P1708R0: Simple Statistical Functions

Date:	2019-06-17 (Pre-Cologne mailing): 10 AM ET
Project:	ISO JTC1/SC22/WG21: Programming Language C++
Audience:	SG19, WG21, LEWG

Authors:	Richard Dosselmann (U of Regina) Michael Wong (Codeplay)
Contributors:	N/A
Emails:	dosselmr@cs.uregina.ca michael@codeplay.com
Reply to:	dosselmr@cs.uregina.ca

Introduction

This document proposes **an** extension to the C++ **numerics** library to support **simple statistics**.

Revision History

N/A

Motivation

There are important **statistical** functions that are used in **scientific**, **industrial** and **general** programming domains that do **not** presently exist in the C++ standard, including the special math library.

Impact on the Standard

This proposal is pure **library** extension.

Proposals

We propose the addition of the basic **statistical functions** `mean`, `median`, `mode`, `population_stddev`, `sample_stddev`, `population_var` and `sample_var` to

`<numeric>` to compute the arithmetic **mean**, **median**, **mode** and **population** and **sample standard deviation** and **variance**, respectively, of the elements in the range `[first, last)`. These statistics are used in virtually **all research**, **scientific** and **industrial** domains, as well as **general** programming. They are found in the *Boost Accumulators* package [1]. Moreover, these functions exist in *Python* [2], the foremost competitor to C++ in the area of machine learning. The proposed forms of these functions are given below.

Mean

The arithmetic *mean* of the given **range** is the **sum** of the elements in the range **divided** by the **number** of elements in the range. The proposed form of this function is:

```
template<class T = double, class InputIt>
constexpr T mean(InputIt first, InputIt last);
```

Parameters

`first`, `last` **-** the **range** of elements of which to compute the mean

Return Value

The **mean** of the elements in the given **range**.

Exceptions

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

Example

```
std::vector<int> v{1, 2, 3, 4, 5, 6};

double m1 = std::mean(v.begin(), v.end());
std::cout << "mean: " << m1 << '\n'; // mean: 3.5

float m2 = std::mean<float>(v.begin(), v.end())
std::cout << "mean: " << m2 << '\n'; // mean: 3.5
```

Median

The *median* of the given **range** is the **middle** element of the range if the range is of **odd** length and the **two middle** elements otherwise. The proposed form of this function is:

```
template<class InputIt>
constexpr std::pair<InputIt,InputIt>
      median(InputIt first, InputIt last);
```

Parameters

`first`, `last` - the **sorted range** of elements of which to compute the median

Return Value

A **pair** consisting of an **iterator** to the **first** element of the median and an iterator to the **last** element of the median. Returns `std::make_pair(first, first)` if the range is **empty**.

Example

```
std::vector<int> v1{9, 3, 12, -1, 4, 7};
std::sort(v1.begin(), v1.end());
auto p1 = std::median(v1.begin(), v1.end());
std::cout << "median 1: " << (double)(*p1.first + *p1.second) / 2.0
      << '\n'; // median 1: 5.5

std::vector<std::string> v2{"cyan", "yellow", "magenta", "black"};
std::sort(v2.begin(), v2.end());
auto p2 = std::median(v2.begin(), v2.end());
std::cout << "median 2: " << (*p2.first).c_str() << " or "
      << (*p2.second).c_str() << '\n'; // median 2: cyan or magenta
```

Mode

The *mode* of the given **range** is the element of the range with the **highest frequency**. The proposed forms of this function are:

```
template<class InputIt>
constexpr InputIt mode(InputIt first, InputIt last); // (1)

template<class InputIt, class BinaryPredicate>
constexpr InputIt
      mode(InputIt first, InputIt last, BinaryPredicate p); // (2)
```

Parameters

`first`, `last` - the **sorted range** of elements 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

An iterator to the first element equal to the **mode**. Returns `last` if the range is **empty**.

Exceptions

If the mode is **not unique**, `stats_error` is **thrown** (just as **Python** throws an exception).

Example

```
std::vector<int> v{19, 2, 8, 3, 2};
std::sort(v.begin(), v.end());
std::vector<int>::iterator i1 = std::mode(v.begin(), v.end()); // (1)
std::cout << "mode: " << *i1 << " at position "
      << std::distance(v.begin(), i1) << '\n'; // mode: 2 at position
0

struct POINT { int x, y; };
POINT A[] = {{2,5}, {6,2}, {9,4}, {6,13}};
std::sort(A, A + 4,
      [](const POINT& p1, const POINT& p2)
      { return (p1.x < p2.x) || (p1.x == p2.x && p1.y < p2.y); });

auto i2 = std::mode(A, A + 4,
      [](const POINT& p1, const POINT& p2) { return p1.x == p2.x; });
      // (2)
std::cout << "mode: " << (*i2).x << "," << (*i2).y << '\n';
// mode: 6,2 at position 1
```

Standard Deviation

The population *standard deviation* [3] of the given **range** is

$$\sqrt{\frac{1}{N}\sum_{i=1}^{N}(x_i - X)^2}$$

where x_i is an element of the range, X is the **mean** of the range and N is the **number** of elements in the range. The sample standard deviation is the same as the sample standard deviation with the exception that it is **scaled** by 1/(N-1) rather than 1/N. The proposed forms of these functions are:

```
template<class T = double, class InputIt>
```

```
constexpr T population_stddev(InputIt first, InputIt last);

template<class T = double, class InputIt>
constexpr T sample_stddev(InputIt first, InputIt last);
```

Parameters

`first`, `last` - the **range** of elements of which to compute the standard deviation

Return Value

The **standard deviation** of the elements in the given **range**.

Exceptions

If the **range** is a **single** value, `stats_error` is **thrown** (just as **Python** throws an exception).

Example

```
std::vector<int> v{1, 2, 3, 4, 5};

double s1 = std::population_stddev(v.begin(), v.end());
std::cout << "stddev: " << s1 << '\n'; // stddev: 1.4142135 ...

float s2 = std::sample_stddev<float>(v.begin(), v.end());
std::cout << "stddev: " << s2 << '\n'; // stddev: 1.5811388 ...
```

Variance

The population *variance* [4] of the given **range** is the **square** of the population **standard deviation** and the sample variance is the **square** of the sample **standard deviation**. The proposed forms of these functions are:

```
template<class T = double, class InputIt>
constexpr T population_var(InputIt first, InputIt last);

template<class T = double, class InputIt>
constexpr T sample_var(InputIt first, InputIt last);
```

Parameters

`first`, `last` - the **range** of elements of which to compute the variance

Return Value

The **variance** of the elements in the given **range**.

Exceptions

If the **range** is a **single** value, `stats_error` is **thrown**.

Example

```
std::vector<int> v{8, 6, 5, -3, 0};

float s1 = std::population_var<float>(v.begin(), v.end());
std::cout << "var: " << s1 << '\n'; // var: 16.56

double s2 = std::sample_var(v.begin(), v.end());
std::cout << "var: " << s2 << '\n'; // var: 20.7
```

Future Proposals

Additional statistical functions, such as those found in Boost accumulators [1], might be considered for future standardization. Such functions, **not** found in **Python**, include covariance, kurtosis and skewness.

Acknowledgements

This paper is thanks to discussion in SG19 Machine Learning.

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

Appendix

The `stats_error` class is defined as (following the model of **isocpp.org** [5]):

```
class stats_error : public std::runtime_error {
public:
   stats_error() : std::runtime_error("stats_error") { }
};
```

References

1. Boost Accumulators
2. statistics - Mathematical statistics functions
3. Standard deviation
4. Variance
5. Exceptions and Error Handling