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

R2681R0

- Approved by SG19 on October 13, 2022 by a 6/1/0/0 vote.

R2681R1

- **Linear regression** is deferred to a future proposal, given that it is a part of a larger family of regression models.

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 freestsanding **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 [20] 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 $100p\%$ 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

$$\sigma_{xy} = \frac{1}{n} \sum_{i=1}^{n} (x_i - \mu_x)(y_i - \mu_y), \tag{4}$$

where μ_x and μ_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

$$s_{xy} = \frac{1}{n-1} \sum_{i=1}^{n} (x_i - \bar{x})(y_i - \bar{y}), \tag{5}$$

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

$$\hat{\sigma}_{xy} = \frac{1}{\sum_{i=1}^{n} w_i} \sum_{i=1}^{n} w_i \left(x_i - \mu_x^*\right)\left(y_i - \mu_y^*\right) \tag{6}$$

where μ_x^* and μ_y^* are the **weighted population** means [28, 29]. The **weighted sample** covariance is defined as

$$\hat{s}_{xy} = \frac{\sum_{i=1}^{n} w_i}{\left(\sum_{i=1}^{n} w_i\right)^2 - \sum_{i=1}^{n} w_i^2} \sum_{i=1}^{n} w_i \left(x_i - \bar{x}^*\right)\left(y_i - \bar{y}^*\right), \tag{7}$$

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

$$\rho_{xy} = \frac{\sigma_{xy}}{\sigma_x \sigma_y}, \tag{8}$$

where σ_x and σ_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

$$r_{xy} = \frac{s_{xy}}{s_x s_y}, \tag{9}$$

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

$$\hat{\rho}_{xy} = \frac{\hat{\sigma}_{xy}}{\hat{\sigma}_x \hat{\sigma}_y}, \tag{10}$$

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

$$\hat{\rho}_{xy} = \frac{\hat{s}_{xy}}{\hat{s}_x \hat{s}_y}, \tag{11}$$

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

```
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

```
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]

```
#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, 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<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>>;

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;
```

```
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>>;

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>>;

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>> 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>>;
```

```
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>>;
}
```

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), ∞ or $-\infty$, secondly, NaN, ∞ 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_accumul` 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

```
template<class T1, class T2>
requires convertible_to<T1, T2> || convertible_to<T2, T1>
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>>
requires convertible_to<T1, T2> || convertible_to<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.

```
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.

```
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.
4. *Complexity*: Constant.

5.2.3 Correlation Accumulator Class Templates

```
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>;
};
```

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

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

```
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.

```
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

```
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 `acc` are valid accumulator objects.
2. *Effects*: The (weighted) statistics of the accumulator objects `acc` over the values of `r` (weighted by the corresponding values of the associated range `w`) are computed.
3. *Complexity*: Linear in `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 `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 `acc` is a kurtosis accumulator object, `r` has at least 3 values if any of the accumulator objects of `acc` is a skewness accumulator object and `r` has at least 1 value otherwise, and `acc` are valid accumulator objects.

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, ∞ or $-\infty$, secondly, NaN, ∞ 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

```
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

```
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;
```

```
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;
```

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 `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.

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

5.3.3 Freestanding Covariance Functions

```
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>>;

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>>;

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>> 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

```
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 `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 `ranges::distance(r)`.

6 Acknowledgements

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

References

[1] Richard Dosselmann et al. P1708R6 simple statistics. ISO JTC1/SC22/WG21: Programming Language C++, accessed 31 Jul. 2022. https://open-std.org/JTC1/SC22/WG21/docs/papers/2022/p1708r6.pdf.

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

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

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

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

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

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

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

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

[10] The SURVEYMEANS procedure. sas, accessed 11 Jun. 2020. https://support.sas.com/documentation/cdl/en/statug/65328/HTML/default/viewer.htm#statug_surveymeans_details06.htm.

[11] Statistical functions. IBM, accessed 28 Aug. 2020. https://www.ibm.com/support/knowledgecenter/SSLVMB_sub/statistics_reference_project_ddita/spss/base/syn_transformation_expressions_statistical_functions.html.

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

[13] Paul Bristow. A proposal to add mathematical functions for statistics to the C++ standard library. JTC 1/SC22/WG14/N1069, WG21/N1668, accessed 12 Jun. 2020. http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1069.pdf.

[14] Walter E. Brown et al. Random number generation in C++0X: A comprehensive proposal, version2. WG21/N2032 = J16/06/0102, accessed 13 Jun. 2020. www.open-std.org/jtc1/sc22/wg21/docs/papers/2006/n2032.pdf.

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

[16] Nikhar Agrawal et al. Chapter 5. Statistical distributions and functions, Boost: C++ libraries, accessed 12 Jun. 2020. https://www.boost.org/doc/libs/1_73_0/libs/math/doc/html/dist.html.

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

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

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

[20] Mike Allen. *The SAGE Encyclopedia of Communication Research Methods*, volume 1. SAGE Publications, 2017.

[21] Amar Sahay. *Essentials of Data Science and Analytics: Statistical Tools, Machine Learning, and R-Statistical Software Overview*. Business Expert Press, 2021.

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

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

[24] Percentile. Wikipedia.com, accessed 31 Jul. 2022.
`https://en.wikipedia.org/wiki/Percentile#Weighted_percentile`.

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

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

[27] Weighted correlation weighted covariance weighted cosine distance weighted cosine similarity. NIST, accessed 30 May 2022.
`https://www.itl.nist.gov/div898/software/dataplot/refman2/auxillar/weigcorr.htm`.

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

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

[30] Sarah Boslaugh. *Statistics in a Nutshell*. O'Reilly, 2013.

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

[32] Gregory J. Privitera and Darryl J. Mayeaux. *Core Statistical Concepts With Excel: An Interactive Modular Approach*. SAGE Publications, 2018.

Appendix A Examples

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

```
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.

```
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.

```
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.

```
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);
```

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

```
/* 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::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();
```