Document number: P0631R2
Date: 2018-02-06
Project: Programming Language C++
Audience: SG6 (Numerics), Library Evolution Working Group
Author: Lev Minkovsky
Reply-to: Lev Minkovsky lminkovsky@outlook.com; John McFarlane john@mcfarlane.name

Math Constants

1. Changelog

Changes from R1:

- Several typos fixed
- A better URL for Boost math constant
- Design goal 4) is replaced
- Radian, Catalan's, Apéry's and Glaisher's constants are no longer proposed
- Motivation behind Euler-Mascheroni and golden ratio constants added
- The inverse constants now have underscores in their names
- A link to the list of Wolfram constants removed as no longer relevant

Changes from R0:

- Added changelog, header and footer
- Several readability improvements
- Chapters are numbered
- The 4th and 6th chapters subdivided into subchapters
- Design goals stated
- A different set of constants proposed
- Naming conventions are now different
- A drop-in replacement for POSIX constants no longer proposed
- All definitions are in the new math_constants namespace
- Variable template types are inline
- Added new implementation requirements
- *float* and *double* typed constants proposed
- Boost constants described in the chapter 3.
- The 5th and 6th chapters reworked according to the abovementioned changes
- Links to the lists of Wolfram and Boosts constants added to the 7th chapter
- The types of constants should be directly or indirectly constexpr constructible from a fundamental floating-point type.
- Examples added of user-defined types suitable for instantiation of math constants

2. Introduction

C++ inherited from C a rich library of mathematical functions which continues to grow with every release. Amid all this abundance, there is a strange gap: none of the major mathematical constants is defined in the standard. This proposal is aimed to rectify this omission.

3. Motivation

http://reference.wolfram.com/language/ref/E.html

With the possible exception of Pi, E is the most important constant in mathematics. It appears in many sums, products, integrals, in equations involving the compounding of interest, in growth laws involving exponential growth or decay, and in formulas from a wide range of other mathematical and scientific fields.

Mathematical constants such as π and *e* frequently appear in mathematical algorithms. A software engineer can easily define them, but from their perspective, this is akin to making a reservation at a restaurant and being asked to bring their own salt. The C++ implementers appreciate this need and attempt to fulfil it with non-standard extensions.

The IEEE Standard 1003.1™-2008 a.k.a POSIX.1-2008 stipulates that on all systems supporting the X/Open System Interface Extension, "the <math.h> header shall define the following symbolic constants. The values shall have type *double* and shall be accurate to at least the precision of the *double* type."

`M_E`	- value of e
`M_LOG2E`	- value of $\log_2 e$
`M_LOG10E`	- value of $\log_{10} e$
`M_LN2`	- value of ln2
`M_LN10`	- value of ln10
`M_PI`	- value of π
`M_PI_2`	- value of $\frac{\pi}{2}$
`M_PI_4`	- value of $\frac{\pi}{4}$
`M_1_PI`	- value of $\frac{1}{\pi}$
`M_2_PI`	- value of $\frac{2}{\pi}$
`M_2_SQRTPI`	- value of $\frac{2}{\sqrt{\pi}}$
`M_SQRT2`	- value of $\sqrt{2}$
`M_SQRT1_2`	value of $\frac{\sqrt{2}}{2}$

POSIX.1-2008 explicitly states that these constants are outside of the ISO C standard and should be hidden behind an appropriate feature test macro. On some POSIX-compliant systems, this macro is defined as _USE_MATH_DEFINES, which led to a common assumption that defining this macro prior to the inclusion of math.h makes these constants accessible. In reality, this is true only in the following scenario:

1) The implementation defines these constants, and

2) It uses _USE_MATH_DEFINES as a feature test macro, and
3) This macro is defined prior to the first inclusion of math.h or any header file that directly or indirectly includes math.h.

These makes the availability of these constants extremely fragile when the code base is ported from one implementation to another or to a newer version of the same implementation. In fact, something as benign as including a new header file may cause them to disappear.

The OpenCL standard by the Kronos Group offers the same set of preprocessor macros in three variants: with a suffix _H, with a suffix _F and without a suffix, to be used in fp16, fp32 and fp64 calculations respectively. The first and the last sets are macro-protected. It also defines in the cl namespace the following variable templates:

e_v, log2e_v, log10e_v, ln2_v, ln10_v , pi_v, pi_2_v, pi_4_v, one_pi_v, two_pi_v, two_sqrtpi_v, sqrt2_v, sqrt1_2_v,

as well as their instantiations based on a variety of floating-point types and abovementioned macros. An OpenCL developer can therefore utilize a value of cl::pi_v<*float*>; they can also access cl::pi_v<*double*>, but only if the cl_khr_fp64 macro is defined.

The GNU C++ library offers an alternative approach. It includes an implementation-specific file ext\cmath that defines in the __gnu_cxx namespace the templated definitions of the following constants:

__pi,__pi_half,__pi_third,__pi_quarter,__root_pi_div_2,__one_div_pi,__two_div_pi,__two_div_root_pi ,__e,__one_div_e, __log2_e, __log10_e, __ln_2, __ln_3, __ln_10, __gamma_e, __phi, __root_2, __root_3,__root_5, __root_7, __one_div_root_2

The access to these constants is quite awkward. For example, to use a *double* value of π, a programmer would have to write __gnu_cxx::__math_constants::__pi<*double*>.

The Boost library has its own extensive set of constants, comprised of the following subsets:

- rational fractions (including $\frac{1}{2}$)
- functions of 2 and 10
- functions of π, e, ϕ (golden ratio) and Euler-Mascheroni γ constant
- trigonometric constants
- values of Riemann ζ (zeta) function
- statistical constants (various values of skewness and kurtosis)
- Catalan's, Glaisher's and Khinchin's constants

Components of their names are subdivided by an underscore, for example: one_div_root_pi. Boost provides their definitions for fundamental floating-point types in the following namespaces:

```
boost::math::constants::float_constants
boost::math::constants::double_constants
boost::math::constants::long_double_constants
```

For user-defined types, Boost constants are accessed through a function call, for example:

```
boost::math::constants::pi<MyFPType>();
```

All these efforts, although helpful, clearly indicate the need for standard C++ to provide a set of math constants that would be both easy to use and appropriately accurate.

4. Design considerations and proposed definitions

4.0. Design goals.

1) The user should be able to easily replace all POSIX constants with standard C++ constants.
2) The constants should be available for all fundamental floating-point types without type conversion and with maximum precision of their respective types.
3) It should be possible to easily create a set of values of basic trigonometric functions of common angles, also with their maximum precision.
4) The constants should provide tangible benefits for C++ users interested in numerical analysis.
5) They shouldn't cause name collisions. The code that compiled before them should compile with them.
6) It should be possible to instantiate them for user defined types.

4.1. The set of constants and their names

To achieve the design goals 1), 3) and 4) we need to provide the following three groups of constants:

Group 1:

e	- value of e
log2e	- value of $\log_2 e$
log10e	- value of $\log_{10} e$
ln2	- value of ln2
ln10	- value of ln10
pi	- value of π
inv_pi	- value of $\frac{1}{\pi}$
inv_sqrtpi	- value of $\frac{1}{\sqrt{\pi}}$
sqrt2	- value of $\sqrt{2}$

Group 2:

sqrt3	- value of $\sqrt{3}$
inv_sqrt3	- value of $\frac{1}{\sqrt{3}}$

Group 3:

egamma	- value of Euler-Mascheroni γ constant
phi	- value of golden ratio constant $\phi = (1+\sqrt{5})/2$

The group 1 constants will help us to achieve the design goals 1) and 4), while the group 2 will do the same for the goals 2) and 4). Although the members of group 3 are used less frequently, they are still helpful for the design goal 4). For example, the Euler-Mascheroni constant γ appears in the formula for a Bessel function of a second kind of order ν (source: www.mhtlab.uwaterloo.ca/courses/me755/web_chap4.pdf):

$$Y_\nu(x) = \frac{2}{\pi}J_\nu(x)\left(\ln\frac{x}{2}+\gamma\right) - \frac{1}{\pi}\sum_{k=0}^{\nu-1}\frac{(\nu-k-1)!}{k!}\left(\frac{x}{2}\right)^{2k-\nu} +$$
$$+\frac{1}{\pi}\sum_{k=0}^{\infty}\frac{(-1)^{k-1}\left[\left(1+\frac{1}{2}+\cdots+\frac{1}{k}\right)+\left(1+\frac{1}{2}+\cdots+\frac{1}{k+\nu}\right)\right]}{k!(k+\nu)!}\left(\frac{x}{2}\right)^{2k+\nu}$$

With the addition of the Euler-Mascheroni constant, it will become possible to numerically calculate the value of Y_v using only the constants presented in this proposal and the C++ 17 standard functions. As to the golden ratio *ϕ,* besides its traditional design applications, it is also used in the golden-section search, a method of finding a function extremum, see https://en.wikipedia.org/wiki/Golden-section_search.

It should be noted that all fundamental floating-point types are stored internally as a combination of a sign bit, a binary exponent and a binary normalized significand. If a ratio of two floating-point numbers of the same type is an exact power of 2 (within a certain limit), their significands will be identical. Therefore, in order to achieve the design goal 1), we don't have to provide replacements for both M_PI and M_PI_2 and M_PI_4. The user will be able to divide the M_PI replacement by 2 and by 4 and achieve the goals 2), 3) and 5).

4.2. Headers and namespaces

We can insert the definitions of math constants into <cmath> or <numeric>, or alternatively we can create for them a new standard library header <math_constants>. The related pros and cons are as follows:

1) <cmath>
   Pros:
   a) Math constants are typically used together with math functions, so the translation unit that depends on them will almost definitely #include either <math.h> or <cmath>
   b) The proposed names of *float*, *double* and *long double* constants follow the naming conversion of C functions, for example: float pif, double pi, long double pil.

   Cons:
   a) As per C++ standard, "The contents and meaning of the header <cmath> are the same as the C standard library header <math.h>, with the addition of a three-dimensional hypotenuse function (29.9.3) and the mathematical special functions described in 29.9.5."
      The addition of 3D hypotenuse and special functions expanded a set of standard C++ math functions without splitting it between different headers. It didn't bring in any new types of objects. We however introduce such type, constants, without the justification of continuity.
   b) More importantly, in order to let the user instantiate math constants for user-defined types, we have to define these constants as variable templates. C doesn't have templates, so it would be counter-intuitive to add them into <cmath>.

2) <numerics>

Pros:

a) This is a purely C++ header that we have full control over.
b) All its definitions are templates, so variable templates would be appropriate there.

Cons:

a) This header is described in the standard as “Generalized numeric operations”. Math constants do not fit into this description, so we will have to change it.
b) The *float*/*double*/*long double* constants would be out of place there, because as of now this header has nothing but templates.

3) new header <math_constants>

Pros:

a) None of the cons of <cmath> and <numerics>.

Cons:

a) The new header would increase the complexity of the language and the related documentation overhead.
b) The standard already has several short headers such as <initializer_list>, but they are either needed for C compatibility or target a significant fraction of C++ users. We also have domain-specific headers such as <ratio> that contain a good deal of functionality. The <math_constants> however would be a short domain-specific header that has nothing to do with C compatibility.

The best alternative appears to be the header <numeric>.

As stated in the design goal 6), it is essential to avoid possible name collisions with the existing customer code base. Consider, for example, the following code fragment:

```
#include <numeric>
using namespace std;
constexpr double e = 2.71828;
constexpr double esqr = e*e;
```

If we are to introduce an std::e constant, this fragment will no longer compile. The GNU C++ library resolves this problem by using a __math_constants namespace. The C++ standard already has an std::regex_constants namespace, presumably serving the same purpose. There appears to be a strong existing precedent for an introduction of a new namespace std::math_constants. Without it, we would have to make variable names long enough to minimize the chance of collisions. This would not help us to achieve the design goal 5).

4.3. Definitions

Math constant definitions should begin with the following set of templates:

```
template<typename T > inline constexpr T e_v;
template<typename T > inline constexpr T log2e_v;
template<typename T > inline constexpr T log10e_v;
template<typename T > inline constexpr T pi_v;
template<typename T > inline constexpr T inv_pi_v;
```

```
template<typename T > inline constexpr T inv_sqrtpi_v;
template<typename T > inline constexpr T ln2_v;
template<typename T > inline constexpr T ln10_v;
template<typename T > inline constexpr T sqrt2_v;
template<typename T > inline constexpr T sqrt3_v;
template<typename T > inline constexpr T inv_sqrt3_v;
template<typename T > inline constexpr T egamma_v;
template<typename T > inline constexpr T phi_v;
```

Alternatively, we can place template definitions into their own namespace:

```
namespace std {
	namespace math_constants {
		namespace templates {
			template<typename T > inline constexpr T e;
			template<typename T > inline constexpr T log2e;
			template<typename T > inline constexpr T log10e;
			template<typename T > inline constexpr T pi;
			template<typename T > inline constexpr T inv_pi;
			template<typename T > inline constexpr T inv_sqrtpi;
			template<typename T > inline constexpr T ln2;
			template<typename T > inline constexpr T ln10;
			template<typename T > inline constexpr T sqrt2;
			template<typename T > inline constexpr T sqrt3;
			template<typename T > inline constexpr T inv_sqrt3;
			template<typename T > inline constexpr T egamma;
			template<typename T > inline constexpr T phi;
		} //templates
	} //math_constants
}//std
```

The initialization part of these definitions will be implementation-specific. The implementation may at its discretion supply specializations of these variable templates for some or all fundamental floating-point types. The following requirements however need to be imposed:

1) Every implementation should guarantee that math constants can be instantiated for all fundamental floating-point types and for user-defined types constructible from a fundamental floating-point type through a sequence of constexpr constructors. For example, all types from the <complex> header would satisfy this requirement. Another example would be a possible implementation of quaternions:

```
template <typename T> class quaternion: public std::complex<T>
{
    T m_c;
    T m_d;
public:
    constexpr quaternion(const std::complex<T> value) : std::complex<T>(value),
    m_c(0), m_d(0) {}
    /*
    A lot of other functionality
    */
};
```

A yet another example, a high-precision floating-point type:

```
template <typename N, typename D, typename E, typename F> class floating_t
{
    N m_numerator;
    D m_denomenator;
    E m_exponent;

    static_assert(std::numeric_limits<N>::is_integer);
    static_assert(std::numeric_limits<D>::is_integer);
    static_assert(std::numeric_limits<E>::is_integer);
    static_assert(!std::numeric_limits<F>::is_integer);

    static_assert(std::numeric_limits<N>::is_signed);
    static_assert(!std::numeric_limits<D>::is_signed);
    static_assert(std::numeric_limits<E>::is_signed);
    static_assert(std::numeric_limits<N>::digits ==
        std::numeric_limits<D>::digits - 1);

    static constexpr unsigned exponent_length = CHAR_BIT * sizeof(F) -
        std::numeric_limits<F>::digits;
    static constexpr unsigned mantissa_length=std::numeric_limits<F>::digits-1;

    static_assert(CHAR_BIT * sizeof(D) > mantissa_length);
    static_assert(std::numeric_limits<E>::digits >= exponent_length);
    static_assert(std::numeric_limits<N>::digits >
        std::numeric_limits<F>::digits);

public:

    constexpr floating_t(F value) :m_numerator(0),
        m_denomenator(1),m_exponent(std::numeric_limits<F>::max_exponent-1)
    {

        m_denomenator <<= mantissa_length;

        bool isNegative = false;
        if (value < 0)
        {
            isNegative = true;
            value = -value;
        }

        if (value < 1)
            do
                m_exponent--;
            while ((value = 2 * value) < 1);
        else if (value > 2)
            do
                m_exponent++;
            while ((value = value / 2) > 2);

        m_numerator = static_cast<N>(value*static_cast<F>(m_denomenator));

        if (isNegative)
            m_numerator = - m_numerator;
    }

    constexpr bool validate(F value) const
```

```
            {
                double val = static_cast<F>(m_numerator)/
                    static_cast<F>(m_denomenator);
                for (E i=std::numeric_limits<F>::max_exponent;i <= m_exponent;   i++)
                    val *= 2;
                return value == val;
            }
            /*
            Many other lines of code
            */
        };
```

2) Every implementation needs to ensure that the instantiations of math constants for fundamental floating-point types are the most accurate approximations of underlying real numbers for these types (design goal 2)). This entails that if two implementations provide fundamental floating-point types with identical lengths of significands, the constants instantiated for these types will be equal. For example, all IEEE-754 compliant implementations will have the value of pi<*double*> equal to 0x1.921fb54442d18p+1. All numerical libraries having the same internal precision will therefore have identical values of their respective math constants.

After the templated constants, the following definitions should be made:

```
inline constexpr float ef = e_v<float>;
inline constexpr float log2ef = log2e_v<float>;
inline constexpr float log10ef = log10e_v<float>;
inline constexpr float pif = pi_v<float>;
inline constexpr float inv_pif = inv_pi_v<float>;
inline constexpr float inv_sqrtpif = inv_sqrtpi_v<float>;
inline constexpr float ln2f = ln2_v<float>;
inline constexpr float ln10f = ln10_v<float>;
inline constexpr float sqrt2f = sqrt2_v<float>;
inline constexpr float sqrt3f = sqrt3_v<float>;
inline constexpr float inv_sqrt3f = inv_sqrt3_v<float>;
inline constexpr float egammaf = egamma_v<float>;
inline constexpr float phif = phi_v<float>;

inline constexpr double e = e_v<double>;
inline constexpr double log2e = log2e_v<double>;
inline constexpr double log10e = log10e_v<double>;
inline constexpr double pi = pi_v<double>;
inline constexpr double inv_pi = inv_pi_v<double>;
inline constexpr double inv_sqrtpi = inv_sqrtpi_v<double>;
inline constexpr double ln2 = ln2_v<double>;
inline constexpr double ln10 = ln10_v<double>;
inline constexpr double sqrt2 = sqrt2_v<double>;
inline constexpr double sqrt3 = sqrt3_v<double>;
inline constexpr double inv_sqrt3 = inv_sqrt3_v<double>;
inline constexpr double egamma = egamma_v<double>;
inline constexpr double phi = phi_v<double>;

inline constexpr long double el = e_v<long double>;
inline constexpr long double log2el = log2e_v<long double>;
inline constexpr long double log10el = log10e_v<long double>;
inline constexpr long double pil = pi_v<long double>;
inline constexpr long double inv_pil = inv_pi_v<long double>;
inline constexpr long double inv_sqrtpil = inv_sqrtpi_v<long double>;
inline constexpr long double ln2l = ln2_v<long double>;
inline constexpr long double ln10l = ln10_v<long double>;
inline constexpr long double sqrt2l = sqrt2_v<long double>;
inline constexpr long double sqrt3l = sqrt3_v<long double>;
inline constexpr long double inv_sqrt3l = inv_sqrt3_v<long double>;
inline constexpr long double egammal = egamma_v<long double>;
inline constexpr long double phil = phi_v<long double>;
```

```
The way these variable and variable template definitions are injected into
std::math_constants will be implementation-specific.
```

4.4. Access patterns

Because the standard won't provide a drop-in replacement for POSIX/OpenCL/GNU constants, it will be up to the user how, or even whether, to transition to standardized constants. Some motivated users may do this via a global search-and-replace. It is likely however that many C++ projects will have the standard constants introduced alongside with the extant POSIX or user-defined constants. This may cause readability problems as well as subtle computational issues. For example, let's consider the following code fragment:

```
#define _USE_MATH_DEFINES
#include "math.h"

template<typename T> constexpr T pi =3.14159265358979323846L;

constexpr long double MY_OLD_PI = M_PI; //has been here for 10+ years
constexpr long double MY_NEW_PI = pi<long double>;

static_assert(MY_OLD_PI == MY_NEW_PI, "OMG!");
```

It compiles on Windows, where *long double* is 64-bit, but fails on Linux, where it is 128-bit. The users that need to support 128-bit *long double* will have to carefully assess the risk of having slightly different values of math constants in the same project.

If an existing codebase already has user-defined math constants, their definitions can easily be updated with standard constants, for example:

```
const double PI = std::math_constants::pi;
```

In a more "greenfield" situation, where math constants are just being introduced, they can be imported into a global scope by the `using` directive, for example:

using `std::math_constants::pi;`

5. A "Hello world" program for math constants

```
#include <numeric>

using std::math_constants::pi;
using std::math_constants::pi_v;

template<typename T> constexpr T circle_area(T r) { return pi_v<T> * r * r; }

int main()
{
	static_assert(!!pi);
	static_assert(!!circle_area(1.0));
	return 0;
}
```

6. Proposed changes in the standard

The clause 29.1 General

The subclause 29.1.2 should be updated as follows:

2 The following subclauses describe components for complex number types, random number generation, numeric (*n*-at-a-time) arrays, generalized numeric algorithms, mathematic constants and mathematical functions for floating-point types, as summarized in Table 93.

In the table 93, the subclause [numeric.ops] should be updated as follows:

[numeric.ops] Generalized numeric operations and mathematical constants <numeric>

The clause 29.8 Generalized numeric operations

The clause title should be updated as follows:

29.8 Generalized numeric operations and mathematical constants

In the subclause 29.8.1, after

```
// [numeric.ops.lcm], least common multiple
template <class M, class N>
	constexpr common_type_t<M, N> lcm(M m, N n);
```

the following should be inserted:

```
// [math.constants], mathematical constants
namespace math_constants {
		template<typename T > inline constexpr T e_v          see below
		template<typename T > inline constexpr T log2e_v      see below
		template<typename T > inline constexpr T log10e_v     see below
		template<typename T > inline constexpr T pi_v         see below
		template<typename T > inline constexpr T inv_pi_v     see below
		template<typename T > inline constexpr T inv_sqrtpi_v see below
		template<typename T > inline constexpr T ln2_v        see below
		template<typename T > inline constexpr T ln10_v       see below
		template<typename T > inline constexpr T sqrt2_v      see below
		template<typename T > inline constexpr T sqrt3_v      see below
		template<typename T > inline constexpr T inv_sqrt3_v  see below
```

```
              template<typename T > inline constexpr T egamma_v      see below
              template<typename T > inline constexpr T phi_v         see below

              inline constexpr float ef = e_v<float>;
              inline constexpr float log2ef = log2e_v<float>;
              inline constexpr float log10ef = log10e_v<float>;
              inline constexpr float pif = pi_v<float>;
              inline constexpr float inv_pif = inv_pi_v<float>;
              inline constexpr float inv_sqrtpif = inv_sqrtpi_v<float>;
              inline constexpr float ln2f = ln2_v<float>;
              inline constexpr float ln10f = ln10_v<float>;
              inline constexpr float sqrt2f = sqrt2_v<float>;
              inline constexpr float sqrt3f = sqrt3_v<float>;
              inline constexpr float inv_sqrt3f = inv_sqrt3_v<float>;
              inline constexpr float egammaf = egamma_v<float>;
              inline constexpr float phif = phi_v<float>;

              inline constexpr double e = e_v<double>;
              inline constexpr double log2e = log2e_v<double>;
              inline constexpr double log10e = log10e_v<double>;
              inline constexpr double pi = pi_v<double>;
              inline constexpr double inv_pi = inv_pi_v<double>;
              inline constexpr double inv_sqrtpi = inv_sqrtpi_v<double>;
              inline constexpr double ln2 = ln2_v<double>;
              inline constexpr double ln10 = ln10_v<double>;
              inline constexpr double sqrt2 = sqrt2_v<double>;
              inline constexpr double sqrt3 = sqrt3_v<double>;
              inline constexpr double inv_sqrt3 = inv_sqrt3_v<double>;
              inline constexpr double egamma = egamma_v<double>;
              inline constexpr double phi = phi_v<double>;

              inline constexpr long double el = e_v<long double>;
              inline constexpr long double log2el = log2e_v<long double>;
              inline constexpr long double log10el = log10e_v<long double>;
              inline constexpr long double pil = pi_v<long double>;
              inline constexpr long double inv_pil = inv_pi_v<long double>;
              inline constexpr long double inv_sqrtpil = inv_sqrtpi_v<long double>;
              inline constexpr long double ln2l = ln2_v<long double>;
              inline constexpr long double ln10l = ln10_v<long double>;
              inline constexpr long double sqrt2l = sqrt2_v<long double>;
              inline constexpr long double sqrt3l = sqrt3_v<long double>;
              inline constexpr long double inv_sqrt3l = inv_sqrt3_v<long double>;
              inline constexpr long double egammal = egamma_v<long double>;
              inline constexpr long double phil = phi_v<long double>;
}
```

After the subclause 29.8.14 [numeric.ops.lcm], a new subclause should be inserted:

29.8.15 Mathematical constants [math.constants]

```
namespace math_constants {
              template<typename T > inline constexpr T e_v           see below
              template<typename T > inline constexpr T log2e_v       see below
              template<typename T > inline constexpr T log10e_v      see below
              template<typename T > inline constexpr T pi_v          see below
              template<typename T > inline constexpr T inv_pi_v      see below
              template<typename T > inline constexpr T inv_sqrtpi_v  see below
```

```
template<typename T > inline constexpr T ln2_v        see below
template<typename T > inline constexpr T ln10_v       see below
template<typename T > inline constexpr T sqrt2_v      see below
template<typename T > inline constexpr T sqrt3_v      see below
template<typename T > inline constexpr T inv_sqrt3_v  see below
template<typename T > inline constexpr T egamma_v     see below
template<typename T > inline constexpr T phi_v        see below

inline constexpr float ef = e_v<float>;
inline constexpr float log2ef = log2e_v<float>;
inline constexpr float log10ef = log10e_v<float>;
inline constexpr float pif = pi_v<float>;
inline constexpr float inv_pif = inv_pi_v<float>;
inline constexpr float inv_sqrtpif = inv_sqrtpi_v<float>;
inline constexpr float ln2f = ln2_v<float>;
inline constexpr float ln10f = ln10_v<float>;
inline constexpr float sqrt2f = sqrt2_v<float>;
inline constexpr float sqrt3f = sqrt3_v<float>;
inline constexpr float inv_sqrt3f = inv_sqrt3_v<float>;
inline constexpr float egammaf = egamma_v<float>;
inline constexpr float phif = phi_v<float>;

inline constexpr double e = e_v<double>;
inline constexpr double log2e = log2e_v<double>;
inline constexpr double log10e = log10e_v<double>;
inline constexpr double pi = pi_v<double>;
inline constexpr double inv_pi = inv_pi_v<double>;
inline constexpr double inv_sqrtpi = inv_sqrtpi_v<double>;
inline constexpr double ln2 = ln2_v<double>;
inline constexpr double ln10 = ln10_v<double>;
inline constexpr double sqrt2 = sqrt2_v<double>;
inline constexpr double sqrt3 = sqrt3_v<double>;
inline constexpr double inv_sqrt3 = inv_sqrt3_v<double>;
inline constexpr double egamma = egamma_v<double>;
inline constexpr double phi = phi_v<double>;

inline constexpr long double el = e_v<long double>;
inline constexpr long double log2el = log2e_v<long double>;
inline constexpr long double log10el = log10e_v<long double>;
inline constexpr long double pil = pi_v<long double>;
inline constexpr long double inv_pil = inv_pi_v<long double>;
inline constexpr long double inv_sqrtpil = inv_sqrtpi_v<long double>;
inline constexpr long double ln2l = ln2_v<long double>;
inline constexpr long double ln10l = ln10_v<long double>;
inline constexpr long double sqrt2l = sqrt2_v<long double>;
inline constexpr long double sqrt3l = sqrt3_v<long double>;
inline constexpr long double inv_sqrt3l = inv_sqrt3_v<long double>;
inline constexpr long double egammal = egamma_v<long double>;
inline constexpr long double phil = phi_v<long double>;
```

1 *Requires:* T shall either be a fundamental floating-point type or be constructable from such type through a series of constexpr constructors.

2 *Remarks:* These variable templates should be initialized with implementation-defined values of e, $\log_2 e$, $\log_{10} e$, π, $\frac{1}{\pi}$, $\frac{1}{\sqrt{\pi}}$, ln2, ln10, $\sqrt{2}$, $\sqrt{3}$, $\frac{1}{\sqrt{3}}$, Euler-Mascheroni γ constant and golden ratio ϕ constant ($\frac{1+\sqrt{5}}{2}$), respectively. The implementation may provide their specializations for some or all

fundamental floating-point types (see **3.9.1)**. For each fundamental floating-point type, an instantiation of every variable template should be equal to the closest approximation of the underlying real number among the type's set of values.

7. References

The POSIX version of math.h is described at
http://pubs.opengroup.org/onlinepubs/9699919799/basedefs/math.h.html.

The OpenCL mathematical constants are defined in a file opencl_math_constants, see
https://raw.githubusercontent.com/KhronosGroup/libclcxx/master/include/opencl_math_constants.

The GNU math extensions: https://gcc.gnu.org/onlinedocs/gcc-6.1.0/libstdc++/api/a01120_source.html

A list of Boost math constants is at
http://www.boost.org/doc/libs/release/libs/math/doc/html/math_toolkit/constants.html

8. Acknowledgments

The author would like to thank Edward Smith-Rowland for his review of the draft proposal, Vishal Oza, Daniel Krügler and Matthew Woehlke for their participation in the related thread at the std-proposals user group, Walter E. Brown and John McFarlane for volunteering to present the proposal and helpful comments, and participants of discussions at LEWG and SG6 mailing lists for their valuable feedback.