## Contents

1 Introduction ............................................ 2  
   1.1 Revision History ....................................... 2  

2 Impact on the Standard ................................. 2  

3 Statistics .............................................. 2  
   3.1 Mean .................................................. 2  
      3.1.1 Median ............................................ 2  
      3.1.2 Mode .............................................. 2  
      3.1.3 Standard Deviation ................................ 3  
      3.1.4 Variance ........................................... 3  

4 Proposal .................................................. 3  
   4.1 stats Class ........................................... 3  
      4.1.1 Function Synopses .................................. 4  
      4.1.2 Examples .......................................... 7  

5 Future Proposals ......................................... 9  

6 Acknowledgements ....................................... 9
1 Introduction

This document proposes an extension to the C++ numerics library (where functions such as iota, accumulate and inner_product are found) 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
- TBA

P1708R2
- TBA

2 Impact on the Standard

This proposal is a pure library extension.

3 Statistics

Five statistics are defined in this proposal.

3.1 Mean

The (arithmetic) mean [2], denoted μ or $\bar{x}$ in the case of a population [2] or sample [2], respectively, is defined as

$$\frac{1}{n} \sum_{i=1}^{n} x_i. \quad (1)$$

Equation (1) has a linear run-time.

3.1.1 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 [2]. This procedure can be performed (without sorting) in linear time using the quickselect algorithm [3].

3.1.2 Mode

The mode is defined as the (perhaps not unique) value having the highest frequency [2]. This procedure can be performed in linear time by counting consecutive (repeated) values.
3.1.3 Standard Deviation

The population standard deviation [2], denoted \( \sigma \), is defined as
\[
\sqrt{\frac{1}{n} \sum_{i=1}^{n} (x_i - \mu)^2}.
\]
(2)

The sample standard deviation [2], denoted \( s \), is defined as
\[
\sqrt{\frac{1}{n-1} \sum_{i=1}^{n} (x_i - \bar{x})^2}.
\]
(3)

The standard deviation (and variance) may be computed in a single pass over the values [4].

3.1.4 Variance

The population variance, denoted \( \sigma^2 \), is defined as the square of the population standard deviation [2]. The sample variance, denoted \( s^2 \), is defined as the square of the sample standard deviation [2].

4 Proposal

This document proposes the addition of the simple statistical functions, as part of a statistics (stats) class, to compute the mean, median, mode, standard deviation and variance, each defined in the following subsections, of the (sorted) values \( x_1, x_2, \ldots, x_n \).

4.1 stats Class

All statistical functions are aggregated into a class, namely stats, so that they may be computed in a single pass over the (sorted) values, like the Boost Accumulators [5]. The proposed form of the stats class is

```cpp
template<typename T = double, typename Allocator = allocator<T>>
// ... requires ...
class stats {
public:

/* construction/destruction */

constexpr stats() noexcept;
constexpr stats(int m);
constexpr stats(const stats& other);
constexpr stats(stats&& other);
stats& operator=(const stats& other);
stats& operator=(stats&& other);
~stats() = default;

/* calculation */

template<
typename ForwardIt>
// ... requires ...
```


void calc(ForwardIt first, ForwardIt last);

template<
typename ForwardIt, typename UnaryPredicate>
// ... requires ...
void calc(ForwardIt first, ForwardIt last, UnaryPredicate p);

void calc(range r);
void calc(range r, UnaryPredicate p);

/* metrics */

static const int metric_mean = 0b0000001;
static const int metric_median = 0b0000010;
static const int metric_mode = 0b0000100;
static const int metric_population_stddev = 0b0001000;
static const int metric_sample_stddev = 0b0010000;
static const int metric_population_var = 0b0100000;
static const int metric_sample_var = 0b1000000;
static const int metric_all = 0b1111111;

void metrics(int m);
constexpr int metrics() const noexcept;

constexpr T mean() const noexcept;
tuple<bool,T,T> median() const noexcept;
constexpr std::list<T> mode() const noexcept;
constexpr T population_stddev() const noexcept;
constexpr T sample_stddev() const noexcept;
constexpr T population_var() const noexcept;
constexpr T sample_var() const noexcept;
};

4.1.1 Function Synopses

constexpr stats() noexcept;
constexpr stats(int m);
constexpr stats(const stats& other);
constexpr stats(stats&& other);
stats& operator=(const stats& other);
stats& operator=(stats&& other);
~stats() = default;

Purpose

Construction (and destruction) of a stats object.

Parameters

- m - the statistic(s), or metric(s), to compute
• other - another stats (object) to be used as source to initialize the elements of the stats (object) with

Return Value
A stats object.

Exceptions
TBA

```cpp
template<typename ForwardIt>
// ... requires ...
void calc(ForwardIt first, ForwardIt last);

template<typename ForwardIt, typename UnaryPredicate>
// ... requires ...
void calc(ForwardIt first, ForwardIt last, UnaryPredicate p);

void calc(range r);
void calc(range r, UnaryPredicate p);
```

Purpose
Compute, or calculate, the specified statistic(s) over the specified range.

Parameters
• first, last - the range of elements over which to compute the statistic(s)
• r - a range of elements over which to compute the statistic(s)
• p - unary predicate. The expression p(v) must be convertible to T for every argument v of type (possibly const) VT, where VT is the value type of ForwardIt, regardless of value category, and must not modify v. Thus, a parameter type of VT& is not allowed, nor is VT unless for VT a move is equivalent to a copy.

Exceptions
If the range is empty, stats_error is thrown. If the range is a single element, and a variance or standard deviation is specified, stats_error is thrown.

Complexity
At most last - first applications of the predicate (linear).

```cpp
void metrics(int m);
constexpr int metrics() const noexcept;
```

Purpose
Specify and obtain the statistic(s) to compute.
Return Value
The metric(s) to compute.

constexpr T mean() const noexcept;

Purpose
Obtain the mean of the values.

Return Value
The mean of the values if metric_mean or metric_all is specified and undefined otherwise.

tuple<bool,T,T> median() const noexcept;

Purpose
Obtain the median of the values.

Return Value
A tuple consisting of a Boolean indicating whether or not the median is unique, the first (and perhaps only) median and the second median (if it exists) if metric_median or metric_all is specified and undefined otherwise.

cconstexpr std::list<T> mode() const noexcept;

Purpose
Obtain the mode of the values.

Return Value
A list of the mode(s) of the values if metric_mode or metric_all is specified and undefined otherwise

cconstexpr T population_stddev() const noexcept;

Purpose
Obtain the population standard deviation of the values.

Return Value
The population standard deviation of the values if metric_population_stddev or metric_all is specified and undefined otherwise.

cconstexpr T sample_stddev() const noexcept;
Purpose
Obtain the sample standard deviation of the values.

Return Value
The sample standard deviation of the values if `metric_sample_stddev` or `metric_all` is specified and undefined otherwise.

```cpp
constexpr T population_var() const noexcept;
```

Purpose
Obtain the population variance of the values.

Return Value
The population variance of the values if `metric_population_var` or `metric_all` is specified and undefined otherwise.

```cpp
constexpr T sample_var() const noexcept;
```

Purpose
Obtain the sample variance of the values.

Return Value
The sample variance of the values if `metric_sample_var` or `metric_all` is specified and undefined otherwise.

4.1.2 Examples

```cpp
// example 1 --------------------------------------------------------------------------------
std::vector<int> v = {1, 1, 1, 2, 3, 4, 4, 4, 5, 6};
stats<double> s(stats<>::metric_all);
s.calc(v.cbegin(), v.cend());

std::cout << "\tmean: " << s.mean() << "\n";
auto result = std::move(s.median());
if(std::get<0>(result))
    std::cout << "\tmedian: " << std::get<1>(result) << "\n";
else
    std::cout << "\tmedian: " << (std::get<1>(result) + std::get<2>(result)) / 2.0 << "\n";
auto m = s.mode();
std::cout << "\tmode(s): ";
```
for (const auto &x : m)
    std::cout << x << " ";

std::cout << "\n\tpopulation standard deviation: " << s.population_stddev() << "\n";
std::cout << "\tsample standard deviation: " << s.sample_stddev() << "\n";
std::cout << "\tpopulation variance: " << s.population_var() << "\n";
std::cout << "\tsample variance: " << s.sample_var() << "\n\n";

// example 2 --------------------------------------------------------------------------------
std::pair<float, int> a[] = {{5.2f, 1}, {-1.7f, 2}, {9.2f, 5}, {4.4f, 7}, {0.3f, 3}};
std::stats<double> s;
s.metrics(std::stats<>::metric_all);

// perform required sorting before calculating
std::sort(a, a + 5,
    [] (const auto &x1, const auto &x2) { return x1.first < x2.first; });
s.calc(a, a + 5, [] (const auto &p) { return p.first; });

std::cout << "\tmean: " << s.mean() << "\n";

auto result = std::move(s.median());
if (std::get<0>(result))
    std::cout << "\tmedian: " << std::get<1>(result) << "\n";
else
    std::cout << "\tmedian: " << (std::get<1>(result) + std::get<2>(result)) / 2.0 << "\n";

auto m = s.mode();
std::cout << "\tmode(s): ";
for (const auto &x : m)
    std::cout << x << " ";

std::cout << "\n\tpopulation standard deviation: " << s.population_stddev() << "\n";
std::cout << "\tsample standard deviation: " << s.sample_stddev() << "\n";
std::cout << "\tpopulation variance: " << s.population_var() << "\n";
std::cout << "\tsample variance: " << s.sample_var() << "\n\n";

// example 3 --------------------------------------------------------------------------------
std::unordered_map<std::string, int> um;
um.insert("red", 3);
um.insert("green", 17);
um.insert("blue", 9);
auto [unique, m, firstm, secondm] = std::median(
    um.begin(), um.end(), (const auto &e) { return e.second; });
std::cout << "median: " << m; // 9
// example 2
std::vector<std::string> v{"cyan", "yellow", "magenta", "black");

if(auto& [one, m, m1, m2] = std::median(v); one)
    std::cout << "median: " << m;
else
{
    "(first) median 1: " << m1; // median: "cyan"
    "(second) median 2: " << m2; // median: "magenta"
}

5 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.

6 Acknowledgements

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

References