More Basic Statistics

Document Number: P2681R1
Author: Richard Dosselmann: dosselmr@uregina.ca
Contributors: Michael Chiu: chiu@cs.toronto.edu,
Guy Davidson: guy.davidson@hatcat.com
Oleksandr Koval: oleksandr.koval.dev@gmail.com
Larry Lewis: Larry.Lewis@sas.com
Johan Lundburg: lundberj@gmail.com
Jens Maurer: Jens.Maurer@gmx.net
Eric Niebler: eniebler@fb.com
Phillip Ratzloff: phil.ratzloff@sas.com
Vincent Reverdy: vreverdy@illinois.edu
John True
Michael Wong: michael@codeplay.com
Date: January 15, 2023 (mailing)
Project: ISO JTC1/SC22/WG21: Programming Language C++
Audience: SG6, SG19, LEWG
Contents

0 Revision History 2
1 Introduction 2
2 Motivation and Scope 2
  2.1 Univariate Statistics 2
    2.1.1 Percentile 2
    2.1.2 Mode 2
  2.2 Bivariate Statistics 3
    2.2.1 Covariance 3
    2.2.2 Correlation 3
3 Impact on the Standard 3
4 Design Decisions 3
  4.1 Naming 4
  4.2 Percentile Accumulator Objects 4
  4.3 Quantiles 4
5 Technical Specifications 4
  5.1 Header <stats> synopsis [stats.syn] 4
  5.2 Accumulator Objects 8
    5.2.1 Mode Accumulator Class Templates 8
    5.2.2 Covariance Accumulator Class Templates 10
    5.2.3 Correlation Accumulator Class Templates 11
    5.2.4 Accumulator Objects Accumulation Functions 12
  5.3 Freestanding Functions 12
    5.3.1 Freestanding Percentile Functions 12
    5.3.2 Freestanding Mode Functions 13
    5.3.3 Freestanding Covariance Functions 14
    5.3.4 Freestanding Correlation Functions 15
6 Acknowledgements 16
A Examples 17
1 Introduction
This document proposes an extension to the C++ library, to support more basic statistics.

2 Motivation and Scope
Basic statistics, five of which are proposed as part of P1708R6 [1], frequently arise in scientific and industrial, as well as general, applications. These functions do exist in Python [2], the foremost competitor to C++ in the area of machine learning, along with Calc [3], Excel [4], Julia [5], MATLAB [6], PHP [7], R [8], Rust [9], SAS [10] and SPSS [11]. Further need for such functions has been identified as part of SG19 (machine learning) [12].

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

Four more statistics are defined in this proposal. Linear regression is not included in this proposal, seeing that it is part of a larger family of regression models. As in P1708R6 [1], both freestanding functions and accumulator objects are proposed. This statistic, part of a family of regression models, is deferred to a future proposal. Like existing entities of the (C++) standard library, this proposal specifies only the interface of functions and objects, meaning that a variety of implementations are possible. This enables a vendor to favor accuracy [19] over performance for instance.

2.1 Univariate Statistics
Univariate statistics, which include those of P1708R6 [1], are computed over the (single set of) values \( x_1, x_2, \ldots, x_n \) \((n \geq 1)\). The following two univariate statistics are put forward in this proposal.

2.1.1 Percentile
A percentile [21] is defined as the value \( p \), in the range \((0, 1)\), below which 100\(p\)% of the (sorted in ascending order) values fall. As an example, the \( p = 0.5 \) percentile, known as the median [22], divides the values into two intervals (halves) of equal size. It is therefore the “middle” value. Both even and odd \( n \) [23] must be considered in the case of discrete data. At this point, “[t]here is no standard function for a weighted percentile” [24]. However, for weights \( w_1, w_2, \ldots, w_n \), such that

\[
\sum_{i=1}^{n} w_i = 1, \tag{1}
\]

the weighted median is defined as the \( x_i \) for which

\[
\sum_{i=1}^{k-1} w_i \leq \frac{1}{2} \tag{2}
\]

and

\[
\sum_{i=k+1}^{n} w_i \leq \frac{1}{2} \tag{3}
\]

A percentile (of the sorted in ascending order values) can be found in linear time.

2.1.2 Mode
The mode [22] is defined as the (perhaps not unique) most frequent value of the values. The weighted mode [25] is defined as the (perhaps not unique) value with the highest total weight. The mode can be found in linear time by comparing adjacent (sorted) values.
2.2 Bivariate Statistics

**Bivariate** [20] statistics are computed over the (two sets of) values \(x_1, x_2, \ldots, x_n\) and \(y_1, y_2, \ldots, y_n\). The following two bivariate statistics are put forward in this proposal.

### 2.2.1 Covariance

The **population** [22] covariance [26], a measure of the joint variability of the values \((n \geq 1)\), is defined as

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

where \(\mu_x\) and \(\mu_y\) are the (arithmetic) **population** means [22] of the values \(x_1, x_2, \ldots, x_n\) and \(y_1, y_2, \ldots, y_n\), respectively. The **sample** [22] covariance [26] \((n \geq 2)\) is defined as

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

where \(\bar{x}\) and \(\bar{y}\) are the **sample** means [22] of the values. The **weighted population** covariance [27] is defined as

\[
\hat{\sigma}_{xy} = \frac{1}{n} \sum_{i=1}^{n} w_i (x_i - \mu_x^*)(y_i - \mu_y^*)
\]

where \(\mu_x^*\) and \(\mu_y^*\) are the **weighted population** means [28, 29]. The **weighted sample** covariance is defined as

\[
\hat{s}_{xy} = \frac{\sum_{i=1}^{n} w_i (x_i - \bar{x}^*)(y_i - \bar{y}^*)}{(\sum_{i=1}^{n} w_i)^2 - \sum_{i=1}^{n} w_i^2}.
\]

and \(\bar{x}^*\) and \(\bar{y}^*\) are the **weighted sample** means [28, 29], respectively.

### 2.2.2 Correlation

The **Pearson population** product-moment correlation coefficient [30], in the range \([-1, 1]\], a measure of the linear association of the values \((n \geq 2)\), is defined as

\[
\rho_{xy} = \frac{\sigma_{xy}}{\sigma_x \sigma_y},
\]

where \(\sigma_x\) and \(\sigma_y\) are the **population** standard deviations [22] of the values \(x_1, x_2, \ldots, x_n\) and \(y_1, y_2, \ldots, y_n\), respectively. The **Pearson sample** product-moment correlation coefficient [30] is defined as

\[
r_{xy} = \frac{s_{xy}}{s_x s_y},
\]

where \(s_x\) and \(s_y\) are the **sample** standard deviations [22]. The **weighted Pearson population** product-moment correlation coefficient [27] is defined as

\[
\hat{\rho}_{xy} = \frac{\hat{\sigma}_{xy}}{\hat{\sigma}_x \hat{\sigma}_y},
\]

where \(\hat{\sigma}_x\) and \(\hat{\sigma}_y\) are the **weighted population** standard deviations. The **weighted Pearson sample** product-moment correlation coefficient [27] is defined as

\[
\hat{\rho}_{xy} = \frac{\hat{s}_{xy}}{\hat{s}_x \hat{s}_y},
\]

where \(\hat{s}_x\) and \(\hat{s}_y\) are the **weighted sample** standard deviations.

3 Impact on the Standard

This proposal is a pure library extension.

4 Design Decisions

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

Unlike the statistics of P1708R6 [1], the percentile and mode (freestanding) functions and (accumulator) objects of this proposal require that ranges be sorted, specifically in ascending order in the case of a percentile. To ensure that users are aware of this fact, the names of the percentile and mode functions (and mode objects) include the word “sorted”. Initially, it was proposed that the names of these functions be prepended by sorted_, which would have resulted in names such as sorted_percentiles. It is believed though that these names would be misleading, as they would suggest that the statistics (percentiles in this example) are sorted, rather than the values of the ranges. Thus, _of_sorted is instead appended to the names of these functions, resulting in names that emphasize the fact that the values of the ranges must be sorted, such as percentiles_of_sorted.

4.2 Percentile Accumulator Objects

Though percentile functions are proposed, percentile objects are not proposed. Objects will most likely maintain a counter corresponding to the position of the current value (of a range). Objects would therefore need to perform a (costly) conditional test for each value of a range in order to determine if the counter corresponds to a particular percentile. This repeated testing results in an unacceptably high run-time complexity. Functions, by comparison, allow one to (quickly) advance an iterator to the computed position of a percentile without having to visit (any) intermediate values. Even better, because the determination of a percentile requires that a range be sorted, a random_access_range will often be used, for which there is direct access (to a computed position).

4.3 Quantiles

A quantile [31] is defined as the value \( x_i \) that divides the (sorted in ascending order) values into intervals of equal size. As an example, the 4-quantiles, or quartiles [32], divide the values into four intervals, bounded by the \( 1/4 = 0.25, 2/4 = 0.5 \) and \( 3/4 = 0.75 \) percentiles. A user wishing to compute the 4-quantiles can readily do so via a percentile function, perhaps

```cpp
std::percentiles_of_sorted(data, std::vector<double>{0.25, 0.5, 0.75}, results.begin());
```

Some might suggest that quantiles be provided as a convenience function, perhaps something of the form

```cpp
std::quantiles_of_sorted(data, 4, results.begin());
```

Unfortunately, if the desired number of quantiles is not known at run-time, then a container, such as a list or vector, must be passed to any function. Along with a container, a user would logically expect to have the ability to specify an allocator, further complicating the situation. For this reason, quantiles are not provided.

5 Technical Specifications

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

5.1 Header <stats> synopsis [stats.syn]

```cpp
#include <execution>

namespace std {

// ... existing accumulator objects ...

// accumulator objects

template<class T>
requires std::equality_comparable<T>
class mode_of_sorted_accumulator;

template<class T, class W = T>
requires std::equality_comparable<T>
class weighted_mode_of_sorted_accumulator;
```
template<\std::weakly_incrementable O, class T>
requires \std::equality_comparable\<T\> \&\& \std::convertible_to\<T, \std::iter_value\_t\<O\>\>
class modes\_of\_sorted\_accumulator;

template<\std::weakly_incrementable O, class T, class W = T>
requires \std::equality_comparable\<T\> \&\& \std::convertible_to\<T, \std::iter_value\_t\<O\>\>
class weighted\_modes\_of\_sorted\_accumulator;

template<class T1, class T2>
requires convertible_to\<T1, T2\> \|\| convertible_to\<T2, T1\>
class covariance\_accumulator;

template<class T1, class T2,>
  class W = \std::conditional\_t\<\std::is\_convertible\_v\<T1, T2\>, T2, T1\>>
requires convertible_to\<T1, T2\> \|\| convertible_to\<T2, T1\>
class weighted\_covariance\_accumulator;

template<class T1, class T2>
requires convertible_to\<T1, T2\> \|\| convertible_to\<T2, T1\>
class correlation\_accumulator;

template<class T1, class T2,>
  class W = \std::conditional\_t\<\std::is\_convertible\_v\<T1, T2\>, T2, T1\>>
requires convertible_to\<T1, T2\> \|\| convertible_to\<T2, T1\>
class weighted\_correlation\_accumulator;

// ... existing accumulator objects accumulation functions ...

// accumulator objects accumulation functions

template<\ranges::input\_range R1, \ranges::input\_range R2, \ranges::input\_range R2, \class ...Accumulators>
constexpr \void stats\_accumulate(\ranges::zip\_view\<R1, R2\>\&\& r, Accumulators\&\&... acc);

template<\ranges::input\_range R1, \ranges::input\_range R2, \ranges::input\_range W, \class ...Accumulators>
constexpr \void stats\_accumulate(
  \ranges::zip\_view\<R1, R2\>\&\& r, W\&\& w, Accumulators\&\&... acc);

template<class ExecutionPolicy, 
  \ranges::input\_range R1, \ranges::input\_range R2, \class ...Accumulators>
requires \std::is\_execution\_policy\_v<\std::remove\_cvref\_t<ExecutionPolicy>>
\void stats\_accumulate( 
  ExecutionPolicy\&\& policy, \ranges::zip\_view\<R1, R2\>\&\& r, Accumulators\&\&... acc);

template<class ExecutionPolicy, 
  \ranges::input\_range R1, \ranges::input\_range R2, \ranges::input\_range W, 
  \class ...Accumulators>
requires \std::is\_execution\_policy\_v<\std::remove\_cvref\_t<ExecutionPolicy>>
\void stats\_accumulate( 
  ExecutionPolicy\&\& policy, \ranges::zip\_view\<R1, R2\>\&\& r, W\&\& w, Accumulators\&\&... acc);

// ... existing freestanding functions ...

// freestanding functions

template<\ranges::sized\_range R>
constexpr auto percentile_of_sorted(R&& r, double p) ->
    ranges::subrange<ranges::iterator_t<R>>;

template<ranges::weakly_incrementable O, ranges::sized_range R, ranges::input_range P>
constexpr auto percentiles_of_sorted(R&& r, P&& p, O it) -> O;

template<ranges::sized_range R>
constexpr auto median_of_sorted(R&& r) -> ranges::subrange<ranges::iterator_t<R>>;

template<ranges::sized_range R, ranges::input_range W>
constexpr auto median_of_sorted(R&& r, W&& w) -> ranges::subrange<ranges::iterator_t<R>>;

template<class ExecutionPolicy, ranges::sized_range R>
requires std::is_execution_policy_v<std::remove_cvref_t<ExecutionPolicy>>
constexpr auto percentile_of_sorted(ExecutionPolicy&& policy, R&& r, double p) ->
    ranges::subrange<ranges::iterator_t<R>>;

template<class ExecutionPolicy, ranges::sized_range R>
requires std::is_execution_policy_v<std::remove_cvref_t<ExecutionPolicy>>
constexpr auto percentiles_of_sorted(ExecutionPolicy&& policy, R&& r, P&& p, O it) -> O;

template<class ExecutionPolicy, ranges::sized_range R>
requires std::is_execution_policy_v<std::remove_cvref_t<ExecutionPolicy>>
constexpr auto median_of_sorted(ExecutionPolicy&& policy, R&& r) ->
    ranges::subrange<ranges::iterator_t<R>>;

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

template<ranges::input_range R>
requires std::equality_comparable<ranges::iter_value_t<R>>
constexpr auto mode_of_sorted(R&& r) -> ranges::iterator_t<R>::value_type;

template<ranges::input_range R, ranges::input_range W>
requires std::equality_comparable<ranges::iter_value_t<R>>
constexpr auto mode_of_sorted(R&& r, W&& w) -> ranges::iterator_t<R>::value_type;

template<ranges::weakly_incrementable O, ranges::input_range R>
requires std::equality_comparable<ranges::iter_value_t<R>> &&
    std::indirectly_copyable<ranges::iterator_t<R>, O>
constexpr auto modes_of_sorted(R&& r, size_t n, O it) -> O;

template<ranges::weakly_incrementable O,
    ranges::input_range R, ranges::input_range W>
requires std::equality_comparable<ranges::iter_value_t<R>> &&
    std::indirectly_copyable<ranges::iterator_t<R>, O>
constexpr auto modes_of_sorted(R&& r, W&& w, size_t n, O it) -> O;

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

template<
class ExecutionPolicy, std::weakly_incrementable O, ranges::input_range R>
requires std::is_execution_policy_v<
std::remove_cvref_t<ExecutionPolicy>> &&
std::equality_comparable<std::iter_value_t<R>> &&
std::indirectly_copyable<ranges::iterator_t<R>, O>
constexpr auto modes_of_sorted(ExecutionPolicy&& policy, R&& r, size_t n, O it) -> O;

template<
class ExecutionPolicy,
std::weakly_incrementable O,
ranges::input_range R,
ranges::input_range W>
requires std::is_execution_policy_v<
std::remove_cvref_t<ExecutionPolicy>> &&
std::equality_comparable<std::iter_value_t<R>> &&
std::indirectly_copyable<ranges::iterator_t<R>, O>
constexpr auto modes_of_sorted(
   ExecutionPolicy&& policy, R&& r, W&& w, size_t n, O it) -> O;

template<
ranges::input_range R1, ranges::input_range R2>
requires convertible_to<R1, R2> || convertible_to<R2, R1>
constexpr auto covariance(ranges::zip_view<R1, R2>&& r,
   typename std::conditional_t<
      std::is_convertible_v<std::iter_value_t<R1>, std::iter_value_t<R2>>,
      std::iter_value_t<R1>, std::iter_value_t<R2>>, ddof) ->
   std::conditional_t<std::is_convertible_v<std::iter_value_t<R1>, std::iter_value_t<R2>>,
   std::iter_value_t<R1>, std::iter_value_t<R2>>,
   std::iter_value_t<R1>, std::iter_value_t<R2>>;

template<
ranges::input_range R1, ranges::input_range R2, ranges::input_range W>
requires convertible_to<R1, R2> || convertible_to<R2, R1>
constexpr auto covariance(ranges::zip_view<R1, R2>&& r, W&& w) ->
   std::conditional_t<std::is_convertible_v<std::iter_value_t<R1>, std::iter_value_t<R2>>,
   std::iter_value_t<R1>, std::iter_value_t<R2>>,
   std::iter_value_t<R1>, std::iter_value_t<R2>>,
   std::iter_value_t<R1>, std::iter_value_t<R2>>;

template<
class ExecutionPolicy, ranges::input_range R1, ranges::input_range R2>
requires std::is_execution_policy_v<
std::remove_cvref_t<ExecutionPolicy>> &&
( convertible_to<R1, R2> || convertible_to<R2, R1>)
constexpr auto covariance(ExecutionPolicy&& policy,
ranges::zip_view<R1, R2>&& r,
   typename std::conditional_t<
      std::is_convertible_v<std::iter_value_t<R1>, std::iter_value_t<R2>>,
      std::iter_value_t<R1>, std::iter_value_t<R2>>,
      std::iter_value_t<R1>, std::iter_value_t<R2>>, ddof) ->
   std::conditional_t<std::is_convertible_v<std::iter_value_t<R1>, std::iter_value_t<R2>>,
   std::iter_value_t<R1>, std::iter_value_t<R2>>,
   std::iter_value_t<R1>, std::iter_value_t<R2>>,
   std::iter_value_t<R1>, std::iter_value_t<R2>>;

template<
class ExecutionPolicy,
ranges::input_range R1, ranges::input_range R2, ranges::input_range W>
requires std::is_execution_policy_v<
std::remove_cvref_t<ExecutionPolicy>> &&
( convertible_to<R1, R2> || convertible_to<R2, R1>)
constexpr auto covariance(
   ExecutionPolicy&& policy, ranges::zip_view<R1, R2>&& r, W&& w) ->
   std::conditional_t<std::is_convertible_v<std::iter_value_t<R1>, std::iter_value_t<R2>>,
   std::iter_value_t<R1>, std::iter_value_t<R2>>,
   std::iter_value_t<R1>, std::iter_value_t<R2>>;
5.2 Accumulator Objects

The accumulator objects specified in this section are trivially copyable. If, first, any or all of the values of \( x \), \( y \) or \( w \) of the `operator()` specified in this section is a NaN (Not a Number), \( \infty \) or \(-\infty \), secondly, NaN, \( \infty \) or \(-\infty \) occurs, or, thirdly, overflow or underflow occurs, which might even occur in the case of finite ranges of values, the function `value` returns an unspecified value.

5.2.1 Mode Accumulator Class Templates

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

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

template<
    std::weakly_incrementable O,
    class T>
requires std::equality_comparable<T> && std::convertible_to<T, std::iter_value_t<O>>
class modes_of_sorted_accumulator
{
public:
    explicit modes_of_sorted_accumulator(size_t n, O it) noexcept;
    constexpr void operator()(const T& x);
    constexpr auto value() -> O;
};

template<
    std::weakly_incrementable O,
    class T, class W = T>
requires std::equality_comparable<T> && std::convertible_to<T, std::iter_value_t<O>>
class weighted_modes_of_sorted_accumulator
{
public:
    explicit weighted_modes_of_sorted_accumulator(size_t n, O it) noexcept;
    constexpr void operator()(const T& x, const W& w);
    constexpr auto value() -> O;
};

explicit mode_of_sorted_accumulator() noexcept;
explicit weighted_mode_of_sorted_accumulator() noexcept;
explicit modes_of_sorted_accumulator(size_t n, O it) noexcept;
explicit weighted_modes_of_sorted_accumulator(size_t n, O it) noexcept;

1. **Preconditions**: it is a valid iterator.
2. **Effects**: A (weighted) mode(s) accumulator object is constructed.
3. **Complexity**: Constant.

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

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

constexpr auto value() const -> T;
constexpr auto value() const -> O;

1. **Preconditions**: The (weighted) values of the associated range r (weighted by the corresponding values of the associated range w) have been accumulated, where r is a sorted in ascending order range that has at least 1 value and the length of r is less than or equal to the length of w.
2. **Effects**: Any remaining computations relating to the (weighted) mode(s) are performed.
3. **Returns**: The (weighted) mode of the values of r (weighted by the corresponding values of w) in the case of mode_of_sorted_accumulator and weighted_mode_of_sorted_accumulator and an output iterator past the last element copied to any n (weighted) modes in any order in the case of modes_of_sorted_accumulator and weighted_modes_of_sorted_accumulator if the preconditions have been met and an unspecified value otherwise.
4. **Complexity**: Constant.
5.2.2 Covariance Accumulator Class Templates

```cpp
template<class T1, class T2>
class covariance_accumulator
{
  public:
    explicit constexpr covariance_accumulator(
      std::conditional_t<std::is_convertible_v<T1, T2>, T2, T1> ddof) noexcept;
    constexpr void operator()(const T1& x, const T2& y);
    constexpr auto value() -> std::conditional_t<std::is_convertible_v<T1, T2>, T2, T1>;
};

template<class T1, class T2, class W = std::conditional_t<std::is_convertible_v<T1, T2>, T2, T1>>
class weighted_covariance_accumulator
{
  public:
    explicit constexpr weighted_covariance_accumulator(
      std::stats_data_kind dkind) noexcept;
    constexpr void operator()(const T1& x, const T2& y, const W& w);
    constexpr auto value() -> std::conditional_t<std::is_convertible_v<T1, T2>, T2, T1>;
};

explicit constexpr covariance_accumulator(
  std::conditional_t<std::is_convertible_v<T1, T2>, T2, T1> ddof) noexcept;
explicit constexpr weighted_covariance_accumulator(
  std::stats_data_kind dkind) noexcept;
```

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

```cpp
constexpr void operator()(const T1& x, const T2& y);
constexpr void operator()(const T1& x, const T2& y, const W& w);
```

1. Effects: The values of $x$ and $y$ (weighted by $w$) are accumulated.
2. Complexity: Constant.

```cpp
std::conditional_t<std::is_convertible_v<T1, T2>, T2, T1>;
```

1. Preconditions: The (weighted) values of the associated range $r$ (weighted by the corresponding values of the associated range $w$) have been accumulated, where $r$ has at least 2 values and the length of $r$ is less than or equal to the length of $w$, and $ddof$ is not equal to $ranges::distance(r)$.
2. Effects: Any remaining computations relating to the (weighted) covariance are performed.
3. Returns: The (weighted) covariance of the values of $r$ (weighted by the corresponding values of $w$) if the preconditions have been met and an unspecified value otherwise.
5.2.3 Correlation Accumulator Class Templates

```cpp
template<class T1, class T2>
requires convertible_to<T1, T2> || convertible_to<T2, T1>
class correlation_accumulator
{
public:
    explicit constexpr correlation_accumulator(
        std::conditional_t<std::is_convertible_v<T1, T2>, T2, T1> ddof) noexcept;
    constexpr void operator()(const T1& x, const T2& y);
    constexpr auto value() -> std::conditional_t<std::is_convertible_v<T1, T2>, T2, T1>;
};

template<class T1, class T2,
    class W = std::conditional_t<std::is_convertible_v<T1, T2>, T2, T1>>
requires convertible_to<T1, T2> || convertible_to<T2, T1>
class weighted_correlation_accumulator
{
public:
    explicit constexpr weighted_correlation_accumulator(
        std::stats_data_kind dkind) noexcept;
    constexpr void operator()(const T1& x, const T2& y, const W& w);
    constexpr auto value() -> std::conditional_t<std::is_convertible_v<T1, T2>, T2, T1>;
};
```

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

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

1. **Effects:** The values of x and y (weighted by w) are accumulated.
2. **Complexity:** Constant.

```cpp
constexpr auto value() -> std::conditional_t<std::is_convertible_v<T1, T2>, T2, T1>;
```

1. **Preconditions:** The (weighted) values of the associated range r (weighted by the corresponding values of the associated range w) have been accumulated, where r has at least 2 values and the length of r is less than or equal to the length of w, and ddof is not equal to ranges::distance(r).
2. **Effects:** Any remaining computations relating to the (weighted) correlation are performed.
3. **Returns:** The (weighted) correlation of the values of r (weighted by the corresponding values of w) if the preconditions have been met and an unspecified value otherwise.
4. **Complexity:** Constant.
5.2.4 Accumulator Objects Accumulation Functions

```cpp
template<ranges::input_range R1, ranges::input_range R2, class ...Accumulators>
constexpr void stats_accumulate(ranges::zip_view<R1, R2>&& r, Accumulators&&... acc);

template<ranges::input_range R1, ranges::input_range R2, ranges::input_range W, class ...Accumulators>
constexpr void stats_accumulate(
    ranges::zip_view<R1, R2>&& r, W&& w, Accumulators&&... acc);

template<class ExecutionPolicy,
    ranges::input_range R1, ranges::input_range R2, class ...Accumulators>
requires std::is_execution_policy_v<std::remove_cvref_t<ExecutionPolicy>>
void stats_accumulate(
    ExecutionPolicy&& policy, ranges::zip_view<R1, R2>&& r, Accumulators&&... acc);

template<class ExecutionPolicy,
    ranges::input_range R1, ranges::input_range R2, ranges::input_range W,
    class ...Accumulators>
requires std::is_execution_policy_v<std::remove_cvref_t<ExecutionPolicy>>
void stats_accumulate(
    ExecutionPolicy&& policy, ranges::zip_view<R1, R2>&& r, W&& w, Accumulators&&... acc);
```

1. ** Preconditions:** \( r \) and \( w \) are ranges of finite values, where \( r \) has at least 2 values, the length of \( r \) is less than or equal to the length of \( w \) and \( \text{acc} \) are valid accumulator objects.
2. ** Effects:** The (weighted) statistics of the accumulator objects \( \text{acc} \) over the values of \( r \) (weighted by the corresponding values of the associated range \( w \)) are computed.
3. ** Complexity:** Linear in \( \text{ranges::distance}(r) \).

The preconditions of the Accumulator Objects Accumulation Functions section of P1708R6 [1] is to be updated to be:

** Preconditions:** \( r \) and \( w \) are ranges of finite values, sorted if any of the accumulator objects of \( \text{acc} \) is a mode accumulator object, where the length of \( r \) is less than or equal to the length of \( w \). \( r \) has at least 4 values if any of the accumulator objects of \( \text{acc} \) is a kurtosis accumulator object, \( r \) has at least 3 values if any of the accumulator objects of \( \text{acc} \) is a skewness accumulator object and \( r \) has at least 1 value otherwise, and \( \text{acc} \) are valid accumulator objects.

5.3 Freestanding Functions

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

5.3.1 Freestanding Percentile Functions

```cpp
template<ranges::sized_range R>
constexpr auto percentile_of_sorted(R&& r, double p) ->
    ranges::subrange<ranges::iterator_t<R>>;

template<std::weakly_incrementable O, ranges::sized_range R, ranges::input_range P>
constexpr auto percentiles_of_sorted(R&& r, P&& p, O it) -> O;

template<ranges::sized_range R>
constexpr auto median_of_sorted(R&& r) -> ranges::subrange<ranges::iterator_t<R>>;

template<ranges::sized_range R, ranges::input_range W>
constexpr auto median_of_sorted(R&& r, W&& w) -> ranges::subrange<ranges::iterator_t<R>>;
```
template<class ExecutionPolicy, ranges::sized_range R>
requires std::is_execution_policy_v<std::remove_cvref_t<ExecutionPolicy>>
constexpr auto percentile_of_sorted(ExecutionPolicy&& policy, R&& r, double p) ->
ranges::subrange<ranges::iterator_t<R>>;

template<class ExecutionPolicy, std::weakly_incrementable O, ranges::sized_range R, ranges::input_range P>
requires std::is_execution_policy_v<std::remove_cvref_t<ExecutionPolicy>>
constexpr auto percentiles_of_sorted(ExecutionPolicy&& policy, R&& r, P&& p, O it) -> O;

template<class ExecutionPolicy, ranges::sized_range R>
requires std::is_execution_policy_v<std::remove_cvref_t<ExecutionPolicy>>
constexpr auto median_of_sorted(ExecutionPolicy&& policy, R&& r) ->
ranges::subrange<ranges::iterator_t<R>>;

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

1. **Preconditions:** $r$ and $w$ are ranges of finite values, where $r$ has at least 1 value and the length of $r$ is less than or equal to the length of $w$ and it is a valid iterator.

2. **Returns:** The (weighted) percentile(s) of the values of $r$ (weighted by the corresponding values of the associated range $w$) if the preconditions have been met and an unspecified value otherwise.

3. **Complexity:** Linear in $n$ if $r$ (and $w$) is or is derived from `random_access_range` and linear in `ranges::distance(r)` otherwise.

### 5.3.2 Freestanding Mode Functions

```cpp
template<ranges::input_range R>
requires std::equality_comparable<std::iter_value_t<R>>
constexpr auto mode_of_sorted(R&& r) -> ranges::iterator_t<R>::value_type;

template<ranges::input_range R, ranges::input_range W>
requires std::equality_comparable<std::iter_value_t<R>>
constexpr auto mode_of_sorted(R&& r, W&& w) -> ranges::iterator_t<R>::value_type;

template<std::weakly_incrementable O, ranges::input_range R>
requires std::equality_comparable<std::iter_value_t<R>> &&
std::indirectly_copyable<ranges::iterator_t<R>, O>
constexpr auto modes_of_sorted(R&& r, size_t n, O it) -> O;

template<std::weakly_incrementable O, ranges::input_range R, ranges::input_range W>
requires std::equality_comparable<std::iter_value_t<R>> &&
std::indirectly_copyable<ranges::iterator_t<R>, O>
constexpr auto modes_of_sorted(R&& r, W&& w, size_t n, O it) -> O;

template<class ExecutionPolicy, ranges::input_range R>
requires std::is_execution_policy_v<std::remove_cvref_t<ExecutionPolicy>> &&
std::equality_comparable<std::iter_value_t<R>>
constexpr auto mode_of_sorted(ExecutionPolicy&& policy, R&& r) ->
ranges::iterator_t<R>::value_type;
```
1. **Preconditions**: \( r \) and \( w \) are ranges of finite values, where \( r \) is a sorted in ascending order range that has at least 1 value and the length of \( r \) is less than or equal to the length of \( w \) and it is a valid iterator.

2. **Returns**: The (weighted) mode of the values of the associated range \( r \) (weighted by the corresponding values of the associated range \( w \)) in the case of \texttt{mode_of_sorted_accumulator} and \texttt{weighted_mode_of_sorted_accumulator} and an output iterator past the last element copied to any \( n \) (weighted) modes in any order in the case of \texttt{modes_of_sorted_accumulator} and \texttt{weighted_modes_of_sorted_accumulator} if the preconditions have been met and an unspecified value otherwise.

3. **Complexity**: Linear in \texttt{ranges::distance(r)}.

### 5.3.3 Freestanding Covariance Functions

```cpp
template<ranges::input_range R1, ranges::input_range R2>
requires convertible_to<R1, R2> || convertible_to<R2, R1>
constexpr auto covariance(ranges::zip_view<R1, R2>&& r,
    typename std::conditional_t<
        std::is_convertible_v<std::iter_value_t<R1>, std::iter_value_t<R2>>, std::iter_value_t<R1>, std::iter_value_t<R2>>, std::iter_value_t<R1>, std::iter_value_t<R2>>;

template<ranges::input_range R1, ranges::input_range R2, ranges::input_range W>
requires convertible_to<R1, R2> || convertible_to<R2, R1>
constexpr auto covariance(ranges::zip_view<R1, R2>&& r, W&& w) ->
    std::conditional_t<
        std::is_convertible_v<std::iter_value_t<R1>, std::iter_value_t<R2>>, std::iter_value_t<R1>, std::iter_value_t<R2>>, std::iter_value_t<R1>, std::iter_value_t<R2>>;

template<class ExecutionPolicy, ranges::input_range R1, ranges::input_range R2>
requires std::is_execution_policy_v<std::remove_cvref_t<ExecutionPolicy>> &&
    (convertible_to<R1, R2> || convertible_to<R2, R1>)
constexpr auto covariance(ExecutionPolicy&& policy,
    ranges::zip_view<R1, R2>&& r,
    typename std::conditional_t<
        std::is_convertible_v<std::iter_value_t<R1>, std::iter_value_t<R2>>, std::iter_value_t<R1>, std::iter_value_t<R2>>,
    std::iter_value_t<R1>, std::iter_value_t<R2>);```
std::iter_value_t<R1>, std::iter_value_t<R2>> ddof) ->
std::conditional_t<std::is_convertible_v<std::iter_value_t<R1>, std::iter_value_t<R2>>, std::iter_value_t<R1>, std::iter_value_t<R2>>;

template<class ExecutionPolicy,
ranges::input_range R1, ranges::input_range R2, ranges::input_range W>
requires std::is_execution_policy_v<std::remove_cvref_t<ExecutionPolicy>> &&
(convertible_to<R1, R2> || convertible_to<R2, R1>)
constexpr auto covariance(
  ExecutionPolicy&& policy, ranges::zip_view<R1, R2>&& r, W&& w) ->
std::conditional_t<std::is_convertible_v<std::iter_value_t<R1>, std::iter_value_t<R2>>, std::iter_value_t<R1>, std::iter_value_t<R2>>;

1. **Preconditions:** $r$ and $w$ are ranges of finite values, where $r$ has at least 2 values, the length of $r$ is less than or equal to the length of $w$ and ddof is not equal to ranges::distance($r$).

2. **Returns:** The (weighted) covariance of the values of $r$ (weighted by the corresponding values of the associated range $w$) if the preconditions have been met and an unspecified value otherwise.

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

### 5.3.4 Freestanding Correlation Functions

```cpp
template<ranges::input_range R1, ranges::input_range R2>
requires convertible_to<R1, R2> || convertible_to<R2, R1>
constexpr auto correlation(ranges::zip_view<R1, R2>&& r, 
  typename std::conditional_t<
    std::is_convertible_v<std::iter_value_t<R1>, std::iter_value_t<R2>>, std::iter_value_t<R1>, std::iter_value_t<R2>> ddof) ->
  std::conditional_t<std::is_convertible_v<std::iter_value_t<R1>, std::iter_value_t<R2>>, std::iter_value_t<R1>, std::iter_value_t<R2>>;

template<ranges::input_range R1, ranges::input_range R2, ranges::input_range W>
requires convertible_to<R1, R2> || convertible_to<R2, R1>
constexpr auto correlation(ranges::zip_view<R1, R2>&& r, W&& w) ->
  std::conditional_t<std::is_convertible_v<std::iter_value_t<R1>, std::iter_value_t<R2>>, std::iter_value_t<R1>, std::iter_value_t<R2>>;

template<class ExecutionPolicy, ranges::input_range R1, ranges::input_range R2>
requires std::is_execution_policy_v<std::remove_cvref_t<ExecutionPolicy>> &&
  (convertible_to<R1, R2> || convertible_to<R2, R1>)
constexpr auto correlation(ExecutionPolicy&& policy, 
  ranges::zip_view<R1, R2>&& r, 
  typename std::conditional_t<
    std::is_convertible_v<std::iter_value_t<R1>, std::iter_value_t<R2>>, std::iter_value_t<R1>, std::iter_value_t<R2>> ddof) ->
  std::conditional_t<std::is_convertible_v<std::iter_value_t<R1>, std::iter_value_t<R2>>, std::iter_value_t<R1>, std::iter_value_t<R2>>;

template<class ExecutionPolicy, 
ranges::input_range R1, ranges::input_range R2, ranges::input_range W>
requires std::is_execution_policy_v<std::remove_cvref_t<ExecutionPolicy>> &&
  (convertible_to<R1, R2> || convertible_to<R2, R1>)
constexpr auto correlation(ExecutionPolicy&& policy, 
  ranges::zip_view<R1, R2>&& r, W&& w) ->
  std::conditional_t<std::is_convertible_v<std::iter_value_t<R1>, std::iter_value_t<R2>>, std::iter_value_t<R1>, std::iter_value_t<R2>>;
```
1. **Preconditions**: \( r \) and \( w \) are ranges of finite values, where \( r \) has at least 2 values, the length of \( r \) is less than or equal to the length of \( w \) and \( ddof \) is not equal to \( \text{ranges::distance}(r) \).

2. **Returns**: The (weighted) correlation of the values of \( r \) (weighted by the corresponding values of the associated range \( w \)) if the preconditions have been met and an unspecified value otherwise.

3. **Complexity**: Linear in \( \text{ranges::distance}(r) \).

6 **Acknowledgements**

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

**References**


## Appendix A Examples

The following example showcases the use of **percentile** functions.

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

std::vector<std::ranges::subrange<std::vector<double>::iterator>> percentiles(3); 
std::percentiles_of_sorted(v, std::vector<double>{0.25, 0.5, 0.75}, percentiles.begin()); 
for (auto p : percentiles) 
    std::cout << (*p.begin() + *std::prev(p.end())) / 
              std::distance(p.begin(), p.end()) << " ";
```

The following example showcases the use of a **mode** function.

```cpp
std::string text = "throughput"; 
std::ranges::sort(text); 

std::vector<char> modes(text.size()); 
auto s_end = std::modes_of_sorted(text, modes.size(), modes.begin()); 

std::cout << "modes = " ;
for (auto it = modes.cbegin(); it != s_end; ++it) 
    std::cout << *it << " " ;
```

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

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

std::vector<int> modes(4); 
std::mode_of_sorted_accumulator<int> m1; 
std::modes_of_sorted_accumulator<int, std::vector<int>::iterator> m2(4, modes.begin()); 

std::stats_accumulate(L, m1, m2); 

std::cout << "mode = " << m1.value(); 
std::cout << "\nmodes = " ;
auto end = m2.value(); 
for (auto i = modes.cbegin(); i != end; i++) 
    std::cout << *i << " ";
```

The following example showcases the use of **covariance** and **correlation** functions.
The following example showcases the use of **covariance**, **correlation** and **custom** accumulator objects.

```cpp
/* custom accumulator */
class mean_squared_error_accumulator
{
public:
    constexpr mean_squared_error_accumulator() noexcept { MSE_=0; n_=0; }
    constexpr void operator() (double x, double y) { MSE_ += (x-y) * (x-y); n_++; }
    double value() { return std::sqrt(MSE_/n_); }
private:
    double MSE_; 
    size_t n_; }

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

std::cout << "covariance = " << std::covariance(std::views::zip(v1, v2), 0); 
std::cout << "\ncorrelation = " << std::correlation(std::views::zip(v1, v2), 0); 

std::list<double> L1 = { 1, 2, 2, 3, 3 }, L2 = { 4, 7, 8, 9, 9 }; 

std::covariance_accumulator<double> covar(0); 
std::correlation_accumulator<double> corr(0); 
mean_squared_error_accumulator mse; 

std::stats_accumulate(std::views::zip(L1, L2), covar, corr, mse); 
std::cout << "covariance = " << covar.value(); 
std::cout << "\ncorrelation = " << corr.value(); 
std::cout << "\nMSE = " << mse.value(); 
```