Extensible Math Functions for C++

Document #:	P4188R1
Date:	2026-07-04
Project:	Programming Language C++
Audience:	SG6 SG18 SG20 LEWG

Reply-to: Stéphane Gros-Lemesre

<stephane.groslemesre@gmail.com>

Abstract

This paper proposes making standard library mathematical functions extensible to user-defined types via a member function or ADL. To preserve backward compatibility, the proposed approach introduces a new `std::math` sub-namespace. The new namespace also offers an opportunity to introduce a C++-idiomatic math library leveraging overloading and templates, with an emphasis on teachability.

Contents

1 Motivation **3**
- 1.1 Problem . . . 3
  - 1.1.1 Naive call sites . . . 3
  - 1.1.2 The `using`-statement workaround . . . 3
  - 1.1.3 Standard Library numeric types . . . 5
- 1.2 High-level proposal . . . 6
- 1.3 Use Cases . . . 6
  - 1.3.1 Semantic Types . . . 6
  - 1.3.2 Arbitrary types . . . 6
  - 1.3.3 Code usage examples . . . 6
- 1.4 Context . . . 10

2 Impact on the Standard **10**
- 2.1 `std::math` . . . 10
  - 2.1.1 Proposed naming . . . 11
- 2.2 Bridging through opt-in . . . 12
- 2.3 Scope . . . 13
  - 2.3.1 Future extensions . . . 13
- 2.4 Summary . . . 13

3 Design Principles **14**
- 3.1 Two spaces, different behaviours . . . 14
- 3.2 New contract . . . 14
- 3.3 CPO-like customization . . . 14
  - 3.3.1 Dispatch . . . 14
  - 3.3.2 Alternative to plain CPO . . . 15
- 3.4 Single header . . . 15
- 3.5 Canonical function names . . . 15
- 3.6 Customisation-level specifications . . . 15

4 General Specifications for built-in arithmetic types **16**
- 4.1 Conservative addition . . . 16
- 4.2 `constexpr` . . . 16
- 4.3 `[[nodiscard]]` . . . 16

4.4 Error Handling 16

5 Technical Specification **16**
5.1 Functions 16
5.1.1 Power Functions 16
5.1.2 Exponential 18
5.1.3 Trigonometric 21
5.1.4 Hyperbolic 23
5.1.5 Rounding 25
5.1.6 Sign 26
5.1.7 Classification 28
5.1.8 Norm 30
5.2 Proposed Implementation 31
5.2.1 CPO 31
5.2.2 Compatibility layer 32
5.2.3 Alternative implementation with [P2806R3] (do expressions) and [P2826R2] (replacement functions) 33

6 References **33**

1 Motivation

1.1 Problem

User-defined types are central to C++ programming. The standard library inherited from the Standard Template Library its foundational principle of separating data structures and algorithms through generic programming. Containers, iterators, Customization Point Objects (CPOs), custom operators all contribute to enable a natural integration of any type, user-defined or not, in a common ecosystem.

Mathematical functions are a notable gap in this picture. Unlike operators, they are not part of overload resolution in a way that extends to user-defined types, and unlike CPOs, there is no standard hook to participate in. An author implementing an arithmetic-like type will find it easy to implement binary operations such as addition, subtraction, multiplication, and division; comparisons, hashing, or integration with containers for their user-defined type, until they face the problem of implementing the square-root operation or similar mathematical function.

The fundamental problem with defining mathematical functions for a custom type is that because there is no well-defined contract as there is for the operators, and no CPO in the standard library for that purpose, the mechanism for their integration is not in the hands of the author. A custom implementation can be offered, but whether it will be picked up by generic code depends entirely on how that code was implemented.

This is not what good separation of concerns looks like: for the author of numeric types, providing custom mathematical functions is a leap of faith, while for generic code authors, supporting custom types might not be the primary concern and requires a significant amount of boilerplate.

1.1.1 Naive call sites

It is difficult to quantify the amount of generic code calling mathematical functions without support for user-defined numeric types. This is made even more difficult as these functions can live in the global namespace if `<math.h>` is included. Using GitHub search functionality shows 115k C++ files using the idiom `using std::pow;` for 3.5M C++ files calling `pow(`, of which 741k C++ files calling explicitly `std::pow(`.

Although these numbers have to be taken with caution, as there are other ways to leverage ADL for the `pow` function than `using std::pow;` specifically, and some of the qualified `std::pow` calls might come from non-generic code, it seems likely that a non-negligible amount of generic code does not allow custom types because of this extensibility limitation.

The library **nholthaus/units**, for instance, provides `units::math::sqrt` which calls `std::sqrt` directly on the underlying scalar value, which means it does not enable ADL resolution and thus does not extend to custom underlying types. [nholthaus-units]

This should not be surprising: the responsibility of making custom types work intuitively should belong to their authors, not to their users or third-party. It takes conscious effort for a generic library author to anticipate the needs for wrappers of hypothetical custom types and explicitly provide support for them.

Yet, the consequence of this status quo is substantial: the successful integration of a custom-type in a code-base no longer depends on its own design and implementation, but on the implementation of every other piece of code that will manipulate it. It is effectively a dependency of that type on all of the generic code it interacts and will interact with in the future. Such open-ended, long-term consequences are often unacceptable.

This is significant because custom numeric types can be used to support **correctness and safety** by encoding constraints that are enforced by the type system automatically. This pattern is similar to *newtypes* in Rust, and is sometimes referred to as "semantic-typing". Whether it is by preventing inappropriate conversion or ensuring unit consistency, it can prevent entire classes of bugs and is a well-understood tool for writing safer numeric code. As it stands, its practicality in C++ is significantly restrained by how well such custom types compose with the broader ecosystem, and mathematical functions are a substantial weak point in that composition.

1.1.2 The `using`-statement workaround

The idiomatic workaround is to enable ADL via a `using` declaration:

```
using std::sqrt;
return sqrt(x);
```

This allows a user-defined mathematical function in the same namespace as the type passed into the function to be found via ADL, while falling back to the `std`-defined version for built-in types. However, this idiom has a critical limitation: it requires a statement, and is therefore unavailable in contexts that only accept expressions.

1.1.2.1 Statement-only

Constructor member initializer lists:

```
MyType(A aSq, B b, C c)
    : a(sqrt(aSq)),   // std::sqrt not found through conversion
      b(b),
      c(abs(c))     // std::abs not found through conversion
{}
```

The same applies to default member initializers:

```
struct Foo {
    double x = sqrt(v);  // std::sqrt not found through conversion
};
```

`requires` expressions:

```
template<typename T>
concept numeric = requires(T x) {
    { abs(x) };  // requirement fails
};
```

There are other, less obvious situations where the same limitation applies, such as constant expressions (array size from a new-type) and template arguments.

In all of these cases, the programmer faces the same three unsatisfactory choices:

Option 1: Call `std::` explicitly

```
MyType(A aSq, B b, C c)
    : a(std::sqrt(aSq)),  // silently breaks extensibility
      b(b),
      c(std::abs(c))
{}
```

This compiles and works for built-in types, but silently cuts off any user-defined overload.

Option 2: Restructure to allow a statement

For member initializer lists, this means moving initialization to the constructor body:

```
MyType(A aSq, B b, C c)
    : b(b)
{
    using namespace std;
    this->a = sqrt(aSq);  // requires a to be default-constructible
    this->c = abs(c);   // loses const and reference member support
}
```

This restores ADL but members can no longer be `const` or references, and all members must be default-constructible. Not all contexts can be restructured this way.

Option 3: Write boilerplate helper functions

```
template<typename T> auto my_sqrt(T&& x) {
    using std::sqrt;
    return sqrt(std::forward<T>(x));
}

MyType(A aSq, B b, C c)
    : a(my_sqrt(aSq)),
      b(b),
      c(my_abs(c))
{}
```

This restores extensibility but requires every author of generic code to write and maintain their own dispatch layer; one wrapper per math function, 45+ at least, and likely more in the future (see [P3935R1]).

The ownership of these wrappers is unclear. Custom numeric-type authors should probably embed them in their math function implementations, but authors of generic code using such functions cannot rely on this being provided and may need to implement them redundantly to support a wider range of types.

1.1.2.2 Existing Practice

The existence of multiple widely-used C++ libraries implementing exactly this machinery is evidence that there are real use-cases and that the status quo is encouraging unnecessary duplication.

mp-units, **Eigen**, and **Boost.Units** have all independently converged on the same core mechanism: bring `std::sqrt` into scope and let ADL do the work.

```
using std::sqrt;
return sqrt(x);
```

The surrounding machinery differs:

— mp-units adds an explicit member function check. [mp-units]

— Eigen wraps the dispatch in a traits struct with SIMD specialisations and relies on macros to minimize the duplication between different functions. [eigen-math]

```
#define EIGEN_MATHFUNC_IMPL(func, scalar) \
  Eigen::internal::func##_impl<typename \
  Eigen::internal::global_math_functions_filtering_base<scalar>::type>
```

— Boost.Units applies the pattern to the inner value of a quantity type. [boost-units-sqrt]

But the fundamental approach is identical in all three.

1.1.2.3 Teachability

A difficulty with the `using` ADL idiom is its limited discoverability. Without specific guidance, it takes very specific circumstances to stumble on the problem, let alone its solution. Calling the `std` operations looks correct, compiles cleanly, and works as expected with native types. It is only when this code is used with custom types that its limitation becomes problematic.

The following exchange from nholthaus/units GitHub issue #39[nholthaus-issue39] is instructive. A user reports that generic algorithms using ADL-found math functions do not work with unit types, and the library author responds:

> *"Honestly, I guess I just never use ADL because I pretty much exclusively use fully qualified namespaces in my code, and I didn't put thought into it."*

In the current situation, it is easy to do the wrong thing, and difficult to do the right one.

1.1.3 Standard Library numeric types

The problem also impacts the standard library when non-arithmetic numeric types are defined, such as for simd.

Specifically, Mateusz Pusz, the author of the mp-units library and main contributor to the C++26 unit library [P3045R8], called out these very limitations at C++ on Sea 2025, specifically calling for a proposal to introduce CPOs for mathematical functions. [pusz-cpponsea2025]

The linear algebra facilities currently use helper functions in order to allow custom arithmetic types, and it has been acknowledged in committee that having a well defined contract to define these customization points would be very helpful.

1.2 High-level proposal

This proposal suggests introducing a new `std::math` namespace where extensible basic math functions would be defined. The extension would be made possible through a member function defined on the type, or via ADL.

Implementations would be provided for built-in types such as floating-points and for `std::complex` where appropriate.

In addition, an opt-in compatibility layer would also be added for basic math functions defined in the `std` namespace, opening a conservative and targeted bridge between legacy code calling `std::` operations directly and custom-types.

1.3 Use Cases

1.3.1 Semantic Types

Often ambiguously called "strong-typing", we will use the term "semantic-typing" to describe the practice of preferring types that are specific and carry more semantic over general-purpose types. For instance, preferring a `GeoCoordinate` type over an `std::pair` to store a latitude and longitude.

In other languages (Haskell, Rust, Python), this is sometimes facilitated by a "newtype" feature. Newtypes are typically strictly-typed thin wrappers around existing primitive or data types for semantic-typing purposes.

C++26 Quantities and Units library [P3045R8] offers an opportunity to replace built-in types by types carrying the semantic of the unit the quantity is expressed as. E.g.: `std::quantity<std::si::metre>` instead of `double`.

Another example would be defining an index type paired with the container it is meant to index into, offering compile-time guarantees that it is not used with the wrong container.

For these examples, changing the type of variables in existing code would work for operator usage, hashing and other common operations if the replacement type defines these operations, but it would fail as soon as a mathematical function is called without resorting to the `using`-statement workaround.

This proposal is not limited to scalar types, and would also allow defining specialisations of mathematical functions for multidimensional types such as vectors, quaternions, tensors, or matrices.

1.3.2 Arbitrary types

Any type can leverage the extension mechanism presented in this paper without restriction. It is for the implementers to decide whether a specific mathematical operation is meaningful for a specific type. This proposal does not define an explicit boundary in this regard, but defines the dispatching mechanism allowing the extension.

This can be likened to operator overloading where the facility for defining the operation is made available for implementers to leverage as they see fit without any contract enforcement beyond the admitted operator signatures.

1.3.3 Code usage examples

1.3.3.1 Newton p^{th} root

```
template<typename T>
T newton_pth_root(T a, T x, int p, T tolerance)
{
```

```
    T prev;
    do
    {
        prev = x;
        // f(x) = x^p - a, f'(x) = p * x^(p-1)
        x = ((p-1)*x + a / std::math::pow(x, p-1)) / p;
    }
    while (std::math::abs(x - prev) > tolerance);
    return x;
}
```

This implementation would work out of the box with `T` as `float`, `double`, `long double`, but also with `T` as a unit such as a surface in m^2, or even a user-defined matrix type, assuming an implementation for `std::math::pow` and `std::math::abs` are provided.

It is worth noting that this function could be implemented in the current ecosystem with one or two `using` statements: either with `using std::pow;` and `using std::abs;`, or `using namespace std;` and then calling unqualified `pow` and `abs`, enabling ADL. The benefits of the proposed approach are perceived as providing a better-defined correct path with clearer separation of concerns (making it easier to do the right thing), and addressing the other limitation of the `using` workaround that cannot be used in expression-only contexts.

1.3.3.2 Compatibility layer example

A typical use case for the compatibility layer is a custom numeric type used with an existing library that calls `std::sqrt` directly and cannot be modified:

```
// Generic library using std::sqrt directly
namespace someLib
{
    template<typename T>
    auto someFunction(T&& someValue)
    {
        // some code...
        return std::sqrt(std::forward<T>(someValue));
    }
}

// User's custom type, with its own square root implementation in its namespace
namespace mylib
{
    struct Scalar { ... };
    Scalar sqrt(Scalar x) { ... }
}

// Opt in to the compatibility layer
template<>
inline constexpr bool std::is_math_extensible<mylib::Scalar> = true;

// Now where std::sqrt(mylib::Scalar{}) is used in the library, it is
// forwarded to mylib::sqrt via std::math::sqrt

someLib::someFunction(mylib::Scalar{5.2});  // now works
```

The call to `someFunction` with an argument of a custom type is made possible without changing the function itself, through the compatibility layer.

1.3.3.3 Eigen math function implementation

Here is how Eigen handles implementing the `sqrt` function for just scalar and complex types.

1.3.3.3.1 Current

```
template <typename T, typename dummy = void>
struct global_math_functions_filtering_base {
  typedef T type;
};

template <typename T>
struct always_void {
  typedef void type;
};

template <typename T>
struct global_math_functions_filtering_base<
    T,
    typename always_void<typename T::Eigen_BaseClassForSpecializationOfGlobalMathFuncImpl>::type>
{
  typedef typename T::Eigen_BaseClassForSpecializationOfGlobalMathFuncImpl type;
};

#define EIGEN_MATHFUNC_IMPL(func, scalar) \
  Eigen::internal::func##_impl< \
    typename Eigen::internal::global_math_functions_filtering_base<scalar>::type>
#define EIGEN_MATHFUNC_RETVAL(func, scalar) \
  typename Eigen::internal::func##_retval<  \
      typename Eigen::internal::global_math_functions_filtering_base<scalar>::type>::type
```

```
/****************************************************************************
 * Implementation of sqrt/rsqrt                                             *
 ****************************************************************************/

template <typename Scalar>
struct sqrt_impl {
  EIGEN_DEVICE_FUNC static EIGEN_ALWAYS_INLINE Scalar run(const Scalar& x) {
    EIGEN_USING_STD(sqrt);
    return sqrt(x);
  }
};

// Complex sqrt defined in MathFunctionsImpl.h.
template <typename ComplexT>
EIGEN_DEVICE_FUNC constexpr ComplexT complex_sqrt(const ComplexT& a_x);

// Custom implementation is faster than `std::sqrt`, works on
// GPU, and correctly handles special cases (unlike MSVC).
template <typename T>
struct sqrt_impl<std::complex<T>> {
  EIGEN_DEVICE_FUNC static EIGEN_ALWAYS_INLINE std::complex<T> run(const std::complex<T>& x)
  { return complex_sqrt(x); }
};
```

```
EIGEN_STRONG_INLINE EIGEN_DEVICE_FUNC half sqrt(const half& a) {
#if defined(EIGEN_CUDA_ARCH) || defined(EIGEN_HIP_DEVICE_COMPILE)
  return half(hsqrt(::__half(a)));
#else
  return half(::sqrtf(float(a)));
#endif
}
```

```
EIGEN_STRONG_INLINE EIGEN_DEVICE_FUNC bfloat16 sqrt(const bfloat16& a)
{ return bfloat16(::sqrtf(float(a))); }
```

1.3.3.3.2 With P4188

```
namespace Eigen {
    template <typename ComplexT>
    EIGEN_DEVICE_FUNC constexpr ComplexT complex_sqrt(const ComplexT& a_x);

    template <typename T>
    EIGEN_DEVICE_FUNC EIGEN_ALWAYS_INLINE std::complex<T> sqrt(const std::complex<T>& x) {
        return complex_sqrt(x);
    }

    EIGEN_STRONG_INLINE EIGEN_DEVICE_FUNC half sqrt(const half& a) {
#if defined(EIGEN_CUDA_ARCH) || defined(EIGEN_HIP_DEVICE_COMPILE)
        return half(hsqrt(::__half(a)));
#else
        return half(::sqrtf(float(a)));
#endif
    }

    EIGEN_STRONG_INLINE EIGEN_DEVICE_FUNC bfloat16 sqrt(const bfloat16& a) {
        return bfloat16(::sqrtf(float(a)));
    }
}
```

1.3.3.4 Initializing an immutable structure with a square root

Suppose we have an immutable `struct` as follows:

```
template<typename T>
struct ElementWithCachedSqrt {
    const T element;
    const T element_sqrt;

    ElementWithCachedSqrt(T e)
        : element(e),
          element_sqrt(std::sqrt(e))
    {}
};
```

In this form it can only accept elements of type `T` for which an `std::sqrt` definition exists (float, double, std::complex, etc.)

1.3.3.4.1 Workaround

```
template<typename T>
auto sqrt_dispatch(T&& x) {
    using std::sqrt;
    return sqrt(std::forward<T>(x));
}

template<typename T>
struct ElementWithCachedSqrt {
    const T element;
    const T element_sqrt;

    ElementWithCachedSqrt(T e)
```

```
        : element(e),
          element_sqrt(sqrt_dispatch(e))
    {}
};
```

This would make it work in the current version of the language.

1.3.3.4.2 With P4188

```
template<typename T>
struct ElementWithCachedSqrt {
    const T element;
    const T element_sqrt;

    ElementWithCachedSqrt(T e)
        : element(e),
          element_sqrt(std::math::sqrt(e))
    {}
};
```

1.4 Context

The mathematical functions of C++ have been inherited from C, before the introduction of namespaces in the language and therefore lived in the global name space. With C++98, the `<cmath>` header was added, as the new correct way to access the math functions inherited from C. The `<math.h>` header was retained, though, for backward compatibility, leading to the present-day situation where many mathematical functions are accessible in both the global namespace and `std`. C++98, C++11, and C++17 also introduced additional functions to `<cmath>`. C++29 could add more to it [P3935R1].

As a result of this evolution, these functions are unlike other parts of the standard library:

— Their names do not follow the `snake_case` pattern.
— There are several variants of each function differentiated by prefix or suffix instead of using function overloading.
— Their error handling uses `errno` instead of exceptions or the more modern `std::expected`.

Encountering this API is surprising and confusing for new C++ learners without a C background, as the justification for these divergences are largely historical.

Another consequence of this long progression is also that there is a huge body of code relying on all of these successive evolutions. These APIs are essentially frozen, as the contracts established over the years can't reasonably be invalidated.

2 Impact on the Standard

Consequently, based on the points discussed above, this proposal aims to address some of the limitations to the extensibility of the math functions in C++ while preserving existing behaviours.

Since the core problem is the absence of a standard extension mechanism, the solution must be introduced by the standard library itself. It cannot be provided by user code or third-party libraries alone.

2.1 `std::math`

Specifically, this proposal introduces a new namespace `std::math` to clearly separate the new customizable contract from the current functions inherited from C.

— Math functions in this namespace allow customization through a member function or ADL.
— Their naming strives to be consistent with modern standard library convention.
— They leverage overloading and generic programming instead of prefix and suffix.
— Error handling for user-defined types can leverage exceptions, contracts or other modern error handling patterns.

In addition, tools offering autocompletion would also benefit from a focused, dedicated namespace narrowing the relevant options early by category rather than name only.

The introduction of a new namespace and a new contract also provides an opportunity to resolve longstanding ambiguities such as the semantics of `abs` and `norm`. The specific meaning of math functions becomes especially consequential once user-defined customization is possible.

Finally, the combination of a reduced number of function names, more predictable naming patterns, better autocompletion, and clearer semantic of the operations should help improve the teachability issues mentioned previously.

2.1.1 Proposed naming

2.1.1.1 Guiding principle

To facilitate teachability and avoid surprising the user, the naming of functions in `std::math` favours well-established names over strictly descriptive ones. Names that are consistently used across languages — `sqrt`, `cbrt`, `pow`, `hypot`, `exp`, `sin`, `cos`, `abs` — are kept as-is. C and C++ have, over decades, contributed to establishing a de-facto standard vocabulary for mathematical functions that has been adopted across virtually every mainstream programming language. Departing from that vocabulary would make the new namespace less familiar, not more, which is contrary to the teachability goal.

Where `<cmath>` names carry C-era type-distinguishing prefixes or suffixes (`fabs`, `fabsf`, `fabsl`, `fmod`) that exist solely because C lacked overloading, these affixes are dropped. `std::math` uses overloading and templates instead, making such variants redundant. The resulting names are cleaner and more consistent with the rest of the standard library.

2.1.1.2 Cross-language naming reference

`<cmath>`	Python	Java	Rust	Julia	Haskell	Go
`sqrt`	`math.sqrt`	`Math.sqrt`	`f64::sqrt`	`sqrt`	`sqrt`	`math.Sqrt`
`cbrt`	`math.cbrt`	`Math.cbrt`	`f64::cbrt`	`cbrt`	—	`math.Cbrt`
`pow`	`math.pow`/`**`	`Math.pow`	`f64::powf`	`^`	`**`	`math.Pow`
`hypot`	`math.hypot`	`Math.hypot`	`f64::hypot`	`hypot`	—	`math.Hypot`
`abs`/`fabs`	`abs`/`math.fabs`	`Math.abs`	`f64::abs`	`abs`	`abs`	`math.Abs`
`exp`	`math.exp`	`Math.exp`	`f64::exp`	`exp`	`exp`	`math.Exp`
`exp2`	`math.exp2`	—	`f64::exp2`	`exp2`	—	`math.Exp2`
`expm1`	`math.expm1`	`Math.expm1`	`f64::exp_m1`	`expm1`	—	`math.Expm1`
`log`	`math.log`	`Math.log`	`f64::ln`	`log`	`log`	`math.Log`
`log2`	`math.log2`	—	`f64::log2`	`log2`	`logBase 2`	`math.Log2`
`log10`	`math.log10`	`Math.log10`	`f64::log10`	`log10`	`logBase 10`	`math.Log10`
`log1p`	`math.log1p`	`Math.log1p`	`f64::ln_1p`	`log1p`	—	`math.Log1p`
`sin`	`math.sin`	`Math.sin`	`f64::sin`	`sin`	`sin`	`math.Sin`
`cos`	`math.cos`	`Math.cos`	`f64::cos`	`cos`	`cos`	`math.Cos`
`tan`	`math.tan`	`Math.tan`	`f64::tan`	`tan`	`tan`	`math.Tan`
`asin`	`math.asin`	`Math.asin`	`f64::asin`	`asin`	`asin`	`math.Asin`
`acos`	`math.acos`	`Math.acos`	`f64::acos`	`acos`	`acos`	`math.Acos`
`atan`	`math.atan`	`Math.atan`	`f64::atan`	`atan`	`atan`	`math.Atan`
`atan2`	`math.atan2`	`Math.atan2`	`f64::atan2`	`atan2`	`atan2`	`math.Atan2`
`sinh`	`math.sinh`	`Math.sinh`	`f64::sinh`	`sinh`	`sinh`	`math.Sinh`
`cosh`	`math.cosh`	`Math.cosh`	`f64::cosh`	`cosh`	`cosh`	`math.Cosh`
`tanh`	`math.tanh`	`Math.tanh`	`f64::tanh`	`tanh`	`tanh`	`math.Tanh`
`asinh`	`math.asinh`	`Math.asinh`	`f64::asinh`	`asinh`	`asinh`	`math.Asinh`
`acosh`	`math.acosh`	`Math.acosh`	`f64::acosh`	`acosh`	`acosh`	`math.Acosh`
`atanh`	`math.atanh`	`Math.atanh`	`f64::atanh`	`atanh`	`atanh`	`math.Atanh`
`ceil`	`math.ceil`	`Math.ceil`	`f64::ceil`	`ceil`	`ceiling`	`math.Ceil`
`floor`	`math.floor`	`Math.floor`	`f64::floor`	`floor`	`floor`	`math.Floor`
`round`	`round`	`Math.round`	`f64::round`	`round`	`round`	`math.Round`
`trunc`	`math.trunc`	—	`f64::trunc`	`trunc`	`truncate`	`math.Trunc`
`copysign`	`math.copysign`	`Math.copySign`	`f64::copysign`	`copysign`	—	`math.Copysign`

<cmath>	Python	Java	Rust	Julia	Haskell	Go
fmod	math.fmod / %	%	%	rem / %	mod	math.Mod
modf	math.modf	—	—	modf	properFraction	math.Modf
isnan	math.isnan	Double.isNaN	f64::is_nan	isnan	isNaN	math.IsNaN
isfinite	math.isfinite	Double.isFinite	f64::is_finite	isfinite	—	—
isinf	math.isinf	Double.isInfinite	f64::is_infinite	isinf	isInfinite	math.IsInf
isnormal	—	—	f64::is_normal	—	—	—

— denotes cases where there is no exact corresponding function in the language.

2.1.1.3 Kept names

The vast majority of function names are therefore kept identical to their <cmath> equivalents: `sqrt`, `cbrt`, `pow`, `hypot`, `abs`, `exp`, `exp2`, `expm1`, `log`, `log2`, `log10`, `log1p`, `sin`, `cos`, `tan`, `asin`, `acos`, `atan`, `atan2`, `sinh`, `cosh`, `tanh`, `asinh`, `acosh`, `atanh`, `ceil`, `floor`, `round`, `trunc`, `copysign`.

2.1.1.4 Renamed or replaced

A small number of names are changed where the existing name is a C-specific artifact, a genuine source of confusion, or where the semantics need to be clarified to make user-defined customization unambiguous.

In some cases, clarifying the contract of an operation requires giving it a distinct name, especially where the existing name has acquired ambiguous or inconsistent semantics across types. `std::abs` for `std::complex` returning the Euclidean norm, and `std::norm` returning the squared modulus rather than any standard mathematical norm, is a concrete example of this: if left unchanged, it would be unclear how a user-defined multi-dimensional type should implement these operations. To disambiguate:

- `std::math::abs` returns the component-wise absolute value consistently across all types.
- `std::math::norm` is templated on a `std::math::norm_order` non-type template parameter specifying L_1, L_2, L_∞, or any other positive-integer norm order, defaulting to L_2.
- `std::math::norm_squared` returns the square of the L_2 norm as a distinct, unambiguous operation.

Other renamings address C-specific artifacts:

- `fabs` — dropped; subsumed by `abs`.
- `fmod` → `mod` — the `f` prefix is a C artifact; `mod` is consistent with cross-language convention.
- `modf` → `div_mod`, returning a structured binding-friendly result type with named `quotient` and `remainder` members; the near-anagram relationship between `modf` and `fmod` for unrelated operations is a well-known source of confusion.
- `fpclassify` — dropped; replaced by `is_nan`, `is_finite`, `is_infinite`, `is_normal` in snake_case, consistent with standard library naming conventions.
- `isnan`, `isinf`, `isfinite`, `isnormal` → `is_nan`, `is_infinite`, `is_finite`, `is_normal`.

2.2 Bridging through opt-in

In addition to the new namespace, this proposal also introduces a mechanism in the `std` namespace to enable custom implementation for types explicitly opted-in, with math functions. This is a compatibility layer with a strict contract to maintain the existing code behaviour: custom implementations are only called for explicitly opted-in types and when the alternative resolution is failing compilation (ill-formed).

We acknowledge the limit that existing behaviours relying on SFINAE based on mathematical functions not being resolved and applied on explicitly opted-in types might break. Opt-in should thus be exercised with caution, especially in contexts where SFINAE is conditioned on mathematical function availability.

The behaviour of this compatibility layer is deliberately different from the behaviour in `std::math`: the former prioritizes backward compatibility with explicit opt-in and user-specified implementations as fallbacks; the latter prioritizes user-defined customization, without opt-in, with `std` variant as the fall-back.

The compatibility layer contract is deliberately narrow, opening a path where failure was the alternative. This is to prevent edge-cases where the opt-in could result in behaviour change at a distance. It leaves open the possibility of widening this contract in the future based on real usage data.

2.3 Scope

Although the introduction of a new namespace provides an opportunity to fix some contract ambiguity, the scope of this proposal is limited to enabling the extension of existing mathematical operations to user-defined types.

This involves using a different name for some operations in the new namespace (e.g.: to match the naming patterns of the standard library), or in some cases to define distinct names for an existing operation (e.g.: `std::norm` split in the new namespace into `std::math::norm<>` and `std::math::norm_squared`).

However, this proposal does **not** attempt to introduce other new features such as separate new mechanisms for selecting local rounding behaviours or performance/precision trade-offs. It limits itself with not precluding such additions to the language at a later time.

This proposal is intended to introduce a minimal extensible library for common math functions. It is composed of three main parts:

— the API of the new library,
— the compatibility layer in `std`,
— a minimal set of implementations for the built-in types.

This minimality is especially motivated by the precautionary principle and the concern for standard library implementers bandwidth.

2.3.1 Future extensions

Although it would be possible to use the dispatch mechanism to define specialised functions that fallback on default implementation if a specialised implementation is not found for the given type (e.g. a fast square root function dispatching to dedicated fast algorithm if it is found for the type of the argument, or defaulting to `std::math::sqrt` otherwise), this is not leveraged as part of the current proposal. Users are free to make use of this possibility, but the provided implementations don't.

It does not provide algorithms or other mathematical functions leveraging other (potentially user-defined) mathematical operations it defines in their implementation. For instance, it would be possible to define the power operation in terms of exponentiation and logarithm by default. This possibility is left open for user-defined implementation but not provided as part of this proposal.

2.3.1.1 Rounding modes, precision/performance control

As stated before, adding such additional features to the language is out of the scope of this proposal. This said, if such additions were made simultaneously or in a future version of the language, the CPOs of `std::math` namespace could adapt to support it.

While it is difficult to know the form these new features would take, we can consider four options they could be implemented as:

— An additional template parameter.
— Square brackets.
— An additional parameter.
— New prefixed or suffixed functions.

In all of these cases, the CPOs in the `std::math` namespace could follow the same pattern in the same way. In the case of additional prefixed or suffixed functions, it could be decided whether the rationale for this choice in the namespace `std` holds for `std::math` and make a decision on this basis (considering if an additional template parameter would be preferred to maintain the design goal of minimizing the number of variants for the same operation, see Canonical function names).

2.4 Summary

This proposal doesn't require any new language feature. It would introduce a new `std::math` namespace and a compatibility layer inside of the `std` namespace, each of these two additions introducing approximately one new declaration per mathematical function, comparable in surface area to `<cmath>`.

3 Design Principles

3.1 Two spaces, different behaviours

The design space for this proposal is effectively split in two, between the forward-looking `std::math` layer, and the compatibility layer in `std`.

A solution in `std` only would have to compromise between extensibility and backward compatibility. Breaking the established contract there would be dangerous, and an opt-in-only solution leads to no good outcome: if adoption is low, generic code cannot rely on it for extensibility; if most types opt in, the mechanism ceases to be optional in any meaningful sense and only gets in the way.

Conversely, a new namespace alone would not help with code that calls `std` mathematical functions directly.

By combining both, the new code can benefit from the new approach in the new namespace with a new contract, while the compatibility layer in `std` remains available for targeted, case-by-case opt-ins.

The new `std::math` namespace splits from the mathematical functions inherited from C. It follows standard library naming patterns and leverages overloading and templates. It is intended as a native C++ library.

Compromising between C legacy and modern C++ patterns would hinder both objectives. The clear separation allows both sides to remain, each in its defined space.

3.2 New contract

The math functions in the `std::math` namespace have a new contract, independent from the `std` namespace contract.

User-customization implies a looser contract for these functions when they are dispatched to a user-defined implementation, where few aspects of such a contract can be enforced. Similarly to overloaded operators or other CPOs, the contract is owned by the implementation the call gets dispatched to.

The implementations for native types can define their own contracts, potentially forwarding the same contract as their `std` equivalent, but no such guarantee can be provided for user-defined implementations.

Since the new namespace adopts the standard library naming patterns, a small number of operations have a different name between the two namespaces (see Proposed naming). In such cases, they will read as distinct functions, and users will expect a different contract.

More importantly, with user-customization enabled, the function contract needs to be as clear as possible to avoid confusion. To that end, function names are chosen to reflect their purpose as precisely as possible.

Such an example of ambiguity arose with the `abs` and `norm` functions for the `std::complex` type, where `std::abs` was defined to be the L2 norm, and `std::norm` to be the square of the L2 norm (the squared modulus, which is not a mathematical norm). This would make it unclear how the `abs` and `norm` functions should be customized for user-defined multi-dimensional types.

3.3 CPO-like customization

Rather than a plain function template, `std::math` operations are proposed as CPO-like constructs. This design, established by the Ranges library (`std::ranges::begin`, `std::ranges::swap` etc.), offers these two advantages:

— The customization point cannot be found by ADL on user types, since it is an object rather than a function.
— The `operator()` can be constrained, making the customization point SFINAE-friendly and allowing it to be used correctly inside `requires` expressions and concepts.

3.3.1 Dispatch

The dispatch priority of these CPO-like constructs is explicitly:

1. To a member function (not subject to ADL).
2. To a free function found via ADL in the type's own namespace
3. To an equivalent `std` operation as the final fallback for all legacy types

The parameters are forwarded by forward reference.

User-defined implementations are not required to be `constexpr`, but the dispatch mechanism does not inhibit `constexpr` when it is supported.

3.3.2 Alternative to plain CPO

The term "CPO-like" is used instead of "CPO" directly to reflect the possibility of using upcoming improvement over the current CPO pattern. Specifically, if [P2806R3] (do expressions) and [P2826R3] (expression aliases) are accepted for C++29, the implementation of `std::math::sqrt` reduces significantly.

The do expression makes the ADL idiom available in expression contexts, and expression aliases provide expression-equivalence guarantees. This would allow the entire CPO to be expressed as a single declaration, without explicit dispatch concepts or a poison pill. The direction proposed here would compose naturally with these future language improvements.

While do-expressions alleviate the impossibility to use the `using` workaround in expression-only contexts, it doesn't address on its own the problems of discoverability, teachability, and separation of concern issues detailed in the motivation section of this document.

3.4 Single header

All functions are defined in a single header. The specific name of this header remains to be decided.

- `<math>` is easy to remember and consistent with other header names, but might conflict with `<math.h>`.
- `<cppmath>` or `<stdmath>` would avoid that confusion.

Specialisations for built-in types would also be defined in that same header. Specialisation for other types would be defined as part of the header associated with these types (e.g. `<complex>` for `std::complex`).

3.5 Canonical function names

Functions in the `std::math` namespace should define a single name per operation. Introducing name variants (prefix, suffix) should be avoided. Overloading and templating are preferred instead.

This helps keeping the surface area of the library as compact as possible and allows users to reach for the mathematical operation they need in a unified way.

The purpose of each mathematical operation should be made as clear as possible from its name and signature (as exercised by the native types) to make their user-customization as unambiguous as possible.

For the `abs` example, this would mean a single `abs` operation (no `labs`, `llabs`, `imaxabs`, `fabs`, `fabsf`, or `fabsl`) which is meant to return a value of the same type as its received parameter, where all negative components of the input value have had their sign switched (component-wise abs).

`norm` would be templated on a `norm_order` non-type template parameter defining which norm order is requested (L_1, L_2, L_∞, or any other positive integer order), and defaulted to L_2 when omitted. An additional `norm_squared` function could be considered, returning specifically the square of the L2 norm to offer a more performant alternative where the squared value is sufficient, without sacrificing the semantic of the operation.

See Proposed naming.

3.6 Customisation-level specifications

The complexity, performance profile, rounding-mode and precision of an operation is typically defined at the customisation-level, allowing for types specialised for fast or precise implementations, or implementing a specific rounding mode.

For built-in types, the contract from the `std` version of the equivalent operation is usually maintained for backward-compatibility reasons.

4 General Specifications for built-in arithmetic types

Implementations for the fundamental arithmetic types (`char`, `short`, `int`, `long`, `long long`, `float`, `double`, `long double`, and `bool` where relevant) are provided where conservatively appropriate.

Specializations for `std::complex<T>` are provided where appropriate, with semantics defined independently from their std counterparts. All other standard library and user-defined types participate through the ADL dispatch mechanism.

These Standard Library implementations abide by the following rules.

4.1 Conservative addition

`std::math` functions are not specialized for every fundamental arithmetic type through this proposal. Specifically, when the contract of a specialization is unclear and its presence not indispensable, it is deferred to a later revision.

For instance, `std::math::sqrt` return type when called on an integral type is unclear. It could be a `float`, a `double`, an `int`, or alternatively, this specialization of the operation could be templated to require an explicit return type as a template parameter. [P3605R1] separately explores this question, proposing a different name for reasons that are similar to the motivations for a separate namespace in this proposal. This proposal therefore chooses to leave `std::math::sqrt` undefined for integral types, keeping this design space open for future improvements that could align with or build on that work. This is both a precautionary principle and to avoid overwhelming implementers.

4.2 `constexpr`

All Standard-Library-defined specializations of functions in `std::math` are `constexpr`.

4.3 `[[nodiscard]]`

All functions in `std::math` with a return type are `[[nodiscard]]` unless specified otherwise.

4.4 Error Handling

Domain errors are considered contract violations and do not constitute runtime errors. As such they don't prevent the function from being `noexcept`.

`std::math` functions define their contract explicitly. They do not mandate `errno` update. Whether `errno` is set depends on the underlying implementation: standard library specializations inherited from `std` typically do, but it is not required.

Floating-point domain errors follow `IEEE754` semantics (NaN/inf). Integral-type domain errors are Undefined Behaviour.

Standard Library specialisations for fundamental arithmetic types and `std::complex` are `noexcept`.

5 Technical Specification

5.1 Functions

The contract of each function is defined at two levels. The general contract describes the mathematical intent of the operation. The specializations define the precise contract for the types provided by the standard library, typically deferring to the equivalent `std` operation. For user-defined types, the contract is owned by the customization.

5.1.1 Power Functions

5.1.1.1 `std::math::sqrt`

Signature	std Equivalent
`R sqrt(T x)`	`std::sqrt`

See implementation example.

Contract

Returns the square root of `x`. The return type may differ from the input type ($\sqrt{m^2} \to m$).

Specializations

```
/*floating-point-type*/ sqrt(/*floating-point-type*/)
```

Same contract as `std::sqrt(/*floating-point-type*/)`.

```
template<typename T>
std::complex<T> sqrt(const std::complex<T>&)
```

Same contract as `std::sqrt(const std::complex<T>&)` (complex square root in the range of the right half-plane).

Possible user-defined customizations

— For a matrix A: $\sqrt{A}$ via Schur decomposition (e.g. Björck–Hammarling algorithm).

5.1.1.2 `std::math::cbrt`

Signature	std Equivalent
`R cbrt(T x)`	`std::cbrt`

Contract

Returns the cube root of `x`. The return type may differ from the input type ($\sqrt[3]{m^3} \to m$).

Specializations

```
/*floating-point-type*/ cbrt(/*floating-point-type*/)
```

Same contract as `std::cbrt(/*floating-point-type*/)`.

5.1.1.3 `std::math::pow`

Signature	std Equivalent
`R pow(T base, U exp)`	`std::pow`

Contract

Returns `base` raised to the power `exp`.

Specializations

```
/*floating-point-type*/ pow(/*floating-point-type*/ base,
                           /*floating-point-type*/ exp)
```

Same contract as `std::pow(/*floating-point-type*/, /*floating-point-type*/)`.

```
template<class T1, class T2>
std::complex<std::common_type_t<T1, T2>>
    pow(const std::complex<T1>& x, const std::complex<T2>& y);

template<class T, class NonComplex>
std::complex<std::common_type_t<T, NonComplex>>
    pow(const std::complex<T>& x, const NonComplex& y);

template<class T, class NonComplex>
std::complex<std::common_type_t<T, NonComplex>>
    pow(const NonComplex& x, const std::complex<T>& y);
```

Same contracts as `std::pow` for the corresponding complex overloads.

Possible user-defined customizations

— For a matrix A: $A^p = \exp(p \cdot \log(A))$, noting that the matrix logarithm is not unique in general.

5.1.1.4 `std::math::hypot`

Signature	`std` Equivalent
`T hypot(T x, T y, Ts...)`	`std::hypot`[1]

[1] `std::hypot` is only equivalent for 2 to 3 arguments.

Contract

Returns $\sqrt{\sum x_i^2}$, computed so as to avoid intermediate overflow and underflow.

Specializations

```
template<class T, class... Ts>
requires (std::floating_point<T> && (std::same_as<T, Ts> && ...))
T hypot(T x, T y, Ts...);
```

Same contract as `std::hypot` for 2 and 3 arguments; generalised for more arguments.

5.1.2 Exponential

5.1.2.1 `std::math::exp`

Signature	`std` Equivalent
`T exp(T x)`	`std::exp`

Contract

Returns the natural exponential of `x`.

Specializations

```
/*floating-point-type*/ exp(/*floating-point-type*/)
```

Same contract as `std::exp(/*floating-point-type*/)`.

Possible user-defined customizations

— For a matrix A: $\sum_{k=0}^{\infty} \frac{A^k}{k!}$ (scaling-and-squaring with Padé approximation).
— For a quaternion $q = a + bi + cj + dk$: $e^a \left(\cos \|v\| + \hat{v} \sin \|v\|\right)$ where $v = (b, c, d)$.

5.1.2.2 `std::math::exp2`

Signature	`std` Equivalent
`T exp2(T x)`	`std::exp2`

Contract

Returns 2 raised to the power `x`.

Specializations

```
/*floating-point-type*/ exp2(/*floating-point-type*/)
```

Same contract as `std::exp2(/*floating-point-type*/)`.

5.1.2.3 `std::math::expm1`

Signature	`std` Equivalent
`T expm1(T x)`	`std::expm1`

Contract

Returns $e^x - 1$.

Specializations

```
/*floating-point-type*/ expm1(/*floating-point-type*/)
```

Same contract as `std::expm1(/*floating-point-type*/)`.

5.1.2.4 `std::math::log`

Signature	std Equivalent
`T log(T x)`	`std::log`

Contract

Returns the natural logarithm of `x`.

Specializations

```
/*floating-point-type*/ log(/*floating-point-type*/)
```

Same contract as `std::log(/*floating-point-type*/)`.

Possible user-defined customizations

— For a matrix A: the principal matrix logarithm via inverse scaling-and-squaring using Schur decomposition.

5.1.2.5 `std::math::log2`

Signature	std Equivalent
`T log2(T x)`	`std::log2`

Contract

Returns the base-2 logarithm of `x`.

Specializations

```
/*floating-point-type*/ log2(/*floating-point-type*/)
```

Same contract as `std::log2(/*floating-point-type*/)`.

5.1.2.6 `std::math::log10`

Signature	std Equivalent
`T log10(T x)`	`std::log10`

Contract

Returns the base-10 logarithm of `x`.

Specializations

```
/*floating-point-type*/ log10(/*floating-point-type*/)
```

Same contract as `std::log10(/*floating-point-type*/)`.

5.1.2.7 `std::math::log1p`

Signature	`std` Equivalent
`T log1p(T x)`	`std::log1p`

Contract

Returns $\log(1 + x)$.

Specializations

```
/*floating-point-type*/ log1p(/*floating-point-type*/)
```

Same contract as `std::log1p(/*floating-point-type*/)`.

5.1.3 Trigonometric

5.1.3.1 `std::math::sin`

Signature	`std` Equivalent
`T sin(T x)`	`std::sin`

Contract

Returns the sine of `x` (in radians).

Specializations

```
/*floating-point-type*/ sin(/*floating-point-type*/)
```

Same contract as `std::sin(/*floating-point-type*/)`.

5.1.3.2 `std::math::cos`

Signature	`std` Equivalent
`T cos(T x)`	`std::cos`

Contract

Returns the cosine of `x` (in radians).

Specializations

```
/*floating-point-type*/ cos(/*floating-point-type*/)
```

Same contract as `std::cos(/*floating-point-type*/)`.

5.1.3.3 `std::math::tan`

Signature	`std` Equivalent
`T tan(T x)`	`std::tan`

Contract

Returns the tangent of `x` (in radians).

Specializations

```
/*floating-point-type*/ tan(/*floating-point-type*/)
```

Same contract as `std::tan(/*floating-point-type*/)`.

5.1.3.4 `std::math::asin`

Signature	`std` Equivalent
`T asin(T x)`	`std::asin`

Contract

Returns the arc sine of `x`, in radians, in the range $[-\pi/2, \pi/2]$.

Specializations

```
/*floating-point-type*/ asin(/*floating-point-type*/)
```

Same contract as `std::asin(/*floating-point-type*/)`.

5.1.3.5 `std::math::acos`

Signature	`std` Equivalent
`T acos(T x)`	`std::acos`

Contract

Returns the arc cosine of `x`, in radians, in the range $[0, \pi]$.

Specializations

```
/*floating-point-type*/ acos(/*floating-point-type*/)
```

Same contract as `std::acos(/*floating-point-type*/)`.

5.1.3.6 `std::math::atan`

Signature	std Equivalent
`T atan(T x)`	`std::atan`

Contract

Returns the arc tangent of `x`, in radians, in the range $(-\pi/2, \pi/2)$.

Specializations

```
/*floating-point-type*/ atan(/*floating-point-type*/)
```

Same contract as `std::atan(/*floating-point-type*/)`.

5.1.3.7 `std::math::atan2`

Signature	std Equivalent
`T atan2(T y, T x)`	`std::atan2`

Contract

Returns the arc tangent of `y/x`, in radians, in the range $(-\pi, \pi]$, using the signs of both arguments to determine the quadrant of the result.

Specializations

```
/*floating-point-type*/ atan2(/*floating-point-type*/ y,
                             /*floating-point-type*/ x)
```

Same contract as `std::atan2(/*floating-point-type*/, /*floating-point-type*/)`.

5.1.4 Hyperbolic

5.1.4.1 `std::math::sinh`

Signature	std Equivalent
`T sinh(T x)`	`std::sinh`

Contract

Returns the hyperbolic sine of `x`.

Specializations

```
/*floating-point-type*/ sinh(/*floating-point-type*/)
```

Same contract as `std::sinh(/*floating-point-type*/)`.

5.1.4.2 `std::math::cosh`

Signature	`std` Equivalent
`T cosh(T x)`	`std::cosh`

Contract

Returns the hyperbolic cosine of `x`.

Specializations

```
/*floating-point-type*/ cosh(/*floating-point-type*/)
```

Same contract as `std::cosh(/*floating-point-type*/)`.

5.1.4.3 `std::math::tanh`

Signature	`std` Equivalent
`T tanh(T x)`	`std::tanh`

Contract

Returns the hyperbolic tangent of `x`.

Specializations

```
/*floating-point-type*/ tanh(/*floating-point-type*/)
```

Same contract as `std::tanh(/*floating-point-type*/)`.

5.1.4.4 `std::math::asinh`

Signature	`std` Equivalent
`T asinh(T x)`	`std::asinh`

Contract

Returns the inverse hyperbolic sine of `x`.

Specializations

```
/*floating-point-type*/ asinh(/*floating-point-type*/)
```

Same contract as `std::asinh(/*floating-point-type*/)`.

5.1.4.5 `std::math::acosh`

Signature	std Equivalent
`T acosh(T x)`	`std::acosh`

Contract

Returns the inverse hyperbolic cosine of `x`.

Specializations

```
/*floating-point-type*/ acosh(/*floating-point-type*/)
```

Same contract as `std::acosh(/*floating-point-type*/)`.

5.1.4.6 std::math::atanh

Signature	std Equivalent
`T atanh(T x)`	`std::atanh`

Contract

Returns the inverse hyperbolic tangent of `x`.

Specializations

```
/*floating-point-type*/ atanh(/*floating-point-type*/)
```

Same contract as `std::atanh(/*floating-point-type*/)`.

5.1.5 Rounding

5.1.5.1 std::math::ceil

Signature	std Equivalent
`T ceil(T x)`	`std::ceil`

Contract

Returns the smallest integer value not less than `x`.

Specializations

```
/*floating-point-type*/ ceil(/*floating-point-type*/)
```

Same contract as `std::ceil(/*floating-point-type*/)`.

5.1.5.2 std::math::floor

Signature	std Equivalent
`T floor(T x)`	`std::floor`

Contract

Returns the largest integer value not greater than `x`.

Specializations

```
/*floating-point-type*/ floor(/*floating-point-type*/)
```

Same contract as `std::floor(/*floating-point-type*/)`.

5.1.5.3 std::math::round

Signature	std Equivalent
`T round(T x)`	`std::round`

Contract

Returns `x` rounded to the nearest integer, rounding halfway cases away from zero.

Specializations

```
/*floating-point-type*/ round(/*floating-point-type*/)
```

Same contract as `std::round(/*floating-point-type*/)`.

5.1.5.4 std::math::trunc

Signature	std Equivalent
`T trunc(T x)`	`std::trunc`

Contract

Returns `x` rounded toward zero to the nearest integer.

Specializations

```
/*floating-point-type*/ trunc(/*floating-point-type*/)
```

Same contract as `std::trunc(/*floating-point-type*/)`.

5.1.6 Sign

5.1.6.1 std::math::abs

Signature	std Equivalent
`T abs(T x)`	`std::abs`

Contract

Returns the element-wise absolute value of `x`. For scalar types, negates `x` if negative. For multi-dimensional types, returns a value of the same type where each component has had its sign negated if negative.

Specializations

```
/*arithmetic-type*/ abs(/*arithmetic-type*/)
```

Same contract as `std::abs(/*arithmetic-type*/)`.

```
template<typename T>
std::complex<T> abs(const std::complex<T>&)
```

Returns `std::complex<T>{abs(re), abs(im)}`: element-wise, **not** the Euclidean norm which is obtained through `std::math::norm<2u>`. This differs from `std::abs(std::complex<T>)`.

Possible user-defined customizations

— For a vector: returns a vector with each component negated if negative.
— For a matrix: returns a matrix with each element negated if negative.

5.1.6.2 `std::math::copysign`

Signature	std Equivalent
`T copysign(T x, T y)`	`std::copysign`

Contract

Returns a value with the magnitude of `x` and the sign of `y`.

Specializations

```
/*floating-point-type*/ copysign(/*floating-point-type*/ x,
                                  /*floating-point-type*/ y)
```

Same contract as `std::copysign(/*floating-point-type*/, /*floating-point-type*/)`.

5.1.6.3 `std::math::mod`

Signature	std Equivalent
`T mod(T x, T y)`	`std::fmod`

Contract

Returns the floating-point remainder of `x/y`, with the same sign as `x`.

Specializations

```
/*floating-point-type*/ mod(/*floating-point-type*/ x,
                           /*floating-point-type*/ y)
```

Same contract as `std::fmod(/*floating-point-type*/, /*floating-point-type*/)`.

5.1.6.4 `std::math::div_mod`

Signature	`std` Equivalent
`div_mod_result<T> div_mod(T x, T y)`	—

`div_mod_result<T> std::modf div_mod(T x)` ——————— ———————

Contract

Returns the quotient truncated towards 0 and corresponding remainder of `x/y` with the same sign as `x`, or the integral and fractional parts of `x` if a single parameter is provided (equivalent to `y` defaulted to 1 for arithmetic types). Returns a structured-binding-friendly type with named members:

```
template<typename T>
struct div_mod_result {
    T quotient;
    T remainder;
};
```

Specializations

```
div_mod_result</*floating-point-type*/>
    div_mod(/*floating-point-type*/, /*floating-point-type*/ = 1)
```

Returns a `div_mod_result` object where `quotient` is `std::trunc(x/y)` and `remainder` satisfies `quotient` $\times$ `y` + `remainder` = `x`, with `remainder` having the same sign as `x`, consistent with `std::fmod(x, y)`. When called with a single argument, `y` defaults to `1`, giving the integral and fractional parts of `x`, consistent with `std::modf(x, ...)`.

5.1.7 Classification

5.1.7.1 `std::math::is_nan`

Signature	`std` Equivalent
`bool is_nan(T x)`	`std::isnan`

Contract

Returns `true` if `x` is NaN, `false` otherwise.

Specializations

```
bool is_nan(/*floating-point-type*/)
```

Same contract as `std::isnan(/*floating-point-type*/)`.

5.1.7.2 `std::math::is_finite`

Signature	`std` Equivalent
`bool is_finite(T x)`	`std::isfinite`

Contract

Returns `true` if `x` is finite (not infinite and not NaN), `false` otherwise.

Specializations

```
bool is_finite(/*floating-point-type*/)
```

Same contract as `std::isfinite(/*floating-point-type*/)`.

5.1.7.3 `std::math::is_infinite`

Signature	`std` Equivalent
`bool is_infinite(T x)`	`std::isinf`

Contract

Returns `true` if `x` is positive or negative infinity, `false` otherwise.

Specializations

```
bool is_infinite(/*floating-point-type*/)
```

Same contract as `std::isinf(/*floating-point-type*/)`.

5.1.7.4 `std::math::is_normal`

Signature	`std` Equivalent
`bool is_normal(T x)`	`std::isnormal`

Contract

Returns `true` if `x` is a normal floating-point value (not zero, subnormal, infinite, or NaN), `false` otherwise.

Specializations

```
bool is_normal(/*floating-point-type*/)
```

Same contract as `std::isnormal(/*floating-point-type*/)`.

5.1.8 Norm

5.1.8.1 `std::math::norm_order`

`norm_order` is a structural type representing the order N of an L_N norm: a positive integer, or a sentinel value representing infinity (for the L_∞ norm). It is intended for use as a non-type template parameter.

```
namespace std::math
{

struct norm_order
{
    unsigned int value;

    consteval norm_order(unsigned int n)
        pre(n != std::numeric_limits<unsigned int>::max())
        : value(n)
    {   }

private:
    consteval norm_order()
        : value(std::numeric_limits<unsigned int>::max())
    {   }

public:
    static constexpr norm_order infinity()
    {
        return {};
    }

    consteval bool is_infinity() const
    {
        return value == std::numeric_limits<unsigned int>::max();
    }

    friend auto operator<=>(norm_order, norm_order) = default;
};

} // namespace std::math
```

The public constructor is `consteval`, reflecting that `norm_order` is only meaningful as a compile-time value. The precondition reserves the maximum representable `unsigned int` value as a sentinel for infinity, expressed as a contract so that it is checked at compile time when violated.

The infinity sentinel is produced by a private default constructor, accessible only via the `infinity()` factory function. This ensures the sentinel value cannot be constructed by any other means — the precondition on the public constructor and the private constructor together guarantee that `is_infinity()` is true if and only if the value was obtained via `norm_order::infinity()`.

The `consteval` choice for the constructors is deliberate and conservative: it can be relaxed to `constexpr` in a future revision, but the reverse would be a breaking change.

```
std::math::norm<2u>(v)                          // L2 norm
std::math::norm<1u>(v)                          // L1 norm
std::math::norm<norm_order::infinity()>(v)      // L∞ norm
```

5.1.8.2 `std::math::norm`

Signature	std Equivalent
`auto norm<norm_order N = 2u>(T x)`	—

Contract

Returns the norm of `x` of the order specified by `N`. Defaults to the L_2 norm.

Specializations

```
/*floating-point-type*/ norm(/*floating-point-type*/)
```

Returns `abs(x)` — for scalar types, the norm is equivalent for any order `N`.

Possible user-defined customizations

— For a vector v: $\|v\|_1 = \sum |v_i|$, $\|v\|_2 = \sqrt{\sum v_i^2}$, $\|v\|_\infty = \max |v_i|$, generalising to $\|v\|_N = \left(\sum |v_i|^N\right)^{1/N}$ for any positive integer N.
— For a matrix A: column-sum norm, Frobenius norm, or spectral norm, corresponding respectively to $N = 1$, $N = 2$, and `norm_order::infinity()`.

A type author may specialize `norm` for one or a handful of specific values of `N`, or provide a single implementation generic over `N` where a convenient closed form exists (such as the $\|v\|_N$ formula above).

5.1.8.3 `std::math::norm_squared`

Signature	`std` Equivalent
`auto norm_squared(T x)`	`std::norm`[1]

[1] `std::norm` for `std::complex` returns the squared modulus, not a mathematical norm.

Contract

Returns the square of the L_2 norm of `x`. Equivalent to `norm<2u>(x) * norm<2u>(x)` but potentially more efficient.

Specializations

```
/*floating-point-type*/ norm_squared(/*floating-point-type*/)
```

Returns `x * x`.

```
template<typename T>
T norm_squared(const std::complex<T>&)
```

Same contract as `std::norm(const std::complex<T>&)` (returns $a^2 + b^2$ for $x = a + bi$).

5.2 Proposed Implementation

This section presents some code illustrations implementing `std::math::sqrt` as a representative example.

5.2.1 CPO

```
namespace std::math {
namespace __sqrt {

    // Poison pill: prevents unqualified ADL calls from accidentally
    // finding std::math::sqrt during concept checking
```

```
    void sqrt(auto) = delete;

    template<typename T>
    concept has_member_sqrt = requires(T&& x)
    {
        std::forward<T>(x).sqrt();
    };

    template<typename T>
    concept has_adl_sqrt = !has_member_sqrt<T> && requires(T&& x)
    {
        sqrt(std::forward<T>(x)); // ADL only; poison pill blocks std::math::sqrt
    };

    template<typename T>
    concept has_std_sqrt = !has_member_sqrt<T> && !has_adl_sqrt<T> && requires(T&& x)
    {
        std::sqrt(std::forward<T>(x));
    };

    struct __fn
    {
        template<typename T>
        requires has_member_sqrt<T>
        [[nodiscard]] constexpr auto operator()(T&& x) const
            noexcept(noexcept(std::forward<T>(x).sqrt()))
        {
            return std::forward<T>(x).sqrt();           // member customization
        }

        template<typename T>
        requires (!has_member_sqrt<T> && has_adl_sqrt<T>)
        [[nodiscard]] constexpr auto operator()(T&& x) const
            noexcept(noexcept(sqrt(std::forward<T>(x))))
        {
            return sqrt(std::forward<T>(x));            // ADL
        }

        template<typename T>
        requires (!has_member_sqrt<T> && !has_adl_sqrt<T> && has_std_sqrt<T>)
        [[nodiscard]] constexpr auto operator()(T&& x) const
            noexcept(noexcept(std::sqrt(std::forward<T>(x))))
        {
            return std::sqrt(std::forward<T>(x));       // std fallback
        }
    };

} // namespace __sqrt

inline namespace __cpo {
    inline constexpr __sqrt::__fn sqrt{};
}
} // namespace std::math
```

5.2.2 Compatibility layer

```cpp
namespace std {

// Opt-in trait for the compatibility forwarding layer.
// Users specialise this for their own types.
template<typename T>
inline constexpr bool is_math_extensible = false;

namespace __sqrt_compat {

    template<typename T>
    concept has_native_sqrt = requires(T&& x)
    {
        std::sqrt(std::forward<T>(x)); // qualified: only the existing std::sqrt overload
                                       // set, including conversion, excluding ADL
    };

} // namespace __sqrt_compat

// Forward to std::math for opted-in types, but only when no
// existing std::sqrt overload (including via conversion) applies.
template<typename T>
    requires is_math_extensible<std::remove_cvref_t<T>>
          && (!__sqrt_compat::has_native_sqrt<T>)
constexpr auto sqrt(T&& x)
    noexcept(noexcept(math::sqrt(std::forward<T>(x))))
    -> decltype(math::sqrt(std::forward<T>(x)))
{
    return math::sqrt(std::forward<T>(x));
}

} // namespace std
```

A proof of concept implementing `std::math::sqrt`, `std::math::abs`, `std::math::norm_order`, and the compatibility layer, verified under GCC, Clang, and MSVC, is available at [extmath-poc].

5.2.3 Alternative implementation with [P2806R3] (do expressions) and [P2826R2] (replacement functions)

This example is for illustration purposes, and doesn't necessarily represent the proposal syntax.

```cpp
namespace std::math
{
  static constexpr struct
  {
    using operator()(auto&& x) = (do { using std::sqrt; do_return sqrt(std::forward<decltype(x)>(x)); });
  } sqrt;
}
```

6 References

[boost-units-sqrt] Boost Developers. Boost.Units: sqrt Implementation.
https://github.com/boostorg/units/blob/develop/include/boost/units/cmath.hpp

[eigen-math] Eigen Developers. Eigen: MathFunctions.h Implementation.
https://gitlab.com/libeigen/eigen/-/blob/master/Eigen/src/Core/MathFunctions.h

[extmath-poc] Stéphane Gros-Lemesre. Proof of Concept Implementation for std::math.
https://godbolt.org/z/h7jsMjGEs

[mp-units] Mateusz Pusz. mp-units: Math Dispatch Implementation.
https://github.com/mpusz/mp-units/blob/b0e72810b983841b260d570b241c52586aa78999/src/core/include/mp-

units/framework/representation_concepts.h#L227-L262
[nholthaus-issue39] Nick Holthaus. nholthaus/units Issue #39: ADL and Math Functions.
https://github.com/nholthaus/units/issues/39
[nholthaus-units] Nick Holthaus. nholthaus/units Library.
https://github.com/nholthaus/units
[P2806R3] Barry Revzin, Bruno Cardoso Lopez, Zach Laine, Michael Park. 2025-01-12. do expressions.
https://wg21.link/p2806r3
[P2826R2] Gašper Ažman. 2024-03-18. Replacement functions.
https://wg21.link/p2826r2
[P2826R3] Gašper Ažman. 2026-05-12. Expression Aliases.
https://wg21.link/p2826r3
[P3045R8] Mateusz Pusz, Dominik Berner, Johel Ernesto Guerrero Peña, Charles Hogg, Nicolas Holthaus, Roth Michaels, Vincent Reverdy. 2026-05-12. Quantities and units library.
https://wg21.link/p3045r8
[P3605R1] Nikita Sakharin. 2026-03-12. isqrt: A function to calculate integer square root of the nonnegative integer.
https://wg21.link/p3605r1
[P3935R1] Jan Schultke. 2026-05-11. Rebasing <cmath> on C23.
https://wg21.link/p3935r1
[pusz-cpponsea2025] Mateusz Pusz. The 10 Essential Features for the Future of C++ Libraries. C++ on Sea 2025.
https://www.youtube.com/watch?v=TJg37Sh9j78&t=3158s