Simple Statistical Functions

Richard Dosselmann, Eric Niebler, Phillip Ratzloff, Vincent Reverdy, Michael Wong

Document Number: P1708R1
Date: October 10, 2019 (Pre-Belfast mailing): 10 AM ET
Project: ISO JTC1/SC22/WG21: Programming Language C++
Audience: SG19, WG21, LEWG
Emails: dosselmr@cs.uregina.ca (corresponding author),
eniebler@fb.com,
phil.ratzloff@sas.com,
vreverdy@illinois.edu,
michael@codeplay.com

Contents

1 Introduction

This document proposes an extension to the C++ **numerics** library to support simple statistical functions. Such functions, **not** presently found in the standard (including the special math library), frequently arise in **scientific** and **industrial**, as well as **general**, applications. These functions do exist in Python, the foremost competitor to C++ in the area of **machine learning** [1].

1.1 Revision History

P1708R1

- Reformatted using LaTeX
- Introduction of accumulator set

2 Impact on the Standard

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

3 Proposal

This document proposes the addition of the simple statistical functions `mean`, `median`, `mode`, `stddev` and `var` to compute the **mean**, **median**, **mode**, **standard deviation** and **variance**, respectively, of the values $x_1, x_2, ..., x_n$. This (revised) proposal follows the model of the Boost Accumulators library, in which an **accumulator set** (of simple statistical functions) makes a **single** pass over the given values [2].

3.1 Accumulator Set

Inspired by the Boost Accumulators library, an *accumulator set* is an object that contains **one or more** simple statistical functions to be evaluated over the given values. The proposed form of the accumulator set is

```
template<typename T, auto& ... /* simple statistical functions */>
requires ...
struct accumulator_set
{
   // ... push range r ...
   // ... push range given by first and last ...
};
```

Parameters

- `r` - the **range** of elements to modify
- `first, last` - the (**start** and **end** of the) range of elements to modify

Exception

If the range is **empty**, `stats_error` is thrown.

3.2 Mean

The (arithmetic) *mean* [3], denoted μ or $\bar{x}$ in the case of a **population** [3] or **sample** [3], respectively, is defined as

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

Equation (1) has a **linear** run-time. The proposed forms of the mean function are

```
constexpr std::accumulator_set::value_type mean(std::accumulator_set& acc); // (1)

constexpr std::accumulator_set::value_type
   mean(std::accumulator_set& acc, BinaryOperation op); // (2)
```

Parameters

- `acc` - the accumulator set of which to compute the mean
- `op` - binary operation function object that will be applied. The binary operation takes the **current accumulation** value `a` and the value `b` of the **current element**. The signature of the function should be equivalent to the following:

  ```
  Ret fun(const Type1 &a, const Type2 &b);
  ```

Return Value

The mean of the values.

Examples

```
// example 1
std::vector<int> v{1, 2, 3, 4, 5, 6};
std::accumulator_set<double>(std::mean) acc1;
acc1(v);
std::cout << "mean 1: " << std::mean(acc1) << '\n'; // mean: 3.5

// example 2
struct POINT { int x, y; };
POINT A[] = {{2,5}, {6,2}, {9,4}, {6,13}};
std::accumulator_set<POINT>(std::mean) acc2;
acc2(A.begin()+1, A.end());
std::cout << "mean 2: "
   << std::mean(acc2, [](const Type1& a, const Type2& b) { return a + b.x; })
   << '\n'; // mean: 7
```

3.3 Median

The *median* (of the **sorted** values) is defined as the **middle** value if n is **odd** and the mean of the two middle values if n is **even** [3]. This procedure can be performed (without sorting) in **linear** time using the **quickselect** algorithm [4]. The proposed forms of the median function are

```
constexpr std::tuple<bool unique,
   std::accumulator_set::value_type& median1,
   std::accumulator_set::value_type& median2>
      median(std::accumulator_set& acc); // (1)

constexpr std::tuple<bool unique,
   std::accumulator_set::value_type& median1,
   std::accumulator_set::value_type& median2>
      median(std::accumulator_set& acc, Compare comp); // (2)
```

Parameters

- `acc` - the accumulator set of which to compute the median
- `comp` - comparison function object which returns `true` if the **first argument** is less than (i.e. is ordered before) the **second**. The signature of the function should be equivalent to the following:

  ```
  bool comp(const Type1 &a, const Type2 &b);
  ```

Return Values

- `unique` - `true` if there is **one** median and `false` otherwise
- `median1` - the **first** (and perhaps only) median of the values
- `median2` - the **second** median of the values (if it exists)

Examples

```
// example 1
std::vector<int> v1{9, 3, 12, -1, 4, 7, 27};
std::accumulator_set<int>(std::median) acc1;
acc1(v1);
std::cout << "median 1: " << get<1>(std::median(acc1)) << '\n'; // median: 7

// example 2
std::vector<int> v2{9, 3, 12, -1, 4, 7};
std::accumulator_set<double>(std::mean, std::median) acc2;
acc2(v2);
std::cout << "mean: " << std::mean(acc2) << '\n'; // mean: 5.666...
std::cout << "median 2: " << get<1>(std::median(acc2)) << '\n'; // median: 5.5

// example 3
std::vector<std::string> v3{"cyan", "yellow", "magenta", "black"};
std::accumulator_set<std::string>(std::median) acc3;
acc3(v3);

if(auto& [val, median1, median2] = std::median(acc3); val)
   std::cout << "median 3: " << get<1>(std::median(acc3)) << '\n';
```

```
else
{
   "(first) median 3: " << get<1>(std::median(acc3)) << '\n'; // median: "cyan"
   "(second) median 3: " << get<2>(std::median(acc3)) << '\n'; // median: "magenta"
}
```

3.4 Mode

The *mode* is defined as the (perhaps not unique) value having the **highest frequency** [3]. This procedure can be performed in **linear** time using a **hash table** of the given values. The proposed forms of the mode function are

```
constexpr std::optional<std::accumulator_set::value_type>
   mode(std::accumulator_set& acc); // (1)

constexpr std::optional<std::accumulator_set::value_type>
   mode(std::accumulator_set& acc, BinaryPredicate p); // (2)
```

Parameters

- `acc` - the accumulator set of which to compute the mode
- `p` - binary predicate which returns `true` if the elements should be treated as **equal**. The signature of the predicate function should be equivalent to the following:

  ```
  bool pred(const Type1 &a, const Type2 &b);
  ```

Return Value

A **vector** of the mode(s) of the values.

Examples

```
// example 1
std::vector<int> v{19, 2, 8, 3, 2};
std::accumulator_set<int>(std::mode) acc1;
acc1(v.begin()+1, v.end());

if(std::mode(acc1).front().has_value())
   std::cout << "mode: " << std::mode(acc1).front().value() << '\n'; // mode: 2

// example 2
struct product { double price; int quantity; };
product data[] = {{3.29, 11}, {1.59, 6}, {11.99, 6}, {5.99, 11}, {6.39, 5}};
std::accumulator_set<product>(std::median, std::mode) acc2;
acc2(data);
auto p = [](const Type1 &a, const Type &b) { return a.quantity == b.quantity; };
std::cout << "median: " << std::median(acc2).quantity << '\n'; // median: 5
auto modes = std::mode(acc2, p);
```

```
for(const auto &m : modes)
   std::cout << "mode: " << m.value().quantity << '\n'; // mode: 6, 11
```

3.5 Standard Deviation

The **population** *standard deviation* [3], denoted σ, is defined as

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

The proposed forms of the population standard deviation functions are

```
constexpr std::accumulator_set::value_type
   population_stddev(std::accumulator_set& acc); // (1)

constexpr std::accumulator_set::value_type
   population_stddev(std::accumulator_set& acc, BinaryOperation op); // (2)
```

The **sample** *standard deviation* [3], denoted σ, is defined as

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

The proposed forms of the sample standard deviation functions are

```
constexpr std::accumulator_set::value_type
   sample_stddev(std::accumulator_set& acc); // (1)

constexpr std::accumulator_set::value_type
   sample_stddev(std::accumulator_set& acc, BinaryOperation op); // (2)
```

Parameters

- `acc` - the accumulator set of which to compute the standard deviation
- `op` - binary operation function object that will be applied. The binary operation takes the **current accumulation** value `a` and the value `b` of the **current element**. The signature of the function should be equivalent to the following:

  ```
  Ret fun(const Type1 &a, const Type2 &b);
  ```

Return Value

The standard deviation of the values.

Exception

If the range (of the accumulator set) is a **single** value, `stats_error` is thrown.

Examples

```
// example 1
std::vector<int> v{1, 2, 3, 4, 5};
std::accumulator_set<double>(std::mean, std::population_stddev) acc1;
acc1(v);
std::cout << "mean: " << std::mean(acc1) << '\n'; // mean: 5
std::cout << "population stddev: " << std::population_stddev(acc1) << '\n';
// population stddev: 1.414...

// example 2
std::list<std::tuple<int,int,int>> L{{1,2,3}, {4,3,2}, {5,5,5}};
std::accumulator_set<std::tuple<int,int,int>>(std::sample_stddev) acc2;
acc2(L);
std::cout << "sample stddev: " << std::sample_stddev(acc2) << '\n';
// sample stddev: 1.581...
```

3.6 Variance

The **population** *variance*, denoted σ^2, is defined as the **square** of the population standard deviation [3]. The proposed forms of the population variance functions are

```
constexpr std::accumulator_set::value_type
   population_var(std::accumulator_set& acc); // (1)

constexpr std::accumulator_set::value_type
   population_var(std::accumulator_set& acc, BinaryOperation op); // (2)
```

The **sample** *variance*, denoted s^2, is defined as the **square** of the sample standard deviation [3]. The proposed forms of the sample variance functions are

```
constexpr std::accumulator_set::value_type
   sample_var(std::accumulator_set& acc); // (1)

constexpr std::accumulator_set::value_type
   sample_var(std::accumulator_set& acc, BinaryOperation op); // (2)
```

Parameters

- `acc` - the accumulator set of which to compute the variance
- `op` - binary operation function object that will be applied. The binary operation takes the **current accumulation** value `a` and the value `b` of the **current element**. The signature of the function should be equivalent to the following:

  ```
  Ret fun(const Type1 &a, const Type2 &b);
  ```

Return Value

The variance of the values.

Exceptions

If the range (of the accumulator set) is a **single** value, `stats_error` is thrown.

Example

```
std::vector<int> v{8, 6, 5, -3, 0};
std::accumulator_set<float>(std::population_var, std::sample_var) acc;
acc(v);
std::cout << "population var: " << std::population_var(acc) << '\n';
// population var: 16.56
std::cout << "sample var: " << std::sample_var(acc) << '\n';
// sample var: 20.7
```

4 Future Proposals

Additional statistical functions, such as those found in the Boost Accumulators library, might be considered for future standardization. Such functions, **not** found in Python, include covariance, kurtosis and skewness.

5 Acknowledgements

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

References

[1] "statistics - Mathematical statistics functions" *Python*. Web. 17 Aug. 2019 (`https://docs.python.org/3/library/statistics.html`).

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

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

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