Simple Statistical Functions

Michael Chiu
Guy Davidson
Richard Dosselmann
Oleksandr Koval
Larry Lewis
Johan Lundburg
Jens Maurer
Eric Niebler
Phillip Ratzloff
Vincent Ratzloff
Michael Reverdy
Michael Wong
## Contents

1 Introduction 2
   1.1 Revision History ................................................................. 2

2 Impact on the Standard ......................................................... 3

3 Statistics ................................................................. 3
   3.1 Mean ................................................................................. 3
   3.2 Skewness ........................................................................ 4
   3.3 Kurtosis .......................................................................... 4
   3.4 Variance .......................................................................... 5
   3.5 Standard Deviation ........................................................... 5

4 Proposal ................................................................. 5
   4.1 Freestanding Functions .................................................... 6
      4.1.1 Mean .......................................................................... 6
      4.1.2 Skewness ................................................................... 8
      4.1.3 Kurtosis ...................................................................... 11
      4.1.4 Variance .................................................................... 13
      4.1.5 Standard Deviation .................................................... 16
   4.2 Accumulator Objects ......................................................... 18
      4.2.1 Accumulator ............................................................... 18
   4.3 Mean .................................................................................. 20
   4.4 Skewness .......................................................................... 23
   4.5 Kurtosis ........................................................................... 24
   4.6 Variance ........................................................................... 25
   4.7 Standard Deviation ........................................................... 27

5 Discussions ................................................................. 28
   5.1 Freestanding Functions vs. Accumulator Objects ...................... 28
   5.2 Trimmed Mean ................................................................. 29
   5.3 Special Values .................................................................... 29

6 Acknowledgements ................................................................... 29
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 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 elements 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 functions do not throw exceptions.
- 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.

2 Impact on the Standard

This proposal is a pure library extension.

3 Statistics

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.

3.1 Mean

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

\[
\frac{1}{n} \sum_{i=1}^{n} x_i.
\] (1)

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

\[
\frac{\sum_{i=1}^{n} w_i x_i}{\sum_{i=1}^{n} w_i}.
\] (2)

The geometric mean [19] (of non-negative values \( x_i \geq 0 \)) is defined as

\[
\left( \prod_{i=1}^{n} x_i \right)^{\frac{1}{n}}.
\] (3)

and the weighted geometric mean [20] is defined as

\[
\left( \prod_{i=1}^{n} x_i^{w_i} \right)^{\left( \frac{1}{\sum_{i=1}^{n} w_i} \right)^{-1}}.
\] (4)

The harmonic mean [19] (of positive values \( x_i > 0 \)) is defined as

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

and the weighted harmonic mean [22] is defined as

\[
\frac{\sum_{i=1}^{n} w_i}{\sum_{i=1}^{n} w_i x_i}.
\] (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.
3.2 Skewness

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

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

and the sample skewness [19] is defined as

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

where \(\sigma\) and \(s\) are defined in Section 3.5. The weighted population skewness [21] is defined as

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

and the weighted sample skewness [21] is defined as

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

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

\[
\sqrt{\frac{n(n-1)}{n-2}},
\]  
(11)

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

3.3 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

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

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

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

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

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

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

\[
\frac{(\sum_{i=1}^{n} w_i)^2}{W \sigma^4} \left( \frac{\sum_{i=1}^{n} w_i}{\sum_{i=1}^{n} w_i} \right)^4 - 4 \frac{\sum_{i=1}^{n} w_i \sum_{i=1}^{n} w_i^3}{W} - 3 \frac{(\sum_{i=1}^{n} w_i^2)^2}{W} \sum_{i=1}^{n} w_i (x_i - \mu)^4 - 3 \frac{(\sum_{i=1}^{n} w_i^2)^2}{W} \sum_{i=1}^{n} w_i \sum_{i=1}^{n} w_i^2 + 4 \frac{\sum_{i=1}^{n} w_i \sum_{i=1}^{n} w_i^3}{W} \sum_{i=1}^{n} w_i \sum_{i=1}^{n} w_i^3 - 3 \frac{(\sum_{i=1}^{n} w_i^2)^2}{W},
\]  
(15)

where

\[
W = \left( \frac{n}{\sum_{i=1}^{n} w_i^2} - \frac{n}{\sum_{i=1}^{n} w_i^2} \right) \left( \frac{n}{\sum_{i=1}^{n} w_i^2} - \frac{n}{\sum_{i=1}^{n} w_i^2} \right) \left( \frac{n}{\sum_{i=1}^{n} w_i^2} - \frac{n}{\sum_{i=1}^{n} w_i^2} \right) \left( \frac{n}{\sum_{i=1}^{n} w_i^2} - \frac{n}{\sum_{i=1}^{n} w_i^2} \right).
\]  
(15)

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

\[
\frac{1}{n\sigma^4} \sum_{i=1}^{n} (x_i - \bar{x})^4
\]  
(16)
and the (Pearson) **sample kurtosis** [27] is defined as

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

The **weighted (Pearson) population kurtosis** [27] 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** [27] is defined as

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

### 3.4 Variance

The **population variance** [19] \((n \geq 1)\), denoted \(\sigma^2\), 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

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

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

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

Variance can be computed in **linear** time [30, 31]. In formulas (20) and (21), variance (and standard deviation) is computed using factors of 1/n and 1/(n - 1), respectively. In some situations, these factors are replaced by others, 1/(n - 1.5) as an example [32, 33]. To allow for such factors, this proposal, like NumPy [34], enables one to specify a **delta degrees of freedom** [34], a value subtracted from \(n\).

### 3.5 Standard Deviation

The **population standard deviation** [19] \((n \geq 1)\), denoted \(\sigma\), 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.

### 4 Proposal

This document proposes the inclusion of the statistics **mean** (arithmetic, geometric and harmonic), **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>`. These functions and accumulator objects are limited to **existing** (C++) data types, although this restriction is expected to be relaxed at some point in the future, thus allowing for **custom** data types.
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 type below.

```cpp
template<std::ranges::input_range R, typename P>
using stats_result_t = std::conditional_t<
    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>;
```

4.1.1 Mean

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

```cpp
// (1)
template<std::ranges::input_range R, typename P = std::identity,
    std::floating_point Result = stats_result_t<R, P>>
constexpr auto mean(R&& r, P proj = {}) -> Result;

// (2)
template<std::ranges::input_range R,
    typename P1 = std::identity,
    std::ranges::input_range W,
    typename P2 = std::identity,
    std::floating_point Result = stats_result_t<R, P1>>
requires std::is_arithmetic_v<typename std::projected<std::ranges::iterator_t<W>, P2>::value_type>
constexpr auto mean(R&& r, W&& w, P1 proj1 = {}, P2 proj2 = {}) -> Result;

// (3)
template<typename ExecutionPolicy,
    std::ranges::forward_range R,
    typename P = std::identity,
    std::floating_point Result = stats_result_t<R, P>>
requires std::is_execution_policy_v<std::remove_cvref_t<ExecutionPolicy>>
auto mean(ExecutionPolicy&& policy, R&& r, P proj = {}) -> Result;

// (4)
template<typename ExecutionPolicy,
    std::ranges::forward_range R,
    typename P1 = std::identity,
    std::ranges::forward_range W,
    typename P2 = std::identity,
    std::floating_point Result = stats_result_t<R, P1>>
requires std::is_execution_policy_v<std::remove_cvref_t<ExecutionPolicy>> &&
    std::is_arithmetic_v<typename std::projected<std::ranges::iterator_t<W>, P2>::value_type>
auto mean(ExecutionPolicy&& policy, R&& r, W&& w, P1 proj1 = {}, P2 proj2 = {}) -> Result;

// (5)
template<std::ranges::input_range R, typename P = std::identity,
    std::floating_point Result = stats_result_t<R, P>>
constexpr auto geometric_mean(R&& r, P proj = {}) -> Result;

// (6)
template<std::ranges::input_range R,
    typename P1 = std::identity,
    std::ranges::input_range W,
    typename P2 = std::identity,
    std::floating_point Result = stats_result_t<R, P1>>
requires std::is_arithmetic_v<typename std::projected<std::ranges::iterator_t<W>, P2>::value_type>
constexpr auto geometric_mean(R&& r, W&& w, P1 proj1 = {}, P2 proj2 = {}) -> Result;

// (7)
```
template<
  typename ExecutionPolicy,
  std::ranges::forward_range R,
  typename P = std::identity,
  std::floating_point Result = stats_result_t<R, P>>
requires std::is_execution_policy_v<std::remove_cvref_t<ExecutionPolicy>>
auto geometric_mean(ExecutionPolicy&& policy, R&& r, P proj = {}) -> Result;

// (8)

template<
  typename ExecutionPolicy,
  std::ranges::forward_range R,
  typename P1 = std::identity,
  std::ranges::forward_range W,
  typename P2 = std::identity,
  std::floating_point Result = stats_result_t<R, P1>>
requires std::is_execution_policy_v<std::remove_cvref_t<ExecutionPolicy>> &&
  std::is_arithmetic_v<typename std::projected<std::ranges::iterator_t<W>, P2>::value_type>
auto geometric_mean(
  ExecutionPolicy&& policy, R&& r, W&& w, P1 proj1 = {}, P2 proj2 = {}) -> Result;

// (9)

template<
  std::ranges::input_range R,
  typename P = std::identity,
  std::floating_point Result = stats_result_t<R, P>>
constexpr auto harmonic_mean(R&& r, P proj = {}) -> Result;

// (10)

template<
  std::ranges::input_range R,
  typename P1 = std::identity,
  std::ranges::input_range W,
  typename P2 = std::identity,
  std::floating_point Result = stats_result_t<R, P1>>
requires std::is_arithmetic_v<typename std::projected<std::ranges::iterator_t<W>, P2>::value_type>
constexpr auto harmonic_mean(R&& r, W&& w, P1 proj1 = {}, P2 proj2 = {}) -> Result;

// (11)

template<
  typename ExecutionPolicy,
  std::ranges::forward_range R,
  typename P = std::identity,
  std::floating_point Result = stats_result_t<R, P>>
requires std::is_execution_policy_v<std::remove_cvref_t<ExecutionPolicy>>
auto harmonic_mean(ExecutionPolicy&& policy, R&& r, P proj = {}) -> Result;

// (12)

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 \( r \) is returned.

Complexity
\( O(N) \), where \( N = \text{std::ranges::distance}(r) \).

Error Handling
- If \( r \) or \( w \) is empty, \( +\infty \) is returned.
- If any element of \( r \) or \( w \) is \( \pm\infty \), NaN is returned.
- If any element of \( r \) or \( w \) is NaN, NaN is returned.
- If the sum of the weights of \( w \) is 0, then \( +\infty \) is returned.
- In the case of geometric mean, if the product of the elements of \( r \) is negative and \( n \) is even, NaN is returned.
- In the case of harmonic mean, if any element of \( r \) is negative, NaN is returned.
- In the case of harmonic mean, if any element of \( r \) is 0, \( +\infty \) is returned.

Example
```cpp
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::mean(A, &PRODUCT::price);
std::cout << " \nmean 2 = " << std::geometric_mean(A, &PRODUCT::quantity);
std::cout << " \nmean 3 = " << std::harmonic_mean(
    A, std::vector<double>{ 0.2, 0.1, 0.3, 0.05, 0.05, 0.05, 0.1, 0.15 },
    &PRODUCT::price);
```

4.1.2 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.

```cpp
enum class stats_data_kind { population, sample };
```

These functions further make use of the enumerated values below.

```cpp
enum class stats_skewness_kind { unadjusted, adjusted_fisher_pearson };
```

Additionally, the abbreviation stddev is selected for “standard deviation”, as this abbreviation already appears in the standard library [36]. Variance, however, is expressed in full (as variance), so as to avoid confusion with “variable”.

```cpp
// (1)
template< std::ranges::input_range R, typename P = std::identity,
          std::floating_point Result = stats_result_t<R, P>,
          constexpr auto skewness(R&& r,
          stats_data_kind dkind, stats_skewness_kind skind = stats_skewness_kind::unadjusted, 
          P proj = {}) -> Result;
```

```cpp
// (2)
template< std::ranges::input_range R, typename P = std::identity,
          std::floating_point Result = stats_result_t<R, P>,
```

typename T1, typename T2>
requires std::is_arithmetic_v<T1> && std::is_arithmetic_v<T2>
constexpr auto skewness(R&& r,
    T1 mean, T2 stddev,
    stats_data_kind dkind, stats_skewness_kind skind = stats_skewness_kind::unadjusted,
P proj = { }) -> Result;

// (3)
template<
    std::ranges::input_range R,
    typename P1 = std::identity,
    std::ranges::input_range W,
    typename P2 = std::identity,
    std::floating_point Result = stats_result_t<R, P1>>
requires std::is_arithmetic_v<
    typename stats_data_kind dkind, stats_skewness_kind skind = stats_skewness_kind::unadjusted,
P1 proj1 = { }, P2 proj2 = { }) -> Result;

// (4)
template<
    std::ranges::input_range R,
    typename P1 = std::identity,
    std::ranges::input_range W,
    typename P2 = std::identity,
    std::floating_point Result = stats_result_t<R, P1>,
    typename T1, typename T2>
requires std::is_arithmetic_v<
    typename stats_data_kind dkind, stats_skewness_kind skind = stats_skewness_kind::unadjusted,
P1 proj1 = { }, P2 proj2 = { }) -> Result;

// (5)
template<
    typename ExecutionPolicy,
    std::ranges::forward_range R,
    typename P = std::identity,
    std::floating_point Result = stats_result_t<R, P>>
requires std::is_execution_policy_v<std::remove_cvref_t<ExecutionPolicy>>
auto skewness(ExecutionPolicy&& policy,
    R&& r,
    stats_data_kind dkind, stats_skewness_kind skind = stats_skewness_kind::unadjusted,
P proj = { }) -> Result;

// (6)
template<
    typename ExecutionPolicy,
    std::ranges::forward_range R,
    typename P = std::identity,
    typename T2>
requires std::is_execution_policy_v<std::remove_cvref_t<ExecutionPolicy>> &&
    std::is_arithmetic_v<T1> && std::is_arithmetic_v<T2>
auto skewness(ExecutionPolicy&& policy,
    R&& r,
    T1 mean, T2 stddev,
    stats_data_kind dkind, stats_skewness_kind skind = stats_skewness_kind::unadjusted,
P proj = { }) -> Result;

// (7)
template<
    typename ExecutionPolicy,
    std::ranges::forward_range R,
    typename P1 = std::identity,
    std::ranges::forward_range W,
    typename P2 = std::identity,
    std::floating_point Result = stats_result_t<R, P1>>
requires std::is_execution_policy_v<std::remove_cvref_t<ExecutionPolicy>> &&
   std::is_arithmetic_v<typename std::projected<std::ranges::iterator_t<W>, P2>::value_type>
auto skewness(ExecutionPolicy&& policy,
R&& r, W&& w,
stats_data_kind dkind, stats_skewness_kind skind = stats_skewness_kind::unadjusted,
P1 proj1 = {}, P2 proj2 = {}) -> Result;

// (8)

template<typename ExecutionPolicy,
  std::ranges::forward_range R, typename P1 = std::identity,
  std::ranges::forward_range W, typename P2 = std::identity,
  std::floating_point Result = stats_result_t<R, P1>,
  typename T1, typename T2>
requires std::is_execution_policy_v<std::remove_cvref_t<ExecutionPolicy>> &&
   std::is_arithmetic_v<typename std::projected<std::ranges::iterator_t<W>, P2>::value_type> &&
   std::is_arithmetic_v<T1> && std::is_arithmetic_v<T2>
auto skewness(ExecutionPolicy&& policy,
R&& r, W&& w,
T1 mean, T2 stddev,
stats_data_kind dkind, stats_skewness_kind skind = stats_skewness_kind::unadjusted,
P1 proj1 = {}, P2 proj2 = {}) -> Result;

Parameters
• r - the range of the elements to examine
• dkind - the type of data represented by the elements of r, either population or sample
• skind - the type of skewness represented by the elements of r, either unadjusted or adjusted_fisher_pearson
• proj - the projection to apply to the elements of r
• mean - the (precomputed) mean of the elements of r
• stddev - 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, NaN is returned.
• If any element of r or w is ±∞, NaN is returned.
• If any element of r or w is NaN, NaN is returned.
• If the sum of the weights of w is 0, then +∞ is returned.
• If stddev is negative or 0, NaN is returned.
Example

```cpp
std::cout << "skewness = " << std::skewness(
    std::vector<int>{ 2, 3, 5, 7, 11, 13, 17, 19 }, std::stats_data_kind::population);
```

### 4.1.3 Kurtosis

The proposed forms of the kurtosis functions are given as follows. These functions make use of the enumerated values below.

```cpp
enum class stats_kurtosis_kind { fisher, pearson }
```

// (1)
```cpp
template<std::ranges::input_range R, typename P = std::identity,
         std::floating_point Result = stats_result_t<R, P>>
constexpr auto kurtosis(R&& r,
    stats_data_kind dkind, stats_kurtosis_kind kkind = stats_kurtosis_kind::fisher,
    P proj = {}) -> Result;
```

// (2)
```cpp
template<std::ranges::input_range R, typename P = std::identity,
         std::floating_point Result = stats_result_t<R, P>,
         typename T1, typename T2>
requires std::is_arithmetic_v<T1> && std::is_arithmetic_v<T2>
constexpr auto kurtosis(R&& r,
    T1 mean, T2 stddev,
    stats_data_kind dkind, stats_kurtosis_kind kkind = stats_kurtosis_kind::fisher,
    P proj = {}) -> Result;
```

// (3)
```cpp
template<std::ranges::input_range R, typename P1 = std::identity,
         std::ranges::input_range W,
         typename P2 = std::identity,
         std::floating_point Result = stats_result_t<R, P1>>
requires std::is_arithmetic_v<typename std::projected<std::ranges::iterator_t<W>, P2>::value_type>
constexpr auto kurtosis(R&& r, W&& w,
    stats_data_kind dkind, stats_kurtosis_kind kkind = stats_kurtosis_kind::fisher,
    P1 proj1 = {}, P2 proj2 = {}) -> Result;
```

// (4)
```cpp
template<std::ranges::input_range R, typename P1 = std::identity,
         std::ranges::input_range W,
         typename P2 = std::identity,
         std::floating_point Result = stats_result_t<R, P1>,
         typename T1, typename T2>
requires std::is_arithmetic_v<typename std::projected<std::ranges::iterator_t<W>, P2>::value_type> &&
std::is_arithmetic_v<T1> && std::is_arithmetic_v<T2>
constexpr auto kurtosis(R&& r, W&& w,
    T1 mean, T2 stddev,
    stats_data_kind dkind, stats_kurtosis_kind kkind = stats_kurtosis_kind::fisher,
    P1 proj1 = {}, P2 proj2 = {}) -> Result;
```

// (5)
```cpp
template<typename ExecutionPolicy,
         std::ranges::forward_range R, typename P = std::identity,
         std::floating_point Result = stats_result_t<R, P>>
requires std::is_execution_policy_v<ExecutionPolicy> &&
std::is_arithmetic_v<
```
auto kurtosis(ExecutionPolicy&& policy,
    R&& r,
    stats_data_kind dkind, stats_kurtosis_kind kkind = stats_kurtosis_kind::fisher,
    P proj = {}) -> Result;

// (6)

auto kurtosis(ExecutionPolicy&& policy,
    R&& r,
    stats_data_kind dkind, stats_kurtosis_kind kkind = stats_kurtosis_kind::fisher,
    P proj = {}) -> Result;

// (7)

auto kurtosis(ExecutionPolicy&& policy,
    R&& r,
    T1 mean, T2 stddev,
    stats_data_kind dkind, stats_kurtosis_kind kkind = stats_kurtosis_kind::fisher,
    P proj = {}) -> Result;

// (8)

auto kurtosis(ExecutionPolicy&& policy,
    R&& r, W&& w,
    stats_data_kind dkind, stats_kurtosis_kind kkind = stats_kurtosis_kind::fisher,
    P1 proj1 = {}, P2 proj2 = {}) -> Result;

Parameters

- r - the range of the elements to examine
- dkind - the type of data represented by the elements of r, either population or sample
- kkind - the type of kurtosis, either Fisher or Pearson
- proj - the projection to apply to the elements of r
- mean - the (precomputed) mean of the elements of r
• stddev - 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 \( N = \text{std::ranges::distance}(r) \).

**Error Handling**
- If the size of \( r \) or \( w \) is less than 4, NaN is returned.
- If any element of \( r \) or \( w \) is \( \pm\infty \), NaN is returned.
- If any element of \( r \) or \( w \) is NaN, NaN is returned.
- If the sum of the weights of \( w \) is 0, then \( +\infty \) is returned.
- If stddev is negative or 0, NaN is returned.

**Example**
```cpp
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::kurtosis(
    v, v_wgts, std::stats_data_kind::sample, std::stats_kurtosis_kind::fisher);
```

### 4.1.4 Variance

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

```
// (1)
template<std::ranges::input_range R, typename P = std::identity,
    std::floating_point Result = stats_result_t<R, P>,
    typename D>
requires std::is_arithmetic_v<D>
constexpr auto variance(R&& r, D ddof, P proj = {}) -> Result;

// (2)
template<std::ranges::input_range R, typename P = std::identity,
    std::floating_point Result = stats_result_t<R, P>,
    typename T,
    typename D>
requires std::is_arithmetic_v<T> &&
    std::is_arithmetic_v<D>
constexpr auto variance(R&& r, T mean, D ddof, P proj = {}) -> Result;

// (3)
template<std::ranges::input_range R, typename P1 = std::identity,
    std::ranges::input_range W, typename P2 = std::identity,
    std::floating_point Result = stats_result_t<R, P1>,
    typename D>
```
requires std::is_arithmetic_v<
  typename std::projected<std::ranges::iterator_t<W>, P2>::value_type>
  &&
  std::is_arithmetic_v<D>
constexpr auto variance(
  R&& r, W&& w, stats_data_kind dkind, P1 proj1 = {}, P2 proj2 = {}) -> Result;

// (4)
template<
  std::ranges::input_range R,
  typename P1 = std::identity,
  std::ranges::input_range W,
  typename P2 = std::identity,
  std::floating_point Result = stats_result_t<R, P1>,
  typename T,
  typename D>
requires std::is_arithmetic_v<
  typename std::projected<std::ranges::iterator_t<W>, P2>::value_type>
  &&
  std::is_arithmetic_v<T> &&
  std::is_arithmetic_v<D>
constexpr auto variance(
  R&& r, W&& w, T mean, stats_data_kind dkind, P1 proj1 = {}, P2 proj2 = {}) -> Result;

// (5)
template<typename ExecutionPolicy,
  std::ranges::forward_range R,
  typename P = std::identity,
  std::floating_point Result = stats_result_t<R, P>,
  typename T,
  typename D>
requires std::is_execution_policy_v<std::remove_cvref_t<ExecutionPolicy>>
  &&
  std::is_arithmetic_v<D>
auto variance(ExecutionPolicy&& policy, R&& r, D ddof, P proj = {}) -> Result;

// (6)
template<typename ExecutionPolicy,
  std::ranges::forward_range R,
  typename P1 = std::identity,
  std::ranges::forward_range W,
  typename P2 = std::identity,
  std::floating_point Result = stats_result_t<R, P1>,
  typename D>
requires std::is_execution_policy_v<std::remove_cvref_t<ExecutionPolicy>>
  &&
  std::is_arithmetic_v<P2>
auto variance(ExecutionPolicy&& policy, R&& r, W&& w, stats_data_kind dkind, P1 proj1 = {}, P2 proj2 = {}) -> Result;

// (7)
template<typename ExecutionPolicy,
  std::ranges::forward_range R,
  typename P1 = std::identity,
  std::ranges::forward_range W,
  typename P2 = std::identity,
  std::floating_point Result = stats_result_t<R, P1>,
  typename D>
requires std::is_execution_policy_v<std::remove_cvref_t<ExecutionPolicy>>
  &&
  std::is_arithmetic_v<P2>
auto variance(ExecutionPolicy&& policy,
  R&& r, W&& w,
  stats_data_kind dkind,
  P1 proj1 = {}, P2 proj2 = {}) -> Result;

// (8)
template<typename ExecutionPolicy,
  std::ranges::forward_range R,
  typename P1 = std::identity,
  std::ranges::forward_range W,
  typename P2 = std::identity,
std::floating_point Result = stats_result_t<R, P1>,
  typename T,
  typename D>
requires std::is_execution_policy_v<std::remove_cvref_t<ExecutionPolicy>> &&
  typename std::projected<std::ranges::iterator_t<W>, P2>::value_type> &&
  std::is_arithmetic_v<T> &&
  std::is_arithmetic_v<D>
auto variance(ExecutionPolicy&& policy,
  R&& r, W&& w,
  T mean,
  stats_data_kind dkind,
  P1 proj1 = {}, P2 proj2 = {}) -> Result;

Parameters

- r - the range of the elements to examine
- ddof - delta degrees of freedom subtracted from the size of r, namely N, yielding the factor 1/(N−ddof) by which the variance is scaled
- proj - the projection to apply to the elements of r
- mean - the (precomputed) mean of the elements of r
- w - the range of the weights to use
- dkind - the type of data represented by the elements of r, either population or sample
- 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 is returned.

Complexity

$O(N)$, where $N = \text{std::ranges::distance}(r)$.

Error Handling

- If the $N \leq \text{ddof}$, NaN is returned.
- If any element of $r$ or $w$ is $\pm \infty$, NaN is returned.
- If any element of $r$ or $w$ is NaN, NaN is returned.
- If the sum of the weights of $w$ is 0, then $+\infty$ is returned.

Example

```cpp
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::variance(A, 0, &PRODUCT::price);
std::cout << "\nvariance 2 = " << std::variance(A, 1, &PRODUCT::price);
```
4.1.5 Standard Deviation

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

```cpp
// (1)
template<std::ranges::input_range R, typename P = std::identity,
         std::floating_point Result = stats_result_t<R, P>,
         typename D>
requires std::is_arithmetic_v<D>
constexpr auto stddev(R&& r, D ddof, P proj = {}) -> Result;

// (2)
template<std::ranges::input_range R, typename P = std::identity,
         std::floating_point Result = stats_result_t<R, P>,
         typename T,
         typename D>
requires std::is_arithmetic_v<T> &&
         std::is_arithmetic_v<D>
constexpr auto stddev(R&& r, T mean, D ddof, P proj = {}) -> Result;

// (3)
template<std::ranges::input_range R, typename P1 = std::identity,
         std::ranges::input_range W, typename P2 = std::identity,
         std::floating_point Result = stats_result_t<R, P1>,
         typename D>
requires std::is_arithmetic_v<
         typename std::projected<std::ranges::iterator_t<W>, P2>::value_type> &&
         std::is_arithmetic_v<D>
constexpr auto stddev(
    R&& r, W&& w, stats_data_kind dkind, P1 proj1 = {}, P2 proj2 = {}) -> Result;

// (4)
template<std::ranges::input_range R, typename P1 = std::identity,
         std::ranges::input_range W, typename P2 = std::identity,
         std::floating_point Result = stats_result_t<R, P1>,
         typename T,
         typename D>
requires std::is_arithmetic_v<
         typename std::projected<std::ranges::iterator_t<W>, P2>::value_type> &&
         std::is_arithmetic_v<T> &&
         std::is_arithmetic_v<D>
constexpr auto stddev(
    R&& r, W&& w, T mean, stats_data_kind dkind, P1 proj1 = {}, P2 proj2 = {}) -> Result;

// (5)
template<typename ExecutionPolicy,
         std::ranges::forward_range R, typename P = std::identity,
         std::floating_point Result = stats_result_t<R, P>,
         typename D>
requires std::is_execution_policy_v<std::remove_cvref_t<ExecutionPolicy>> &&
         std::is_arithmetic_v<D>
auto stddev(ExecutionPolicy&& policy, R&& r, D ddof, P proj = {}) -> Result;

// (6)
template<typename ExecutionPolicy,
         std::ranges::forward_range R, typename P = std::identity,
         std::floating_point Result = stats_result_t<R, P>,
         typename T,
         typename D>
```
auto stddev(ExecutionPolicy&& policy, R&& r, T mean, D ddof, P proj = {}) -> Result;

// (7)
-template<
typename ExecutionPolicy,
  std::ranges::forward_range R,
  typename P1 = std::identity,
  std::ranges::forward_range W,
  typename P2 = std::identity,
  std::floating_point Result = stats_result_t<R, P1>,
  typename D>
requires std::is_execution_policy_v<std::remove_cvref_t<ExecutionPolicy>> &&
  std::is_arithmetic_v<typename std::projected<std::ranges::iterator_t<W>, P2>::value_type> &&
  std::is_arithmetic_v<D>
auto stddev(ExecutionPolicy&& policy,
  R&& r, W&& w,
  stats_data_kind dkind,
  P1 proj1 = {}, P2 proj2 = {}) -> Result;

// (8)
-template<
typename ExecutionPolicy,
  std::ranges::forward_range R,
  typename P1 = std::identity,
  std::ranges::forward_range W,
  typename P2 = std::identity,
  std::floating_point Result = stats_result_t<R, P1>,
  typename T,
  typename D>
requires std::is_execution_policy_v<std::remove_cvref_t<ExecutionPolicy>> &&
  std::is_arithmetic_v<typename std::projected<std::ranges::iterator_t<W>, P2>::value_type> &&
  std::is_arithmetic_v<T> &&
  std::is_arithmetic_v<D>
auto stddev(ExecutionPolicy&& policy,
  R&& r, W&& w,
  T mean,
  stats_data_kind dkind,
  P1 proj1 = {}, P2 proj2 = {}) -> Result;

Parameters

- \( r \) - the range of the elements to examine
- \( \text{ddof} \) - delta degrees of freedom subtracted from the size of \( r \), namely \( N \), yielding the factor \( 1/(N - \text{ddof}) \) by which the standard deviation is scaled
- \( \text{proj} \) - the projection to apply to the elements of \( r \)
- \( \text{mean} \) - the (precomputed) mean of the elements of \( r \)
- \( w \) - the range of the weights to use
- \( \text{dkind} \) - the type of data represented by the elements of \( r \), either \text{population} or \text{sample}
- \( \text{proj1} \) - the projection to apply to the elements of \( r \)
- \( \text{proj2} \) - the projection to apply to the elements of \( w \)
- \( \text{policy} \) - the execution policy to use

Return Value

If no errors occur, the (weighted) standard deviation of the elements of \( r \) is returned.
Complexity

\( O(N) \), where \( N = \text{std::ranges::distance(r)} \).

Error Handling

- If the \( N \leq \text{ddof} \), NaN is returned.
- If any element of \( r \) or \( w \) is \( \pm \infty \), NaN is returned.
- If any element of \( r \) or \( w \) is NaN, NaN is returned.
- If the sum of the weights of \( w \) is 0, then \( +\infty \) is returned.

Example

```cpp
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::stddev(v, 0);
std::cout << "standard deviation 2 = " << std::stddev(v, v_wgts, std::stats_data_kind::sample);
```

4.2 Accumulator Objects

Each statistic is secondly aggregated into a class, derived from either `stat_accum` or `weighted_stat_accum`. 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 [37]. Accumulator objects are passed to an accumulator function `stats_accum` that makes a single pass over the values. The proposed forms of these classes and functions are given in the following sections.

4.2.1 Accumulator

The proposed forms of the `stat_accum` and `weighted_stat_accum` classes are given as follows.

```cpp
// (1)
template<typename T>
class stat_accum
{
public:
  constexpr stat_accum() noexcept; // (1)
  constexpr stat_accum(const stat_accum& other) noexcept; // (2)
  constexpr stat_accum(stat_accum&& other) noexcept; // (3)
  constexpr stat_accum& operator=(const stat_accum& other) noexcept; // (4)
  constexpr stat_accum& operator=(stat_accum&& other) noexcept; // (5)
  constexpr ~stat_accum() noexcept;
};
// (2)
template<typename T, typename W = double>
class weighted_stat_accum
{
public:
  constexpr weighted_stat_accum() noexcept; // (1)
  constexpr weighted_stat_accum(const weighted_stat_accum& other) noexcept; // (2)
  constexpr weighted_stat_accum(weighted_stat_accum&& other) noexcept; // (3)
  constexpr weighted_stat_accum& operator=(const weighted_stat_accum& other) noexcept; // (4)
  constexpr weighted_stat_accum& operator=(weighted_stat_accum&& other) noexcept; // (5)
  constexpr ~weighted_stat_accum() noexcept;
};
```
Member Functions

- constructor - constructs the accumulator object
- operator= - assigns values to the accumulator object

Parameters

- other - another accumulator object to be used as source to initialize the elements of the accumulator object with

The proposed forms of the `stats_accum` functions are given as follows.

```cpp
// (1)
template<std::ranges::input_range R, typename ...Args>
requires std::derived_from<Args, stat_accum<std::iter_value_t<R>>> ||
    std::derived_from<Args, weighted_stat_accum<std::iter_value_t<R>>>
constexpr void stats_accum(
    R&& r, stat_accum<std::iter_value_t<R>>& stat, Args& ... stats);

// (2)
template<std::ranges::input_range R, std::ranges::input_range W, typename ...Args>
requires std::derived_from<Args, stat_accum<std::iter_value_t<R>>> ||
    std::derived_from<Args, weighted_stat_accum<std::iter_value_t<R>>>
constexpr void stats_accum(R&& r, W&& w,
    weighted_stat_accum<std::iter_value_t<R>, std::iter_value_t<W>>& stat,
    Args& ... stats);

// (3)
template<typename ExecutionPolicy, std::ranges::forward_range R, typename ...Args>
requires std::is_execution_policy_v<std::remove_cvref_t<ExecutionPolicy>> &&
    (std::derived_from<Args, stat_accum<std::iter_value_t<R>>> ||
        std::derived_from<Args, weighted_stat_accum<std::iter_value_t<R>>>)
void stats_accum(ExecutionPolicy&& policy,
    R&& r,
    stat_accum<std::iter_value_t<R>>& stat, Args& ... stats);

// (4)
template<typename ExecutionPolicy, 
    std::ranges::forward_range R, std::ranges::forward_range W, 
    typename ...Args>
requires std::is_execution_policy_v<std::remove_cvref_t<ExecutionPolicy>> &&
    (std::derived_from<Args, stat_accum<std::iter_value_t<R>>> ||
        std::derived_from<Args, weighted_stat_accum<std::iter_value_t<R>>>)
void stats_accum(ExecutionPolicy&& policy,
    R&& r, W&& w,
    weighted_stat_accum<std::iter_value_t<R>, std::iter_value_t<W>>& stat,
    Args& ... stats);
```

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
- w - the range of the weights to use
- policy - the execution policy to use
Error Handling

- If \( r \) or \( w \) is empty, \( +\infty \) is returned.
- If any element of \( r \) or \( w \) is \( \pm\infty \), \( +\infty \) is returned.
- If any element of \( r \) or \( w \) is NaN, NaN is returned.

4.3 Mean

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

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

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

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

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

template<typename T, typename P = std::identity, std::floating_point Result = double>
class harmonic_mean_accum : public stat_accum<T>
{
public:
    constexpr harmonic_mean_accum(P proj = {}) noexcept; // (1)
    constexpr harmonic_mean_accum(const harmonic_mean_accum& other) noexcept; // (2)
    constexpr harmonic_mean_accum(harmonic_mean_accum& other) noexcept; // (3)
    constexpr harmonic_mean_accum& operator=(
        const harmonic_mean_accum& other) noexcept; // (4)
    constexpr harmonic_mean_accum& operator=(harmonic_mean_accum& other) noexcept; // (5)
    constexpr auto harmonic_mean() const noexcept -> Result;
}; // (6)

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

Member Functions

• constructor - constructs the accumulator object
• operator= - assigns values to the accumulator object
• mean - returns the (weighted) mean
• geometric_mean - returns the (weighted) geometric mean
• harmonic_mean - returns the (weighted) harmonic mean
Parameters

- proj - the projection to apply to the elements
- 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
- 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 is returned.

Error Handling

- If the associated range (or weights) is empty, $+\infty$ is returned.
- If any element of the associated range is $\pm\infty$, NaN is returned.
- If any element of the associated range is NaN, NaN is returned.
- If the sum of the weights is 0, then $+\infty$ is returned.
- In the case of geometric_mean_accum or weighted_geometric_mean_accum, if the product of the elements of the associated range is negative and the length of that range is even, NaN is returned.
- In the case of harmonic_mean_accum or weighted_harmonic_mean_accum, if any element of the associated range is negative, NaN is returned.
- In the case of harmonic_mean_accum or weighted_harmonic_mean_accum, if any element of the associated range is negative or 0, $+\infty$ is returned.

Example

```cpp
std::mean_accum<int> m1;
std::geometric_mean_accum<int> gm;
std::harmonic_mean_accum<int> hm;
std::stats_accum(std::list<int>{ 3, 3, 1, 2, 2 }, m1, gm, hm);
std::cout << "mean 1 = " << m1.mean();
std::cout << "\nmean 2 = " << gm.geometric_mean();
std::cout << "\nmean 3 = " << hm.harmonic_mean();

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::mean_accum<s_t, std::function<double(const s_t&)>> m2(f);
std::stddev_accum<s_t, std::function<double(const s_t&)>> s(0, f);
std::stats_accum(s_t, std::function<double(const s_t&)>> s(0, f);
std::cout << "\nmean 4 = " << m2.mean();
std::cout << "\nstandard deviation 1 = " << s.stddev();
```
4.4 Skewness

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

```cpp
// (1)
template<typename T, typename P = std::identity, std::floating_point Result = double>
class skewness_accum : public stat_accum<T>
{
    public:
        constexpr skewness_accum() noexcept; // (1)
        constexpr skewness_accum(stats_data_kind dkind, 
            stats_skewness_kind = stats_skewness_kind::unadjusted, 
            P proj = {}) noexcept; // (2)
        constexpr skewness_accum(const skewness_accum& other) noexcept; // (3)
        constexpr skewness_accum(skewness_accum&& other) noexcept; // (4)
        constexpr skewness_accum& operator=(const skewness_accum& other) noexcept; // (5)
        constexpr skewness_accum& operator=(skewness_accum&& other) noexcept; // (6)
        constexpr auto skewness() const noexcept -> Result;
    }

// (2)
template<typename T, typename P1 = std::identity, 
          typename W = double, typename P2 = std::identity, 
          std::floating_point Result = double>
class weighted_skewness_accum : public weighted_stat_accum<T, W>
{
    public:
        constexpr weighted_skewness_accum() noexcept; // (1)
        constexpr weighted_skewness_accum(stats_data_kind dkind, 
            stats_skewness_kind = stats_skewness_kind::unadjusted, 
            P1 proj1 = {}, P2 proj2 = {}) noexcept; // (2)
        constexpr weighted_skewness_accum(const weighted_skewness_accum& other) noexcept; // (3)
        constexpr weighted_skewness_accum(weighted_skewness_accum&& other) noexcept; // (4)
        constexpr weighted_skewness_accum& operator=(const weighted_skewness_accum& other) noexcept; // (5)
        constexpr weighted_skewness_accum& operator=(weighted_skewness_accum&& other) noexcept; // (6)
        constexpr auto skewness() const noexcept -> Result;
    }
```

Member Functions

- **constructor** - constructs the accumulator object
- **operator=** - assigns values to the accumulator object
- **skewness** - returns the (weighted) skewness

Parameters

- **dkind** - the type of data represented by the elements, either population or sample
- **skind** - the type of skewness represented by the elements, either unadjusted or adjusted_fisher_pearson
- **proj** - the projection to apply to the elements
- **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
- **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 is returned.

Error Handling

- If the size of the associated range (or weights) is less than 3, NaN is returned.
- If any element of the associated range is ±∞, NaN is returned.
- If any element of the associated range is NaN, NaN is returned.

Example

```cpp
std::list<int> L = { 3, 3, 1, 2, 2 };
std::mean_accum<int> m;
std::skewness_accum<int> sk(std::stats_data_kind::population);
std::stats_accum(L, m, sk);
std::cout << "mean = " << m.mean();
std::cout << "skewness = " << sk.skewness();
```

4.5 Kurtosis

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

```cpp
// (1)
template<typename T, typename P = std::identity, std::floating_point Result = double>
class kurtosis_accum : public stat_accum<T>
{
public:
    constexpr kurtosis_accum() noexcept; // (1)
constexpr kurtosis_accum(stats_data_kind dkind,
    stats_kurtosis_kind kkind = stats_kurtosis_kind::fisher,
    P proj = {}) noexcept; // (2)
constexpr kurtosis_accum(const kurtosis_accum& other) noexcept; // (3)
constexpr kurtosis_accum(kurtosis_accum&& other) noexcept; // (4)
constexpr kurtosis_accum& operator=(const kurtosis_accum& other) noexcept; // (5)
constexpr kurtosis_accum& operator=(kurtosis_accum&& other) noexcept; // (6)
constexpr auto kurtosis() const noexcept -> Result;
};

// (2)
template<typename T, typename P1 = std::identity,
    typename W = double, typename P2 = std::identity,
    std::floating_point Result = double>
class weighted_kurtosis_accum : public weighted_stat_accum<T, W>
{
public:
    constexpr weighted_kurtosis_accum() noexcept; // (1)
constexpr weighted_kurtosis_accum(stats_data_kind dkind,
    stats_kurtosis_kind kkind = stats_kurtosis_kind::fisher,
    P1 proj1 = {}, P2 proj2 = {}) noexcept; // (2)
constexpr weighted_kurtosis_accum(const weighted_kurtosis_accum& other) noexcept; // (3)
constexpr weighted_kurtosis_accum(weighted_kurtosis_accum&& other) noexcept; // (4)
constexpr weighted_kurtosis_accum& operator=(const weighted_kurtosis_accum& other) noexcept; // (5)
constexpr weighted_kurtosis_accum& operator=(weighted_kurtosis_accum&& other) noexcept; // (6)
constexpr weighted_kurtosis_accum& operator=(
```

24
Member Functions

- constructor - constructs the accumulator object
- operator= - assigns values to the accumulator object
- kurtosis - returns the (weighted) kurtosis

Parameters

- dkind - the type of data represented by the elements, either population or sample
- kkind - the type of kurtosis, either Fisher or Pearson
- proj - the projection to apply to the elements
- 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
- 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 is returned.

Error Handling

- If the size of the associated range (or weights) is less than 4, NaN is returned.
- If any element of the associated range is ±∞, NaN is returned.
- If any element of the associated range is NaN, NaN is returned.

Example

```cpp
std::mean_accum<int> m;
std::kurtosis_accum<int> k(
    std::stats_data_kind::sample, std::stats_kurtosis_kind::fisher);
std::stats_accum(std::list<int>{ 3, 3, 1, 2, 2 }, m, k);
std::cout << "mean = " << m.mean();
std::cout << "\n\nkurtosis = " << k.kurtosis();
```

4.6 Variance

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

```cpp
// (1)
template<typename T, typename P = std::identity,
    std::floating_point Result = double,
    typename D>
class variance_accum : public stat_accum<T>
{
public:
    constexpr variance_accum() noexcept; // (1)
    constexpr variance_accum(D ddof, P proj = {}) noexcept; // (2)
    constexpr variance_accum(const variance_accum& other) noexcept; // (3)
```
constexpr variance_accum(variance_accum&& other) noexcept; // (4)
constexpr variance_accum& operator=(const variance_accum& other) noexcept; // (5)
constexpr variance_accum& operator=(variance_accum&& other) noexcept; // (6)
constexpr auto variance() const noexcept -> Result;
};

// (2)
template<typename T, typename P1 = std::identity,
   typename W = double, typename P2 = std::identity,
   std::floating_point Result = double,
   typename D>
class weighted_variance_accum : public weighted_stat_accum<T, W>
{
public:
  constexpr weighted_variance_accum() noexcept; // (1)
  constexpr weighted_variance_accum(
      stats_data_kind dkind, P1 proj1 = {}, P2 proj2 = {}) noexcept; // (2)
  constexpr weighted_variance_accum(
      const weighted_variance_accum& other) noexcept; // (3)
  constexpr weighted_variance_accum(weighted_variance_accum&& other) noexcept; // (4)
  constexpr weighted_variance_accum& operator=(
      const weighted_variance_accum& other) noexcept; // (5)
  constexpr weighted_variance_accum& operator=(
      weighted_variance_accum&& other) noexcept; // (6)
  constexpr auto variance() const noexcept -> Result;
};

**Member Functions**

- **constructor** - constructs the accumulator object
- **operator=** - assigns values to the accumulator object
- **variance** - returns the (weighted) variance

**Parameters**

- **ddof** - delta degrees of freedom subtracted from the size of `r`, namely `N`, yielding the factor \(1/(N - \text{ddof})\) by which the variance is scaled
- **proj** - the projection to apply to the elements
- **other** - another accumulator object to be used as source to initialize the elements of the accumulator object with
- **dkind** - the type of data represented by the elements, either population or sample
- **proj1** - the projection to apply to the elements
- **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 is returned.

**Error Handling**

- If the size of the associated range (or weights) is less than or equal to `ddof`, NaN is returned.
- If any element of the associated range is ±\(∞\), NaN is returned.
- If any element of the associated range is NaN, NaN is returned.
- If the sum of the weights of is 0, then \(+∞\) is returned.
Example

```cpp
std::list<int> L = { 3, 3, 1, 2, 2 };
std::geometric_mean_accum<int> gm;
std::harmonic_mean_accum<int> hm;
std::variance_accum<int> v(1);
std::stats_accum(std::execution::par, L, gm, hm, v);
std::cout << "mean 1 = " << gm.geometric_mean();
std::cout << 
mean 2 = " << hm.harmonic_mean();
std::cout << "\nvariance = " << v.variance();
```

4.7 Standard Deviation

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

```cpp
// (1)
template<
typename T,
typename P = std::identity,
        typename Result = double,
        typename D>
class stddev_accum : public variance_accum<T, P, Result, D>
{
public:
    constexpr stddev_accum() noexcept; // (1)
    constexpr stddev_accum(D ddof, P proj = {}) noexcept; // (2)
    constexpr stddev_accum(const stddev_accum& other) noexcept; // (3)
    constexpr stddev_accum(stddev_accum&& other) noexcept; // (4)
    constexpr auto stddev() const noexcept -> Result;
};

// (2)
template<
typename T,
typename P1 = std::identity,
        typename W = double, typename P2 = std::identity,
        typename Result = double,
        typename D>
class weighted_stddev_accum : public weighted_variance_accum<T, P1, W, P2, Result, D>
{
public:
    constexpr weighted_stddev_accum() noexcept; // (1)
    constexpr weighted_stddev_accum(stats_data_kind dkind, P1 proj1 = {}, P2 proj2 = {}) noexcept; // (2)
    constexpr weighted_stddev_accum(const weighted_stddev_accum& other) noexcept; // (3)
    constexpr weighted_stddev_accum(weighted_stddev_accum& other) noexcept; // (4)
    constexpr weighted_stddev_accum& operator=(const weighted_stddev_accum& other) noexcept; // (5)
    constexpr weighted_stddev_accum& operator=(weighted_stddev_accum& other) noexcept; // (6)
    constexpr auto stddev() const noexcept -> Result;
};
```

Member Functions

- constructor - constructs the accumulator object
- operator= - assigns values to the accumulator object
- stddev - returns the (weighted) standard deviation
Parameters

- **ddof** - delta degrees of freedom subtracted from the size of \( r \), namely \( N \), yielding the factor \( 1/(N - \text{ddof}) \) by which the standard deviation is scaled
- **proj** - the projection to apply to the elements
- **other** - another accumulator object to be used as source to initialize the elements of the accumulator object with
- **dkind** - the type of data represented by the elements, either *population* or *sample*
- **proj1** - the projection to apply to the elements
- **proj2** - the projection to apply to the elements (of the associated weights)

Return Value

In the case of \( \text{value} \), if no errors occur, the (weighted) standard deviation of the elements is returned.

Error Handling

- If the size of the associated range (or weights) is less than or equal to \( \text{ddof} \), NaN is returned.
- If any element of the associated range is \( \pm \infty \), NaN is returned.
- If any element of the associated range is NaN, NaN is returned.
- If the sum of the weights of is 0, then \( +\infty \) is returned.

Example

```cpp
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::weighted_mean_accum<int> m;  
std::weighted_stddev_accum<int> s(0);  
std::stats_accum(L, W, m, s);  
std::cout << "mean = " << m.mean();  
std::cout << "standard deviation = " << s.stddev();
```

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 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 accumulator objects), as this might ultimately require that an implementation include a (rather inefficient) branching statement or virtual function call for each value (of a range).
5.2 Trimmed Mean

The issue of a trimmed mean is raised in [38]. A \((p\%)\) trimmed mean 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

\[
\text{auto } m = \text{data | std::ranges::sort | trim(p) | std::mean;} 
\]

5.3 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 version of the proposal. As a result, a programmer might handle such values using, as an example, a statement of the form

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

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.

References

https://docs.python.org/3/library/statistics.html


https://docs.julialang.org/en/v1/sttdlib/Statistics/


https://www.rdocumentation.org/packages/stats/versions/3.6.2

https://docs.rs/statistical/1.0.0/statistical/

https://support.sas.com/documentation/cdl/en/statug/65328/HTML/default/viewer.htm#statug_surveymeans_details06.htm


http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1069.pdf


https://en.cppreference.com/w/cpp/numeric/random

https://www.boost.org/doc/libs/1_73_0/libs/math/doc/html/dist.html

29