Basic Statistics

Document Number:	P1708R7
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:	SG19, LEWG

Contents

0 Revision History

P1708R1

- ◦ An accumulator object is proposed to allow for the computation of statistics in a **single** pass over a sequence of values.

P1708R2

- ◦ Reformatted using LaTeX.
- ◦ A (possible) return to freestanding functions is proposed following discussions of the accumulator object of the previous version.

P1708R3

- ◦ **Geometric mean** is proposed, since it exists in Calc, Excel, Julia, MATLAB, Python, R and Rust.
- ◦ **Harmonic mean** is proposed, since it exists in Calc, Excel, Julia, MATLAB, PHP, Python, R and Rust.
- ◦ **Weighted means**, **median**, **mode**, **variances** and **standard deviations** are proposed, since they exist (with the exception of mode) in MATLAB and R.
- ◦ **Quantile** is proposed, since it is more generic than median and exists in Calc (percentile), Excel (percentile), Julia, MATLAB, PHP (percentile), R and SQL (percentile).
- ◦ **Skewness** is proposed, since it exists in Calc, Excel, Julia, MATLAB, PHP, R, Rust, SAS and SQL and was recommended as part of a presentation to SAS corporation.
- ◦ **Kurtosis** is proposed, since it exists in Calc, Excel, Julia, MATLAB, PHP, R, Rust, SAS and SQL and was recommended as part of a presentation to SAS corporation.
- ◦ Both **freestanding functions** and **accumulator objects** are proposed, since they (largely) have distinct purposes.
- ◦ **Iterator pairs** are replaced by **ranges**, since ranges simplify predicates (as comparisons and projections).

P1708R4

- Parameter `data_t` (corresponding to values `population_t` and `sample_t`) of **variance** and **standard deviation** are replaced by **delta degrees of freedom**, since this is done in Python (NumPy).
- In the case of a **quantile** (or median), specific methods of interpolation between adjacent values is proposed, since this is done in Python (NumPy).
- `stats_error`, previously a constant, is replaced by a **class**.

P1708R5

- **Quantile** (and **median**) and **mode** are deferred to a future proposal, given ongoing unresolved issues relating to these statistics.
- `stats_error`, an **exception**, is removed, since (C++) math funtions do not throw exceptions.
- The ability to create **custom** accumulator objects is proposed, since this is done in Boost Accumulators.
- `stats_result_t` is introduced so as to simplify (function) signatures.
- Various errors in statistical formulas are corrected.
- Various functions, objects (classes) and parameters are renamed so as to be more meaningful.
- Various technical errors relating to ranges and execution policy are corrected.

P1708R6

- `stats_result_t` is removed, since return type is deduced from projection.
- **Accumulator** objects are revised so as to be simpler and allow for parallel implementations.
- `stat_accum` and `weighted_stat_accum` are removed, since they are no longer needed.
- **Concepts** are removed so as to allow for **custom** data types.
- **Projections** are removed, since **views** already offer such functionality.
- Numerous functions and classes are renamed so as to be more meaningful.
- Reformatted so as to fulfill the specification style guidelines and **standardese**.

P1708R7

- **Unweighted** and **weighted** functions are combined so as to take advantage of **overloading**.
- The presentation of formulas is simplified.
- **Derivations** of skewness and kurtosis formulas are given.
- The wording of the technical specifications is updated.
- Further reformatted so as to fulfill the specification style guidelines and **standardese**.

1 Introduction

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

2 Motivation and Scope

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

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

Five statistics are defined in this proposal. Two (univariate) statistics, specifically **percentile** (and **median**) and **mode**, are **not** included in this proposal. These more involved statistics are 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 Mean

The *arithmetic mean* [20, 21] of the values $x_1, x_2, \ldots, x_n$ ($n \geq 1$), denoted μ or $\bar{x}$ in the case of a **population** [20] or **sample** [20], respectively, is defined as

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

The *weighted* arithmetic mean [22, 21], for weights $w_1, w_2, \ldots, w_n$, denoted μ^* or $\bar{x}^*$ in the case of a **population** or **sample**, respectively, is defined as

$$\frac{1}{V_1}\sum_{i=1}^{n} w_i x_i, \tag{2}$$

where $V_1 = \sum_{i=1}^{n} w_i$. The *geometric* mean [20] is defined as

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

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

$$\left(\prod_{i=1}^{n} x_i^{w_i}\right)^{V_1^{-1}}. \tag{4}$$

The *harmonic* mean [20] of the positive values $x_i > 0$ is defined as

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

and the **weighted** harmonic mean [23] is defined as

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

Each of the arithmetic, geometric and harmonic means can be (accurately) computed in **linear** time [24]. When computing the associated sums of these means, and indeed any sum in this proposal, robust methods [25, 26] ought to be considered.

2.2 Skewness

The **population** *skewness* [20, 21], a measure of the **symmetry** [20] of the values ($n \geq 3$), is defined as

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

and the **sample** skewness [20, 21] is defined as

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

where σ and s (and $\hat{\sigma}$) are defined in Section 2.5. The **weighted population** skewness [21] is defined as

$$\frac{1}{V_1\hat{\sigma}^3}\sum_{i=1}^{n} w_i (x_i - \mu^*)^3 \tag{9}$$

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

$$\frac{\left(V_1^2 - V_2\right)^{3/2}}{V_1\left(V_1^3 - 3V_1V_2 + 2V_3\right)\hat{\sigma}^3}\sum_{i=1}^{n} w_i(x_i - \bar{x}^*)^3, \tag{10}$$

where $V_2 = \sum_{i=1}^{n} w_i^2$ and $V_3 = \sum_{i=1}^{n} w_i^3$. Skewness (and kurtosis) can be computed in **linear** time [24, 27]. When scaled by the factor

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

skewness becomes the *adjusted Fisher-Pearson standardized moment coefficient* [28]. Derivations of Equations (7), (8), (9) and (10) are given in Appendix B.1.

2.3 Kurtosis

The *Pearson* [29] (non-excess) **population** *kurtosis* [21, 30, 31], a measure of the "**tailedness**" [31] of the values ($n \geq 4$), is defined as

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

and the Pearson **sample** kurtosis [21] is defined as

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

The **weighted** Pearson **population** kurtosis [21] is defined as

$$\frac{1}{V_1\hat{\sigma}^4}\sum_{i=1}^{n}w_i\left(x_i-\mu^*\right)^4 \tag{14}$$

and the **weighted** Pearson **sample** kurtosis [21] is defined as

$$\frac{\left(V_1^2-V_2\right)\left(V_1^4-3V_1^2V_2+2V_1V_3+3V_2^2-3V_4\right)}{V_1^3\left(V_1^4-6V_1^2V_2+8V_1V_3+3V_2^2-6V_4\right)\hat{\sigma}^4}\sum_{i=1}^{n}w_i(x_i-\bar{x}^*)^4-\frac{3\left(V_1^2-V_2\right)\left(2V_1^2V_2-2V_1V_3-3V_2^2+3V_4\right)}{V_1^2\left(V_1^4-6V_1^2V_2+8V_1V_3+3V_2^2-6V_4\right)}, \tag{15}$$

where $V_4=\sum_{i=1}^{n}w_i^4$. The *Fisher* [29] or *excess* [30] **population** kurtosis is defined as

$$\frac{1}{n\sigma^4}\sum_{i=1}^{n}\left(x_i-\mu\right)^4-3 \tag{16}$$

and the excess **sample** kurtosis [21] is defined as

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

The **weighted** excess **population** kurtosis [21] is defined as

$$\frac{1}{V_1\hat{\sigma}^4}\sum_{i=1}^{n}w_i\left(x_i-\mu^*\right)^4-3 \tag{18}$$

and the **weighted** excess **sample** kurtosis [21] is defined as

$$\frac{\left(V_1^2-V_2\right)\left(V_1^4-4V_1V_3+3V_2^2\right)}{V_1^3\left(V_1^4-6V_1^2V_2+8V_1V_3+3V_2^2-6V_4\right)\hat{\sigma}^4}\sum_{i=1}^{n}w_i(x_i-\bar{x}^*)^4-\frac{3\left(V_1^2-V_2\right)\left(V_1^4-2V_1^2V_2+4V_1V_3-3V_2^2\right)}{V_1^2\left(V_1^4-6V_1^2V_2+8V_1V_3+3V_2^2-6V_4\right)}. \tag{19}$$

Derivations of Equations (12), (13), (14), (15), (16), (17), (18) and (19) are given in Appendix B.2.

2.4 Variance

The **population** *variance* [20, 21] of the values ($n \geq 1$), denoted σ^2, is defined as

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

and the **sample** variance [20, 21] ($n \geq 2$), denoted s^2, is defined as

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

The **weighted population** variance [21, 32], denoted $\hat{\sigma}^2$, is defined as

$$\frac{1}{V_1}\sum_{i=1}^{n}w_i\left(x_i-\mu^*\right)^2 \tag{22}$$

and the **weighted sample** variance [21, 32], also used in the case of *reliability weights* [33], denoted $\hat{s}^2$, is defined as

$$\frac{V_1}{V_1^2 - V_2} \sum_{i=1}^{n} w_i \left(x_i - \bar{x}^*\right)^2 . \tag{23}$$

Population and sample variance (and standard deviation) are computed using factors of $1/n$ and $1/(n-1)$, respectively. Other factors might be used instead, $1/(n-1.5)$ as an example [34, 35]. To allow for such factors, this proposal, like NumPy [36], enables one to specify *delta degrees of freedom* [36], a value subtracted from n. Variance (and standard deviation) can be computed in **linear** time [24, 37, 38].

2.5 Standard Deviation

The **population** *standard deviation* [20] of the values ($n \geq 1$), denoted σ, is defined as the square root of the population variance. The **sample** standard deviation [20] ($n \geq 2$), denoted s, is defined as the square root of the sample variance. Likewise, the **weighted population** standard deviation, denoted $\hat{\sigma}$, is defined as the square root of the weighted population variance and the **weighted sample** standard deviation [39], denoted $\hat{s}$, is defined as the square root of the weighted sample variance.

3 Impact on the Standard

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

4 Design Decisions

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

4.1 Freestanding Functions vs. Accumulator Objects

Perhaps the most significant concern stemming from this proposal is that of (freestanding) functions versus (accumulator) objects. In the first incarnation of this proposal, namely P1708R0, functions were (exclusively) proposed. **Functions** are useful when one wishes to compute a **single** statistic. Objects were introduced in P1708R1 and P1708R2, which allow a user to (efficiently) compute **more than one** statistic in a **single** pass over the values, an idea borrowed from Boost Accumulators [40]. Like Boost Accumulators, a programmer has the ability to create **custom** objects. Given that each of these two paradigms has merit, with functions again being most useful in the case of the computation of a single statistic and objects being more attractive in instances in which multiple statistics are computed, the decision has been made to incorporate **both** such models into this proposal. Users are thus able to choose the approach that best fits with their design rather than being forced to use one of two paradigms. Note that objects are declared prior to functions in Section 5, as some vendors might wish to implement functions using objects.

4.2 Overloaded Accumulator Objects

It has been suggested that unweighted and weighted variants of an object be combined into a single object. This combined object would have two (overloaded) functions `operator()`. This might be feasible in the some cases, mean for instance. Such an object could take on the form

```
template<class T, class W = T>
class mean_accumulator
{
public:
  explicit constexpr mean_accumulator() noexcept;
  constexpr void operator()(const T& x);              // unweighted variant
  constexpr void operator()(const T& x, const W& w); // weighted variant
  constexpr void unweighted_finalize();               // unweighted variant
  constexpr void weighted_finalize();                 // weighted variant
  constexpr auto value() -> T;
};
```

Note the need for (two) functions `unweighted_finalize` and `weighted_finalize` to perform the correct post-processing following accumulation, functionality that is presently merged into the (single) function `value`. Strictly unweighted or weighted **custom** objects would not require two functions `operator()` and two post-processing functions, another potential point of confusion.

Moving on, combined objects are less intuitive in the case of variance and standard deviation, in which population and sample variants exist, namely

```
template<class T, class W = T>
class variance_accumulator
{
public:
  // unweighted variant uses ddof
  explicit constexpr variance_accumulator(T ddof) noexcept;

  // weighted variant uses enumerated value
  explicit constexpr variance_accumulator(std::stats_data_kind dkind) noexcept;

  constexpr void operator()(const T& x);              // unweighted variant
  constexpr void operator()(const T& x, const W& w); // weighted variant
  constexpr void unweighted_finalize();              // unweighted variant
  constexpr void weighted_finalize();                // weighted variant
  constexpr auto value() -> T;
};
```

Delta degrees of freedom (`ddof`) distinguish population and sample variants in the unweighted case, whereas enumerated values are used in the weighted case. As a result, these objects require multiple constructors, of which the correct one must be invoked in order to compute the desired statistic.

4.3 Trimmed Mean

The issue of a trimmed mean is raised in [41]. A $p\%$ *trimmed* mean [42] is one in which each of the $(p/2)\%$ **highest** and **lowest** values (of a **sorted** range) are excluded from the computation of that mean. This feature would require that the values of a given range either be **presorted** or **sorted** as part of the computation of a mean. As an author, Phillip Ratzloff feels (a sentiment that was echoed by the author of [41]) that one might handle this (and other similar) matter via **ranges**, specifically by using a statement of the form

```
auto m = data | std::ranges::sort | trim(p) | std::mean;
```

4.4 Special Values

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

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

4.5 Projections

The functions and objects of P1708R3, P1708R4 and P1708R5 employ projections as a means of accessing individual components of aggregate entities. Given that such functionality is available through the use of **views**, projections have been removed, thereby yielding simpler functions and objects. This is much like the approach suggested in Sections 4.3 and 4.4. An example that demonstrates the use of views is presented in Appendix A.

4.6 Concepts

Much like `std::complex`, the proposed (template) functions and objects are defined for each of the (C++) **arithmetic** types, **except** for **`bool`**. Also like `std::complex`, the effect of instantiating the templates for any other type is unspecified. A programmer can therefore attempt to use **custom** types with the proposed functions and objects. It is felt that the added flexibility afforded by not using **concepts** to strictly limit functions and objects to arithmetic types is in the interest of the C++ community. In fact, several concerned parties reached out to the authors of this proposal in regards to this matter, all of whom suggested that this flexible approach be taken. Note that concepts are still employed in the case of **execution policy**, namely `std::is_execution_policy_v`, in which a fixed set of policies exists.

4.7 Header and Namespace

Early versions of this proposal, specifically P1708R0, P1708R1 and P17082, request that the proposed functions and objects be placed into the `<numeric>` header. Since P1708R3, it has instead been suggested that the function and objects be placed into a (new) **header** `<stats>`, just as was done with the rational arithmetic of `<ratio>`, probability distributions of `<random>`, bit operations of `<bit>` and constants of `<numbers>`. Like rational arithmetic, probability distributions, bit operations and constants, basic statistics fit into the existing `std` **namespace**.

5 Technical Specifications

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

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

```
#include <execution>

namespace std {

// data types

enum class stats_data_kind { population, sample };

enum class stats_skewness_kind { adjusted, unadjusted };

enum class stats_kurtosis_kind { excess, nonexcess };

// accumulator objects

template<class T>
class mean_accumulator;

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

template<class T>
class geometric_mean_accumulator;

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

template<class T>
class harmonic_mean_accumulator;

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

template<class T>
class skewness_accumulator;

template<class T, class W = T>
class weighted_skewness_accumulator;

template<class T>
class kurtosis_accumulator;

template<class T, class W = T>
```

```
class weighted_kurtosis_accumulator;

template<class T>
class variance_accumulator;

template<class T, class W = T>
class weighted_variance_accumulator;

template<class T>
class standard_deviation_accumulator;

template<class T, class W = T>
class weighted_standard_deviation_accumulator;

// accumulator objects accumulation functions

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

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

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

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

// freestanding functions

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

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

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

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

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

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

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

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

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

template<ranges::input_range R>
constexpr auto skewness(R&& r, stats_data_kind dkind, stats_skewness_kind skind) ->
  ranges::iterator_t<R>::value_type;

template<ranges::input_range R, ranges::input_range W>
constexpr auto skewness(R&& r, W&& w, stats_data_kind dkind, stats_skewness_kind skind) ->
  ranges::iterator_t<R>::value_type;

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

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

template<ranges::input_range R>
constexpr auto kurtosis(R&& r, stats_data_kind dkind, stats_kurtosis_kind kkind) ->
  ranges::iterator_t<R>::value_type;

template<ranges::input_range R, ranges::input_range W>
constexpr auto kurtosis(R&& r, W&& w, stats_data_kind dkind, stats_kurtosis_kind kkind) ->
  ranges::iterator_t<R>::value_type;

template<class ExecutionPolicy, ranges::input_range R>
requires std::is_execution_policy_v<std::remove_cvref_t<ExecutionPolicy>>
constexpr auto kurtosis(
  ExecutionPolicy&& policy,
  R&& r,
  stats_data_kind dkind, stats_kurtosis_kind kkind) ->
```

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

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

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

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

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

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

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

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

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

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

```
}
```

5.2 Accumulator Objects

The accumulator objects specified in this section are trivially copyable. If, first, either or both of the values of `x` 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 Mean Accumulator Class Templates

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

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

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

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

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

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 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) mean are performed.
3. *Returns*: The (weighted) mean 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.

5.2.2 Geometric Mean Accumulator Class Templates

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

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

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

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

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

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 1 value and the length of `r` is less than or equal to the length of `w`, and, if the product of the values of `r` is negative, then `ranges::distance(r)` is odd.
2. *Effects*: Any remaining computations relating to the (weighted) geometric mean are performed.
3. *Returns*: The (weighted) geometric mean 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 Harmonic Mean Accumulator Class Templates

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

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

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

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

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

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 `1` value and the length of `r` is less than or equal to the length of `w`, and all of the values of `r` are positive.
2. *Effects*: Any remaining computations relating to the (weighted) harmonic mean are performed.
3. *Returns*: The (weighted) harmonic mean 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 Skewness Accumulator Class Templates

```
template<class T>
class skewness_accumulator
{
public:
  explicit constexpr skewness_accumulator(
    std::stats_data_kind dkind, std::stats_skewness_kind skind) noexcept;
  constexpr void operator()(const T& x);
  constexpr auto value() -> T;
};

template<class T, class W = T>
class weighted_skewness_accumulator
{
public:
  explicit constexpr weighted_skewness_accumulator(
    std::stats_data_kind dkind, std::stats_skewness_kind skind) noexcept;
  constexpr void operator()(const T& x, const W& w);
  constexpr auto value() -> T;
};
```

```
explicit constexpr skewness_accumulator(
  std::stats_data_kind dkind, std::stats_skewness_kind skind) noexcept;
explicit constexpr weighted_skewness_accumulator(
  std::stats_data_kind dkind, std::stats_skewness_kind skind) noexcept;
```

1. *Effects*: A (weighted) skewness accumulator object is constructed.
2. *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() -> T;
```

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 3 values and the length of `r` is less than or equal to the length of `w`.
2. *Effects*: Any remaining computations relating to the (weighted) skewness are performed.
3. *Returns*: The (weighted) skewness 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.

5.2.5 Kurtosis Accumulator Class Templates

```
template<class T>
class kurtosis_accumulator
{
public:
  explicit constexpr kurtosis_accumulator(
    std::stats_data_kind dkind, std::stats_kurtosis_kind kkind) noexcept;
  constexpr void operator()(const T& x);
  constexpr auto value() -> T;
};

template<class T, class W = T>
class weighted_kurtosis_accumulator
{
public:
  explicit constexpr weighted_kurtosis_accumulator(
    std::stats_data_kind dkind, std::stats_kurtosis_kind kkind) noexcept;
  constexpr void operator()(const T& x, const W& w);
  constexpr auto value() -> T;
};
```

```
explicit constexpr kurtosis_accumulator(
  std::stats_data_kind dkind, std::stats_kurtosis_kind kkind) noexcept;
explicit constexpr weighted_kurtosis_accumulator(
  std::stats_data_kind dkind, std::stats_kurtosis_kind kkind) noexcept;
```

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

2. *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() -> T;
```

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 4 values and the length of `r` is less than or equal to the length of `w`.
2. *Effects*: Any remaining computations relating to the (weighted) kurtosis are performed.
3. *Returns*: The (weighted) kurtosis 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.

5.2.6 Variance Accumulator Class Templates

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

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

```
explicit constexpr variance_accumulator(T ddof) noexcept;
explicit constexpr weighted_variance_accumulator(std::stats_data_kind dkind) noexcept;
```

1. *Effects*: A (weighted) variance accumulator object is constructed.
2. *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() -> T;
```

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 1 value 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) variance are performed.
3. *Returns*: The (weighted) variance 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.

5.2.7 Standard Deviation Accumulator Class Templates

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

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

```
explicit constexpr standard_deviation_accumulator(T ddof) noexcept;
explicit constexpr weighted_standard_deviation_accumulator(
  std::stats_data_kind dkind) noexcept;
```

1. *Effects*: A (weighted) standard deviation accumulator object is constructed.
2. *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() -> T;
```

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 1 value 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) standard deviation are performed.
3. *Returns*: The (weighted) standard deviation 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.

5.2.8 Accumulator Objects Accumulation Functions

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

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

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

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

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

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

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

5.3 Freestanding Functions

If, first, either or both 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 Mean Functions

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

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

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

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

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

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

5.3.2 Freestanding Geometric Mean Functions

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

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

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

1. *Preconditions*: `r` and `w` are ranges of finite values, where `r` has at least 1 value and the length of `r` is less than or equal to the length of `w`, and, if the product of the values of `r` is negative, then `ranges::distance(r)` is odd.
2. *Returns*: The (weighted) geometric mean 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).`

5.3.3 Freestanding Harmonic Mean Functions

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

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

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

1. *Preconditions*: `r` and `w` are ranges of finite values, where `r` has at least 1 value and the length of `r` is less than or equal to the length of `w`, and all of the values of `r` are positive.
2. *Returns*: The (weighted) harmonic mean 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 Skewness Functions

```
template<ranges::input_range R>
constexpr auto skewness(R&& r, stats_data_kind dkind, stats_skewness_kind skind) ->
  ranges::iterator_t<R>::value_type;

template<ranges::input_range R, ranges::input_range W>
constexpr auto skewness(R&& r, W&& w, stats_data_kind dkind, stats_skewness_kind skind) ->
```

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

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

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

1. *Preconditions*: `r` and `w` are ranges of finite values, where `r` has at least 3 values and the length of `r` is less than or equal to the length of `w`.
2. *Returns*: The (weighted) skewness of the values of `r` (weighted by the corresponding values of the associated range `w`) if the preconditions have been met and an unspecified value otherwise.
3. *Complexity*: Linear in `ranges::distance(r)`.

5.3.5 Freestanding Kurtosis Functions

```
template<ranges::input_range R>
constexpr auto kurtosis(R&& r, stats_data_kind dkind, stats_kurtosis_kind kkind) ->
  ranges::iterator_t<R>::value_type;

template<ranges::input_range R, ranges::input_range W>
constexpr auto kurtosis(R&& r, W&& w, stats_data_kind dkind, stats_kurtosis_kind kkind) ->
  ranges::iterator_t<R>::value_type;

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

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

1. *Preconditions*: `r` and `w` are ranges of finite values, where `r` has at least 4 values and the length of `r` is less than or equal to the length of `w`.
2. *Returns*: The (weighted) kurtosis of the values of `r` (weighted by the corresponding values of the associated range `w`) if the preconditions have been met and an unspecified value otherwise.
3. *Complexity*: Linear in `ranges::distance(r)`.

5.3.6 Freestanding Variance Functions

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

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

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

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

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

5.3.7 Freestanding Standard Deviation Functions

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

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

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

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

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

2. *Returns*: The (weighted) standard deviation 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] statistics - mathematical statistics functions, python. Python, accessed 14 Apr. 2020. `https://docs.python.org/3/library/statistics.html`.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

[24] Philippe Pébay, Timothy B. Terriberry, Hemanth Kolla, and Janine Bennett. Numerically stable, scalable formulas for parallel and online computation of higher-order multivariate central moments with arbitrary weights. *Computational Statistics*, 31(4):1305–1325, 2016.

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

[26] Johan Hoffman. *Methods in Computational Science*. Society for Industrial and Applied Mathematics, 2021.

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

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

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

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

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

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

[33] Weighted arithmetic mean. Wikipedia, accessed 26 December 2022. https://en.wikipedia.org/wiki/Weighted_arithmetic_mean#Weighted_sample_variance.

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

[35] John Gurland and Ram C. Tripathi. A simple approximation for unbiased estimation of the standard deviation. *The American Statistician*, 25(4):30–32, Oct. 1971.

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

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

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

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

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

[41] Jolanta Opara. P2119R0 feedback on P1708: Simple statistical functions. JTC1/SC22/WG21, accessed 14 Apr. 2020. http://open-std.org/JTC1/SC22/WG21/docs/papers/2020/p2119r0.html.

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

[43] Central moment. Wikipedia, accessed 29 December 2022. https://en.wikipedia.org/wiki/Central_moment.

[44] Cumulant. Wikipedia, accessed 29 December 2022. https://en.wikipedia.org/wiki/Cumulant.

[45] Kurtosis. Wikipedia, accessed 29 December 2022. https://en.wikipedia.org/wiki/Kurtosis.

Appendix A Examples

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

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

std::array<PRODUCT,5> A = { {{5.2f, 1}, {1.7f, 2}, {9.2f, 5}, {4.4f, 7}, {1.7f, 3}} };
auto A_ = A | std::views::transform([](const auto& product) { return product.price; });
```

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

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

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

std::cout << "kurtosis = " << std::kurtosis(v, v_wgts,
  std::stats_data_kind::population, std::stats_kurtosis_kind::excess);
```

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

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

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

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

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

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

// ...

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

std::mean_accumulator<int> m;
std::skewness_accumulator<int> sk(
  std::stats_data_kind::population, std::stats_skewness_kind::unadjusted);
sum_squares_accumulator ssq;

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

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

Appendix B Derivations of Skewness and Kurtosis Formulas

Building on the results of [21], derivations of the (less familiar) skewness and kurtosis formulas of Sections 2.2 and 2.3, respectively, are presented in what follows. Let m_2, m_3 and m_4 be the *second*, *third* and *fourth* **population** *central moments* [21, 43], respectively, of the values, defined as

$$m_2 = \sigma^2 = \frac{1}{n}\sum_{i=1}^{n}\left(x_i - \mu\right)^2, \tag{24}$$

$$m_3 = \frac{1}{n}\sum_{i=1}^{n}\left(x_i - \mu\right)^3 \text{ and} \tag{25}$$

$$m_4 = \frac{1}{n}\sum_{i=1}^{n}\left(x_i - \mu\right)^4. \tag{26}$$

Let M_2, M_3 and M_4 be the second, third and fourth **sample** central moments [21], respectively, defined as

$$M_2 = \frac{n}{n-1}m_2, \tag{27}$$

$$M_3 = \frac{n^2}{(n-1)(n-2)}m_3 \text{ and} \tag{28}$$

$$M_4 = \frac{n\left(n^2-2n+3\right)}{(n-1)(n-2)(n-3)}m_4 - \frac{3n\left(2n-3\right)}{(n-1)(n-2)(n-3)}m_2^2. \tag{29}$$

Let t_1, t_2 and t_3 be the **weighted** second, third and fourth **population** central moments [21], respectively, defined as

$$t_2 = \hat{\sigma}^2 = \frac{1}{V_1}\sum_{i=1}^{n} w_i\left(x_i - \mu^*\right)^2, \tag{30}$$

$$t_3 = \frac{1}{V_1}\sum_{i=1}^{n} w_i\left(x_i - \mu^*\right)^3 \text{ and} \tag{31}$$

$$t_4 = \frac{1}{V_1}\sum_{i=1}^{n} w_i\left(x_i - \mu^*\right)^4. \tag{32}$$

Let T_2, T_3 and T_4 be the **weighted** second, third and fourth **sample** central moments [21], respectively, defined as

$$T_2 = \frac{V_1^2}{V_1^2 - V_2}t_2, \tag{33}$$

$$T_3 = \frac{V_1^3}{V_1^3 - 3V_1V_2 + 2V_3}t_3 \text{ and} \tag{34}$$

$$T_4 = \frac{V_1^2\left(V_1^4 - 3V_1^2V_2 + 2V_1V_3 + 3V_2^2 - 3V_4\right)}{\left(V_1^2 - V_2\right)\left(V_1^4 - 6V_1^2V_2 + 8V_1V_3 + 3V_2^2 - 6V_4\right)}t_4 - \frac{3V_1^2\left(2V_1^2V_2 - 2V_1V_3 - 3V_2^2 + 3V_4\right)}{\left(V_1^2 - V_2\right)\left(V_1^4 - 6V_1^2V_2 + 8V_1V_3 + 3V_2^2 - 6V_4\right)}t_2^2. \tag{35}$$

Let c_4 be the *fourth* **population** *cumulant* [21, 44], defined as

$$c_4 = m_4 - 3m_2^2. \tag{36}$$

Let C_4 be the *fourth* **sample** *cumulant* [21], defined as

$$C_4 = \frac{n^2\left(n+1\right)}{(n-1)(n-2)(n-3)}m_4 - \frac{3n^2}{(n-2)(n-3)}m_2^2. \tag{37}$$

Let k_4 be the **weighted** *fourth* **population** *cumulant* [21], defined as

$$k_4 = t_4 - 3t_2^2. \tag{38}$$

Let K_4 be the **weighted** *fourth* **sample** *cumulant* [21], defined as

$$K_4 = \frac{V_1^2\left(V_1^4 - 4V_1V_3 + 3V_2^2\right)}{\left(V_1^2 - V_2\right)\left(V_1^4 - 6V_1^2V_2 + 8V_1V_3 + 3V_2^2 - 6V_4\right)}t_4 - \frac{3V_1^2\left(V_1^4 - 2V_1^2V_2 + 4V_1V_3 - 3V_2^2\right)}{\left(V_1^2 - V_2\right)\left(V_1^4 - 6V_1^2V_2 + 8V_1V_3 + 3V_2^2 - 6V_4\right)}t_2^2. \tag{39}$$

B.1 Skewness

The **population** skewness [21] is

$$\frac{m_3}{m_2^{3/2}} = \frac{m_3}{\sigma^3} \tag{40}$$

$$= \frac{1}{n\sigma^3} \sum_{i=1}^{n} (x - \mu)^3 \, . \tag{41}$$

The **sample** skewness [21] is

$$\frac{M_3}{M_2^{3/2}} = \frac{\dfrac{n^2}{(n-1)(n-2)} m_3}{\left(\dfrac{n}{n-1} m_2\right)^{3/2}} \tag{42}$$

$$= \frac{\dfrac{n^2}{(n-1)(n-2)} m_3}{(s^2)^{3/2}} \tag{43}$$

$$= \frac{n}{(n-1)(n-2)\, s^3} \sum_{i=1}^{n} (x - \bar{x})^3 \, . \tag{44}$$

The **weighted population** skewness [21] is

$$\frac{t_3}{t_2^{3/2}} = \frac{t_3}{\hat{\sigma}^3} \tag{45}$$

$$= \frac{1}{V_1 \sigma^3} \sum_{i=1}^{n} w_i (x - \mu^*)^3 \, . \tag{46}$$

The **weighted sample** skewness [21] is

$$\frac{T_3}{T_2^{3/2}} = \frac{\dfrac{V_1^3}{V_1^3 - 3V_1V_2 + 2V_3} t_3}{\left(\dfrac{V_1^2}{V_1^2 - V_2} t_2\right)^{3/2}} \tag{47}$$

$$= \frac{V_1^3}{V_1^3 - 3V_1V_2 + 2V_3} t_3 \cdot \frac{\left(V_1^2 - V_2\right)^{3/2}}{V_1^3 (t_2)^{3/2}} \tag{48}$$

$$= \frac{\left(V_1^2 - V_2\right)^{3/2}}{V_1 \left(V_1^3 - 3V_1V_2 + 2V_3\right) \hat{\sigma}^3} \sum_{i=1}^{n} w_i (x - \bar{x}^*)^3 \, . \tag{49}$$

B.2 Kurtosis

The Pearson **population** kurtosis [21, 45] is

$$\frac{m_4}{m_2^2} = \frac{m_4}{\sigma^4} \tag{50}$$

$$= \frac{1}{n\sigma^4} \sum_{i=1}^{n} (x - \mu)^4 \, . \tag{51}$$

The Pearson **sample** kurtosis [21] is

$$\frac{M_4}{M_2^2} = \frac{\frac{n\left(n^2-2n+3\right)}{(n-1)(n-2)(n-3)}m_4 - \frac{3n(2n-3)}{(n-1)(n-2)(n-3)}m_2^2}{\left(\frac{n}{n-1}m_2\right)^2} \tag{52}$$

$$= \frac{\frac{n\left(n^2-2n+3\right)}{(n-1)(n-2)(n-3)}m_4 - \frac{3n(2n-3)}{(n-1)(n-2)(n-3)}\sigma^4}{\left(s^2\right)^2} \tag{53}$$

$$= \frac{n^2-2n+3}{(n-1)(n-2)(n-3)s^4}\sum_{i=1}^{n}(x_i-\bar{x})^4 - \frac{3n(2n-3)\sigma^4}{(n-1)(n-2)(n-3)s^4}. \tag{54}$$

The **weighted** Pearson **population** kurtosis [21] is

$$\frac{t_4}{t_2^2} = \frac{t_4}{\hat{\sigma}^4} \tag{55}$$

$$= \frac{1}{n\hat{\sigma}^4}\sum_{i=1}^{n}\left(x-\mu^*\right)^4. \tag{56}$$

The **weighted** Pearson **sample** kurtosis [21] is

$$\frac{T_4}{T_2^2} = \frac{\frac{V_1^2\left(V_1^4-3V_1^2V_2+2V_1V_3+3V_2^2-3V_4\right)}{\left(V_1^2-V_2\right)\left(V_1^4-6V_1^2V_2+8V_1V_3+3V_2^2-6V_4\right)}t_4 - \frac{3V_1^2\left(2V_1^2V_2-2V_1V_3-3V_2^2+3V_4\right)}{\left(V_1^2-V_2\right)\left(V_1^4-6V_1^2V_2+8V_1V_3+3V_2^2-6V_4\right)}t_2^2}{\left(\frac{V_1^2}{V_1^2-V_2}t_2\right)^2} \tag{57}$$

$$= \frac{V_1^2\left(V_1^4-3V_1^2V_2+2V_1V_3+3V_2^2-3V_4\right)}{\left(V_1^2-V_2\right)\left(V_1^4-6V_1^2V_2+8V_1V_3+3V_2^2-6V_4\right)}t_4 - \tag{58}$$

$$\frac{3V_1^2\left(2V_1^2V_2-2V_1V_3-3V_2^2+3V_4\right)}{\left(V_1^2-V_2\right)\left(V_1^4-6V_1^2V_2+8V_1V_3+3V_2^2-6V_4\right)}t_2^2 \cdot \frac{\left(V_1^2-V_2\right)^2}{V_1^4\left(t_2\right)^2} \tag{59}$$

$$= \frac{\left(V_1^2-V_2\right)\left(V_1^4-3V_1^2V_2+2V_1V_3+3V_2^2-3V_4\right)}{V_1^3\left(V_1^4-6V_1^2V_2+8V_1V_3+3V_2^2-6V_4\right)\hat{\sigma}^4}\sum_{i=1}^{n}w_i(x_i-\bar{x}^*)^4 - \frac{3\left(V_1^2-V_2\right)\left(2V_1^2V_2-2V_1V_3-3V_2^2+3V_4\right)}{V_1^2\left(V_1^4-6V_1^2V_2+8V_1V_3+3V_2^2-6V_4\right)}. \tag{60}$$

The excess **population** kurtosis [21] is

$$\frac{c_4}{m_2^2} = \frac{m_4-3m_2^2}{m_2^2} \tag{61}$$

$$= \frac{m_4}{\sigma^4} - 3 \tag{62}$$

$$= \frac{1}{n\sigma^4}\sum_{i=1}^{n}\left(x-\mu\right)^4 - 3. \tag{63}$$

The excess **sample** kurtosis [21] is

$$\frac{C_4}{M_2^2} = \frac{\frac{n^2(n+1)}{(n-1)(n-2)(n-3)}m_4 - \frac{3n^2}{(n-2)(n-3)}m_2^2}{\left(\frac{n}{n-1}m_2\right)^2} \tag{64}$$

$$= \frac{\frac{n^2(n+1)}{(n-1)(n-2)(n-3)}m_4 - \frac{3n^2}{(n-2)(n-3)}m_2^2}{\left(s^2\right)^2} \tag{65}$$

$$= \frac{n(n+1)}{(n-1)(n-2)(n-3)s^4}\sum_{i=1}^{n}\left(x_i-\bar{x}\right)^4 - \frac{3n^2\sigma^4}{(n-2)(n-3)s^4}. \tag{66}$$

The **weighted** excess **population** kurtosis [21] is

$$\frac{k_4}{t_2^2} = \frac{t_4 - 3t_2^2}{t_2^2} \tag{67}$$

$$= \frac{t_4}{\hat{\sigma}^4} - 3 \tag{68}$$

$$= \frac{1}{n\hat{\sigma}^4} \sum_{i=1}^{n} (x - \mu^*)^4 - 3. \tag{69}$$

The **weighted** excess **sample** kurtosis [21] is

$$\frac{K_4}{T_2^2} = \frac{\frac{V_1^2\left(V_1^4 - 4V_1V_3 + 3V_2^2\right)}{\left(V_1^2 - V_2\right)\left(V_1^4 - 6V_1^2V_2 + 8V_1V_3 + 3V_2^2 - 6V_4\right)}t_4 - \frac{3V_1^2\left(V_1^4 - 2V_1^2V_2 + 4V_1V_3 - 3V_2^2\right)}{\left(V_1^2 - V_2\right)\left(V_1^4 - 6V_1^2V_2 + 8V_1V_3 + 3V_2^2 - 6V_4\right)}t_2^2}{\left(\frac{V_1^2}{V_1^2 - V_2}t_2\right)^2} \tag{70}$$

$$= \frac{V_1^2\left(V_1^4 - 4V_1V_3 + 3V_2^2\right)}{\left(V_1^2 - V_2\right)\left(V_1^4 - 6V_1^2V_2 + 8V_1V_3 + 3V_2^2 - 6V_4\right)}t_4 - \tag{71}$$

$$\frac{3V_1^2\left(V_1^4 - 2V_1^2V_2 + 4V_1V_3 - 3V_2^2\right)}{\left(V_1^2 - V_2\right)\left(V_1^4 - 6V_1^2V_2 + 8V_1V_3 + 3V_2^2 - 6V_4\right)}t_2^2 \cdot \frac{\left(V_1^2 - V_2\right)^2}{V_1^4\left(t_2\right)^2} \tag{72}$$

$$= \frac{\left(V_1^2 - V_2\right)\left(V_1^4 - 4V_1V_3 + 3V_2^2\right)}{V_1^3\left(V_1^4 - 6V_1^2V_2 + 8V_1V_3 + 3V_2^2 - 6V_4\right)\hat{\sigma}^4} \sum_{i=1}^{n} w_i(x_i - \bar{x}^*)^4 - \frac{3\left(V_1^2 - V_2\right)\left(V_1^4 - 2V_1^2V_2 + 4V_1V_3 - 3V_2^2\right)}{V_1^2\left(V_1^4 - 6V_1^2V_2 + 8V_1V_3 + 3V_2^2 - 6V_4\right)}. \tag{73}$$