Remarks on Basis Statistics, P1708R9

Document: P3495R0
Date: November 13, 2024
Project: Programming Language C++, Library Working Group
Audience: SG19 & LEWG & LWG
Reply to: Oliver J. Rosten `oliver.rosten@gmail.com`
Mark Hoemmen, NVIDIA, `mhoemmen@nvidia.com`

Abstract

P1708R9, "Basic Statistics" proposes adding library functionality to compute various elementary statistics. While the proposal is very welcome, it is underspecified and offers inadequate exploration of the design space. This paper seeks to highlight the areas in need of refinement, with a view to improving the chances of a future iteration of P1708R9 making it into the standard.

CONTENTS

I. INTRODUCTION

The importance of statistical techniques can hardly be overstated, and the absence of any support for computing even elementary properties within the C++ standard library is notable. There are two active papers proposing to address this: [P1708R9],"Basic Statistics" and its companion [P2681R1], "More Basic Statistics". The goal of this paper is to identify shortcomings in the first of these, as it is further along the standardization process. However, many of the considerations are relevant to both.

The purpose of this paper is not to provide a counter-proposal. Rather, it seeks to help tighten up the design in areas where there appears to be an obvious defect, or to encourage a broader discussion where there are more subtle decisions to be made.

[P1708R9] proposes two mechanisms for computing statistics. First, there are free functions, designed to efficiently compute single statistics. To facilitate efficient computation of multiple statistics in a single pass, Accumulators are proposed. These are stateful objects to which data can be added, and multiple statistics extracted.

II. UNSPECIFIED VALUES

[P1708R9] presents a large surface area of unspecified behaviour. There are several sources, which will be grouped below according to whether they relate to identified properties of the inputs or the calculation itself. It is most likely undesirable to have so many sources of unspecified behaviour. This is not only potentially problematic at runtime, but begs a serious question as to what should happen during constant evaluation. The paper is silent on this matter, but there is precedent [P0533R9]:

> A call to a C standard library function is a non-constant library call ([defns.nonconst.libcall]) if it raises a floating-point exception other than `FE_INEXACT`.

First, it needs to be decided if this behaviour carries over to statistical functions/accumulators. Secondly, it needs to be specified whether particular floating-point exception flags are raised (and/or `errno` is set) in circumstances such as encountering infinities or NaNs.

According to [P1708R9], a statistic is unspecified if the ranges consumed by statistical functions / accumulators

1. Contain NaNs or infinities;
2. Have insufficient elements for a meaningful calculation. For example, the mean requires at least one element, and the sample variance at least two.

During the calculation, the result is unspecified if

1. Underflow or overflow occurs.

These will be dealt with in turn.

A. NaNs and Infinities

The paper provides no justification for why input of this form should yield an unspecified result. A first point of comparison is the `<cmath>` functions, which are precisely specified in such situations—see Annex F of the C standard [N2176]. Furthermore, WG21 has recently expressed a preference to make behaviour with infinities and NaNs well-defined, for example in [P3008R2] "Atomic floating-point min/max". In the latter, users are even given a choice between "propagate NaNs" and "treat NaNs as missing values."

An important question that [P1708R9] needs to answer is whether, when an infinity or NaN is encountered, `FE_INVALID` is raised. (Not forgetting the specific case of at least one positive infinity and at least one negative infinity which are added.)

B. Insufficient Elements

How to deal with this scenario is not so clearcut and so will not be discussed further in this section, but deferred to section IV which talks about the design space.

C. Underflow or Overflow

It is not obvious why the result should be unspecified in the case of underflow. For both underflow and overflow, specifying what happens during constant evaluation could be easily done by following [P0533R9]: raising any floating-point exception flag other than `FE_INEXACT` prohibits an expression from being a constant expression.

III. INADEQUATE STATE OF THE ART

The discussion on accumulators cites Boost Accumulators, providing no more details besides a broken link. Beyond fixing the link, it would be helpful for the reader to have some detail in [P1708R9] itself. Furthermore, the charts quantifying the performance of accumulators versus functions could be improved. There are no error bars, which has a certain irony for a paper on statistics. Moreover, lack of error bars notwithstanding, it appears that a merged accumulator is markedly inferior for computing the mean and variance. Why is this the case? Furthermore, how do these charts change when parallel execution is taken into account?

IV. THE DESIGN SPACE

A. Ranges of Insufficient Size

All of the statistics considered in [P1708R9] consume ranges which must have a non-zero number of elements. For example, the mean and variance require at least one element, and the sample variance at least two. The question as to what should happen if an insufficiently large range is presented is not trivial. However, [P1708R9] barely discusses the matter: dismissing `std::expected` in a rather cursory manner without any discussion of the broader design space and the pros and cons of different approaches.

It is certainly plausible that providing so many ways for clients to end up with unspecified values may produce a design which is rather user-hostile. At the very least, a detailed justification needs to be provided for why this isn't expected to be the case. Beyond this, there remains the question of what should happen during constant evaluation.

B. Accuracy

Given a data set, consider computing the mean. If these data are represented by floating-point numbers then, in general, the result will depend on the order in which the reduction is performed. In particular, clients may wish to prioritize accuracy over speed and so accumulate starting from values with the smallest magnitude to those of the largest. For the mean, the natural way to do this would be to presort the range before feeding it to one of the statistical functions. However, as things stand, there is no guarantee of any ordering of operations in the statistical functions and so pre-sorting in this way may be futile. Furthermore, if guarantees are provided for the free functions, how would these guarantees be replicated for accumulators?

There is also the question of whether free functions and accumulators should give the same result if they consume the same range. The implementation provided by [6] does not satisfy this property. Does this matter? Indeed, there are interesting questions about how an accumulator should work, not least when one considers its interaction with parallel acceleration (see below). Again, there is a design space here with tradeoffs which need to be properly discussed.

Finally, for statistics that involve floating-point operations, users may specify a result type with higher precision than the range's elements. In such cases the proposal should guarantee that computations happen in at least the precision of the result type, as has been done for linear algebra, https://eel.is/c++draft/linalg#algs.blas1.dot-7.

C. Parallelization

The free functions in [P1708R9] have overloads accepting execution policies, which is entirely reasonable. However, the paper is silent as to if/how accumulator objects might be amenable to parallel acceleration. One could imagine (recursively) dividing up the reduction of a range between workers which would call for a way to

1. Create a separate copy of the (stateful) reducer for each thread;
2. Initialize the reducer's state to the identity for the reduction operation;

3. Combine intermediate reduction results.

[P1708R9] does not try to define a reducer concept for user-defined reducers. However, it may be helpful to consider how it might be defined. Parallel programming models such as [7] may offer inspiration.

V. API CONCERNS

A. Defaulted Booleans

The presence of defaulted booleans could lead to client code which is cryptic. For example,

```
kurtosis(r, true, false)
```

is rather mystifying to the uninitiated. It may be preferable for each algorithms's input parameters to be carried by a struct. For example:

```
struct kurtosis_parameters {
  bool sample = true;
  bool excess = true;
};

template<class T, ranges::input_range R>
constexpr T kurtosis(
    R&& r, kurtosis_parameters params = {});
```

B. Expressive Return Types

As things stand, functions such as `mean_variance` return a `std::pair`. It may be preferable to instead return a named struct, whose fields express what is being returned.

C. Constructing Accumulators

Is there any good reason for accumulators not to support construction with a range of elements?

D. Return value of accumulators' `operator()`

Currently, this operator returns `void`. Has any consideration been given to returning a reference to the accumulator itself? Given an accumulator `acc` and `data`, this could support patterns such as:

```
if(auto mean = acc(data).mean(); mean > 0)
   ...
```

E. Explicit Template Parameters

Overloads for the statistical functions exist in which clients can explicitly specify the return type, for instances where this is different from the input range's value type, viz.

```
template<class T, ranges::input_range R>
constexpr auto mean(R&& r) -> T;
```

A better alternative may be to imitate `std::reduce` by taking an initial value as an input parameter, for example:

```
template<class T, ranges::input_range R>
constexpr auto mean(R&& r, T init) -> T;
```

This has the following advantages:

1. Users would not need to give an explicit template parameter `T` for the result; they could just pass in a number like 0.0.
2. It would support use cases like computing statistics over part of a range, then continuing the computation on the rest of the range.
3. It would help distinguish overloads. The Standard Algorithms generally don't have overloads with the same number of arguments but different numbers of template parameters.
4. The Standard Algorithms don't generally permit explicit template arguments.

VI. CONCLUSION

A suite of statistical functions would be a beneficial addition to the C++ Standard Library. However, significant extra work is required to bring the design of [P1708R9] up to the required level of rigour. It is hoped that this paper will help the endeavour.

REFERENCES

[P1708R9] Richard Dosselmann, Basic Statistics https://isocpp.org/files/papers/P1708R9.pdf
[P2681R1] Richard Dosselmann, More Basic Statistics https://isocpp.org/files/papers/P2681R1.pdf
[N2176] ISO/IEC 9899:2018 Standard for Programming Languages — C

[P3008R2] Gonzalo Brio Gadeschi and David Sankel, Atomic floating-point min/max https://isocpp.org/files/papers/P3008R2.html
[P0533R9] Edward J. Rosten and Oliver J. Rosten, `constexpr` for `<cmath>` and `<cstdlib>`https://isocpp.org/files/papers/P0533R9.pdf
[6] Richard Dosselmann, Reference Implementation https://github.com/dosselmann/statistics/blob/main/statistics.hpp
[7] https://kokkos.org/kokkos-core-wiki/API/core/builtinreducers/ReducerConcept.html