Simple Statistical Functions

Micheal Chiu
Richard Dosselmann
Eric Niebler
Phillip Ratzloff
Vincent Reverdy
Michael Wong

Document Number: P1708R3
Date: February 22, 2021 (Pre-Kona mailing): 10 AM ET
Project: ISO JTC1/SC22/WG21: Programming Language C++
Audience: SG6, SG19, WG21, LEWG
Emails: chiu@cs.toronto.edu,
dosselmr@cs.uregina.ca (corresponding author),
eniebler@fb.com,
phil.ratzloff@sas.com,
vreverdy@illinois.edu,
michael@codeplay.com

Contents

1 Introduction

This document proposes an extension to the C++ library, to support **simple statistical functions**. Such 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]. More such 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.

1.1 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**, **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).

2 Impact on the Standard

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

3 Statistics

Five statistics are defined in this proposal.

3.1 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], 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 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* [21, 22] is defined as

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

The *harmonic mean* [19] (of 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* [23] 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 [24] ought to be considered.

3.2 Quantile

A *quantile* [25] (of **sorted** values) is defined as the value x_i that divides the values $x_1, x_2, \ldots, x_n$ ($n \geq 1$) into intervals of **equal** size. As an example, the $q = 0.5$th ($0 \leq q \leq 1$) quantile, or *median* [19], is the value that divides the values into two intervals, specifically halves, of equal size. It is therefore the "middle" value. Both **even** and **odd** n [26] must be considered in the case of discrete data. A quantile can be found (without sorting) in **linear** time using the **quickselect** [27] algorithm, an algorithm that has a **quadratic** worst case run-time complexity. The **introselect** [28] algorithm is another approach, one having a **log-linear** run-time complexity. A *weighted quantile* [29, 30] (of **sorted** values) is defined as the first value x_i for which $w_1 + w_2 + \ldots + w_i \geq q$. A weighted quantile (of **sorted** values) can be found in **linear** time.

3.3 Mode

The *mode* [19] is defined as the (perhaps not unique) **most frequent** value. The *weighted mode* [31] is defined as the (perhaps not unique) value with the highest total weight. The mode can be found in **linear** time using a **hashtable**.

3.4 Skewness

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

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

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

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

where σ and s are defined in Section 3.7. The *weighted* **population** *skewness* [32] is defined as

$$\frac{\sum_{i=1}^{n} w_i^{3/2}(x_i - \mu)^3}{n\sigma^3} \tag{9}$$

and the *weighted* **sample** *skewness* [32] is defined as

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

3.5 Kurtosis

The **population** *kurtosis* [32], known also as *excess kurtosis* [33], a measure of the peakedness (or flatness) of the values ($n \geq 4$), is defined as

$$\frac{\sum_{i=1}^{n}(x_i - \mu)^4}{n\sigma^4} - 3 \tag{11}$$

and the **sample** *kurtosis* [32] is defined as

$$\frac{n(n+1)\sum_{i=1}^{n}(x_i - \bar{x})^4}{(n-1)(n-2)(n-3)s^4} - \frac{3(n-1)}{(n-2)(n-3)}. \tag{12}$$

The *weighted* **population** *kurtosis* [32] is defined as

$$\frac{\sum_{i=1}^{n} w_i^2(x_i - \mu)^4}{n\sigma^4} - 3 \tag{13}$$

and the *weighted* **sample** *kurtosis* [32] is defined as

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

An alternate definition [19, 34] of kurtosis omits the factor of 3. Both the skewness and kurtosis can be computed in **linear** time [35].

3.6 Variance

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

$$\frac{1}{n}\sum_{i=1}^{n}(x_i - \mu)^2 \tag{15}$$

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

$$\frac{1}{n-1}\sum_{i=1}^{n}(x_i - \bar{x})^2. \tag{16}$$

The *weighted* **population** *variance* [36] is defined as

$$\frac{\sum_{i=1}^{n} w_i(x_i - \mu^*)^2}{\sum_{i=1}^{n} w_i} \tag{17}$$

and the *weighted* **sample** *variance* [36] is defined as

$$\frac{\sum_{i=1}^{n} w_i(x_i - \mu^*)^2}{\frac{n-1}{n}\sum_{i=1}^{n} w_i}. \tag{18}$$

The variance can be computed in **linear** time [37, 38].

3.7 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* [39] is defined as the square root of the weighted sample variance.

4 Proposal

This document proposes the inclusion of the statistics **mean** (arithmetic, geometric and harmonic), **quantile** (and median), **mode**, **skewness**, **kurtosis**, **variance** and **standard deviation** in the C++ library as both **freestanding** functions and **accumulator** objects. It is further proposed that these items be placed into a (new) header `<stats>`, just as was done with the distributions of `<random>`.

4.1 Freestanding Functions

Each statistic is first given as a **function**. Such functions are useful in cases in which a user wishes to compute only **one** statistic. The proposed forms of these functions are given in the following sections. These functions make use of the **concepts** below.

```
template<typename R, typename P>
concept stats_range = std::ranges::input_range<R> &&
   std::is_arithmetic_v<typename std::projected<
```

```
      std::ranges::iterator_t<R>, P>::value_type>;

template<typename R, typename P1, typename Weights, typename P2>
concept weighted_stats_range = std::stats::stats_range<R, P1> &&
   std::stats::stats_range<Weights, P2>;

template<class T>
concept stats_ordered =
   requires(const std::remove_reference_t<T> &a,
      const std::remove_reference_t<T> &b) {
   { a < b } -> std::convertible_to<bool>;
};
```

4.1.1 Mean

The proposed forms of the mean functions are given as follows.

```
// (1)
template<typename R, typename P = std::identity,
   typename Result = std::conditional<
      std::is_integral_v<typename std::projected<
         std::ranges::iterator_t<R>, P>::value_type>,
      double,
      typename std::projected<
         std::ranges::iterator_t<R>, P>::value_type>::type>
requires std::stats::stats_range<R, P> && std::is_arithmetic_v<Result>
constexpr auto mean(R&& r, P proj = P{});

// (2)
template<typename R, typename P1 = std::identity,
   typename Weights, typename P2 = std::identity,
   typename Result = std::conditional<
      std::is_integral_v<typename std::projected<
         std::ranges::iterator_t<R>, P1>::value_type>,
      double,
      typename std::projected<
         std::ranges::iterator_t<R>, P1>::value_type>::type>
requires std::stats::weighted_stats_range<R, P1, Weights, P2> &&
   std::is_arithmetic_v<Result>
constexpr auto mean(R&& r, Weights&& w, P1 proj1 = P1{}, P2 proj2 = P2{});

// (3)
template<typename ExecutionPolicy,
   typename R, typename P = std::identity,
   typename Result = std::conditional<
      std::is_integral_v<typename std::projected<
         std::ranges::iterator_t<R>, P>::value_type>,
      double,
      typename std::projected<
         std::ranges::iterator_t<R>, P>::value_type>::type>
```

```
requires std::is_execution_policy_v<std::decay_t<ExecutionPolicy>> &&
   std::stats::stats_range<R, P> &&
   std::is_arithmetic_v<Result>
constexpr auto mean(ExecutionPolicy&& policy, R&& r, P proj = P{});

// (4)
template<typename ExecutionPolicy,
   typename R, typename P1 = std::identity,
   typename Weights, typename P2 = std::identity,
   typename Result = std::conditional<
      std::is_integral_v<typename std::projected<
         std::ranges::iterator_t<R>, P1>::value_type>,
      double,
      typename std::projected<
         std::ranges::iterator_t<R>, P1>::value_type>::type>
requires std::is_execution_policy_v<std::decay_t<ExecutionPolicy>> &&
   std::stats::weighted_stats_range<R, P1, Weights, P2> &&
   std::is_arithmetic_v<Result>
constexpr auto mean(ExecutionPolicy&& policy,
   R&& r,
   Weights&& w,
   P1 proj1 = P1{},
   P2 proj2 = P2{});

// (5)
template<typename R, typename P = std::identity,
   typename Result = std::conditional<
      std::is_integral_v<typename std::projected<
         std::ranges::iterator_t<R>, P>::value_type>,
      double,
      typename std::projected<
         std::ranges::iterator_t<R>, P>::value_type>::type>
requires std::stats::stats_range<R, P> && std::is_arithmetic_v<Result>
constexpr auto geometric_mean(R&& r, P proj = P{});

// (6)
template<typename R, typename P1 = std::identity,
   typename Weights, typename P2 = std::identity,
   typename Result = std::conditional<
      std::is_integral_v<typename std::projected<
         std::ranges::iterator_t<R>, P1>::value_type>,
      double,
      typename std::projected<
         std::ranges::iterator_t<R>, P1>::value_type>::type>
requires std::stats::weighted_stats_range<R, P1, Weights, P2> &&
   std::is_arithmetic_v<Result>
constexpr auto geometric_mean(
   R&& r, Weights&& w, P1 proj1 = P1{}, P2 proj2 = P2{});
```

```
// (7)
template<typename ExecutionPolicy,
   typename R, typename P = std::identity,
   typename Result = std::conditional<
      std::is_integral_v<typename std::projected<
         std::ranges::iterator_t<R>, P>::value_type>,
      double,
      typename std::projected<
         std::ranges::iterator_t<R>, P>::value_type>::type>
requires std::is_execution_policy_v<std::decay_t<ExecutionPolicy>> &&
   std::stats::stats_range<R, P> &&
   std::is_arithmetic_v<Result>
constexpr auto geometric_mean(ExecutionPolicy&& policy, R&& r, P proj = P{});

// (8)
template<typename ExecutionPolicy,
   typename R, typename P1 = std::identity,
   typename Weights, typename P2 = std::identity,
   typename Result = std::conditional<
      std::is_integral_v<typename std::projected<
         std::ranges::iterator_t<R>, P1>::value_type>,
      double,
      typename std::projected<
         std::ranges::iterator_t<R>, P1>::value_type>::type>
requires std::is_execution_policy_v<std::decay_t<ExecutionPolicy>> &&
   std::stats::weighted_stats_range<R, P1, Weights, P2> &&
   std::is_arithmetic_v<Result>
constexpr auto geometric_mean(ExecutionPolicy&& policy,
   R&& r,
   Weights&& w,
   P1 proj1 = P1{},
   P2 proj2 = P2{});

// (9)
template<typename R, typename P = std::identity,
   typename Result = std::conditional<
      std::is_integral_v<typename std::projected<
         std::ranges::iterator_t<R>, P>::value_type>,
      double,
      typename std::projected<
         std::ranges::iterator_t<R>, P>::value_type>::type>
requires std::stats::stats_range<R, P> && std::is_arithmetic_v<Result>
constexpr auto harmonic_mean(R&& r, P proj = P{});

// (10)
template<typename R, typename P1 = std::identity,
   typename Weights, typename P2 = std::identity,
   typename Result = std::conditional<
      std::is_integral_v<typename std::projected<
```

```
         std::ranges::iterator_t<R>, P1>::value_type>,
      double,
      typename std::projected<
         std::ranges::iterator_t<R>, P1>::value_type>::type>
requires std::stats::weighted_stats_range<R, P1, Weights, P2> &&
   std::is_arithmetic_v<Result>
constexpr auto harmonic_mean(
   R&& r, Weights&& w, P1 proj1 = P1{}, P2 proj2 = P2{});

// (11)
template<typename ExecutionPolicy,
   typename R, typename P = std::identity,
   typename Result = std::conditional<
      std::is_integral_v<typename std::projected<
         std::ranges::iterator_t<R>, P>::value_type>,
      double,
      typename std::projected<
         std::ranges::iterator_t<R>, P>::value_type>::type>
requires std::is_execution_policy_v<std::decay_t<ExecutionPolicy>> &&
   std::stats::stats_range<R, P> &&
   std::is_arithmetic_v<Result>
constexpr auto harmonic_mean(ExecutionPolicy&& policy, R&& r, P proj = P{});

// (12)
template<typename ExecutionPolicy,
   typename R, typename P1 = std::identity,
   typename Weights, typename P2 = std::identity,
   typename Result = std::conditional<
      std::is_integral_v<typename std::projected<
         std::ranges::iterator_t<R>, P1>::value_type>,
      double,
      typename std::projected<
         std::ranges::iterator_t<R>, P1>::value_type>::type>
requires std::is_execution_policy_v<std::decay_t<ExecutionPolicy>> &&
   std::stats::weighted_stats_range<R, P1, Weights, P2> &&
   std::is_arithmetic_v<Result>
constexpr auto harmonic_mean(ExecutionPolicy&& policy,
   R&& r,
   Weights&& w,
   P1 proj1 = P1{},
   P2 proj2 = P2{});
```

Parameters

- `r` - the range of the elements to examine
- `proj` - the projection to apply to the elements of `r`
- `w` - the range of the weights to use

- `proj1` - the projection to apply to the elements of `r`
- `proj2` - the projection to apply to the elements of `w`
- `policy` - the execution policy to use

Return Value

If no errors occur, the (weighted) mean of the elements of `r` (using the weights of `w`) is returned.

Complexity

$O(N)$, where `N = std::ranges::distance(R)`.

Error Handling

- If `r` or `w` is empty (or the sum of `w` is 0) or `r` and `w` are of different sizes,`stats_error` is thrown.
- In the case of `geometric_mean`, if any element of `r` is negative or an element, along with its associated weight, is 0, `stats_error` is thrown.
- In the case of `harmonic_mean`, if any element of `r` is 0, `stats_error` is thrown.

Example

```
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} } };

std::cout << "mean 1 = " << std::stats::mean(A, &PRODUCT::price);

std::vector<int> v = { 2, 3, 5, 7, 11, 13, 17, 19 };
std::vector<double> v_wgts = { 0.2, 0.1, 0.3, 0.05, 0.05, 0.05, 0.1, 0.15 };

std::cout << "\nmean 2 = " << std::stats::geometric_mean(v);
std::cout << "\nmean 3 = " << std::stats::harmonic_mean(v, v_wgts);
```

4.1.2 Quantile

The proposed forms of the quantile (and median) functions are given as follows.

```
// (1)
template<typename R, typename T = double>
requires std::ranges::input_range<R> && std::is_floating_point_v<T>
constexpr auto sorted_quantile(R&& r, T q);

// (2)
template<typename R>
```

```
requires std::ranges::input_range<R>
constexpr auto sorted_median(R&& r);

// (3)
template<typename R,
   typename Weights, typename P = std::identity,
   typename T = double>
requires std::ranges::input_range<R> &&
   std::stats::stats_range<Weights, P> &&
   std::is_floating_point_v<T>
constexpr auto sorted_quantile(R&& r, Weights&& w, T q, P proj = P{});

// (4)
template<typename R, typename Weights, typename P = std::identity>
requires std::ranges::input_range<R> &&
   std::stats::stats_range<Weights, P>
constexpr auto sorted_median(R&& r, Weights&& w, P proj = P{});

// (5)
template<typename ExecutionPolicy, typename R, typename T = double>
requires std::is_execution_policy_v<std::decay_t<ExecutionPolicy>> &&
   std::ranges::input_range<R> &&
   std::is_floating_point_v<T>
constexpr auto sorted_quantile(ExecutionPolicy&& policy, R&& r, T q);

// (6)
template<typename ExecutionPolicy, typename R>
requires std::is_execution_policy_v<std::decay_t<ExecutionPolicy>> &&
   std::ranges::input_range<R>
constexpr auto sorted_median(ExecutionPolicy&& policy, R&& r);

// (7)
template<typename ExecutionPolicy,
   typename R,
   typename Weights, typename P = std::identity,
   typename T = double>
requires std::is_execution_policy_v<std::decay_t<ExecutionPolicy>> &&
   std::ranges::input_range<R> &&
   std::stats::stats_range<Weights, P> &&
   std::is_floating_point_v<T>
constexpr auto sorted_quantile(
   ExecutionPolicy&& policy, R&& r, Weights&& w, T q, P proj = P{});

// (8)
template<typename ExecutionPolicy,
   typename R,
   typename Weights, typename P = std::identity>
requires std::is_execution_policy_v<std::decay_t<ExecutionPolicy>> &&
   std::ranges::input_range<R> &&
```

```
   std::stats::stats_range<Weights, P>
constexpr auto sorted_median(
   ExecutionPolicy&& policy, R&& r, Weights&& w, P proj = P{});

// (9)
template<typename R,
   typename Quantiles, typename P = std::identity,
   typename O>
requires std::ranges::input_range<R> &&
   std::stats::stats_range<Quantiles, P> &&
   std::output_iterator<O, std::tuple<std::iter_value_t<R>,
      std::optional<std::iter_value_t<R>>>>
constexpr auto sorted_quantiles(
   R&& r, Quantiles&& q, O quantiles, P proj = P{});

// (10)
template<typename R,
   typename Weights, typename P1 = std::identity,
   typename Quantiles, typename P2 = std::identity,
   typename O>
requires std::ranges::input_range<R> &&
   std::stats::stats_range<Weights, P1> &&
   std::stats::stats_range<Quantiles, P2> &&
   std::output_iterator<O, std::iter_value_t<R>>
constexpr auto sorted_quantiles(R&& r,
   Weights&& w,
   Quantiles&& q,
   O quantiles,
   P1 proj1 = P1{},
   P2 proj2 = P2{});

// (11)
template<typename ExecutionPolicy,
   typename R,
   typename Quantiles, typename P = std::identity,
   typename O>
requires std::is_execution_policy_v<std::decay_t<ExecutionPolicy>> &&
   std::ranges::input_range<R> &&
   std::stats::stats_range<Quantiles, P> &&
   std::output_iterator<O, std::tuple<std::iter_value_t<R>,
      std::optional<std::iter_value_t<R>>>>
constexpr auto sorted_quantiles(
   ExecutionPolicy&& policy, R&& r, Quantiles&& q, O quantiles, P proj = P{});

// (12)
template<typename ExecutionPolicy,
   typename R,
   typename Weights, typename P1 = std::identity,
   typename Quantiles, typename P2 = std::identity,
```

```
   typename O>
requires std::is_execution_policy_v<std::decay_t<ExecutionPolicy>> &&
   std::ranges::input_range<R> &&
   std::stats::stats_range<Weights, P1> &&
   std::stats::stats_range<Quantiles, P2> &&
   std::output_iterator<O, std::iter_value_t<R>>
constexpr auto sorted_quantiles(ExecutionPolicy&& policy,
   R&& r,
   Weights&& w,
   Quantiles&& q,
   O quantiles,
   P1 proj1 = P1{},
   P2 proj2 = P2{});

// (13)
template<typename R,
   typename C = std::ranges::less,
   typename P = std::identity,
   typename T = double>
requires std::ranges::random_access_range<R> &&
   std::stats::stats_ordered<R> &&
   std::is_arithmetic_v<
      typename std::projected<std::ranges::iterator_t<R>, P>::value_type> &&
   std::is_floating_point_v<T>
constexpr auto unsorted_quantile(R&& r, T q, C pred = C{}, P proj = P{});

// (14)
template<typename R,
   typename C = std::ranges::less,
   typename P = std::identity>
requires std::ranges::random_access_range<R> &&
   std::stats::stats_ordered<R> &&
   std::is_arithmetic_v<
      typename std::projected<std::ranges::iterator_t<R>, P>::value_type>
constexpr auto unsorted_median(R&& r, C pred = C{}, P proj = P{});

// (15)
template<typename ExecutionPolicy,
   typename R,
   typename C = std::ranges::less,
   typename P = std::identity,
   typename T>
requires std::is_execution_policy_v<std::decay_t<ExecutionPolicy>> &&
   std::ranges::random_access_range<R> &&
   std::stats::stats_ordered<R> &&
   std::is_arithmetic_v<
      typename std::projected<std::ranges::iterator_t<R>, P>::value_type> &&
   std::is_floating_point_v<T>
constexpr auto unsorted_quantile(
```

```
   ExecutionPolicy&& policy, R&& r, T q, C pred = C{}, P proj = P{});

// (16)
template<typename ExecutionPolicy,
   typename R,
   typename C = std::ranges::less,
   typename P = std::identity>
requires std::is_execution_policy_v<std::decay_t<ExecutionPolicy>> &&
   std::ranges::random_access_range<R> &&
   std::stats::stats_ordered<R> &&
   std::is_arithmetic_v<
      typename std::projected<std::ranges::iterator_t<R>, P>::value_type>
constexpr auto unsorted_median(
   ExecutionPolicy&& policy, R&& r, C pred = C{}, P proj = P{});
```

Parameters

- `r` - the range of the elements to examine
- `q` - the quantile or range of quantiles
- `w` - the range of the weights to use
- `proj` - the projection to apply to the elements of `w`
- `policy` - the execution policy to use
- `quantiles` - the beginning of the destination range of quantiles
- `proj1` - the projection to apply to the elements of `r`
- `proj2` - the projection to apply to the elements of `w`
- `pred` - the predicate to apply to the elements of `w`

Return Value

If no errors occur, the specified (weighted) quantile of the elements of `r` is returned.

Complexity

- (1) to (12): Linear in `N = std::ranges::distance(R)` on average.
- (13), (14), (15) and (16): $O(N)$ applications of the projection, and $O(2N \log N)$ swaps, where `N = std::ranges::distance(R)`.

Error Handling

- If `r` is or `w` is empty or `r` and `w` are of different sizes, `stats_error` is thrown.
- If $q < 0$ or $q > 1$ (or any element of the range of quantiles), `stats_error` is thrown.

Example

```
std::vector<int> v = { 2, 3, 5, 7, 11, 13, 17, 19 };
std::vector<double> v_wgts = { 0.2, 0.1, 0.3, 0.05, 0.05, 0.05, 0.1, 0.15 };

std::cout << "median 1 = ";
if (auto [m1, m2] = std::stats::sorted_median(v); m2.has_value())
   std::cout << (m1 + m2.value()) / 2.0;
else
   std::cout << m1;

std::cout << "\nquantile 1 = " << std::stats::sorted_quantile(v, v_wgts, 0.5);

std::cout << "\nquantiles 2 = ";
std::vector<std::tuple<double, std::optional<double>>> quantiles;
std::vector<double> q = { 0.25, 0.75 };
std::stats::sorted_quantiles(v, q, std::back_inserter(quantiles));

for (auto [q1, q2] : quantiles)
{
   if (q2.has_value())
      std::cout << (q1 + q2.value()) / 2.0 << " ";
   else
      std::cout << q1 << " ";
}

std::string text = "throughput";

std::cout << "\nmedian 2 =";
if (auto [m1, m2] = std::stats::unsorted_quantile(text, 0.5); m2.has_value())
   std::cout << m1 << " and " << m2.value();
else
   std::cout << m1;

std::vector<int> w = { 5, 0, -1, 19, 22, 13, -98 };
std::cout << "\nmedian 3 = ";
if (auto [m3, m4] = std::stats::unsorted_median(w); m3.has_value())
   std::cout << (m3 + m4.value()) / 2.0;
else
   std::cout << m3;
```

4.1.3 Mode

The proposed forms of the mode functions are given as follows.

```
// (1)
template<typename R,
   typename C = std::ranges::equal_to,
   typename P = std::identity,
   typename O,
```

```
   typename Key = std::iter_value_t<R>,
   typename T = size_t,
   typename Hash = std::hash<Key>,
   typename KeyEqual = std::equal_to<Key>,
   typename Allocator = std::allocator<std::pair<const Key, T>>>
requires std::stats::stats_range<R, P> &&
   std::equality_comparable<std::iter_value_t<R>> &&
   std::output_iterator<O, std::iter_value_t<R>>
constexpr auto mode(R&& r, O modes, C pred = C{}, P proj = P{});

// (2)
template<typename R,
   typename C1 = std::ranges::equal_to,
   typename P1 = std::identity,
   typename Weights,
   typename C2 = std::ranges::equal_to,
   typename P2 = std::identity,
   typename O,
   typename Key = std::iter_value_t<R>,
   typename T = std::iter_value_t<Weights>,
   typename Hash = std::hash<Key>,
   typename KeyEqual = std::equal_to<Key>,
   typename Allocator = std::allocator<std::pair<const Key, T>>>
requires std::stats::weighted_stats_range<R, P1, Weights, P2> &&
   std::equality_comparable<std::iter_value_t<R>> &&
   std::equality_comparable<std::iter_value_t<Weights>> &&
   std::output_iterator<O, std::iter_value_t<R>>
constexpr auto mode(R&& r,
   Weights&& w,
   O modes,
   C1 pred1 = C1{},
   C2 pred2 = C2{},
   P1 proj1 = P1{},
   P2 proj2 = P2{});

// (3)
template<typename ExecutionPolicy,
   typename R,
   typename C = std::ranges::equal_to,
   typename P = std::identity,
   typename O,
   typename Key = std::iter_value_t<R>,
   typename T = size_t,
   typename Hash = std::hash<Key>,
   typename KeyEqual = std::equal_to<Key>,
   typename Allocator = std::allocator<std::pair<const Key, T>>>
requires std::is_execution_policy_v<std::decay_t<ExecutionPolicy>> &&
   std::stats::stats_range<R, P> &&
   std::equality_comparable<std::iter_value_t<R>> &&
```

```
   std::output_iterator<O, std::iter_value_t<R>>
constexpr auto mode(
   ExecutionPolicy&& policy, R&& r, O modes, C pred = C{}, P proj = P{});

// (4)
template<typename ExecutionPolicy,
   typename R,
   typename C1 = std::ranges::equal_to,
   typename P1 = std::identity,
   typename Weights,
   typename C2 = std::ranges::equal_to,
   typename P2 = std::identity,
   typename O,
   typename Key = std::iter_value_t<R>,
   typename T = size_t,
   typename Hash = std::hash<Key>,
   typename KeyEqual = std::equal_to<Key>,
   typename Allocator = std::allocator<std::pair<const Key, T>>>
requires std::is_execution_policy_v<std::decay_t<ExecutionPolicy>> &&
   std::stats::weighted_stats_range<R, P1, Weights, P2> &&
   std::equality_comparable<std::iter_value_t<R>> &&
   std::equality_comparable<std::iter_value_t<Weights>> &&
   std::output_iterator<O, std::iter_value_t<R>>
constexpr auto mode(ExecutionPolicy&& policy,
   R&& r,
   Weights&& w,
   O modes,
   C1 pred1 = C1{},
   C2 pred2 = C2{},
   P1 proj1 = P1{},
   P2 proj2 = P2{});
```

Parameters

- `r` - the range of the elements to examine
- `modes` - the beginning of the destination range of modes
- `pred` - the predicate to apply to the elements of `r`
- `proj` - the projection to apply to the elements of `r`
- `w` - the range of the weights to use
- `pred1` - the predicate to apply to the elements of `r`
- `pred2` - the predicate to apply to the elements of `w`
- `proj1` - the projection to apply to the elements of `r`
- `proj2` - the projection to apply to the elements of `w`
- `policy` - the execution policy to use

Return Value

If no errors occur, the (weighted) mode(s) of the elements of `r` is (are) returned.

Complexity

$O(N)$ applications of the projection and at most $O(2N)$ scans of the hashtable, where `N = std::ranges::distance(R)`.

Error Handling

If `r` or `w` is empty or `r` and `w` are of different sizes, `stats_error` is thrown.

Example

```
std::vector<int> v = { 2, 3, 5, 7, 11, 13, 17, 19 };

std::vector<int> modes;
std::stats::mode(std::execution::par, v, std::back_inserter(modes));
std::cout << "mode(s) 1 = ";
for (const auto& m : modes)
   std::cout << m << " ";

std::string s1("throughput"), s2;

std::stats::mode(s1, std::back_inserter(s2));
std::cout << "\nmode(s) 2 = " << s2;
```

4.1.4 Skewness

The proposed forms of the skewness functions are given as follows. These functions (and those of kurtosis, variance and standard deviation) make use of the enumerated values below.

```
enum data_t { population, sample };
```

```
// (1)
template<typename R, typename P = std::identity,
   typename Result = std::conditional<
      std::is_integral_v<typename std::projected<
         std::ranges::iterator_t<R>, P>::value_type>,
      double,
      typename std::projected<
         std::ranges::iterator_t<R>, P>::value_type>::type>
requires std::stats::stats_range<R, P> && std::is_arithmetic_v<Result>
constexpr auto skewness(R&& r, data_t d, P proj = P{});

// (2)
template<typename R, typename P = std::identity,
   typename T1, typename T2,
   typename Result = std::conditional<
```

```
      std::is_integral_v<typename std::projected<
         std::ranges::iterator_t<R>, P>::value_type>,
      double,
      typename std::projected<
         std::ranges::iterator_t<R>, P>::value_type>::type>
requires std::stats::stats_range<R, P> &&
   std::is_arithmetic_v<T1> && std::is_arithmetic_v<T2> &&
   std::is_arithmetic_v<Result>
constexpr auto skewness(R&& r, T1 m, T2 s, data_t d, P proj = P{});

// (3)
template<typename R, typename P1 = std::identity,
   typename Weights, typename P2 = std::identity,
   typename Result = std::conditional<
      std::is_integral_v<typename std::projected<
         std::ranges::iterator_t<R>, P1>::value_type>,
      double,
      typename std::projected<
         std::ranges::iterator_t<R>, P1>::value_type>::type>
requires std::stats::weighted_stats_range<R, P1, Weights, P2> &&
   std::is_arithmetic_v<Result>
constexpr auto skewness(
   R&& r, Weights&& w, data_t d, P1 proj1 = P1{}, P2 proj2 = P2{});

// (4)
template<typename R, typename P1 = std::identity,
   typename Weights, typename P2 = std::identity,
   typename T1, typename T2,
   typename Result = std::conditional<
      std::is_integral_v<typename std::projected<
         std::ranges::iterator_t<R>, P1>::value_type>,
      double,
      typename std::projected<
         std::ranges::iterator_t<R>, P1>::value_type>::type>
requires std::stats::weighted_stats_range<R, P1, Weights, P2> &&
   std::is_arithmetic_v<T1> && std::is_arithmetic_v<T2> &&
   std::is_arithmetic_v<Result>
constexpr auto skewness(R&& r,
   Weights&& w,
   T1 m,
   T2 s,
   data_t d,
   P1 proj1 = P1{},
   P2 proj2 = P2{});

// (5)
template<typename ExecutionPolicy,
   typename R, typename P = std::identity,
   typename Result = std::conditional<
```

```
      std::is_integral_v<typename std::projected<
         std::ranges::iterator_t<R>, P>::value_type>,
      double,
      typename std::projected<
         std::ranges::iterator_t<R>, P>::value_type>::type>
requires std::is_execution_policy_v<std::decay_t<ExecutionPolicy>> &&
   std::stats::stats_range<R, P> &&
   std::is_arithmetic_v<Result>
constexpr auto skewness(
   ExecutionPolicy&& policy, R&& r, data_t d, P proj = P{});

// (6)
template<typename ExecutionPolicy,
   typename R, typename P = std::identity,
   typename T1, typename T2,
   typename Result = std::conditional<
      std::is_integral_v<typename std::projected<
         std::ranges::iterator_t<R>, P>::value_type>,
      double,
      typename std::projected<
         std::ranges::iterator_t<R>, P>::value_type>::type>
requires std::is_execution_policy_v<std::decay_t<ExecutionPolicy>> &&
   std::stats::stats_range<R, P> &&
   std::is_arithmetic_v<T1> && std::is_arithmetic_v<T2> &&
   std::is_arithmetic_v<Result>
constexpr auto skewness(
   ExecutionPolicy&& policy, R&& r, T1 m, T2 s, data_t d, P proj = P{});

// (7)
template<typename ExecutionPolicy,
   typename R, typename P1 = std::identity,
   typename Weights, typename P2 = std::identity,
   typename Result = std::conditional<
      std::is_integral_v<typename std::projected<
         std::ranges::iterator_t<R>, P1>::value_type>,
      double,
      typename std::projected<
         std::ranges::iterator_t<R>, P1>::value_type>::type>
requires std::is_execution_policy_v<std::decay_t<ExecutionPolicy>> &&
   std::stats::weighted_stats_range<R, P1, Weights, P2> &&
   std::is_arithmetic_v<Result>
constexpr auto skewness(ExecutionPolicy&& policy,
   R&& r,
   Weights&& w,
   data_t d,
   P1 proj1 = P1{},
   P2 proj2 = P2{});

// (8)
```

```
template<typename ExecutionPolicy,
   typename R, typename P1 = std::identity,
   typename Weights, typename P2 = std::identity,
   typename T1, typename T2,
   typename Result = std::conditional<
      std::is_integral_v<typename std::projected<
         std::ranges::iterator_t<R>, P1>::value_type>,
      double,
      typename std::projected<
         std::ranges::iterator_t<R>, P1>::value_type>::type>
requires std::is_execution_policy_v<std::decay_t<ExecutionPolicy>> &&
   std::stats::weighted_stats_range<R, P1, Weights, P2> &&
   std::is_arithmetic_v<T1> && std::is_arithmetic_v<T2> &&
   std::is_arithmetic_v<Result>
constexpr auto skewness(ExecutionPolicy&& policy,
   R&& r,
   Weights&& w,
   T1 m, T2 s,
   data_t d,
   P1 proj1 = P1{},
   P2 proj2 = P2{});
```

Parameters

- `r` - the range of the elements to examine
- `d` - the type of data represented by the elements of `r`, either `population` or `sample`
- `proj` - the projection to apply to the elements of `r`
- `m` - the (precomputed) mean of the elements of `r`
- `s` - the (precomputed) standard deviation of the elements of `r`
- `w` - the range of the weights to use
- `proj1` - the projection to apply to the elements of `r`
- `proj2` - the projection to apply to the elements of `w`
- `policy` - the execution policy to use

Return Value

If no errors occur, the (weighted) skewness of the elements of `r` is returned.

Complexity

$O(N)$, where `N = std::ranges::distance(R)`.

Error Handling

If the size of `r` or `w` is less than 3 or `r` and `w` are of different sizes, `stats_error` is thrown.

Example

```
std::vector<int> v = { 2, 3, 5, 7, 11, 13, 17, 19 };

std::cout << "skewness = " << std::stats::skewness(v, std::stats::population);
```

4.1.5 Kurtosis

The proposed forms of the kurtosis functions are given as follows.

```
// (1)
template<typename R, typename P = std::identity,
   typename Result = std::conditional<
      std::is_integral_v<typename std::projected<
         std::ranges::iterator_t<R>, P>::value_type>,
      double,
      typename std::projected<
         std::ranges::iterator_t<R>, P>::value_type>::type>
requires std::stats::stats_range<R, P> && std::is_arithmetic_v<Result>
constexpr auto kurtosis(R&& r, data_t d, bool excess = true, P proj = P{});

// (2)
template<typename R, typename P = std::identity,
   typename T1, typename T2,
   typename Result = std::conditional<
      std::is_integral_v<typename std::projected<
         std::ranges::iterator_t<R>, P>::value_type>,
      double,
      typename std::projected<
         std::ranges::iterator_t<R>, P>::value_type>::type>
requires std::stats::stats_range<R, P> &&
   std::is_arithmetic_v<T1> && std::is_arithmetic_v<T2> &&
   std::is_arithmetic_v<Result>
constexpr auto kurtosis(
   R&& r, T1 m, T2 s, data_t d, bool excess = true, P proj = P{});

// (3)
template<typename R, typename P1 = std::identity,
   typename Weights, typename P2 = std::identity,
   typename Result = std::conditional<
      std::is_integral_v<typename std::projected<
         std::ranges::iterator_t<R>, P1>::value_type>,
      double,
      typename std::projected<
         std::ranges::iterator_t<R>, P1>::value_type>::type>
requires std::stats::weighted_stats_range<R, P1, Weights, P2> &&
   std::is_arithmetic_v<Result>
constexpr auto kurtosis(R&& r,
   Weights&& w,
   data_t d,
```

```
   bool excess = true,
   P1 proj1 = P1{},
   P2 proj2 = P2{});

// (4)
template<typename R, typename P1 = std::identity,
   typename Weights, typename P2 = std::identity,
   typename T1, typename T2,
   typename Result = std::conditional<
      std::is_integral_v<
         typename std::projected<
            std::ranges::iterator_t<R>, P1>::value_type>,
      double,
      typename std::projected<
         std::ranges::iterator_t<R>, P1>::value_type>::type>
requires std::stats::weighted_stats_range<R, P1, Weights, P2> &&
   std::is_arithmetic_v<T1> && std::is_arithmetic_v<T2> &&
   std::is_arithmetic_v<Result>
constexpr auto kurtosis(R&& r,
   Weights&& w,
   T1 m,
   T2 s,
   data_t d,
   bool excess = true,
   P1 proj1 = P1{},
   P2 proj2 = P2{});

// (5)
template<typename ExecutionPolicy,
   typename R, typename P = std::identity,
   typename Result = std::conditional<
      std::is_integral_v<typename std::projected<
         std::ranges::iterator_t<R>, P>::value_type>,
      double,
      typename std::projected<
         std::ranges::iterator_t<R>, P>::value_type>::type>
requires std::is_execution_policy_v<std::decay_t<ExecutionPolicy>> &&
   std::stats::stats_range<R, P> &&
   std::is_arithmetic_v<Result>
constexpr auto kurtosis(ExecutionPolicy&& policy,
   R&& r,
   data_t d,
   bool excess = true,
   P proj = P{});

// (6)
template<typename ExecutionPolicy,
   typename R, typename P = std::identity,
   typename T1, typename T2,
```

```
   typename Result = std::conditional<
      std::is_integral_v<typename std::projected<
         std::ranges::iterator_t<R>, P>::value_type>,
      double,
      typename std::projected<
         std::ranges::iterator_t<R>, P>::value_type>::type>
requires std::is_execution_policy_v<std::decay_t<ExecutionPolicy>> &&
   std::stats::stats_range<R, P> &&
   std::is_arithmetic_v<T1> && std::is_arithmetic_v<T2> &&
   std::is_arithmetic_v<Result>
constexpr auto kurtosis(ExecutionPolicy&& policy,
   R&& r,
   T1 m,
   T2 s,
   data_t d,
   bool excess = true,
   P proj = P{});

// (7)
template<typename ExecutionPolicy,
   typename R, typename P1 = std::identity,
   typename Weights, typename P2 = std::identity,
   typename Result = std::conditional<
      std::is_integral_v<typename std::projected<
         std::ranges::iterator_t<R>, P1>::value_type>,
      double,
      typename std::projected<
         std::ranges::iterator_t<R>, P1>::value_type>::type>
requires std::is_execution_policy_v<std::decay_t<ExecutionPolicy>> &&
   std::stats::weighted_stats_range<R, P1, Weights, P2> &&
   std::is_arithmetic_v<Result>
constexpr auto kurtosis(ExecutionPolicy&& policy,
   R&& r,
   Weights&& w,
   data_t d,
   bool excess = true,
   P1 proj1 = P1{},
   P2 proj2 = P2{});

// (8)
template<typename ExecutionPolicy,
   typename R, typename P1 = std::identity,
   typename Weights, typename P2 = std::identity,
   typename T1, typename T2,
   typename Result = std::conditional<
      std::is_integral_v<typename std::projected<
         std::ranges::iterator_t<R>, P1>::value_type>,
      double,
      typename std::projected<
```

```
         std::ranges::iterator_t<R>, P1>::value_type>::type>
requires std::is_execution_policy_v<std::decay_t<ExecutionPolicy>> &&
   std::stats::weighted_stats_range<R, P1, Weights, P2> &&
   std::is_arithmetic_v<T1> && std::is_arithmetic_v<T2> &&
   std::is_arithmetic_v<Result>
constexpr auto kurtosis(ExecutionPolicy&& policy,
   R&& r,
   Weights&& w,
   T1 m,
   T2 s,
   data_t d,
   bool excess = true,
   P1 proj1 = P1{},
   P2 proj2 = P2{});
```

Parameters

- `r` - the range of the elements to examine
- `d` - the type of data represented by the elements of `r`, either `population` or `sample`
- `excess` - the type of kurtosis, either excess (`true`) or non-excess (`false`)
- `proj` - the projection to apply to the elements of `r`
- `m` - the (precomputed) mean of the elements of `r`
- `s` - the (precomputed) standard deviation of the elements of `r`
- `w` - the range of the weights to use
- `proj1` - the projection to apply to the elements of `r`
- `proj2` - the projection to apply to the elements of `w`
- `policy` - the execution policy to use

Return Value

If no errors occur, the (weighted) kurtosis of the elements of `r` is returned.

Complexity

$O(N)$, where `N = std::ranges::distance(R)`.

Error Handling

If the size of `r` or `w` is less than 4 or `r` and `w` are of different sizes, `stats_error` is thrown.

Example

```
std::vector<int> v = { 2, 3, 5, 7, 11, 13, 17, 19 };
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::stats::kurtosis(v, v_wgts, std::stats::sample);
```

4.1.6 Variance

The proposed forms of the variance functions are given as follows.

```
// (1)
template<typename R, typename P = std::identity,
   typename Result = std::conditional<
      std::is_integral_v<typename std::projected<
         std::ranges::iterator_t<R>, P>::value_type>,
      double,
      typename std::projected<
         std::ranges::iterator_t<R>, P>::value_type>::type>
requires std::stats::stats_range<R, P> && std::is_arithmetic_v<Result>
constexpr auto var(R&& r, data_t d, P proj = P{});

// (2)
template<typename R, typename P = std::identity,
   typename T,
   typename Result = std::conditional<
      std::is_integral_v<typename std::projected<
         std::ranges::iterator_t<R>, P>::value_type>,
      double,
      typename std::projected<
         std::ranges::iterator_t<R>, P>::value_type>::type>
requires std::stats::stats_range<R, P> &&
   std::is_arithmetic_v<T> &&
   std::is_arithmetic_v<Result>
constexpr auto var(R&& r, T m, data_t d, P proj = P{});

// (3)
template<typename R, typename P1 = std::identity,
   typename Weights, typename P2 = std::identity,
   typename Result = std::conditional<
      std::is_integral_v<typename std::projected<
         std::ranges::iterator_t<R>, P1>::value_type>,
      double,
      typename std::projected<
         std::ranges::iterator_t<R>, P1>::value_type>::type>
requires std::stats::weighted_stats_range<R, P1, Weights, P2> &&
   std::is_arithmetic_v<Result>
constexpr auto var(
   R&& r, Weights&& w, data_t d, P1 proj1 = P1{}, P2 proj2 = P2{});
```

```
// (4)
template<typename R, typename P1 = std::identity,
   typename Weights, typename P2 = std::identity,
   typename T,
   typename Result = std::conditional<
      std::is_integral_v<typename std::projected<
         std::ranges::iterator_t<R>, P1>::value_type>,
      double,
      typename std::projected<
         std::ranges::iterator_t<R>, P1>::value_type>::type>
requires std::stats::weighted_stats_range<R, P1, Weights, P2> &&
   std::is_arithmetic_v<T> &&
   std::is_arithmetic_v<Result>
constexpr auto var(
   R&& r, Weights&& w, T m, data_t d, P1 proj1 = P1{}, P2 proj2 = P2{});

// (5)
template<typename ExecutionPolicy,
   typename R, typename P = std::identity,
   typename Result = std::conditional<
      std::is_integral_v<typename std::projected<
         std::ranges::iterator_t<R>, P>::value_type>,
      double,
      typename std::projected<
         std::ranges::iterator_t<R>, P>::value_type>::type>
requires std::is_execution_policy_v<std::decay_t<ExecutionPolicy>> &&
   std::stats::stats_range<R, P> &&
   std::is_arithmetic_v<Result>
constexpr auto var(ExecutionPolicy&& policy, R&& r, data_t d, P proj = P{});

// (6)
template<typename ExecutionPolicy,
   typename R, typename P = std::identity,
   typename T,
   typename Result = std::conditional<
      std::is_integral_v<typename std::projected<
         std::ranges::iterator_t<R>, P>::value_type>,
      double,
      typename std::projected<
         std::ranges::iterator_t<R>, P>::value_type>::type>
requires std::is_execution_policy_v<std::decay_t<ExecutionPolicy>> &&
   std::stats::stats_range<R, P> &&
   std::is_arithmetic_v<T> &&
   std::is_arithmetic_v<Result>
constexpr auto var(
   ExecutionPolicy&& policy, R&& r, T m, data_t d, P proj = P{});

// (7)
```

```
template<typename ExecutionPolicy,
   typename R, typename P1 = std::identity,
   typename Weights, typename P2 = std::identity,
   typename Result = std::conditional<
      std::is_integral_v<typename std::projected<
         std::ranges::iterator_t<R>, P1>::value_type>,
      double,
      typename std::projected<
         std::ranges::iterator_t<R>, P1>::value_type>::type>
requires std::is_execution_policy_v<std::decay_t<ExecutionPolicy>> &&
   std::stats::weighted_stats_range<R, P1, Weights, P2> &&
   std::is_arithmetic_v<Result>
constexpr auto var(ExecutionPolicy&& policy,
   R&& r,
   Weights&& w,
   data_t d,
   P1 proj1 = P1{},
   P2 proj2 = P2{});

// (8)
template<typename ExecutionPolicy,
   typename R, typename P1 = std::identity,
   typename Weights, typename P2 = std::identity,
   typename T,
   typename Result = std::conditional<
      std::is_integral_v<typename std::projected<
         std::ranges::iterator_t<R>, P1>::value_type>,
      double,
      typename std::projected<
         std::ranges::iterator_t<R>, P1>::value_type>::type>
requires std::is_execution_policy_v<std::decay_t<ExecutionPolicy>> &&
   std::stats::weighted_stats_range<R, P1, Weights, P2> &&
   std::is_arithmetic_v<T> &&
   std::is_arithmetic_v<Result>
constexpr auto var(ExecutionPolicy&& policy,
   R&& r,
   Weights&& w,
   T m,
   data_t d,
   P1 proj1 = P1{},
   P2 proj2 = P2{});
```

Parameters

- `r` - the range of the elements to examine
- `d` - the type of data represented by the elements of `r`, either `population` or `sample`
- `proj` - the projection to apply to the elements of `r`

- `m` - the (precomputed) mean of the elements of `r`
- `w` - the range of the weights to use
- `proj1` - the projection to apply to the elements of `r`
- `proj2` - the projection to apply to the elements of `w`
- `policy` - the execution policy to use

Return Value

If no errors occur, the (weighted) variance of the elements of `r` (using the weights of `w`) is returned.

Complexity

$O(N)$, where `N = std::ranges::distance(R)`.

Error Handling

If the size of `r` or `w` is less than 1 (`population`) or 2 (`sample`) (or the sum of `w` is 0) or `r` and `w` are of different sizes, `stats_error` is thrown.

Example

```
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}} };

std::cout << "variance 1 = "
   << std::stats::var(A, std::stats::population, &PRODUCT::price);
std::cout << "\nvariance 2 = "
   << std::stats::var(A, std::stats::sample, &PRODUCT::price);
```

4.1.7 Standard Deviation

The proposed forms of the standard deviations functions are given as follows.

```
// (1)
template<typename R, typename P = std::identity,
   typename Result = std::conditional<
      std::is_integral_v<typename std::projected<
         std::ranges::iterator_t<R>, P>::value_type>,
      double,
      typename std::projected<
         std::ranges::iterator_t<R>, P>::value_type>::type>
requires std::stats::stats_range<R, P> && std::is_arithmetic_v<Result>
constexpr auto stddev(R&& r, data_t d, P proj = P{});
```

```
// (2)
template<typename R, typename P = std::identity,
   typename T,
   typename Result = std::conditional<
      std::is_integral_v<typename std::projected<
         std::ranges::iterator_t<R>, P>::value_type>,
      double,
      typename std::projected<
         std::ranges::iterator_t<R>, P>::value_type>::type>
requires std::stats::stats_range<R, P> &&
   std::is_arithmetic_v<T> &&
   std::is_arithmetic_v<Result>
constexpr auto stddev(R&& r, T m, data_t d, P proj = P{});

// (3)
template<typename R, typename P1 = std::identity,
   typename Weights, typename P2 = std::identity,
   typename Result = std::conditional<
      std::is_integral_v<typename std::projected<
         std::ranges::iterator_t<R>, P1>::value_type>,
      double,
      typename std::projected<
         std::ranges::iterator_t<R>, P1>::value_type>::type>
requires std::stats::weighted_stats_range<R, P1, Weights, P2> &&
   std::is_arithmetic_v<Result>
constexpr auto stddev(
   R&& r, Weights&& w, data_t d, P1 proj1 = P1{}, P2 proj2 = P2{});

// (4)
template<typename R, typename P1 = std::identity,
   typename Weights, typename P2 = std::identity,
   typename T,
   typename Result = std::conditional<
      std::is_integral_v<typename std::projected<
         std::ranges::iterator_t<R>, P1>::value_type>,
      double,
      typename std::projected<
         std::ranges::iterator_t<R>, P1>::value_type>::type>
requires std::stats::weighted_stats_range<R, P1, Weights, P2> &&
   std::is_arithmetic_v<T> &&
   std::is_arithmetic_v<Result>
constexpr auto stddev(
   R&& r, Weights&& w, T m, data_t d, P1 proj1 = P1{}, P2 proj2 = P2{});

// (5)
template<typename ExecutionPolicy,
   typename R, typename P = std::identity,
   typename Result = std::conditional<
```

```
      std::is_integral_v<typename std::projected<
         std::ranges::iterator_t<R>, P>::value_type>,
      double,
      typename std::projected<
         std::ranges::iterator_t<R>, P>::value_type>::type>
requires std::is_execution_policy_v<std::decay_t<ExecutionPolicy>> &&
   std::stats::stats_range<R, P> &&
   std::is_arithmetic_v<Result>
constexpr auto stddev(
   ExecutionPolicy&& policy, R&& r, data_t d, P proj = P{});

// (6)
template<typename ExecutionPolicy,
   typename R, typename P = std::identity,
   typename T,
   typename Result = std::conditional<
      std::is_integral_v<typename std::projected<
         std::ranges::iterator_t<R>, P>::value_type>,
      double,
      typename std::projected<
         std::ranges::iterator_t<R>, P>::value_type>::type>
requires std::is_execution_policy_v<std::decay_t<ExecutionPolicy>> &&
   std::stats::stats_range<R, P> &&
   std::is_arithmetic_v<T> &&
   std::is_arithmetic_v<Result>
constexpr auto stddev(
   ExecutionPolicy&& policy, R&& r, T m, data_t d, P proj = P{});

// (7)
template<typename ExecutionPolicy,
   typename R, typename P1 = std::identity,
   typename Weights, typename P2 = std::identity,
   typename Result = std::conditional<
      std::is_integral_v<typename std::projected<
         std::ranges::iterator_t<R>, P1>::value_type>,
      double,
      typename std::projected<
         std::ranges::iterator_t<R>, P1>::value_type>::type>
requires std::is_execution_policy_v<std::decay_t<ExecutionPolicy>> &&
   std::stats::weighted_stats_range<R, P1, Weights, P2> &&
   std::is_arithmetic_v<Result>
constexpr auto stddev(ExecutionPolicy&& policy,
   R&& r,
   Weights&& w,
   data_t d,
   P1 proj1 = P1{},
   P2 proj2 = P2{});

// (8)
```

```
template<typename ExecutionPolicy,
   typename R, typename P1 = std::identity,
   typename Weights, typename P2 = std::identity,
   typename T,
   typename Result = std::conditional<
      std::is_integral_v<typename std::projected<
         std::ranges::iterator_t<R>, P1>::value_type>,
      double,
      typename std::projected<
         std::ranges::iterator_t<R>, P2>::value_type>::type>
requires std::is_execution_policy_v<std::decay_t<ExecutionPolicy>> &&
   std::stats::weighted_stats_range<R, P1, Weights, P2> &&
   std::is_arithmetic_v<T> &&
   std::is_arithmetic_v<Result>
constexpr auto stddev(ExecutionPolicy&& policy,
   R&& r,
   Weights&& w,
   T m,
   data_t d,
   P1 proj1 = P1{},
   P2 proj2 = P2{});
```

Parameters

- `r` - the range of the elements to examine
- `d` - the type of data represented by the elements of `r`, either `population` or `sample`
- `proj` - the projection to apply to the elements of `r`
- `m` - the (precomputed) mean of the elements of `r`
- `w` - the range of the weights to use
- `proj1` - the projection to apply to the elements of `r`
- `proj2` - the projection to apply to the elements of `w`
- `policy` - the execution policy to use

Return Value

If no errors occur, the (weighted) standard deviation of the elements of `r` (using the weights of `w`) is returned.

Complexity

$O(N)$, where `N = std::ranges::distance(R)`.

Error Handling

If the size of `r` or `w` is less than 1 (`population`) or 2 (`sample`) (or the sum of `w` is 0) or `r` and `w` are of different sizes, `stats_error` is thrown.

Example

```
std::vector<int> v = { 2, 3, 5, 7, 11, 13, 17, 19 };
std::vector<double> v_wgts = { 0.2, 0.1, 0.3, 0.05, 0.05, 0.05, 0.1, 0.15 };

std::cout << "standard deviation 1 = "
   << std::stats::stddev(v, std::stats::population);
std::cout << "\nstandard deviation 2 = "
   << std::stats::stddev(v, v_wgts, std::stats::population);
```

4.2 Accumulator Objects

Each statistic is secondly aggregated into a **class**, derived from either `stat_accum` or `stat_accum_weighted`, with the exception of quantile (as discussed in Section 5.2). Such objects are useful in cases in which a user wishes to (efficiently) compute **more than one** statistic, specifically in a **single** pass over the values, an idea borrowed from the Boost Accumulators [40]. These objects are passed to an accumulator function (shown as part of `stat_accum` and `stat_accum_weighted`) that makes a single pass over the values. The proposed forms of these objects (and functions) are given in the following sections.

4.2.1 Accumulator

The proposed forms of the (base) `stat_accum` and `stat_accum_weighted` objects are given as follows.

```
// (1)
template<typename T>
class stat_accum
{
   template<typename R, typename ...Args>
   requires std::ranges::input_range<R>
   friend constexpr void accum(R&& r,
      stat_accum<std::iter_value_t<R>>& stat,
      Args& ... stats);

   template<typename ExecutionPolicy, typename R, typename ...Args>
   requires std::ranges::input_range<R>
   friend constexpr void accum(
      ExecutionPolicy&& policy,
      R&& r,
      stat_accum<std::iter_value_t<R>>& stat,
      Args& ... stats);

public:
   constexpr stat_accum() noexcept; // (1)
   constexpr stat_accum(const stat_accum& other); // (2)
   constexpr stat_accum(stat_accum&& other); // (3)
   constexpr stat_accum& operator=(const stat_accum& other); // (4)
   constexpr stat_accum& operator=(stat_accum&& other); // (5)
   ~stat_accum();
};
```

```
// (2)
template<typename T, typename Weight = double>
class weighted_stat_accum
{
   template<typename R, typename Weights, typename ...Args>
   requires std::ranges::input_range<R> && std::ranges::input_range<Weights>
   friend constexpr void accum(R&& r,
      Weights&& w,
      weighted_stat_accum<std::iter_value_t<R>,
         std::iter_value_t<Weights>>& stat,
      Args& ... stats);

   template<typename ExecutionPolicy,
      typename R,
      typename Weights,
      typename ...Args>
   requires std::ranges::input_range<R> && std::ranges::input_range<Weights>
   friend constexpr void accum(ExecutionPolicy&& policy,
      R&& r,
      Weights&& w,
      weighted_stat_accum<std::iter_value_t<R>,
         std::iter_value_t<Weights>>& stat,
      Args& ... stats);

public:
   constexpr weighted_stat_accum() noexcept; // (1)
   constexpr weighted_stat_accum(const weighted_stat_accum& other); // (2)
   constexpr weighted_stat_accum(weighted_stat_accum&& other); // (3)
   constexpr weighted_stat_accum& operator=(
      const weighted_stat_accum& other); // (4)
   constexpr weighted_stat_accum& operator=(
      weighted_stat_accum&& other); // (5)
   ~weighted_stat_accum();
};
```

Parameters

- `r` - the range of the elements to examine
- `stat` - the first (and perhaps only) statistic to compute over `r`
- `stats` - the remaining (if any) statistic(s) to compute over `r`
- `policy` - the execution policy to use
- `other` - another accumulator object to be used as source to initialize the elements of the accumulator object with
- `w` - the range of the weights to use

4.3 Mean

The proposed forms of the mean accumulator objects are given as follows.

```
// (1)
template<typename T, typename P = std::identity, typename Result = double>
class mean_accum : public stat_accum<T>
{
public:
   constexpr mean_accum(P proj = P{}) noexcept; // (1)
   constexpr mean_accum(const mean_accum& other); // (2)
   constexpr mean_accum(mean_accum&& other); // (3)
   constexpr mean_accum& operator=(const mean_accum& other); // (4)
   constexpr mean_accum& operator=(mean_accum&& other); // (5)
   ~mean_accum();
   constexpr auto value() const noexcept;
};

// (2)
template<typename T, typename P1 = std::identity,
   typename Weight = double, typename P2 = std::identity,
   typename Result = double>
class weighted_mean_accum : public weighted_stat_accum<T, Weight>
{
public:
   constexpr weighted_mean_accum(
      P1 proj1 = P1{}, P2 proj2 = P2{}) noexcept; // (1)
   constexpr weighted_mean_accum(const weighted_mean_accum& other); // (2)
   constexpr weighted_mean_accum(weighted_mean_accum&& other); // (3)
   constexpr weighted_mean_accum& operator=(
      const weighted_mean_accum& other); // (4)
   constexpr weighted_mean_accum& operator=(
      weighted_mean_accum&& other); // (5)
   ~weighted_mean_accum();
   constexpr auto value() const noexcept;
};

// (3)
template<typename T, typename P = std::identity, typename Result = double>
class geometric_mean_accum : public stat_accum<T>
{
public:
   constexpr geometric_mean_accum(P proj = P{}) noexcept; // (1)
   constexpr geometric_mean_accum(const geometric_mean_accum& other); // (2)
   constexpr geometric_mean_accum(geometric_mean_accum&& other); // (3)
   constexpr geometric_mean_accum& operator=(
      const geometric_mean_accum& other); // (4)
   constexpr geometric_mean_accum& operator=(
      geometric_mean_accum&& other); // (5)
   ~geometric_mean_accum();
```

```
   constexpr auto value() const noexcept;
};

// (4)
template<typename T, typename P1 = std::identity,
   typename Weight = double, typename P2 = std::identity,
   typename Result = double>
class weighted_geometric_mean_accum : public weighted_stat_accum<T, Weight>
{
public:
   constexpr weighted_geometric_mean_accum(
      P1 proj1 = P1{}, P2 proj2 = P2{}) noexcept; // (1)
   constexpr weighted_geometric_mean_accum(
      const weighted_geometric_mean_accum& other); // (2)
   constexpr weighted_geometric_mean_accum(
      weighted_geometric_mean_accum&& other); // (3)
   constexpr weighted_geometric_mean_accum& operator=(
      const weighted_geometric_mean_accum& other); // (4)
   constexpr weighted_geometric_mean_accum& operator=(
      weighted_geometric_mean_accum&& other); // (5)
   ~weighted_geometric_mean_accum();
   constexpr auto value() const noexcept;
};

// (5)
template<typename T, typename P = std::identity, typename Result = double>
class harmonic_mean_accum : public stat_accum<T>
{
public:
   constexpr harmonic_mean_accum(P proj = P{}) noexcept; // (1)
   constexpr harmonic_mean_accum(const harmonic_mean_accum& other); // (2)
   constexpr harmonic_mean_accum(harmonic_mean_accum&& other); // (3)
   constexpr harmonic_mean_accum& operator=(
      const harmonic_mean_accum& other); // (4)
   constexpr harmonic_mean_accum& operator=(
      harmonic_mean_accum&& other); // (5)
   ~harmonic_mean_accum();
   constexpr auto value() const noexcept;
};

// (6)
template<typename T, typename P1 = std::identity,
   typename Weight = double, typename P2 = std::identity,
   typename Result = double>
class weighted_harmonic_mean_accum : public weighted_stat_accum<T, Weight>
{
public:
   constexpr weighted_harmonic_mean_accum(
      P1 proj1 = P1{}, P2 proj2 = P2{}) noexcept; // (1)
```

```
   constexpr weighted_harmonic_mean_accum(
      const weighted_harmonic_mean_accum& other); // (2)
   constexpr weighted_harmonic_mean_accum(
      weighted_harmonic_mean_accum&& other); // (3)
   constexpr weighted_harmonic_mean_accum& operator=(
      const weighted_harmonic_mean_accum& other); // (4)
   constexpr weighted_harmonic_mean_accum& operator=(
      weighted_harmonic_mean_accum&& other); // (5)
   ~weighted_harmonic_mean_accum();
   constexpr auto value() const noexcept;
};
```

Parameters

- `proj` - the projection to apply to the elements (of the associated range)
- `other` - another accumulator object to be used as source to initialize the elements of the accumulator object with
- `proj1` - the projection to apply to the elements (of the associated range)
- `proj2` - the projection to apply to the elements (of the associated weights)

Return Value

In the case of `value`, if no errors occur, the (weighted) mean of the elements (of the associated range) is returned.

Error Handling

- If the associated range or weights is empty (or the sum of the weights is 0) or the range and weights are of different sizes, `stats_error` is thrown.
- In the case of `geometric_mean`, if any element of the associated range is negative or an element, along with its associated weight, is 0, `stats_error` is thrown.
- In the case of `harmonic_mean`, if any element of the associated range is 0, `stats_error` is thrown.

Example

```
std::list<int> L = { 3, 3, 1, 2, 2 };
std::stats::mean_accum<int> m1;
std::stats::geometric_mean_accum<int> gm;
std::stats::harmonic_mean_accum<int> hm;

std::stats::accum(L, m1, gm, hm);
std::cout << "mean 1 = " << m1.value();
std::cout << "\nmean 2 = " << gm.value();
std::cout << "\nmean 3 = " << hm.value();

typedef std::tuple<int, double> s_t;
```

```
std::set<s_t> S = {
   std::make_tuple(1, 1.1),
   std::make_tuple(6, 2.6),
   std::make_tuple(8, -0.4),
   std::make_tuple(9, 5.1),
};

auto f = [](const s_t& val) { return std::get<1>(val); };
std::stats::mean_accum<s_t, std::function<double(const s_t&)>> m2(f);
std::stats::stddev_accum<s_t, std::function<double(const s_t&)>>
   s(std::stats::population, f);

std::stats::accum(S, m2, s);
std::cout << "\nmean 4 = " << m2.value();
std::cout << "\nstandard deviation 1 = " << s.value();
```

4.4 Mode

The proposed forms of the mode accumulator objects are given as follows.

```
// (1)
template<typename T,
   typename C = std::ranges::equal_to,
   typename P = std::identity,
   typename Key = T,
   typename U = size_t,
   typename Hash = std::hash<Key>,
   typename KeyEqual = std::equal_to<Key>,
   typename Allocator = std::allocator<std::pair<const Key, U>>>
class mode_accum : public stat_accum<T>
{
public:
   constexpr mode_accum(C pred = C{}, P proj = P{}) noexcept; // (1)
   constexpr mode_accum(const mode_accum& other); // (2)
   constexpr mode_accum(mode_accum&& other); // (3)
   constexpr mode_accum& operator=(const mode_accum& other); // (4)
   constexpr mode_accum& operator=(mode_accum&& other); // (5)
   ~mode_accum();
   template<typename OutputIt> constexpr auto value(OutputIt modes) const;
};

// (2)
template<typename T,
   typename C1 = std::ranges::equal_to,
   typename P1 = std::identity,
   typename Weight = double,
   typename C2 = std::ranges::equal_to,
   typename P2 = std::identity,
   typename Key = T,
   typename U = size_t,
```

```
   typename Hash = std::hash<Key>,
   typename KeyEqual = std::equal_to<Key>,
   typename Allocator = std::allocator<std::pair<const Key, U>>>
class weighted_mode_accum : public weighted_stat_accum<T, Weight>
{
public:
   constexpr weighted_mode_accum(C1 pred1 = C1{}, P1 proj1 = P1{},
      C2 pred2 = C2{}, P2 proj2 = P2{}) noexcept; // (1)
   constexpr weighted_mode_accum(const weighted_mode_accum& other); // (2)
   constexpr weighted_mode_accum(weighted_mode_accum&& other); // (3)
   constexpr weighted_mode_accum& operator=(
      const weighted_mode_accum& other); // (4)
   constexpr weighted_mode_accum& operator=(
      weighted_mode_accum&& other); // (5)
   ~weighted_mode_accum();
   template<typename OutputIt> constexpr void value(OutputIt modes) const;
};
```

Parameters

- `pred` - the predicate to apply to the elements (of the associated range)
- `proj` - the projection to apply to the elements (of the associated range)
- `other` - another accumulator object to be used as source to initialize the elements of the accumulator object with
- `modes` - the beginning of the associated destination range of modes
- `pred1` - the predicate to apply to the elements (of the associated range)
- `proj1` - the projection to apply to the elements (of the associated range)
- `pred2` - the predicate to apply to the elements (of the associated weights)
- `proj2` - the projection to apply to the elements (of the associated weights)

Error Handling

If the associated range or weights is empty (or the sum of the weights is 0), or the range and weights are of different sizes, `stats_error` is thrown.

Example

```
std::list<int> L = { 3, 3, 1, 2, 2 };
std::stats::mean_accum<int> m;
std::stats::mode_accum<int> d;

std::stats::accum(L, m, d);
std::cout << "mean = " << m.value();

std::vector<int> modes;
```

```
d.value(std::back_inserter(modes));
std::cout << "\nmode(s) = ";
for (auto i = modes.cbegin(); i != modes.cend(); i++)
   std::cout << *i << " ";
```

4.5 Skewness

The proposed forms of the skewness accumulator objects are given as follows.

```
// (1)
template<typename T, typename P = std::identity, typename Result = double>
class skewness_accum : public stat_accum<T>
{
public:
   constexpr skewness_accum() noexcept; // (1)
   constexpr skewness_accum(data_t d, P proj = P{}) noexcept; // (2)
   constexpr skewness_accum(const skewness_accum& other); // (3)
   constexpr skewness_accum(skewness_accum&& other); // (4)
   constexpr skewness_accum& operator=(const skewness_accum& other); // (5)
   constexpr skewness_accum& operator=(skewness_accum&& other); // (6)
   ~skewness_accum();
   constexpr auto value() const noexcept;
};

// (2)
template<typename T, typename P1 = std::identity,
   typename Weight = double, typename P2 = std::identity,
   typename Result = double>
class weighted_skewness_accum : public weighted_stat_accum<T, Weight>
{
public:
   constexpr weighted_skewness_accum() noexcept; // (1)
   constexpr weighted_skewness_accum(
      data_t d, P1 proj1 = P1{}, P2 proj2 = P2{}) noexcept; // (2)
   constexpr weighted_skewness_accum(
      const weighted_skewness_accum& other); // (3)
   constexpr weighted_skewness_accum(weighted_skewness_accum&& other); // (4)
   constexpr weighted_skewness_accum& operator=(
      const weighted_skewness_accum& other); // (5)
   constexpr weighted_skewness_accum& operator=(
      weighted_skewness_accum&& other); // (6)
   ~weighted_skewness_accum();
   constexpr auto value() const noexcept;
};
```

Parameters

- `d` - the type of data represented by the elements (of the associated range), either `population` or `sample`
- `proj` - the projection to apply to the elements (of the associated range)

- `other` - another accumulator object to be used as source to initialize the elements of the accumulator object with
- `proj1` - the projection to apply to the elements (of the associated range)
- `proj2` - the projection to apply to the elements (of the associated weights)

Return Value

In the case of `value`, if no errors occur, the (weighted) skewness of the elements (of the associated range) is returned.

Error Handling

If the size of the associated range or weights is less than 3 or the range and weights are of different sizes, `stats_error` is thrown.

Example

```
std::list<int> L = { 3, 3, 1, 2, 2 };
std::stats::mean_accum<int> m;
std::stats::skewness_accum<int> sk(std::stats::population);

std::stats::accum(L, m, sk);
std::cout << "mean = " << m.value();
std::cout << "\nskewness = " << sk.value();
```

4.6 Kurtosis

The proposed forms of the kurtosis accumulator objects are given as follows.

```
// (1)
template<typename T, typename P = std::identity, typename Result = double>
class kurtosis_accum : public stat_accum<T>
{
public:
   constexpr kurtosis_accum() noexcept; // (1)
   constexpr kurtosis_accum(
      data_t d, bool excess = true, P proj = P{}) noexcept; // (2)
   constexpr kurtosis_accum(const kurtosis_accum& other); // (3)
   constexpr kurtosis_accum(kurtosis_accum&& other); // (4)
   constexpr kurtosis_accum& operator=(const kurtosis_accum& other); // (5)
   constexpr kurtosis_accum& operator=(kurtosis_accum&& other); // (6)
   ~kurtosis_accum();
   constexpr auto value() const noexcept;
};

// (2)
template<typename T, typename P1 = std::identity,
   typename Weight = double, typename P2 = std::identity,
   typename Result = double>
class weighted_kurtosis_accum : public weighted_stat_accum<T, Weight>
```

```
{
public:
   constexpr weighted_kurtosis_accum() noexcept; // (1)
   constexpr weighted_kurtosis_accum(data_t d,
      bool excess = true,
      P1 proj1 = P1{},
      P2 proj2 = P2{}) noexcept; // (2)
   constexpr weighted_kurtosis_accum(
      const weighted_kurtosis_accum& other); // (3)
   constexpr weighted_kurtosis_accum(
      weighted_kurtosis_accum&& other); // (4)
   constexpr weighted_kurtosis_accum& operator=(
      const weighted_kurtosis_accum& other); // (5)
   constexpr weighted_kurtosis_accum& operator=(
      weighted_kurtosis_accum&& other); // (6)
   ~weighted_kurtosis_accum();
   constexpr auto value() const noexcept;
};
```

Parameters

- `d` - the type of data represented by the elements (of the associated range), either `population` or `sample`
- `excess` - the type of kurtosis, either excess (`true`) or non-excess (`false`)
- `proj` - the projection to apply to the elements (of the associated range)
- `other` - another accumulator object to be used as source to initialize the elements of the accumulator object with
- `proj1` - the projection to apply to the elements (of the associated range)
- `proj2` - the projection to apply to the elements (of the associated weights)

Return Value

In the case of `value`, if no errors occur, the (weighted) kurtosis of the elements (of the associated range) is returned.

Error Handling

If the size of the associated range or weights is less than 4 or the range and weights are of different sizes, `stats_error` is thrown.

Example

```
std::list<int> L = { 3, 3, 1, 2, 2 };
std::stats::mean_accum<int> m;
std::stats::kurtosis_accum<int> k(std::stats::sample);

std::stats::accum(L, m, k);
std::cout << "mean = " << m.value();
std::cout << "\nkurtosis = " << k.value();
```

4.7 Variance

The proposed forms of the variance accumulator objects are given as follows.

```
// (1)
template<typename T, typename P = std::identity, typename Result = double>
class var_accum : public stat_accum<T>
{
public:
   constexpr var_accum() noexcept; // (1)
   constexpr var_accum(data_t d, P proj = P{}) noexcept; // (2)
   constexpr var_accum(const var_accum& other); // (3)
   constexpr var_accum(var_accum&& other); // (4)
   constexpr var_accum& operator=(const var_accum& other); // (5)
   constexpr var_accum& operator=(var_accum&& other); // (6)
   ~var_accum();
   constexpr auto value() const noexcept;
};

// (2)
template<typename T, typename P1 = std::identity,
   typename Weight = double, typename P2 = std::identity,
   typename Result = double>
class weighted_var_accum : public weighted_stat_accum<T, Weight>
{
public:
   constexpr weighted_var_accum() noexcept; // (1)
   constexpr weighted_var_accum(
      data_t d, P1 proj1 = P1{}, P2 proj2 = P2{}) noexcept; // (2)
   constexpr weighted_var_accum(const weighted_var_accum& other); // (3)
   constexpr weighted_var_accum(weighted_var_accum&& other); // (4)
   constexpr weighted_var_accum& operator=(
      const weighted_var_accum& other); // (5)
   constexpr weighted_var_accum& operator=(weighted_var_accum&& other); // (6)
   ~weighted_var_accum();
   constexpr auto value() const noexcept;
};
```

Parameters

- `d` - the type of data represented by the elements (of the associated range), either `population` or `sample`
- `proj` - the projection to apply to the elements (of the associated range)
- `other` - another accumulator object to be used as source to initialize the elements of the accumulator object with
- `proj1` - the projection to apply to the elements (of the associated range)
- `proj2` - the projection to apply to the elements (of the associated weights)

Return Value

In the case of `value`, if no errors occur, the (weighted) variance of the elements (of the associated range) is returned.

Error Handling

If the size of the associated range or weights is less than 1 (`population`) or 2 (`sample`) (or the sum of the weights is 0) or the range and weights are of different sizes, `stats_error` is thrown.

Example

```
std::list<int> L = { 3, 3, 1, 2, 2 };
std::stats::geometric_mean_accum<int> gm;
std::stats::harmonic_mean_accum<int> hm;
std::stats::var_accum<int> v(std::stats::population);

std::stats::accum(std::execution::par, L, gm, hm, v);
std::cout << "mean 1 = " << gm.value();
std::cout << "\nmean 2 = " << hm.value();
std::cout << "\nvariance = " << v.value();
```

4.8 Standard Deviation

The proposed forms of the standard deviation accumulator objects are given as follows.

```
// (1)
template<typename T, typename P = std::identity, typename Result = double>
class stddev_accum : public var_accum<T, P, Result>
{
public:
   constexpr stddev_accum() noexcept; // (1)
   constexpr stddev_accum(data_t d, P proj = P{}) noexcept; // (2)
   constexpr stddev_accum(const stddev_accum& other); // (3)
   constexpr stddev_accum(stddev_accum&& other); // (4)
   constexpr stddev_accum& operator=(const stddev_accum& other); // (5)
   constexpr stddev_accum& operator=(stddev_accum&& other); // (6)
   ~stddev_accum();
   constexpr auto value() const noexcept;
};

// (2)
template<typename T, typename P1 = std::identity,
   typename Weight = double, typename P2 = std::identity,
   typename Result = double>
class weighted_stddev_accum
   : public weighted_var_accum<T, P1, Weight, P2, Result>
{
public:
   constexpr weighted_stddev_accum() noexcept; // (1)
   constexpr weighted_stddev_accum(
```

```
      data_t d, P1 proj1 = P1{}, P2 proj2 = P2{}) noexcept; // (2)
   constexpr weighted_stddev_accum(const weighted_stddev_accum& other); // (3)
   constexpr weighted_stddev_accum(weighted_stddev_accum&& other); // (4)
   constexpr weighted_stddev_accum& operator=(
      const weighted_stddev_accum& other); // (5)
   constexpr weighted_stddev_accum& operator=(
      weighted_stddev_accum&& other); // (6)
   ~weighted_stddev_accum();
   constexpr auto value() const noexcept;
};
```

Parameters

- `d` - the type of data represented by the elements (of the associated range), either `population` or `sample`
- `proj` - the projection to apply to the elements (of the associated range)
- `other` - another accumulator object to be used as source to initialize the elements of the accumulator object with
- `proj1` - the projection to apply to the elements (of the associated range)
- `proj2` - the projection to apply to the elements (of the associated weights)

Return Value

In the case of `value`, if no errors occur, the (weighted) standard deviation of the elements (of the associated range) is returned.

Error Handling

If the size of the associated range or weights is less than 1 (`population`) or 2 (`sample`) (or the sum of the weights is 0) or the range and weights are of different sizes, `stats_error` is thrown.

Example

```
std::forward_list<int> L = { 3, 3, 1, 2, 2 };
std::vector<double> W = { 0.5, 0.1, 0.2, 0.1, 0.1 };
std::stats::weighted_mean_accum<int> m;
std::stats::weighted_stddev_accum<int> s(std::stats::population);

std::stats::accum(L, W, m, s);
std::cout << "mean = " << m.value();
std::cout << "\nstandard deviation = " << s.value();
```

5 Discussions

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

5.1 Freestanding Functions vs. Accumulator Objects

Perhaps the most significant concern stemming from this proposal is that of free standing functions versus accumulator objects. In the first incarnation of this proposal, namely P1708R0, free standing functions were exclusively proposed. Then, in P1708R1 and P1708R2 an accumulator object was introduced, believing that the increased (run-time) **performance** that it offered would be in the interest of the C++ community. Given that each of these paradigms have merit, with freestanding functions again being most useful in the case of the computation of a single statistic and accumulator objects being more attractive in instances in which multiple statistics are computed, the decision has been made to incorporate **both** such models into this version of the proposal. Users are thus able to choose the approach that best fits with their design rather than being forced to use one of two configurations.

5.2 Quantile Accumulator Object

The process of finding a quantile is quite different from that of the other statistics in that it mandates that the values of a (unsorted) range be **reorganized**. Accordingly, the computation of a quantile, using the functions of Section 4.1.2, requires that a range be of the `range_access_range` variety. Were it to be made into an accumulator object, this object would require that the range over which it is calculated also be of type `range_access_range`, a rather restrictive demand. Additionally, this object would need to allocate **memory** so as to accumulate the values of a range (in a container) for later (partial) sorting and examination. Unless the number of such values is known beforehand, memory may need to be reallocated several times over, an issue that also arises in the case of a sorted range. Given the manner in which this statistic is computed (and the lack of benefits from its use as an accumulator object), it is **only** made available as a freestanding function.

5.3 Weighted Quantile Freestanding Functions

While freestanding functions to compute the weighted quantile in the case of a **sorted** range are provided, **unsorted** versions are not proposed. A weighted quantile requires that a range be (mostly) **sorted**. Given that this proposal emphasizes statistics rather than sorting algorithms, such functions are not proposed.

5.4 Trimmed Mean

The issue of a trimmed mean is raised in [41]. A ($p\%$) *trimmed mean* [42] 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. Like the weighted quantile, 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 [41]) that one might handle this (and other similar) matter via **ranges**, specifically by using a statement of the form

```
auto result = values | std::ranges::sort | trim(p) | std::stats::mean;
```

5.5 Special Values

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

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

6 Acknowledgements

Michael Wong's work is made possible by Codeplay Software Ltd., ISOCPP Foundation, Khronos and the Standards Council of Canada.

References

[1] "statistics - Mathematical statistics functions, Python", *Python*. Web. 14 Apr. 2020
https://docs.python.org/3/library/statistics.html

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

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

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

[5] "Computing with Descriptive Statistic", *MathWorks*. Web. 23 May 2020
https://www.mathworks.com/help/matlab/data_analysis/descriptive-statistics.html

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

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

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

[9] "The SURVEYMEANS Procedure", *sas*. Web. 11 Jun. 2020
https://support.sas.com/documentation/cdl/en/statug/65328/HTML/default/viewer.htm#statug_surveymeans_details06.htm

[10] "Statistical functions", *IBM*. Web. 28 Aug. 2020https://www.ibm.com/support/knowledgecenter/SSLVMB_sub/statistics_refe

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

[12] Wong, Michael et al. "P1415R1: SG19 Machine Learning Layered List", *ISO JTC1/SC22/WG21: Programming Language C++*. Web. 9 Aug. 2020
http://open-std.org/JTC1/SC22/WG21/docs/papers/2019/p1415r1.pdf

[13] Bristow, Paul. "A Proposal to add Mathematical Functions for Statistics to the C++ Standard Library", *JTC 1/SC22/WG14/N1069, WG21/N1668*. Web. 12 Jun. 2020
http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1069.pdf

[14] Brown, Walter E. et al. "Random Number Generation in C++0X: A Comprehensive Proposal, version2", *WG21/N2032 = J16/06/0102*. Web. 13 Jun. 2020
www.open-std.org/jtc1/sc22/wg21/docs/papers/2006/n2032.pdf

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

[16] Agrawal, Nikhar et al. "Chapter 5. Statistical Distributions and Functions", *Boost: C++ Libraries*. Web. 12 Jun. 2020.
https://www.boost.org/doc/libs/1_73_0/libs/math/doc/html/dist.html

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

[18] "GNU Scientific Library", *GNU Operating System*. Web. 13 Jun. 2020
https://www.gnu.org/software/gsl/doc/html/index.html#

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

[20] Saksena, R. S. A Hand Book of Statistics, *Motilal Banarsidass*, 1981.

[21] "Weighted geometric mean", *Wikipedia.com*. Web. 13 Jun. 2020
https://en.wikipedia.org/wiki/Weighted_geometric_mean

[22] Jain, D. R. and Jhunjhunwala, Bharat. Business Statistics, *Tata McGray-Hill*, 2007.

[23] Bishnu, Partha Sarathi and Bhattacherjee, Vandana. DATA Analysis - Using Statistics and Probability with R Language, *PHI Learning Private Limited*, 2018.

[24] McNamee, John Michael. A Comparison Of Methods For Accurate Summation. *ACM SIGSAM Bulletin*, **38**(1), March 2004.

[25] Berthouex, Paul Mac and Brown, Linfield C. Statistics for Environmental Engineers, second edition, *Lewis Publishers*, 2002.

[26] Bernstein, Stephen and Bernstein, Ruth. Schaum's Outline of Elements of Statistics I: Descriptive Statistics and Probability, *McGraw Hill Professional*, 1999.

[27] Hoare, C.A.R. Algorithm 65: find. *Communications of the ACM*, **4**(7), July 1961, pp. 321-322.

[28] Musser, David R. "Introspective Sorting and Selection Algorithms" *Software: Practice and Experience*, **27**(8), August 1997, pp. 983-993.

[29] Wicklin, Richard "Weighted percentiles", *sas*. Web. 28 Jun. 2020
`https://blogs.sas.com/content/iml/2016/08/29/weighted-percentiles.html`

[30] Hubert, Mia et al. (editors). Theory and Applications of Recent Robust Methods. *Springer Basel AG*, 2004.

[31] Cichosz, Paweł. Data Mining Algorithms: Explained Using R. *Wiley*, 2015.

[32] "Appendix 1: SAS Elementary Statistics Procedures", *Worcester Polytechnic Institute*. Web. 28 Aug. 2020
`http://www.math.wpi.edu/saspdf/proc/a01.pdf`

[33] Warner, Rebecca M. Applied Statistics: From Bivariate Through Multivate Techniques, *SAGE Publications*, 2008.

[34] Hinton, Perry R. Statistics Explained, third edition, *Routledge*, 2014.

[35] "Computing skewness and kurtosis in one pass", *John D. Cook Consulting*. Web. 20 Aug. 2020
`https://www.johndcook.com/blog/skewness_kurtosis/`

[36] "Weighted sample variance", *Wikipedia*. Web. 13 Jun. 2020
`https://en.wikipedia.org/wiki/Weighted_arithmetic_mean`

[37] "Algorithms for calculating variance", *Wikipedia*. Web. 19 Oct. 2019
`https://en.wikipedia.org/wiki/Algorithms_for_calculating_variance`

[38] "Algorithms for Calculating Variance", *Project Gutenberg Self Publishing Press*. Web. 23 Aug. 2020
`http://www.self.gutenberg.org/articles/Algorithms_for_calculating_variance`

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

[40] Niebler, Eric. "Chapter 1. Boost.Accumulators", *Boost: C++ Libraries*. Web. 14 Sept. 2019
`https://www.boost.org/doc/libs/1_71_0/doc/html/accumulators.html`

[41] Opara, Jolanta. "P2119R0 Feedback on P1708: Simple Statistical Functions", *JTC1/SC22/WG21*. Web. 14 Apr. 2020
`http://open-std.org/JTC1/SC22/WG21/docs/papers/2020/p2119r0.html`

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