input_range W>
constexpr auto weighted_sample_standard_deviation(R&& r, W&& w) ->
  ranges::iterator_t<R>::value_type;

template<class ExecutionPolicy, ranges::forward_range R>
requires std::is_execution_policy_v<std::remove_cvref_t<ExecutionPolicy>>
auto standard_deviation(
  ExecutionPolicy&& policy, R&& r, typename ranges::iterator_t<R>::value_type ddof) ->
    ranges::iterator_t<R>::value_type;

template<class ExecutionPolicy, ranges::forward_range R, ranges::forward_range W>
requires std::is_execution_policy_v<std::remove_cvref_t<ExecutionPolicy>>
auto weighted_population_standard_deviation(ExecutionPolicy&& policy, R&& r, W&& w) ->
  ranges::iterator_t<R>::value_type;

template<class ExecutionPolicy, ranges::forward_range R, ranges::forward_range W>
requires std::is_execution_policy_v<std::remove_cvref_t<ExecutionPolicy>>
auto weighted_sample_standard_deviation(ExecutionPolicy&& policy, R&& r, W&& w) ->
  ranges::iterator_t<R>::value_type;

// mean accumulator class templates
template<class T>
class mean_accumulator;

template<class T, class W = double>
class weighted_mean_accumulator;
```

```
// geometric mean accumulator class templates
template<class T>
class geometric_mean_accumulator;

template<class T, class W = double>
class weighted_geometric_mean_accumulator;

// harmonic mean accumulator class templates
template<class T>
class harmonic_mean_accumulator;

template<class T, class W = double>
class weighted_harmonic_mean_accumulator;

// skewness accumulator class templates
template<class T>
class unadjusted_population_skewness_accumulator;

template<class T>
class unadjusted_sample_skewness_accumulator;

template<class T>
class adjusted_fisher_pearson_population_skewness_accumulator;

template<class T>
class adjusted_fisher_pearson_sample_skewness_accumulator;

template<class T, class W = double>
class weighted_unadjusted_population_skewness_accumulator;

template<class T, class W = double>
class weighted_unadjusted_sample_skewness_accumulator;

template<class T, class W = double>
class weighted_adjusted_fisher_pearson_population_skewness_accumulator;

template<class T, class W = double>
class weighted_adjusted_fisher_pearson_sample_skewness_accumulator;

// kurtosis accumulator class templates
template<class T>
class fisher_population_kurtosis_accumulator;

template<class T>
class fisher_sample_kurtosis_accumulator;

template<class T>
class pearson_population_kurtosis_accumulator;

template<class T>
class pearson_sample_kurtosis_accumulator;

template<class T, class W = double>
class weighted_fisher_population_kurtosis_accumulator;

template<class T, class W = double>
class weighted_fisher_sample_kurtosis_accumulator;
```

```
template<class T, class W = double>
class weighted_pearson_population_kurtosis_accumulator;

template<class T, class W = double>
class weighted_pearson_sample_kurtosis_accumulator;

// variance accumulator class templates
template<class T>
class variance_accumulator;

template<class T, class W = double>
class weighted_population_variance_accumulator;

template<class T, class W = double>
class weighted_sample_variance_accumulator;

// standard deviation accumulator class templates
template<class T>
class standard_deviation_accumulator;

template<class T, class W = double>
class weighted_population_standard_deviation_accumulator;

template<class T, class W = double>
class weighted_sample_standard_deviation_accumulator;

// accumulator objects accumulation functions
template<ranges::input_range R, class ...Accumulators>
constexpr void stats_accumulate(R&& r, Accumulators&& ... acc);

template<ranges::input_range R, ranges::input_range W, class ...Accumulators>
constexpr void stats_weighted_accumulate(R&& r, W&& w, Accumulators&& ... acc);

template<class ExecutionPolicy, ranges::input_range R, class ...Accumulators>
requires std::is_execution_policy_v<std::remove_cvref_t<ExecutionPolicy>>
void stats_accumulate(ExecutionPolicy&& policy, R&& r, Accumulators&& ... acc);

template<class ExecutionPolicy,
  ranges::input_range R, ranges::input_range W,
  class ...Accumulators>
requires std::is_execution_policy_v<std::remove_cvref_t<ExecutionPolicy>>
void stats_weighted_accumulate(
  ExecutionPolicy&& policy, R&& r, W&& w, Accumulators&& ... acc);
}
```

5.2 Freestanding Functions

If any of the values of the ranges `r` or `w` of the functions specified in this section is a NaN (Not a Number), ∞ or $-\infty$, the function shall return an unspecified value, but it shall not report a domain error.

5.2.1 Freestanding Mean Functions

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

template<ranges::input_range R, ranges::input_range W>
constexpr auto weighted_mean(R&& r, W&& w) -> ranges::iterator_t<R>::value_type;
```

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

template<class ExecutionPolicy, ranges::forward_range R, ranges::forward_range W>
requires std::is_execution_policy_v<std::remove_cvref_t<ExecutionPolicy>>
auto weighted_mean(ExecutionPolicy&& policy, R&& r, W&& w) ->
  ranges::iterator_t<R>::value_type;
```

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) mean of the values of `r` (weighted by the corresponding values of `w`).
3. *Complexity*: Linear in `ranges::distance(r)`.

5.2.2 Freestanding Geometric Mean Functions

```
template<ranges::input_range R>
constexpr auto geometric_mean(R&& r) -> ranges::iterator_t<R>::value_type;

template<ranges::input_range R, ranges::input_range W>
constexpr auto weighted_geometric_mean(R&& r, W&& w) -> ranges::iterator_t<R>::value_type;

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

template<class ExecutionPolicy, ranges::forward_range R, ranges::forward_range W>
requires std::is_execution_policy_v<std::remove_cvref_t<ExecutionPolicy>>
auto weighted_geometric_mean(ExecutionPolicy&& policy, R&& r, W&& w) ->
  ranges::iterator_t<R>::value_type;
```

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) geometric mean of the values of `r` (weighted by the corresponding values of `w`). If the product of the values of `r` is negative and `ranges::distance(r)` is even, the function shall return an unspecified value, but it shall not report a domain error.
3. *Complexity*: Linear in `ranges::distance(r)`.

5.2.3 Freestanding Harmonic Mean Functions

```
template<ranges::input_range R>
constexpr auto harmonic_mean(R&& r) -> ranges::iterator_t<R>::value_type;

template<ranges::input_range R, ranges::input_range W>
constexpr auto weighted_harmonic_mean(R&& r, W&& w) -> ranges::iterator_t<R>::value_type;

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

template<class ExecutionPolicy, ranges::forward_range R, ranges::forward_range W>
requires std::is_execution_policy_v<std::remove_cvref_t<ExecutionPolicy>>
auto weighted_harmonic_mean(ExecutionPolicy&& policy, R&& r, W&& w) ->
  ranges::iterator_t<R>::value_type;
```

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) harmonic mean of the values of `r` (weighted by the corresponding values of `w`). If any of the values of `r` is negative or zero, the function shall return an unspecified value, but it shall not report a domain error.

3. *Complexity*: Linear in `ranges::distance(r)`.

5.2.4 Freestanding Skewness Functions

```
template<ranges::input_range R>
constexpr auto unadjusted_population_skewness(R&& r) -> ranges::iterator_t<R>::value_type;

template<ranges::input_range R>
constexpr auto unadjusted_sample_skewness(R&& r) -> ranges::iterator_t<R>::value_type;

template<ranges::input_range R>
constexpr auto adjusted_fisher_pearson_population_skewness(R&& r) ->
  ranges::iterator_t<R>::value_type;

template<ranges::input_range R>
constexpr auto adjusted_fisher_pearson_sample_skewness(R&& r) ->
  ranges::iterator_t<R>::value_type;

template<ranges::input_range R, ranges::input_range W>
constexpr auto weighted_unadjusted_population_skewness(R&& r, W&& w) ->
  ranges::iterator_t<R>::value_type;

template<ranges::input_range R, ranges::input_range W>
constexpr auto weighted_unadjusted_sample_skewness(R&& r, W&& w) ->
  ranges::iterator_t<R>::value_type;

template<ranges::input_range R, ranges::input_range W>
constexpr auto weighted_adjusted_fisher_pearson_population_skewness(R&& r, W&& w) ->
  ranges::iterator_t<R>::value_type;

template<ranges::input_range R, ranges::input_range W>
constexpr auto weighted_adjusted_fisher_pearson_sample_skewness(R&& r, W&& w) ->
  ranges::iterator_t<R>::value_type;

template<class ExecutionPolicy, ranges::input_range R>
requires std::is_execution_policy_v<std::remove_cvref_t<ExecutionPolicy>>
constexpr auto unadjusted_population_skewness(ExecutionPolicy&& policy, R&& r) ->
  ranges::iterator_t<R>::value_type;

template<class ExecutionPolicy, ranges::input_range R>
requires std::is_execution_policy_v<std::remove_cvref_t<ExecutionPolicy>>
constexpr auto unadjusted_sample_skewness(ExecutionPolicy&& policy, R&& r) ->
  ranges::iterator_t<R>::value_type;

template<class ExecutionPolicy, ranges::input_range R>
requires std::is_execution_policy_v<std::remove_cvref_t<ExecutionPolicy>>
constexpr auto adjusted_fisher_pearson_population_skewness(
  ExecutionPolicy&& policy, R&& r) -> ranges::iterator_t<R>::value_type;

template<class ExecutionPolicy, ranges::input_range R>
requires std::is_execution_policy_v<std::remove_cvref_t<ExecutionPolicy>>
constexpr auto adjusted_fisher_pearson_sample_skewness(ExecutionPolicy&& policy, R&& r) ->
  ranges::iterator_t<R>::value_type;
```

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

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

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

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

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

5.2.5 Freestanding Kurtosis Functions

```
template<ranges::input_range R>
constexpr auto fisher_population_kurtosis(R&& r) -> ranges::iterator_t<R>::value_type;

template<ranges::input_range R>
constexpr auto fisher_sample_kurtosis(R&& r) -> ranges::iterator_t<R>::value_type;

template<ranges::input_range R>
constexpr auto pearson_population_kurtosis(R&& r) -> ranges::iterator_t<R>::value_type;

template<ranges::input_range R>
constexpr auto pearson_sample_kurtosis(R&& r) -> ranges::iterator_t<R>::value_type;

template<ranges::input_range R, ranges::input_range W>
constexpr auto weighted_fisher_population_kurtosis(R&& r, W&& w) ->
  ranges::iterator_t<R>::value_type;

template<ranges::input_range R, ranges::input_range W>
constexpr auto weighted_fisher_sample_kurtosis(R&& r, W&& w) ->
  ranges::iterator_t<R>::value_type;

template<ranges::input_range R, ranges::input_range W>
constexpr auto weighted_pearson_population_kurtosis(R&& r, W&& w) ->
  ranges::iterator_t<R>::value_type;

template<ranges::input_range R, ranges::input_range W>
constexpr auto weighted_pearson_sample_kurtosis(R&& r, W&& w) ->
  ranges::iterator_t<R>::value_type;
```

```
template<class ExecutionPolicy, ranges::input_range R>
requires std::is_execution_policy_v<std::remove_cvref_t<ExecutionPolicy>>
constexpr auto fisher_population_kurtosis(ExecutionPolicy&& policy, R&& r) ->
  ranges::iterator_t<R>::value_type;

template<class ExecutionPolicy, ranges::input_range R>
requires std::is_execution_policy_v<std::remove_cvref_t<ExecutionPolicy>>
constexpr auto fisher_sample_kurtosis(ExecutionPolicy&& policy, R&& r) ->
  ranges::iterator_t<R>::value_type;

template<class ExecutionPolicy, ranges::input_range R>
requires std::is_execution_policy_v<std::remove_cvref_t<ExecutionPolicy>>
constexpr auto pearson_population_kurtosis(ExecutionPolicy&& policy, R&& r) ->
  ranges::iterator_t<R>::value_type;

template<class ExecutionPolicy, ranges::input_range R>
requires std::is_execution_policy_v<std::remove_cvref_t<ExecutionPolicy>>
constexpr auto pearson_sample_kurtosis(ExecutionPolicy&& policy, R&& r) ->
  ranges::iterator_t<R>::value_type;

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

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

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

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

1. *Preconditions*: `r` and `w` are ranges of finite values, where `r` has at least `4` values and the length of `r` is less than or equal to the length of `w`.
2. *Returns*: The (weighted) kurtosis of the values of `r` (weighted by the corresponding values of `w`).
3. *Complexity*: Linear in `ranges::distance(r)`.

5.2.6 Freestanding Variance Functions

```
template<ranges::input_range R>
constexpr auto variance(R&& r, typename ranges::iterator_t<R>::value_type ddof) ->
  ranges::iterator_t<R>::value_type;

template<ranges::input_range R, ranges::input_range W>
constexpr auto weighted_population_variance(R&& r, W&& w) ->
  ranges::iterator_t<R>::value_type;

template<ranges::input_range R, ranges::input_range W>
```

```
constexpr auto weighted_sample_variance(R&& r, W&& w) ->
  ranges::iterator_t<R>::value_type;

template<class ExecutionPolicy, ranges::forward_range R>
requires std::is_execution_policy_v<std::remove_cvref_t<ExecutionPolicy>>
auto variance(
  ExecutionPolicy&& policy, R&& r, typename ranges::iterator_t<R>::value_type ddof) ->
    ranges::iterator_t<R>::value_type;

template<class ExecutionPolicy, ranges::forward_range R, ranges::forward_range W>
requires std::is_execution_policy_v<std::remove_cvref_t<ExecutionPolicy>>
auto weighted_population_variance(ExecutionPolicy&& policy, R&& r, W&& w) ->
  ranges::iterator_t<R>::value_type;

template<class ExecutionPolicy, ranges::forward_range R, ranges::forward_range W>
requires std::is_execution_policy_v<std::remove_cvref_t<ExecutionPolicy>>
auto weighted_sample_variance(ExecutionPolicy&& policy, R&& r, W&& w) ->
  ranges::iterator_t<R>::value_type;
```

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) variance of the values of `r` (weighted by the corresponding values of `w`). If `ddof` is equal to `ranges::distance(r)`, the function shall return an unspecified value, but it shall not report a domain error.
3. *Complexity*: Linear in `ranges::distance(r)`.

5.2.7 Freestanding Standard Deviation Functions

```
template<ranges::input_range R>
constexpr auto standard_deviation(
  R&& r, typename ranges::iterator_t<R>::value_type ddof) ->
    ranges::iterator_t<R>::value_type;

template<ranges::input_range R, ranges::input_range W>
constexpr auto weighted_population_standard_deviation(R&& r, W&& w) ->
  ranges::iterator_t<R>::value_type;

template<ranges::input_range R, ranges::input_range W>
constexpr auto weighted_sample_standard_deviation(R&& r, W&& w) ->
  ranges::iterator_t<R>::value_type;

template<class ExecutionPolicy, ranges::forward_range R>
requires std::is_execution_policy_v<std::remove_cvref_t<ExecutionPolicy>>
auto standard_deviation(
  ExecutionPolicy&& policy, R&& r, typename ranges::iterator_t<R>::value_type ddof) ->
    ranges::iterator_t<R>::value_type;

template<class ExecutionPolicy, ranges::forward_range R, ranges::forward_range W>
requires std::is_execution_policy_v<std::remove_cvref_t<ExecutionPolicy>>
auto weighted_population_standard_deviation(ExecutionPolicy&& policy, R&& r, W&& w) ->
  ranges::iterator_t<R>::value_type;

template<class ExecutionPolicy, ranges::forward_range R, ranges::forward_range W>
requires std::is_execution_policy_v<std::remove_cvref_t<ExecutionPolicy>>
auto weighted_sample_standard_deviation(ExecutionPolicy&& policy, R&& r, W&& w) ->
  ranges::iterator_t<R>::value_type;
```

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) standard deviation of the values of `r` (weighted by the corresponding values of `w`). If `ddof` is equal to `ranges::distance(r)`, the function shall return an unspecified value, but it shall not report a domain error.
3. *Complexity*: Linear in `ranges::distance(r)`.

5.3 Accumulator Objects

The accumulator objects specified in this section shall be trivially copyable. If any of the values of `x` or `w` of the function **`operator()`** specified in this section is a NaN, ∞ or $-\infty$, the function `value` shall return an unspecified value, but it shall not report a domain error.

5.3.1 `mean_accumulator` class templates

```
template<class T>
class mean_accumulator
{
public:
  explicit constexpr mean_accumulator() noexcept;
  constexpr void operator()(const T& x);
  constexpr auto value() const -> T;
};

template<class T, class W = double>
class weighted_mean_accumulator
{
public:
  explicit constexpr weighted_mean_accumulator() noexcept;
  constexpr void operator()(const T& x, const W& w);
  constexpr auto value() const -> T;
};
```

```
explicit constexpr mean_accumulator() noexcept;
explicit constexpr weighted_mean_accumulator() noexcept;
```

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

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

1. *Effects*: The value of `x` is accumulated.
2. *Complexity*: Constant.

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

1. *Effects*: The value of `x`, weighted by `w`, is accumulated.
2. *Complexity*: Constant.

```
constexpr auto value() const -> Result;
```

1. *Preconditions*: The (weighted) values of the associated range `r` (weighted by the corresponding values of `w`) have been accumulated, where `r` has at least 1 value and the length of `r` is less than or equal to the length of `w`.
2. *Effects*: Any remaining computations relating to the mean are performed.
3. *Returns*: The (weighted) mean of the values of the associated range `r` (weighted by the corresponding values of `w`).
4. *Complexity*: Constant.

5.3.2 `geometric_mean_accumulator` class templates

```
template<class T>
class geometric_mean_accumulator
{
public:
  explicit constexpr geometric_mean_accumulator() noexcept;
  constexpr void operator()(const T& x);
  constexpr auto value() const -> T;
};

template<class T, class W = double>
class weighted_geometric_mean_accumulator
{
public:
  explicit constexpr weighted_geometric_mean_accumulator() noexcept;
  constexpr void operator()(const T& x, const W& w);
  constexpr auto value() const -> T;
};
```

```
explicit constexpr geometric_mean_accumulator() noexcept;
explicit constexpr weighted_geometric_mean_accumulator() noexcept;
```

1. *Effects*: A (weighted) geometric mean accumulator object is constructed.
2. *Complexity*: Constant.

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

1. *Effects*: The value of `x` is accumulated.
2. *Complexity*: Constant.

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

1. *Effects*: The value of `x`, weighted by `w`, is accumulated.
2. *Complexity*: Constant.

```
constexpr auto value() const -> Result;
```

1. *Preconditions*: The (weighted) values of the associated range `r` (weighted by the corresponding values of `w`) have been accumulated, where `r` has at least 1 value and the length of `r` is less than or equal to the length of `w`.
2. *Effects*: Any remaining computations relating to the geometric mean are performed.
3. *Returns*: The (weighted) geometric mean of the values of the associated range `r` (weighted by the corresponding values of `w`).
4. *Complexity*: Constant.

5.3.3 `harmonic_mean_accumulator` class templates

```
template<class T>
class harmonic_mean_accumulator
{
public:
  explicit constexpr harmonic_mean_accumulator() noexcept;
  constexpr void operator()(const T& x);
  constexpr auto value() const -> T;
};

template<class T, class W = double>
class weighted_harmonic_mean_accumulator
{
public:
  explicit constexpr weighted_harmonic_mean_accumulator() noexcept;
  constexpr void operator()(const T& x, const W& w);
  constexpr auto value() const -> T;
};
```

```
explicit constexpr harmonic_mean_accumulator() noexcept;
explicit constexpr weighted_harmonic_mean_accumulator() noexcept;
```

1. *Effects*: A (weighted) harmonic mean accumulator object is constructed.
2. *Complexity*: Constant.

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

1. *Effects*: The value of `x` is accumulated.
2. *Complexity*: Constant.

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

1. *Effects*: The value of `x`, weighted by `w`, is accumulated.
2. *Complexity*: Constant.

```
constexpr auto value() const -> Result;
```

1. *Preconditions*: The (weighted) values of the associated range `r` (weighted by the corresponding values of `w`) have been accumulated, where `r` has at least 1 value and the length of `r` is less than or equal to the length of `w`.
2. *Effects*: Any remaining computations relating to the harmonic mean are performed.
3. *Returns*: The (weighted) harmonic mean of the values of the associated range `r` (weighted by the corresponding values of `w`).
4. *Complexity*: Constant.

5.3.4 `skewness_accumulator` class templates

```
template<class T>
class unadjusted_population_skewness_accumulator
{
public:
  explicit constexpr unadjusted_population_skewness_accumulator() noexcept;
  constexpr void operator()(const T& x);
  constexpr auto value() const -> T;
};

template<class T>
class unadjusted_sample_skewness_accumulator
{
public:
  explicit constexpr unadjusted_sample_skewness_accumulator() noexcept;
  constexpr void operator()(const T& x);
  constexpr auto value() const -> T;
};

template<class T>
class adjusted_fisher_pearson_population_skewness_accumulator
{
public:
  explicit constexpr adjusted_fisher_pearson_population_skewness_accumulator() noexcept;
  constexpr void operator()(const T& x);
  constexpr auto value() const -> T;
};

template<class T>
class adjusted_fisher_pearson_sample_skewness_accumulator
{
public:
  explicit constexpr adjusted_fisher_pearson_sample_skewness_accumulator() noexcept;
  constexpr void operator()(const T& x);
  constexpr auto value() const -> T;
};

template<class T>
class weighted_unadjusted_population_skewness_accumulator
{
public:
  explicit constexpr weighted_unadjusted_population_skewness_accumulator() noexcept;
  constexpr void operator()(const T& x, const W& w);
  constexpr auto value() const -> T;
};

template<class T>
class weighted_unadjusted_sample_skewness_accumulator
{
public:
  explicit constexpr weighted_unadjusted_sample_skewness_accumulator() noexcept;
  constexpr void operator()(const T& x, const W& w);
  constexpr auto value() const -> T;
};

template<class T>
class weighted_adjusted_fisher_pearson_population_skewness_accumulator
```

```
{
public:
  explicit constexpr
    weighted_adjusted_fisher_pearson_population_skewness_accumulator() noexcept;
  constexpr void operator()(const T& x, const W& w);
  constexpr auto value() const -> T;
};

template<class T>
class weighted_adjusted_fisher_pearson_sample_skewness_accumulator
{
public:
  explicit constexpr
    weighted_adjusted_fisher_pearson_sample_skewness_accumulator() noexcept;
  constexpr void operator()(const T& x, const W& w);
  constexpr auto value() const -> T;
};
```

```
explicit constexpr unadjusted_population_skewness_accumulator() noexcept;
explicit constexpr unadjusted_sample_skewness_accumulator() noexcept;
explicit constexpr adjusted_fisher_pearson_population_skewness_accumulator() noexcept;
explicit constexpr adjusted_fisher_pearson_sample_skewness_accumulator() noexcept;
explicit constexpr weighted_unadjusted_population_skewness_accumulator() noexcept;
explicit constexpr weighted_unadjusted_sample_skewness_accumulator() noexcept;
explicit constexpr
  weighted_adjusted_fisher_pearson_population_skewness_accumulator() noexcept;
explicit constexpr
  weighted_adjusted_fisher_pearson_sample_skewness_accumulator() noexcept;
```

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

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

1. *Effects*: The value of `x` is accumulated.
2. *Complexity*: Constant.

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

1. *Effects*: The value of `x`, weighted by `w`, is accumulated.
2. *Complexity*: Constant.

```
constexpr auto value() const -> Result;
```

1. *Preconditions*: The (weighted) values of the associated range `r` (weighted by the corresponding values of `w`) have been accumulated, where `r` has at least 3 value and the length of `r` is less than or equal to the length of `w`.
2. *Effects*: Any remaining computations relating to the skewness are performed.
3. *Returns*: The (weighted) skewness of the values of the associated range `r` (weighted by the corresponding values of `w`).
4. *Complexity*: Constant.

5.3.5 `kurtosis_accumulator` class templates

```
template<class T>
class fisher_population_kurtosis_accumulator
{
public:
  explicit constexpr fisher_population_kurtosis_accumulator() noexcept;
  constexpr void operator()(const T& x);
  constexpr auto value() const -> T;
};

template<class T>
class fisher_sample_kurtosis_accumulator
{
public:
  explicit constexpr fisher_sample_kurtosis_accumulator() noexcept;
  constexpr void operator()(const T& x);
  constexpr auto value() const -> T;
};

template<class T>
class pearson_population_kurtosis_accumulator
{
public:
  explicit constexpr pearson_population_kurtosis_accumulator() noexcept;
  constexpr void operator()(const T& x);
  constexpr auto value() const -> T;
};

template<class T>
class pearson_sample_kurtosis_accumulator
{
public:
  explicit constexpr pearson_sample_kurtosis_accumulator() noexcept;
  constexpr void operator()(const T& x);
  constexpr auto value() const -> T;
};

template<class T>
class weighted_fisher_population_kurtosis_accumulator
{
public:
  explicit constexpr weighted_fisher_population_kurtosis_accumulator() noexcept;
  constexpr void operator()(const T& x, const W& w);
  constexpr auto value() const -> T;
};

template<class T>
class weighted_fisher_sample_kurtosis_accumulator
{
public:
  explicit constexpr weighted_fisher_sample_kurtosis_accumulator() noexcept;
  constexpr void operator()(const T& x, const W& w);
  constexpr auto value() const -> T;
};

template<class T>
class weighted_pearson_population_kurtosis_accumulator
```

```
{
public:
  explicit constexpr weighted_pearson_population_kurtosis_accumulator() noexcept;
  constexpr void operator()(const T& x, const W& w);
  constexpr auto value() const -> T;
};

template<class T>
class weighted_pearson_sample_kurtosis_accumulator
{
public:
  explicit constexpr weighted_pearson_sample_kurtosis_accumulator() noexcept;
  constexpr void operator()(const T& x, const W& w);
  constexpr auto value() const -> T;
};
```

```
explicit constexpr fisher_population_kurtosis_accumulator() noexcept;
explicit constexpr fisher_sample_kurtosis_accumulator() noexcept;
explicit constexpr pearson_population_kurtosis_accumulator() noexcept;
explicit constexpr pearson_sample_kurtosis_accumulator() noexcept;
explicit constexpr weighted_fisher_population_kurtosis_accumulator() noexcept;
explicit constexpr weighted_fisher_sample_kurtosis_accumulator() noexcept;
explicit constexpr weighted_pearson_population_kurtosis_accumulator() noexcept;
explicit constexpr weighted_pearson_sample_kurtosis_accumulator() noexcept;
```

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

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

1. *Effects*: The value of `x` is accumulated.
2. *Complexity*: Constant.

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

1. *Effects*: The value of `x`, weighted by `w`, is accumulated.
2. *Complexity*: Constant.

```
constexpr auto value() const -> Result;
```

1. *Preconditions*: The (weighted) values of the associated range `r` (weighted by the corresponding values of `w`) have been accumulated, where `r` has at least 4 value and the length of `r` is less than or equal to the length of `w`.
2. *Effects*: Any remaining computations relating to the kurtosis are performed.
3. *Returns*: The (weighted) kurtosis of the values of the associated range `r` (weighted by the corresponding values of `w`).
4. *Complexity*: Constant.

5.3.6 `variance_accumulator` class templates

```
template<class T>
class variance_accumulator
{
public:
  explicit constexpr variance_accumulator(T ddof) noexcept noexcept;
  constexpr void operator()(const T& x);
  constexpr auto value() const -> T;
};

template<class T, class W = double>
class weighted_population_variance_accumulator
{
public:
  explicit constexpr weighted_population_variance_accumulator(T ddof) noexcept noexcept;
  constexpr void operator()(const T& x, const W& w);
  constexpr auto value() const -> T;
};

template<class T, class W = double>
class weighted_sample_variance_accumulator
{
public:
  explicit constexpr weighted_sample_variance_accumulator(T ddof) noexcept noexcept;
  constexpr void operator()(const T& x, const W& w);
  constexpr auto value() const -> T;
};
```

```
explicit constexpr variance_accumulator(T ddof) noexcept noexcept;
explicit constexpr weighted_population_variance_accumulator(T ddof) noexcept noexcept;
explicit constexpr weighted_sample_variance_accumulator(T ddof) noexcept noexcept;
```

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

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

1. *Effects*: The value of `x` is accumulated.
2. *Complexity*: Constant.

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

1. *Effects*: The value of `x`, weighted by `w`, is accumulated.
2. *Complexity*: Constant.

```
constexpr auto value() const -> Result;
```

1. *Preconditions*: The (weighted) values of the associated range `r` (weighted by the corresponding values of `w`) have been accumulated, where `r` has at least 1 value and the length of `r` is less than or equal to the length of `w`.
2. *Effects*: Any remaining computations relating to the variance are performed.
3. *Returns*: The (weighted) variance of the values of the associated range `r` (weighted by the corresponding values of `w`).
4. *Complexity*: Constant.

5.3.7 `standard_deviation_accumulator` class templates

```
template<class T>
class standard_deviation_accumulator
{
public:
  explicit constexpr standard_deviation_accumulator(T ddof) noexcept noexcept;
  constexpr void operator()(const T& x);
  constexpr auto value() const -> T;
};

template<class T, class W = double>
class weighted_population_standard_deviation_accumulator
{
public:
  explicit constexpr
    weighted_population_standard_deviation_accumulator(T ddof) noexcept noexcept;
  constexpr void operator()(const T& x, const W& w);
  constexpr auto value() const -> T;
};

template<class T, class W = double>
class weighted_sample_standard_deviation_accumulator
{
public:
  explicit constexpr
    weighted_sample_standard_deviation_accumulator(T ddof) noexcept noexcept;
  constexpr void operator()(const T& x, const W& w);
  constexpr auto value() const -> T;
};
```

```
explicit constexpr standard_deviation_accumulator(T ddof) noexcept noexcept;
explicit constexpr
  weighted_population_standard_deviation_accumulator(T ddof) noexcept noexcept;
explicit constexpr
  weighted_sample_standard_deviation_accumulator(T ddof) noexcept noexcept;
```

1. *Effects*: A (weighted) standard deviation accumulator object is constructed.
2. *Complexity*: Constant.

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

1. *Effects*: The value of `x` is accumulated.
2. *Complexity*: Constant.

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

1. *Effects*: The value of `x`, weighted by `w`, is accumulated.
2. *Complexity*: Constant.

```
constexpr auto value() const -> Result;
```

1. *Preconditions*: The (weighted) values of the associated range `r` (weighted by the corresponding values of `w`) have been accumulated, where `r` has at least 1 value and the length of `r` is less than or equal to the length of `w`.

2. *Effects*: Any remaining computations relating to the standard deviation are performed.

3. *Returns*: The (weighted) standard deviation of the values of the associated range `r` (weighted by the corresponding values of `w`).

4. *Complexity*: Constant.

5.3.8 Accumulator Objects Accumulation Functions

```
template<ranges::input_range R, class ...Accumulators>
constexpr void stats_accumulate(R&& r, Accumulators&& ... acc);

template<ranges::input_range R, ranges::input_range W, class ...Accumulators>
constexpr void stats_weighted_accumulate(R&& r, W&& w, Accumulators&& ... acc);

template<class ExecutionPolicy, ranges::input_range R, class ...Accumulators>
requires std::is_execution_policy_v<std::remove_cvref_t<ExecutionPolicy>>
void stats_accumulate(ExecutionPolicy&& policy, R&& r, Accumulators&& ... acc);

template<class ExecutionPolicy,
  ranges::input_range R, ranges::input_range W,
  class ...Accumulators>
requires std::is_execution_policy_v<std::remove_cvref_t<ExecutionPolicy>>
void stats_weighted_accumulate(
  ExecutionPolicy&& policy, R&& r, W&& w, Accumulators&& ... acc);
```

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`, `r` has at least 4 values if any of the accumulator objects of `acc` is a kurtosis accumulator object, `r` has at least 3 values if any of the accumulator objects of `acc` is a skewness accumulator object and `r` has at least 1 value otherwise, and `acc` are valid accumulator objects.

2. *Effects*: The (weighted) statistics of the accumulator objects `acc` over the values of `r` (weighted by the corresponding values of `w`) are computed.

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

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

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

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

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

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

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

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

[9] 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.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] 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.

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

[16] 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.

[17] 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.

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

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

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

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

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

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

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

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

[26] Zhiqiang Liang, Jianming Wei, Junyu Zhao, Haitao Liu, Baoqing Li, Jie Shen, and Chunlei Zheng. The statistical meaning of kurtosis and its new application to identification of persons based on seismic signals. *Sensors*, 8(8):5106–5119, Aug. 2008.

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

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

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

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

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

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

[33] 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.

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

[35] WeightedStDev (weighted standard deviation of a sample). MicroStrategy, accessed 13 Jun. 2019. https://doc-archives.microstrategy.com/producthelp/10.10/FunctionsRef/Content/FuncRef/WeightedStDev__weighted_standard_deviation_of_a_sa.htm.

[36] 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.

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

[38] 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.

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

A Examples

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

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

std::array<PRODUCT,5> A = { {{5.2f, 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::cout << "mean = " << std::mean(std::execution::par, A_);
std::cout << "\nvariance = " << std::variance(A_, 0);
std::cout << "\nstandard deviation = " << std::weighted_sample_standard_deviation(
  A_, std::array<float,5>{ 0.2f, 0.2f, 0.1f, 0.25f, 0.25f });
```

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

```
std::vector<double> v = { 2.0, 3.0, 5.0, 7.0, 11.0, 13.0, 17.0, 19.0 };
std::vector<double> v_wgts = { 0.2, 0.1, 0.3, 0.05, 0.05, 0.05, 0.1, 0.15 };

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

The following example showcases the use of **mean** accumulator objects.

```
std::mean_accumulator<int> m;
std::geometric_mean_accumulator<int> gm;
std::harmonic_mean_accumulator<int> hm;

std::stats_accumulate(std::list<int>{ 3, 3, 1, 2, 2, 9 }, m, gm, hm);

std::cout << "mean = " << m.value();
std::cout << "\ngeometric mean = " << gm.value();
std::cout << "\nharmonic mean = " << hm.value();
```

The following example showcases the use of **mean**, **skewness** and **custom** accumulator objects.

```
/* custom accumulator */
class sum_squares_accumulator
{
public:
  constexpr sum_squares_accumulator() noexcept { sum_squares_ = 0; }
  constexpr void operator()(int x) { sum_squares_ += x * x; }
  constexpr int value() const { return sum_squares_; }
private:
  int sum_squares_;
};

// ...

std::list<int> L = { 3, 3, 1, 2, 2, 9 };

std::mean_accumulator<int> m;
std::unadjusted_population_skewness_accumulator<int> sk;
sum_squares_accumulator ssq;

std::stats_accumulate(L, m, sk, ssq);
```

```
std::cout << "mean = " << m.value();
std::cout << "\nskewness = " << sk.value();
std::cout << "\nsum of squares = " << ssq.value();
```