Less `constexpr` for `<cmath>`

Document: -

Date: March 09, 2021

Project: Programming Language C++, Library Working Group

Audience: SG6 → LEWG → LWG

Reply to: Nicholas G. Timmons (ngt26@cam.ac.uk)

Abstract

Proposal P1383 [2] calls for the inclusion of `constexpr` to the headers of of the mathematical functions defined in `<cmath>` and `<complex>` to would allow the functions to be evaluated at compile time. This has the potential to cause problems of determinism as most mathematical libraries linked at run-time do not produce correctly-rounded results. As a result, the value which is statically compiled into the executable for a given call to `sin` may not match the result produced at run-time for the same input. In this position paper we highlight this issue and propose ways that `constexpr` could be used with the mathematical functions in the future.

Contents

1 Introduction

Proposal P1383 wants to add a `constexpr` to elementary functions and uses the justification of compilers already performing constant folding at compile time to justify that this should be an acceptable change.

In theory, this is something that should be true as the *IEEE-754 Standard for Floating-Point arithmetic* [1] states that the result of calls to the elementary functions should return a correctly-rounded result. However, current run-time mathematical libraries which ship with operating systems are not correctly-rounded. This means if a program with precomputed mathematical results is linked to a mathematical library there is a chance that the call which was statically evaluated may not match calls to the same function with the same input.

Many people believe one bit of error in floating-point does not affect the final output of a program, but this is simply not true. Many applications rely on bit-precise floating-point mathematics — and they have every reason to. Much work was put into IEEE-754 to guarantee deterministic and portable results so that these results could be relied on when needed. Problems of non-determinism and inconsistency with results only occur when the rules are not followed and by the nature of it being such a popular and widely adopted standard, people do rely on the results.

This paper seeks to delay the addition of the `constexpr` specifier to the elementary functions in `<cmath>` until the common mathematical libraries which are used provide results that are deterministic between implementations.

We will present the current problems with *Constant Folding* for mathematical functions as it is implemented in *gcc* and *Clang* and then show how these existing problems will be present if `constexpr` is added to the elementary functions in `<cmath>`.

As the addition of `constexpr` would be a benefit to the standard library, at the end of the paper is a proposal for possible alternatives which can be implemented.

2 Existing Constant Folding Problem

The core of the problem with evaluating the elementary functions at compile time is that the current libraries which are used at run-time are not correctly-rounded. It is impossible to give precise results to the scale of the problem in 64-bit floating-point, but

for 32-bit floating-point, we can say that some functions have a high proportion of all results incorrectly-rounded. For example, in *glibc 2.27* 22.67 of all valid inputs to `acosh` produce incorrectly-rounded results in 32-bit. Figure 1 shows a comparison of different implementations of some of the elementary functions with highlights for where an incorrectly-rounded result is produced. You will notice that the different implementations result not only in incorrect results but a mismatch between libraries of where the incorrect results reside.

If the libraries all produce the same outputs for the same input (regardless of whether they are correctly-rounded) then we would not have a portability issue where a compiled result may not match a run-time one. Unfortunately, there is no common canonical result list that is adhered to.

How does this affect the use of *Constant Folding* for mathematical functions that are already being used in many compilers? For that, we need to show how different compilers are implementing this feature.

For Clang, this is implemented in its *Constant Folding* optimisation pass. When a call to an elementary function is found, it will try and evaluate the result. Using the example of `tanf`(x): the call to `tanf` is identified, the type is promoted to `double`, and the return value is fetched using the **host** library and then cast back to the input precision.

For gcc, similar steps are taken except that the value is calculated using GNU MPFR to produce the correctly-rounded value at the input precision.

The Clang implementation has four problems; it is using the host library which could be implemented to any precision, it evaluates 32-bit as 64-bit, the final result is not guaranteed to be correctly-rounded, the final result may not match the run-time library. The gcc implementation only has that final problem.

All of these factors could result in a situation where `mathfn`(x) $\neq$ `mathfn`(x) if one of the sides is evaluated at compile time.

3 Future Constexpr Problems

If `constexpr` was to be added to these functions there would be a requirement for the compilers to adhere to the command. As there is currently no good solution for what number should be returned for any input to these functions, there is a high chance of the values not matching between compilers. Regardless of the run-time environment that the executable is finally ran in.

This would make code compilation non-deterministic due to not knowing which value would be output, and then non-portable due to not knowing if the run-time linked mathematical library will match with the results generated at compile-time.

It must be stated that there is already other a long list of other issues which cause similar problem, but it is the authors belief that the list should not be expanded if it can be avoided.

4 Solution

Many compilers support the flag `-ffast-math` which are used to signal compliance with the IEEE-754 standard. The existing constant folding of the mathematical functions should be moved to only take place when compiling with this flag enabled as the current state of the mathematical libraries cannot guarantee that the results returned will be compliant with the standard. This should be the case for any externally linked library with no guarantee of correctness.

To reach the goal of adding `constexpr` to the elementary mathematical functions there must be further development of the existing mathematical libraries to support correct-rounding as dictated by IEEE-754. Alternatively, only allow the specifier to be added to the `cr{fn}` version of the mathematical functions which were proposed in *ISO/IEC TS 18661-4:2015* which looks to be adopted into *C2x*. This would mean that the result had to be correctly-rounded and guarantee compile-time and run-time results are equal.

5 Conclusion

The current proposals to allow for the `constexpr` specifier to be added to the mathematical functions is incompatible with the current mathematical libraries in common use, as they do not comply with the IEEE-754-2019 standard because they do not provide the correctly-rounded result for the valid inputs in their domain. If the specifier was allowed it would introduce non-determinism between the compiled value and the run-time value for the same inputs if the same versions of the same library were not used in both instances. This would reduce the reliability of mathematical computing in `C++`.

References

[1] "IEEE Standard for Floating-Point Arithmetic". In: *IEEE Std 754-2019 (Revision of IEEE 754-*

2008) (2019), pp. 1–84. DOI: 10.1109/IEEESTD.2019.8766229.

[2] Edward J Rosten and Rosten.Oliver J. “More constexpr for ¡cmath¿ and ¡cstdlib¿”. In: (2019).

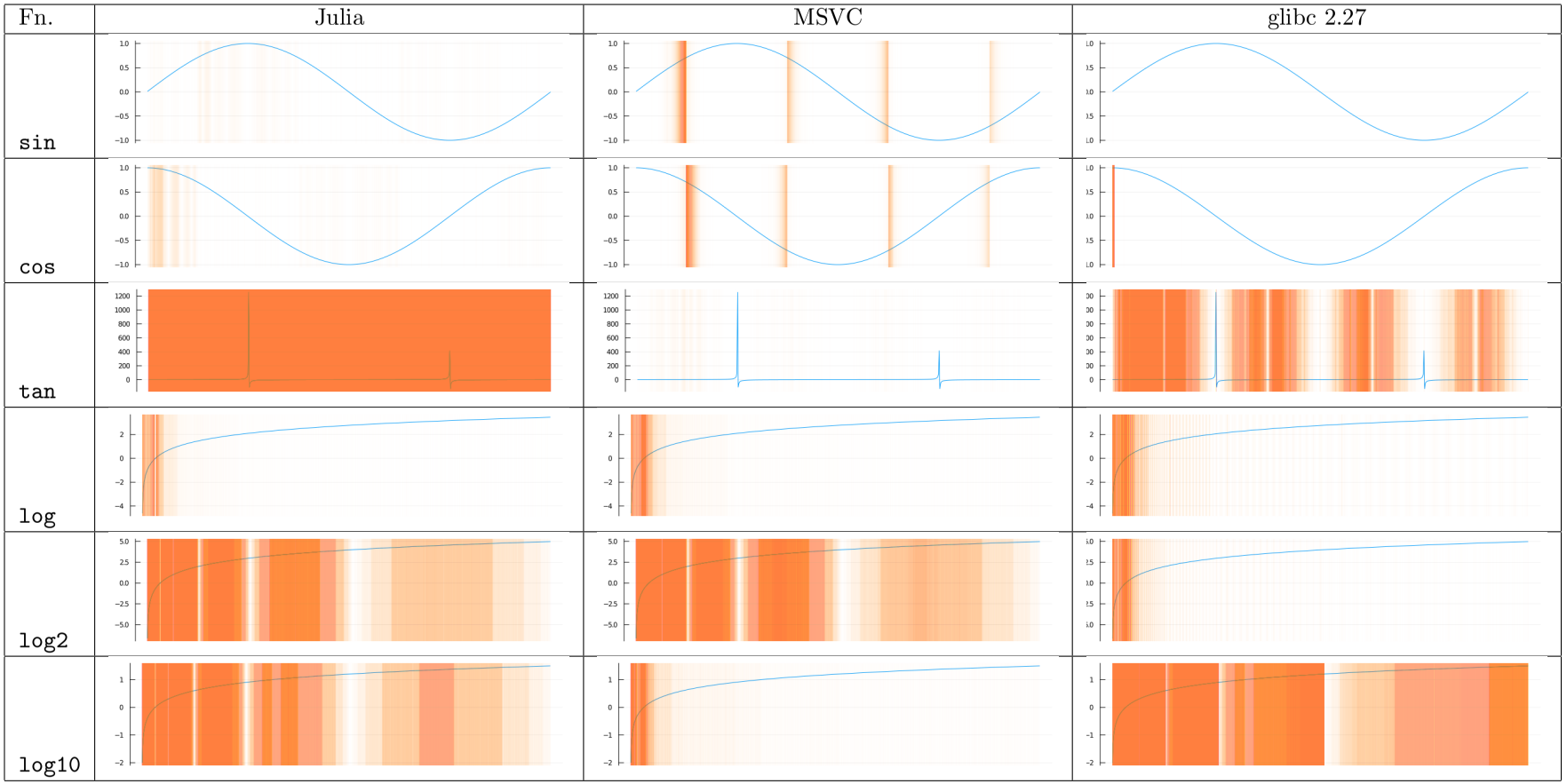

Figure 1: Sample of elementary functions showing the distribution of incorrectly-rounded results on the CPU for Julia, MSVC and glibc 2.27. Different libraries produce error in different places, meaning $f(x)$ in one library might not be equal to $f(x)$ in another library for some inputs — this is a threat to portability. **NOTE:** *In this table a translucent vertical orange line is drawn for each incorrectly rounded result, for some graphs such as Julia* `sin`*, where there are relatively few incorrectly-rounded results the errors are quite faint. This is needed so that most functions with lots of error are not all an orange bar.*